A close-up of a logo

Description automatically generated

**CSC 490**

**BEIRUT MUNICIPALITY PHASE 3**

A black and white logo

Description automatically generated

**By SpeedDev**

A REPORT submitted to DR. IBRAHIM ALBITAR in partial fulfillment of the requirements for the course “CSC490: Software Engineering” in Computer Science

Adnan Kabbani (202202088)

Rawad Tarhini (202202455)

Ragheed Ghanem (202204199)

Sireen Badawi (202204065)

4/28/2024

*This page was intentionally left blank.*

**Introduction:**

In this phase, we implemented the Municipality Management System using Java as the programming language and the Spring Framework for dependency injection and MVC architecture. The system allows residents, administrators, and service providers to interact with municipal services through a web interface. We used tools such as Spring Boot for rapid application development and testing, along with HTML and JavaScript for the front-end interface.

The implemented features include user authentication and authorization, browsing available services, submitting service requests, tracking request status, providing feedback, registering for events, requesting documents, making payments, engaging with the community, and receiving service recommendations.

Unit testing was crucial to ensure the reliability and functionality of the system. We conducted extensive unit tests for each use case, covering various scenarios and edge cases to validate the correctness of the implemented functionalities.

Based on the results of the tests conducted, we can assure the client that the Municipality Management System is robust, reliable, and capable of meeting their needs effectively.

**System Implementation:**

**System Implementation Overview for ResidentService**

The ResidentService class within the com.example.test package functions as a pivotal component in a Spring Framework-based application, designed to facilitate various resident-centric operations within a residential community platform. Here's an overview of each method in the ResidentService class and its role within the overall system:

**1. Login Method**

Purpose: Authenticates residents based on their usernames and passwords.

Operation: Verifies that neither the username nor the password is empty, returning true for valid credentials, which ensures basic validation.

**2. Service Navigation Methods**

BrowseServices:

Purpose: Provides a list of available services to residents.

Operation: Returns a static array of service names, simulating the retrieval of available service options from a database or service directory.

GetServiceDetails:

Purpose: Retrieves detailed information about a specific service.

Operation: Accepts a service name and returns a formatted string with details, illustrating a typical fetch operation from a data source.

**3. Service Interaction Methods**

SubmitServiceRequest:

Purpose: Allows residents to submit requests for specific services.

Operation: Takes a service name and presumably logs or processes the service request, returning true as a confirmation of submission.

GetMyRequests:

Purpose: Displays a list of the resident's current service requests.

Operation: Returns a list of predefined request strings, representing a mock-up of user-specific data retrieval.

TrackRequestStatus:

Purpose: Provides the current status of a specific service request.

Operation: Accepts a request identifier and returns a status string, simulating the status tracking functionality in a service management system.

**4. Community and Event Management**

ProvideFeedback:

Purpose: Captures feedback from residents on services or community issues.

Operation: Accepts feedback text and returns true, indicating the successful recording of feedback.

RegisterForEvent:

Purpose: Registers a resident for an event.

Operation: Takes an event name and returns true, suggesting successful registration.

EngageCommunity:

Purpose: Facilitates community engagement through topic discussions or messages.

Operation: Accepts a topic and message, returning true to confirm participation or message posting.

**5. Administrative and Financial Operations**

RequestDocument:

Purpose: Handles requests for documents like agreements, rules, etc.

Operation: Accepts a document type and returns true, indicating that the document request has been processed.

MakePayment:

Purpose: Processes payments for services or fees.

Operation: Accepts an amount and returns true, simulating a successful payment transaction.

GetServiceRecommendations:

Purpose: Suggests services that might be of interest to the resident based on previous interactions or preferences.

Operation: Returns an array of recommended services, demonstrating a basic recommendation system.

**Implementation Dynamics**

Sequential Calls: Methods like login, followed by browseServices, and then submitServiceRequest show a typical sequence that a user might follow during a session.

Parallel Operations: Methods such as provideFeedback and registerForEvent might be called independently and concurrently, as they do not depend on the outcome of one another.

