Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

ПРАКТИЧНА РОБОТА № 5

З дисципліни «Інженерія програмного забезпечення»

на тему «ШАБЛОНИ ПОВЕДІНКИ. ШАБЛОНИ VISITOR, MEMENTO, STATE, COMMAND, INTERPRETER.»

Виконав:
студент групи ІО-42
Куліков М. М.
Залікова: 4214

Перевірила:
Ст. викладач кафедри ОТ
Васильєва М. Д.

Київ – 2025

Практична робота №5

Тема: «ШАБЛОНИ ПОВЕДІНКИ. ШАБЛОНИ VISITOR, MEMENTO, STATE, COMMAND, INTERPRETER».

Мета: Вивчення шаблонів поведінки. Отримання базових навичок з застосування шаблонів Visitor, Memento, State, Command, Interpreter.

Виконання роботи:

Завдання 1

1. Повторити шаблони поведінки для проектування ПЗ. Знати загальну характеристику шаблонів поведінки та призначення кожного з них.
2. Детально вивчити шаблони поведінки для проєктування ПЗ – Visitor, Memento, State, Command, Interpreter. Для кожного з них:
    - вивчити Шаблон, його призначення, мотивацію, випадки коли його застосування є доцільним та результати такого застосування;
    - знати особливості реалізації Шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;
    - вільно володіти структурою Шаблону, призначенням його класів та відносинами між ними;
    - вміти розпізнавати Шаблон в UML діаграмі класів та будувати сирцеві коди Java-класів, що реалізують шаблон.
3. В підготованому проєкті створити програмний пакет work5. В пакеті розробити інтерфейси і класи, що реалізують завдання 1 та 2 (згідно варіанту) з застосуванням одного чи декількох шаблонів (п.2). В класах, що розробляються, повністю реалізувати методи, пов'язані з функціюванням Шаблону. Методи, що реалізують бізнес-логіку, закрити заглушками з виводом на консоль інформації про викликаний метод та його аргументи. Приклад реалізації бізнес-методу: void draw(int x, int y) {System.out.println(“Метод draw з параметрами x=”+x+” y=”+y);}
4. За допомогою автоматизованих засобів виконати повне документування розроблених класів (також методів і полів), при цьому документація має в достатній мірі висвітлювати роль певного класу в загальній структурі Шаблону та особливості конкретної реалізації.

Визначимо варіант для індивідуальних завдань: 4214%8 = 6, 4214%7 = 0.

Варіант 6 до завдання 1:

Необхідно створити модель ігрового персонажа, який містить такі атрибути:

- Position (позиція) – координати персонажа в ігровому просторі;
- Inventory (склад артефактів) – набір предметів, які має персонаж;
- HealthLevel (рівень здоров'я) – кількість очок життя персонажа;
- Character – комплексний об'єкт, який об'єднує всі атрибути і надає механізм збереження та відновлення стану.

➢ Вимоги:

- Персонаж повинен мати методи для отримання та оновлення стану, а також для взаємодії з механізмом збереження.
- Персонаж повинен мати можливість:
  - створювати знімок свого поточного стану (позиція, інвентар, рівень здоров'я);
  - відновлювати стан із збереженого знімка.
- Знімок стану зберігається у відокремленому об'єкті, який не порушує інкапсуляції персонажа.
- Система повинна підтримувати історію станів персонажа, що дозволяє реалізувати функціонал Undo/Redo.
  - У методі main продемонструвати:
- Створити персонажа з початковими атрибутами (позиція, інвентар, рівень здоров'я).
- Змінювати стан персонажа під час гри (збирати артефакти, змінювати позицію, отримувати шкоду).
- Зберігати стан після ключових змін.
- Відновлювати попередні стани персонажа за допомогою механізму Memento (Undo).
- Показати можливість додавання нових атрибутів або операцій без зміни існуючих класів персонажа.

Код до завдання 1:

Вміст файлу main.java:

```
package work5;
/**
 * Головний клас для демонстрації роботи шаблонів Memento та Visitor.
 * @author Broniev
 * @version 1.0
 */
public class Main {
    public static void main(String[] args) {
```

```
        System.out.println(">>-- Creating character");
        Position startPos = new Position(0, 0);
        Inventory startInv = new Inventory();
        startInv.addItem("Tasty potato");
        Character player = new Character(startPos, startInv, 100);
        System.out.println("First stat: " + player);
        GameHistory history = new GameHistory(player);
        history.save();
        System.out.println("\n>>-- Changing stats");
        player.move(10, 5);
        player.pickUpItem("Iron sword");
        System.out.println("New stat: " + player);
        history.save();
        System.out.println("\n>>-- Changing stats");
        player.takeDamage(30);
        System.out.println("Stat after damage: " + player);
        System.out.println("\n>>-- Using Memento (Undo)");
        System.out.println("using Undo (before taking damage)...");
        history.undo();
        System.out.println("Returned stats: " + player);
        System.out.println("\nusing second Undo (to the start stats)...");
        history.undo();
        System.out.println("Returned stats: " + player);
        System.out.println("\n>>-- Using Memento (Redo)");
        System.out.println("Using Redo (get back to 'find sword')...");
        history.redo();
        System.out.println("Returned stats: " + player);
        System.out.println("\n>>-- Using Visitor (Adding operation) ---");
        System.out.println("Adding new operation");
        CharacterVisitor statusReportVisitor = new CharacterStatusVisitor();
        player.accept(statusReportVisitor);
    }
}
```

Вміст файлу Memento.java:

```
package work5;
/**
 * Інтерфейс-маркер для паттерну Memento.
 */
public interface Memento {
}
```

Вміст файлу Visitable.java:

```
package work5;
/**
 * Інтерфейс для елементів, які можуть прийняти Visitor.
 * Його реалізує клас Character.
 */
public interface Visitable {
    /**
     * Стандартний метод accept для патерну Visitor.
     * @param visitor Об'єкт Відвідувача, який виконає операцію.
     */
    void accept(CharacterVisitor visitor);
}
```

Вміст файлу CharacterVisitor.java:

```
package work5;
/**
 * Інтерфейс Visitor (Відвідувач)
 * Описує методи для обходу об'єктів
 */
public interface CharacterVisitor {
    /**
     * Метод для відвідування персонажа
     * @param character об'єкт, який ми "відвідуємо"
     */
    void visit(Character character);
}
```

Вміст файлу Character.java:

```
package work5;
/**
 * Основний клас персонажа.
 * У патерні Memento виступає як Originator.
 * У патерні Visitor виступає як ConcreteElement.
 */
public class Character implements Visitable {
    private Position position;
    private Inventory inventory;
    private int healthLevel;
    /**
     * Конструктор персонажа
     * @param position Початкова позиція
     * @param inventory Початковий інвентар
     * @param healthLevel Початковий рівень здоров'я
     */
    public Character(Position position, Inventory inventory, int healthLevel) {
        this.position = position;
        this.inventory = inventory;
        this.healthLevel = healthLevel;
    }
    /**
     * Метод зміни позиції персонажа
     * @param newX Нова координата X
     * @param newY Нова координата Y
     */
    public void move(int newX, int newY) {
        System.out.println("Moving to x=" + newX + " y=" + newY);
        this.position.setX(newX);
        this.position.setY(newY);
    }
    /**
     * Метод підбору артефакту гравцем
     * @param item Назва артефакту
     */
    public void pickUpItem(String item) {
        System.out.println("Picked item: " + item);
        this.inventory.addItem(item);
    }
    /**
     * Метод отримання шкоди гравцем
     * @param amount Кількість шкоди
     */
    public void takeDamage(int amount) {
        System.out.println("Damage taken: " + amount);
```

```
    this.healthLevel -= amount;
}
/**
 * Клас Memento
 * Внутрішній приватний клас, що реалізує знімок стану
 */
private static class CharacterMemento implements Memento {
    private final Position savedPosition;
    private final Inventory savedInventory;
    private final int savedHealth;
    /**
     * Приватний конструктор, який викликається лише методом save() класу Character
     */
    private CharacterMemento(Position posToSave, Inventory invToSave, int healthToSave) {
        this.savedPosition = new Position(posToSave);
        this.savedInventory = new Inventory(invToSave);
        this.savedHealth = healthToSave;
    }
    private Position getSavedPosition() { return savedPosition; }
    private Inventory getSavedInventory() { return savedInventory; }
    private int getSavedHealth() { return savedHealth; }
}
/**
 * Створює знімок поточного стану персонажа
 * @return Об'єкт Memento, що містить копію стану
 */
public Memento save() {
    System.out.println(">> Character.save() -> Saved");
    return new CharacterMemento(this.position, this.inventory, this.healthLevel);
}
/**
 * Відновлює стан персонажа із вказаного знімка
 * @param memento Об'єкт Memento, з якого відновлюється стан
 */
public void restore(Memento memento) {
    CharacterMemento m = (CharacterMemento) memento;
    this.position = m.getSavedPosition();
    this.inventory = m.getSavedInventory();
    this.healthLevel = m.getSavedHealth();
    System.out.println(">> Character.restore() -> Restored!");
}
@Override
public void accept(CharacterVisitor visitor) {
    System.out.println(">> Character.accept() -> Visitor allowed to visit me");
    visitor.visit(this);
}
/**
 * Повертає поточну позицію
 * @return Поточна позиція
 */
public Position getPosition() {
    return position;
}
/**
 * Повертає поточний інвентар
 * @return Поточний інвентар
 */
public Inventory getInventory() {
    return inventory;
}
```

```
    /**
     * Повертає поточний рівень здоров'я
     * @return Поточний рівень здоров'я
     */
    public int getHealthLevel() {
        return healthLevel;
    }
    /**
     * Стандартний метод для виводу поточної статистики персонажа.
     */
    @Override
    public String toString() {
        return "Character [Health=" + healthLevel + ", " + position + ", " + inventory + "]";
    }
}
```

