A close-up of a logo

AI-generated content may be incorrect.A group of students'data

AI-generated content may be incorrect.

**Table of Contents**

[**Assumptions** 4](#_Toc190112000)

[**Flowchart** 6](#_Toc190112001)

[**Sample Input/Output** 18](#_Toc190112002)

[3.1 Login Page 18](#_Toc190112003)

[3.2 Registration 26](#_Toc190112004)

[3.3 Manager Dashboard 30](#_Toc190112005)

[**3.3.1 Pending Registration** 30](#_Toc190112006)

[3.3.2 User Management 35](#_Toc190112007)

[3.3.3 Rate Management 47](#_Toc190112008)

[3.3.4 Hostel Information 54](#_Toc190112009)

[3.3.5 Rules & Regulations 57](#_Toc190112010)

[3.3.6 Profile 60](#_Toc190112011)

[3.4 Staff Dashboard 64](#_Toc190112012)

[3.4.1 Update Account 64](#_Toc190112013)

[3.4.2 Make Payment 65](#_Toc190112014)

[3.4.3 Generate Receipt 68](#_Toc190112015)

[3.4.4 Hostel Information 71](#_Toc190112016)

[3.4.5 Rules & Regulations 73](#_Toc190112017)

[3.4.6 Logout 74](#_Toc190112018)

[3.5 Residents Dashboard 75](#_Toc190112019)

[3.5.2 View Payment Records 76](#_Toc190112020)

[3.5.5 Logout 80](#_Toc190112021)

[**Program source code** 82](#_Toc190112022)

[**Object-oriented Concepts** 82](#_Toc190112023)

[**Java Features** 88](#_Toc190112024)

[**Control Structures** 92](#_Toc190112025)

[**Additional features source code** 93](#_Toc190112026)

[**Object-oriented Concepts** 93](#_Toc190112027)

[Control Structures 104](#_Toc190112028)

[**Java Features** 104](#_Toc190112029)

[**References** 107](#_Toc190112030)

[**Task Division** 108](#_Toc190112031)

# **Assumptions**

1. The User ID must start with "**AD**" for admins, "**ST**" for staff, or "**TP**" for students, followed by exactly **six digits**.
2. The user password must be **at least** **eight characters** long, contain at least one **uppercase letter**, include at least **three digits**, and must **not have any spaces**.
3. The email address must be in a valid format, containing "**@**" and following a proper domain structure.
4. Users are allowed a maximum of **three login attempts** and **three email validation** **attempts** during registration.
5. The system has a default manager account with the User ID "**AD123456**" and the password "**Admin123**"
6. There can only be **one manager account** in the system, and a manager **cannot register a new account** since the default manager account already exists.
7. Only **manager** have the authority to **add, edit, or delete user accounts**
8. Only **manager** have the authority **approve or reject** user registrations.
9. The default **manager account** **cannot be deleted**.
10. Manager can modify their account information but is **not allowed** to **change** their **User ID**.
11. During registration, a **security keyword** is required for **password recovery**.
12. Users can check their registration status by clicking “Check Status”. There are four registration status states: **PENDING** (processing), **APPROVED** (accepted), **REJECTED** (denied), and **NOT\_FOUND** (no record exists).
13. If a user account has not been approved or rejected, it will be stored in “**pending\_registration.txt**” while rejected accounts will be saved in “**reject\_registration.txt**”.
14. Rate management includes three types of rates: **Rental**, **Utility**, and **Penalty**, and each rate must have a unique ID.
15. Only **manager** have the authority to **modify Hostel Information** and **Hostel Rules & Regulations**.
16. Only **manager** has the authority to **add, edit, and delete rates** in rate management.
17. The hostel offers three **types of rooms**: **Single Room**, **Shared Room(2)** for two persons, and **Shared Room(4)** for four persons.
18. The default **Rental Rates** are:

* R001: Single Room (RM1000)
* R002: Shared Room(2) (RM700)
* R003: Shared Room(4) (RM500)

1. The default **Utility Rates** are:

* U001: Water (RM30)
* U002: Electricity (RM70)

1. The default **Penalty Rates** are:

* P001: Late Payment (RM50)
* P002: Weekly Increment (RM5)

1. Rate information is stored in separate files: "**rent\_rate.txt**" for Rental rates, "**utility\_rate.txt**" for Utility rates, and "**penalty\_rates.txt**" for Penalty rates.

# **Flowchart**

|  |  |
| --- | --- |
| **Login and Register** |  |
| Login | **A diagram of a software flowchart  Description automatically generated** |
| **Register** | A diagram of a computer program  Description automatically generated |

|  |  |
| --- | --- |
| **Manager Menu** |  |
| Main Menu | A diagram of a software company  Description automatically generated with medium confidence |
| Pending Registration Menu | A diagram of a diagram  Description automatically generated |
| User Management Menu | A diagram of a flowchart  Description automatically generated |
| Rate Management Menu | A diagram of a flowchart  Description automatically generated |
| **Staff Menu** |  |
| Main Menu |  |
| Update Account Menu |  |
| Generate Payment Menu |  |
| Generate Receipt Menu |  |
| **Resident Menu** |  |
| Main Menu |  |
| Update Account Menu |  |
| View Payment Receipt Menu |  |

# **Sample Input/Output**

## 3.1 Login Page

|  |  |
| --- | --- |
| **Input** | **Output** |
| **A screenshot of a login screen  Description automatically generated**  Figure 3.1.1 – Login Page | **A screenshot of a computer  Description automatically generated**  Figure 3.1.2 – Manager Menu |
| If a user already has an account, they can enter their **User ID and Password** in the provided fields and click the **'Login' button** to access the system. | After logging in, the user will be directed to the **Manager Menu**. |
| **A screenshot of a computer  Description automatically generated**  Figure 3.1.3 – Login Page | A screenshot of a computer  Description automatically generated  Figure 3.1.4 – Registration Page |
| If a user doesn't have an account, they can click the **'Register' button** to create one.  **Login Attempts** | The user will be directed to the Registration Page. |
| Figure 3.1.5 – Login Page with Filled Fields | Figure 3.1.6 – Login Error with 2 attempts remaining |
| If the user enters the **wrong** User ID or Password for the **first time**. | A **'Login Error' pop-up window** appears, indicating invalid credentials and 2 **attempts remaining**. Users have a maximum of **3 login attempts** before being locked out. |
| Figure 3.1.7 – Login Page with Filled Fields | Figure 3.1.8 – Login Error with 1 attempt remaining |
| If the user enters the **wrong** User ID or Password for the **second time**. | A **'Login Error' pop-up window** appears, indicating invalid credentials and **1 attempt remaining**. |
| Figure 3.1.9 – Login Page with Filled Fields | Figure 3.1.10 – 'Login Error' with Account locked |
| If the user enters the **wrong** User ID or Password for the **third time**. | A **'Login Error'** pop-up window appears, indicating that the **account is locked**. The user must restart the application to try again. |
| **Hide and Show Password** |  |
| Figure 3.1.11 – Login Page with the Eye Icon Button | Figure 3.1.12 – Login Page with the Visible Password |
| User can click the **eye icon button** to view the hidden password. | The password will then be **visible**. |
| Figure 3.1.13 – Login Page with the Eye Icon Button | Figure 3.1.14 – Login Page with "Show/Hide Password" Tooltip |
| When the user **hovers** over the eye or lock icon button, a tooltip will appear. | The **tooltip** displays **'Show/Hide Password'** and allows the user to toggle the password visibility. |
| **Clear Fields to Reset** |  |
| A screenshot of a login screen  Description automatically generated  Figure 3.1.15 – Login Page with the Clear Button | A screenshot of a login screen  Description automatically generated  Figure 3.1.16 – Login Page with Empty Fields |
| User can click the **'Clear' button** to reset the fields. | The User ID and Password **fields** will be **empty**. |
| **Forgot Password & Reset Password** |  |
| Figure 3.1.17 - Login Page with the Forgot Password Button | Figure 3.1.18 - 'Reset Password' Pop-up Window |
| If a user forgets their password, they can click **'Forgot Password?'** to reset it. | A **'Reset Password' pop-up window** appears. The user must enter the correct User ID and security keyword before they can reset a new password. |
| Figure 3.1.19 - Reset Password | A screenshot of a computer  Description automatically generated  Figure 3.1.20 – 'Error' Pop-up Window |
| If user enter **wrong** User ID or Security Keyword. | An **'Error' pop-up window** appears, indicating invalid User ID or Security Keyword. |
| A screenshot of a computer  Description automatically generated  Figure 3.1.21 - Reset Password | A screenshot of a computer  Description automatically generated  Figure 3.1.22 – 'Success Pop-up' Window |
| If user enter **correct** User ID or Security Keyword. | A **'Success' pop-up window** appears and password successfully reset. |
| **Check Account Registration Status** |  |
| A screenshot of a computer  Description automatically generated  Figure 3.1.23 - Login Page with the Check Status Button | A screenshot of a computer  Description automatically generated  Figure 3.1.24 - 'Check Registration Status' Pop-up Window |
| User can click the **'Check Status' button** to check whether their account is Approved, Rejected, or Pending after registration. | A **'Check Registration Status' pop-up window** will appear and prompt the user to enter the User ID they want to check. |
| Figure 3.1.25 - Check Registration Status Pop-up Window with UserID | A screenshot of a computer  Description automatically generated  Figure 3.1.26 - 'Registration Status' Pop-up Window |
| User enters their User ID and clicks **'OK' button**. | A **'Registration Status' pop-up window** will appear, indicating that the account is **approved**. |
| A screenshot of a computer  Description automatically generated  Figure 3.1.27 - Check Registration Status Pop-up Window with UserID | A screenshot of a computer  Description automatically generated  Figure 3.1.28 - 'Registration Status' Pop-up Window |
| User enters their User ID and clicks **'OK' button**. | A **'Registration Status' pop-up window** will appear, indicating that the account is **rejected**. |
| A screenshot of a computer  Description automatically generated  Figure 3.1.29 - Check Registration Status Pop-up Window with UserID | A screenshot of a computer  Description automatically generated  Figure 3.1.30 - 'Registration Status' Pop-up Window |
| User enters their User ID and clicks **'OK' button**. | A **'Registration Status' pop-up window** will appear, indicating that the account is **pending**. |
| **Log Out** |  |
| Figure 3.1.31 – Log Out | A screenshot of a computer  Description automatically generated  Figure 3.1.32 – 'Confirm Exit' Pop-up Window |
| User clicks the **Log Out icon** to log out. | A **'Confirm Exit' pop-up window** will appear, asking the user, "Are you sure you want to exit?" and prompt user to select Yes or No. |
| Figure 3.1.33 – Log Out | Figure 3.1.34 – Login Page with "Show/Hide Password" Tooltip |
| When the user **hovers** over the **home icon button**, a tooltip will appear. | The **tooltip** displays **"Exit Application"** and allows the user to log out. |

