2025 Spring: CSAS 2124BA Intro Object Orient Design II- Mini Project 2: Customer Support chatbot "PirateEase"

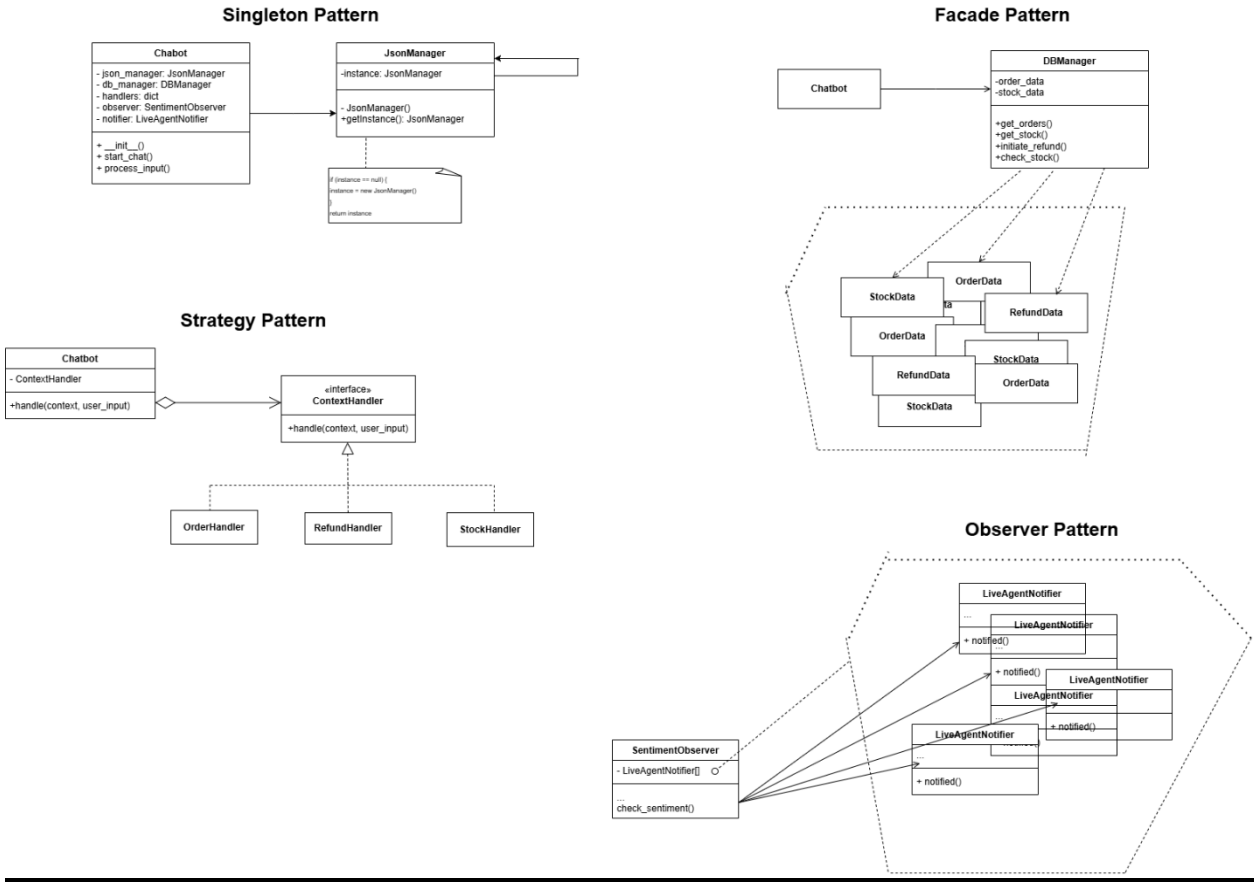
Due Date: 31st March 2025

Professor: Shajina Anand

Author: Kabir Ansari

Contents

UML Class Diagram



Design Patterns Applied

- Singleton Pattern (**Creational**): This is used for the **JsonManager** class, and its purpose is to only have one instance, where a global access point can be generated to the sharing of resources. Additionally, this design pattern ensures consistent data by preventing multiple instances from accessing the database simultaneously, which as a result will reduce connection overhead.