Вміст файлу CharacterStatusVisitor.java:

```
package work5;

/**
 * Конкретний Visitor.
 * Реалізує виведення статистики про персонажа.
 */
public class CharacterStatusVisitor implements CharacterVisitor {
    /**
     * Реалізація операції друку стану персонажа.
     * @param character Персонаж, стан якого потрібно вивести.
     */
    @Override
    public void visit(Character character) {
        System.out.println(">- Character summary (Visitor):");
        System.out.println("- Health: " + character.getHealthLevel());
        System.out.println("- Position: " + character.getPosition());
        System.out.println("- Inventory: " + character.getInventory());
    }
}
```

Вміст файлу GameHistory.java:

```
package work5;
import java.util.Stack;
/**
 * Клас, який відповідає за збереження історії гри.
 * Виконує роль Caretaker (Опікуна) у патерні Memento.
 * Дозволяє зберігати стан, повертатись до попереднього (undo) та відмінити зміни (redo)
 */
public class GameHistory {
    private final Stack<Memento> undoStack = new Stack<>();
    private final Stack<Memento> redoStack = new Stack<>();
    private final Character character;
    /**
     * Конструктор Опікуна.
     * @param character Персонаж, за яким ведеться нагляд.
     */
    public GameHistory(Character character) {
        this.character = character;
    }
    /**
     * Зберігає поточний стан персонажа в історії
```

```
     * Очищує історію Redo.
     */
    public void save() {
        System.out.println("[GameHistory] Saving...");
        undoStack.push(character.save());
        redoStack.clear();
    }
    /**
     * Скасовує останню дію (відновлює попередній стан).
     */
    public void undo() {
        if (undoStack.isEmpty()) {
            System.out.println("[GameHistory] There is no any records (Undo).");
            return;
        }
        redoStack.push(character.save());
        Memento mementoToRestore = undoStack.pop();
        character.restore(mementoToRestore);
        System.out.println("[GameHistory] Undo.");
    }
    /**
     * Повторює скасовану дію (відновлює "майбутній" стан)
     */
    public void redo() {
        if (redoStack.isEmpty()) {
            System.out.println("[GameHistory] There is no any records (Redo).");
            return;
        }
        undoStack.push(character.save());
        Memento mementoToRestore = redoStack.pop();
        character.restore(mementoToRestore);
        System.out.println("[GameHistory] Redo.");
    }
}
```

Вміст файлу Inventory.java:

```
package work5;

import java.util.ArrayList;
import java.util.List;

/**
 * Клас інвентаря. Зберігає список речей персонажа
 */
public class Inventory {
    private List<String> items;
    /**
     * Конструктор для створення пустого інвентаря
     */
    public Inventory() {
        this.items = new ArrayList<>();
    }
    /**
     * Конструктор для копіювання
     * @param other Об'єкт Inventory для копіювання
     */
    public Inventory(Inventory other) {
        this.items = new ArrayList<>(other.items);
    }
```

```
    /**
     * Метод для додавання предмету.
     * @param item Назва предмету.
     */
    public void addItem(String item) {
        System.out.println("New item in inventory: " + item);
        this.items.add(item);
    }
    @Override
    public String toString() {
        return "Inventory: " + String.join(", ", items);
    }
}
```