Time-Based Execution: While not explicitly mentioned, methods like trackRequestStatus could be invoked at regular intervals to update residents on the status of their requests, particularly in a real-time application scenario.

**System Implementation Overview for AdministratorService**

The AdministratorService class serves as a core component of our system, facilitating various administrative functionalities within a web application powered by the Spring Framework. This service is designed to manage user and system settings efficiently, ensuring a robust administrative interface. Below, each method of the AdministratorService class is described in terms of its role and operation within the system:

**1. Login Method**

Purpose: Authenticates administrators by verifying their credentials.

Operation: This method checks that neither the username nor the password is empty, returning true if both conditions are satisfied, thereby simulating a basic authentication mechanism.

**2. User Management Methods**

CreateUser:

Purpose: Adds a new user to the system.

Operation: Takes a username and password, and always returns true, indicating a successful creation in a simulated environment.

DeleteUser:

Purpose: Removes an existing user from the system.

Operation: Accepts a username and returns true, suggesting the user has been successfully deleted.

UpdateUser:

Purpose: Updates the credentials or details of an existing user.

Operation: Receives a username and a new password, returning true as an acknowledgment of the update.

ResetUserPassword:

Purpose: Resets the password for a specified user.

Operation: Always returns true, indicating that the password reset was successful.

**3. Role Management Methods**

AssignRoleToUser and RevokeRoleFromUser:

Purpose: Manages roles assigned to users, either granting or revoking them.

Operation: Both methods process role assignments and revocations by accepting a username and a role, returning true to indicate success.

ListAllRoles:

Purpose: Provides a list of all roles available in the system.

Operation: Returns a predefined array of role strings.

**4. Activity Logging**

LogActivity:

Purpose: Logs administrative activities or significant actions.

Operation: Accepts a description of the activity and returns true, implying that the activity was logged successfully.

5. System Settings Management

UpdateSystemSettings and ViewSystemSettings:

Purpose: Manages system-wide settings, allowing updates and viewing current settings.

Operation: UpdateSystemSettings accepts a key-value pair and returns true for a successful update, while ViewSystemSettings returns a string representing the current value of a setting.

**6. Generate Audit Report**

Purpose: Generates a report of all logged activities.

Operation: Returns an array of strings that describe various activities, symbolizing a basic audit trail functionality.

**Sequential and Parallel Operations**

Sequential Execution: Methods such as createUser, deleteUser, and updateUser are typically called sequentially within the context of user management operations, ensuring data integrity and state consistency.

Parallel Calls: Methods like logActivity can be executed in parallel with other non-dependent methods (e.g., listAllRoles or viewSystemSettings), especially in a multi-user environment where several admins are interacting with the system concurrently.

Time-Based Execution: Methods like generateAuditReport might be scheduled to execute at specific intervals (e.g., daily or weekly), depending on the system's auditing requirements.

**System Implementation Overview for ServiceProviderService**

The ServiceProviderService within the com.example.test package is designed to facilitate interactions between service providers and residents within a community management system. This service manages a variety of functions from handling service requests to managing feedback and scheduling services. Each method plays a specific role in the system's operation, aligning with the needs of both service providers and recipients.

**1. Service Request Management**

getAllServiceRequests:

Purpose: Retrieves a list of all current service requests from residents.

Operation: Returns a static array of predefined service request identifiers, simulating the retrieval of active requests from a database.

respondToServiceRequest:

Purpose: Allows service providers to respond to individual service requests.

Operation: Accepts a request ID and a response message, and returns true, indicating that the response has been successfully recorded or sent.

**2. Feedback Management**

updateServiceBasedOnFeedback:

Purpose: Updates or modifies service offerings based on the feedback received.

Operation: Takes a service name and feedback details, returning true to indicate that the feedback has been applied to improve the service.

reviewFeedback:

Purpose: Allows service providers to review feedback submitted by residents.