## 3.2 Registration

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Invalid Input** |  |
| A screenshot of a computer  Description automatically generated  Figure 3.2.1 – User Registration with the Full Name Input Filled | A screenshot of a computer  Description automatically generated  Figure 3.2.2 – 'Validation Error' Pop-up Window |
| The user enters an **invalid full name**. Full Name can **only contain alphabets**. | A **'Validation Error' pop-up window** appears, indicating **invalid Full Name**. |
| A screenshot of a computer  Description automatically generated  Figure 3.2.3 – User Registration with the User ID Input Filled | A screenshot of a computer  Description automatically generated  Figure 3.2.4 – 'Validation Error' Pop-up Window |
| The user enters an **invalid User ID format**. The User ID must be in the format TPxxxxxx for Residents and STxxxxxx for Staff. | A **' Validation Error' pop-up window** appears, indicating **invalid User ID** **format**. |
| A screenshot of a computer  Description automatically generated  Figure 3.2.5 – User Registration with the Password Input Filled | A screenshot of a computer  Description automatically generated  Figure 3.2.6 – 'Validation Error' Pop-up Window |
| The user enters an **invalid password** **format**. Password must be at least 8 characters long, include at least one uppercase letter, contain at least three digits, and must not have any spaces. | A **' Validation Error' pop-up window** appears, indicating **invalid Password format**. |
| A screenshot of a computer  Description automatically generated  Figure 3.2.7 – User Registration with the Email Input Filled | A screenshot of a computer  Description automatically generated  Figure 3.2.8 – 'Validation Error' Pop-up Window |
| The user enters an **invalid email format**. Email address must be in a valid format, containing "@" and following a proper domain structure. | A **' Validation Error' pop-up window** appears, indicating **invalid Email format**. |
| Figure 3.2.9 – User Registration with the Security Keyword Input Filled | Figure 3.2.10 – 'Validation Error' Pop-up Window |
| The user enters an **invalid Security Keyword** format. The Security Keyword must be at least 6 characters long and can include letters, numbers, and special characters. | A **'Validation Error' pop-up window** appears, indicating **invalid Security Keyword format**. |
| **Valid Input** |  |
| Figure 3.2.11 – User Registration with All Input Filled | Figure 3.2.12 – 'Registration Success' Pop-up Window |
| The user inputs the **correct format** for all fields. | A **'Registration Success' pop-up window** will appear, indicating that the registration has been **submitted successfully**. |
| Figure 3.2.13 – User Registration with the Scroll Bar | Figure 3.2.14 – Registration Information |
| The user can scroll the **scrollbar** to view additional information. | As the user scrolls the scrollbar, **more information** becomes visible. |
| Figure 3.2.15 – User Registration with the Exist User ID | Figure 3.2.16 – 'Registration Error' pop-up window |
| The user tries to register an account with a User ID that **has already been registered**. | A **'Registration Error' pop-up window** appears, indicating **the account already exist.** |
| Figure 3.2.17 – User Registration with the 'Cancel' Button | Figure 3.2.18 – Login Page |
| The user can click the **'Cancel' button** to return to the Login page. | The user is brought back to the **Login page**. |

## 3.3 Manager Dashboard

### **3.3.1 Pending Registration**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Figure 3.3.1.1 – Manager Menu | Figure 3.3.1.2 – Pending Registration |
| The user clicks on **'Pending Registration'** to navigate to the Pending Registration page. | The user will be brought to the Pending Registration page. |
| Figure 3.3.1.3 – Pending Registration with the 'Filter by Sort' Option | Figure 3.3.1.4 – Pending Registration with 'Filter by Sort' Option |
| The user can click the **'Filter by Sort' dropdown arrow** to view the list of options. | It now displays a list of options: **All Users, Staff, and Residents**. |
| Figure 3.3.1.5 – Pending Registration with the 'Filter by Sort' Option Selected | Figure 3.3.1.6 – Pending Registration with the 'Filter by Sort' Option Selected for Staff |
| The user clicks the **'Staff'** option. | It now displays only the pending registrations of **Staff accounts**. |
| Figure 3.3.1.7 – Pending Registration with the 'Filter by Sort' Option Selected | Figure 3.3.1.8 – Pending Registration with the 'Filter by Sort' Option Selected for Resident |
| The user clicks the **'Residents'** option. | It now displays only the pending registrations of **Residents accounts**. |
| Figure 3.3.1.9 – Approve Pending Registration | Figure 3.3.1.10 – 'Confirm Approval' Pop-up Window |
| The user clicks the account they want to approve and then clicks the **'Approve' button.** | A **'Confirm Approval' pop-up window** will appear, prompting the user to confirm with a "Yes" or "No" option. |
| Figure 3.3.1.11 – 'Confirm Approval' Pop-up Window | Figure 3.3.1.12 – 'Success' Pop-up Window |
| The user clicks 'Yes'. | A **'Success' pop-up window** will appear, indicating that the registration has been **approved successfully**. |
| Figure 3.3.1.13 – Reject Pending Registration | Figure 3.3.1.14 – 'Confirm Rejection' Pop-up Window |
| The user clicks the account they want to reject and then clicks the **'Reject' button.** | A **'Confirm Approval' pop-up window** will appear, prompting the user to confirm with a "Yes" or "No" option. |
| Figure 3.3.1.15 – 'Confirm Approval' Pop-up Window | Figure 3.3.1.16 – 'Success' Pop-up Window |
| The user clicks 'Yes'. | A **'Success' pop-up window** will appear, indicating that the registration has been **rejected successfully**. |
| Figure 3.3.1.17 – Pending Registration with Back Arrow Button | Figure 3.3.1.18 – Pending Registration with "Show/Hide Password" Tooltip |
| When the user hovers over the **back arrow button**, a **tooltip** will appear. | The tooltip displays **'Back to Dashboard'** and allows the user to return to the Manager Menu. |
| Figure 3.3.1.19 – Pending Registration with Back Arrow Button | **A screenshot of a computer  Description automatically generated**  Figure 3.3.1.20 – Manager Menu |
| The user clicks the **back arrow button**. | The user is brought back to the **Manager Menu**. |

