"SMART NEIGHBORHOOD" INTELLIGENT PROPERTY SERVICE MANAGEMENT SYSTEM

WANPUDONG, Prof. Dr. Mohd Juzaiddin Ab Aziz

FACULTY SCIENCE AND TECHNOLOGY
UNIVERSITI KEBANGSAAN MALAYSIA 43600
Bangi, Selangor

Abstract

Fungsi utama "Smart Neighborhood" adalah untuk menyediakan platform pengurusan harta yang cekap bagi pemilik Blok18 dan kakitangan berkaitan Zhonghai Property . Melalui "Smart Neighborhood", pemilik boleh membuat tempahan kemudahan, menghantar maklum balas penyelenggaraan, melihat pengumuman dan operasi lain dalam talian, manakala kakitangan COSL boleh mengurus maklum balas pemilik, mengemas kini status kemudahan, dan menerbitkan pengumuman komuniti.

Dalam fungsi "Tempahan Kemudahan", pemilik boleh melihat status semasa kemudahan awam (kosong, sedang digunakan, dalam penyelenggaraan) dan memilih tempoh masa yang tersedia untuk membuat tempahan. Pada masa yang sama, sistem akan mengelakkan berlakunya konflik masa tempahan. Jika pemilik perlu menghantar permohonan penyelenggaraan kemudahan atau cadangan lain, beliau boleh menghantar maklumat terperinci melalui modul "Maklum Balas Kejadian", dan kakitangan harta akan memprosesnya serta mengemas kini status berdasarkan kandungan yang dihantar. Dalam modul pengumuman harta, pemilik boleh menyemak notis komuniti terkini atau peringatan keselamatan pada bila-bila masa untuk memastikan ketepatan dan ketelusan maklumat komuniti.

Untuk pengurus harta, sistem menyediakan fungsi pengurusan pangkalan data yang boleh mengurus semua maklumat pemilik, rekod maklum balas dan status kemudahan. Kakitangan boleh mengeluarkan notis penting melalui sistem, melihat hasil tinjauan kepuasan pemilik terhadap perkhidmatan dan melihat keperluan perkhidmatan berbeza pemilik yang diperoleh melalui analisis data oleh "Kejiranan Pintar" untuk memperuntukkan sumber harta dengan lebih rasional. Selain itu, sistem ini juga menyokong kakitangan hartanah untuk melihat dan memproses maklum balas insiden pemilik untuk mengoptimumkan kualiti perkhidmatan masyarakat.

Projek "Smart Neighborhood" terutamanya menyelesaikan masalah kecekapan rendah dan kehilangan data mudah yang disebabkan oleh rekod kertas tradisional dan mod pengurusan manual. Tingkatkan pengurusan hartanah melalui cara digital, tingkatkan kecekapan

komunikasi antara hartanah dan pemilik, serta bantu pemilik lebih memahami penggunaan sumber komuniti. Matlamat sistem termasuk mengoptimumkan proses pengurusan maklumat hartanah, meningkatkan penyertaan pemilik dalam perkhidmatan komuniti, dan mengukuhkan hubungan dan kerjasama antara hartanah dan pemilik.

Secara keseluruhan, "Kejiranan Pintar" bukan sahaja membolehkan pengurus hartanah menyelesaikan tugas harian dengan lebih cekap, tetapi juga menyediakan pemilik perkhidmatan komuniti yang lebih mudah dan telus, mewujudkan model pengurusan harta tanah yang lebih pintar.

Abstract

The main function of "Smart Neighborhood" is to provide an efficient property management platform for Block18 owners and relevant staff of China Shipping Property. Through "Smart Neighborhood", owners can make facility reservations, submit maintenance feedback, view announcements and other operations online, while COSL staff can manage owner feedback, update facility status, and publish community announcements.

In the "Facility Reservation" function, owners can view the current status of public facilities (idle, in use, under maintenance) and select an available time period to make a reservation. At the same time, the system will prevent reservation time conflicts from occurring. If the owner needs to submit a facility maintenance application or other suggestions, he can submit detailed information through the "Incident Feedback" module, and the property staff will process it and update the status based on the submitted content. In the property announcement module, owners can check the latest community notices or safety reminders at any time to ensure the immediacy and transparency of community information.

For property managers, the system provides database management functions that can manage all owner information, feedback records, and facility status. Staff can issue important notices through the system, view the results of owner satisfaction surveys on services, and view owners' different service needs obtained through data analysis by "Smart Neighborhood" in order to allocate property resources more rationally. In addition, the system also supports property personnel to view and process owners' incident feedback to optimize the quality of community services.

The "Smart Neighborhood" project mainly solves the problems of low efficiency and easy data loss caused by traditional paper records and manual management mode. Improve property management through digital means, improve communication efficiency between properties and owners, and help owners better understand the use of community resources. The goals of the system include optimizing property information management processes, increasing owners' participation in community services, and strengthening connections and collaboration between properties and owners.

Overall, "Smart Neighborhood" not only enables property managers to complete daily tasks more efficiently, but also provides owners with more convenient and transparent community services, creating a more intelligent property management model.

1.0 INTRODUCTION

In recent days, the property management company of Zhonghai Apartment received a warning notice from the government because the residents of Building 18 jointly complained to the government about the quality of property services. In order to solve the problem and improve the service quality of the property management department of the company, the property management company decided to develop an application called "Smart Neighborhood". The purpose of "Smart Neighborhood" is mainly to help the staff of ZhongHai Property Management better serve the owners. "Smart Neighborhood" will clearly understand owners'

needs for property reactants through data collection and analysis, and enhance the responsiveness of property management and user satisfaction. Through "Smart Neighborhood", property companies hope to establish good interactions with owners, further improve service quality, and thereby enhance the owners' living experience.1:Service type frequency analysis.

Residents of Building 18 of Zhonghai Apartment expressed dissatisfaction with the service speed and quality of the property, as well as the owner's safety protection measures and the construction of public facilities, indicating that Zhonghai Property Management has problems such as low efficiency, slow response, and poor service quality. This affects the

living experience of residents and is the reason why owners complain directly to the government. China Overseas Property's current service process lacks intelligent data analysis and timely useful feedback and cannot accurately identify owner needs and service priorities. In order to solve this problem, the property company decided to develop a "Smart Neighborhood".

“Smart Neighborhood” mainly aims to achieve the following goals: Improve service response efficiency: Through the collection and analysis of service data, we can understand which services owners have higher demands for and rationally allocate human resources. Adopt the opinions of owners: Establish a feedback mechanism for owners, promptly grasp the evaluation and opinions of property services, identify deficiencies in the services, and adopt targeted improvement measures to meet the expectations of owners. Make rectifications based on the areas where owners are dissatisfied with public facilities.

The development of "smart blocks" should improve the management level of Zhonghai Property through intelligent means, achieve the safety protection measures of modern apartments, solve the problems raised by current owners, and achieve the goal of making the owners of Building 18 satisfied with the property services to avoid being punished by the government.

2.0 LITERATURE REVIEW

The progress of social productivity and science and technology has improved people's quality of life, and naturally the demand for life will be higher, and the home is a place that reflects the sense of life needs. People in the home naturally have their own responsibility for the quality of life, but as the owner of a community, the quality of public services provided by the property is responsible for the property company. However, the traditional property service

management mode has gradually exposed many problems (low service efficiency, slow response speed, asymmetric information, etc.), which cannot completely meet the needs of owners for safety and convenient life. In this case, the concept of smart apartment has emerged, meaning that through modern intelligent information technology and Internet of Things technology to optimize the apartment service management process, improve the quality of life of owners. However, the existing intelligent community management system still has shortcomings in security management and public facility maintenance, which can not fully meet the service needs of owners and the requirements of real-time and intelligent management of modern apartments. Therefore, a more intelligent and integrated solution is needed to effectively solve these problems.

In the construction of intelligent apartment, intelligent service property management system is a key component, responsible for the daily security management of the apartment, public facilities maintenance and human, economic and other resources allocation tasks. Intelligent property management systems usually use technologies such as the Internet of Things, artificial intelligence, and big data analysis to provide decision support and automation services for property companies. For example, by monitoring the usage and status of apartment facilities through sensors, the system can report maintenance needs and optimize maintenance plans in real time, thereby improving resource utilization and reducing unnecessary labor input (Chen, 2018).

I cite several examples from the collected literature. Residential property management systems include smart home management, security management, parking lot management, integrated network management, intelligent business interconnection, etc. The system has the characteristics of advanced nature, reliability, scalability, security and compatibility (Wang, 2005). Traditional community management not only can not meet the needs of residents but also consumes a lot of manpower and material resources, and its management is not ideal. Although the new intelligent management system of the community has a relatively mature management mode, it needs to be further improved (Zhao, 2004). "Smart Neighborhood" is a system that integrates the service requirements of owners, integrated security management system, public facility planning, and three functions, mainly to provide owners with safe and convenient community life experience.

Through the overall background analysis, this project will develop an intelligent property service management system that integrates the service demand level of owners, comprehensive security management system and public facility planning, which will solve the pain points in traditional community service management, so as to promote the intelligent and modern development of the community.

Current most apartment management still has many aspects that can be improved in terms of service efficiency for owners, timely information exchange with owners, and safety protection measures for the entire apartment and owners. Traditional property management mainly relies on property staff, and there are problems such as slow response speed, information asymmetry, and high labor costs. In addition, although some modern apartments already have intelligent management systems, most of these systems only have a single function (such as security management or parking management) (Wang et al., 2022), lack a comprehensive system, there is still a great deal of room for improvement. For example, although some intelligent systems can achieve partial automation and data analysis, there are still shortcomings in security protection, timely understanding of the needs of owners, reasonable resource allocation and details of public facility maintenance, and it is difficult to meet the expectations of owners for

quality of life. At the same time, there is often a lack of interoperability between different intelligent management systems, resulting in inefficient use of resources.

Traditional property management methods often make it difficult to quickly respond to owners' service needs. Owners lack modernization and intelligence, which makes it difficult to quickly meet owners' needs. In addition, many smart apartment service management systems currently introduced in apartments have the same problem, the phenomenon of data islands (Wang et al., 2022), and it is difficult to realize information exchange between different systems, thus affecting the reasonable allocation and comprehensive management of resources.

In the existing research and practice, some intelligent community management systems have a single function, which leads to the emergence of data islands. They mainly focus on the development of certain functions (such as smart home, security management, etc.), and lack a comprehensive system platform. Especially in the deep integration of safety management and public facility maintenance, there is still a big gap. Most current intelligent systems leave much to be desired in terms of data integration and speed of response to owners' needs. In addition, with the development of technology and the improvement of productivity, many owners' requirements for quality of life and service have also increased. How to timely feedback the needs of the owners to the property company is also a problem that needs to be solved. Therefore, a timely comprehensive intelligent management system that can reflect the owners' demand for service, security risks, the use of public facilities and the facilities in need of maintenance to the property in a timely manner is a very potential project.Suggest a solution.

In view of the existing problems in different systems, this project proposes to build a "Smart Neighborhood" property service management system to integrate basic service

requirements, security management, repair and maintenance functions of public facilities, and provide support in data integration and information flow. Through simplified technical solutions, such as basic database management and information query, the system can achieve rapid information transfer between the owner and the property to improve the efficiency of property management. In addition, Smart Neighborhood will be designed for future scalability so that more intelligent features can be added as the technology matures.

3.0 METHODOLOGY

This study covers requirements analysis, conceptual model design, application development, usability testing, and results. It also explains the research process carried out.

3.1 DEFINITION OF USER NEEDS

3.1.1 DEGREE OF SERVICE DEMAND

Analyze the frequency of use of different property services collected by the company to determine which services have higher demand from owners and which services have lower demand. By analyzing the data collected by "Service Type Frequency Analysis", we can understand how satisfied Building 18's owners are with various services and identify areas that require maintenance and improvement. (For example: cleaning corridors, timely disposal of public garbage, etc.) (Yilmaz, 2010).

3.1.2 Integrated safety management system

a.Community announcements and safety reminders

ZhangHai Management staff can post announcements in the background (property management staff are divided into three levels: manager, admin, and normal. Only employees with manager and admin privileges can post announcements). Owners can receive and view announcements on the client side through the database refresh mechanism.

b.Incident reporting and recording

Design a feedback form in "Smart Neighborhood", and the owner fills in the incident description and submits it to the database.

3.1.3 Public facilities planning

a. Facility reservation

Owners can view the use schedule of shared facilities (such as gyms and activity rooms) and make reservations for use (a deposit is required to reserve equipment, and after use, the deposit is returned to the account after the employee confirms the end). Record the reservation time and user information of each facility in the database. Use the Room database to store facility reservation data; provide a simple reservation time selection interface on the front end; and ensure that reservation times are not repeated.

b. Maintenance demand feedback function description

After the owner submits a maintenance request, normal-level employees can initiate the implementation of this request, and admin-level employees can approve the completion of the maintenance after confirmation. For maintenance requests with increased maintenance budgets, manager-level employees can approve the implementation of high-budget maintenance.

c. Facility usage status display

Displays the current status of the facility, such as whether it is in use, idle, or under maintenance. Use simple state management, no need for real-time updates. Create a status table or field that allows property staff to update the facility's status in the background, and residents to read and display the facility's status.

d. Takeaway service

ZhongHai Property provides food delivery service. Owners can choose the food and complete the payment. After the food is delivered, the owner confirms the receipt by clicking the confirmation button. Then the food delivery service ends.

3.2 SYSTEM MODEL

The system model section outlines the object-oriented approach used to build the project. The focus of this section is to represent the interaction, processes and functional requirements between visualization and the system through diagrams

3.2.1 Case Diagram

Case diagrams: The diagram describe the functional requirements of the system and illustrate the different actions a user can perform in the application.

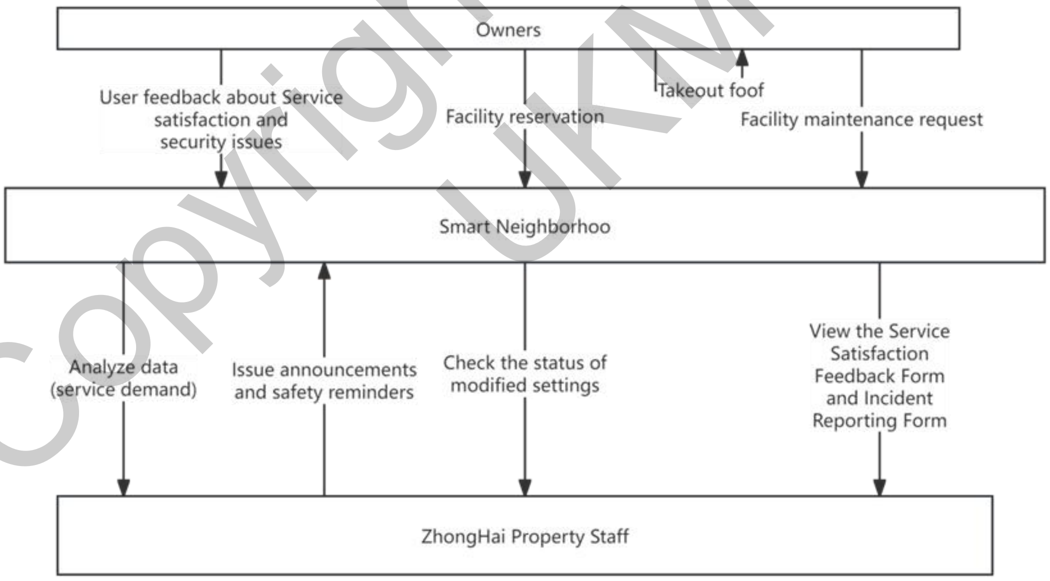

Owners
User feedback about Service satisfaction and security issues
Facility reservation
Takeout foof
Facility maintenance request
Smart Neighborhoo
Analyze data (service demand)
Issue announcements and safety reminders
Check the status of modified settings
View the Service Satisfaction Feedback Form and Incident Reporting Form
ZhongHai Property Staff

Figure 3.1 Use cases of "Smart Neighborhood" intelligent property service management system

3.2.2 Sequence diagram

Sequence diagrams: These diagrams show the sequence of interactions between users and the system during key processes.

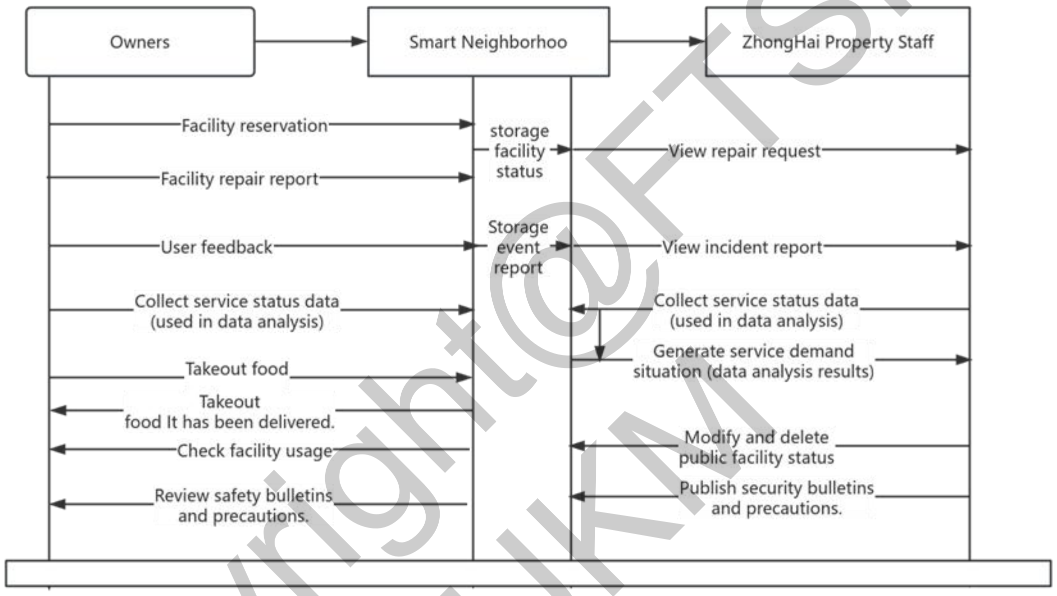

Owners
Smart Neighborhoo
ZhongHai Property Staff
Facility reservation
Facility repair report
User feedback
Collect service status data (used in data analysis)
Takeout food
Takeout food It has been delivered.
Check facility usage
Review safety bulletins and precautions.
storage facility status
Storage event report
View repair request
View incident report
Collect service status data (used in data analysis)
Generate service demand situation (data analysis results)
Modify and delete public facility status
Publish security bulletins and precautions.

Figure 3.2 Use sequence of "Smart Neighborhood" intelligent property service management system

3.2.3 Activity diagram

Activity Diagrams: These diagrams depict the step-by-step flow of processes within a system.

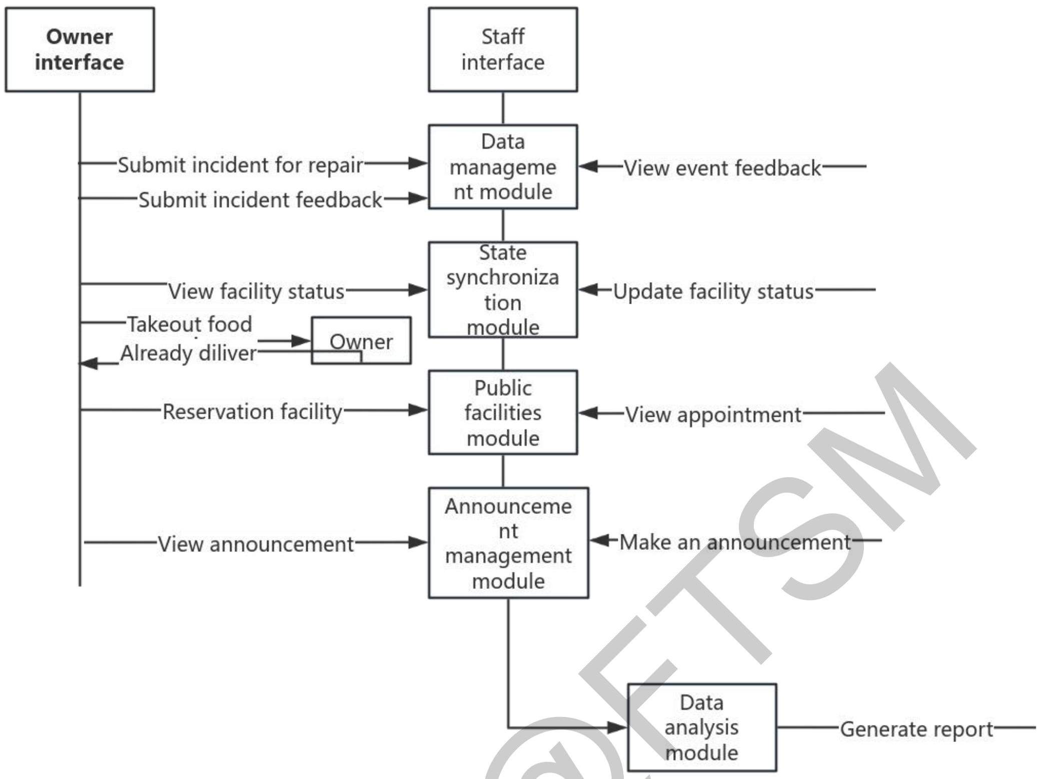


Figure 3.3 Design system interfaces between modules/sub-systems

4.0 RESULTS

4.1 SYSTEM COMPONENTS

a. Register

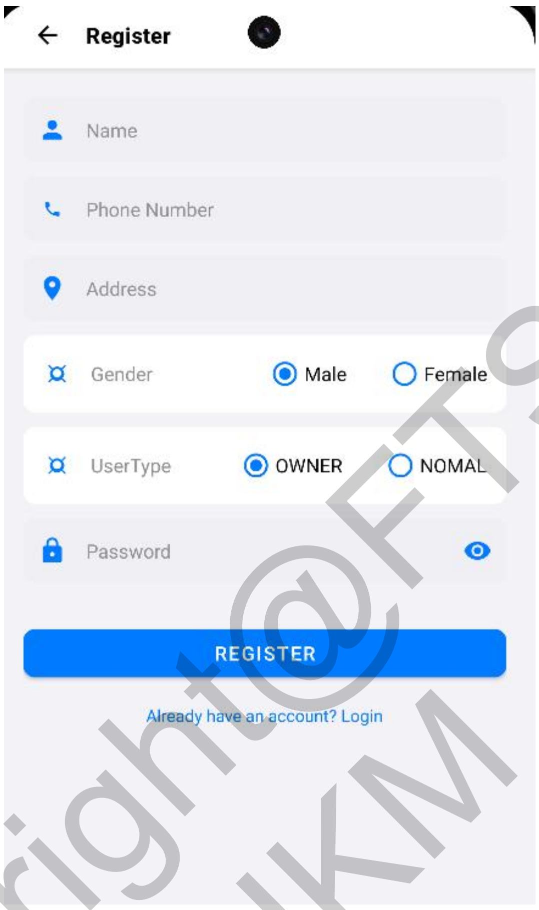
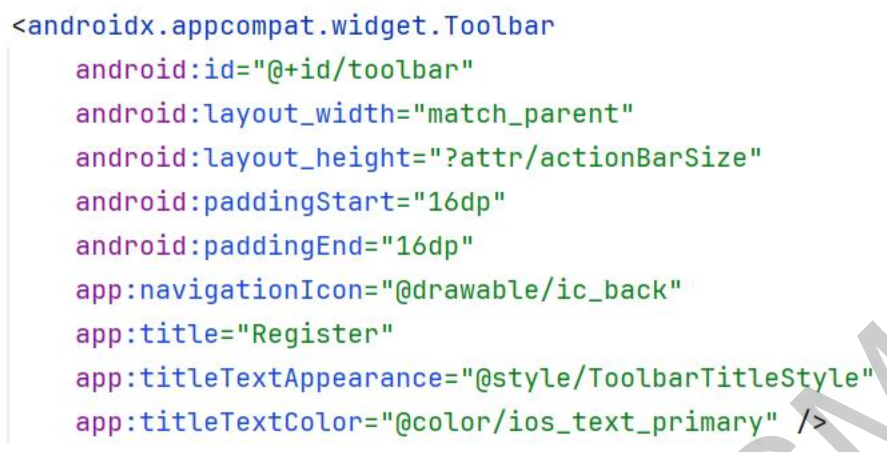


Figure 4.1 Registration interface

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_register);

    // 检查是否来自管理员界面
    isFromAdmin = getIntent().getBooleanExtra( name: "fromAdmin",  defaultValue: false);

    // Initialize the user repository
    userRepository = new UserRepository( context: this);
```

Figure 4.2 initialization phase



```xml
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:paddingStart="16dp"
    android:paddingEnd="16dp"
    app:navigationIcon="@drawable/ic_back"
    app:title="Register"
    app:titleTextAppearance="@style/ToolbarTitleStyle"
    app:titleTextColor="@color/ios_text_primary" />
```

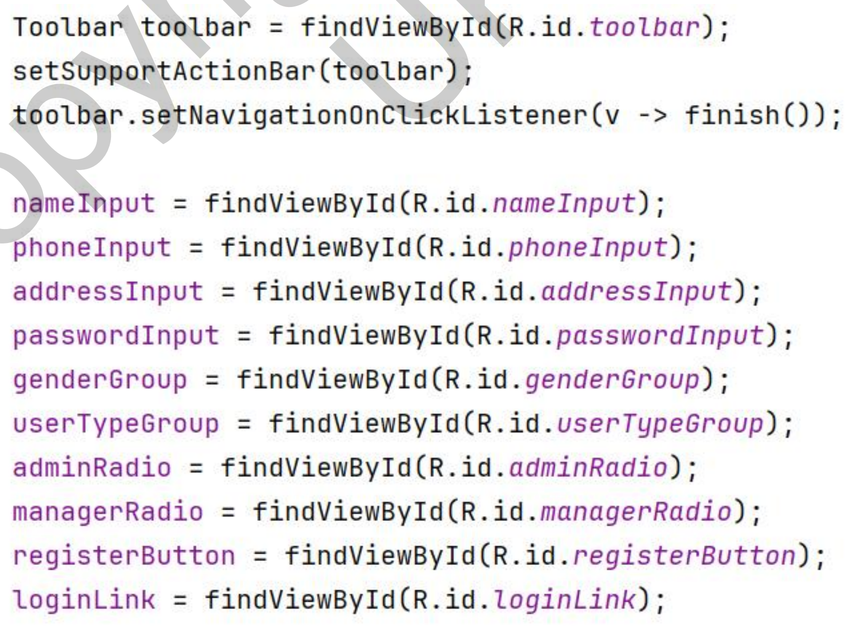
Figure 4.3 Associate the layout file in activity_register.xml with the variables in the code

Set up Toolbar and add back button function. Bind all input controls:

Text input box: name, phone number, address, password

Radio button group: gender, user type

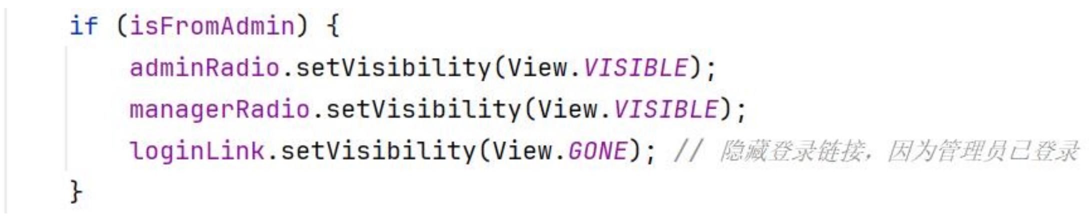
Action button: register button, login link



```java
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
toolbar.setNavigationOnClickListener(v -> finish());

nameInput = findViewById(R.id.nameInput);
phoneInput = findViewById(R.id.phoneInput);
addressInput = findViewById(R.id.addressInput);
passwordInput = findViewById(R.id.passwordInput);
genderGroup = findViewById(R.id.genderGroup);
userTypeGroup = findViewById(R.id.userTypeGroup);
adminRadio = findViewById(R.id.adminRadio);
managerRadio = findViewById(R.id.managerRadio);
registerButton = findViewById(R.id.registerButton);
loginLink = findViewById(R.id.loginLink);
```

Figure 4.4 Interface component binding

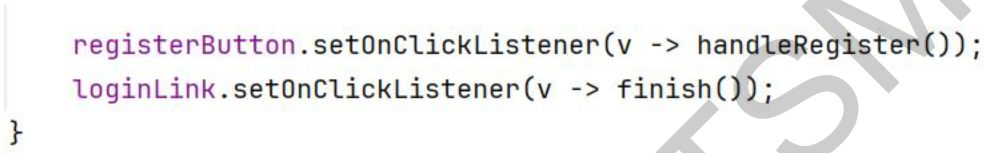
Only Manager can create ADMIN and MANAGER level accounts. Hide the login link when the manager operates (no need to log in again)



```java
if (isFromAdmin) {
    adminRadio.setVisibility(View.VISIBLE);
    managerRadio.setVisibility(View.VISIBLE);
    loginLink.setVisibility(View.GONE); // 隐藏登录链接，因为管理员已登录
}
```

Figure 4.5 Permission control logic

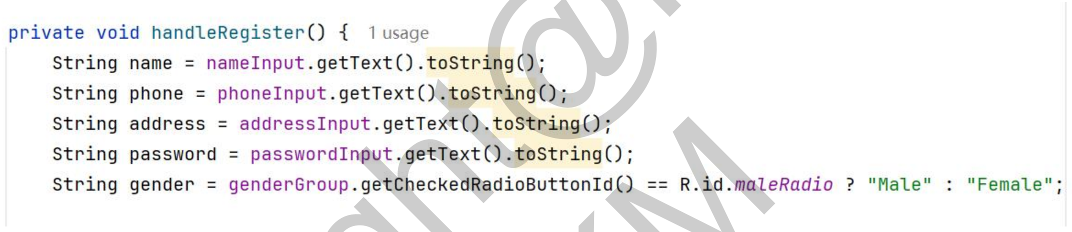
The register button is bound to the handleRegister() method, and the login link directly closes the current interface (returns to the login page).