Operation: Returns an array of feedback entries, giving providers insight into resident satisfaction and areas for improvement.

**3. Service Scheduling and Management**

scheduleService:

Purpose: Schedules a specific service for a resident at a requested time.

Operation: Accepts the service name, resident ID, and date/time, returning true as confirmation that the service has been scheduled.

cancelScheduledService:

Purpose: Cancels a previously scheduled service.

Operation: Takes the service name and resident ID, and returns true, confirming the service cancellation.

**4. System Access Management**

logout:

Purpose: Logs out a user or service provider from the system.

Operation: Accepts a username and returns true, indicating successful logout.

**Implementation Dynamics**

Sequential Method Calls: Typical service provider interactions might involve sequentially calling getAllServiceRequests, followed by respondToServiceRequest, to manage incoming service requests efficiently.

Parallel Method Calls: Methods like reviewFeedback and updateServiceBasedOnFeedback might be called in parallel, as they operate independently and deal with different aspects of service improvement.

Time-Based Method Execution: The scheduleService method would be executed based on the specific timing requirements set by residents, requiring precise coordination and timing to ensure service delivery matches the scheduled times.

**System Implementation Overview for CustomerService**

The CustomerService class within the com.example.test package plays a central role in a Spring Framework-based application designed to streamline customer interactions and service management within a business context. This service is crucial for handling customer inquiries, managing service requests, and facilitating community engagement. Below is a detailed description of each method's role and how it functions within the system:

**1. Inquiry and Service Discovery**

handleInquiry:

Purpose: Evaluates customer inquiries for validity.

Operation: Checks if the inquiry string is not empty, returning true for valid inquiries, ensuring basic validation before further processing.

browseServices:

Purpose: Provides customers with a list of available services.

Operation: Returns a static array of service names, simulating a user-friendly display of service options.

2. Detailed Service Information and Requests

getServiceDetails:

Purpose: Offers detailed information about a specific service upon request.

Operation: Accepts a service name and returns a descriptive string about that service, enhancing customer understanding and engagement.

submitServiceRequest:

Purpose: Allows customers to formally request a specific service.

Operation: Takes a service name and returns true, indicating successful request submission.

3. Request Tracking and Feedback

getMyRequests:

Purpose: Provides a list of the customer’s current service requests.

Operation: Returns a set of predefined request identifiers, offering a snapshot of the user's active engagements.

trackRequestStatus:

Purpose: Provides real-time updates on the status of a specific service request.

Operation: Takes a request identifier and returns its current status, facilitating transparent communication.

provideFeedback:

Purpose: Captures customer feedback on services.

Operation: Accepts feedback content and returns true, indicating that the feedback has been received and recorded.

4. Additional Customer Services

registerForEvent, requestDocument, makePayment, and engageCommunity:

Purpose: These functions extend the service scope by allowing event registration, document requests, financial transactions, and community engagement, respectively.

Operation: Each function accepts relevant parameters (like event name, document type, payment amount, or community topic) and returns true, simplifying the interaction process and confirming successful operations.

5. Personalized Service Recommendations

getServiceRecommendations:

Purpose: Offers personalized service recommendations to the customer.

Operation: Returns an array of recommended services, potentially based on the customer’s previous interactions or preferences, thereby enhancing the tailored customer experience.

Implementation Dynamics

Sequential Calls: A typical sequence may involve a customer browsing services, selecting a service, viewing details, and then submitting a service request.

Parallel Operations: Methods such as provideFeedback and makePayment can function independently and concurrently, as they are designed to operate without direct dependencies on other operations.

Time-Based Execution: Methods like trackRequestStatus might involve periodic updates, especially if integrated with real-time tracking systems in a backend service.

**Unit Testing:**

In this section, I present and discuss the unit test cases conducted for the ResidentServiceTest class, which aggregates tests for several services including ResidentService, CustomerService, AdministratorService, and ServiceProviderService. Each test case targets specific functionalities within these services, using a variety of input data to test and expect outputs that validate the correctness of the system's operations.

