CSE352- Final Project

Fix-ed Technician Service App

(CSE352) Systems Analysis & Design

Khalid Abdelsalam Mohamed 22101310

Abdelrahman Amr Mohamed 21100832

Taher Abdelbary Abdelmalek 21100797

Applied Patterns:

1. Singleton Pattern - Applied to DatabaseConnection

Where:

The DatabaseConnection class is designed using the Singleton pattern to ensure only one instance of the database connection exists during the application's lifecycle.

The DatabaseConnection class has a static field instance to hold the single instance and a static method getInstance() that either creates the instance if it doesn't exist or returns the existing instance.

Why:

Database connections are typically expensive resources, and it's inefficient to create multiple instances of a database connection. The Singleton pattern ensures that only one connection is made and reused throughout the entire application. Ensuring that only one instance of DatabaseConnection exists prevents multiple connections being made to the database, which could be inefficient and problematic for resource management.

The Singleton pattern also allows centralized management of the database connection (e.g., managing connection pooling, opening/closing the connection).

2. Factory Pattern - Applied to ClientHandlerFactory

Where:

The ClientHandlerFactory class is responsible for creating instances of different ClientHandler types (such as UnauthenticatedClientHandler or AuthenticatedClientHandler).

The createClientHandler(client: Client) method defines a factory method to create the appropriate ClientHandler depending on the state of the Client (authenticated or unauthenticated).

Why:

The Factory pattern is useful when there are multiple possible variations of a class (e.g., different types of ClientHandler) and we want to create instances of these classes without exposing the instantiation logic to the client code.

It provides flexibility, making it easier to add new types of ClientHandler without changing existing code in the Server class.

The Factory pattern encapsulates the creation logic, allowing you to switch between different

ClientHandler types based on the context (authenticated or unauthenticated client).

3. Abstract Class - Applied to ClientHandler and its Subclasses

Where:

ClientHandler is an abstract class, and specific client handler types such as UnauthenticatedClientHandler and AuthenticatedClientHandler extend it.

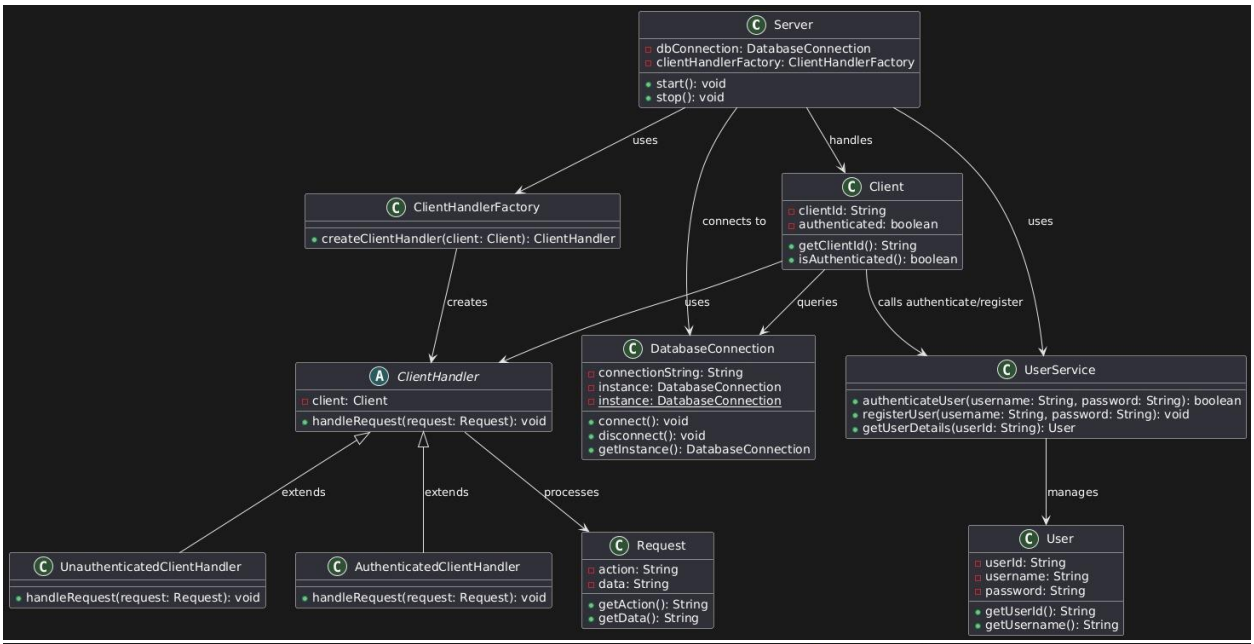
Why:

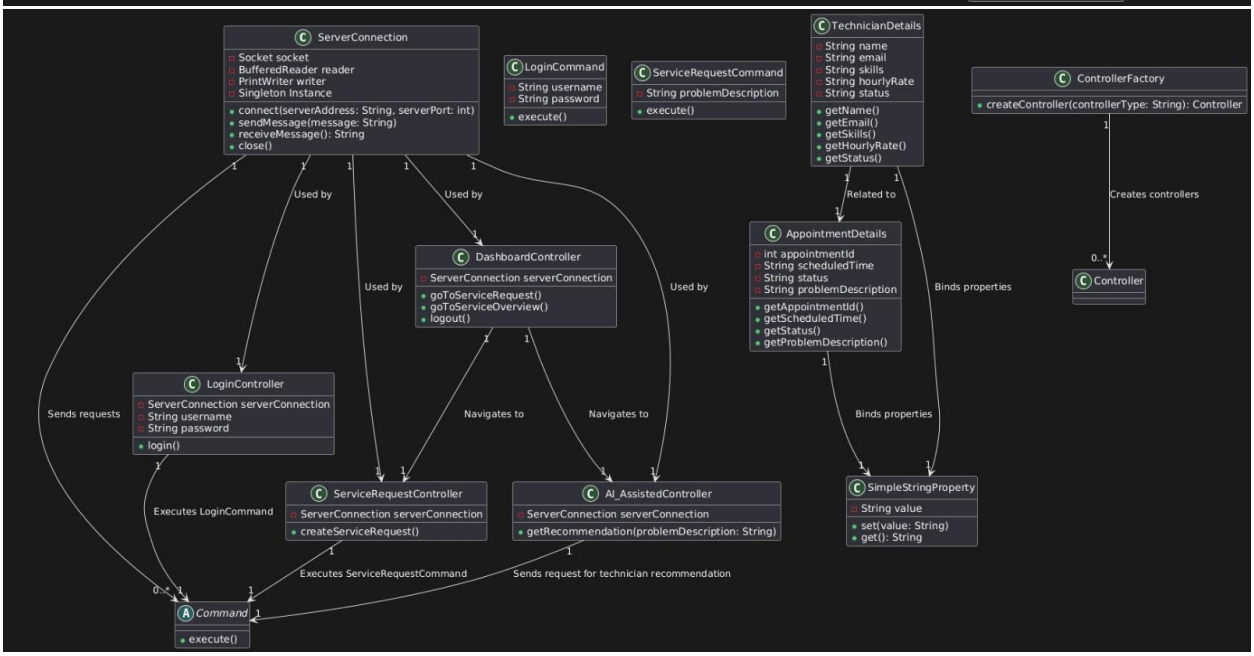
The Abstract class provides a common interface for all types of ClientHandler. By defining the abstract class ClientHandler with common methods (e.g., handleRequest(request: Request)), we allow different concrete classes to implement their own version of the method.

This ensures that the client handler types (authenticated and unauthenticated) follow the same contract, while allowing the flexibility to implement different behavior for each type.

It also promotes polymorphism, where the same method (handleRequest) can behave differently depending on the instance (unauthenticated or authenticated client handler).