- Façade Pattern (**Structural**): This is not heavily utilized but is relevant regarding the **DBManager Class**, as it simplifies the complex operations such as **file reading/writing** into simpler method calls (e.g. **get_orders(), get_stock(), initiate_refund(), etc.).** Although not formally structured, the **DBManager Class** acts as a façade over the order and stock file interactions, shielding the rest of the program from such CSV file complexities. There is no additional façade since the **DBManager Class** handles everything.

- Strategy Pattern (**Behavioral**): The **ContextHandler** abstract class and its subclasses encapsulate distinct behaviors. Each subclass (e.g., **OrderHandler RefundHandler, StockHandler**) implements the **handle()** method, enabling the **DBManager** class to call actions uniformly regardless of the specific behavior.

- Observer Pattern (**Behavioral**): Allows an object (**the observer/SentimentObserver)** to watch another object or situation (**e.g. user input)** and react when something specific happens for instance, when the customer sentiment is 'angry', this triggers a live agent notification. Each prompt is continuously checked regarding the user's sentiment.

OOP Principles Applied

- Abstraction: Abstract class ContextHandler has abstract method handle(). This defines the interface, ensuring the subclasses (e.g. **OrderHandler, RefundHandler, StockHandler)** define their respective specific behavior, without exposing implementation details, specifically with the main Chatbot class.

- Encapsulation: JsonManager, DBManager and SentimentObserver classes encapsulate their respective data (for instance, **query loads, reading orders/stock, checking sentiment/words).** Rather than directly accessing internal dictionaries or files, external classes access data through defined methods such as **get_response(), get_orders(), check_sentiment().** This particularly happens within the methods of the main Chatbot Class.

- Inheritance: The subclasses (**OrderHandler, RefundHandler, StockHandler**) inherit from abstract base class **ContextHandler**, reusing common functionality while overriding the **handle ()** method for specific behavior.

- Polymorphism: All the handler classes (**OrderHandler, RefundHandler, StockHandler)** implement the same **handle()** method, but with different logic that is dependent on the context (e.g. **orders, refunds, stocks).** For the **process_input()** method under the main Chatbot class, the **self.handlers[context].handle(user_input)** is called. The significance behind this is that the main Chatbot class does not need to know which exact class is utilized, rather the **handle()** method is called, and the respective correct method is executed.