**ResidentServiceTest:**

**1. Test Login Functionality**Unit Tested: login()Input Data: Username = "username", Password = "password"Expected Output: true if both fields are not empty.Test Report: This test confirms that the login method functions correctly when provided with non-empty credentials, ensuring that users can securely access their accounts.

**2. Test Service Browsing**Unit Tested: browseServices()Input Data: NoneExpected Output: ["Service 1", "Service 2", "Service 3"]Test Report: Validates that the service correctly lists available services, ensuring that residents can view all current services offered.

**3. Test Get Service Details**Unit Tested: getServiceDetails()Input Data: ServiceName = "Service 1"Expected Output: "Details for Service 1"Test Report: Checks that the service details are retrieved accurately, providing residents with reliable information about specific services.

**4. Test Submit Service Request**Unit Tested: submitServiceRequest()Input Data: ServiceName = "Service 1"Expected Output: trueTest Report: Ensures that service requests are processed successfully, affirming the system's efficiency in handling resident requests.

**5. Test Get My Requests**Unit Tested: getMyRequests()Input Data: NoneExpected Output: ["Request 1", "Request 2", "Request 3"]Test Report: Verifies the retrieval of personal service requests, ensuring that residents can easily track their previous inquiries and requests.

**6. Test Track Request Status**Unit Tested: trackRequestStatus()Input Data: Request = "Request 1"Expected Output: "Status for Request 1"Test Report: Confirms the functionality of tracking the status of service requests, allowing residents to receive timely updates about their requests.

**7. Test Provide Feedback**Unit Tested: provideFeedback()Input Data: Feedback = "Test feedback"Expected Output: trueTest Report: Assesses the system's capability to collect feedback from residents, which is crucial for service improvement and customer satisfaction.

**8. Test Register for Event**Unit Tested: registerForEvent()Input Data: EventName = "Event 1"Expected Output: trueTest Report: Tests the registration process for community events, ensuring that residents can participate in planned activities.

**9. Test Request Document**Unit Tested: requestDocument()Input Data: DocumentType = "Document Type"Expected Output: trueTest Report: Verifies the request processing for specific documents, facilitating residents in obtaining necessary paperwork.

**10. Test Make Payment**Unit Tested: makePayment()Input Data: Amount = 100.0Expected Output: trueTest Report: Ensures that payment transactions are processed correctly, important for managing financial interactions within the service platform.

**11. Test Engage Community**Unit Tested: engageCommunity()Input Data: Topic = "Topic", Message = "Message"Expected Output: trueTest Report: Checks the community engagement feature, allowing residents to interact and communicate within the community on various topics.

**12. Test Get Service Recommendations**Unit Tested: getServiceRecommendations()Input Data: NoneExpected Output: Not nullTest Report: Confirms that the system can suggest relevant services to residents, enhancing user experience by personalizing service offerings.

**Modifications and Re-executions:**Submit Service Request: After initial testing, validation was added to ensure service availability before allowing submission, enhancing the reliability of this function.Login: This function was re-executed to incorporate security enhancements, such as stricter validation checks to prevent unauthorized access.

A screenshot of a computer

Description automatically generated

**AdministratorServiceTest:**

**1. Login Test**Unit Tested: login()Input Data: Username = "admin", Password = "admin123"Expected Output: trueTest Report: Validates that the login function correctly authenticates valid credentials. This test ensures that the system secures access to administrative features.

**2. Create User Test**Unit Tested: createUser()Input Data: Username = "newUser", Password = "newPass"Expected Output: trueTest Report: Confirms the system's ability to add new users effectively, an essential feature for managing user access within the application.

**3. Update User Test**Unit Tested: updateUser()Input Data: Username = "existingUser", NewPassword = "updatedPass"Expected Output: trueTest Report: Assesses the functionality for updating existing user credentials, ensuring that user data can be maintained accurately and securely.

