# Choice Based Customer Support Chatbot

This project addresses a common real-world challenge in customer support: inefficiencies caused by repetitive queries and prolonged response times. The choice-based chatbot provides users with a streamlined, menu-driven interface to access FAQs, track product service status, calculate service fees, or retrieve contact details efficiently. By automating routine interactions, the solution reduces the workload on support teams, enhances user satisfaction through faster service, and optimizes resource allocation for businesses, ensuring that complex issues receive the necessary attention.

## Objectives:

- To develop a responsive chatbot system that automates common customer support queries.
- To provide a structured and menu-driven interaction model for users.
- To reduce customer support workload by automating repetitive tasks.
- To ensure the solution is scalable, robust, and adaptable for future requirements.
- To utilize design patterns like State, Chain of Responsibility, and some other patterns for efficient architecture.

## Features:

- **Dynamic State-Based Interaction:** Implementation of the **State Pattern** for navigating through different menus and submenus.

- **FAQ Assistance:** Provides instant answers to common questions, reducing the need for human intervention.

- **Service Tracking:** Allows users to input product details to retrieve the current service status and other updates.

- **Complaint Handling:** Simplifies the process of lodging complaints by guiding users through predefined steps.

- **Customizable Maintenance Suggestions:** Adapts maintenance recommendations based on product type using the **Decorator Pattern**.

- **Secure Exit and Session Control:** Ensures that multiple chatbot sessions cannot run simultaneously by leveraging the **Singleton Pattern**.

- **Future-Ready Architecture:** Designed to easily integrate with external APIs or additional features like fee calculations.

# Design Patterns:

## State Pattern for Chatbot Navigation

The State Pattern is used to manage the chatbot's behavior as it transitions between different menus (e.g., main menu, FAQ, product info). By encapsulating states into separate classes, the chatbot becomes more modular and easier to maintain.

- **State Interface:** The **ChatState** interface defines the common structure for all chatbot states, with methods like **displayMenu()** and **handleInput(String input)**.

```
1  package State;
2
3  public interface ChatState
4  {
5      void displayMenu(); // Shows the current chat state.
6      void handleInput(String input); // Handles the user input.
7  }
```

Fig 1. ChatState interface.

- **MainMenuState Class:** Implements the **ChatState** interface and represents the main menu. Handles user input to transition to other states like **FAQState**, **InfoState** or the other states.

```
public class MainMenuState implements ChatState
{
    private ChatContext context;

    public MainMenuState(ChatContext context)
    {
        this.context = context;
    }

    @Override
    public void displayMenu()
    {
        System.out.println();
        System.out.println(x:"Main Menu");
        System.out.println(x:"1) FAQ");
        System.out.println(x:"2) Information about my Product");
        System.out.println(x:"3) Send us your complaints");
        System.out.println(x:"4) Contact live support.");
        System.out.println(x:"Q) Exit");
    }

    @Override
    public void handleInput(String input)
    {
```

Fig 2. A part of the MainMenuState implementation.

## Chain of Responsibility (CoR) for Input Processing

In this implementation, the **Chain of Responsibility Pattern** is integrated with the **State** Pattern to manage user inputs within each state. Instead of having a separate chain of handler classes, each ChatState class directly processes inputs related to its context, and unhandled inputs are passed back or retained for further processing.

```java
// Main context class that manages chat states and user interactions
public class ChatContext {
    private ChatState currentState;
    private Scanner scanner;

    // Initialize context with scanner and set initial state
    public ChatContext() {
        this.scanner = new Scanner(System.in);
        // Start with intro state when chat begins to display welcome message
        this.currentState = StateFactory.creatState(stateType:"IntroState", this);
    }
```

Fig 3. ChatContext class implementation.

- **ChatContext Class:** Acts as the central hub that delegates user inputs to the current active state via the **handleInput(String input)** method.