```java
    registerButton.setOnClickListener(v -> handleRegister());
    loginLink.setOnClickListener(v -> finish());
}
```

Figure 4.6 Register button click processing

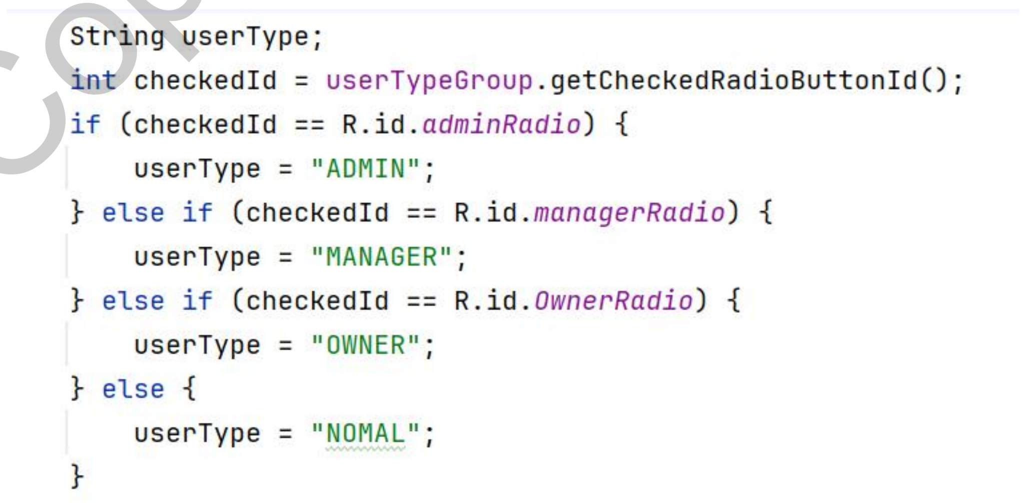
Get text data from input boxes: name, phone number, address, password.



```java
private void handleRegister() {  1 usage
    String name = nameInput.getText().toString();
    String phone = phoneInput.getText().toString();
    String address = addressInput.getText().toString();
    String password = passwordInput.getText().toString();
    String gender = genderGroup.getCheckedRadioButtonId() == R.id.maleRadio ? "Male" : "Female";
```

Figure 4.7 Data collection

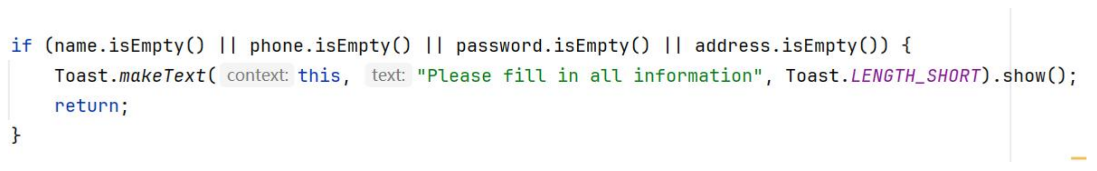
Determine the role type based on the radio button selected by the user. Supports 4 level: ADMIN, MANAGER, OWNER, NORMAL.



```java
String userType;
int checkedId = userTypeGroup.getCheckedRadioButtonId();
if (checkedId == R.id.adminRadio) {
    userType = "ADMIN";
} else if (checkedId == R.id.managerRadio) {
    userType = "MANAGER";
} else if (checkedId == R.id.OwnerRadio) {
    userType = "OWNER";
} else {
    userType = "NOMAL";
}
```

Figure 4.7 Role determination logic

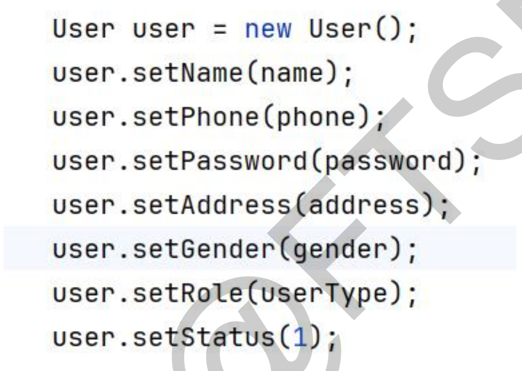
Check if the key fields are empty (name, phone number, password, address). If there are empty fields, display a prompt and terminate the registration process.



```
if (name.isEmpty() || phone.isEmpty() || password.isEmpty() || address.isEmpty()) {
    Toast.makeText( context: this,  text: "Please fill in all information", Toast.LENGTH_SHORT).show();
    return;
}
```

Figure 4.8 Input validation

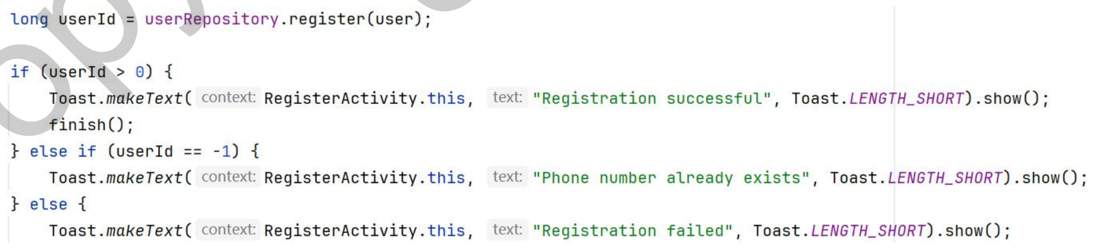
Create a user object and set all properties. Status=1 indicates that the account is activated by default.



```
User user = new User();
user.setName(name);
user.setPhone(phone);
user.setPassword(password);
user.setAddress(address);
user.setGender(gender);
user.setRole(userType);
user.setStatus(1);
```

Figure 4.9 User object creation

Call UserRepository to store user data in the database, and process three results: Success (userId > 0): Display a success prompt and close the interface. Number already exists (userId == -1): Prompt a duplicate number. Other failures: Display registration failure.

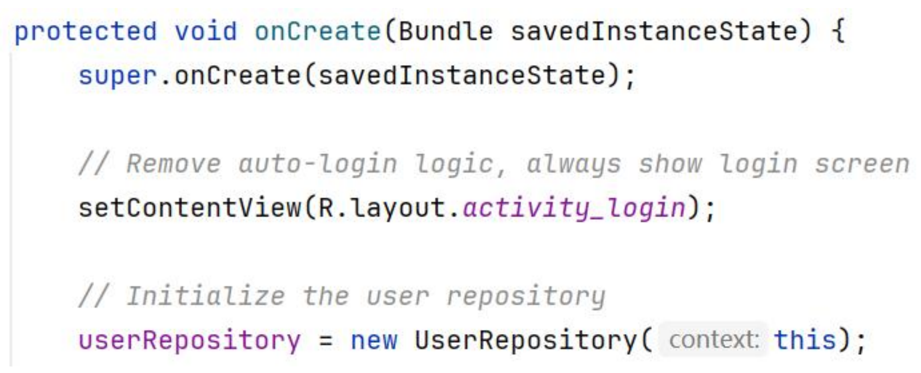


```
long userId = userRepository.register(user);

if (userId > 0) {
    Toast.makeText( context: RegisterActivity.this,  text: "Registration successful", Toast.LENGTH_SHORT).show();
    finish();
} else if (userId == -1) {
    Toast.makeText( context: RegisterActivity.this,  text: "Phone number already exists", Toast.LENGTH_SHORT).show();
} else {
    Toast.makeText( context: RegisterActivity.this,  text: "Registration failed", Toast.LENGTH_SHORT).show();
```

Figure 4.10 Bank Type

b. Login

The layout of the login interface is set from (activity_login.xml). The initial UserRepository is used for database verification operations. Remove the automatic login logic and the login interface will be displayed every time.

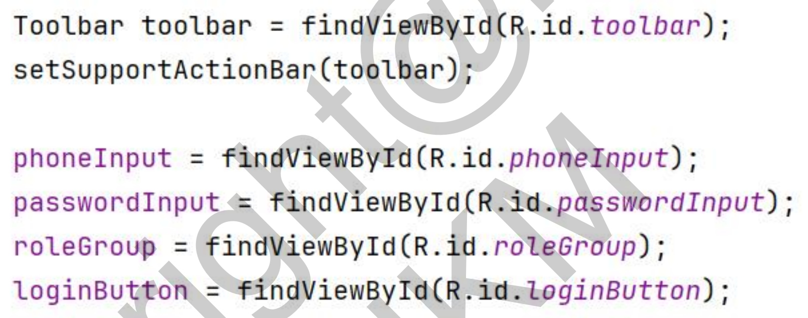


```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Remove auto-login logic, always show login screen
    setContentView(R.layout.activity_login);

    // Initialize the user repository
    userRepository = new UserRepository( context: this);
```

Figure 4.11 Initialization phase

Configure Toolbar as the top operation bar of the interface. Bind key input controls:

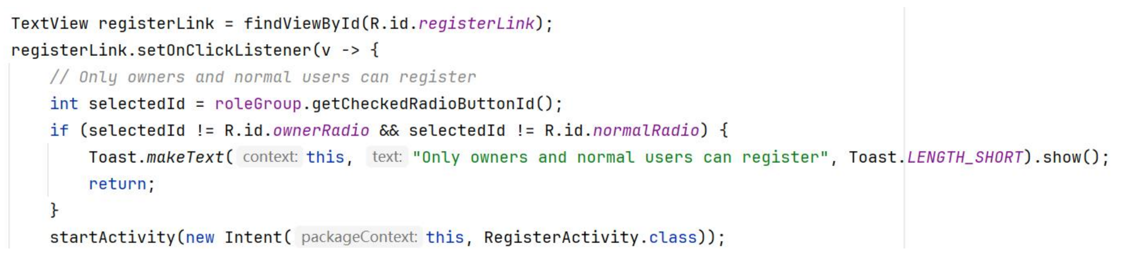
Phone number input box, Password input box, Role selection radio button. Login button.



```
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

phoneInput = findViewById(R.id.phoneInput);
passwordInput = findViewById(R.id.passwordInput);
roleGroup = findViewById(R.id.roleGroup);
loginButton = findViewById(R.id.loginButton);
```

Figure 4.12 Interface component binding

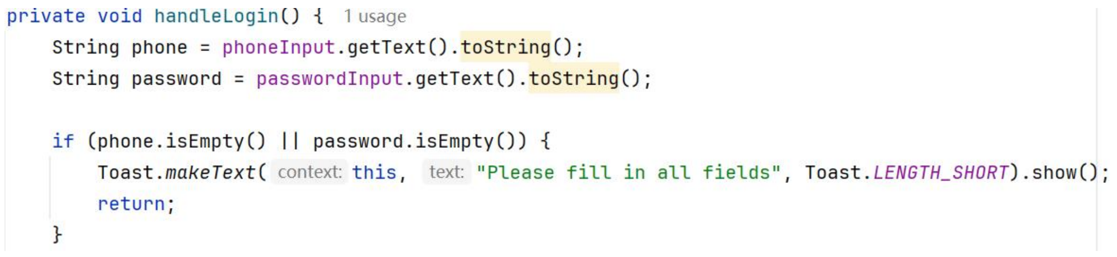
When the registration link is clicked, the currently selected role is checked. Only OWNER and NORMAL level employees are allowed to register. When the admin and manager level are clicked, the permission prompt will be displayed. After the permission verification is passed, jump to RegisterActivity.



```
TextView registerLink = findViewById(R.id.registerLink);
registerLink.setOnClickListener(v -> {
    // Only owners and normal users can register
    int selectedId = roleGroup.getCheckedRadioButtonId();
    if (selectedId != R.id.ownerRadio && selectedId != R.id.normalRadio) {
        Toast.makeText( context: this,  text: "Only owners and normal users can register", Toast.LENGTH_SHORT).show();
        return;
    }
    startActivity(new Intent( packageContext: this, RegisterActivity.class));
```

Figure 4.13 Registration link control logic

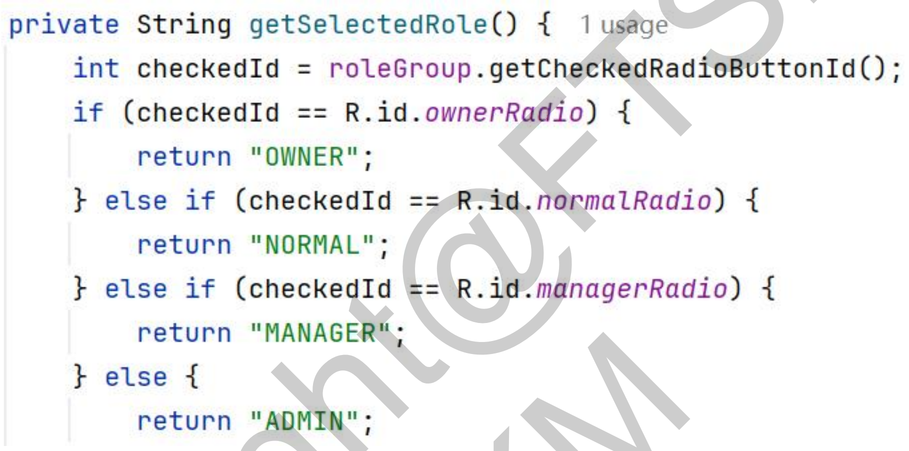
Get the phone number and password entered by the user. Basic validation ensures that both fields are not empty. If there are empty fields, display a prompt and terminate the login process.



```
private void handleLogin() {  1 usage
    String phone = phoneInput.getText().toString();
    String password = passwordInput.getText().toString();

    if (phone.isEmpty() || password.isEmpty()) {
        Toast.makeText( context: this,  text: "Please fill in all fields", Toast.LENGTH_SHORT).show();
        return;
    }
```

Figure 4.14 Login processing entrance

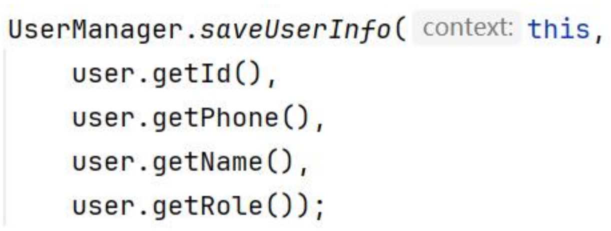
Get the corresponding identity and their respective levels through the button group. Return the string of the corresponding identity (OWNER/NORMAL/MANAGER/ADMIN).



```
private String getSelectedRole() {  1 usage
    int checkedId = roleGroup.getCheckedRadioButtonId();
    if (checkedId == R.id.ownerRadio) {
        return "OWNER";
    } else if (checkedId == R.id.normalRadio) {
        return "NORMAL";
    } else if (checkedId == R.id.managerRadio) {
        return "MANAGER";
    } else {
        return "ADMIN";
```

Figure 4.15 Level acquisition logic

After successful login, use UserManager to save user information. Persistently store user ID, phone number, name and role. Provide user session status for the entire application.



```
UserManager.saveUserInfo( context: this,
    user.getId(),
    user.getPhone(),
    user.getName(),
    user.getRole());
```

Figure 4.16 Session management

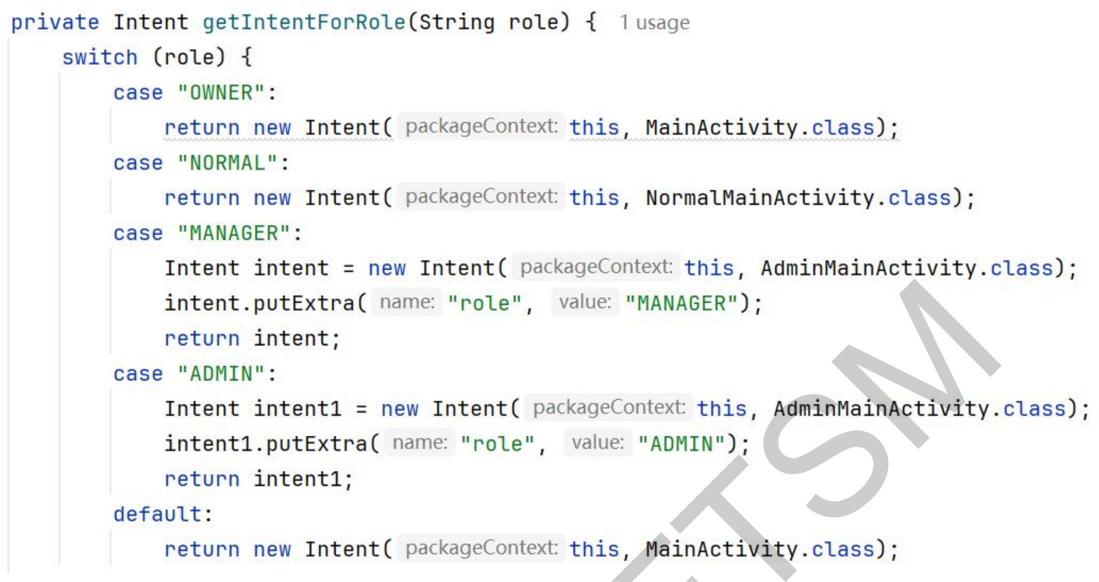
Jump to different main interfaces according to the user

OWNER → MainActivity,

NORMAL → NormalMainActivity,

MANAGER/ADMIN → AdminMainActivity.

Close the login interface through finish() to prevent returning.

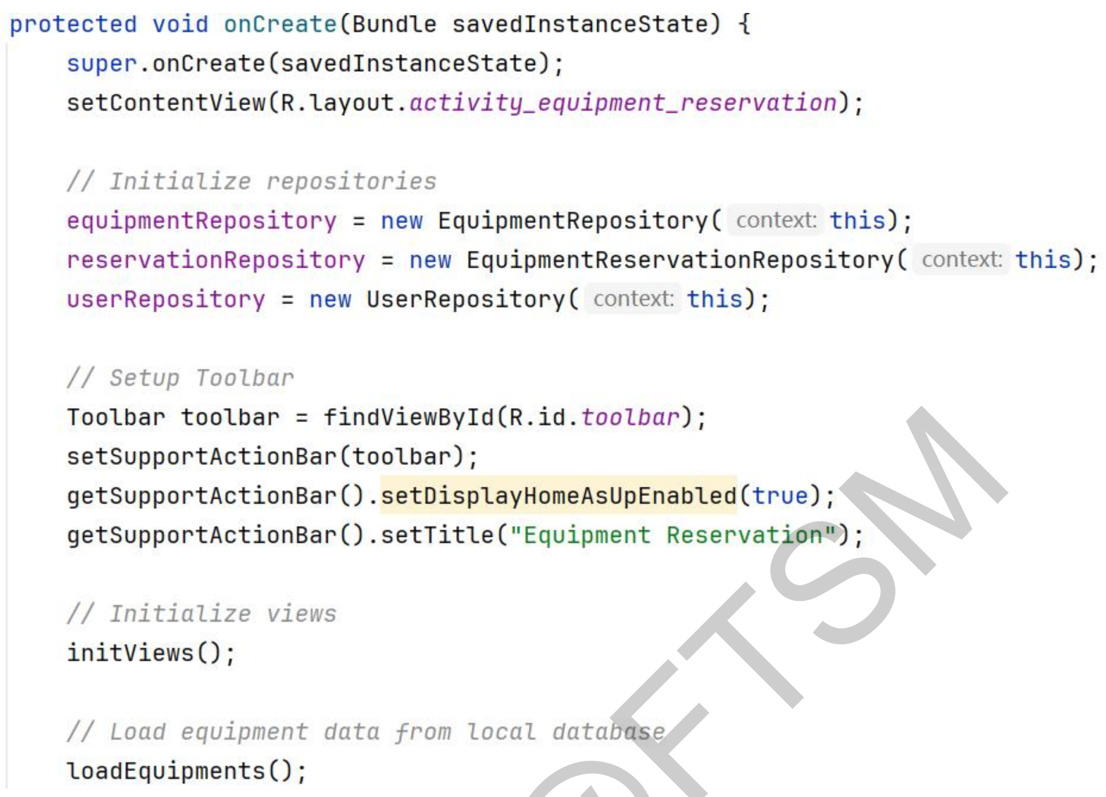


```
private Intent getIntentForRole(String role) {  1 usage
    switch (role) {
        case "OWNER":
            return new Intent( packageContext: this, MainActivity.class);
        case "NORMAL":
            return new Intent( packageContext: this, NormalMainActivity.class);
        case "MANAGER":
            Intent intent = new Intent( packageContext: this, AdminMainActivity.class);
            intent.putExtra( name: "role",  value: "MANAGER");
            return intent;
        case "ADMIN":
            Intent intent1 = new Intent( packageContext: this, AdminMainActivity.class);
            intent1.putExtra( name: "role",  value: "ADMIN");
            return intent1;
        default:
            return new Intent( packageContext: this, MainActivity.class);
```

Figure 4.17 Navigation based on account level and identity

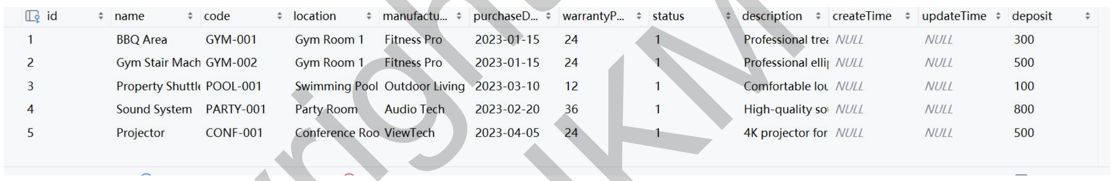
c. Equipment Reservation

Set the layout file activity_equipment_reservation. Initialize three data warehouses (equipment, equipment_reservation, user). Configure Toolbar and set the return button. Call initViews() to initialize the interface components. Call loadEquipments() to load equipment data.



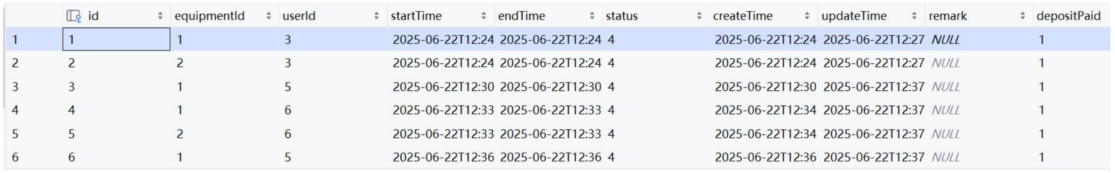
```java
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_equipment_reservation);

    // Initialize repositories
    equipmentRepository = new EquipmentRepository( context: this);
    reservationRepository = new EquipmentReservationRepository( context: this);
    userRepository = new UserRepository( context: this);

    // Setup Toolbar
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    getSupportActionBar().setTitle("Equipment Reservation");

    // Initialize views
    initViews();