### 3.3.2 User Management

|  |  |
| --- | --- |
| Figure 3.3.2.1 – Manager Menu | Figure 3.3.2.2 – User Management |
| The user clicks on **'User Management'** to navigate to the User Management page. | The user will be brought to the **User Management page**. |
| **Add User** |  |
| Figure 3.3.2.3 – User Management with Add User | Figure 3.3.2.4 – 'Add New User' Pop-up Window |
| The user clicks **'Add User'** to add a new user. | An **'Add New User' pop-up window** will appear, allowing the user to enter the new **user's details** to add them. |
| Figure 3.3.2.5 – 'Add New User' Pop-up Window with the 'User Type' dropdown arrow | Figure 3.3.2.5 – 'Add New User' Pop-up Window with the 'User Type' Option |
| The user can click the **'User Type' dropdown arrow** to view the list of options. | It now displays **two options: Staff and Resident.** The Manager is not allowed to have more than one account. |
| Figure 3.3.2.7 – 'Add New User' Pop-up Window | Figure 3.3.2.8 – 'Validation Error' Pop-up Window |
| If the user **doesn't fill in all the fields** and clicks the 'Save' button. | a **'Validation Error' pop-up window** will appear, indicating that **all fields are required**. |
| Figure 3.3.2.9 – 'Add New User' Pop-up Window | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.2.10 – 'Validation Error' Pop-up Window |
| The user enters an **invalid User ID format**, such as selecting 'Staff' in the User Type field but entering a resident User ID (which starts with 'TP'). | A **'Validation Error' pop-up window** will appear, indicating that **Staff user ID** must start with **'ST'**. |
| A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.2.11 – 'Add New User' Pop-up Window | Figure 3.3.2.12 – 'Validation Error' Pop-up Window |
| The user enters an **invalid User ID format**, such as selecting 'Resident' in the User Type field but entering a staff User ID (which starts with 'ST'). | a **'Validation Error' pop-up window** will appear, indicating that Resident **user ID** must start with **'TP'**. |
| Figure 3.3.2.13 – 'Add New User' Pop-up Window | Figure 3.3.2.14 – 'Validation Error' Pop-up Window |
| The user enters an **invalid full name**. Full Name can **only contain alphabets**. | A **'Validation Error' pop-up window** will appear, indicating that **Invalid full name format**. |
| Figure 3.3.2.15 – 'Add New User' Pop-up Window | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.2.16 – 'Validation Error' Pop-up Window |
| The user enters an **invalid email format**. Email address must be in a valid format, containing "@" and following a proper domain structure. | a **'Validation Error' pop-up window** will appear, indicating that **Invalid email format**. |
| Figure 3.3.2.17 – 'Add New User' Pop-up Window | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.2.18 – 'Validation Error' Pop-up Window |
| The user enters an **invalid password** **format**. Password must be at least 8 characters long, include at least one uppercase letter, contain at least three digits, and must not have any spaces. | a **'Validation Error' pop-up window** will appear, indicating that **Invalid password format**. |
| Figure 3.3.2.19 – 'Add New User' Pop-up Window | Figure 3.3.2.20 – User Management |
| If all fields are **correctly filled in**, the user clicks **'Save'** to proceed. | The user account will be **saved successfully** and will immediately appear in User Management. |
| **Edit User** |  |
| Figure 3.3.2.21 – User Management | Figure 3.3.2.22 – 'No Selection' Pop-up Window |
| If the user **doesn't select an account** and clicks the **'Edit User'** button. | A **'No Selection' pop-up window** will appear, stating 'Please select a user to edit'. |
| Figure 3.3.2.23 – User Management with Edit User | Figure 3.3.2.24 – 'Edit User' Pop-up Window |
| The user clicks on an account and then clicks the **'Edit User' button** to **edit the account details**. | An **'Edit User' pop-up window** will appear, displaying the user's **current account information** and allowing them to **make** **edits**. |
| Figure 3.3.2.25 – 'Edit User' Pop-up Window with Save Button | Figure 3.3.2.26 – User Management |
| After editing, the user clicks the **'Save' button** to save the **updated information**. | The user account will be **saved successfully** and will immediately appear in User Management. |
| **Delete User** |  |
| Figure 3.3.2.27 – Rate Management | Figure 3.3.2.28 – 'No Selection' Pop-up Window |
| If the user **doesn't select a user** and clicks the **'Delete User'** button. | A **'No Selection' pop-up window** will appear, stating 'Please select a user to delete'. |
| Figure 3.3.2.25 – User Management with Delete User | Figure 3.3.2.26 – 'Confirm Deletion' Pop-up Window |
| The user clicks on an account and then clicks the **'Delete User' button** to **delete the account**. | A **'Confirm Deletion' pop-up window** will appear, prompting the user with 'Are you sure you want to delete? |
| Figure 3.3.2.27 – 'Confirm Deletion' Pop-up Window | Figure 3.3.2.28 – 'Success' Pop-up Window |
| The user clicks the **'Yes' button** to confirm the deletion. | A **'Success' pop-up window** will appear, The user account will be **deleted successfully.** |
| Figure 3.3.2.29 – 'Success' Pop-up Window | Figure 3.3.2.30 – User Management |
| The user clicks the **'OK' button.** | The user account is **deleted successfully** and will immediately disappear from User Management. |
| **Filter By** |  |
| Figure 3.3.2.30 – User Management with the 'Filter by Sort' Option | Figure 3.3.2.31 – User Management with 'Filter by Sort' Option |
| The user can click the **'Filter by Sort' dropdown arrow** to view the list of options. | It now displays a list of options: **All Users, Staff, and Residents**. |
| Figure 3.3.2.32 – User Management with the 'Filter by Sort' Option Selected | Figure 3.3.1 33 – User Management with the 'Filter by Sort' Option Selected for All Users |
| The user clicks the **'All Users'** option. | It now displays **all the user account** in this system. |
| Figure 3.3.2.34 – User Management with the 'Filter by Sort' Option Selected | Figure 3.3.1 35 – User Management with the 'Filter by Sort' Option Selected for All Users |
| The user clicks the **'Staff'** option. | It now displays only **staff user accounts**. |
| Figure 3.3.2.34 – User Management with the 'Filter by Sort' Option Selected | Figure 3.3.1 35 – User Management with the 'Filter by Sort' Option Selected for Staff |
| The user clicks the **'Staff'** option. | It now displays only **staff user accounts**. |
| Figure 3.3.2.34 – User Management with the 'Filter by Sort' Option Selected | Figure 3.3.2.35 – User Management with the 'Filter by Sort' Option Selected for Residents |
| The user clicks the **'Residents'** option. | It now displays only **resident user accounts**. |
| **Search Bar** |  |
| Figure 3.3.2.36 – User Management with the Search Bar | Figure 3.3.2.37 – User Management |
| A user can enter a **specific user ID** in the search bar to find it. | It will then **display the user ID**. |
| Figure 3.3.2.38 – User Management with the Search Bar | Figure 3.3.2.39 – User Management |
| A user can enter a **specific name** in the search bar to find it. | It will then **display the name**. |
| Figure 3.3.2.40 – User Management with the Search Bar | Figure 3.3.2.41 – User Management |
| A user can enter an **alphabet** in the search bar to find it. | It will then **display the User ID or Full name** that includes the **alphabet.** |
| Figure 3.3.2.42 – User Management with the Search Bar | Figure 3.3.2.43 – User Management |
| A user can enter a **number** in the search bar to find it. | It will then **display the User ID** that includes the **number.** |
|  |  |