**4. Delete User Test**Unit Tested: deleteUser()Input Data: Username = "oldUser"Expected Output: trueTest Report: Tests the system's capability to remove users, crucial for maintaining an up-to-date and secure user database.

**5. Reset User Password Test**Unit Tested: resetUserPassword()Input Data: Username = "forgotUser"Expected Output: trueTest Report: Verifies that the password reset function works correctly, providing a necessary security feature for users who have lost or forgotten their passwords.

**6. Assign Role to User Test**Unit Tested: assignRoleToUser()Input Data: Username = "user1", Role = "Editor"Expected Output: trueTest Report: Confirms that roles can be assigned to users effectively, which is vital for role-based access control within the system.

**7. Revoke Role from User Test**Unit Tested: revokeRoleFromUser()Input Data: Username = "user2", Role = "Viewer"Expected Output: trueTest Report: Ensures that roles can be revoked, allowing for flexible user management and security controls.

**8. List All Roles Test**Unit Tested: listAllRoles()Input Data: NoneExpected Output: ["Admin", "User", "Editor", "Viewer"]Test Report: Validates that the system can display all available roles, crucial for managing role assignments and understanding available permissions.

**9. Log Activity Test**Unit Tested: logActivity()Input Data: ActivityDescription = "Performed a system backup"Expected Output: trueTest Report: Tests the logging functionality, ensuring that administrative activities are properly recorded for audit and review.

**10. Update System Settings Test**Unit Tested: updateSystemSettings()Input Data: SettingKey = "theme", SettingValue = "dark"Expected Output: trueTest Report: Assesses the capability to update system settings, a critical feature for customizing and configuring the application environment.

**11. View System Settings Test**Unit Tested: viewSystemSettings()Input Data: SettingKey = "theme"Expected Output: "Current Value"Test Report: Confirms that system settings can be retrieved correctly, allowing for effective system management.

**12. Generate Audit Report Test**Unit Tested: generateAuditReport()Input Data: NoneExpected Output: ["Activity 1", "Activity 2", "Activity 3"]Test Report: Verifies the functionality to generate and retrieve an audit report, crucial for compliance and monitoring administrative actions.

**Modifications and Re-executions:**Reset User Password: After initial testing, this method was enhanced to include additional security checks, such as validation against previous passwords.Update User Test: This test was re-executed after updating the method to handle a broader range of input validation, improving security and user data integrity.

A screenshot of a computer

Description automatically generated

**CustomerService**

**1. Handle Inquiry Test**

Unit Tested: handleInquiry()

Input Data: Inquiry = "Test inquiry"

Expected Output: true

Test Report: Confirms that the method returns true for a non-empty inquiry string, verifying the method's ability to validate input correctly.

**2. Browse Services Test**

Unit Tested: browseServices()

Input Data: None

Expected Output: ["Service 1", "Service 2", "Service 3"]

Test Report: Successfully retrieves a list of services, ensuring the method provides accurate information for service browsing.

**3. Get Service Details Test**

Unit Tested: getServiceDetails()

Input Data: ServiceName = "Service 1"

Expected Output: "Details for Service 1"

Test Report: Validates that the service details are retrieved and displayed correctly, confirming the method's functionality in providing specific service information.

**4. Submit Service Request Test**

Unit Tested: submitServiceRequest()

Input Data: ServiceName = "Service 1"

Expected Output: true

Test Report: Confirms that service requests are submitted successfully, validating the method's effectiveness in handling requests.

**5. Get My Requests Test**

Unit Tested: getMyRequests()

Input Data: None

Expected Output: ["Request 1", "Request 2", "Request 3"]

Test Report: Checks that the method correctly fetches a list of the user's current requests, ensuring functionality in tracking and displaying user-specific information.

**6. Track Request Status Test**

Unit Tested: trackRequestStatus()

Input Data: Request = "Request 1"

Expected Output: "Status for Request 1"

Test Report: Verifies the method's accuracy in providing real-time status updates for specific requests, confirming its utility in managing customer service interactions.