1. **Class Diagram(before,After):**


Server
- dbConnection: DatabaseConnection
- clientHandlerFactory: ClientHandlerFactory
+ start(): void
+ stop(): void
uses
handles
ClientHandlerFactory
+ createClientHandler(client: Client): ClientHandler
Client
- clientId: String
- authenticated: boolean
+ getClientId(): String
+ isAuthenticated(): boolean
connects to
uses
creates
uses
queries
calls authenticate/register
ClientHandler
- client: Client
+ handleRequest(request: Request): void
DatabaseConnection
- connectionString: String
- instance: DatabaseConnection
- instance: DatabaseConnection
+ connect(): void
+ disconnect(): void
+ getInstance(): DatabaseConnection
UserService
+ authenticateUser(username: String, password: String): boolean
+ registerUser(username: String, password: String): void
+ getUserDetails(userId: String): User
extends
extends
processes
manages
UnauthenticatedClientHandler
+ handleRequest(request: Request): void
AuthenticatedClientHandler
+ handleRequest(request: Request): void
Request
- action: String
- data: String
+ getAction(): String
+ getData(): String
User
- userId: String
- username: String
- password: String
+ getUserId(): String
+ getUsername(): String


ServerConnection
- Socket socket
- BufferedReader reader
- PrintWriter writer
- Singleton Instance
+ connect(serverAddress: String, serverPort: int)
+ sendMessage(message: String)
+ receiveMessage(): String
+ close()
LoginCommand
- String username
- String password
+ execute()
ServiceRequestCommand
- String problemDescription
+ execute()
TechnicianDetails
- String name
- String email
- String skills
- String hourlyRate
- String status
+ getName()
+ getEmail()
+ getSkills()
+ getHourlyRate()
+ getStatus()
ControllerFactory
+ createController(controllerType: String): Controller
Creates controllers
Controller
Used by
Used by
Used by
Used by
DashboardController
- ServerConnection serverConnection
+ goToServiceRequest()
+ goToServiceOverview()
+ logout()
Related to
AppointmentDetails
- int appointmentId
- String scheduledTime
- String status
- String problemDescription
+ getAppointmentId()
+ getScheduledTime()
+ getStatus()
+ getProblemDescription()
Binds properties
Binds properties
LoginController
- ServerConnection serverConnection
- String username
- String password
+ login()
Sends requests
Navigates to
Navigates to
ServiceRequestController
- ServerConnection serverConnection
+ createServiceRequest()
AI_AssistedController
- ServerConnection serverConnection
+ getRecommendation(problemDescription: String)
SimpleStringProperty
- String value
+ set(value: String)
+ get(): String
Executes LoginCommand
Executes ServiceRequestCommand
Sends request for technician recommendation
Command
+ execute()

The diagram represents the architecture of the client system based on Object-Oriented Design (OOD) patterns. These patterns help solve common design problems and improve flexibility, maintainability, and scalability. Here's an explanation of each pattern used in the diagram and where and why it is applied:

1. Singleton Pattern: ServerConnection

Where: The ServerConnection class.

Why: The ServerConnection class uses the Singleton pattern to ensure that only one instance of the server connection exists throughout the application. A single, consistent connection is needed to communicate with the server, and this pattern helps to manage that by restricting the instantiation of the ServerConnection class to one object. This is important to avoid multiple connections and to ensure resources (e.g., sockets) are not overused.

How: The ServerConnection class has a private constructor and a static method to return the single instance.

2. Command Pattern: Command, LoginCommand, ServiceRequestCommand

Where: The Command class, and its concrete implementations: LoginCommand and ServiceRequestCommand.

Why: The Command pattern is used to encapsulate all details of a request, such as the action to be performed and the data required. This decouples the sender (controllers) from the receiver (server) of the request, allowing for flexibility in the way requests are executed. It also allows for easier testing, undo operations, and adding new commands in the future.

How: The abstract Command class defines the execute() method, and concrete command classes like LoginCommand and ServiceRequestCommand implement this method to perform specific actions (like logging in or making a service request). Controllers use these command objects to send requests to the server.

3. MVC Pattern (Model-View-Controller)

Where: The entire structure of the system (with separate Controller, Model, and View components).

Why: The MVC pattern is a design pattern that separates the application into three interconnected components:

Model: Represents the data (TechnicianDetails, AppointmentDetails).

View: Represents the user interface (the part that shows the data to the user, such as the forms and tables, though it is not explicitly shown here in terms of GUI).