The Codes

chatbot.py

```
import json
import csv
import time
import sys
from abc import ABC, abstractmethod

#Chatbot has an interactive approach of responding
def typing_animation(texting, delay=0.05):
    for char in texting:
        sys.stdout.write(char)
        sys.stdout.flush()
        time.sleep(delay)
    print()

#Singleton Pattern
QMU_DOWN = "our system is down! connecting you to a live agent..."
class JsonManager:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            try:
                with open("queries.json", "r") as f:
                    cls._instance.queries = json.load(f)
            except Exception as e:
                return False
        return cls._instance
```

```
    #Retrieval from queries.json
    def get_response(self, query):
        return self.queries.get(query.lower(), {}).get("response")

    def get_context(self, query):
        return self.queries.get(query.lower(), {}).get("context")

    def get_response_type(self, query):
        return self.queries.get(query.lower(), {}).get("response_type")

    def provide_product(self, query):
        return self.queries.get(query.lower(), {}).get("product")

#Facade Pattern
class DBManager:
    def __init__(self):
        self.orders = None
        self.stock = None

        #Open and read csv files for orders and stock (backend service)
        with open("orders.csv", "r") as f:
            self.orders = list(csv.DictReader(f))

        with open("stock.csv", "r") as f:
            self.stock = list(csv.DictReader(f))
```

```
    #Retrieve orders
    def get_orders(self):
        return self.orders

    #Retrieve stock
    def get_stock(self):
        return self.stock

    #Begin refund
    def initiate_refund(self, order_id):
        for order in self.orders:
            if order["order_id"] == order_id:
                if order["order_status"] == "cancelled":
                    print("Order already has been cancelled")
                else:
                    print("Cancelling order and initiating refund...")
                    time.sleep(2)
                    print("Your order has been cancelled and the refund has been initiated.")
                    order["order_status"] = "cancelled"
        with open("orders.csv", "w") as f:
            writer = csv.DictWriter(f,
fieldnames=("order_id","prod_id","order_price","customer_id","order_status"))
            writer.writeheader()
            writer.writerows(self.get_orders())

    #Check stock
    def check_stock(self, prod_name):
```

```
        time.sleep(2)
        for product in self.stock:
            if product["prod_name"].lower() == prod_name.lower():
                typing_animation(f"There are ({product["prod_qty"]}) {prod_name} available.")
                return

        typing_animation(f"{prod_name} is out of stock.")



#Strategy Pattern
#Abstract Method
class ContextHandler(ABC):
    @abstractmethod
    def handle(self, qmu=False):
        pass

#Handle orders
class OrderHandler(ContextHandler):
    def handle(self, order_id=None, qmu=True):
        if not qmu:
            return QMU_DOWN
        db = DBManager()
        orders = db.get_orders()
        print([f"order status: {i['order_status']}" for i in orders if str(order_id)== i["order_id"]] or
"not found")


#Handle refunds
```

```
class RefundHandler(ContextHandler):
    def handle(self, order_id=None, qmu=True):
        if not qmu:
            return QMU_DOWN
        db = DBManager()
        orders = db.get_orders()
        print([f"order status: {i['order_status']}" for i in orders if str(order_id)== i["order_id"]] or
"not found")
        db.initiate_refund(str(order_id))


#Handle stock
class StockHandler(ContextHandler):
    def handle(self, product, qmu=True):
        if not qmu:
            return QMU_DOWN
        db = DBManager()
        db.check_stock(product)


#Observer Pattern
class SentimentObserver:
    def __init__(self):
        self.sentiment = "content"


    #Detect angry words
    def check_sentiment(self, words):
        words = words.split(" ")
        with open("angry_words.txt", "r") as f:
```

```
        angry_words = f.read().split("\n")
        for word in words:
            if word.lower() in angry_words:
                self.sentiment = "angry"
    return self.sentiment

#Direct to Live Agent when customer is angry
class LiveAgentNotifier:
    def notified(self):
        typing_animation("connecting to a live agent...")
        time.sleep(2)
        typing_animation("agent notified!")

#The main operation
class Chatbot:
    def __init__(self): #Handlers are called
        self.handlers = {
            "order": OrderHandler(),
            "refund": RefundHandler(),
            "stock": StockHandler()
        }

        #Call sessions, JsonManager, LiveAgent and SentimentObserver
        self.sessions = {}
        self.jsonmanager = JsonManager()
        self.liveagent = LiveAgentNotifier()
        self.observer = SentimentObserver()
```

```
#Live Agent directed when customer's sentiment is angry
def process_input(self, user_id, user_input):
    sentiment = self.observer.check_sentiment(user_input)
    if sentiment == "angry":
        return self.liveagent.notified()

    #Retrive Json Queries
    response = self.jsonmanager.get_response(user_input)
    context = self.jsonmanager.get_context(user_input)
    response_type = self.jsonmanager.get_response_type(user_input)

    #If customer's question does not make sense
    if not response:
        print("I am sorry. I did not understand that. Please rephrase.")
        return

    #Chatbot providing a non-asking response to customer's question
    if response_type == "non-asking":
        typing_animation(response)
        if context == "stock":
            prod = self.jsonmanager.provide_product(user_input)
            if context in self.handlers:
                self.handlers[context].handle(prod)
        else:
            if context in self.handlers:
                self.handlers[context].handle(prod)
```

```
        #Chatbot providing asking a question in response to customer's question
        if response_type == "asking":
            if context == "refund":
                typing_animation("why do you want to return your product?")
                input(">>>: ").strip().lower()
            user_input = input((f"{response}>>>: "))
            self.handlers[context].handle(user_input)


    #Chat procedure
    def start_chat(self, user_id):
        typing_animation("Welcome! How can I help you?")
        while True:
            user_input = input(">>>: ").strip().lower()
            self.process_input(user_id, user_input)
            further = input("Is there anything else I can help you with? (yes/no): ").strip().lower()
#Follow up


            if further == "no": #Customer wants to end chat
                typing_animation("Sure. Have a good day! Bye!")
                break


if __name__ == "__main__":
    c1 = Chatbot()
    c1.start_chat("user1")
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

>>>: why did my payment fail?
check with your bank
Is there anything else I can help you with? (yes/no): yes
>>>: how do i track my order?
You can track your order using the tracking link sent to your email after your purchase.
Is there anything else I can help you with? (yes/no): no
Sure. Have a good day! Bye!
```