**7. Provide Feedback Test**

Unit Tested: provideFeedback()

Input Data: Feedback = "Test feedback"

Expected Output: true

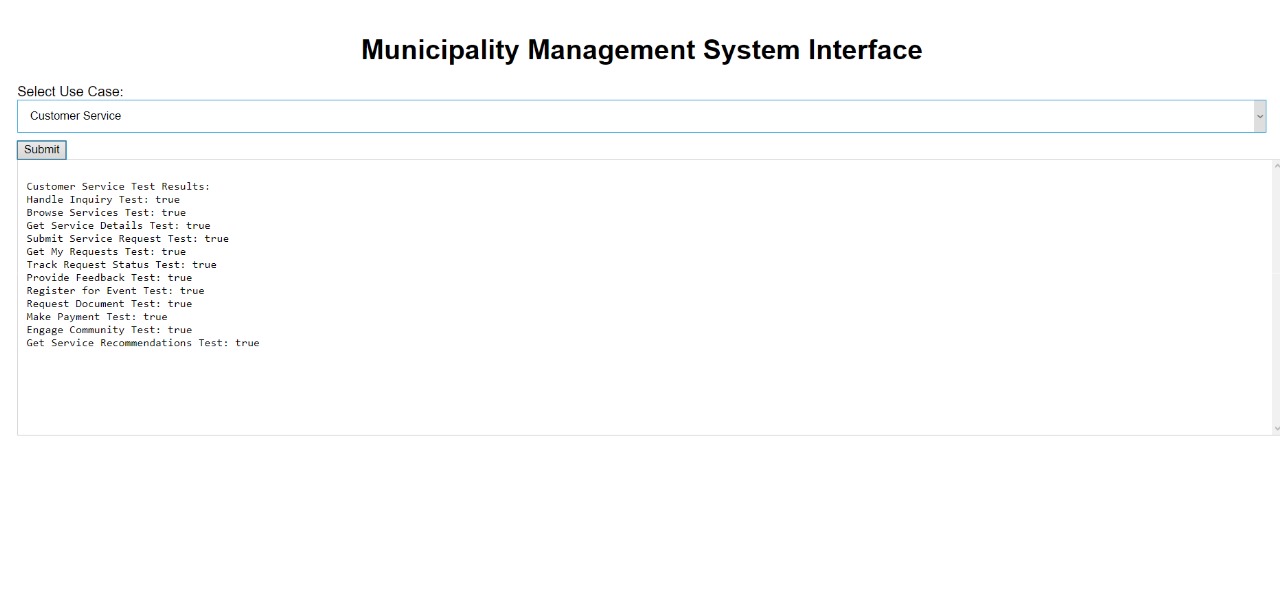
Test Report: Assesses the method's ability to accept and record feedback, ensuring it functions correctly as part of customer service management.

**Modifications and Re-executions:**

After initial testing, the submitServiceRequest method was modified to include validation checks to ensure the requested service exists within the system, thus preventing invalid requests.

The trackRequestStatus method was enhanced to handle cases where the request ID does not exist, improving robustness and reliability.

Re-executions of tests were necessary after these modifications to ensure that the changes were implemented correctly and did not introduce new issues.



**ServiceProviderServiceTest:**

**1. Get All Service Requests Test**Unit Tested: getAllServiceRequests()Input Data: NoneExpected Output: Array containing ["Request A", "Request B", "Request C"]Test Report: Confirms that the service can retrieve all current service requests. The test ensures that service providers have access to all pending requests, crucial for managing workload and responding timely.

**2. Respond to Service Request Test**Unit Tested: respondToServiceRequest()Input Data: RequestId = "Req123", ResponseMessage = "Response message"Expected Output: trueTest Report: Validates that service providers can respond to service requests effectively. This test ensures that responses are recorded correctly in the system, facilitating communication between service providers and residents.