Controller: Handles the input and updates the model and view (e.g., LoginController, DashboardController, ServiceRequestController).

How:

Model: Classes like TechnicianDetails and AppointmentDetails hold the data.

View: The view is implicitly handled by the controllers, where data is passed to the UI.

Controller: Classes like LoginController, DashboardController, etc., handle user input and update the corresponding models or navigate to other views.

The MVC pattern ensures a clear separation of concerns, making the application more modular, maintainable, and scalable.

4. Factory Method Pattern: ControllerFactory

Where: The ControllerFactory class.

Why: The Factory Method pattern is used to create objects (in this case, controllers) without specifying the exact class of object that will be created. This promotes flexibility and loose coupling, as new types of controllers can be introduced without modifying the code that requests the creation of controllers.

How: The ControllerFactory class provides a method to create controllers like LoginController, DashboardController, etc., based on the type of controller requested. The client code doesn't need to know about the specific controller classes—it just calls the factory to get the controller it needs.

5. Observer Pattern: SimpleStringProperty

Where: The SimpleStringProperty class, used in models like TechnicianDetails and AppointmentDetails.

Why: The Observer pattern is used for automatic updates between the model and view. When the model's data changes, the view is updated accordingly. In this case, the properties like name, email, skills,

status, etc., in the model are observed, and any changes are automatically reflected in the UI.

How: SimpleStringProperty is a class that acts as a wrapper for string values and allows listeners to observe when a change occurs. It provides methods like set() and get() to update or retrieve the value, and it can notify bound UI elements when the value changes. This ensures the UI reflects the latest data without manual updates.

Pattern Summary and Why They're Used:

Singleton Pattern: Ensures only one connection is made to the server to prevent multiple redundant connections.

Command Pattern: Encapsulates request logic into command objects, which decouples the actions from the user interface and allows for future flexibility and scalability.

MVC Pattern: Separates concerns into distinct layers (Model, View, Controller), ensuring that the system is easier to maintain and extend, and allowing for clean separation of logic.

Factory Method Pattern: Provides flexibility in creating controllers, which makes the system more extensible and adaptable to future changes in controller types.

Observer Pattern: Binds the model and view in such a way that any change in data automatically updates the view, simplifying synchronization between the two.

---

2. Accomplished Functional Requirements

- **1.1 User Registration & Authentication**
    - Accomplished: Secure registration, login, and password reset.
- **1.2 Service Request Management**
    - Accomplished: Submission and tracking of service requests.
- **1.3 Technician Search & Booking**
    - Accomplished: Technician search, profile viewing, and booking.
- **1.4 Real-Time Scheduling**
    - Accomplished: Appointment coordination and notifications.
- **1.5 Secure Payment Processing**
    - Accomplished: Secure payments, receipts, and refunds.
- **1.6 Admin Dashboard**
    - Accomplished: Monitoring system performance and managing users.
- **1.7 AI Recommendation**
    - Accomplished: Technician recommendations based on user preferences.
    - Model: Data flow diagrams showing the recommendation engine.

Feature Index	Feature Description	Story Points	Effort (Days)	Effort (Weeks)
1.1	User Registration & Authentication	2	$2 \times 1 = 2$ days	$2 \div 5 = 0.4$ weeks
1.2	Service Request Management	4	$4 \times 1 = 4$ days	$4 \div 5 = 0.8$ weeks
1.3	Technician Search & Booking	3	$3 \times 1 = 3$ days	$3 \div 5 = 0.6$ weeks
1.4	Real-Time Scheduling	2	$2 \times 1 = 2$ days	$2 \div 5 = 0.4$ weeks
1.5	Secure Payment Processing	1	$1 \times 1 = 1$ day	$1 \div 5 = 0.2$ weeks
1.6	Admin Dashboard	4	$4 \times 1 = 4$ days	$4 \div 5 = 0.8$ weeks
1.7	**AI Recommendation**	5	$5 \times 1 = 5$ days	$5 \div 5 = 1.0$ weeks