### 3.3.3 Rate Management

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated  Figure 3.3.3.1 – Manager Menu | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.2 – Rate Management |
| The user clicks on **'Rate Management'** to navigate to the Rate Management page. | The user will be brought to the **Rate Management page**. |
| **Add Rate** |  |
| A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.3 – Rate Management with Add Rate | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.4 – 'Add New Rate' Pop-up Window |
| The user clicks **'Add Rate'** to add a new rate. | An **'Add New Rate' pop-up window** will appear, allowing the user to enter the **new rate's details**, including the description and amount, to add them. |
| A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.5 – 'Add New Rate' Pop-up Window with the 'User Type' dropdown arrow | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.6 – 'Add New Rate' Pop-up Window with the 'User Type' Option |
| The user can click the **'Rate Type' dropdown arrow** to view the list of options. | It now displays **three options: Rental, Utility and Penalty.** |
| A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.7 – 'Add New Rate' Pop-up Window | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.8 – 'Error' Pop-up Window |
| The user can enter a **description** for the rate type. If the **amount** is invalid, an error pop-up will appear. | An **'Error' pop-up window** will appear, stating that the amount must have **0 to 2 decimal places**. |
| A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.9 – 'Add New Rate' Pop-up Window with Save Button | A screenshot of a computer  AI-generated content may be incorrect.  Figure 3.3.3.10 – Rate Management |
| If all fields are **correctly filled in**, the user clicks **'Save'** to proceed. | The rate will be saved successfully and will immediately appear in Rate Management. The **Rate ID** will be **generated automatically** and sequentially: Rental as Rxxx, Utility as Uxxx, and Penalty as Pxxx. |
| **Edit Rate** |  |
| Figure 3.3.3.11 – Rate Management | Figure 3.3.3.12 – 'No Selection' Pop-up Window |
| If the user **doesn't select a rate** and clicks the **'Edit Rate'** button. | A **'No Selection' pop-up window** will appear, stating 'Please select a rate to edit'. |
| Figure 3.3.3.13 – Rate Management with Edit User | Figure 3.3.3.14 – 'Edit Rate' Pop-up Window |
| The user clicks on a rate and then clicks the **'Edit Rate' button** to **edit the Rate details**. | An **'Edit Rate' pop-up window** will appear, displaying the **current rate information** and allowing user to **make** **edits**. |
| Figure 3.3.3.15 – 'Edit Rate' Pop-up Window | Figure 3.3.3.16 – 'Error' Pop-up Window |
| If the **amount** is invalid, an error pop-up will appear. | An **'Error' pop-up window** will appear, stating that the amount must have **0 to 2 decimal places**. |
| Figure 3.3.3.17 – 'Edit Rate' Pop-up Window | Figure 3.3.3.18 – Rate Management |
| If all fields are **correctly filled in**, the user clicks **'Save'** to proceed. | The rate will be **modified successfully** as the picture show the R002 description and amount updated and the Last Updated Date will also update to the day modify |
| **Delete Rate** |  |
| Figure 3.3.3.19 – Rate Management with Delete Rate | Figure 3.3.3.20 – 'No Selection' Pop-up Window |
| If the user **doesn't select a rate** and clicks the **'Delete Rate'** button. | A **'No Selection' pop-up window** will appear, stating 'Please select a rate to delete'. |
| Figure 3.3.3.21 – Rate Management with Delete User | Figure 3.3.3.22 – 'Confirm Deletion' Pop-up Window |
| The user clicks on a rate and then clicks the **'Delete Rate' button** to **remove** it. | A **'Confirm Deletion' pop-up window** will appear, prompting the user with 'Are you sure you want to delete? ' |
| Figure 3.3.3.23 – 'Confirm Deletion' Pop-up Window | Figure 3.3.3.24 – 'Success' Pop-up Window |
| The user clicks the **'Yes' button** to confirm the deletion. | A **'Success' pop-up window** will appear, The rate will be **deleted successfully.** |
| Figure 3.3.3.25 – 'Success' Pop-up Window | Figure 3.3.3.26 – Rate Management |
| The user clicks the **'OK' button.** | The rate is **deleted successfully** and will immediately disappear from Rate Management. |
| **Filter by** |  |
| Figure 3.3.3.27 – Rate Management with the 'Filter by Sort' Option | Figure 3.3.3.28 – Rate Management with the 'Filter by Sort' Option |
| The user can click the **'Filter by Sort' dropdown arrow** to view the list of options. | It now displays a list of options: **All Rates, Rental Rates, Utility Rates and Penalty Rates**. |
| Figure 3.3.3.29 – Rate Management with the 'Filter by Sort' Option Selected | Figure 3.3.3.30 – Rate Management with the 'Filter by Sort' Option Selected for All Rates |
| The user clicks the **'All Rates'** option. | It now displays **all the rates** in this system. |
| Figure 3.3.3.31 – Rate Management with the 'Filter by Sort' Option Selected | Figure 3.3.3.32 – Pending Registration with the 'Filter by Sort' Option Selected for Rental Rates |
| The user clicks the **'Rental Rates'** option. | It now displays only **rental rates**. |
| Figure 3.3.3.33 – Rate Management with the 'Filter by Sort' Option Selected | Figure 3.3.3.34 – Pending Registration with the 'Filter by Sort' Option Selected for Utility Rates |
| The user clicks the **'Utility Rates'** option. | It now displays only **utility rates**. |
| Figure 3.3.3.35 – Rate Management with the 'Filter by Sort' Option Selected | Figure 3.3.3.36 – Pending Registration with the 'Filter by Sort' Option Selected for Penalty Rates |
| The user clicks the **'Penalty Rates'** option. | It now displays only **penalty rates**. |

### 3.3.4 Hostel Information

|  |  |
| --- | --- |
| A screenshot of a computer menu  Description automatically generated  Figure 3.3.4.1 – Manager Menu | Figure 3.3.4.2 – APU Hostel Information |
| The user clicks on **'Hostel Information'** to view APU Hostel details. | APU Hostel Information will be displayed. |

|  |  |
| --- | --- |
| Figure 3.3.4.3 – Hostel Information with the Scroll Bar | Figure 3.3.4.4 – Hostel Images |
| The user can scroll the **scrollbar** to view additional information. | As the user scrolls the scrollbar, **more information** becomes visible. |
| Figure 3.3.4.5 – Hostel Information with Modify Information | Figure 3.3.4.6 – Hostel Information with 'Edit Hostel Information' Pop-up Window. |
| The user can click **'Modify Information'** to edit hostel details. | An **'Edit Hostel Information'** pop-up window appears, allowing the user to make modifications. |
| Figure 3.3.4.7 – Hostel Information with Save Changes | Figure 3.3.4.8 – Hostel Information with 'Success' Pop-up Window |
| The user can click **'Save Changes'** to edit hostel details. | A **'Success'** **pop-up window** appears, indicating that the changes have been saved successfully. |
| Figure 3.3.4.9 – Hostel Information with 'Success' Pop-up Window | Figure 3.3.4.10 – APU Hostel Information |
| The user clicks the **'OK' button**. | Now, the user can view the **updated Hostel Information**. |
| Figure 3.3.4.11 – APU Hostel Information with the Back Icon Button | **A screenshot of a computer  Description automatically generated**  Figure 3.3.4.12 – Manager Menu |
| When the user **hovers** over the **back arrow button**, a **tooltip** will appear. **Clicking** the back arrow button will **navigate back** to the **Manager Menu**. | The user is brought back to the **Manager Menu**. |

### 3.3.5 Rules & Regulations

|  |  |
| --- | --- |
| A screenshot of a computer menu  Description automatically generated  Figure 3.3.5.1 – Manager Menu | Figure 3.3.5.2 – Hostel Rules & Regulations |
| The user clicks on **'Rules & Regulations'** to view Hostel Rules details. | APU Hostel Rules will be displayed. |
| Figure 3.3.5.3 – Hostel Rules & Regulations with the Scroll Bar | Figure 3.3.5.4 – Hostel Rules & Regulations |
| The user can scroll the **scrollbar** to view **additional information**. | As the user scrolls the scrollbar, **more information** becomes visible. |
| Figure 3.3.5.5 – Hostel Information with Modify Rules | Figure 3.3.5.6 – Hostel Information with 'Edit Hostel Rules' Pop-up Window |
| The user can click **'Modify Rules'** to edit hostel details. | An **'Edit Hostel Rules'** **pop-up window** appears, allowing the user to make modifications. |
| Figure 3.3.5.7 – Hostel Information with Save Changes | Figure 3.3.5.8 – Hostel Information with 'Success' Pop-up Window |
| The user can click **'Save Changes'** to save the newest changes. | A **'Success'** **pop-up window** appears, indicating that the changes have been saved successfully. |
| Figure 3.3.5.9 – Hostel Information with 'Success' Pop-up Window | Figure 3.3.5.10 – Hostel Rules & Regulations |
| The user clicks the **'OK' button**. | Now, the user can view the **updated Hostel Rules & Regulations**. |
| Figure 3.3.5.11 – Hostel Rules & Regulationa with the Back Icon Button | **A screenshot of a computer  Description automatically generated**  Figure 3.3.5.12 – Manager Menu |
| When the user **hovers** over the **back arrow button**, a **tooltip** will appear. **Clicking** the back arrow button will **navigate back** to the **Manager Menu**. | The user is brought back to the **Manager Menu**. |

### 3.3.6 Profile

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated  Figure 3.3.6.1 – Manager Menu | Figure 3.3.6.2 – Manager Profile |
| The user clicks on **'Profile'** to navigate to the Manager Profile page. | The user will be brought to the **Manager Profile page**. |
| Figure 3.3.6.3 – Manager Profile with Edit Button | Figure 3.3.6.4 – Manager Profile with Edit Enabled |
| The user clicks the **'Edit' button** to **modify the account details**. | The user can directly modify the information in the fields. However, the **User ID** **cannot be modified**. |
| Figure 3.3.6.5 – Manager Profile with Full Name Field | Figure 3.3.6.6 – 'Validation Error' Pop-up Window |
| The user enters an **invalid full name**. Full Name can **only contain alphabets**. | A **'Validation Error' pop-up window** appears, indicating **invalid Full Name**. |
| Figure 3.3.6.7 – Manager Profile with Email Field | Figure 3.3.6.8 – 'Validation Error' Pop-up Window |
| The user enters an **invalid email format**. Email address must be in a valid format, containing "@" and following a proper domain structure. | A **' Validation Error' pop-up window** appears, indicating **invalid Email format**. |
| Figure 3.3.6.9 – Manager Profile with the Eye Icon Button | Figure 3.3.6.10 – Manager Profile with the Visible Password |
| User can click the **eye icon button** to view the **hidden password**. | The password will then be **visible**. |
| Figure 3.3.6.11 – Manager Profile with the Lock Icon Button | Figure 3.3.6.12 – Manager Profile with the Hidden Password |
| User can click the **lock icon button** to **hide** the **password**. | The password will then be **hidden**. |
| Figure 3.3.6.13 – Manager Profile with Password Field | Figure 3.3.6.14 – 'Validation Error' Pop-up Window |
| The user enters an **invalid password** **format**. Password must be at least 8 characters long, include at least one uppercase letter, contain at least three digits, and must not have any spaces. | A **' Validation Error' pop-up window** appears, indicating **invalid Password format**. |
| Figure 3.3.6.15 – Manager Profile with the Save Button | Figure 3.3.6.16 – 'Success' Pop-up Window |
| If the user **correctly modifies** all fields without an invalid format, they can click the **'Save' button**. | A **'Success' pop-up window** appears, indicating **profile updated successfully**. |
| Figure 3.3.6.17 – 'Success' Pop-up Window | Figure 3.3.6.18 – Manager Profile |
| The user clicks the **'OK' button**. | The Manager Account will be **updated successfully** and will immediately appear in the Manager Profile. |
| Figure 3.3.6.19 – Manager Profile with Back Arrow Button | **A screenshot of a computer  Description automatically generated**  Figure 3.3.6.20 – Manager Menu |
| The user clicks the **back arrow button** to return to the **Manager Dashboard.** | Return to the **Manager Dashboard.** |