    // Load equipment data from local database
    loadEquipments();
```

Figure 4.18 Initialization phase



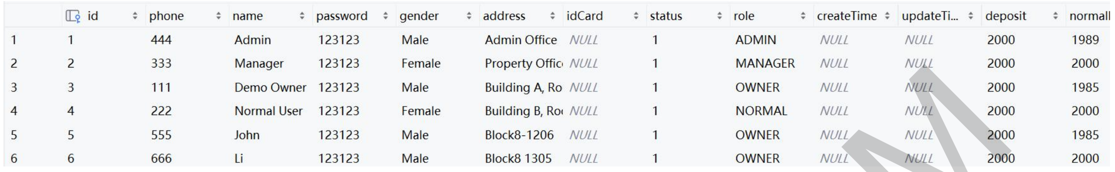
id	name	code	location	manufactu...	purchaseD...	warrantyP...	status	description	createTime	updateTime	deposit
1	BBQ Area	GYM-001	Gym Room 1	Fitness Pro	2023-01-15	24	1	Professional trea	NULL	NULL	300
2	Gym Stair Mach	GYM-002	Gym Room 1	Fitness Pro	2023-01-15	24	1	Professional ellip	NULL	NULL	500
3	Property Shuttl	POOL-001	Swimming Pool	Outdoor Living	2023-03-10	12	1	Comfortable lou	NULL	NULL	100
4	Sound System	PARTY-001	Party Room	Audio Tech	2023-02-20	36	1	High-quality so	NULL	NULL	800
5	Projector	CONF-001	Conference Roo	ViewTech	2023-04-05	24	1	4K projector for	NULL	NULL	500

Figure 4.19 Equipment



	id	equipmentId	userId	startTime	endTime	status	createTime	updateTime	remark	depositPaid
1	1	1	3	2025-06-22T12:24	2025-06-22T12:24	4	2025-06-22T12:24	2025-06-22T12:27	NULL	1
2	2	2	3	2025-06-22T12:24	2025-06-22T12:24	4	2025-06-22T12:24	2025-06-22T12:27	NULL	1
3	3	1	5	2025-06-22T12:30	2025-06-22T12:30	4	2025-06-22T12:30	2025-06-22T12:37	NULL	1
4	4	1	6	2025-06-22T12:33	2025-06-22T12:33	4	2025-06-22T12:34	2025-06-22T12:37	NULL	1
5	5	2	6	2025-06-22T12:33	2025-06-22T12:33	4	2025-06-22T12:34	2025-06-22T12:37	NULL	1
6	6	1	5	2025-06-22T12:36	2025-06-22T12:36	4	2025-06-22T12:36	2025-06-22T12:37	NULL	1

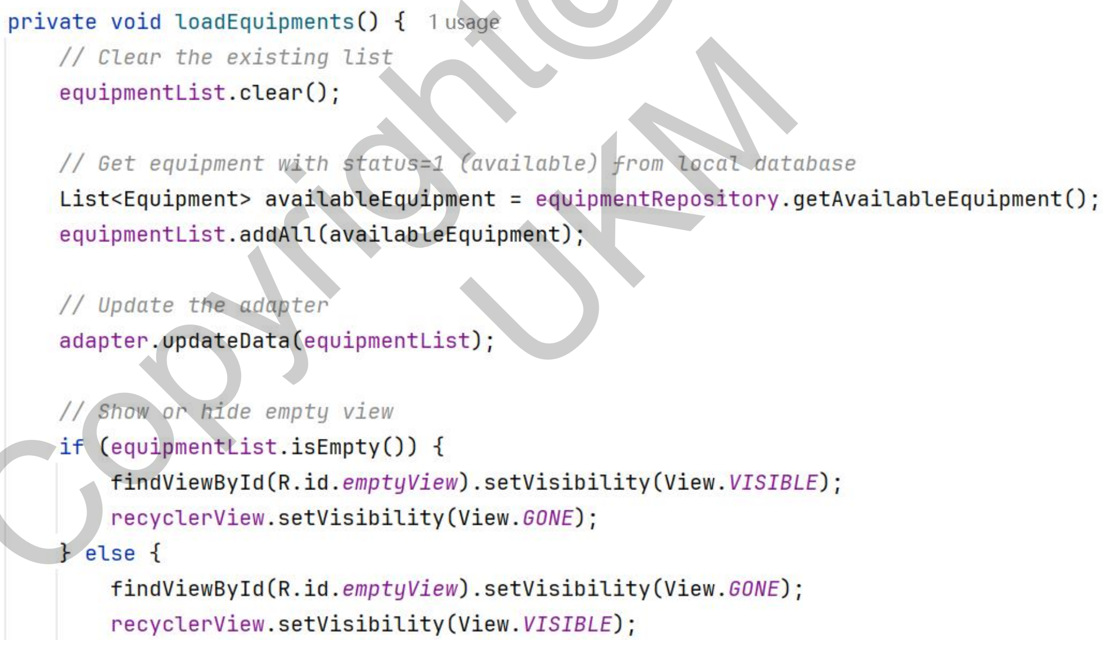
Figure 4.20 Equipment_reservation



	id	phone	name	password	gender	address	idCard	status	role	createTime	updateTi...	deposit	normall
1	1	444	Admin	123123	Male	Admin Office	NULL	1	ADMIN	NULL	NULL	2000	1989
2	2	333	Manager	123123	Female	Property Offic	NULL	1	MANAGER	NULL	NULL	2000	2000
3	3	111	Demo Owner	123123	Male	Building A, Ro	NULL	1	OWNER	NULL	NULL	2000	1985
4	4	222	Normal User	123123	Female	Building B, Ro	NULL	1	NORMAL	NULL	NULL	2000	2000
5	5	555	John	123123	Male	Block8-1206	NULL	1	OWNER	NULL	NULL	2000	1985
6	6	666	Li	123123	Male	Block8 1305	NULL	1	OWNER	NULL	NULL	2000	2000

Figure 4.21 User

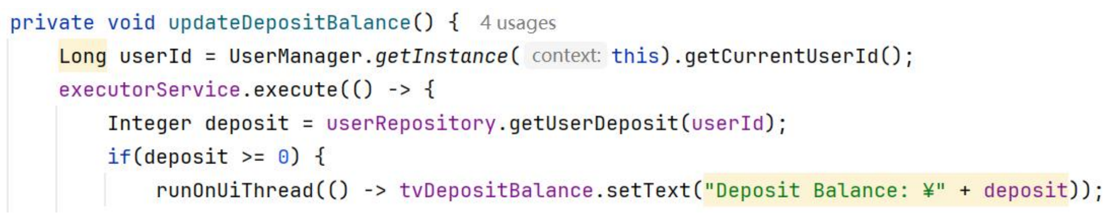
Clear the current device list. Get devices with the status "available" from the database. Update adapter data. Display an empty view or list view depending on whether there are devices.



```
private void loadEquipments() {  1 usage
    // Clear the existing list
    equipmentList.clear();

    // Get equipment with status=1 (available) from local database
    List<Equipment> availableEquipment = equipmentRepository.getAvailableEquipment();
    equipmentList.addAll(availableEquipment);

    // Update the adapter
    adapter.updateData(equipmentList);

    // Show or hide empty view
    if (equipmentList.isEmpty()) {
        findViewById(R.id.emptyView).setVisibility(View.VISIBLE);
        recyclerView.setVisibility(View.GONE);
    } else {
        findViewById(R.id.emptyView).setVisibility(View.GONE);
        recyclerView.setVisibility(View.VISIBLE);
```

Figure 4.22 Device list loading

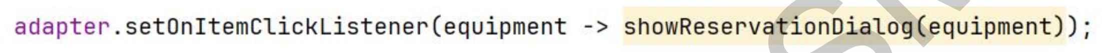
Get the current user ID from UserManager. Query the user's deposit balance in the background thread. Update the UI to display the balance in the main thread.



```
private void updateDepositBalance() {  4 usages
    Long userId = UserManager.getInstance( context: this).getCurrentUserId();
    executorService.execute(() -> {
        Integer deposit = userRepository.getUserDeposit(userId);
        if(deposit >= 0) {
            runOnUiThread(() -> tvDepositBalance.setText("Deposit Balance: ¥" + deposit));
```

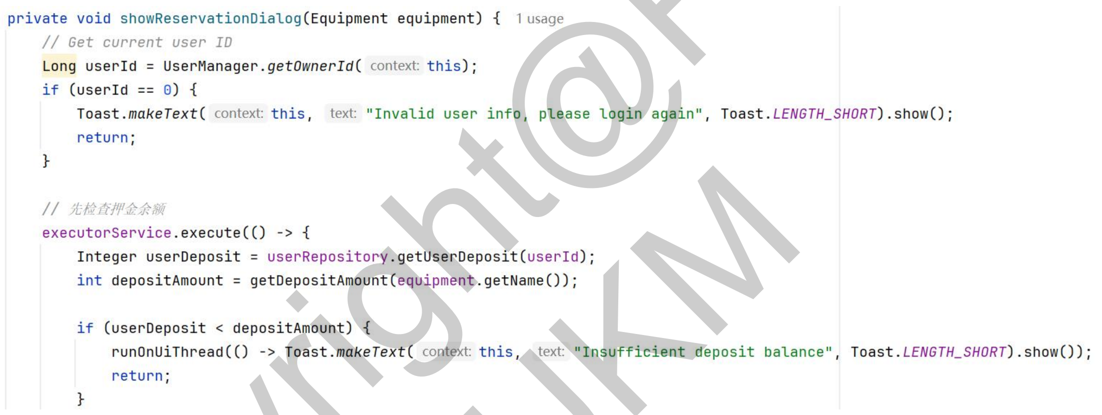
Figure 4.23 Deposit balance management

Set a click listener for each item in RecyclerView. When clicked, call showReservationDialog() to display the reservation dialog.



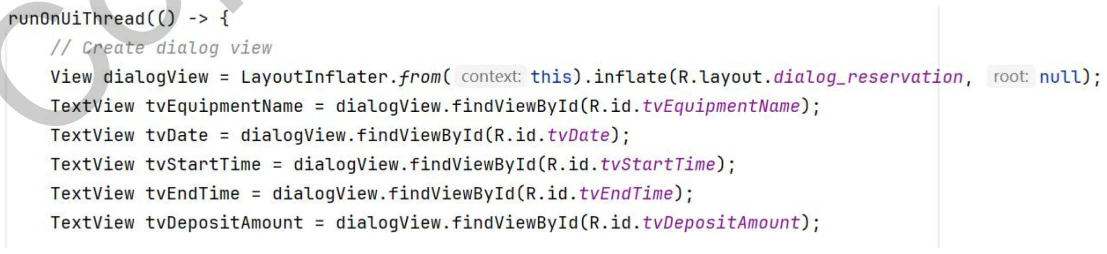
```
adapter.setOnItemClickListener(equipment -> showReservationDialog(equipment));
```

Figure 4.24 Click to trigger the listener to make an appointment.



```
private void showReservationDialog(Equipment equipment) {  1 usage
    // Get current user ID
    Long userId = UserManager.getOwnerId( context: this);
    if (userId == 0) {
        Toast.makeText( context: this,  text: "Invalid user info, please login again", Toast.LENGTH_SHORT).show();
        return;
    }

    // 先检查押金余额
    executorService.execute(() -> {
        Integer userDeposit = userRepository.getUserDeposit(userId);
        int depositAmount = getDepositAmount(equipment.getName());

        if (userDeposit < depositAmount) {
            runOnUiThread(() -> Toast.makeText( context: this,  text: "Insufficient deposit balance", Toast.LENGTH_SHORT).show());
            return;
        }
```

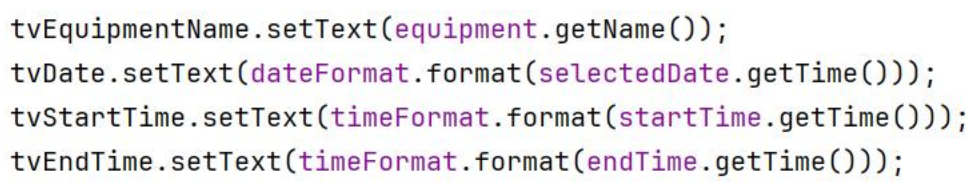
Figure 4.25 Check whether the user's deposit is sufficient



```
runOnUiThread(() -> {
    // Create dialog view
    View dialogView = LayoutInflater.from( context: this).inflate(R.layout.dialog_reservation,  root: null);
    TextView tvEquipmentName = dialogView.findViewById(R.id.tvEquipmentName);
    TextView tvDate = dialogView.findViewById(R.id.tvDate);
    TextView tvStartTime = dialogView.findViewById(R.id.tvStartTime);
    TextView tvEndTime = dialogView.findViewById(R.id.tvEndTime);
    TextView tvDepositAmount = dialogView.findViewById(R.id.tvDepositAmount);
```

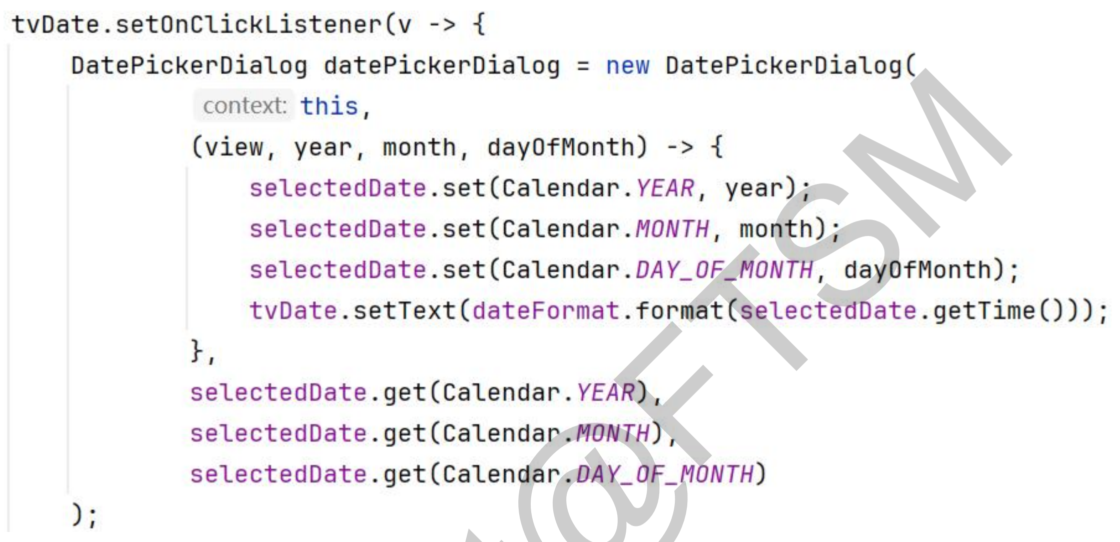
Figure 4.26 The deposit is sufficient, continue the reservation process

The minimum date for date selection is the current date. Time selection uses the 24-hour system. The UI display is updated immediately after selection.



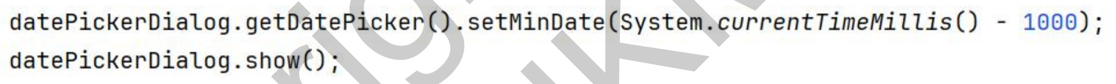
```
tvEquipmentName.setText(equipment.getName());
tvDate.setText(dateFormat.format(selectedDate.getTime()));
tvStartTime.setText(timeFormat.format(startTime.getTime()));
tvEndTime.setText(timeFormat.format(endTime.getTime()));
```

Figure 4.27 Show deposit amount



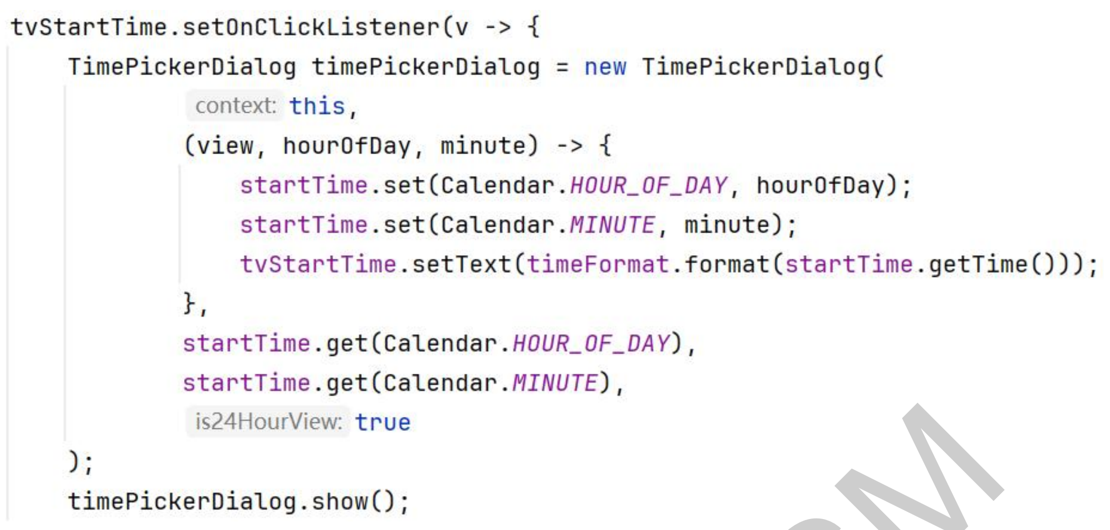
```
tvDate.setOnClickListener(v -> {
    DatePickerDialog datePickerDialog = new DatePickerDialog(
            context: this,
            (view, year, month, dayOfMonth) -> {
                selectedDate.set(Calendar.YEAR, year);
                selectedDate.set(Calendar.MONTH, month);
                selectedDate.set(Calendar.DAY_OF_MONTH, dayOfMonth);
                tvDate.setText(dateFormat.format(selectedDate.getTime()));
            },
            selectedDate.get(Calendar.YEAR),
            selectedDate.get(Calendar.MONTH),
            selectedDate.get(Calendar.DAY_OF_MONTH)
    );
```

Figure 4.28 Set date picker



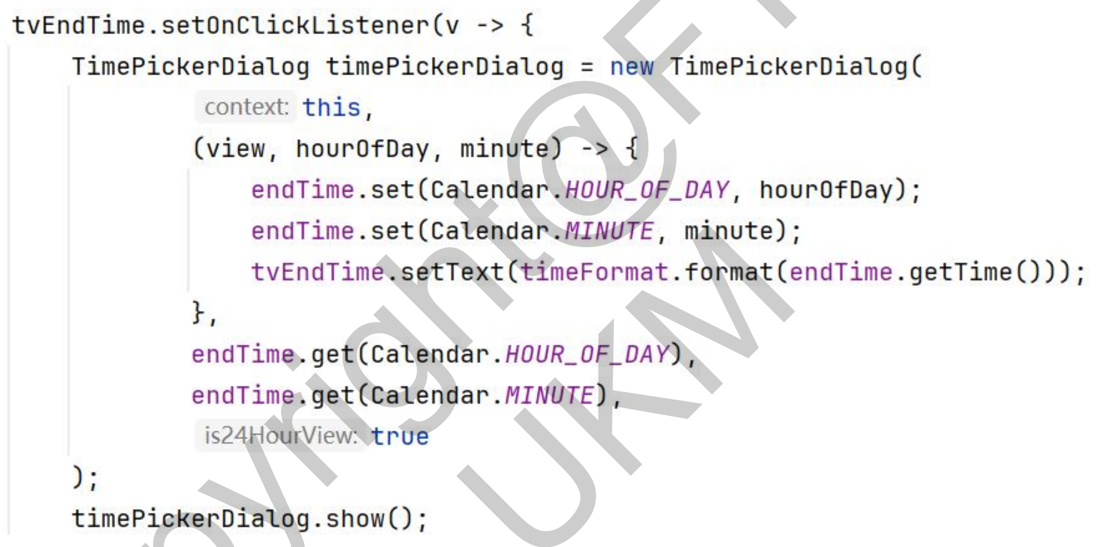
```
datePickerDialog.getDatePicker().setMinDate(System.currentTimeMillis() - 1000);
datePickerDialog.show();
```

Figure 4.29 Set minimum date to today



```
tvStartTime.setOnClickListener(v -> {
    TimePickerDialog timePickerDialog = new TimePickerDialog(
            context: this,
            (view, hourOfDay, minute) -> {
                startTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
                startTime.set(Calendar.MINUTE, minute);
                tvStartTime.setText(timeFormat.format(startTime.getTime()));
            },
            startTime.get(Calendar.HOUR_OF_DAY),
            startTime.get(Calendar.MINUTE),
            is24HourView: true
    );
    timePickerDialog.show();
```

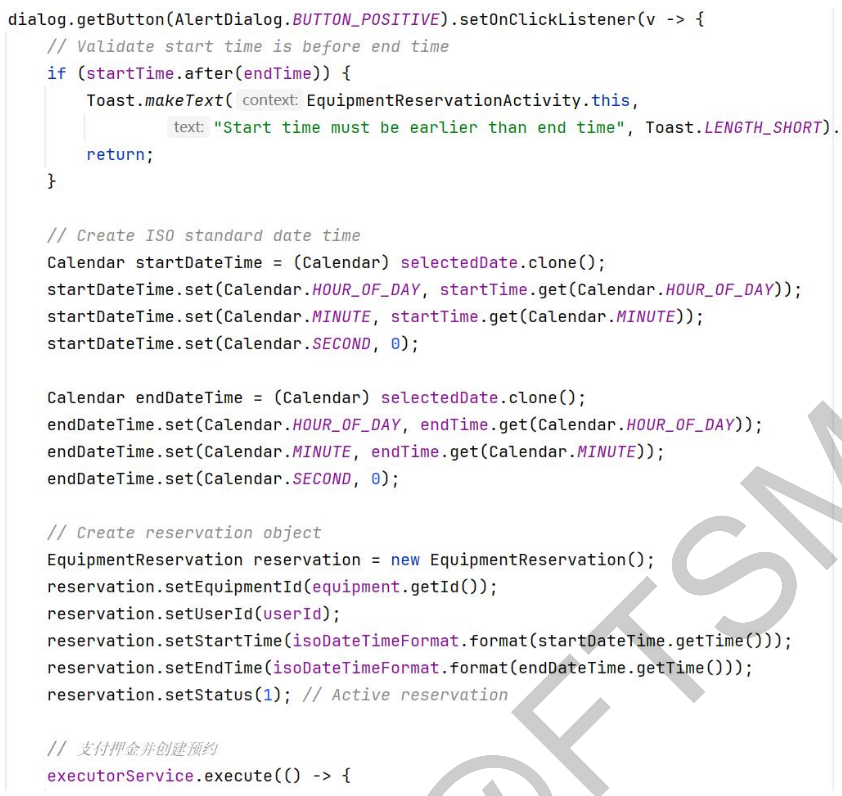
Figure 4.30 Set start time picker



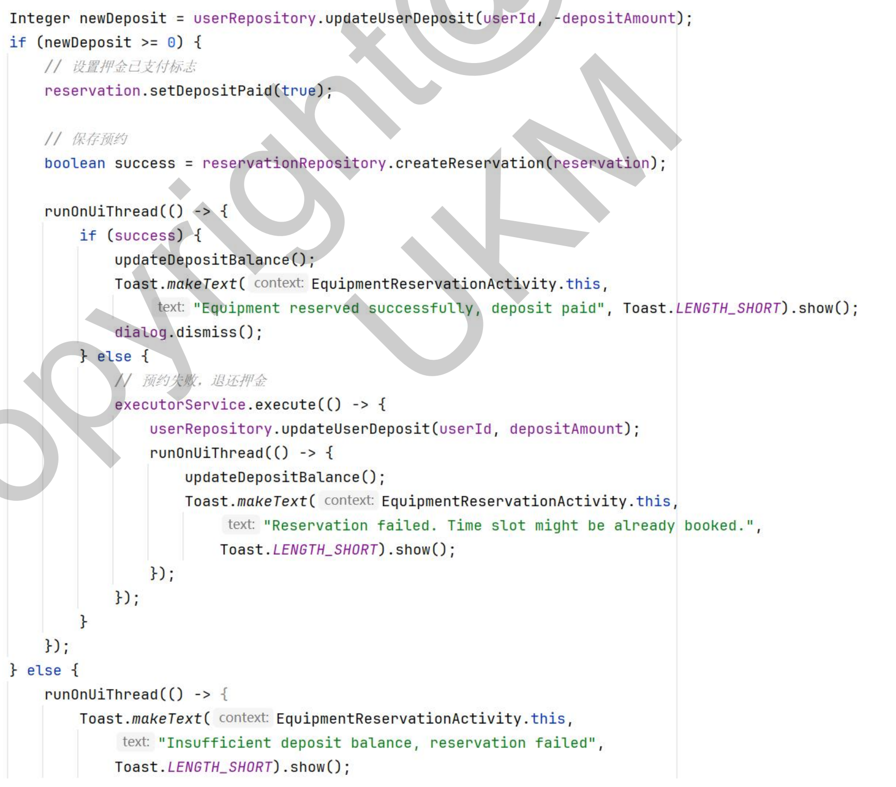
```
tvEndTime.setOnClickListener(v -> {
    TimePickerDialog timePickerDialog = new TimePickerDialog(
            context: this,
            (view, hourOfDay, minute) -> {
                endTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
                endTime.set(Calendar.MINUTE, minute);
                tvEndTime.setText(timeFormat.format(endTime.getTime()));
            },
            endTime.get(Calendar.HOUR_OF_DAY),
            endTime.get(Calendar.MINUTE),
            is24HourView: true
    );
    timePickerDialog.show();
```

Figure 4.31 Set end time picker

Verify that the start time is earlier than the end time. Create an appointment object and set properties. Deduct the corresponding deposit and try to create an appointment. Success: Display a success prompt. Failure: Refund the deposit and display the reason for failure. Update the deposit balance display.

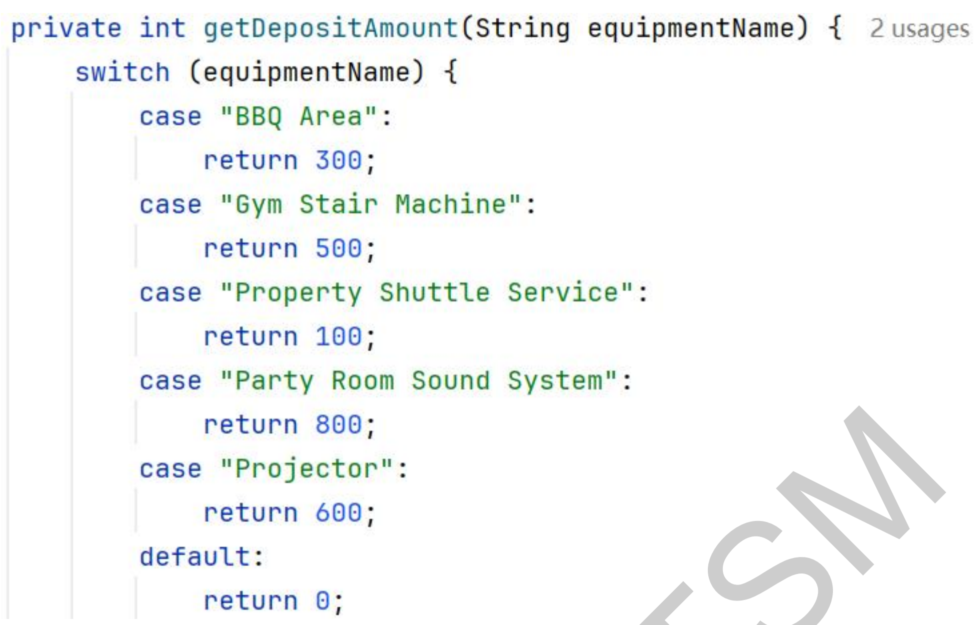