3. Project Management Metrics Summary

Feature Index	Feature Description	Planned Effort (Days)	Actual Effort (Days)	Effort Difference (Days)	Error %
1.1	User Registration & Auth.	2	2.5	+0.5	25%
1.2	Service Request Management	4	3.8	-0.2	5%
1.3	Technician Search & Booking	3	3.2	+0.2	6.7%
1.4	Real-Time Scheduling	2	2.5	+0.5	25%
1.5	Secure Payment Processing	1	1	0	0%
1.6	Admin Dashboard	4	4.5	+0.5	12.5%
1.7	**AI Recommendation**	5	6	+1.0	20%

4. Summary Metrics

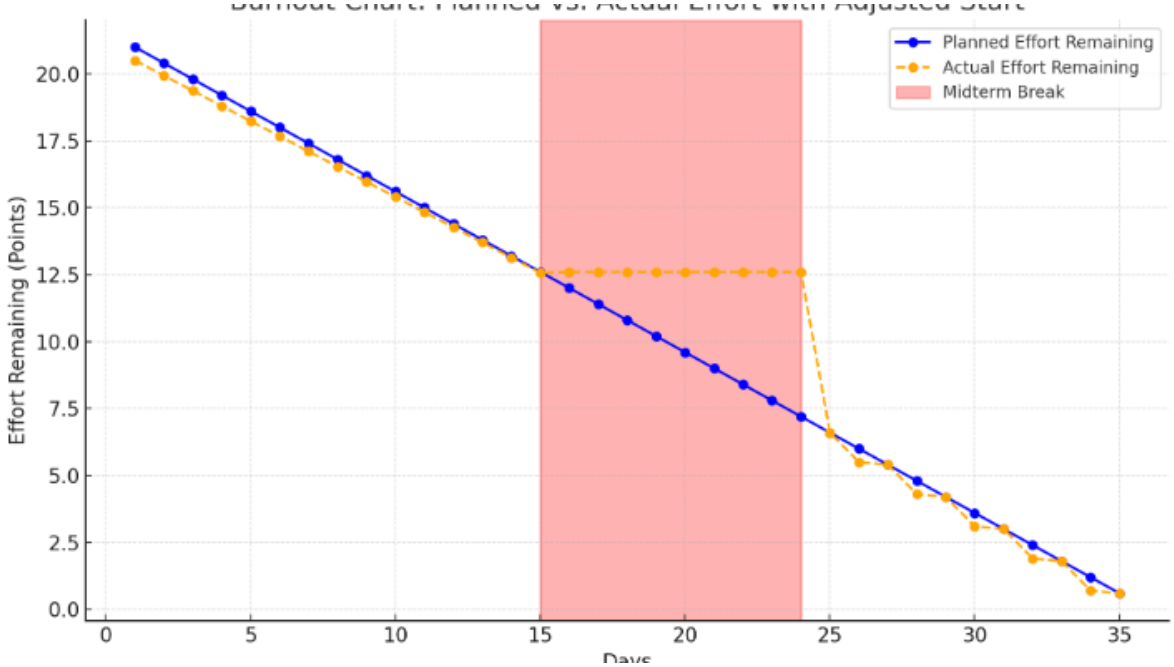
- **Accomplished Effort Percentage:**
  - $\text{Accomplished Effort} = \frac{\text{Actual Effort}}{\text{Planned Effort}} \times 100$
  - $\text{Accomplished Effort} = \frac{(2.5+3.8+3.2+2.5+1+4.5+6)}{(2+4+3+2+1+4+5)} \times 100 \approx 103.3\%$
- **Team Productivity:**
  - Average Effort Points Per Team Member: $\frac{\text{Total Story Points}}{\text{Team Members}}$
  - $\text{Productivity Per Member} = \frac{21}{3} = 7$ points/member.