Вміст файлу Position.java:

```
package work5;
/**
 * Клас для зберігання координат (x, y)
 */
public class Position {
    private int x;
    private int y;
    /**
     * Конструктор для початкової позиції.
     * @param x Координата X
     * @param y Координата Y
     */
    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
    /**
     * Додатковий конструктор для копіювання.
     * @param other Об'єкт Position для копіювання.
     */
    public Position(Position other) {
        this.x = other.x;
        this.y = other.y;
    }
    /**
     * Метод, який встановлює позицію x
     * @param x Розміщення по x
     */
    public void setX(int x) { this.x = x; }
    /**
     * Метод, який встановлює позицію y
     * @param y Розміщення по y
     */
    public void setY(int y) { this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    /**
     * Метод, який виводить позицію y string форматі
     */
    @Override
    public String toString() {  return "Position: (x=" + x + ", y=" + y + ")";  }
}
```

Результат виконання програми:

```
>>-- Creating character
New item in inventory: Tasty potato
First stat: Character [Health=100, Position: (x=0, y=0), Inventory: Tasty potato]
[GameHistory] Saving...
>> Character.save() -> Saved

>>-- Changing stats
Moving to x=10 y=5
Picked item: Iron sword
New item in inventory: Iron sword
New stat: Character [Health=100, Position: (x=10, y=5), Inventory: Tasty potato, Iron sword]
[GameHistory] Saving...
>> Character.save() -> Saved

>>-- Changing stats
Damage taken: 30
Stat after damage: Character [Health=70, Position: (x=10, y=5), Inventory: Tasty potato, Iron sword]

>>-- Using Memento (Undo)
using Undo (before taking damage)...
>> Character.save() -> Saved
>> Character.restore() -> Restored!
[GameHistory] Undo.
Returned stats: Character [Health=100, Position: (x=10, y=5), Inventory: Tasty potato, Iron sword]

using second Undo (to the start stats)...
>> Character.save() -> Saved
>> Character.restore() -> Restored!
[GameHistory] Undo.
Returned stats: Character [Health=100, Position: (x=0, y=0), Inventory: Tasty potato]

>>-- Using Memento (Redo)
Using Redo (get back to 'find sword')...
>> Character.save() -> Saved
>> Character.restore() -> Restored!
[GameHistory] Redo.
Returned stats: Character [Health=100, Position: (x=10, y=5), Inventory: Tasty potato, Iron sword]

>>-- Using Visitor (Adding operation) ---
Adding new operation
>> Character.accept() -> Visitor allowed to visit me
>- Character summary (Visitor):
- Health: 100
- Position: Position: (x=10, y=5)
- Inventory: Inventory: Tasty potato, Iron sword
```