Output from chatbot.py (result from the chatbot). Please however, follow the README.md instructions located below to use the chatbot in the command prompt for better use.

queries.json

```
{
  "where is my order?": {
    "response": "please enter your order id",
    "context": "order",
    "response_type": "asking"
  },
  "i want to return my product": {
    "response":"please enter your order id",
    "context":"refund",
    "response_type": "asking"
  },
  "is the iphone 15 in stock?": {
    "response": "checking inventory...",
    "context":"stock",
    "response_type": "non-asking",
    "product": "iphone 15"
  },

  "is the iphone 14 in stock?": {
```

```
    "response": "checking inventory...",
    "context":"stock",
    "response_type": "non-asking",
    "product": "iphone 14"
},

"is the iphone 13 in stock?": {
    "response": "checking inventory...",
    "context":"stock",
    "response_type": "non-asking",
    "product": "iphone 13"
},

"is the iphone 12 in stock?": {
    "response": "checking inventory...",
    "context":"stock",
    "response_type": "non-asking",
    "product": "iphone 12"
},

"is the iphone 11 in stock?": {
    "response": "checking inventory...",
    "context":"stock",
    "response_type": "non-asking",
    "product": "iphone 11"
},
```

"why did my payment fail?": {
"response": "check with your bank",
"context": "payments",
"response_type": "non-asking"
},
"how do i track my order?": {
"response": "You can track your order using the tracking link sent to your email after your purchase.",
"context": "product",
"response_type": "non-asking"
},

"how can i view my recent purchases?": {
"response": "You can view your recent purchases in your account dashboard under 'Order History'.",
"context": "dashboard",
"response_type": "non-asking"

},

"do you offer warranty for products?": {
"response": "Yes! We offer a range of 1-5 year warranties for our products.",
"context": "product",
"response_type": "non-asking"
},

"where do you operate?": {
"response": "We operate in multiple countries including the US, Canada, and Europe.",

"context": "location",

"response_type": "non-asking"

},

"what discounts do you offer?": {

"response": "Please check our 'Deals' section on the homepage for the latest promotions and special offers.",

"context": "product",

"response_type": "non-asking"

},

"how do i activate my product?": {

"response": "Activation instructions are included in the product packaging. You can also find guides on our Support page.",

"context": "product",

"response_type": "non-asking"

},

"what can i accessorise with my product?": {

"response": "Look through the products that are on display to see compatible accessories.",

"context": "product",

"response_type": "non-asking"

},

"do you have physical stores?": {

"response": "Yes! We have physical stores in major cities across the US such as New York, Los Angeles, and Chicago. Store hours vary by location.",

"context": "location",

"response_type": "non-asking"

},

"what are your shipping options?": {

"response": "We offer standard (5-7 days), express (2-3 days), and next-day shipping options at checkout.",

"context": "shipping",

"response_type": "non-asking"

},

"can you add items to an existing order?": {

"response": "Unfortuanately, once an order is placed, we cannot add items to it. You can place a new order for any additional items.",

"context": "items",

"response_type": "non-asking"

},

"when is the next upcoming sale?": {

"response": "Our next seasonal sale begins on the first of next month. Sign up for our newsletter for early access.",

"context": "sale",

"response_type": "non-asking"

},

"what is your repair policy?": {

"response": "We offer repair services for products under warranty at no charge. Out-of-warranty repairs have variable fees.",

"context": "repair",

"response_type": "non-asking"

},

"where is the nearest store or pickup location?": {

"response": "Please enter your ZIP code to find the nearest pickup location.",

"context": "location",

"response_type": "non-asking"

},

"do you offer student discounts?": {

"response": "Yes, we offer a 10% student discount. Verify your student status via your school email or ID when in person.",

"context": "payments",

"response_type": "non-asking"

},

"is there technical support?": {

"response": "Yes, technical support is available by phone, email, or live chat during business hours.",

"context": "tech support",

"response_type": "non-asking"

},

"can i change my shipping method?" : {

"response": "Shipping methods can only be changed within 1 hour of placing your order. Please contact support immediately.",

"context": "shipping",

"response_type": "non-asking"

},

"are there product demos?": {

"response": "Virtual product demonstrations are available by appointment. In-store demos are available during business hours.",

"context": "demo",

"response_type": "non-asking"