```
dialog.getButton(AlertDialog.BUTTON_POSITIVE).setOnClickListener(v -> {
    // Validate start time is before end time
    if (startTime.after(endTime)) {
        Toast.makeText( context: EquipmentReservationActivity.this,
                 text: "Start time must be earlier than end time", Toast.LENGTH_SHORT).show();
        return;
    }

    // Create ISO standard date time
    Calendar startDateTime = (Calendar) selectedDate.clone();
    startDateTime.set(Calendar.HOUR_OF_DAY, startTime.get(Calendar.HOUR_OF_DAY));
    startDateTime.set(Calendar.MINUTE, startTime.get(Calendar.MINUTE));
    startDateTime.set(Calendar.SECOND, 0);

    Calendar endDateTime = (Calendar) selectedDate.clone();
    endDateTime.set(Calendar.HOUR_OF_DAY, endTime.get(Calendar.HOUR_OF_DAY));
    endDateTime.set(Calendar.MINUTE, endTime.get(Calendar.MINUTE));
    endDateTime.set(Calendar.SECOND, 0);

    // Create reservation object
    EquipmentReservation reservation = new EquipmentReservation();
    reservation.setEquipmentId(equipment.getId());
    reservation.setUserId(userId);
    reservation.setStartTime(isoDateTimeFormat.format(startDateTime.getTime()));
    reservation.setEndTime(isoDateTimeFormat.format(endDateTime.getTime()));
    reservation.setStatus(1); // Active reservation

    // 支付押金并创建预约
    executorService.execute(() -> {
```



```
Integer newDeposit = userRepository.updateUserDeposit(userId, -depositAmount);
if (newDeposit >= 0) {
    // 设置押金已支付标志
    reservation.setDepositPaid(true);

    // 保存预约
    boolean success = reservationRepository.createReservation(reservation);

    runOnUiThread(() -> {
        if (success) {
            updateDepositBalance();
            Toast.makeText( context: EquipmentReservationActivity.this,
                    text: "Equipment reserved successfully, deposit paid", Toast.LENGTH_SHORT).show();
            dialog.dismiss();
        } else {
            // 预约失败，退还押金
            executorService.execute(() -> {
                userRepository.updateUserDeposit(userId, depositAmount);
                runOnUiThread(() -> {
                    updateDepositBalance();
                    Toast.makeText( context: EquipmentReservationActivity.this,
                            text: "Reservation failed. Time slot might be already booked.",
                            Toast.LENGTH_SHORT).show();
                });
            });
        }
    });
} else {
    runOnUiThread(() -> {
        Toast.makeText( context: EquipmentReservationActivity.this,
                text: "Insufficient deposit balance, reservation failed",
                Toast.LENGTH_SHORT).show();
```

Figure 4.32　　Confirm reservation logic

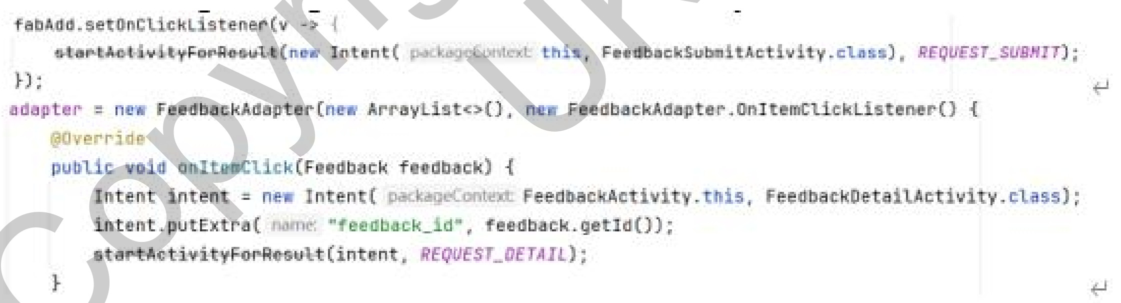


```java
private int getDepositAmount(String equipmentName) {  2 usages
    switch (equipmentName) {
        case "BBQ Area":
            return 300;
        case "Gym Stair Machine":
            return 500;
        case "Property Shuttle Service":
            return 100;
        case "Party Room Sound System":
            return 800;
        case "Projector":
            return 600;
        default:
            return 0;
```

Figure 4.33 Deposit for different projects

d. User feedback: Includes processes for owners and property staff (front-end and back-end).

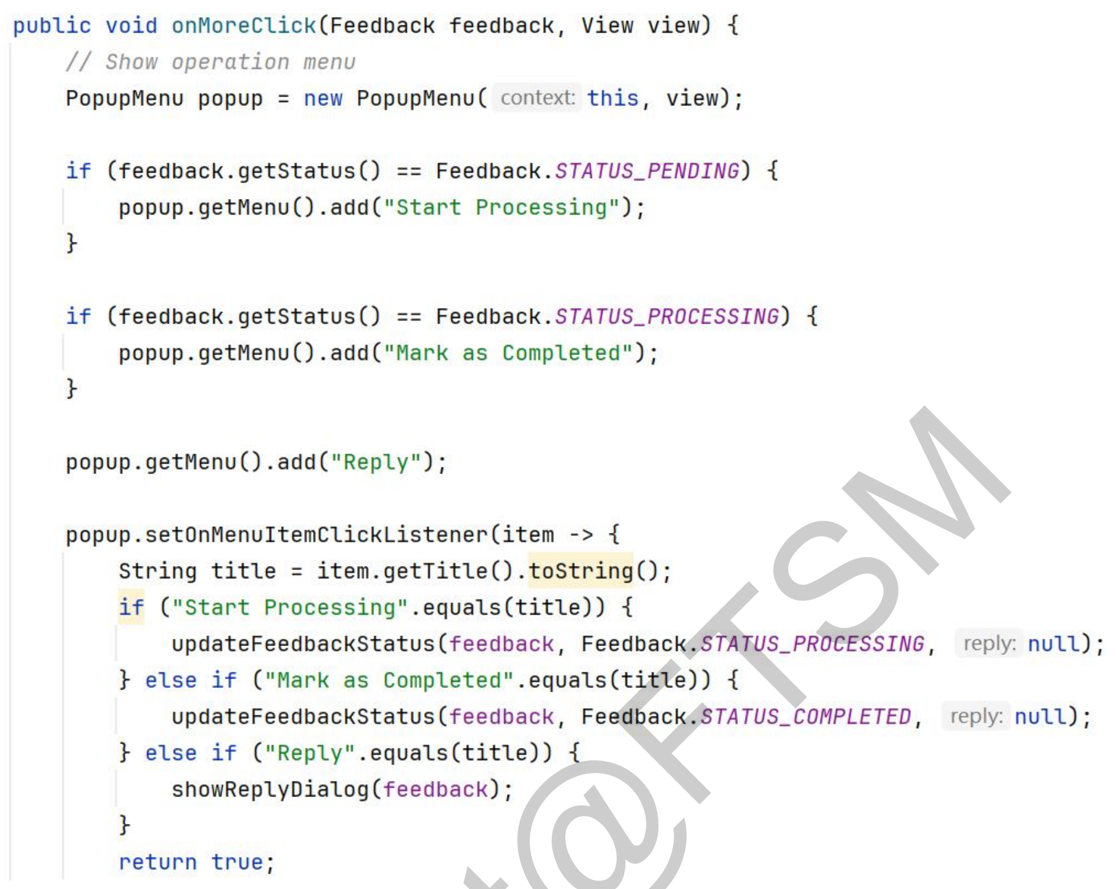
From FeedbackActivity, owners can: Click the "+" button to enter FeedbackSubmitActivity to submit feedback. Click a list item to enter FeedbackDetailActivity to view details. After submitting/viewing, return to the list to automatically refresh the data.



```java
fabAdd.setOnClickListener(v -> {
    startActivityForResult(new Intent( packageContext: this, FeedbackSubmitActivity.class), REQUEST_SUBMIT);
});
adapter = new FeedbackAdapter(new ArrayList<>(), new FeedbackAdapter.OnItemClickListener() {
    @Override
    public void onItemClick(Feedback feedback) {
        Intent intent = new Intent( packageContext: FeedbackActivity.this, FeedbackDetailActivity.class);
        intent.putExtra( name: "feedback_id", feedback.getId());
        startActivityForResult(intent, REQUEST_DETAIL);
    }
```

Figure 4.34 User feedback Owner-side functional process

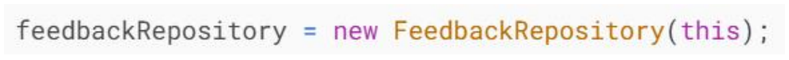
On the list page, you can use the drop-down menu: Start processing (Pending→Processing). Mark completed (Processing→Completed). Reply directly (automatically marked as Completed). Supports multi-condition filtering and search.



```
public void onMoreClick(Feedback feedback, View view) {
    // Show operation menu
    PopupMenu popup = new PopupMenu( context: this, view);

    if (feedback.getStatus() == Feedback.STATUS_PENDING) {
        popup.getMenu().add("Start Processing");
    }

    if (feedback.getStatus() == Feedback.STATUS_PROCESSING) {
        popup.getMenu().add("Mark as Completed");
    }

    popup.getMenu().add("Reply");

    popup.setOnMenuItemClickListener(item -> {
        String title = item.getTitle().toString();
        if ("Start Processing".equals(title)) {
            updateFeedbackStatus(feedback, Feedback.STATUS_PROCESSING,  reply: null);
        } else if ("Mark as Completed".equals(title)) {
            updateFeedbackStatus(feedback, Feedback.STATUS_COMPLETED,  reply: null);
        } else if ("Reply".equals(title)) {
            showReplyDialog(feedback);
        }
        return true;
```

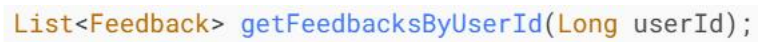
Figure 4.35 Property management company employee process

Unified data access portal. User data isolation (owners can only see their own feedback). Thread-safe database operations.



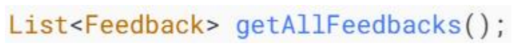
```
feedbackRepository = new FeedbackRepository(this);
```

Figure 4.36 Repository shared by all Activities



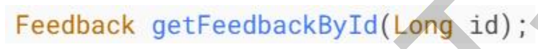
```
List<Feedback> getFeedbacksByUserId(Long userId);
```

Figure 4.37 Owner use



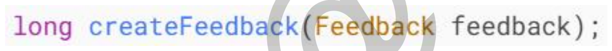
```
List<Feedback> getAllFeedbacks();
```

Figure 4.38 For property staff use



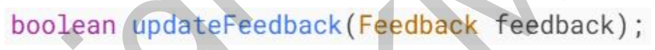
```
Feedback getFeedbackById(Long id);
```

Figure 4.39 Details page usage



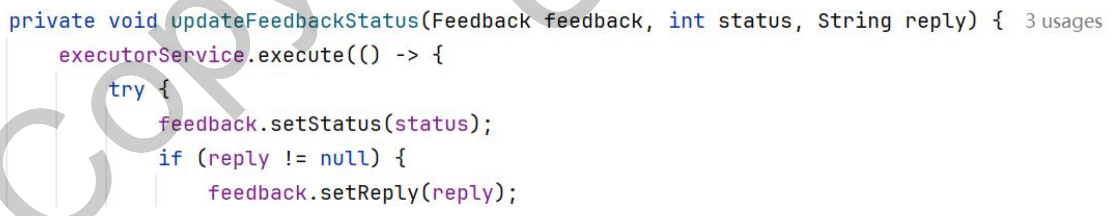
```
long createFeedback(Feedback feedback);
```

Figure 4.40 Submit page using



```
boolean updateFeedback(Feedback feedback);
```

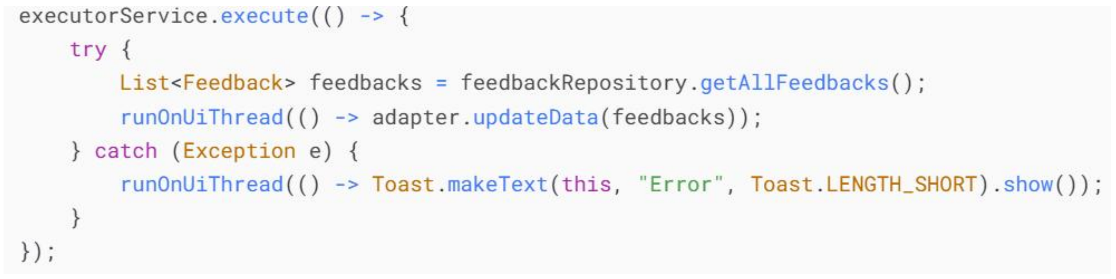
Figure 4.41 Property staff end status update



```
private void updateFeedbackStatus(Feedback feedback, int status, String reply) {  3 usages
    executorService.execute(() -> {
        try {
            feedback.setStatus(status);
            if (reply != null) {
                feedback.setReply(reply);
```

Figure 4.42 Status update example (Admin side)

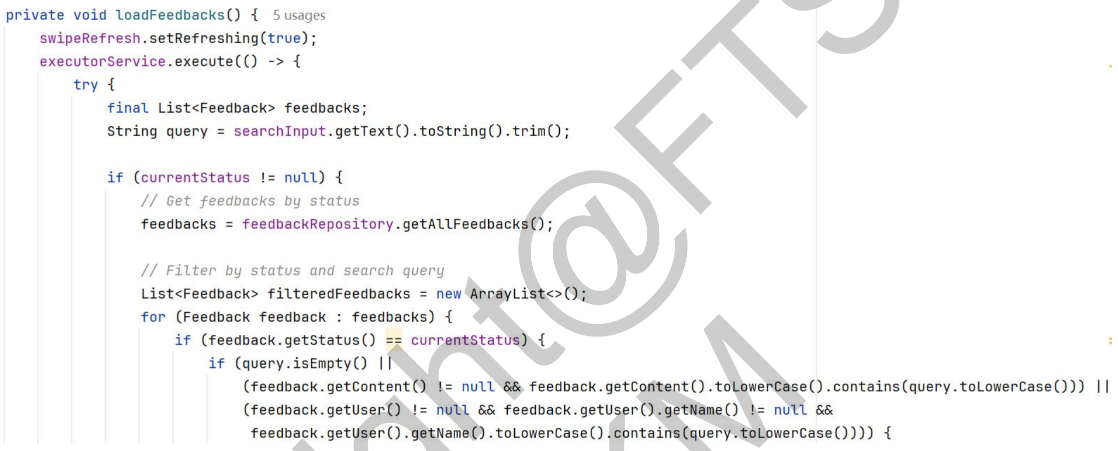
Use ExecutorService to manage background threads. Database operations are performed on non-UI threads. UI updates are safely switched via runOnUiThread.



```java
executorService.execute(() -> {
    try {
        List<Feedback> feedbacks = feedbackRepository.getAllFeedbacks();
        runOnUiThread(() -> adapter.updateData(feedbacks));
    } catch (Exception e) {
        runOnUiThread(() -> Toast.makeText(this, "Error", Toast.LENGTH_SHORT).show());
    }
});
```

Figure 4.43 Multithreading

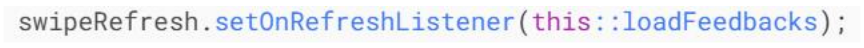
Real-time response to search input (TextWatcher monitoring). Combined condition filtering. Case-insensitive fuzzy matching.



```java
private void loadFeedbacks() {  5 usages
    swipeRefresh.setRefreshing(true);
    executorService.execute(() -> {
        try {
            final List<Feedback> feedbacks;
            String query = searchInput.getText().toString().trim();

            if (currentStatus != null) {
                // Get feedbacks by status
                feedbacks = feedbackRepository.getAllFeedbacks();

                // Filter by status and search query
                List<Feedback> filteredFeedbacks = new ArrayList<>();
                for (Feedback feedback : feedbacks) {
                    if (feedback.getStatus() == currentStatus) {
                        if (query.isEmpty() ||
                            (feedback.getContent() != null && feedback.getContent().toLowerCase().contains(query.toLowerCase())) ||
                            (feedback.getUser() != null && feedback.getUser().getName() != null &&
                             feedback.getUser().getName().toLowerCase().contains(query.toLowerCase()))) {
```

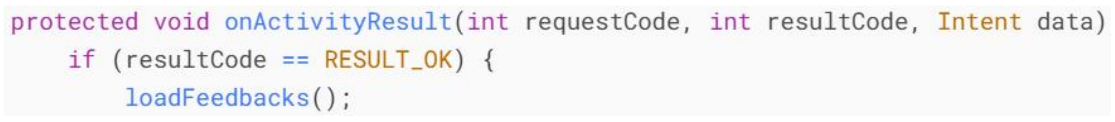
Figure 4.44 Property staff end screening search

Gesture refresh (SwipeRefreshLayout). Intelligent update after operation (avoid manual refresh).



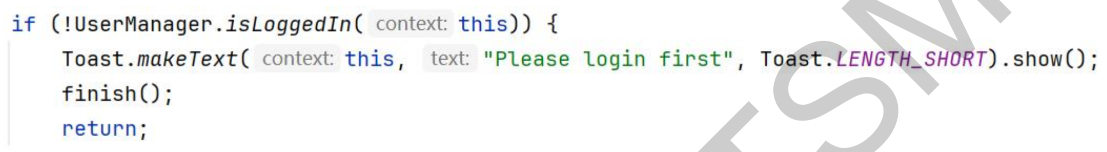
```
swipeRefresh.setOnRefreshListener(this::loadFeedbacks);
```

Figure 4.45 Data refresh on the owner side



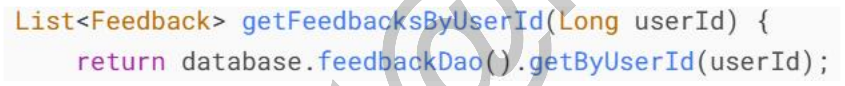
```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
    if (resultCode == RESULT_OK) {
        loadFeedbacks();
```

Figure 4.46 Refresh when returning from a child Activity



```
if (!UserManager.isLoggedIn( context: this)) {
    Toast.makeText( context: this,  text: "Please login first", Toast.LENGTH_SHORT).show();
    finish();
    return;
```

Figure 4.47 Check login status before critical operations



```
List<Feedback> getFeedbacksByUserId(Long userId) {
    return database.feedbackDao().getByUserId(userId);
```

Figure 4.48 Data isolation: Users can only query their own feedback

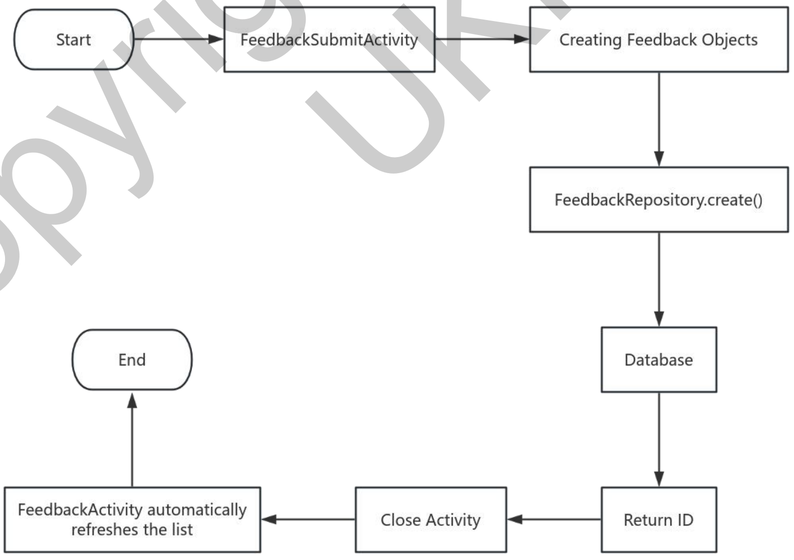
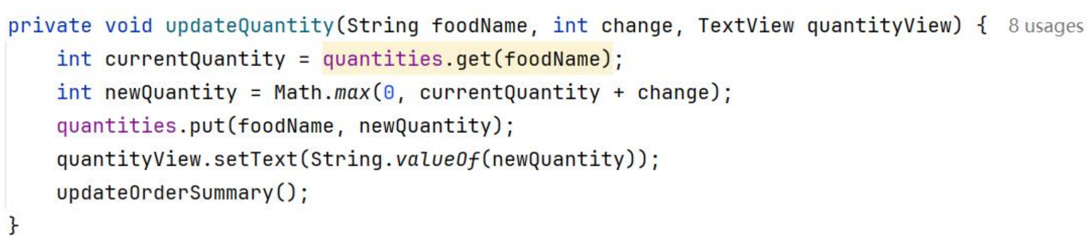

Start
FeedbackSubmitActivity
Creating Feedback Objects
FeedbackRepository.create()
Database
Return ID
Close Activity
FeedbackActivity automatically refreshes the list
End

Figure 4.49 Property staff processing procedures

e. Food Delivery

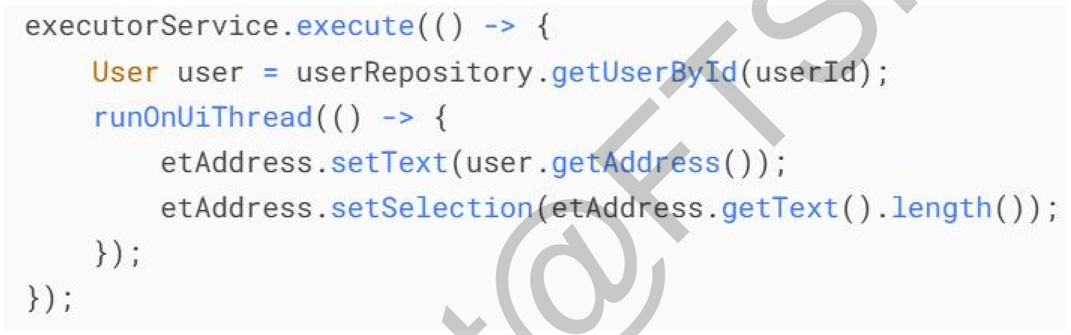
Use HashMap to store the quantity of each food. Make sure the quantity is not less than 0. Automatically update the order summary after each modification.



```
private void updateQuantity(String foodName, int change, TextView quantityView) {  8 usages
    int currentQuantity = quantities.get(foodName);
    int newQuantity = Math.max(0, currentQuantity + change);
    quantities.put(foodName, newQuantity);
    quantityView.setText(String.valueOf(newQuantity));
    updateOrderSummary();
}
```

Figure 4.50 Order management logic

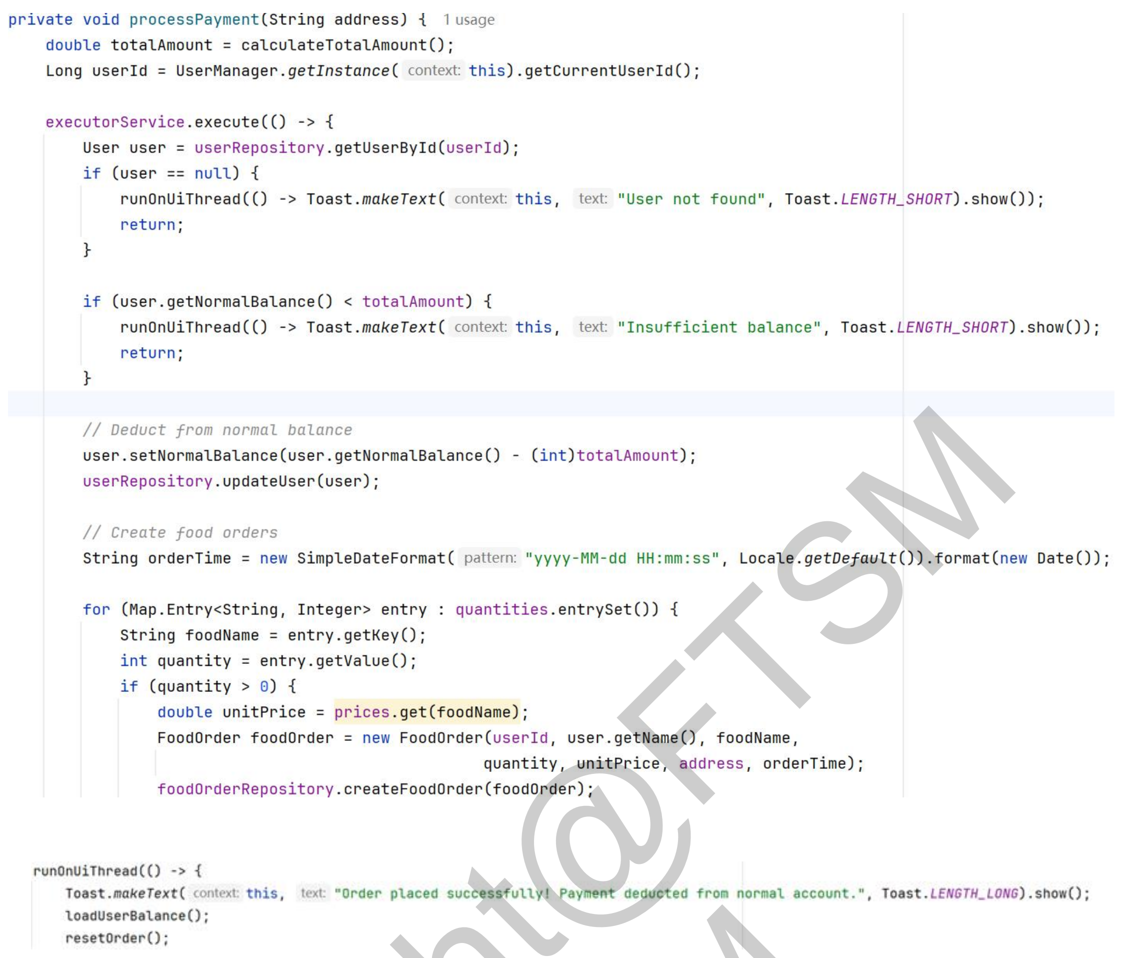
When the owner places an order, the address of the account the owner is using is pre-filled.