## 3.4 Staff Dashboard

### 3.4.1 Update Account

|  |  |
| --- | --- |
| Figure 3.4.1.1: Staff Menu with ‘Update Account’ button | Figure 3.4.1.2: Update Profile Page |
| The **staff** clicks on ‘**Update Account’ button.** | The **staff** go to **Update Profile page.** |
| Figure 3.4.1.3: Update Profile with ‘Save’ Button | Figure 3.4.1.4: Pop-up window “Profile updated Successfully!” |
| The **staff** key in **new** profile data and click **‘Save’ button.** | It will pop up a window that written **“Profile updated Successfully!”** |
| Figure 3.4.1.5: Update Profile with ‘Back’ Button | Figure 3.4.1.6: Staff Menu |
| The **staff** clicks on the **‘Back’ button.** | The system will bring the staff **back to** the **Staff Menu page.** |

### 3.4.2 Make Payment

|  |  |
| --- | --- |
| Figure 3.4.2.1: Staff Menu with ‘Make Payment’ Button | Figure 3.4.2.2: Payment Management Page |
| The **staff** clicks on the **‘Make Payment’ button.** | The **staff** will be bring to **Payment Management page.** |
| Figure 3.4.2.3: Payment Management Page with ‘Search’ Text Field | Figure 3.4.2.4: Searched Result |
| The **staff** use the **search field** to find the **Resident ID**. | The system will **filter the input data** and **list** the **resident information**. |
| Figure 3.4.2.5: Payment Management Page with ‘Change Status’ Button | Figure 3.4.3.6: Pop-up Window “Select new status:” |
| The **staff** choose the specific and resident clicks on the **‘Change Status’ button.** | A **small window** will **pop-up** and written with **“Select new status:”.** |
| Figure 3.4.2.7: Pop-up window with ‘OK’ button | Figure 3.4.2.8: Pop-up window “Status Update Successfully.” |
| The **staff** change the **‘Unpaid’** status to **‘Paid’** status and click **‘OK’ button**. | The **status** of the **resident** will update to ‘**Paid’** and **pop-up a small window** that written **“Status Update Successfully.”**. |
| Figure 3.4.2.9: Pop-up window with ‘Cancel’ Button | Figure 3.4.2.10: Payment Management Page |
| The **staff** clicks on the **‘Cancel’ button**. | The **pop-up window** will **close** and there is **no change** of the ‘**Status’.** |
| Figure 3.4.2.11: Payment Management Page with ‘Generate Payment’ Button | Figure 3.4.2.12: Generate Receipt Page |
| The **staff** clicks on the **‘Generate Payment’ button**. | The **staff** will bring to the **‘Generate Receipt’ page**. |
| Figure 3.4.2.13: Payment Management Page with ‘Back’ Button | Figure 3.4.2.14: Staff Menu |
| **‘Back’ button** clicked by the **staff**. | It will back to the **Staff Menu page.** |

### 3.4.3 Generate Receipt

|  |  |
| --- | --- |
| Figure 3.4.3.1: Staff Menu with ‘Generate Receipt’ Button | Figure 3.4.3.2: Generate Receipt Page |
| The **staff** clicks the **‘Make Payment’ button**. | The staff will be bringing to the **Generate Receipt page**. |
| Figure 3.4.3.3: Generate Receipt Page with ‘Generate’ Button | Figure3.4.3.4: Pop-up window “Receipt saved successfully!” |
| The **staff** clicks the **‘Generate’ button** after filled in the resident’s information. | It will **pop-up** a **small window** that written **“Receipt saved successfully!”**. This proves that the **receipt** has been **generated** and **saved**. |
| Figure 3.4.3.5: Generate Receipt Page with ‘Print’ Button | Figure 3.4.3.6: Pop-up Window of Print Setting |
| The **staff** clicks the **‘Print’ button**. | A **print setting window** will **pop-up** for **staff** to **set** it. |
| Figure 3.4.3.7: Pop-up Window with ‘Print’ Button | Figure 3.4.3.8: Printout of the Receipt |
| The **staff** clicks on the **‘Print’ button**. | The **receipt** will be **printed out** at the **selected location** that had been **set** in the **Print Setting window**. |
| Figure 3.4.3.9: Print Setting Window with ‘Cancel’ Button | Figure 3.4.3.10: Generate Receipt Page |
| The **‘Cancel’ button** clicked by the **staff**. | The **Print Setting pop-up window** **closed** and **remain** on the **Generate Receipt page**. |
| Figure 3.4.3.11: Generate Receipt Page with ‘Reset’ Button | Figure 3.4.3.12: The Filled Data Cleared in Generate Receipt Page |
| The **staff** clicks the **‘Reset’ button**. | **All** of the **filled data** have been **clear**. |
| Figure 3.4.3.13: Generate Receipt Page with ‘Back’ Button | Figure 3.4.3.14: Staff Menu |
| The **staff** clicks the **‘Back’ button**. | It will **back** to the **Staff Menu**. |

### 3.4.4 Hostel Information

|  |  |
| --- | --- |
| Figure 3.4.4.1: Staff Menu with ‘Hostel Information’ | Figure 3.4.4.2: Hostel Information Page |
| The **staff** choose **‘Hostel Information’ button**. | It will go to **Hostel Information Page**. The staff can **scroll** to see the information. |
| Figure 3.4.4.3: Hostel Information Page with ‘Arrow’ Icon | Figure 3.4.4.4: Staff Menu |
| The **staff** clicks on the **‘Arrow’ icon**. | It will **back** to the **Staff Menu**. |

### 3.4.5 Rules & Regulations

|  |  |
| --- | --- |
| Figure 3.4.5.1: Staff Menu with ‘Rules & Regulations’ | Figure 3.4.5.2: Hostel Rules & Regulations Page |
| The **staff** choose **‘Rules & Regulations’**. | **Hostel Rules & Regulations page** will **pop-up**, **staff** can **scroll** to see it. |
| Figure 3.4.5.3: Hostel Rules & Regulations with ‘Arrow’ Button | Figure 3.4.5.4: Staff Menu |
| The **staff** clicks the **‘Arrow’ button**. | It will **back** to the **Staff Menu**. |

### 3.4.6 Logout

|  |  |
| --- | --- |
| Figure 3.4.6.1: Staff Menu with ‘Home’ Button | Figure 3.4.6.2: Confirm Logout Pop-up Window |
| The **staff** clicks the **‘Home’ button**. | There will have a **Confirm Logout Pop-up Window**. |
| Figure 3.4.6.3: Pop-up Window with ‘Yes’ Button | Figure 3.4.6.4: Login Page |
| The **staff** click the **‘Yes’ button**. | It will bring the **staff** to the **Login Page**. |
| Figure 3.4.6.5: Pop-up Window with ‘No’ Button | Figure 3.4.6.6: Staff Menu |
| The **staff** clicks the **‘No’ button**. | The **pop-up window** will **close** and **remain** on the **Staff Menu**. |

## 3.5 Residents Dashboard

3.5.1 Update Account

|  |  |
| --- | --- |
| Figure 3.5.1.1: Resident Menu with ‘Update Account’ | Figure 3.5.1.2: Update Profile Page |
| The **resident** click on the **‘Update Account’ button**. | The **resident** will be bringing to the **Update Profile page.** |
| Figure 3.5.1.3: Update Profile Page with ‘Save’ Button | Figure 3.5.1.4: Pop-up Window “Profile updated successfully!” |
| The **resident** enters the input in the text field and click **‘Save’ button**. | It will **pop-up a window** that written **“Profile updated successfully!”** to let the **resident** knows that **profile** has been **updated**. |
| Figure 3.5.1.5: Updated Profile with ‘Back’ Button | Figure 3.5.1.6: Staff Menu |
| The **resident** clicks the **‘Back’ button**. | It will go **back** to the **Staff Menu**. |