Рисунок 1 – результат виконання програми згідно першого завдання

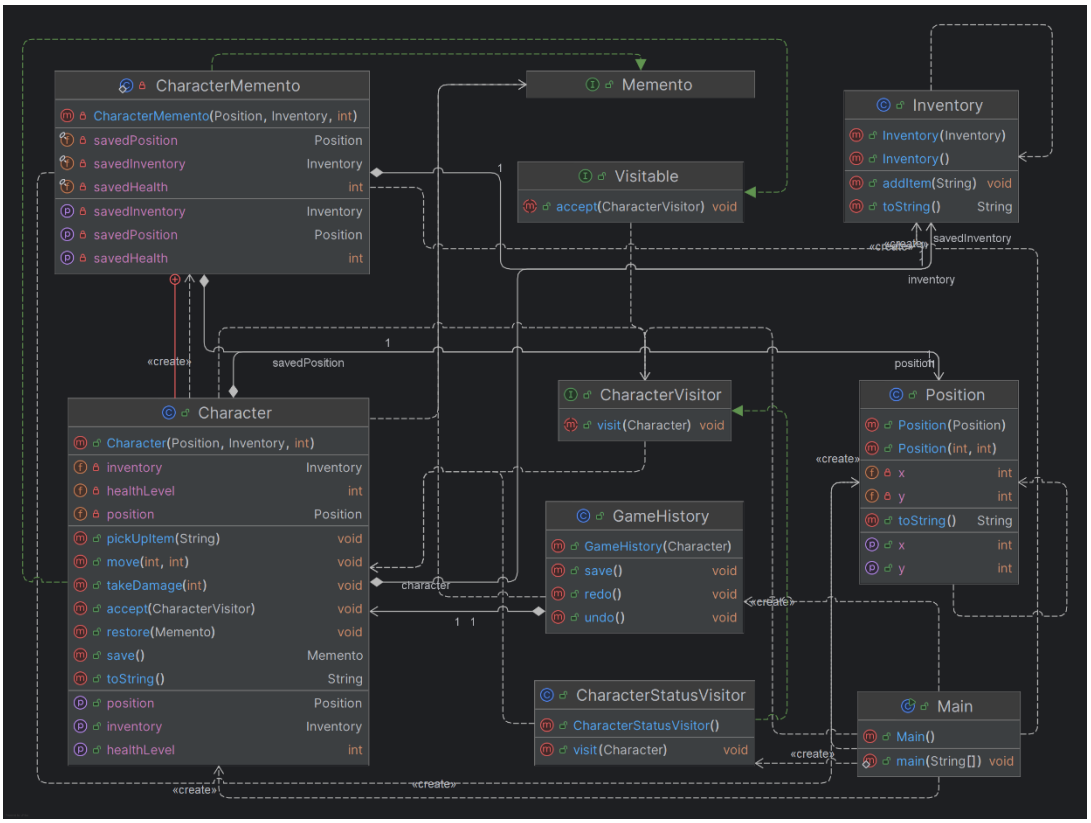

Рисунок 2 – Побудована в IntelliJ IDEA UML діаграма для першого завдання

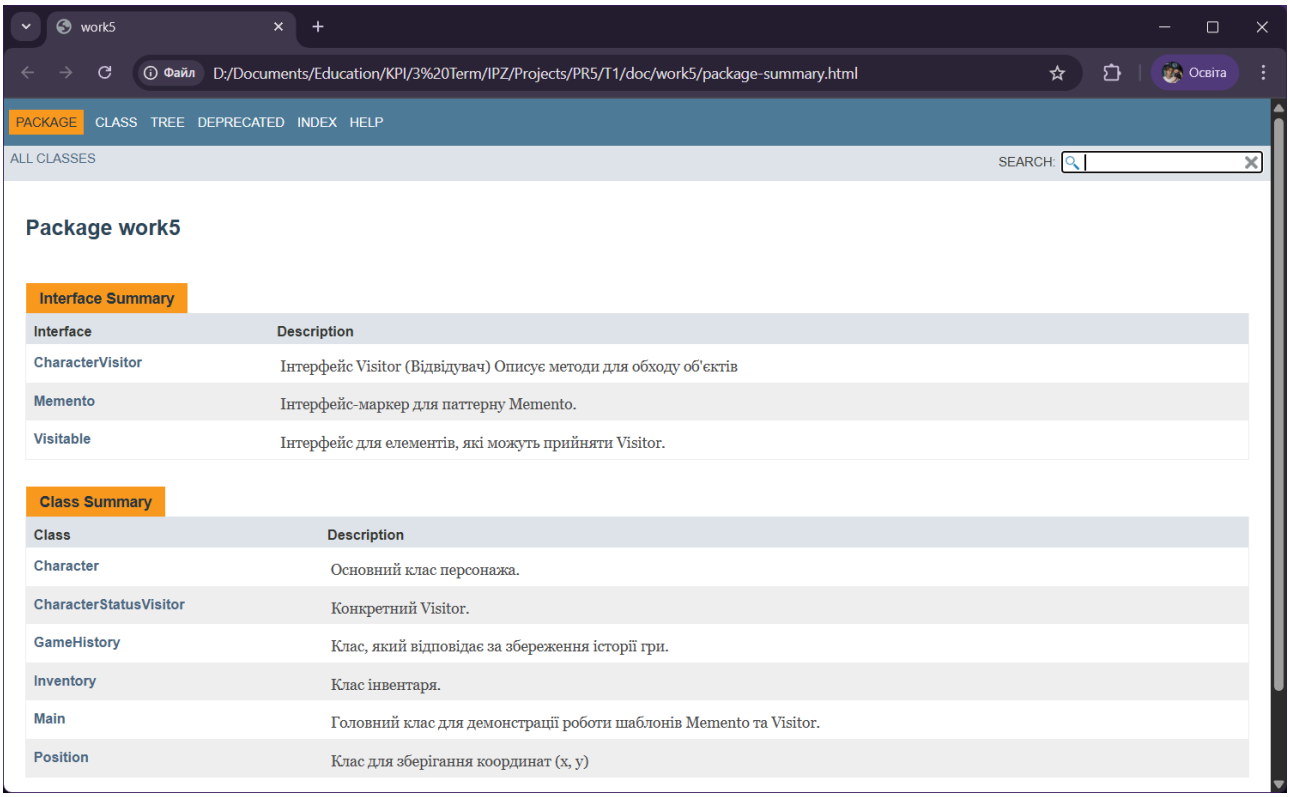

Рисунок 3 – Створена документація у форматі веб сторінки

Варіант 0 до завдання 2:

- Необхідно створити модель TCP-з'єднання, яке може перебувати в одному з кількох станів:
  - LISTENING – очікування вхідних підключень;
  - ESTABLISHED – активне з'єднання;
  - CLOSED – закрите з'єднання.
- TCPConnection – клас, який представляє TCP-з'єднання і делегує виконання дій об'єкту, що описує поточний стан.
- Вимоги:
  - Створити інтерфейс ConnectionState, який описує дії, які можуть виконуватися над TCP-з'єднанням (наприклад, open(), close(), send(data), receive()).
  - Кожен стан реалізує цей інтерфейс і визначає конкретну поведінку для цих операцій.
  - Реалізувати окремі класи для кожного стану TCP-з'єднання: ListeningState, EstablishedState, ClosedState. Кожен клас визначає, як TCP-з'єднання поводиться у своєму стані (наприклад, у стані CLOSED відправка даних заборонена, у LISTENING не можна отримати дані тощо).
  - Клас TCPConnection містить посилання на поточний стан. Усі дії TCPConnection делегуються поточному стану, що дозволяє змінювати поведінку динамічно, просто змінюючи стан об'єкта.

TCPConnection повинен мати можливість перемикатися між станами в залежності від подій (відкриття з'єднання, закриття, прийом/відправка даних). Не використовувати великі умовні оператори (if-else або switch) для визначення поведінки.

- У методі main продемонструвати:
  - Створити TCP-з'єднання у стані LISTENING.
  - Виконати дії open(), send(data), receive(), close() та показати, що поведінка змінюється в залежності від поточного стану.
  - Перемикати стан TCPConnection на ESTABLISHED і CLOSED та демонструвати зміну дій без використання умовних операторів.

Код до завдання 2:

Вміст файлу main.java:

```
package work5;
/**
 * Демонстраційний клас для шаблону State
 */
public class Main {
    public static void main(String[] args) {
        System.out.println(">>>- Creating TCP-Connection (Start state: LISTENING)");
        TCPConnection connection = new TCPConnection();
        System.out.println("\n>>-- Tasks in state LISTENING");
        connection.send("Hello??");
        connection.receive();
        connection.open();
        System.out.println(">- task in state ESTABLISHED");
        connection.open();
        connection.send("Are there anyone alive..?");
        connection.receive();
        connection.close();
        System.out.println(">- Tasks in state CLOSED");
        connection.send("Please.. HELP ME!!");
        connection.receive();
        connection.close();
        System.out.println("\n>>-- Changing state to LISTENING");
        connection.open();
        System.out.println(">- Tasks in LISTENING");
        connection.send("I'm still waiting..");
    }
}
```

Вміст файлу ConnectionState.java:

```
package work5;
/**
 * Інтерфейс стану з'єднання (State).
 * Визначає методи, які мають реалізовувати конкретні стани
 */
public interface ConnectionState {
    /**
     * Метод для відкриття з'єднання.
     * @param connection Контекст (з'єднання, що змінює стан)
     */
    void open(TCPConnection connection);
```

```
    /**
     * Метод для закриття з'єднання
     * @param connection Контекст
     */
    void close(TCPConnection connection);
    /**
     * Метод для надсилання даних
     * @param connection Контекст
     * @param data Дані для відправки
     */
    void send(TCPConnection connection, String data);
    /**
     * Метод для отримання даних.
     * @param connection Контекст
     */
    void receive(TCPConnection connection);
}
```

Вміст файлу EstablishedState.java:

```
package work5;

/**
 * Конкретний стан "З'єднано" (Established).
 * Дозволяє обмін даними.
 */
public class EstablishedState implements ConnectionState {
    @Override
    public void open(TCPConnection connection) {
        System.out.println("[!ERROR]: Connection is already opened");
    }

    @Override
    public void close(TCPConnection connection) {
        System.out.println("[ESTABLISHED] Closing connection...");
        connection.setState(new ClosedState());
        System.out.println(">> State changed: ESTABLISHED -> CLOSED");
    }

    @Override
    public void send(TCPConnection connection, String data) {
        System.out.println("[ESTABLISHED] Sending data: '" + data + "'");
    }

    @Override
    public void receive(TCPConnection connection) {
        System.out.println("[ESTABLISHED] Data received from client");
    }
}
```