```
executorService.execute(() -> {
    User user = userRepository.getUserById(userId);
    runOnUiThread(() -> {
        etAddress.setText(user.getAddress());
        etAddress.setSelection(etAddress.getText().length());
    });
});
```

Figure 4.51 Directly fill in the owner's account registration address

Balance check. Account debit Order record creation. Reset order status after success.

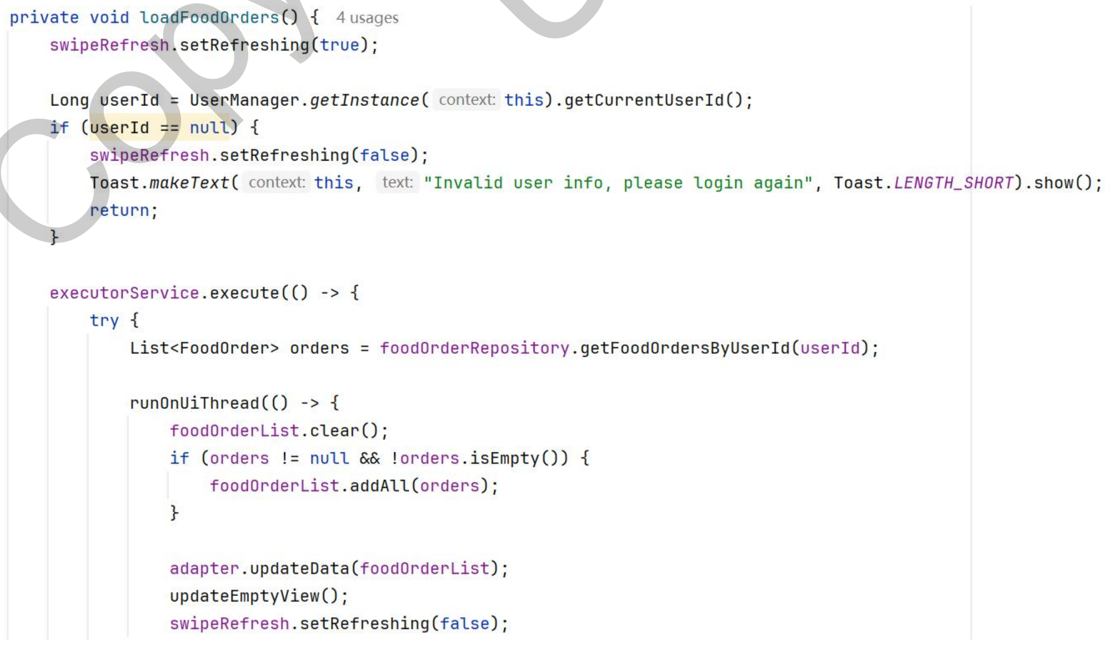


```
private void processPayment(String address) {  1 usage
    double totalAmount = calculateTotalAmount();
    Long userId = UserManager.getInstance( context: this).getCurrentUserId();

    executorService.execute(() -> {
        User user = userRepository.getUserById(userId);
        if (user == null) {
            runOnUiThread(() -> Toast.makeText( context: this,  text: "User not found", Toast.LENGTH_SHORT).show());
            return;
        }

        if (user.getNormalBalance() < totalAmount) {
            runOnUiThread(() -> Toast.makeText( context: this,  text: "Insufficient balance", Toast.LENGTH_SHORT).show());
            return;
        }

        // Deduct from normal balance
        user.setNormalBalance(user.getNormalBalance() - (int)totalAmount);
        userRepository.updateUser(user);

        // Create food orders
        String orderTime = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(new Date());

        for (Map.Entry<String, Integer> entry : quantities.entrySet()) {
            String foodName = entry.getKey();
            int quantity = entry.getValue();
            if (quantity > 0) {
                double unitPrice = prices.get(foodName);
                FoodOrder foodOrder = new FoodOrder(userId, user.getName(), foodName,
                                                    quantity, unitPrice, address, orderTime);
                foodOrderRepository.createFoodOrder(foodOrder);
```

```
runOnUiThread(() -> {
    Toast.makeText( context: this,  text: "Order placed successfully! Payment deducted from normal account.", Toast.LENGTH_LONG).show();
    loadUserBalance();
    resetOrder();
```

Figure 4.52 Payment processing

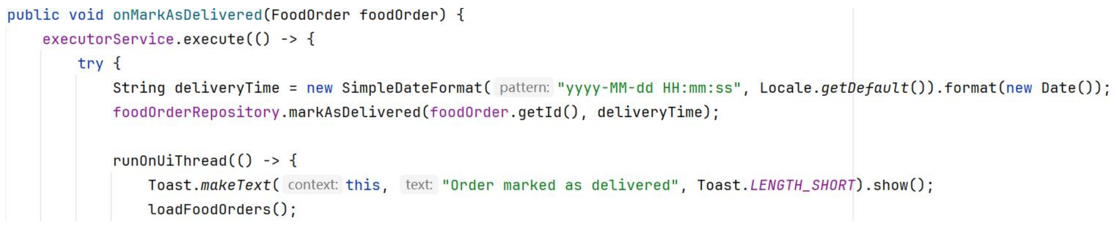
Display loading status. Background thread queries order data. Update the UI display. Handle empty state.



```
private void loadFoodOrders() {  4 usages
    swipeRefresh.setRefreshing(true);

    Long userId = UserManager.getInstance( context: this).getCurrentUserId();
    if (userId == null) {
        swipeRefresh.setRefreshing(false);
        Toast.makeText( context: this,  text: "Invalid user info, please login again", Toast.LENGTH_SHORT).show();
        return;
    }

    executorService.execute(() -> {
        try {
            List<FoodOrder> orders = foodOrderRepository.getFoodOrdersByUserId(userId);

            runOnUiThread(() -> {
                foodOrderList.clear();
                if (orders != null && !orders.isEmpty()) {
                    foodOrderList.addAll(orders);
                }

                adapter.updateData(foodOrderList);
                updateEmptyView();
                swipeRefresh.setRefreshing(false);
```

Figure 4.53 Order loading logic

Record the actual delivery time. Automatically refresh the list to show the latest status.

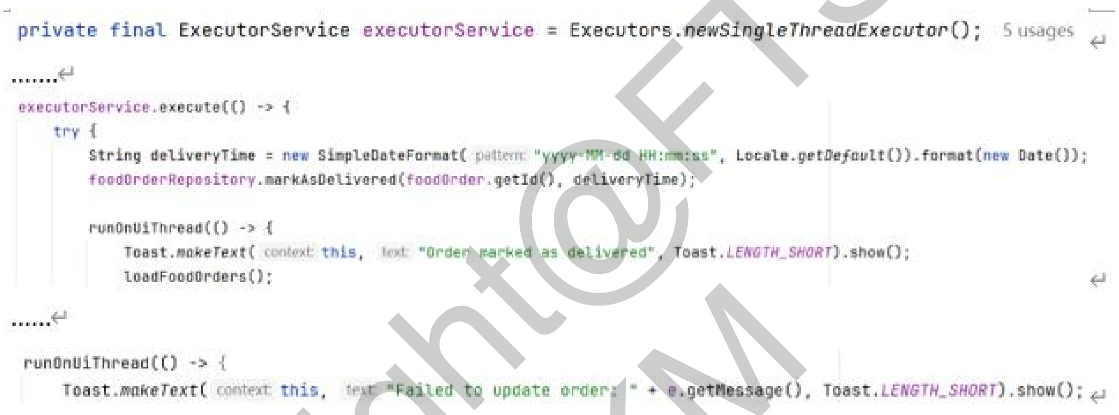


```java
public void onMarkAsDelivered(FoodOrder foodOrder) {
    executorService.execute(() -> {
        try {
            String deliveryTime = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(new Date());
            foodOrderRepository.markAsDelivered(foodOrder.getId(), deliveryTime);

            runOnUiThread(() -> {
                Toast.makeText( context: this,  text: "Order marked as delivered", Toast.LENGTH_SHORT).show();
                loadFoodOrders();
```

Figure 4.54	Order status update

## f.	Core technology implementation

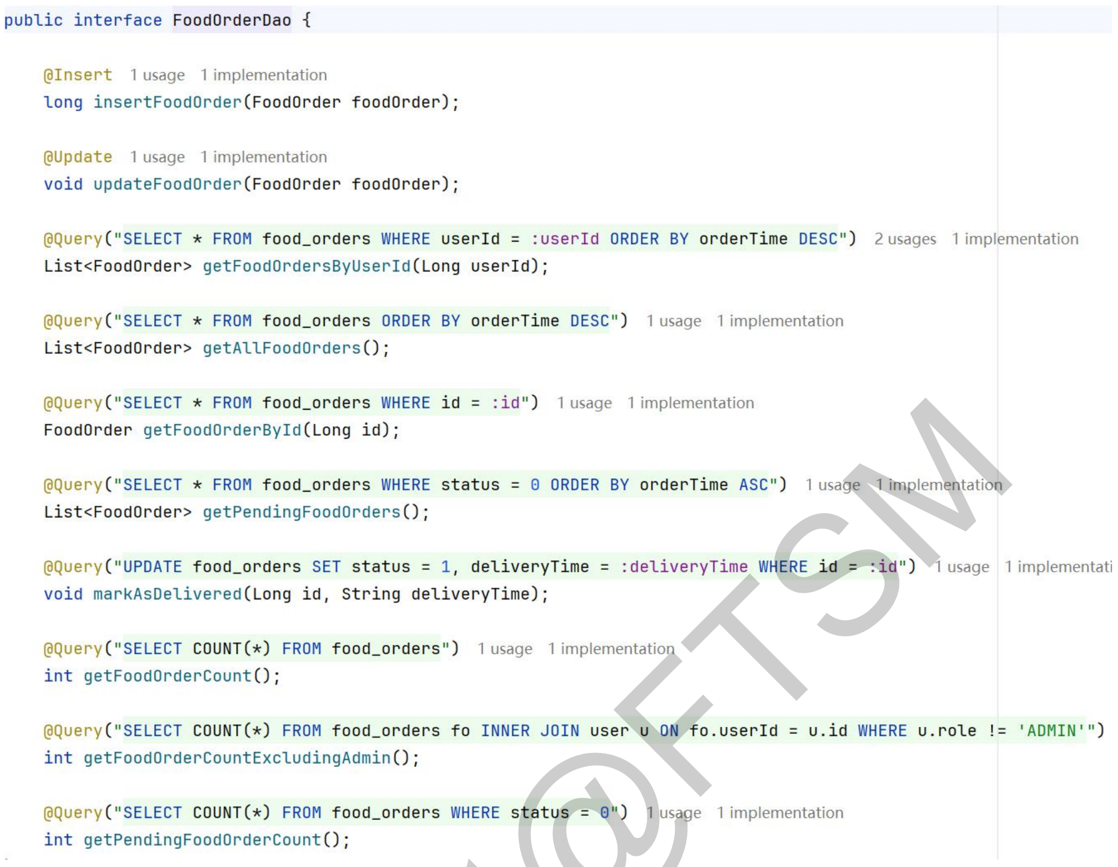
Avoid main thread blocking. Single-thread sequential execution ensures data consistency. Uniform lifecycle management (closed during onDestroy).



```java
private final ExecutorService executorService = Executors.newSingleThreadExecutor();  5 usages
.......
executorService.execute(() -> {
    try {
        String deliveryTime = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(new Date());
        foodOrderRepository.markAsDelivered(foodOrder.getId(), deliveryTime);

        runOnUiThread(() -> {
            Toast.makeText( context: this,  text: "Order marked as delivered", Toast.LENGTH_SHORT).show();
            loadFoodOrders();
......
runOnUiThread(() -> {
    Toast.makeText( context: this,  text: "Failed to update order: " + e.getMessage(), Toast.LENGTH_SHORT).show();
```

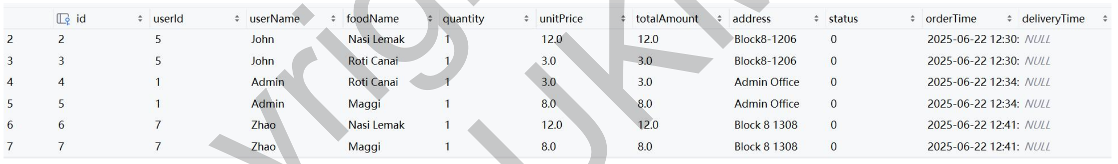
Figure 4.55	Multi-thread management

Record complete order information. Including order time and delivery time. Associated user ID.



```
public interface FoodOrderDao {

    @Insert  1 usage  1 implementation
    long insertFoodOrder(FoodOrder foodOrder);

    @Update  1 usage  1 implementation
    void updateFoodOrder(FoodOrder foodOrder);

    @Query("SELECT * FROM food_orders WHERE userId = :userId ORDER BY orderTime DESC")  2 usages  1 implementation
    List<FoodOrder> getFoodOrdersByUserId(Long userId);

    @Query("SELECT * FROM food_orders ORDER BY orderTime DESC")  1 usage  1 implementation
    List<FoodOrder> getAllFoodOrders();

    @Query("SELECT * FROM food_orders WHERE id = :id")  1 usage  1 implementation
    FoodOrder getFoodOrderById(Long id);

    @Query("SELECT * FROM food_orders WHERE status = 0 ORDER BY orderTime ASC")  1 usage  1 implementation
    List<FoodOrder> getPendingFoodOrders();

    @Query("UPDATE food_orders SET status = 1, deliveryTime = :deliveryTime WHERE id = :id")  1 usage  1 implementati
    void markAsDelivered(Long id, String deliveryTime);

    @Query("SELECT COUNT(*) FROM food_orders")  1 usage  1 implementation
    int getFoodOrderCount();

    @Query("SELECT COUNT(*) FROM food_orders fo INNER JOIN user u ON fo.userId = u.id WHERE u.role != 'ADMIN'")
    int getFoodOrderCountExcludingAdmin();

    @Query("SELECT COUNT(*) FROM food_orders WHERE status = 0")  1 usage  1 implementation
    int getPendingFoodOrderCount();
```

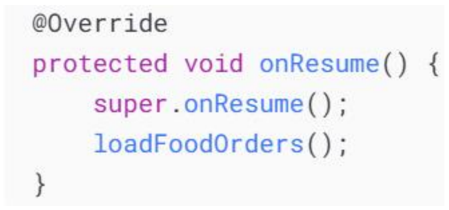
Figure 4.56 Database design



	id	userId	userName	foodName	quantity	unitPrice	totalAmount	address	status	orderTime	deliveryTime
2	2	5	John	Nasi Lemak	1	12.0	12.0	Block8-1206	0	2025-06-22 12:30:	NULL
3	3	5	John	Roti Canai	1	3.0	3.0	Block8-1206	0	2025-06-22 12:30:	NULL
4	4	1	Admin	Roti Canai	1	3.0	3.0	Admin Office	0	2025-06-22 12:34:	NULL
5	5	1	Admin	Maggi	1	8.0	8.0	Admin Office	0	2025-06-22 12:34:	NULL
6	6	7	Zhao	Nasi Lemak	1	12.0	12.0	Block 8 1308	0	2025-06-22 12:41:	NULL
7	7	7	Zhao	Maggi	1	8.0	8.0	Block 8 1308	0	2025-06-22 12:41:	NULL

Figure 4.57 Database

Automatically updates when returning from other screens. Ensures the latest data is displayed.

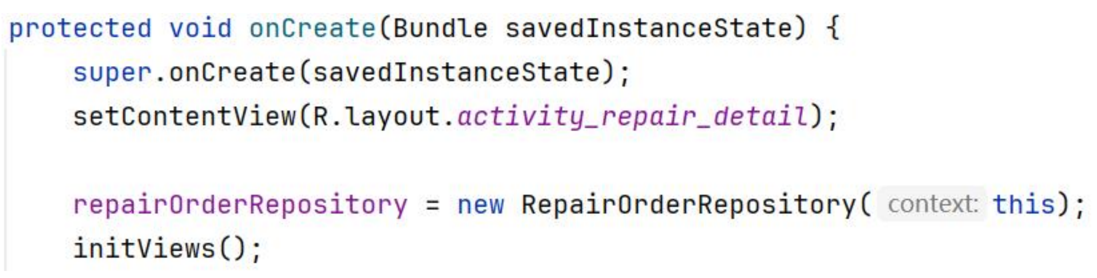


```
@Override
protected void onResume() {
    super.onResume();
    loadFoodOrders();
}
```

Figure 4.58 Real-time data synchronization

g. Repair Records

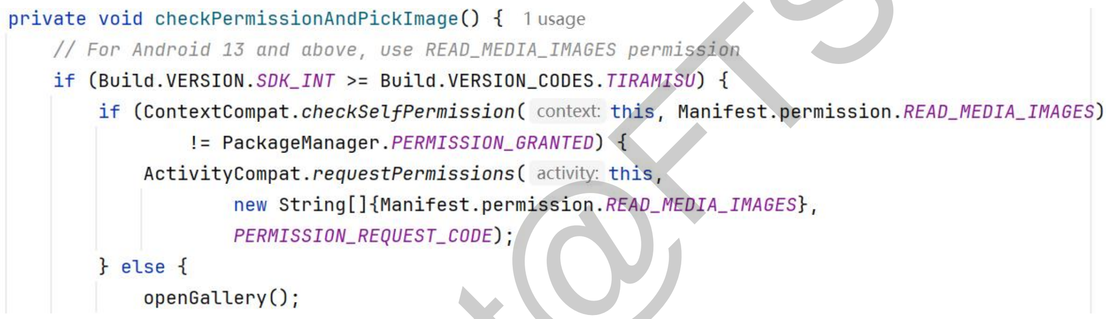
Activity initializes the database and sets the view. initViews() sets the click events of each UI component.



```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_repair_detail);

    repairOrderRepository = new RepairOrderRepository( context: this);
    initViews();
```

Figure 4.59 Picture settings

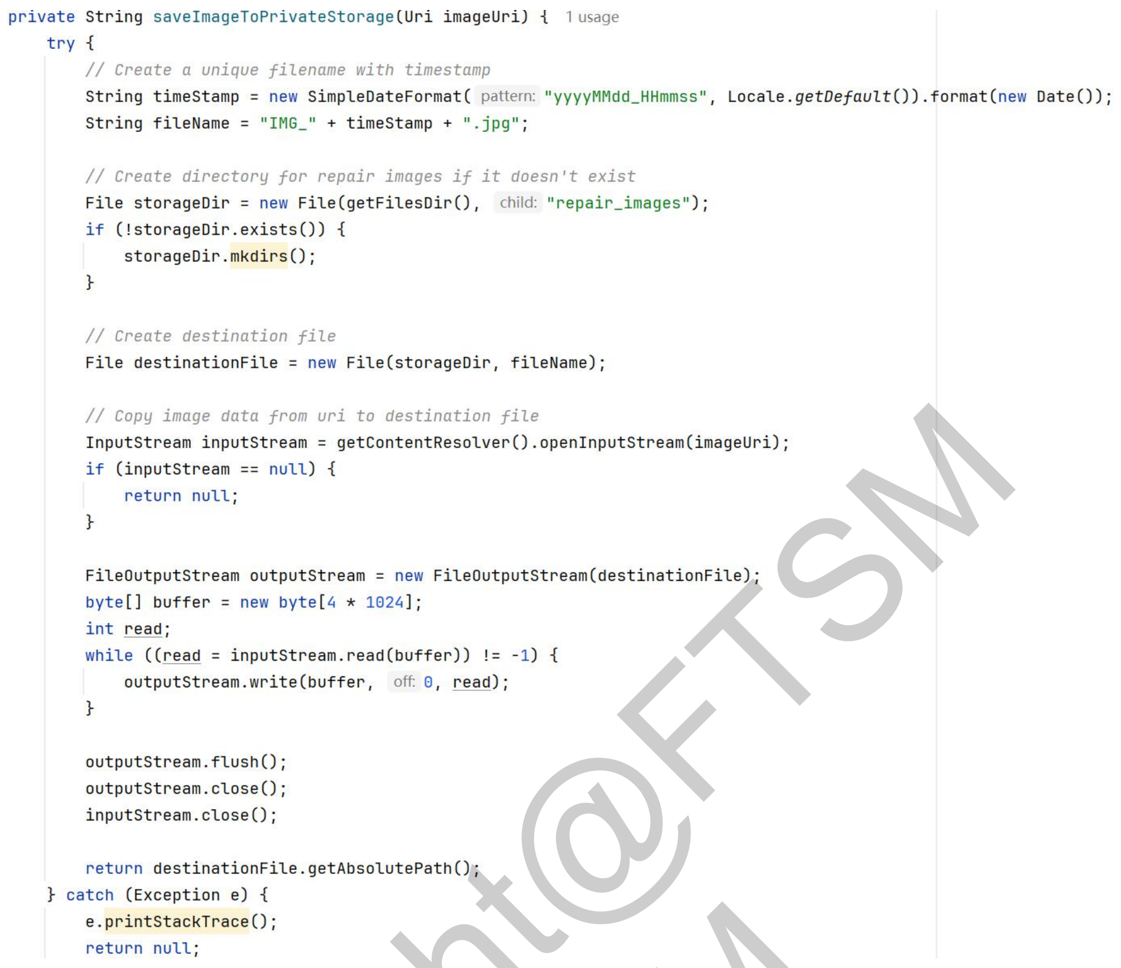
It can handle storage permission requests of different Android versions. After obtaining the permission, call openGallery() to open the album and select pictures.



```
private void checkPermissionAndPickImage() {  1 usage
    // For Android 13 and above, use READ_MEDIA_IMAGES permission
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        if (ContextCompat.checkSelfPermission( context: this, Manifest.permission.READ_MEDIA_IMAGES)
                != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions( activity: this,
                    new String[]{Manifest.permission.READ_MEDIA_IMAGES},
                    PERMISSION_REQUEST_CODE);
        } else {
            openGallery();
```

Figure 4.60 Select picture

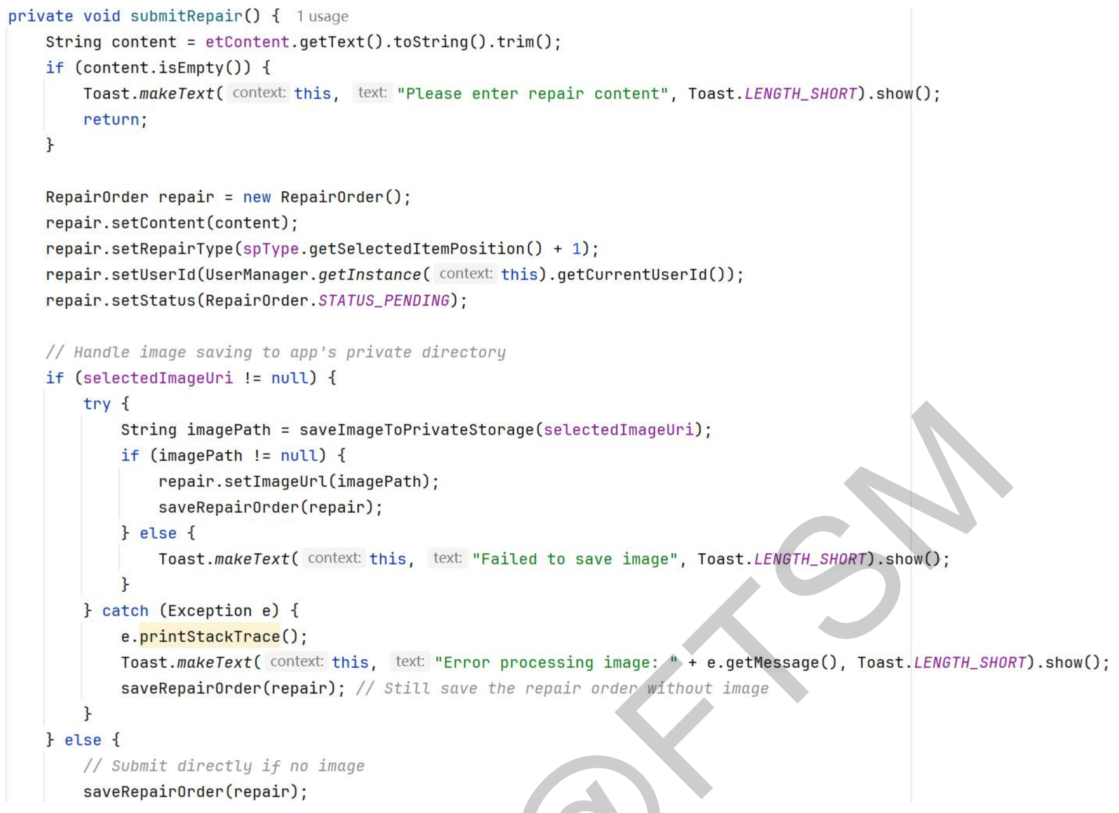
Save the selected images to the Smart Neighborhood's private directory (/data/data/[package]/files/repair_images/), using timestamps to generate unique file names to ensure no conflicts.



```
private String saveImageToPrivateStorage(Uri imageUri) {  1 usage
    try {
        // Create a unique filename with timestamp
        String timeStamp = new SimpleDateFormat( pattern: "yyyyMMdd_HHmmss", Locale.getDefault()).format(new Date());
        String fileName = "IMG_" + timeStamp + ".jpg";

        // Create directory for repair images if it doesn't exist
        File storageDir = new File(getFilesDir(),  child: "repair_images");
        if (!storageDir.exists()) {
            storageDir.mkdirs();
        }

        // Create destination file
        File destinationFile = new File(storageDir, fileName);

        // Copy image data from uri to destination file
        InputStream inputStream = getContentResolver().openInputStream(imageUri);
        if (inputStream == null) {
            return null;
        }

        FileOutputStream outputStream = new FileOutputStream(destinationFile);
        byte[] buffer = new byte[4 * 1024];
        int read;
        while ((read = inputStream.read(buffer)) != -1) {
            outputStream.write(buffer,  off: 0, read);
        }

        outputStream.flush();
        outputStream.close();
        inputStream.close();

        return destinationFile.getAbsolutePath();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
```

Figure 4.61	Save the selected image in Repair Records

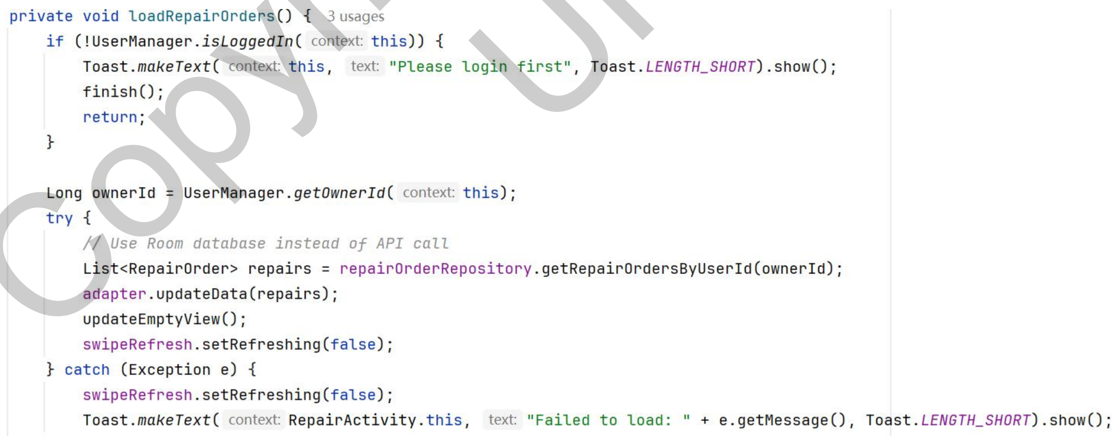
Validate input content. Create RepairOrder object and set properties. Handle image saving. Save to database via RepairOrderRepository.