### 3.5.2 View Payment Records

|  |  |
| --- | --- |
| Figure 3.5.2.1:Resident Menu with ‘View Payment Records’ | Figure 3.5.2.2: Payment Records Page |
| The **resident** choose the **‘View Payment Records’ button**. | It will turn to **Payment Records Page**. |
| Figure 3.5.2.3: Payment Records with ‘View Receipt’ Button | Figure 3.5.2.4: Receipt Shown |
| The **resident** click on the **‘View Receipt’ button**. | The **receipt** will **display** in the **area**. |
| Figure 3.5.2.5: Payment Records with ‘Back’ Button | Figure 3.5.2.6: Resident Menu |
| The **resident** clicks on the **‘Back’ button**. | The **resident** will be bringing **back** to the **Resident Menu**. |

3.5.3 Hostel Information

|  |  |
| --- | --- |
| Figure 3.5.2.7: Resident Menu with ‘Hostel Information’ | Figure 3.5.2.8: Hostel Information Page |
| The **‘Hostel Information’ button** clicked by the **resident**. | The **Hostel Information page** **shown** and the **resident** can **scroll** the page to see more information. |
| Figure 3.5.2.9: Hostel Information Page with ‘Arrow’ | Figure 3.5.2.10: Resident Menu |
| The **resident** click on the **‘Arrow’** on the **top left** of the **Hostel Information Page**. | It will bring the **resident** **back** to **Resident Menu**. |

3.5.4 Rules & Regulations

|  |  |
| --- | --- |
| Figure 3.5.4.1: Resident Menu with ‘Rules & Regulations’ | Figure 3.5.4.2: Hostel Rules & Regulations Page |
| The **resident** chooses **‘Rules & Regulations’**. | There will have a **Hostel Rules & Regulations page** will **pop-up**. The **resident** can **scrolls** to see. |
| Figure 3.5.4.3: ‘Arrow’ Icon in Hostel Rules & Regulations Page | Figure 3.5.4.4: Resident Menu |
| The **staff** clicks **‘Arrow’ icon**. | The **staff** will be bring **back** to the **Resident Menu**. |

### 3.5.5 Logout

|  |  |
| --- | --- |
| Figure 3.5.5.1: ‘Home’ Icon in Resident Menu | Figure 3.5.5.2: Confirm Logout Pop-up Window |
| The **resident** clicks on the **‘Home’ icon**. | It will **pop-up** a **Confirm Logout Window**. |
| Figure 3.5.5.3: ‘Yes’ button in Confirm Logout Pop-up Window | Figure 3.5.5.4: Login Page |
| The **resident** clicks **‘Yes’ button**. | It will **logout** and **go** to **Login Page**. |
| Figure 3.5.5.5: ‘No’ Button in Confirm Logout Pop-up Window | Figure 3.5.5.6: Resident Menu |
| The **resident** choose **‘No’ button**. | The pop-up **Confirm Logout window** will **close** and **remain** on **Resident Menu**. |

# **Program source code**

## **Object-oriented Concepts**

|  |  |
| --- | --- |
| **Classes** |  |
|  | The **ManagerDashboardPanel class** is a blueprint for creating manager dashboard objects in a Hostel Management System. It **inherits** from **JPanel**, gaining its properties and behaviors. The class has **attributes** like **managerId** and **contentPanel**, defining the **state** of each object. By grouping related data and behavior, it follows **encapsulation**, ensuring a structured and reusable design. |
| **Inheritance** |  |
|  | The line **“public class ManagerDashboardPanel extends JPanel {”** demonstrates **inheritance** where ManagerDashboardPanel inherits from JPanel, a built-in Swing class. This allows ManagerDashboardPanel to function as a graphical panel while reusing JPanel's existing properties and methods, such as layout management and component handling. |
| **Constructor** |  |
|  | The constructor public *ManagerDashboardPanel(HostelManagementSystem mainWindow)* is a custom constructor that initializes the ManagerDashboardPanel with a reference to the main application: **mainWindow**. It assigns a **default manager ID: AD123456** and calls the initializeUI() method to set up the panel’s user interface, including the header and main content areas. This constructor ensures that the panel is properly linked to the main system and ready for interaction when instantiated. |
| **Static Members** |  |
|  | This code uses **static members** to define shared instances of UserService, RateService, FileHandler, and Logger. Since they are static, these variables belong to the class rather than individual instances, ensuring only one instance exists for each. |
| **Method Overriding** |  |
|  | Method overriding occurs in the **MouseAdapter** implementation for the **forgotPasswordLink** component. The **mouseEntered** and **mouseExited** methods from the MouseAdapter class are overridden to change the color of the link when the mouse hovers over it. Specifically, in the mouseEntered method, the link's color is darkened: *forgotPasswordLink.setForeground(LINK\_COLOR.darker());,* and in the mouseExited method, it is reverted back to its original color: *forgotPasswordLink.setForeground(LINK\_COLOR);*. |
| **Method Overloading** |  |
|  | This code demonstrates method overloading through **constructor overloading** in the **RateDialog** class. There are two constructors with the same name but different parameter lists. The **first constructor** takes only **owner** and **title** as parameters and calls the second constructor using **this(owner, title, null, null)**, effectively passing **null** for **rateId** and **rateType**. The **second constructor** accepts four parameters **(owner, title, rateId, rateType)** and initializes the **RateDialog** by setting up a **RateService**, **FileHandler**, and storing the rate details. It also calls **initializeUI()** to set up the user interface, and if **editRateId** is not **null**, it loads existing rate data. This approach provides flexibility, allowing the **RateDialog** to be created either with or without specific rate information. |
| **Constructor Overloading** |  |
|  | The **RateDialog** constructor initializes a modal dialog for creating or editing rates. It sets the dialog’s modality and assigns the provided **rateId** and **rateType** to instance variables. It also initializes **rateService** and **fileHandler** for managing data. The **initializeUI()** method sets up the interface, and if **editRateId** is not null, **loadRateData()** is called to prefill the form for editing. This allows the dialog to handle both new and existing rates efficiently. |
|  | **Private Fields**  The RegistrationPanel class defines **private instance variables to** encapsulate data and maintain modularity. The **mainWindow** and **userService** fields are final references used for system interactions and user validation. The input fields: **userIdField, passwordField, fullNameField, emailField, and securityKeywordField** store user registration details. Keeping these fields private ensures that data is only accessed and modified within the class. |
|  | **Private Methods**  The **validateInputs()** method is a private helper function responsible for ensuring that user-provided registration details meet predefined validation criteria. It checks the validity of the full name, user ID, password, email, and security keyword using **userService** methods. If any input is invalid, it displays an error message via **JOptionPane** and returns **false**, preventing further processing. |
| **Exception handling** |  |
|  | The exception handling in the **handleRegistration()** method ensures that registration errors are managed effectively using **try-catch blocks**. When a user attempts to register, the method calls **userService.registerUser(),** which may throw exceptions if there are issues with the input or registration process. The method specifically catches **MaxEmailAttemptsExecption** and **InvalidEmailExecption**, which likely indicate email-related registration failures, and displays an appropriate error message. Additionally, it catches **IllegalArgumentException**, which handles general validation errors, ensuring users receive meaningful feedback. |
| **Polymorphism** |  |
|  | In the **handleSave()** method, different **User** objects (Staff or Resident) are created based on the **selected user type**. This uses **method overriding**, as both extend the User class and may have their own method implementations. The code determines the user type dynamically using **userTypeCombo** and instantiates the appropriate subclass. This demonstrates polymorphism by allowing a single class (User) to take different forms (Staff and Resident) while handling them uniformly. |
| **Data Abstraction** |  |
|  | The private fields (generalConduct, checkInOut, latePaymentPolicy, importantNotes) **hide** the internal representation of hostel rules from external classes. Instead of directly accessing these arrays, external classes must use **getter methods** (getGeneralConduct(), getCheckInOut(), etc.) and the updateRules() method to modify them. This abstraction ensures that the data is **encapsulated**, preventing unauthorized modifications and enforcing controlled access. |
| **Protected Access Modifier** |  |
|  | In the User class, the **protected access modifier** is used for userId, password, fullName, and email, allowing access within the class and its subclasses. This restricts access from outside but permits child classes to inherit and modify these fields. |
| **Interface** |  |

|  |  |
| --- | --- |
|  | This Java Swing code snippet adds an **action listener** to a toggle button: **togglePassword** that controls the visibility of a password field: **passwordField**. When the button is selected, it sets the password field’s echo character to (char) 0, making the password visible, and updates the button’s text to "🔒" with a tooltip "Hide Password". When deselected, it restores the echo character to '•' to mask the password, updates the button’s text to "👁", and changes the tooltip to "Show Password". |
| **Constants** |  |
|  | The code uses the **Constants** concept by defining **RENTAL\_PREFIX**, **UTILITY\_PREFIX**, and **PENALTY\_PREFIX** as **static final** fields, each holding a predefined value ("R", "U", "P") for different rate types. In the **switch** statement, the appropriate constant is assigned based on the **rateType**, ensuring consistent and maintainable usage of these prefixes throughout the code. |