---

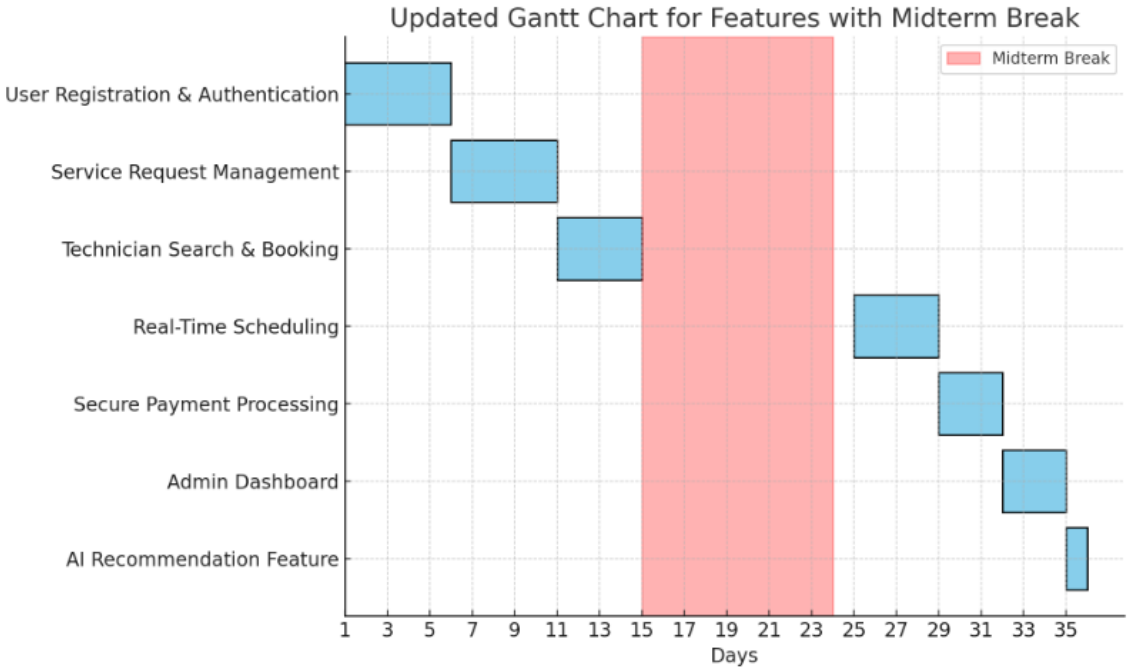
5. Team Member Productivity

Team Member	Contributed Effort (Days)	Effort Points	Productivity (%)
Khalid Abdelsalam Mohamed	7.2	7.5	$\frac{7.5}{21} \times 100 = 35.71\%$
Taher Abdelbary Abdelmalek	7.1	6.9	$\frac{6.9}{21} \times 100 = 32.86\%$
Abdelrahman Amr Mohamed	6.7	6.6	$\frac{6.6}{21} \times 100 = 31.43\%$

6.Burndown Chart



7.Gantt Chart



8. Summarized Table

Project Summary Table

Features	Members	Points	Productivity	Effort Percentage	Error	Actual Time
Authentication	Khalid Abdelsalam, Abdelrahman Amr	5	Khalid = 2 points [Register(User)], Abdelrahman = 3 points [Login(User)]	Khalid = 40%, Abdelrahman = 60%	5%	17/11/2024
Service Request Management	Taher Abdelbary, Khalid Abdelsalam, Abdelrahman Amr	6	Taher = 2 points [Submit Request], Khalid = 2 points [Track Request], Abdelrahman = 2 points [Close Request]	Taher = 33%, Khalid = 33%, Abdelrahman = 33%	3%	27/11/2024
Technician Search & Booking	Khalid Abdelsalam, Taher Abdelbary	4	Khalid = 2 points [Search Technician], Taher = 2 points [Book Technician]	Khalid = 50%, Taher = 50%	2%	07/12/2024
Real-Time Scheduling	Abdelrahman Amr, Taher Abdelbary	3	Abdelrahman = 2 points [Schedule Appointment], Taher = 1 point [Send Notifications]	Abdelrahman = 67%, Taher = 33%	4%	17/12/2024
AI Recommendation Feature	Khalid Abdelsalam, Taher Abdelbary	3	Khalid = 2 points [Recommend Technicians], Taher = 1 point [Enhance Recommendations]	Khalid = 67%, Taher = 33%	5%	28/12/2024