```
private void submitRepair() {  1 usage
    String content = etContent.getText().toString().trim();
    if (content.isEmpty()) {
        Toast.makeText( context: this,  text: "Please enter repair content", Toast.LENGTH_SHORT).show();
        return;
    }

    RepairOrder repair = new RepairOrder();
    repair.setContent(content);
    repair.setRepairType(spType.getSelectedItemPosition() + 1);
    repair.setUserId(UserManager.getInstance( context: this).getCurrentUserId());
    repair.setStatus(RepairOrder.STATUS_PENDING);

    // Handle image saving to app's private directory
    if (selectedImageUri != null) {
        try {
            String imagePath = saveImageToPrivateStorage(selectedImageUri);
            if (imagePath != null) {
                repair.setImageUrl(imagePath);
                saveRepairOrder(repair);
            } else {
                Toast.makeText( context: this,  text: "Failed to save image", Toast.LENGTH_SHORT).show();
            }
        } catch (Exception e) {
            e.printStackTrace();
            Toast.makeText( context: this,  text: "Error processing image: " + e.getMessage(), Toast.LENGTH_SHORT).show();
            saveRepairOrder(repair); // Still save the repair order without image
        }
    } else {
        // Submit directly if no image
        saveRepairOrder(repair);
```

Figure 4.62 Work order submission logic

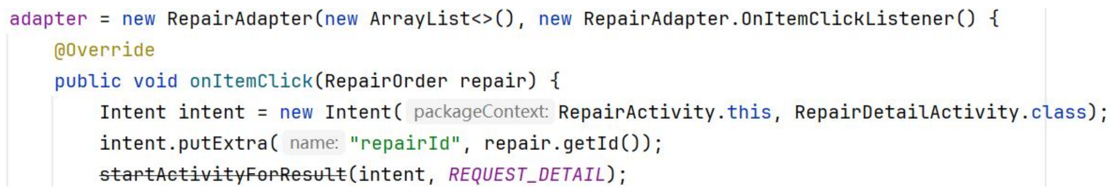
Load the current user's list of repair orders from the database and update the RecyclerView adapter.



```
private void loadRepairOrders() {  3 usages
    if (!UserManager.isLoggedIn( context: this)) {
        Toast.makeText( context: this,  text: "Please login first", Toast.LENGTH_SHORT).show();
        finish();
        return;
    }

    Long ownerId = UserManager.getOwnerId( context: this);
    try {
        // Use Room database instead of API call
        List<RepairOrder> repairs = repairOrderRepository.getRepairOrdersByUserId(ownerId);
        adapter.updateData(repairs);
        updateEmptyView();
        swipeRefresh.setRefreshing(false);
    } catch (Exception e) {
        swipeRefresh.setRefreshing(false);
        Toast.makeText( context: RepairActivity.this,  text: "Failed to load: " + e.getMessage(), Toast.LENGTH_SHORT).show();
```

Figure 4.63 Data loading

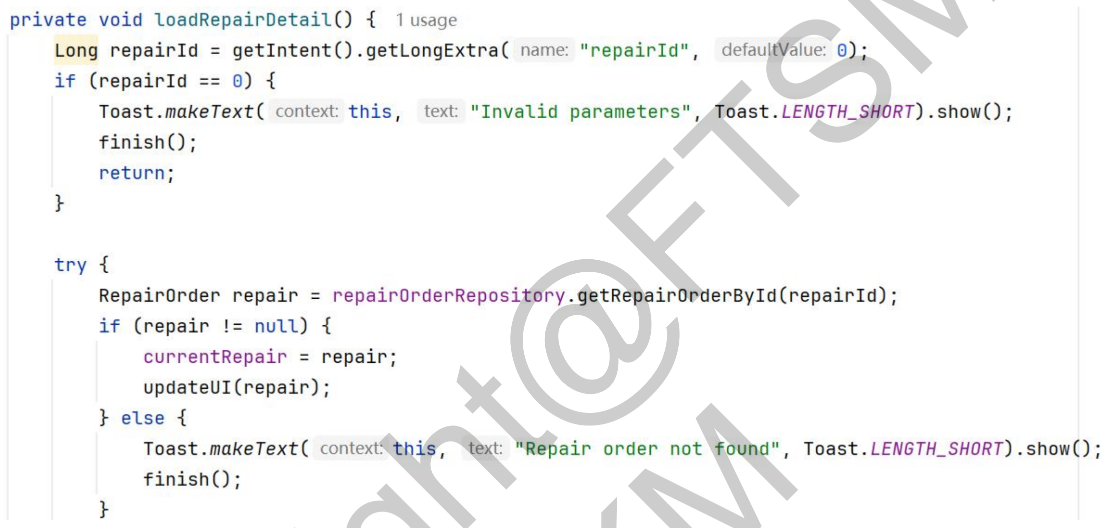
The adapter sets a click event. Clicking a work order item will jump to the details page and pass the work order ID.



```
adapter = new RepairAdapter(new ArrayList<>(), new RepairAdapter.OnItemClickListener() {
    @Override
    public void onItemClick(RepairOrder repair) {
        Intent intent = new Intent( packageContext: RepairActivity.this, RepairDetailActivity.class);
        intent.putExtra( name: "repairId", repair.getId());
        ~~startActivityForResult~~(intent, REQUEST_DETAIL);
```

Figure 4.64 List adapter settings

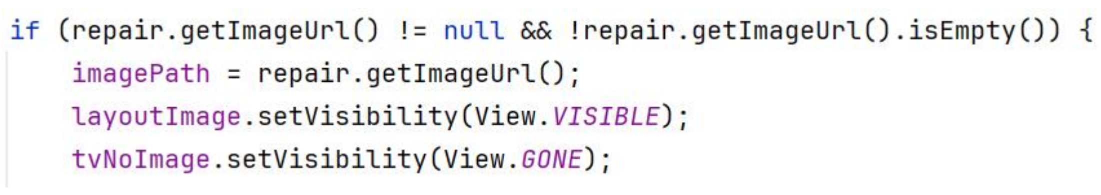
Load specific work order information from the database through the incoming repairId.



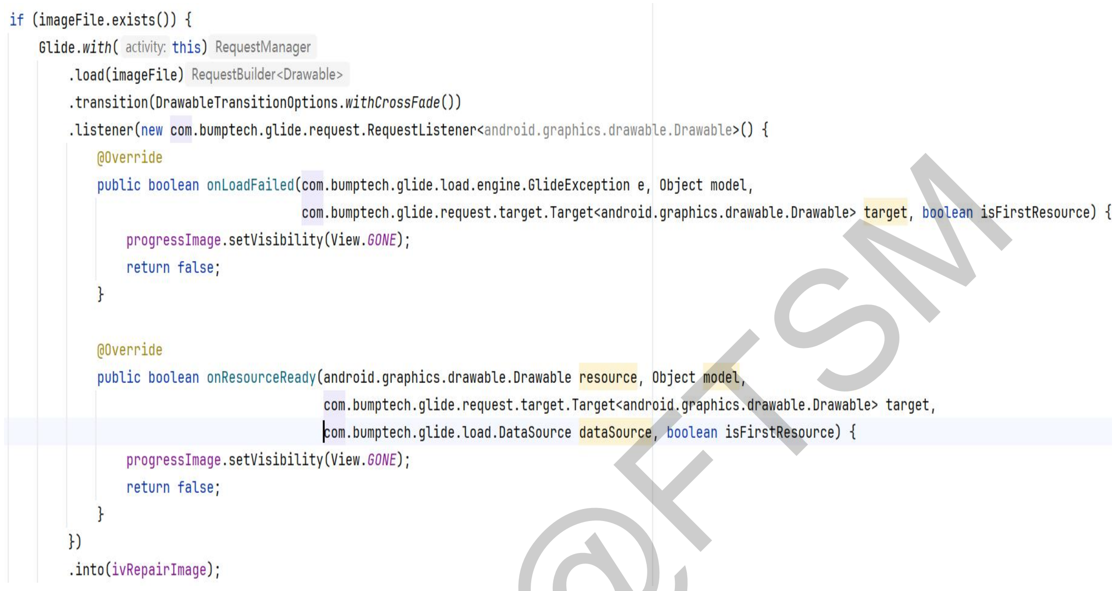
```
private void loadRepairDetail() {  1 usage
    Long repairId = getIntent().getLongExtra( name: "repairId",  defaultValue: 0);
    if (repairId == 0) {
        Toast.makeText( context: this,  text: "Invalid parameters", Toast.LENGTH_SHORT).show();
        finish();
        return;
    }

    try {
        RepairOrder repair = repairOrderRepository.getRepairOrderById(repairId);
        if (repair != null) {
            currentRepair = repair;
            updateUI(repair);
        } else {
            Toast.makeText( context: this,  text: "Repair order not found", Toast.LENGTH_SHORT).show();
            finish();
        }
```

Figure 4.65 Work order details loading

Use the Glide library to load the image previously saved in private storage and handle the loading status.



```
if (repair.getImageUrl() != null && !repair.getImageUrl().isEmpty()) {
    imagePath = repair.getImageUrl();
    layoutImage.setVisibility(View.VISIBLE);
    tvNoImage.setVisibility(View.GONE);
```



```
if (imageFile.exists()) {
    Glide.with( activity: this) RequestManager
        .load(imageFile) RequestBuilder<Drawable>
        .transition(DrawableTransitionOptions.withCrossFade())
        .listener(new com.bumptech.glide.request.RequestListener<android.graphics.drawable.Drawable>() {
            @Override
            public boolean onLoadFailed(com.bumptech.glide.load.engine.GlideException e, Object model,
                                        com.bumptech.glide.request.target.Target<android.graphics.drawable.Drawable> target, boolean isFirstResource) {
                progressImage.setVisibility(View.GONE);
                return false;
            }

            @Override
            public boolean onResourceReady(android.graphics.drawable.Drawable resource, Object model,
                                           com.bumptech.glide.request.target.Target<android.graphics.drawable.Drawable> target,
                                           com.bumptech.glide.load.DataSource dataSource, boolean isFirstResource) {
                progressImage.setVisibility(View.GONE);
                return false;
            }
        })
        .into(ivRepairImage);
```

Figure 4.66 Image loading processing

Update the work order status to "Cancelled" through the warehouse.

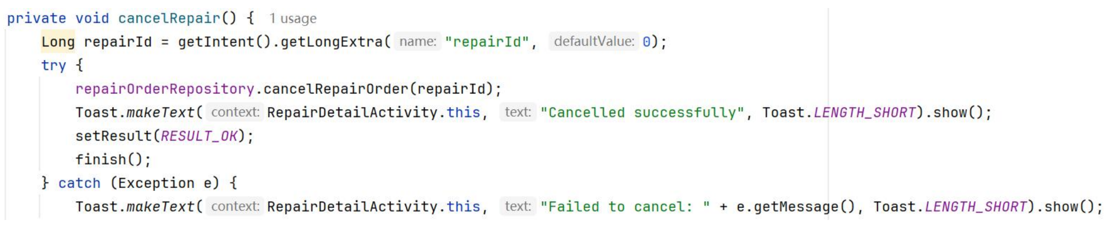

```
private void cancelRepair() {  1 usage
    Long repairId = getIntent().getLongExtra( name: "repairId",  defaultValue: 0);
    try {
        repairOrderRepository.cancelRepairOrder(repairId);
        Toast.makeText( context: RepairDetailActivity.this,  text: "Cancelled successfully", Toast.LENGTH_SHORT).show();
        setResult(RESULT_OK);
        finish();
    } catch (Exception e) {
        Toast.makeText( context: RepairDetailActivity.this,  text: "Failed to cancel: " + e.getMessage(), Toast.LENGTH_SHORT).show();
```

Figure 4.67	Work order cancellation function

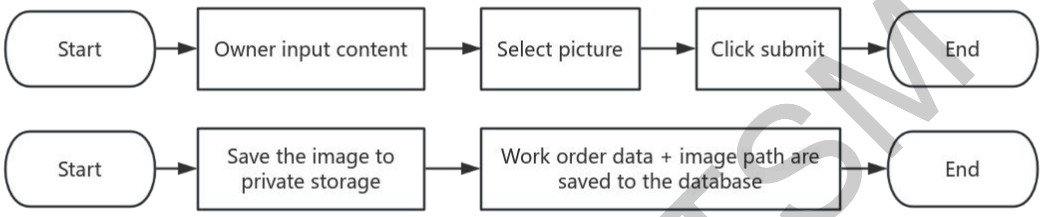

Start
Owner input content
Select picture
Click submit
End
Start
Save the image to private storage
Work order data + image path are saved to the database
End

Figure 4.68	Work order creation process

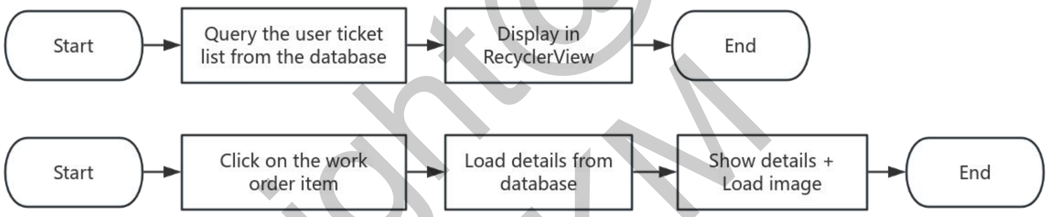

Start
Query the user ticket list from the database
Display in RecyclerView
End
Start
Click on the work order item
Load details from database
Show details + Load image
End

Figure 4.69	Work order display process

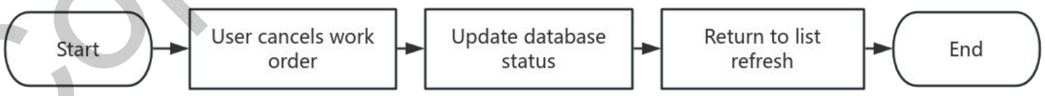
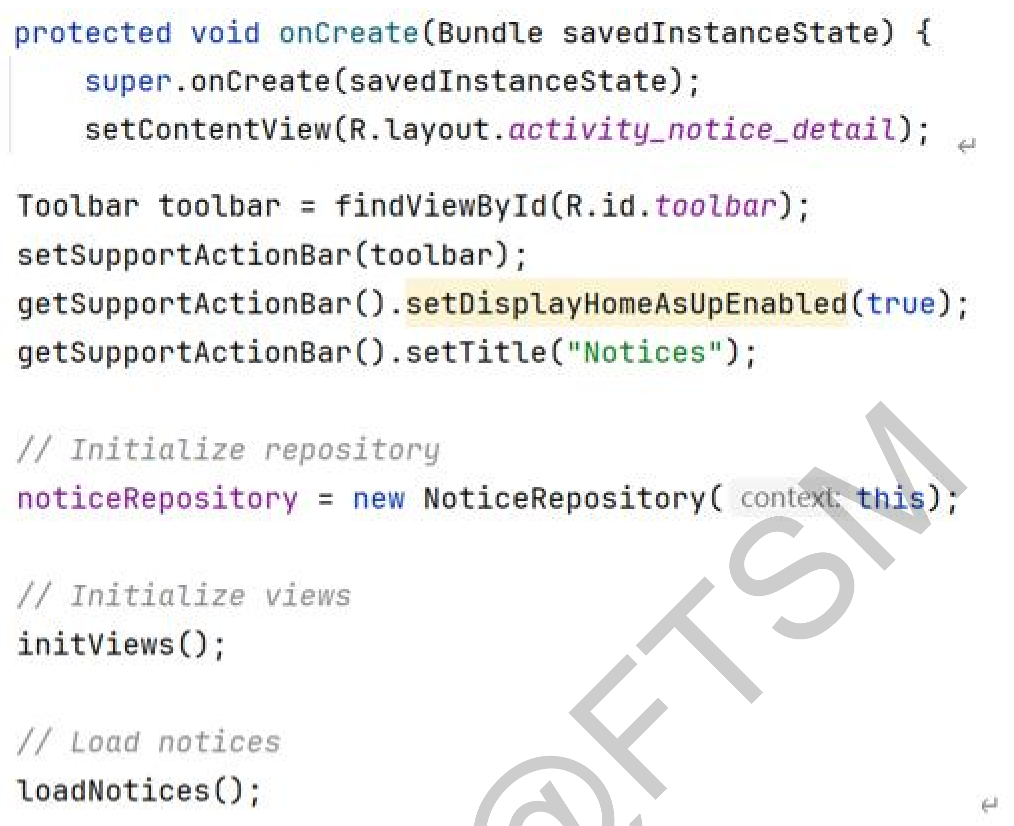


Figure 4.70	Status update process

### h.	Announcement

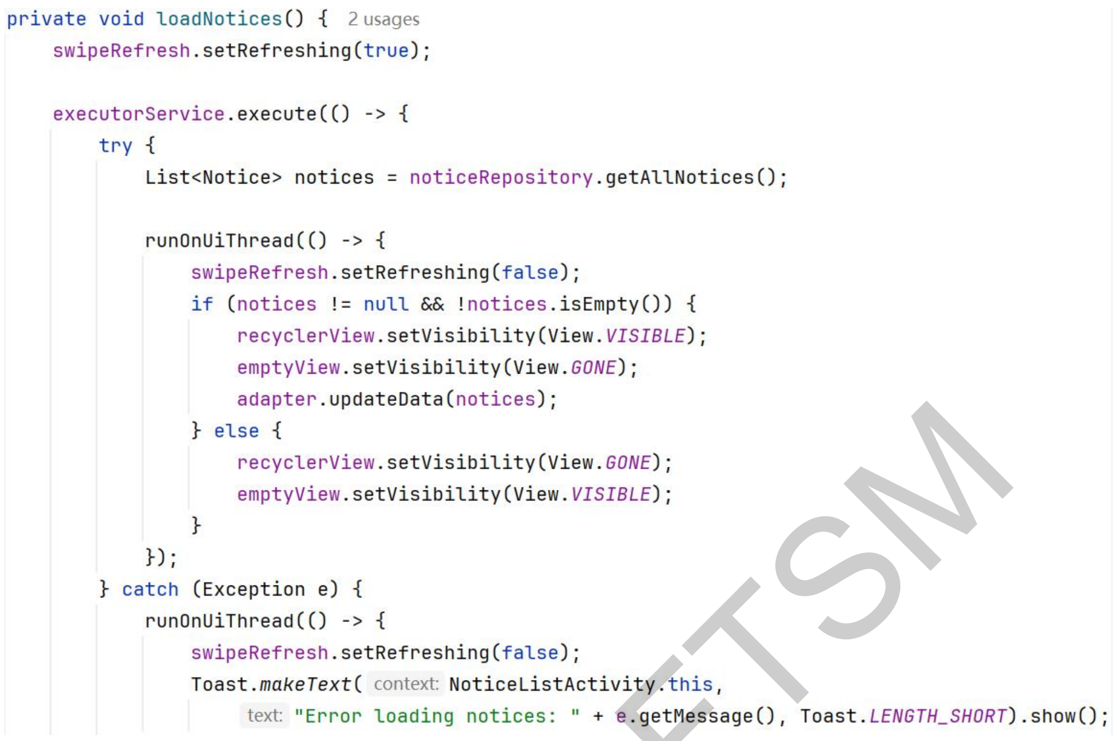
Set Toolbar and configure the back button. Initialize the data repository NoticeRepository. Initialize the view component. Load the notification data.



```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_notice_detail);

Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
getSupportActionBar().setTitle("Notices");

// Initialize repository
noticeRepository = new NoticeRepository( context: this);

// Initialize views
initViews();

// Load notices
loadNotices();
```

Figure 4.71 Initialization and view setup

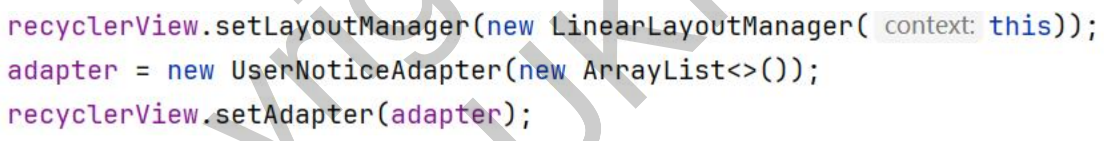
Use the single-thread thread pool created by Executors.newSingle ThreadExecutor() to perform database operations. Switch back to the main thread through runOnUiThread to update the UI. Perfect empty state processing (display emptyView). Pull-down refresh function integration.



```
private void loadNotices() {  2 usages
    swipeRefresh.setRefreshing(true);

    executorService.execute(() -> {
        try {
            List<Notice> notices = noticeRepository.getAllNotices();

            runOnUiThread(() -> {
                swipeRefresh.setRefreshing(false);
                if (notices != null && !notices.isEmpty()) {
                    recyclerView.setVisibility(View.VISIBLE);
                    emptyView.setVisibility(View.GONE);
                    adapter.updateData(notices);
                } else {
                    recyclerView.setVisibility(View.GONE);
                    emptyView.setVisibility(View.VISIBLE);
                }
            });
        } catch (Exception e) {
            runOnUiThread(() -> {
                swipeRefresh.setRefreshing(false);
                Toast.makeText( context: NoticeListActivity.this,
                     text: "Error loading notices: " + e.getMessage(), Toast.LENGTH_SHORT).show();
```

Figure 4.72 Asynchronous data loading mechanism

The list display is implemented using the standard RecyclerView and configured with a linear layout manager.

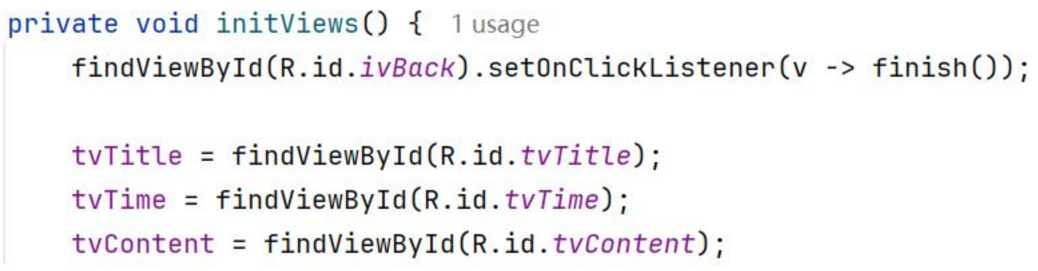


```
recyclerView.setLayoutManager(new LinearLayoutManager( context: this));
adapter = new UserNoticeAdapter(new ArrayList<>());
recyclerView.setAdapter(adapter);
```

Figure 4.73 List adapter configuration

The details page uses a simple view binding method, including:

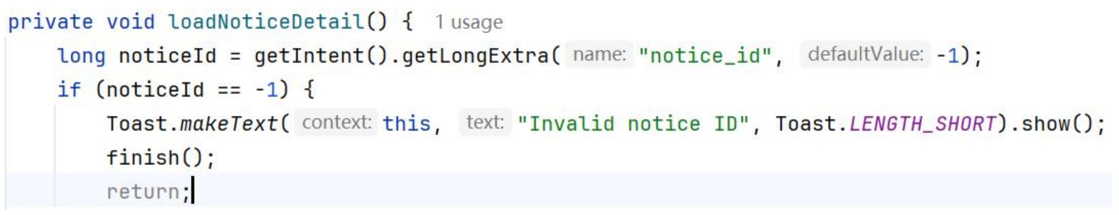
Back button, Title TextView, Time TextView Content TextView.



```
private void initViews() {  1 usage
    findViewById(R.id.ivBack).setOnClickListener(v -> finish());

    tvTitle = findViewById(R.id.tvTitle);
    tvTime = findViewById(R.id.tvTime);
    tvContent = findViewById(R.id.tvContent);
```

Figure 4.74 Basic structure and view initialization

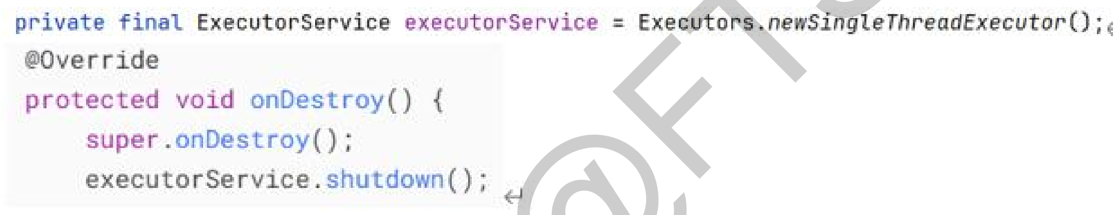
Get the specific notification content through the notice_id parameter passed by Intent, and get the details through Repository.



```
private void loadNoticeDetail() {  1 usage
    long noticeId = getIntent().getLongExtra( name: "notice_id",  defaultValue: -1);
    if (noticeId == -1) {
        Toast.makeText( context: this,  text: "Invalid notice ID", Toast.LENGTH_SHORT).show();
        finish();
        return;
```

Figure 4.75 Notification details loading logic

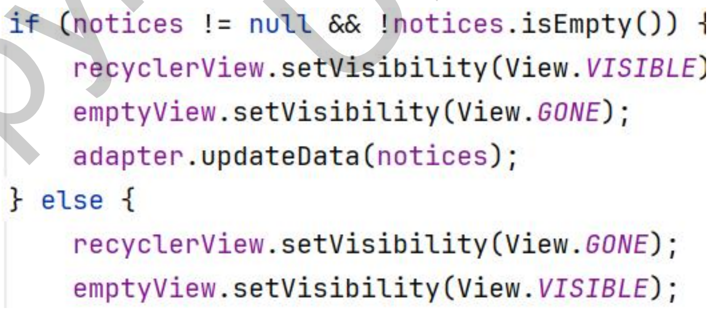
Advantages of using a single-threaded thread pool: Orderly processing of database operations to avoid concurrency issues. More flexible and controllable than using AsyncTask directly. Correctly close the thread pool when the Activity is destroyed to prevent memory leaks.