Вміст файлу ListeningState.java:

```
package work5;

/**
 * Конкретний стан "Очікування" (Listening).
 * Дозволяє встановити з'єднання, але не передавати дані.
 */
public class ListeningState implements ConnectionState {

    @Override
```

```
    public void open(TCPConnection connection) {
        System.out.println("[LISTENING] Connection established.");
        connection.setState(new EstablishedState());
        System.out.println(">> State changed: LISTENING -> ESTABLISHED");
    }

    @Override
    public void close(TCPConnection connection) {
        System.out.println("[LISTENING] Closing server socket..");
        connection.setState(new ClosedState());
        System.out.println(">> State changed: LISTENING -> CLOSED");
    }

    @Override
    public void send(TCPConnection connection, String data) {
        System.out.println("[!ERROR]: Cannot send data while listening");
    }

    @Override
    public void receive(TCPConnection connection) {
        System.out.println("[!ERROR]: No data received yet (waiting for connection)");
    }
}
```

Вміст файлу ClosedState.java:

```
package work5;

/**
 * Конкретний стан "Закрито" (Closed).
 * Забороняє передачу даних, дозволяє лише повторне відкриття
 */
public class ClosedState implements ConnectionState {

    @Override
    public void open(TCPConnection connection) {
        System.out.println("[CLOSED] Re-opening port for listening.");
        connection.setState(new ListeningState());
        System.out.println(">> State changed: CLOSED -> LISTENING");
    }

    @Override
    public void close(TCPConnection connection) {
        System.out.println("[!ERROR]: Connection is already closed");
    }

    @Override
    public void send(TCPConnection connection, String data) {
        System.out.println("[!ERROR] Cannot send data while connection is closed");
    }

    @Override
    public void receive(TCPConnection connection) {
        System.out.println("[!ERROR] Cannot receive data while connection is closed");
    }
}
```

Вміст файлу TCPConnection.java:

```
package work5;

/**
 * Клас, що представляє TCP-з'єднання (Context).
 * Зберігає поточний стан і передає йому виконання операцій
 */
public class TCPConnection {
    private ConnectionState currentState;

    /**
     * Конструктор. Встановлює початковий стан LISTENING
     */
    public TCPConnection() {
        this.currentState = new ListeningState();
        System.out.println("[TCPConnection] Initialized. State: LISTENING");
    }

    /**
     * Метод для відкриття з'єднання.
     */
    public void open() {
        this.currentState.open(this);
    }

    /**
     * Метод для закриття з'єднання.
     */
    public void close() {
        this.currentState.close(this);
    }

    /**
     * Метод для надсилання даних.
     */
    public void send(String data) {
        this.currentState.send(this, data);
    }

    /**
     * Метод для отримання даних.
     */
    public void receive() {
        this.currentState.receive(this);
    }

    /**
     * Дозволяє об'єктам стану змінювати поточний стан Контексту.
     * @param newState Новий стан
     */
    protected void setState(ConnectionState newState) {
        this.currentState = newState;
    }
}
```

Результат виконання програми:

```
>>- Creating TCP-Connection (Start state: LISTENING)
[TCPConnection] Initialized. State: LISTENING

>>-- Tasks in state LISTENING
[!ERROR]: Cannot send data while listening
[!ERROR]: No data received yet (waiting for connection)
[LISTENING] Connection established.
>> State changed: LISTENING -> ESTABLISHED
>- task in state ESTABLISHED
[!ERROR]: Connection is already opened
[ESTABLISHED] Sending data: 'Are there anyone alive..?'
[ESTABLISHED] Data received from client
[ESTABLISHED] Closing connection...
>> State changed: ESTABLISHED -> CLOSED
>- Tasks in state CLOSED
[!ERROR] Cannot send data while connection is closed
[!ERROR] Cannot receive data while connection is closed
[!ERROR]: Connection is already closed

>>-- Changing state to LISTENING
[CLOSED] Re-opening port for listening.
>> State changed: CLOSED -> LISTENING
>- Tasks in LISTENING
[!ERROR]: Cannot send data while listening
```