- **State-Specific Input Handling:** Each state class (e.g., **IntroState**, **MainMenuState**) implements its own **handleInput()** logic. This approach allows clean separation of input handling based on the chatbot's current state.

```java
@Override
public void handleInput(String input)
{
    switch(input)
    {
        case "1":
            context.setState(StateFactory.creatState(stateType:"ContactState", context));
            break;

        case "r":
        case "R":
            context.setState(StateFactory.creatState(stateType:"MainMenuState", context));
            break;
```

Fig 4. FAQState class' own handleInput() implementation.

- **Flexible Transitions:** The CoR pattern here is "virtualized" by chaining state transitions dynamically through user input, where unrecognized inputs either loop back or forward based on state-specific logic.

```java
default:
    System.out.println();
    System.out.println(x:"Unknown choice, please select an input from below.");
    this.context.setState(this);  // Return to the same state.
    break;
```

Fig 5. Unrecognized input loops back on the same state for UX purposes.

## Singleton Pattern for State Management

The Singleton Pattern ensures that there is only one instance of the chatbot's state manager throughout the application lifecycle. This prevents duplication and manages shared state effectively.

- **ChatManager Class:** Implements a static method **getInstance()** to provide a global point of access. This centralizes state transitions and ensures consistency.

```
public static ChatManager getInstance() // global access
{
    if (instance == null)
    {
        instance = new ChatManager();
    }
    return instance;
}
```

Fig 6. Creating a global access.

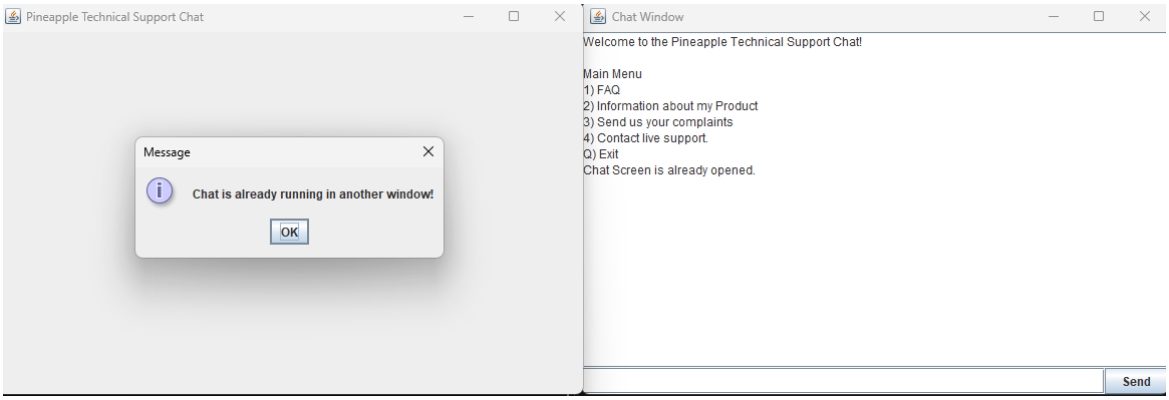


Fig 7. A second chat won't launch with the Singleton Pattern implementation.

## Factory Pattern for State Creation

The Factory Pattern simplifies state object creation by decoupling the instantiation logic from the main code. This promotes scalability as new states can be added without modifying existing logic.

- **StateFactory Class:** Contains a static method **createState(String stateType, ChatContext context)** that returns instances of specific states based on input.

- **Example Usage:** When transitioning from the main menu to the FAQ menu, **StateFactory** dynamically creates an **FAQState** object.

```
public static ChatState creatState(String stateType, ChatContext context)
{
    switch (stateType)
    {
        case "IntroState":
            return new IntroState(context);
```

Fig 8. A part of the StateFactory class.

