Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

# Лабораторна робота №5

## **Технології розроблення програмного забезпечення**

«*ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»*»

Варіант 29

Виконав:
студент групи ІА-13
Хілько І.А.

Перевірив:
Мягкий М. Ю.

Київ 2023

**Тема:** Шаблони Adapter, Builder, Command, Chain of responsibility, Prototype.

**Варіант:**
Система для колективних покупок. (State, **Chain of responsibility**, Abstract factory, Mediator, Composite, Client-server).

Система дозволяє створити список групи для колективної покупки, список що потрібно купити з орієнтовною вартістю кожної позиції та орієнтовною загальною вартістю, запланувати хто що буде купляти. Щоб користувач міг відмітити що він купив, за яку суму, з можливістю прикріпити чек. Система дозволяє користувачу вести списки бажаних для нього покупок, з можливістю позначати списки, які будуть доступні для друзів (як списки, що можна подарувати користувачеві). Система дозволяє добавляти інших користувачів в друзі.

**Хід роботи:**
1. Реалізувати не менше 3-х класів та шаблон Chain of responsibility.

Ланцюжок обов'язків — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

Я використовую цей патерн для обробки команд користувача в залежності від їхнього типу.

Базовий клас Handler:

```
class Handler(abc.ABC):
    @abc.abstractmethod
    def handle(self, server, client_socket, current_user, command_args):
        pass
```

Це абстрактний базовий клас, який визначає метод *handle*. Всі конкретні обробники  повинні успадковувати цей клас і реалізовувати метод handle.

Конкретні реалізації обробників:

```
class RegisterHandler(Handler):
    def handle(self, server, client_socket, current_user, command_args):
        if len(command_args) == 2:
            username, password = command_args
            try:
                with DatabaseManager(SQLiteDatabaseFactory()) as cursor:
                    cursor.execute('SELECT * FROM users WHERE username=?',
(username,))
                    existing_user = cursor.fetchone()
                    if existing_user:
                        client_socket.send('Username is already taken. Please choose
another one.'.encode('utf-8'))
                    else:
                        cursor.execute('INSERT INTO users (username, password) VALUES
(?, ?)', (username, password))
                        client_socket.send('Registration successful!'.encode('utf-
8'))
            except Exception as e:
                print(f'Error registering user: {e}')
                client_socket.send('Error during registration. Please try
again.'.encode('utf-8'))
        else:
            client_socket.send('Invalid registration format. Usage: REGISTER username
password'.encode('utf-8'))
```

```
class LoginHandler(Handler):
    def handle(self, server, client_socket, current_user, command_args):
        if len(command_args) == 2:
            username, password = command_args
            try:
                with DatabaseManager(SQLiteDatabaseFactory()) as cursor:
                    cursor.execute('SELECT * FROM users WHERE username=? AND
password=?', (username, password))
                    user = cursor.fetchone()
                    if user:
                        client_socket.send('Login successful!'.encode('utf-8'))
                        server.current_state = ServerState.LOGGED_IN
                        return user
                    else:
                        client_socket.send('Invalid username or
password.'.encode('utf-8'))
            except Exception as e:
                print(f'Error during login: {e}')
                client_socket.send('Error during login. Please try
again.'.encode('utf-8'))
        else:
            client_socket.send('Invalid login format. Usage: LOGIN username
password'.encode('utf-8'))

        return None
```

Ланцюжок обробників:

```
class LoginRegisterCommandHandler(Handler):
    def handle(self, server, client_socket, current_user, command, command_args):
        if command == 'REGISTER':
            RegisterHandler().handle(server, client_socket, current_user,
command_args)
        elif command == 'LOGIN':
            return LoginHandler().handle(server, client_socket, current_user,
command_args)
        else:
            client_socket.send('Invalid command. Please enter a valid
command.'.encode('utf-8'))
```

LoginRegisterCommandHandler також успадковує Handler і реалізує метод handle. Він перевіряє тип команди і викликає відповідний обробник. Якщо обробник відсутній, відправляється повідомлення про невідому команду.

Використання у Server:

```
class Server:
    def __init__(self, host='127.0.0.1', port=5559):
        self.host = host
        self.port = port
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((self.host, self.port))
        self.server.listen()
        self.current_state = ServerState.NOT_LOGGED_IN
        self.register_handler = RegisterHandler()
        self.login_handler = LoginHandler()
        self.login_register_handler = LoginRegisterCommandHandler()
        self.client_states = {}
        print(f'Server is listening on {self.host}:{self.port}')
```

```
def handle_client(self, client_socket):
        self.client_states[client_socket] = ServerState.NOT_LOGGED_IN
        current_user = None
        while True:
            data = client_socket.recv(1024)
            if not data:
                break
            decoded_data = data.decode('utf-8')
            print(f'Received from client: {decoded_data}')

            if self.client_states[client_socket] == ServerState.NOT_LOGGED_IN:
                if decoded_data.startswith(('REGISTER', 'LOGIN')):
                    command, *command_args = decoded_data.split()
                    current_user = self.login_register_handler.handle(
                        self, client_socket, current_user, command, command_args
                    )
                    if current_user and command == 'LOGIN':
                        self.client_states[client_socket] = ServerState.LOGGED_IN
                else:
                    client_socket.send('Please log in first.'.encode('utf-8'))
```

```
            elif self.client_states[client_socket] == ServerState.LOGGED_IN:
                self.process_user_command(client_socket, current_user,
decoded_data.split())

        del self.client_states[client_socket]
        client_socket.close()
```

LoginRegisterCommandHandler використовується в методі handle_client для обробки команд в залежності від стану сервера.

Цей підхід дозволяє легко додавати нові обробники для різних типів команд без зміни коду вже існуючих класів. Кожен обробник в ланцюжку відповідає за конкретну функціональність, і якщо він не може обробити команду, вона передається наступному обробнику в ланцюжку.

**Висновок:** Отже, під час виконання лабораторної роботи, я реалізував не менше 3-х класів та шаблон Chain of responsibility.