```
private final ExecutorService executorService = Executors.newSingleThreadExecutor();
@Override
protected void onDestroy() {
    super.onDestroy();
    executorService.shutdown();
```

Figure 4.76 Thread management strategy

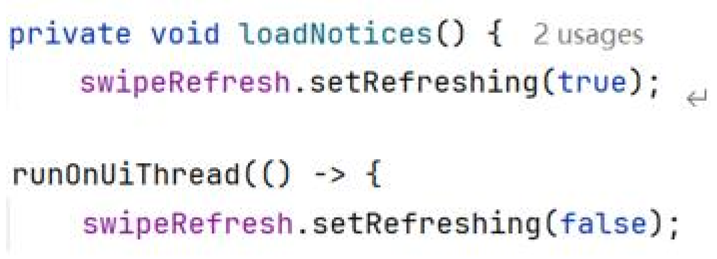
Good user experience design: Show the list when there is data. Show the empty state prompt view when there is no data. Avoid the confusion caused by displaying a blank list.



```
if (notices != null && !notices.isEmpty()) {
    recyclerView.setVisibility(View.VISIBLE)
    emptyView.setVisibility(View.GONE);
    adapter.updateData(notices);
} else {
    recyclerView.setVisibility(View.GONE);
    emptyView.setVisibility(View.VISIBLE);
```

Figure 4.77 Empty status handling

Integrate SwipeRefreshLayout to implement the standard pull-down refresh mode and provide a good user interaction experience.



```
private void loadNotices() {  2 usages
    swipeRefresh.setRefreshing(true);

runOnUiThread(() -> {
    swipeRefresh.setRefreshing(false);
```

Figure 4.78 Pull down to refresh

i. Data analysis

Get each entity DAO object through the Room database.

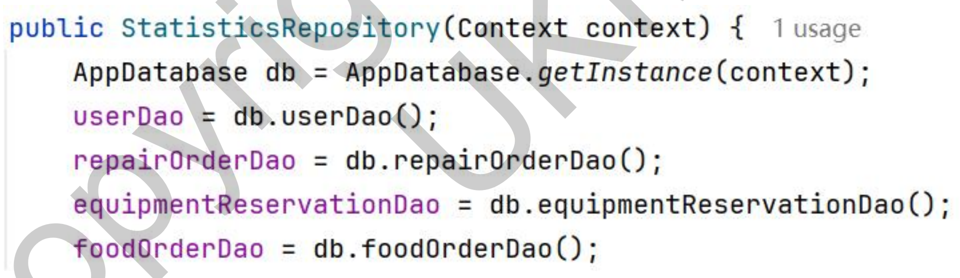
Contains four data access objects:

userDao: user data access.

repairOrderDao: repair work order data access.

equipmentReservationDao: equipment reservation data access.

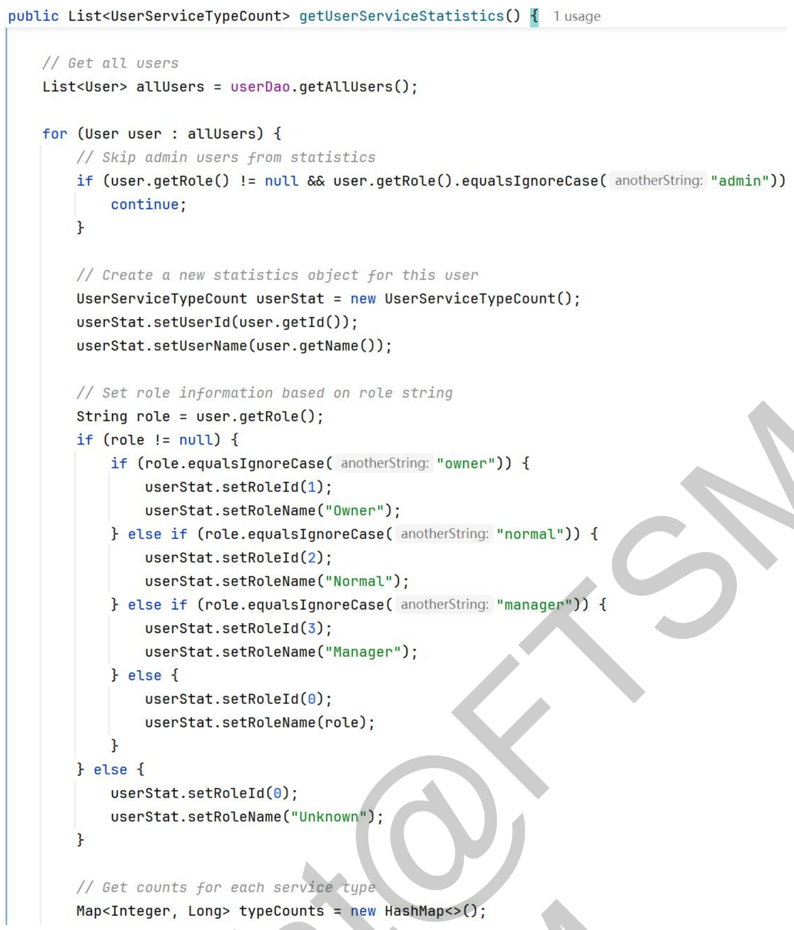
foodOrderDao: food order data access.



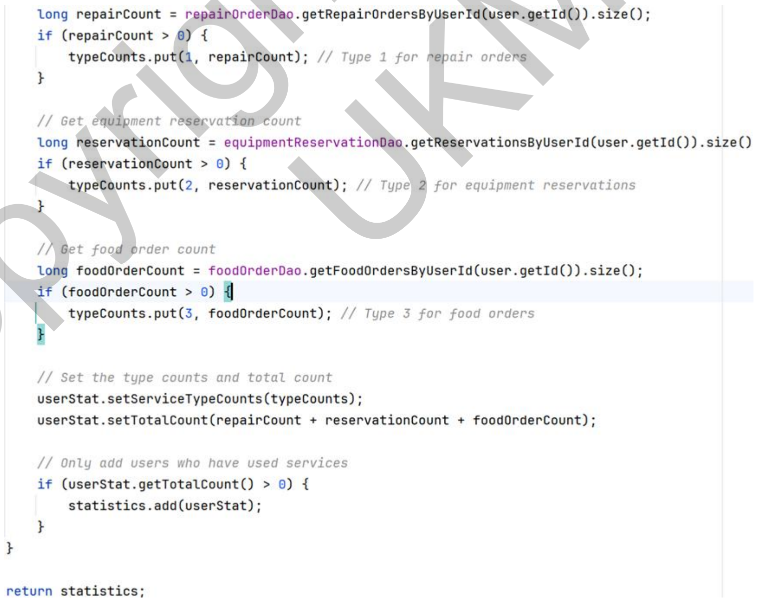
```
public StatisticsRepository(Context context) {  1 usage
    AppDatabase db = AppDatabase.getInstance(context);
    userDao = db.userDao();
    repairOrderDao = db.repairOrderDao();
    equipmentReservationDao = db.equipmentReservationDao();
    foodOrderDao = db.foodOrderDao();
```

Figure 4.79 Data access layer initialization

Traverse all non-administrator users. Count the usage of three services for each user: Maintenance service (type 1), Equipment reservation (type 2), Food order (type 3). Only include users who have used at least one service



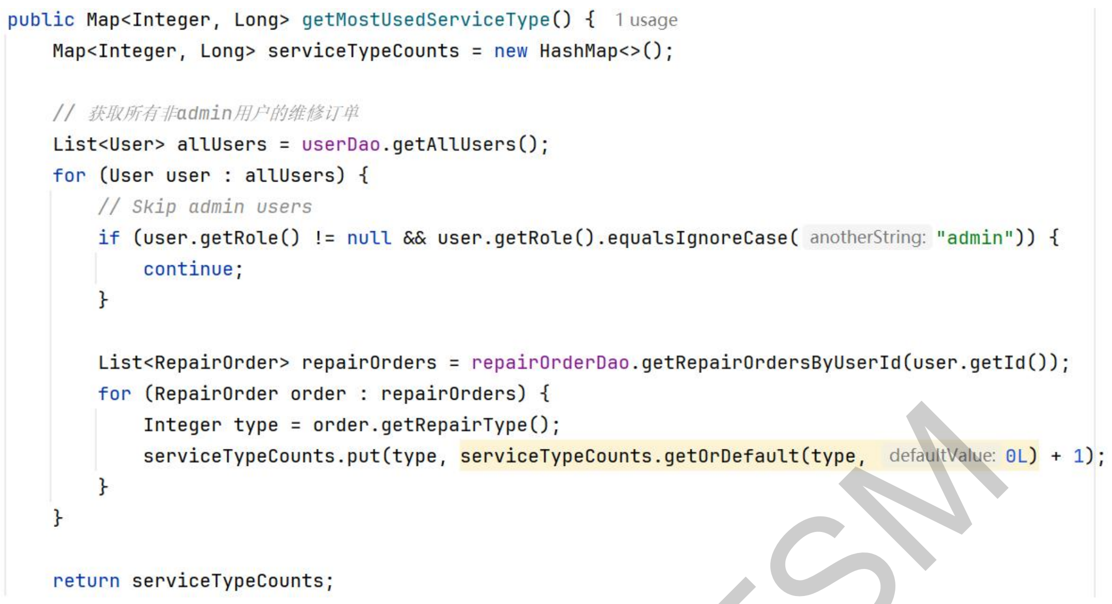
```
public List<UserServiceTypeCount> getUserServiceStatistics() {  1 usage

    // Get all users
    List<User> allUsers = userDao.getAllUsers();

    for (User user : allUsers) {
        // Skip admin users from statistics
        if (user.getRole() != null && user.getRole().equalsIgnoreCase( anotherString: "admin"))
            continue;
        }

        // Create a new statistics object for this user
        UserServiceTypeCount userStat = new UserServiceTypeCount();
        userStat.setUserId(user.getId());
        userStat.setUserName(user.getName());

        // Set role information based on role string
        String role = user.getRole();
        if (role != null) {
            if (role.equalsIgnoreCase( anotherString: "owner")) {
                userStat.setRoleId(1);
                userStat.setRoleName("Owner");
            } else if (role.equalsIgnoreCase( anotherString: "normal")) {
                userStat.setRoleId(2);
                userStat.setRoleName("Normal");
            } else if (role.equalsIgnoreCase( anotherString: "manager")) {
                userStat.setRoleId(3);
                userStat.setRoleName("Manager");
            } else {
                userStat.setRoleId(0);
                userStat.setRoleName(role);
            }
        } else {
            userStat.setRoleId(0);
            userStat.setRoleName("Unknown");
        }

        // Get counts for each service type
        Map<Integer, Long> typeCounts = new HashMap<>();
```



```
        long repairCount = repairOrderDao.getRepairOrdersByUserId(user.getId()).size();
        if (repairCount > 0) {
            typeCounts.put(1, repairCount); // Type 1 for repair orders
        }

        // Get equipment reservation count
        long reservationCount = equipmentReservationDao.getReservationsByUserId(user.getId()).size()
        if (reservationCount > 0) {
            typeCounts.put(2, reservationCount); // Type 2 for equipment reservations
        }

        // Get food order count
        long foodOrderCount = foodOrderDao.getFoodOrdersByUserId(user.getId()).size();
        if (foodOrderCount > 0) {
            typeCounts.put(3, foodOrderCount); // Type 3 for food orders
        }

        // Set the type counts and total count
        userStat.setServiceTypeCounts(typeCounts);
        userStat.setTotalCount(repairCount + reservationCount + foodOrderCount);

        // Only add users who have used services
        if (userStat.getTotalCount() > 0) {
            statistics.add(userStat);
        }
    }

    return statistics;
```

Figure 4.80	Owner service statistics implementation

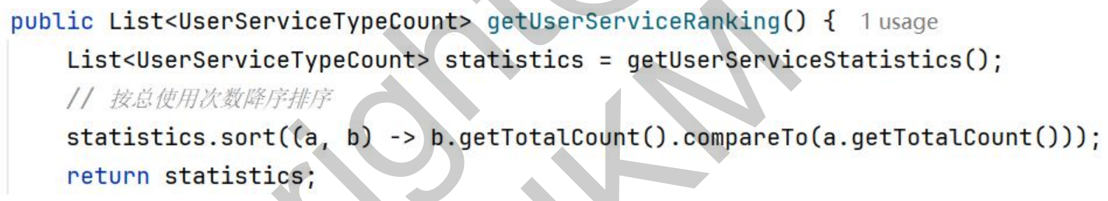
Count the total number of times each maintenance type is used. Use the getOrDefault method to simplify the counting logic. Return a Map structure: maintenance type ID → number of times used.



```
public Map<Integer, Long> getMostUsedServiceType() {  1 usage
    Map<Integer, Long> serviceTypeCounts = new HashMap<>();

    // 获取所有非admin用户的维修订单
    List<User> allUsers = userDao.getAllUsers();
    for (User user : allUsers) {
        // Skip admin users
        if (user.getRole() != null && user.getRole().equalsIgnoreCase( anotherString: "admin")) {
            continue;
        }

        List<RepairOrder> repairOrders = repairOrderDao.getRepairOrdersByUserId(user.getId());
        for (RepairOrder order : repairOrders) {
            Integer type = order.getRepairType();
            serviceTypeCounts.put(type, serviceTypeCounts.getOrDefault(type,  defaultValue: 0L) + 1);
        }
    }

    return serviceTypeCounts;
```

Figure 4.81	Frequency analysis of service type usage

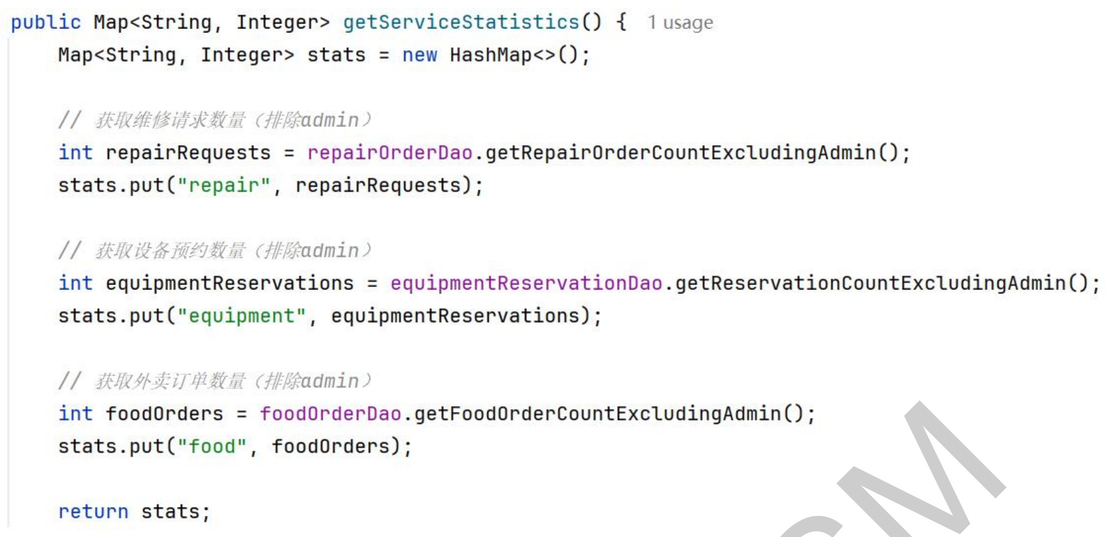
Based on the result of getUserServiceStatistics(). Use Lambda expression to sort in descending order by total usage. Return the sorted user list.



```
public List<UserServiceTypeCount> getUserServiceRanking() {  1 usage
    List<UserServiceTypeCount> statistics = getUserServiceStatistics();
    // 按总使用次数降序排序
    statistics.sort((a, b) -> b.getTotalCount().compareTo(a.getTotalCount()));
    return statistics;
```

Figure 4.82	Owner service ranking

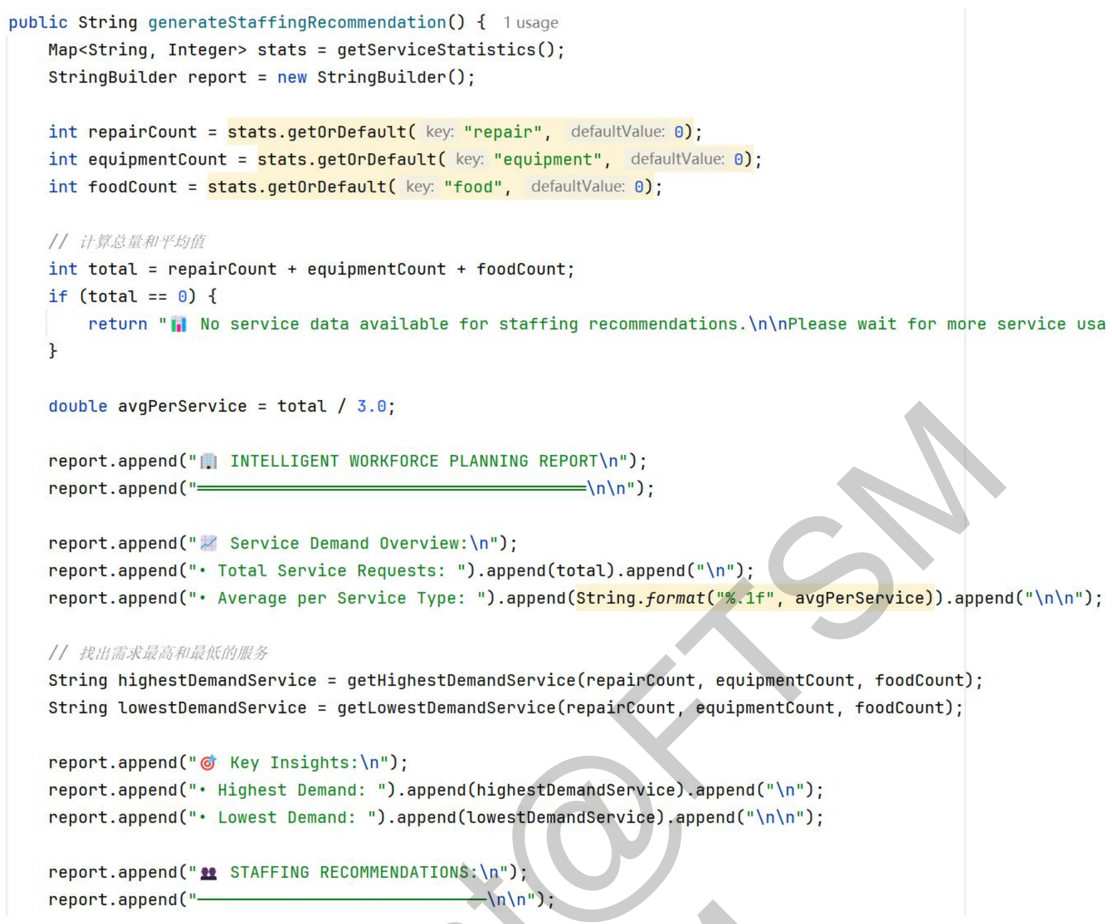
Get the total statistics of three types of services. Each DAO method implements the count excluding administrator users.



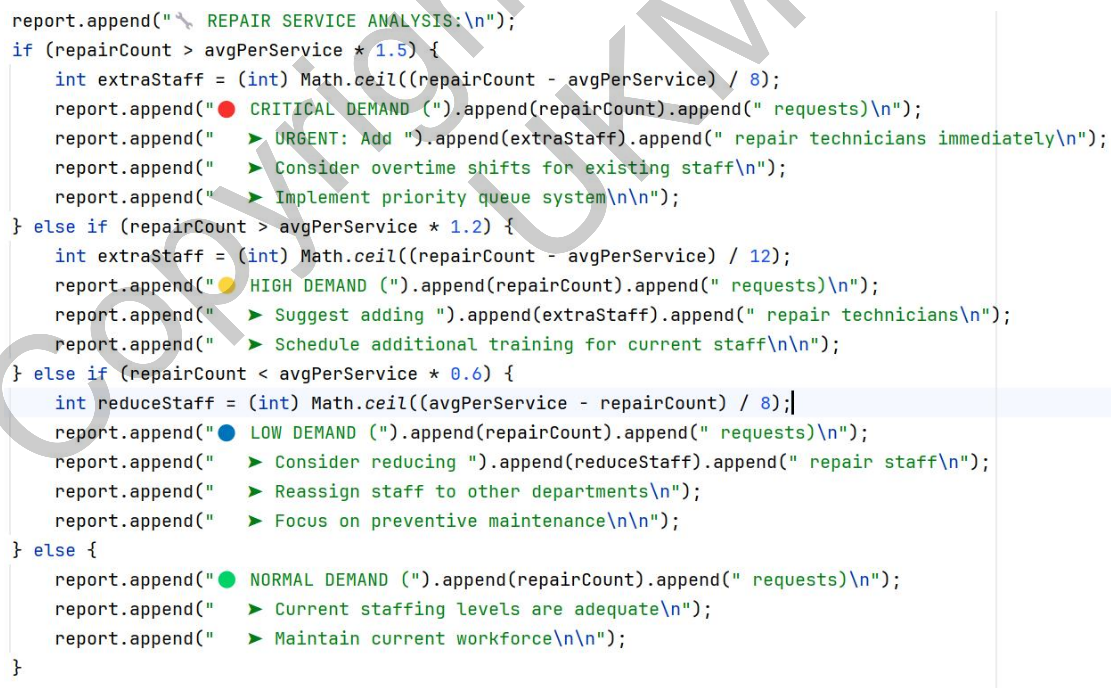
```java
public Map<String, Integer> getServiceStatistics() {  1 usage
    Map<String, Integer> stats = new HashMap<>();

    // 获取维修请求数量（排除admin）
    int repairRequests = repairOrderDao.getRepairOrderCountExcludingAdmin();
    stats.put("repair", repairRequests);

    // 获取设备预约数量（排除admin）
    int equipmentReservations = equipmentReservationDao.getReservationCountExcludingAdmin();
    stats.put("equipment", equipmentReservations);

    // 获取外卖订单数量（排除admin）
    int foodOrders = foodOrderDao.getFoodOrderCountExcludingAdmin();
    stats.put("food", foodOrders);

    return stats;
```

Figure 4.83 Basic service statistics

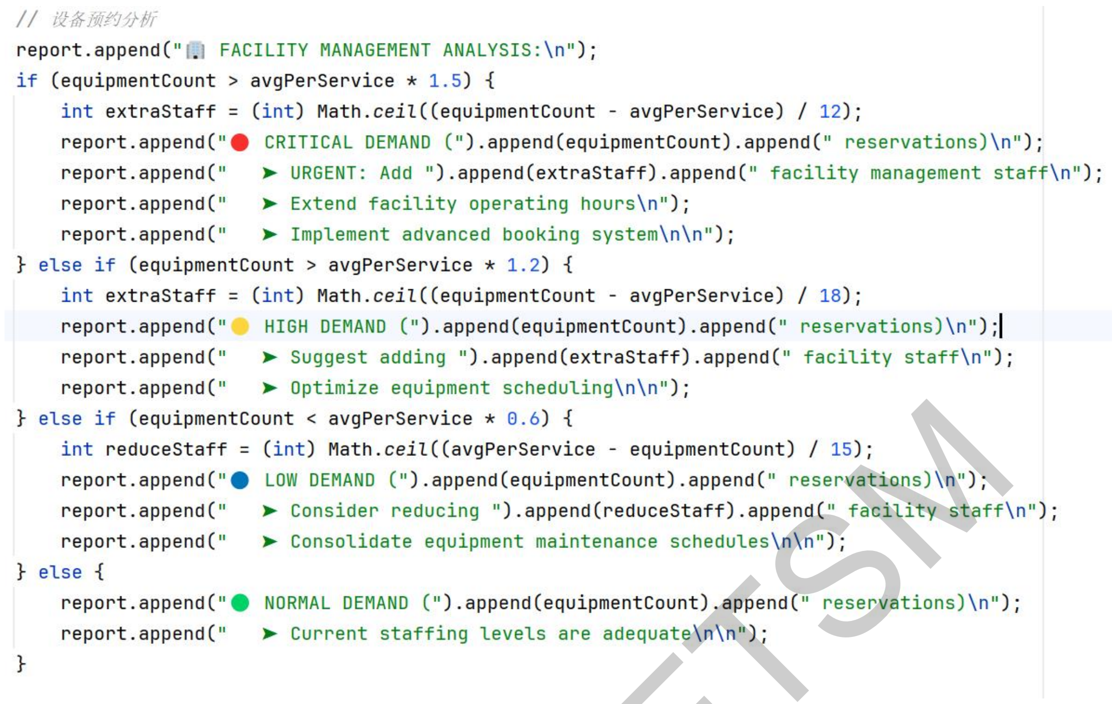
Generate multi-dimensional suggestions based on service volume data. Use thresholds to determine demand levels (CRITICAL/HIGH/NORMAL/LOW). Suggest increasing or decreasing the number of employees based on difference calculations. Includes detailed operational suggestions and workflow optimization tips.



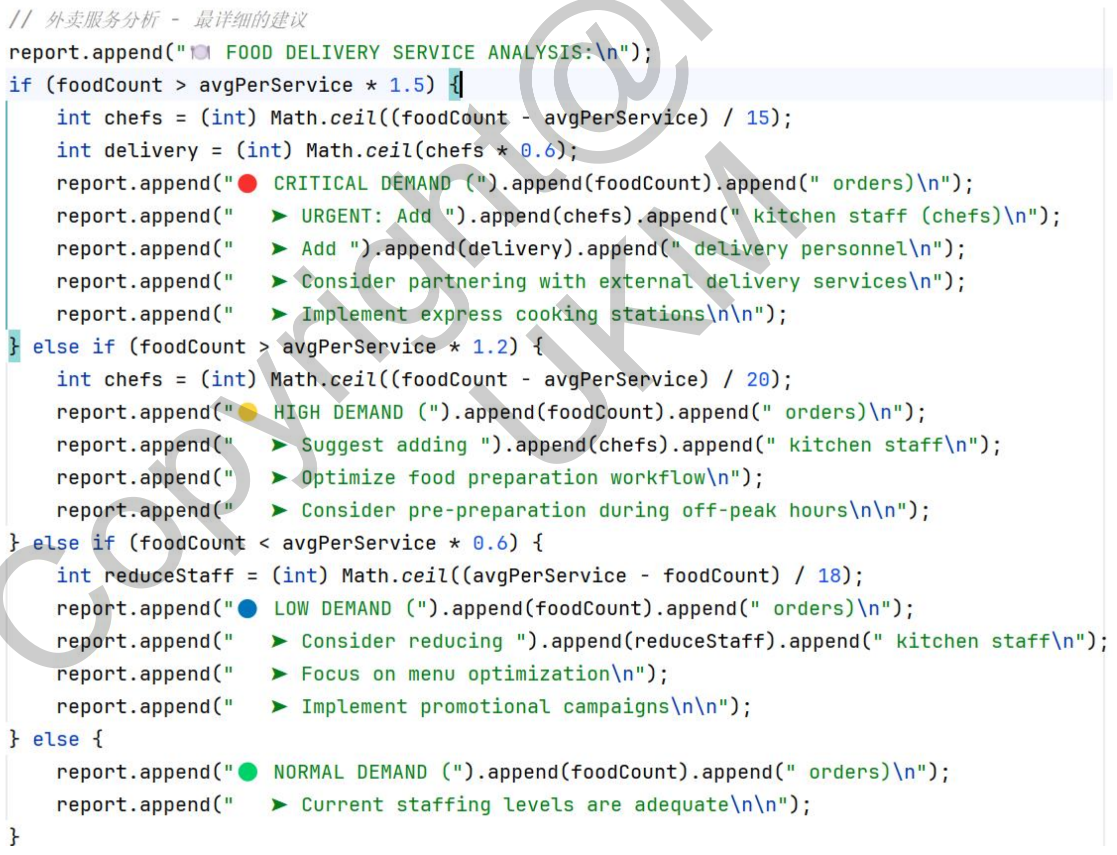
```
public String generateStaffingRecommendation() {  1 usage
    Map<String, Integer> stats = getServiceStatistics();
    StringBuilder report = new StringBuilder();

    int repairCount = stats.getOrDefault( key: "repair",  defaultValue: 0);
    int equipmentCount = stats.getOrDefault( key: "equipment",  defaultValue: 0);
    int foodCount = stats.getOrDefault( key: "food",  defaultValue: 0);