## Decorator Pattern for Flexible Service Fee Calculation

The Decorator Pattern is implemented to dynamically calculate and extend service fees based on the specific parts and damages of a product. This approach ensures modularity, reusability, and scalability, allowing new repair cost calculations to be added without modifying the core product classes.

```
public Product(String productTrackingNo, String productType, String partType, String damageType, String warrantyStartDate,
               String warrantyEndDate, double repairCost, String serviceDate, String status) {
    this.productTrackingNo = productTrackingNo;
    this.productType = productType;
    this.partType = partType;
    this.damageType = damageType;
    this.warrantyStartDate = warrantyStartDate;
    this.warrantyEndDate = warrantyEndDate;
    this.repairCost = repairCost;
    this.serviceDate = serviceDate;
    this.status = status;
    this.warrantyRules = new WarrantyRules();
}
```

Fig 9. A part of the Product class.

- **Component Interface (Product.java):** Defines the structure for all product types, ensuring consistency in behavior like **calculateServiceFee()** and common properties such as warranty status, part type, and repair cost.

- **Concrete Components (Phone.java etc.):** Represent **specific product types**, each with its own default service fee calculation logic. For example, a Notebook charges $50 for out-of-warranty service.

- **Abstract Decorator (RepairCostDecorator.java): Extends the Product class** and wraps a Product instance. This decorator delegates behavior to the wrapped object while providing the flexibility to override or extend functionality.

```
public RepairCostDecorator(Product product) {
    super(
        product.getProductTrackingNo(),
        product.getProductType(),
        product.getPartType(),
        product.getDamageType(),
        product.getWarrantyStartDate(),
        product.getWarrantyEndDate(),
        product.getRepairCost(),
        product.getServiceDate(),
        product.getStatus()
    );
    this.product = product; // We will use the passed Product object within this class
}
```

Fig 10. A part of the RepairCostDecorator.

- **Concrete Decorators (BatteryRepairCost.java etc.):** Each decorator represents a specific part type, such as a "screen" or "battery." These classes override the **getPartType()** and **calculateServiceFee()** methods to adjust the service fee based on part-specific logic.

- **Utility Class for Warranty and Part Coverage Rules (WarrantyRules.java):** Centralizes warranty-related logic, enabling the Decorators to verify:

    1. Whether the warranty is still valid for a given product and service date using the is **WarrantyValid()** method.

    2. Whether a specific part is covered under the warranty for a particular damage type using the **isPartCovered()** method.

    3. This class **encapsulates** the rules for warranty and coverage, ensuring **consistency** and **maintainability** across the system.

## Observer Pattern for Live Support

The Observer Pattern is implemented in the live support feature to enable future implementation for real-time communication between customers and support agents. This pattern allows support agents to be notified automatically when customers send messages.

- **ChatObserver Interface:** Defines the contract for objects that need to receive chat updates through the **update()** method.

```
1   package Observer;
2
3   public interface ChatObserver {
4       void update(String message);
5   }
```

Fig 11. ChatObserver Interface.

- **SupportAgent Class:** Implements the ChatObserver interface to receive and respond to customer messages. Each agent can provide customized responses based on customer inquiries.

```
// Represents a support agent that can respond to customer messages
public class SupportAgent implements ChatObserver {
    private String name;

    public SupportAgent(String name) {
        this.name = name;
    }
```

Fig 12. A part of the SupportAgent Class.

- **SupportState Implementation:** Acts as the **Subject** in the Observer pattern, maintaining a list of observers (support agents) and notifying them when new messages arrive.

```java
// Register new observer (support agent) to receive chat messages
public void addObserver(ChatObserver observer) {
    observers.add(observer);
}

// Notify all support agents about new customer message
public void notifyObservers(String message) {
    for (ChatObserver observer : observers) {
        observer.update(message);
    }
}
```

Fig 13. A part of the SupportAgent Class.