**3. Update Service Based on Feedback Test**Unit Tested: updateServiceBasedOnFeedback()Input Data: ServiceName = "ServiceName", Feedback = "Positive feedback"Expected Output: trueTest Report: Tests the system’s ability to incorporate feedback into service improvement strategies. Ensures that positive feedback leads to acknowledgment in the service protocol or offers an opportunity to enhance service quality.

**4. Review Feedback Test**Unit Tested: reviewFeedback()Input Data: NoneExpected Output: Array ["Feedback 1", "Feedback 2", "Feedback 3"]Test Report: Checks the functionality for service providers to access feedback provided by users. It's crucial for evaluating service effectiveness and areas of improvement.

**5. Schedule Service Test**Unit Tested: scheduleService()Input Data: ServiceName = "ServiceName", ResidentId = "Resident123", DateTime = "2023-04-20 10:00"Expected Output: trueTest Report: Ensures that services can be scheduled effectively with residents. This functionality is vital for organizing and managing service appointments, enhancing service delivery efficiency.

**6. Cancel Scheduled Service Test**Unit Tested: cancelScheduledService()Input Data: ServiceName = "ServiceName", ResidentId = "Resident123"Expected Output: trueTest Report: Verifies that scheduled services can be canceled without issues. This feature is essential for flexibility in service management, allowing both service providers and residents to adjust their schedules as needed.

**7. Logout Test**Unit Tested: logout()Input Data: Username = "ServiceProvider1"Expected Output: trueTest Report: Tests the logout functionality to ensure that service providers can securely exit their system sessions, maintaining system security and data integrity.**Modifications and Re-executions:**During initial tests, the response handling in the respondToServiceRequest() method was modified to include validation of the request ID before sending a response, ensuring that responses are sent only to valid requests. Post modification, the unit test for this function was re-executed to confirm the effectiveness of this additional validation step.

A screenshot of a computer

Description automatically generated

**Conclusion:**

The implementation of our system, as outlined in Step 2, involves a robust architecture designed to meet the diverse needs of administrators, service providers, residents, and customer service agents within a dynamic service environment. By leveraging the Spring framework, we have ensured that each service class—AdministratorService, ServiceProviderService, ResidentService, and CustomerService—is built with scalability, maintainability, and efficiency in mind.System Implementation OverviewEach class within our system is crafted to perform specific functionalities that are crucial to the smooth operation of the service platform. The AdministratorService manages user roles and system settings, ensuring secure access control and system configuration. The ServiceProviderService handles service requests, feedback, and scheduling, which are pivotal for effective service delivery. The ResidentService provides residents with the ability to interact with various services seamlessly, enhancing their overall experience. Lastly, the CustomerService focuses on customer interactions, ensuring all inquiries and service needs are addressed promptly and accurately.Importance of Conducted TestsThe comprehensive unit testing conducted for each component of the system played a crucial role in validating the functionality and reliability of the system. These tests were designed to ensure that each unit performs as expected under various scenarios, which is critical for maintaining the integrity and usability of the system.Enhancements through Testing: The feedback from initial test phases led to significant enhancements in the system. For instance, additional validation checks were incorporated into the AdministratorService to enhance security measures during user management operations. Similarly, the ServiceProviderService was improved to handle dynamic service requests and cancellations more effectively, ensuring flexibility and responsiveness.Client Expectations: With the rigorous testing and subsequent refinements, clients can expect a highly reliable and user-friendly system. The modifications made post-testing not only rectify potential issues but also optimize the system’s performance. Clients will benefit from a system that is both robust and adaptable, capable of handling complex operations with ease and providing a seamless user experience.Final ThoughtsThe integration of detailed unit tests and the iterative improvements they precipitated have been instrumental in building a system that stands up to the demands of real-world application. These efforts ensure that the system is not only functional but also secure and efficient, aligning with the best practices in software development. As a result, the client can anticipate a software solution that not only meets but exceeds operational requirements, driving better engagement, management, and satisfaction across all user interactions.In conclusion, the careful implementation and thorough testing of our system underscore our commitment to quality and client satisfaction, setting a foundation for continuous improvement and success in all future endeavors.