    // 计算总量和平均值
    int total = repairCount + equipmentCount + foodCount;
    if (total == 0) {
        return "📊 No service data available for staffing recommendations.\n\nPlease wait for more service usa
    }

    double avgPerService = total / 3.0;

    report.append("🏢 INTELLIGENT WORKFORCE PLANNING REPORT\n");
    report.append("═══════════════════════════════════════\n\n");

    report.append("📈 Service Demand Overview:\n");
    report.append("• Total Service Requests: ").append(total).append("\n");
    report.append("• Average per Service Type: ").append(String.format("%.1f", avgPerService)).append("\n\n");

    // 找出需求最高和最低的服务
    String highestDemandService = getHighestDemandService(repairCount, equipmentCount, foodCount);
    String lowestDemandService = getLowestDemandService(repairCount, equipmentCount, foodCount);

    report.append("🎯 Key Insights:\n");
    report.append("• Highest Demand: ").append(highestDemandService).append("\n");
    report.append("• Lowest Demand: ").append(lowestDemandService).append("\n\n");

    report.append("👥 STAFFING RECOMMENDATIONS:\n");
    report.append("─────────────────────────────\n\n");
```



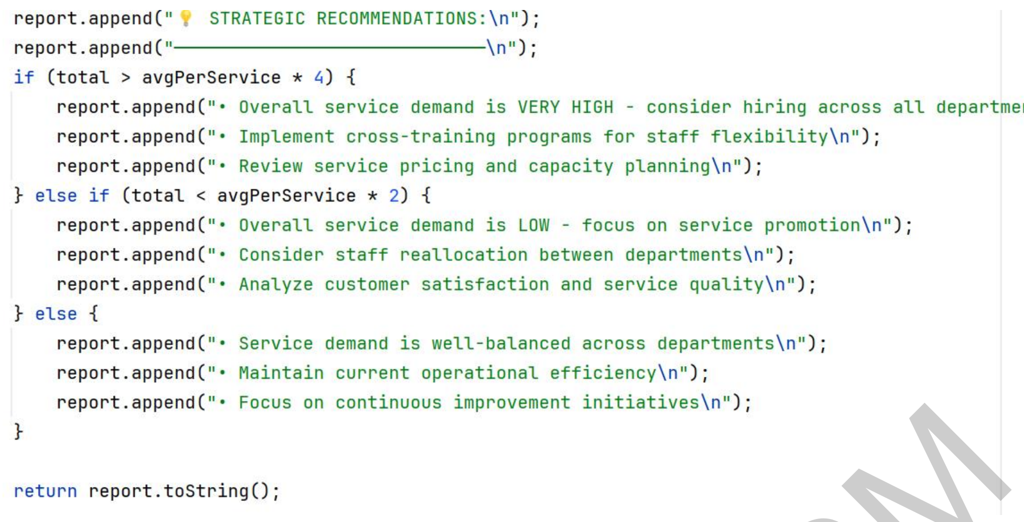
```
report.append("🔧 REPAIR SERVICE ANALYSIS:\n");
if (repairCount > avgPerService * 1.5) {
    int extraStaff = (int) Math.ceil((repairCount - avgPerService) / 8);
    report.append("🔴 CRITICAL DEMAND (").append(repairCount).append(" requests)\n");
    report.append("   ➤ URGENT: Add ").append(extraStaff).append(" repair technicians immediately\n");
    report.append("   ➤ Consider overtime shifts for existing staff\n");
    report.append("   ➤ Implement priority queue system\n\n");
} else if (repairCount > avgPerService * 1.2) {
    int extraStaff = (int) Math.ceil((repairCount - avgPerService) / 12);
    report.append("🟡 HIGH DEMAND (").append(repairCount).append(" requests)\n");
    report.append("   ➤ Suggest adding ").append(extraStaff).append(" repair technicians\n");
    report.append("   ➤ Schedule additional training for current staff\n\n");
} else if (repairCount < avgPerService * 0.6) {
    int reduceStaff = (int) Math.ceil((avgPerService - repairCount) / 8);
    report.append("🔵 LOW DEMAND (").append(repairCount).append(" requests)\n");
    report.append("   ➤ Consider reducing ").append(reduceStaff).append(" repair staff\n");
    report.append("   ➤ Reassign staff to other departments\n");
    report.append("   ➤ Focus on preventive maintenance\n\n");
} else {
    report.append("🟢 NORMAL DEMAND (").append(repairCount).append(" requests)\n");
    report.append("   ➤ Current staffing levels are adequate\n");
    report.append("   ➤ Maintain current workforce\n\n");
}
```



```java
// 设备预约分析
report.append("🏢 FACILITY MANAGEMENT ANALYSIS:\n");
if (equipmentCount > avgPerService * 1.5) {
    int extraStaff = (int) Math.ceil((equipmentCount - avgPerService) / 12);
    report.append("🔴 CRITICAL DEMAND (").append(equipmentCount).append(" reservations)\n");
    report.append("   ➤ URGENT: Add ").append(extraStaff).append(" facility management staff\n");
    report.append("   ➤ Extend facility operating hours\n");
    report.append("   ➤ Implement advanced booking system\n\n");
} else if (equipmentCount > avgPerService * 1.2) {
    int extraStaff = (int) Math.ceil((equipmentCount - avgPerService) / 18);
    report.append("🟡 HIGH DEMAND (").append(equipmentCount).append(" reservations)\n");
    report.append("   ➤ Suggest adding ").append(extraStaff).append(" facility staff\n");
    report.append("   ➤ Optimize equipment scheduling\n\n");
} else if (equipmentCount < avgPerService * 0.6) {
    int reduceStaff = (int) Math.ceil((avgPerService - equipmentCount) / 15);
    report.append("🔵 LOW DEMAND (").append(equipmentCount).append(" reservations)\n");
    report.append("   ➤ Consider reducing ").append(reduceStaff).append(" facility staff\n");
    report.append("   ➤ Consolidate equipment maintenance schedules\n\n");
} else {
    report.append("🟢 NORMAL DEMAND (").append(equipmentCount).append(" reservations)\n");
    report.append("   ➤ Current staffing levels are adequate\n\n");
}
```



```java
// 外卖服务分析 - 最详细的建议
report.append("🍽️ FOOD DELIVERY SERVICE ANALYSIS:\n");
if (foodCount > avgPerService * 1.5) {
    int chefs = (int) Math.ceil((foodCount - avgPerService) / 15);
    int delivery = (int) Math.ceil(chefs * 0.6);
    report.append("🔴 CRITICAL DEMAND (").append(foodCount).append(" orders)\n");
    report.append("   ➤ URGENT: Add ").append(chefs).append(" kitchen staff (chefs)\n");
    report.append("   ➤ Add ").append(delivery).append(" delivery personnel\n");
    report.append("   ➤ Consider partnering with external delivery services\n");
    report.append("   ➤ Implement express cooking stations\n\n");
} else if (foodCount > avgPerService * 1.2) {
    int chefs = (int) Math.ceil((foodCount - avgPerService) / 20);
    report.append("🟡 HIGH DEMAND (").append(foodCount).append(" orders)\n");
    report.append("   ➤ Suggest adding ").append(chefs).append(" kitchen staff\n");
    report.append("   ➤ Optimize food preparation workflow\n");
    report.append("   ➤ Consider pre-preparation during off-peak hours\n\n");
} else if (foodCount < avgPerService * 0.6) {
    int reduceStaff = (int) Math.ceil((avgPerService - foodCount) / 18);
    report.append("🔵 LOW DEMAND (").append(foodCount).append(" orders)\n");
    report.append("   ➤ Consider reducing ").append(reduceStaff).append(" kitchen staff\n");
    report.append("   ➤ Focus on menu optimization\n");
    report.append("   ➤ Implement promotional campaigns\n\n");
} else {
    report.append("🟢 NORMAL DEMAND (").append(foodCount).append(" orders)\n");
    report.append("   ➤ Current staffing levels are adequate\n\n");
}
```



```
report.append("💡 STRATEGIC RECOMMENDATIONS:\n");
report.append("──────────────────────────\n");
if (total > avgPerService * 4) {
    report.append("• Overall service demand is VERY HIGH - consider hiring across all departments\n");
    report.append("• Implement cross-training programs for staff flexibility\n");
    report.append("• Review service pricing and capacity planning\n");
} else if (total < avgPerService * 2) {
    report.append("• Overall service demand is LOW - focus on service promotion\n");
    report.append("• Consider staff reallocation between departments\n");
    report.append("• Analyze customer satisfaction and service quality\n");
} else {
    report.append("• Service demand is well-balanced across departments\n");
    report.append("• Maintain current operational efficiency\n");
    report.append("• Focus on continuous improvement initiatives\n");
}

return report.toString();
```

Figure 4.84 Intelligent Human Proposal Generation

4.2 TEST CASE DESIGN

As described in the Software Development Life Cycle (SDLC), testing is very important to the system. This chapter will document the elements included in the testing phase, including the test plan, the test design, how the tests were conducted, and finally the test results. Testing is evidence that the system is of good quality and meets the functional and non-functional requirements specified in the project proposal

4.2.1 Testing Plan

System testing plays a vital role before the App is released. It can effectively discover potential errors and problems and ensure that the App has good stability and quality before it is released to users.

4.2.2 Testing Objective

The goals of Smart neighbourhood system testing include:

1. Clearing the various tasks that need to be prepared before testing.
2. Studying and selecting testing methods suitable for Smart Neighborhood.
3. Ensuring that a system that meets user needs is developed.

4.2.3 Test materials

The following materials are the operational basis for proper testing:

1. System requirements specification.
2. System design specification.

4.2.4 Traceability Matrix Testing

The traceability matrix test is a document that establishes the relationship between requirements, test cases, and test results. This section helps ensure that all set requirements are thoroughly tested and no requirements are missed during the testing process. By conducting matrix traceability testing regularly, it helps to prove the comprehensiveness and effectiveness of testing. Table 4.1 shows the key functions of Smart Neighborhood APP and the assessed risk level.

Table 4.1 Traceability Matrix Testing

Function ID	Function Name	Risk Level	Function Source
KF1	Register Account	Low	SRS
KF2	Log In	Low	SRS
KF3	Equipment Reservation	High	SRS
KF4	Repair Records	High	SRS
KF5	Repair Staff	Low	SRS
KF6	User Feedback	Low	SRS
KF7	Food Delivery	High	SRS
KF8	Reservation History	Low	SRS
KF9	Data Analysis	High	SRS
KF10	Register User	Low	SRS
KF11	User Management	High	SRS
KF12	Announcement	High	SRS

SRS: Software Requirement Specification

Risk Level: When grading test risks, potential risks should be identified first, and their possible impact and probability of occurrence should be evaluated, so as to make reasonable classifications. Generally speaking, low risk means that the potential problem has little impact on the system and the probability of occurrence is low; while high risk means that once the risk occurs, it may cause a greater impact and its probability of occurrence is relatively high.

4.2.5 Functions Testing

The following table contains the functions that need to be tested according to the System Requirements Specification (SRS) of the APP Management System. The table lists the corresponding functions and the estimated risk level. The following Table 4.2 shows the functions that need to be tested.

Table 4.2 Tested Functional Testing Table

Function ID	Function Name	Risk Level
KF1	Register Account	Low
KF2	Log In	Low
KF3	Equipment Reservation	High
KF4	Repair Records	High
KF5	Food Delivery	High
KF6	Announcement	High
KF7	Data analysis	High

4.2.5.1 Account Register Testing

Table 4.3 Account Register Testing Condition

Test Condition ID	Test Condition
TCON-01-001	No Username
TCON-01-002	No Phone Number
TCON-01-003	No Password
TCON-01-004	No address

Table 4.4 Account Register Decision Table

Condition		TCON	TCON	TCON	TCON
		01-001	01-002	01-003	01-004
No Username	N	N	N	N	N
No Phone Number	N	N	N	N	N
No Password	N	N	N	N	N
No address	N	N	N	N	N

N: Unable to register Y: Can register

The expected effect is that registration is not possible if any of the information is missing.

In the registration interface, you can only enter the registration interface if you select the account type as Owner or Normal.

If any of the inputs of name, phone number, address, and password are empty, you cannot successfully register an account. This is in line with expectations, and the test has passed.

4.2.5.2 Log In Test

Table 4.5 Account Log In Testing Condition

Test Condition ID	Test Condition
TCON-02-001	No Phone Number
TCON-02-002	No Password
TCON-02-003	Wrong phone number
TCON-02-004	Wrong password
TCON-02-005	The phone number password and account type do not match.

Table 4.6 Account Log In Decision Table

Condition	TCON 02-001	TCON 02-002	TCON 02-003	TCON 02-004	TCON 02-005
No Phone Number		P	P	P	P
No Password	P		P	P	P
Wrong phone number		P		I	I
Wrong password	P		I		I
The phone number password and account type do not match.	P	P	I	I	

P: Please fill in all fields. I: Invalid login credentials.

The conditions for successful login must be: the correct phone number and the correct password for the account, and the account type that matches the account must be selected to successfully log in. As expected, the test passed.

4.2.5.3 Equipment Reservation

Table 4.7 Equipment Reservation Test

TCON	A	B	C	D	E	F
03-001	Select equipment	Do you pay the deposit?	After selecting the time you want to reservation, do you want to confirm the reservation?	Check the reservation history function to see if the booking information is available.	Whether the reservation history function interface can cancel the reservation function.	Can the reservation management interface of the Admin staff confirm the refund of the deposit after the use is completed?
03-002	BBQ	CONFIRM	CONFIRM	YES	YES	YES
03-003	BBQ	Cancel	The reservation process stops returning A.			
03-004	BBQ	CONFIRM	Cancel	The reservation process stops returning A.		

On the reservation interface: After selecting the facility you want to reserve, click reserve to confirm whether to pay the deposit. After paying the deposit, select the date and time period. After confirmation, the reservation is successful. After use, the property staff will confirm and return the deposit. The test function is normal and passed as expected.

4.2.5.4 Repair Records Test

j. Function description:

Owners can access the maintenance record page to view the historical maintenance reports of this account. Click "+" to enter the new maintenance application page, select the maintenance type, fill in the specific questions and submit.

Normal-level property staff: can approve the start of maintenance.

Admin and Manager-level staff: can assign specific maintenance personnel.

Admin-level staff: if the budget is found to be too high, it must be submitted to the Manager for approval, and maintenance can only be started after approval.

k. Test target

1. Verify that the function meets the business process requirements.
2. Verify that the permission classification logic is executed correctly.
3. Ensure that the budget approval logic is accurate.

l. Test Types and Methods

1. Functional testing (Black Box Testing)
2. Permission testing
3. Decision path testing (multi-level approval process)

m. Test Case

Table 4.8 Repair Records Test

Use Case number	Test content	Enter description	Expected results	Actual results	State
04-001	Go to the Repair record page	Click on the menu "Repair Records"	Successfully enter the page and display historical maintenance records	As expected	Pass
04-002	Added Repair record	Click "+", select the type, fill in the description and submit	The report was successfully submitted and the status is "Pending"	As expected	Pass

04-003	Normal level approval to start Repair	Log in to the Normal account and click "Approve"	The report status changes to "In Progress" and maintenance personnel cannot be assigned	As expected	Pass
04-004	Admin level assigned Repair personnel	Log in to the Admin account and click the "Assign" button	Successfully assigned personnel, status updated	As expected	Pass
04-005	Admin submits high budget record to Manager for approval	Reporting budget exceeding threshold	The "Manager Approval Required" prompt is displayed, waiting for Manager approval	As expected	Pass
04-006	Manager approves high budget record	Manager logs in and clicks "Approved"	Status updated to "Ready to repair"	As expected	Pass
04-007	Admin attempts to start Repair without Manager approval	Try skipping approval	The message "Manager approval required" appears, prohibiting you from continuing the operation	As expected	Pass

Repair Records functional testing has designed 7 use cases, all of which have been completed. All functions are normal, the permission control logic is clear, and the budget approval process is in line with expectations. There are no functional defects.

4.2.5.5 Food Delivery

n. Function description

Users can select a takeaway menu through the app, which includes four types of dishes (Satsy, Nasi Lemak, Roti, Maggi). After adjusting the number of portions required by clicking the plus (+) or minus (-) sign, click Confirm Order to submit.

The system will automatically fill in the address information filled in when the user registered. After clicking Pay, the order amount will be deducted from the owner's account balance of the account (initial RM2000).

After the meal is delivered, the owner needs to click the Delivered button to complete the order process. Users can view the historical order records on the Food Delivery page.

o. Test target

1. Verify that the takeaway order process is complete and correct
2. Check that the food selection and quantity adjustment functions are working properly
3. Test that the address auto-fill, are accurate
4. Confirm that the order status management (such as "ordered", "delivering", "delivered") is valid
5. Verify that the order is recorded correctly and can be viewed in the history

p. Test method

1. Black Box Testing
2. Functional Testing
3. UI Testing
4. Boundary Value Testing (payment amount, quantity selection, etc.)
5. State Transition Testing (from order placement to delivery)

q. Test Case

Table 4.9 Repair Records Test

Use Case number	Test content	Operation steps	Expected results	Actual results	State
05-001	Show optional Foods	Go to Food Delivery page	Display four takeaway options: Satsy, Nasi Lemak, Roti, Maggi	As expected	Pass
05-002	Adjust the quantity of food	Click the + or - button	Accurately increase or decrease the number of corresponding Food portions	As expected	Pass

05-003	Submit order	Click the Confirm Order button	Automatically fill in the registration address and enter the payment interface	As expected	Pass
05-004	Pay and debit	Click the Pay button	Deduct the total amount from the Owner account balance and jump to the delivery status	As expected	Pass
05-005	Insufficient balance test	The order amount is greater than the account balance (e.g. order amount > RM2000)	A payment failure message is displayed and the order is not generated	As expected	Pass
05-006	Order delivered	After receiving the food, the Owner clicks the Delivered button	Status updated to "Delivered"	As expected	Pass
05-007	View historical order records	Enter the Food Delivery History area	Display the details of completed orders, including items, quantity, amount, address, and status	As expected	Pass

This module has designed 7 core test cases, all of which have been successfully passed. The ordering, payment, and delivery processes of takeout are complete, the system behavior is consistent with the business logic, the data records are accurate, and the user interface is interactive and friendly. No defects that hinder the use of functions

4.2.5.6 Announcement

r. Function description

This function allows authorized property management personnel to publish announcements in the background. Property management personnel are divided into three levels: Manager, Admin and Normal, among which only Manager and Admin level employees have the authority to publish announcements.

The owner user (Owner) automatically receives and displays the latest announcements through the database refresh mechanism, and can view relevant content at any time.

s. Test target

1. Verify whether the permissions of employees at different levels correctly restrict announcement publishing operations
2. Check whether the announcement can be successfully published and stored in the database
3. Ensure that the Owner user end can automatically receive new announcements and display them correctly
4. Verify whether the announcement content is displayed completely and clearly
5. Test whether the system handles unauthorized publishing attempts reasonably

t. Test method

1. Black Box Testing
2. Permission verification test
3. UI display test
4. Data refresh mechanism test
5. Exception handling test (e.g. users with insufficient permissions attempt to publish)

u. Test Case

Table 4.10 Announcement Test

Use Case number	Test content	Operation steps	Expected results	Actual results	State
06-001	Manager user posts announcement	Log in to the Manager account, enter the announcement content and click "Publish"	The announcement was published successfully and the content was saved in the database	As expected	Pass
06-002	Admin user publishes announcement	Log in to the Admin account, enter the announcement content and click "Publish"	The announcement was published successfully and the content was saved in the database	As expected	Pass
06-003	Normal user attempts to post an announcement	Log in to the Normal account and try to post an announcement	It says "No permission to operate" and cannot be published.	As expected	Pass

06-004	Owner user receives new announcements	After the Manager publishes the announcement, the Owner opens the App	New announcements are automatically displayed on the announcement page	As expected	Pass
06-005	Owner View multiple announcements	There are multiple announcement records in the database. The owner opens the App.	All announcements are displayed in chronological order, and the content is complete and readable	As expected	Pass
06-006	Announcement refresh mechanism test	After the Admin posts an announcement, the Owner waits for 5 seconds before entering the announcement page.	The latest announcements are automatically loaded and displayed (no need to refresh manually)	As expected	Pass
06-007	Special characters/extremely long content announcement test	Manager publishes announcements containing special characters or long content	The announcement was published successfully, the Owner displayed complete content, and no interface misalignment occurred	As expected	Pass

This module has designed 7 key test cases, covering permission control, functional logic, interface display and database synchronization mechanism. All test cases passed smoothly, the system behavior was in line with expectations, user operations were smooth, and no functional defects were found.

4.2.5.7 Data Analysis

v. Function description

This module provides module managers with visual function usage statistics and intelligent analysis suggestions. The system analyzes data for the following three services: Facility reservation, Takeaway service, Equipment maintenance feedback.

The analysis results are displayed in a pie chart to show the frequency of service usage and provide staffing optimization suggestions, such as:

Which service has a high or low frequency of use, Which service can reduce manpower, Which service needs to increase manpower.

You can select a specific date to view the analysis data for that day, rather than understanding and predicting user needs in different time periods. The data analysis function is only accessible to Admin and Manager, and Normal employees have no access.

w. Test target

Verify that the data analysis function displays the usage of various services normally.

Check whether the pie chart is clear and accurate. Verify whether the system provides reasonable human resource allocation suggestions. Confirm whether the date filtering function works correctly. Test whether the permission control effectively prevents unauthorized users from accessing.

x. Test method

Black Box Testing. Graphical Data Validation. Role-Based Access Control. Boundary Value Testing (such as processing when there is no data or all are 0). Date Filter Test.

y. Test Case

Table 4.11 Test Cas

Use Case number	Test content	Operation steps	Expected results	Actual results	State
07-001	Admin View data analysis interface	Log in to the Admin account and enter the Data Analysis page	Normal loading of pie chart and staffing suggestions	As expected	Pass
07-002	Manager View data analysis interface	Log in to the Manager account and enter the Data Analysis page	Load the pie chart and suggested content normally	As expected	Pass
07-003	Normal No permission to view	Log in to the Normal account and try to enter the Data Analysis page	There is no data analysis function entry in the Normal interface	As expected	Pass
07-004	Is the pie chart data display correct?	There is service record data, Admin logs in and views	The pie chart shows the proportion of each service, and the total number is accurate	As expected	Pass

07-005	Whether the HR suggestion is reasonable	Assume that the Repair frequency is high and the other two are low	The recommendation says "Increase maintenance staff and reduce takeout/facilities staff"	As expected	Pass
07-006	Date filter function	Select a specific date to view data analysis	Display service statistics and recommendations for the selected date	As expected	Pass
07-007	System response when there is no data	Select a date with no records	The prompt "No data yet", the chart is empty but the interface does not crash	As expected	Pass

The test results of the data analysis module show that the functional logic is complete, the authority control is accurate, the data display is clear, the personnel suggestions are intelligent, and the user interaction is good. The system performs stably under different roles and different data conditions and meets the project requirements.

4.2.6 Testing Result

This section shows the results obtained during the Smart Neighborhood system testing phase. I recorded the actual operation observed after executing the specified test cases and processes. The test results are important for evaluating the quality and stability of the application. Among them, the test log is an effective way to summarize the execution of the entire test process in detail, including whether each test passed or failed. The following table shows the specific log records of this Smart Neighborhood test.

Table 4.12 Test result of the Smart neighbourhood

TCON Number	**Tool**	**Pass /Fail**
1	Manual	Pass
2	Manual	Pass
3	Manual	Pass
4	Manual	Pass
5	Manual	Pass
6	Manual	Pass
7	Manual	Pass

The tabular data is a summary of test cases conducted using manual testing methods, where each test case is identified by a unique test case ID. The results show that all listed test cases were successfully passed, meaning that the corresponding features or functions met the expected standards and no critical issues were encountered. While this data indicates that the tested product or system achieved positive results, a more comprehensive analysis is needed to consider a wider data set and possibly combine other testing methods in order to draw more definitive conclusions about the overall quality and reliability of the system.

4.2.7 Conclusion

This chapter clearly demonstrates the various functions of the Smart Neighborhood system. In addition, it also shows the key codes and the UI design of the user interface. This allows Smart Neighborhood users to understand the development process of the system. This chapter also explains the testing methods and testing processes to enhance the confidence of Smart Neighborhood users. Testing can discover unknown system defects and determine whether the system functions can meet user needs.

5.0 CONCLUSION

The design and development of the "Smart Neighborhood" App aims to improve the management efficiency of ZhongHai Property and the service experience of owners. Through the reasonable planning of functional modules and system testing, the core functions such as announcement release, data analysis, and user feedback were initially realized, verifying the feasibility and practicality of the system. However, during the development process, challenges such as business complexity, data security, and insufficient user participation were also exposed.To address these issues, the report puts forward a series of improvement suggestions, including optimization of technical means, improvement of security mechanisms, introduction of incentive strategies, and integration of external resources. Overall, the application has laid the foundation for building a modern and intelligent community management system, and provided practical reference and development direction for further deepening the construction of "smart communities" in the future.

WANPUDONG(A197914)
Prof. Dr. Mohdjuzaiddin Ab Aziz
FACULTY SCIENCE AND TECHNOLOGY
National University of Malaysia