# UML Diagram:

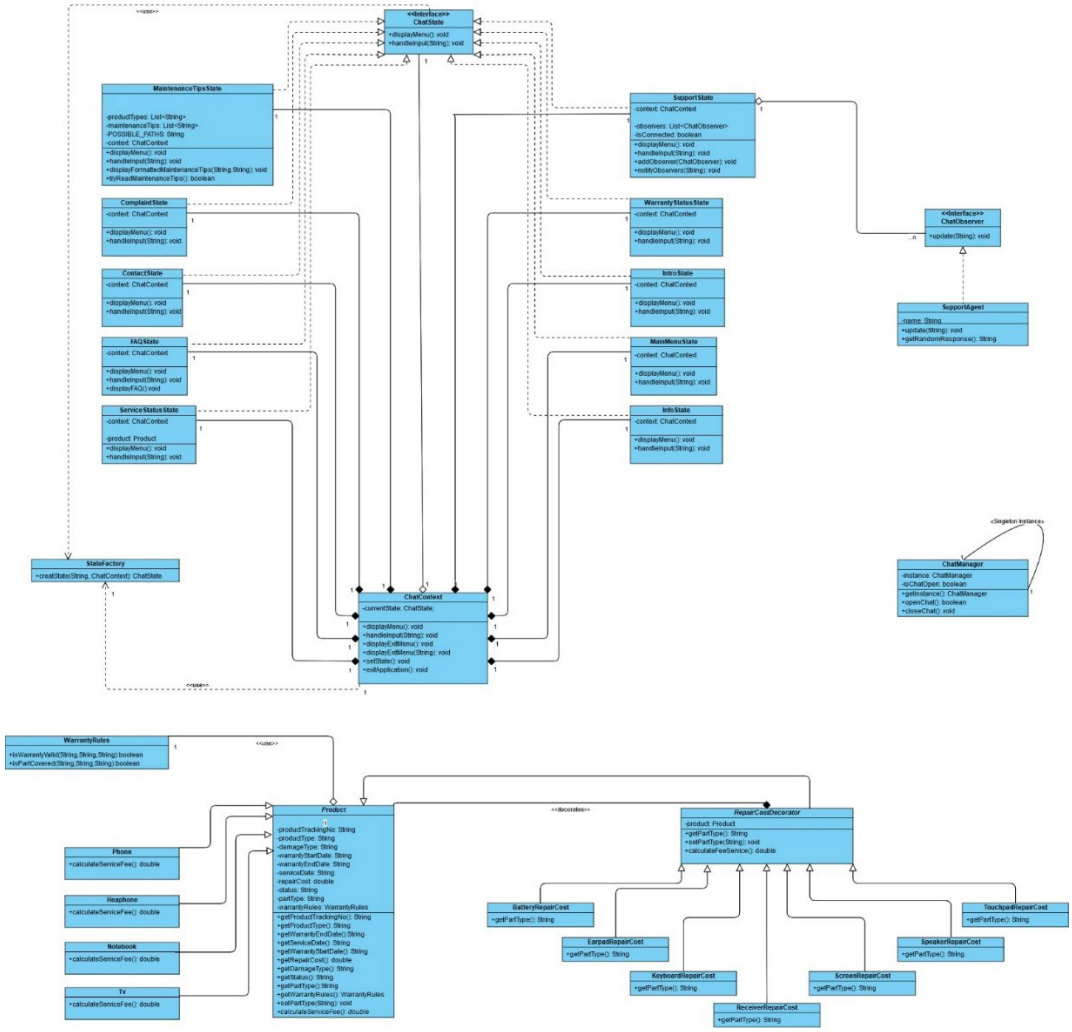

Fig 14. The UML Diagram for the whole project.

# Conclusion:

The **Choice-Based Customer Support Chatbot** project demonstrates the potential of structured, design-pattern-driven software to address real-world challenges in customer support. By employing robust design principles like the State, Chain of Responsibility, Singleton, Factory, Decorator, and Observer patterns, this system ensures modularity, scalability, and adaptability for evolving user needs. It automates routine interactions such as FAQs, service tracking, and maintenance suggestions, reducing human workload and increasing user satisfaction. The chatbot serves as a modern, efficient solution for businesses seeking to optimize customer engagement while ensuring efficient resource utilization.