## **Java Features**

|  |  |
| --- | --- |
| **Input Validation** |  |
|  | The input validation logic is implemented in the **validateInput()** method, which ensures that all fields: **User ID, Full Name, Email, and Password** are filled before proceeding. It checks that the User ID prefix corresponds to the selected user type ("ST" for Staff and "TP" for Resident) and verifies that the email follows a valid format using *userService.isValidEmail()*. If any validation fails, an error message is displayed using *JOptionPane.showMessageDialog()*, preventing invalid data from being processed. This method is called within **handleSave()** to validate input before creating or updating a user. |
| **Data Validation** |  |
|  | The data validation in the code occurs within the **saveChanges()** method, specifically in the section that checks user input before saving profile updates. This method **validates four fields**: fullName, email, password, and securityKeyword, ensuring that none are left empty using **trim().isEmpty()**. If any field is blank, an error message is displayed via **JOptionPane.showMessageDialog()**, preventing invalid submissions. Additionally, userService.isValidFullName(fullName), userService.isValidEmail(email), and userService.isValidPassword(password) are used to **enforce specific formatting rules**, ensuring that the full name, email, and password meet predefined criteria. If any validation fails, an appropriate error message is shown, and the method exits without saving changes. |
| **String Formatting** |  |
|  | In this line: *content.append(String.format("%d. %s\n", i + 1, rules.getGeneralConduct()[i]));*, **String.format** is used to ensure that each item in the **generalConduct** array is consistently formatted. The format **"%d. %s\n"** is a placeholder pattern, where **%d** represents the number (index + 1), and **%s** represents the string value (the specific general conduct rule). This method ensures that the output follows a uniform structure, making it easier to read and display the rules consistently. |
| **Array List** |  |
|  | In the searchResidents() method, **ArrayList** is used to store and filter residents. The line *List<Resident> allResidents = getAllResidents();* initializes an ArrayList to hold all residents, while *List<Resident> filteredResidents = new ArrayList<>();* creates an empty ArrayList for matching residents. Finally, *filteredResidents.add(resident);* adds matching residents to the list, demonstrating how ArrayList is used for dynamic data storage and filtering. |
| **Chained Method Calls** |  |
|  | **setBackground(), setForeground(), and setFont()** return the **button** object, allowing multiple calls in a single statement. This follows the **fluent interface pattern**, improving **readability** and reducing repetitive code. |
| **Multi-catch Exception Handling** |  |
|  | The line *catch (MaxEmailAttemptsExecption | InvalidEmailExecption e)* uses **multi-catch exception handling**, an **OOP concept** that improves **code efficiency and readability** by handling multiple exceptions in one block. It promotes **polymorphism** by treating different exceptions similarly and ensures **encapsulation** by providing user-friendly error messages without exposing internal details. |
| **Control Structures** | The **if-else** statement is fundamental in Java programming for decision-making. The **if statement** used to check whether the **search text** is **empty** (text.trim().isEmpty()). If **true**, it **removes any applied row filters** (sorter.setRowFilter(null);). If **not**, it **applies a case-insentive filter** to the table rows. |