}

}

test_cases.py

```
import pytest

from main import DBManager, JsonManager, SentimentObserver, OrderHandler,
RefundHandler, StockHandler, Chatbot, QMU_DOWN


class TestClass:

    #Test for DBManager class to retrieve orders

    def test_get_orders(self):

        obj = DBManager()

        assert isinstance(obj.get_orders(), list)


    #Test for DBManager class to retrieve stock

    def test_get_stock(self):

        obj = DBManager()

        assert isinstance(obj.get_stock(), list)


    #Test for the angry sentiment to be detected

    def test_check_sentiment(self):

        obj = SentimentObserver()

        assert isinstance(obj.check_sentiment("I am angry"), str)


    #Test for no sentiment detected

    def test_check_sentiment_empty(self):

        obj = SentimentObserver()

        assert isinstance(obj.check_sentiment(""), str)


    #Test for the response of the customer's question regarding order status
```

```
def test_get_response_order(self):
    obj = JsonManager()
    response = obj.get_response("where is my order?")
    assert response == "please enter your order id"

#Test for invalid input to be handled
def test_get_response_invalid(self):
    obj = JsonManager()
    response = obj.get_response("invalid input")
    assert response is None

#Test for the context of the customer's question regarding products in stock
def test_get_context(self):
    obj = JsonManager()
    assert isinstance(obj.get_context("is the iphone 15 in stock?"), str)

#Test for teardown method to reset the singleton instance of JsonManager
def teardown(self):
    JsonManager._instance = None

#Test for the product in question to be provided
def test_provide_product(self):
    obj = JsonManager()
    assert obj.provide_product("is the iphone 15 in stock?") == "iphone 15"

#Test for the QMU_DOWN constant to be defined
def test_QMU_DOWN(self):
```

```
obj = OrderHandler()

assert obj.handle(order_id="123", qmu= False) == QMU_DOWN
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

platform win32 -- Python 3.13.1, pytest-8.3.4, pluggy-1.5.0
rootdir: C:\Users\ansarika\Desktop\MiniProject2
collected 9 items

test_cases.py .........                                                  [100%]

============================================== 9 passed in 0.02s ==============================================

C:\Users\ansarika\Desktop\MiniProject2>
```

angry_words.txt

infuriated

annoyed

mad

furious

irritated

livid

sore

aggravated

help

angry

ridiculous

customers.csv

customer_id,firstname,lastname,email

1,John,Doe,john@doe.gmail.com

2,Jane,Doe,jane@doe.gmail.com

orders.csv

order_id,prod_id,order_price,customer_id,order_status

1,3,2500,1,cancelled

2,5,4500,1,in-transit

3,4,3500,2,completed

stock.csv

prod_id,prod_name,prod_qty,prod_price,prod_description

1,iphone 11,800,1500,apple iphone 11

2,iphone 12,900,2000,apple iphone 12

3,iphone 13,1000,2500,apple iphone 13

4,iphone 14,1100,3500,apple iphone 14

5,iphone 15,0,4500,apple iphone 15

README.md (Instructional)

 # Chat-Bot

This interactive and intelligent ChatBot called PirateEase, allows customers to communicate with. The system is able to handle any such questions that is stored in the JSON database.

Prerequisites

- **pip install pytest**

Installation

1. ****Install Dependencies****

Make sure you are using the latest python version in your IDE

2. **Usage**

Run the chat_bot.py file in the specified path that it is located on

```Command Prompt
  cd C:\path\to\folder
```

For example:

```Command Prompt
  cd C:\Users\username\Desktop\MiniProject2>py chatbot.py
```

Most of the time though when you run this program, the IDE will automatically be able to retrive the correct file where you are trying to access the program from

3. **UML Diagram**

If you want to see the UML Diagram completely or create one of your own:

- Install draw.io diagrams
- Click on my posted UML Diagram and view what I have created
- This application has all the necessary tools and features for UML Diagram creation

Run Program

To make the chat bot more visually appealing, the aim is to hide all the backend code that was implemented. For this to work:

****Open up Command Prompt****

```Command Prompt -> Step 1
 cd C:\path\to\folder
```

Then

```Command Prompt -> Step 2
py chatbot.py
```

After following these steps and typing step 1 and 2 into Command Prompt you should see get the chat bot working in the terminal without having to be in any IDE