Рисунок 4 – результат виконання програми згідно другого завдання

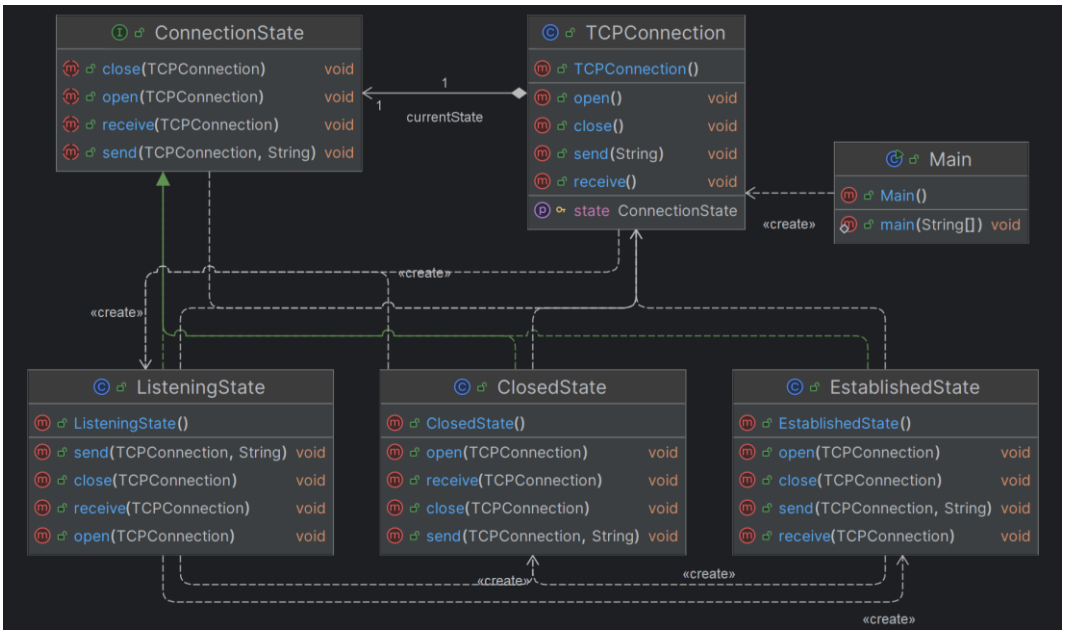

Рисунок 5 – Побудована в IntelliJ IDEA UML діаграма для другого завдання

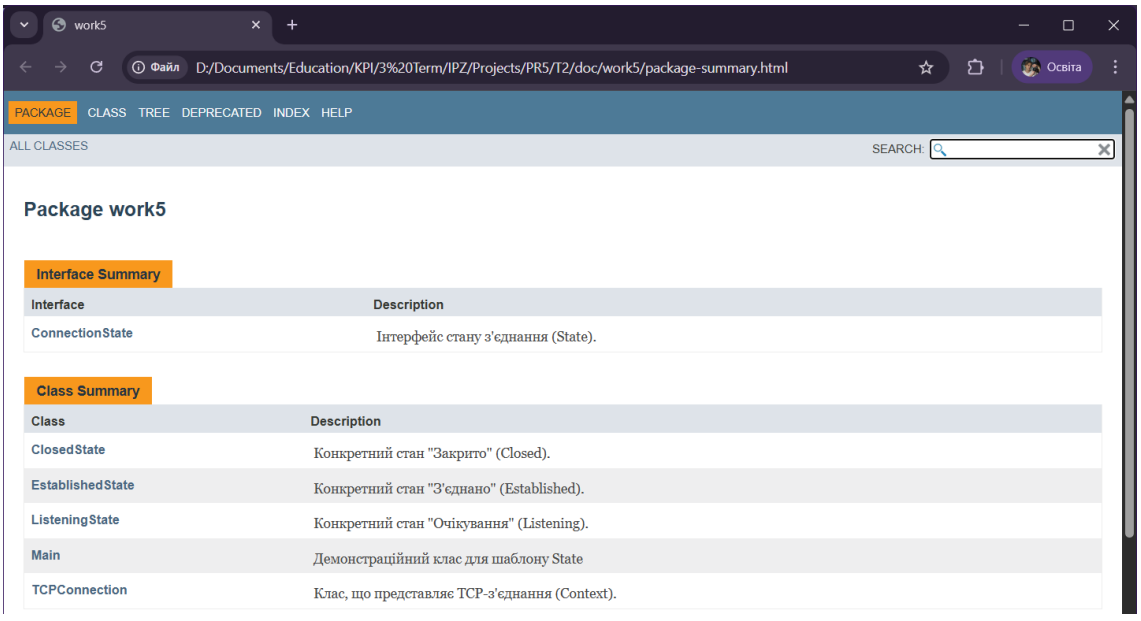

Рисунок 6 – Створена документація у форматі веб сторінки

Висновки:

Отже, у практичній роботі було реалізовано два приклади застосування поведінкових шаблонів. У першому завданні створено модель ігрового персонажа з можливістю збереження та відновлення стану через шаблон Memento, а також розширення функціональності без зміни існуючих класів за допомогою Visitor. У другому завданні побудовано модель TCP-з'єднання, що змінює поведінку відповідно до поточного стану на основі шаблону State. Увесь код міститься на даному GitHub репозиторії: https://github.com/BronievM/KPI-IPZ-2025/tree/main/PR5