# **Additional features source code**

## **Object-oriented Concepts**

|  |  |
| --- | --- |
| **Anonymous inner classes** |  |
|  | The **anonymous inner class** in the LoginPanel is utilized for adding a **MouseAdapter** to the forgotPasswordLink component. By creating an unnamed instance of MouseAdapter, the code overrides methods like mouseClicked, mouseEntered, and mouseExited directly within the listener definition. This allows the link to trigger the handleForgotPassword() method when clicked and change its appearance on hover, providing interactive feedback to users. |
| **Event Handling** |  |
|  | The **filter dropdown (filterCombo)** lets the manager filter user registrations by type, like "All Users," "Customer," or "Admin." When a selection is made, an ActionListener triggers the **applyFilter()** method, which applies a **RowFilter** to show only matching rows based on user type (column 3). If "All Users" is selected, the filter is removed. This ensures quick and dynamic filtering for better data management. |
| **Data Filtering** |  |
|  | This code enables **real-time data filtering** in a Java Swing **JTable** using a **JTextField**. A **DocumentListener** is added to **searchField**, triggering **search()** whenever the text changes. The search() method calls *applyFilter(searchField.getText()),* likely applying a **row filter** to update the displayed table rows dynamically. This allows instant search functionality, making data retrieval faster. |
| **UI Component Organization** |  |
|  | The **createMainPanel()** method organizes the UI modularly by defining the main content panel separately for clarity and maintainability. It uses a **JPanel** with a **GridLayout(2,3)** to arrange buttons neatly, ensuring spacing and symmetry. Styled buttons with icons and text provide a consistent look, with two hidden buttons maintaining layout balance. Action listeners enable smooth navigation, keeping UI logic separate from functionality for easier management and scaling. |
| **File I/O Operations** |  |
| File Path Creation    File Existence Check    Reading File Content | The **File I/O operations** in **viewReceiptActionPerformed()** begin by constructing the file path using the student ID input **(studentIdReceipt + "\_receipts.txt").** It first checks if the file exists; if not, an error message is displayed, and the receipt area is cleared. If the file exists, it is opened using a **BufferedReader**, and its contents are read line by line, appending each line to a **StringBuilder**. The final receipt content is then displayed in the **receiptArea**. If an **IOException** occurs, an error message is shown, and the text area is cleared, ensuring smooth error handling and user feedback. |
| **Interface** |  |
|  | This Java Swing code snippet adds an **action listener** to a toggle button: **togglePassword** that controls the visibility of a password field: **passwordField**. When the button is selected, it sets the password field’s echo character to (char) 0, making the password visible, and updates the button’s text to "🔒" with a tooltip "Hide Password". When deselected, it restores the echo character to '•' to mask the password, updates the button’s text to "👁", and changes the tooltip to "Show Password". |
| **Dependency Injection Pattern** |  |
|  | The constructor *public RegistrationPanel(HostelManagementSystem mainWindow)* uses the Dependency Injection (DI) Pattern by injecting the **mainWindow** and **userService** into the **RegistrationPanel** class. This allows the RegistrationPanel to rely on external components, promoting loose coupling, flexibility, and easier testing by avoiding direct creation of dependencies within the class. |
| **Builder Pattern** |  |
|  | The **Builder Pattern** is demonstrated through the use of **StringBuilder** to efficiently concatenate multiple strings. Each line from the file is appended to the StringBuilder, which prevents the creation of multiple intermediate string objects, thus improving performance when handling large data. |
| **Singleton Pattern** |  |
|  | This line ensures only one instance of **HostelRules** is created and shared across the application. Using **static** and **final**, the **hostelRules** object is initialized once and cannot be reassigned, providing controlled access through the **getHostelRules**() method, which prevents unnecessary object creation and ensures consistency. |
| **Observer Pattern** |  |
|  | The **searchField** is the subject, and the **DocumentListener** is the observer. It listens for text changes in the field and triggers the **search()** method in response, automatically updating when the field's content changes. |
| **Command Pattern** |  |
|  | This code demonstrates the **Command Pattern**, where each button (**addButton, editButton, deleteButton**) triggers a specific command (**showAddRateDialog(), showEditRateDialog(), handleDeleteRate()).** The event listeners act as invokers, executing the commands, which decouples the button actions from the logic and allows for easier modification or extension. |
| **Decorator Pattern** |  |
|  | This code demonstrates the **Decorator Pattern** by combining multiple border decorations for a text field. The **setBorder** method applies a compound border, where a line border: **createLineBorder** is layered with an empty border: **createEmptyBorder**. This pattern allows dynamic addition of functionalities, combining two borders to enhance the field's appearance without modifying the original object. |
| **Null Object Pattern** |  |
|  | This code uses the Null Object Pattern by providing a default behavior when the expected image is not found. When **imageURL** is **null**, the code attempts to load the image from the project directory. |
| **Fluent Interface Pattern** |  |
|  | The **add** method is used to add multiple components to a container, with each method call returning the container itself (or a similar object) to allow subsequent method calls to be chained together. For example, *add(Box.createVerticalStrut(20))* is followed by *add(logoLabel)* and so on. This chaining simplifies the code by eliminating multiple statements, making the UI setup more concise. |
| **Factory Pattern** |  |
|  | This code implements the **Factory Pattern**, where the **generateRateId** method generates a rate ID based on the **rateType**. It uses a **switch** statement to select the appropriate prefix (**RENTAL\_PREFIX, UTILITY\_PREFIX, or PENALTY\_PREFIX**) depending on the rate type. The method then formats the **sequence** number with leading zeros, ensuring consistent ID formatting. This pattern abstracts the creation of rate IDs based on type, providing a flexible and clear way to generate IDs. |
| **Service Layer Pattern** |  |
|  | The **getAllResidents()** method follows the Service Layer Pattern by separating business logic from data access. It delegates the file reading to **fileHandler.readFile()**, processes the data into **Resident** objects, and returns them in a list. This abstracts the data access and focuses on business logic within the service layer. |
| **Factory Method Pattern** |  |
|  | The code implements the **Factory Method Pattern**, where the **createUserObject** method serves as a factory for creating different types of User objects based on the userId. By checking the prefix of the **userId** (either "**TP**" for **Resident** or "**ST**" for **Staff**), the method encapsulates the object creation logic and abstracts it from the rest of the code. |
| **Template Method Pattern** |  |
|  | The **initializeFile(String fileName)** method checks if a file exists, creates it if it doesn't, and writes default data. If file creation fails or an error occurs, it handles it gracefully. This is a template method, where the general steps (checking, creating, writing) are fixed, but the file-writing logic can be customized by subclasses. |
| **Single Responsibility Principle (SRP)** |  |
|  | This method follows the Single Responsibility Principle (SRP) by handling only one responsibility—**updating the amount** while ensuring **lastUpdated** is refreshed. It keeps the logic focused and avoids mixing concerns like data persistence or UI updates, making the class easier to maintain and extend. |
| **Open/Closed Principle (OCP)** |  |
|  | The **createStyledButton** method follows the **Open/Closed Principle (OCP)** by allowing button customization **without modifying existing code**. It provides a reusable structure for button styling, and new variations can be added by extending or overriding the method, ensuring flexibility while maintaining stability. |
| **Immutable Fields** |  |
|  | In this code, the **id** and **type fields** are **immutable** because they are only set during object creation and cannot be changed afterward. The lack of setter methods for these fields ensures they remain constant. The only way to access these values is through the getter methods getId() and getType(), making them read-only and ensuring their values stay unchanged throughout the object's lifecycle. |
| **Static nested classes** |  |
|  | The code defines two static nested classes, **RoomImages** and **RoomType**, within the **HostelInfo** class. **RoomImages** holds data about room categories and their image paths, while **RoomType** stores details like room name, monthly rate, and size. Both classes include constructors for initialization and getter methods for accessing their fields. Since they are static, they can be instantiated independently of the outer **HostelInfo** class. |
| **MVC (Model-View-Controller) pattern** |  |
|  | The **ResidentProfilePanel** class follows the **MVC pattern** by acting as the **View**, responsible for rendering the GUI while keeping business logic separate. It demonstrates **encapsulation** by defining private UI components (**UserID, Email, Password, saveButton,** and **backButton**) that are only accessible within the class, ensuring modularity and maintainability. By handling only UI elements and delegating actions, it ensures **separation of concerns**, allowing UI changes without affecting business logic in the **Controller**. |
| **Separation of Concerns (SoC)** |  |
|  | This code follows the Separation of Concerns (SoC) principle by isolating the **business logic** from the **UI layer**. The method **loadManagerProfile()** retrieves user details by calling **userService.getUserById("AD123456")**, ensuring that the **data-fetching logic** resides within the **UserService** class instead of being directly handled in the UI. If a valid User object is returned, the retrieved data is then set into UI components (userIdField, fullNameField, etc.), keeping the UI responsible only for displaying information. |
| **Composition** |  |
|  | This code demonstrates **Java Composition** by using instance variables (cardLayout, mainPanel, loginPanel, registrationPanel, and managerDashboardPanel) to define a **"has-a" relationship** between the containing class and these objects. Instead of inheriting from them, the class **"has"** a CardLayout for managing different views and multiple JPanel-based components (LoginPanel, RegistrationPanel, and ManagerDashboardPanel) for different sections of the UI. |
| **Service Layer Pattern** |  |
|  | The **getFormattedHostelInfo()** method follows the **Service Layer Pattern** by acting as an intermediary between the **data layer** (FileHandler) and the **UI or business logic layer**. It retrieves raw data from **HOSTEL\_INFO\_FILE**, processes it into a structured format, and returns a clean string for display. This promotes **separation of concerns**, ensuring the UI doesn’t interact directly with file handling. |
| **Dependency Injection** |  |
|  | This constructor demonstrates Dependency Injection (DI) by receiving a HostelManagementSystem instance (**mainWindow**) as a parameter instead of creating it internally. This follows the **Constructor Injection** approach, allowing the **ManagerDashboardPanel** to depend on an external **HostelManagementSystem** instance. |
| **Immutability** |  |
|  | Declaring **rateService**, **fileHandler**, **editRateId**, and **editRateType** as final ensures immutability by preventing reassignment after initialization. This ensures stability, and prevents unintended modifications. |
| Control Structures | This code uses a **if-else** statement. The if-else statement is fundamental in Java programming for decision-making. The if statement used to **check whether the search text is empty** (text.trim().isEmpty()). If **true**, it **removes any applied row filters** (sorter.setRowFilter(null);). If **not**, it **applies a case-insentive filter** to the table rows. |

## **Java Features**

|  |  |
| --- | --- |
| **Validation Logic** |  |
| A computer screen shot of a program  AI-generated content may be incorrect. | The validation logic in this method ensures that the user's full name and password **meet specific format requirements** before updating the profile. It first checks if the full name consists of only letters and spaces using *userService.isValidFullName(fullName)*, and if invalid, it displays an error message and stops execution. Similarly, it validates the password using *userService.isValidPassword(password)*, ensuring it meets the required format. If the password fails to meet these criteria, an error message is displayed, and the process is halted. |
| **Static Utility Methods** |  |
| A screen shot of a computer code  AI-generated content may be incorrect. | The getFormattedHostelInfo() method demonstrates Static Utility Methods by being static, allowing it to be called without creating an instance of the class. It reads hostel information from a file, formats the content, and returns it as a string, serving as a utility function that doesn't require object instantiation. |
| **Lambda expression** |  |
| A close-up of a computer code  AI-generated content may be incorrect. | This code uses a **lambda expression** for concise event handling. When **backButton** is clicked, it finds the nearest **ManagerDashboardPanel** using **SwingUtilities.getAncestorOfClass(...)**. If found, it casts and calls **showMainPanel()**, enabling navigation with minimal boilerplate. |
| **Custom Exception Class** |  |
|  | This code defines a custom exception, **InvalidEmailExecption**, by extending the **Exception** class. It allows the creation of specialized exceptions with custom messages, providing more specific error handling for invalid email scenarios. |
| **Enum constant** |  |
|  | In *super(owner, title, ModalityType.APPLICATION\_MODAL);,* **ModalityType.APPLICATION\_MODAL** is an **enum constant** from java.awt.Dialog.ModalityType, setting the dialog to block input to all other application windows until closed. |
| **Generics** |  |
|  | The line *JComboBox<String> userTypeCombo* uses **generics** to ensure only String values are stored, enhancing **type safety** and preventing invalid data. Likewise, TableRowSorter<DefaultTableModel> sorter restricts sorting to DefaultTableModel, reducing type mismatches. |

## **References**

Coding Time. (2017, Dec 31). *fees receipt system in java by using netbeans*. [Video].

YouTube. <https://www.youtube.com/watch?v=0g0uX6rhksE>

GeeksForGeeks. (2021, Nov 3). *Difference Between System.out.println() and*

*System.err.println() in Java*. GeeksForGeeks.

<https://www.geeksforgeeks.org/difference-between-system-out-println-and-system-err-println-in-java/>

GeeksForGeeks. (2024, Dec 24). *Java StringBuilder Class*. GeeksForGeeks.

<https://www.geeksforgeeks.org/stringbuilder-class-in-java-with-examples/>

Teck Updates. (2022, Dec 30). *Search data from JTable in Java using JTextField*. [Video].

YouTube. <https://www.youtube.com/watch?v=haQIw3YDTPg>

Telusko. (2023, Jan 19). *#63 Anonymous Inner Class in Java* [Video]. YouTube.

<https://www.youtube.com/watch?v=LNpUj80JLGI>

W3Schools. (n.d.). Java Date and Time. W3Schools.

<https://www.w3schools.com/java/java_date.asp>

# **Task Division**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** | **TP Number** | **Task** | **Signature** |
| Siew Zhen Lynn | TP076386 | Login  Register  Manager Function | ***celine*** |
| Yeo Pei Wen | TP077057 | Staff Function  Resident Function | ***wen*** |