Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №2
з дисципліни «Технології розроблення програмного забезпечення»
Тема: «Основи проектування»

Виконав:
студент групи ІА-34
Тунік О.

Перевірив:
асистент кафедри ІСТ
Мягкий М.Ю.

Київ 2025

Зміст

Теоретичні відомості 3
Хід роботи 5
Поставлене завдання 5
Діаграма варіантів використання 5
Діаграма класів системи 5
Сценарії використання 6
Вихідні коди класів системи 8
Структура бази даних 14
Відповіді на контрольні запитання 14
Висновки 17

Мета: Обрати зручну систему побудови UML-діаграм та навчитися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.

Теоретичні відомості

Мова UML є загальноцільовою мовою візуального моделювання, яка розроблена для специфікації, візуалізації, проєктування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем [3]. Мова UML є досить строгим та потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних

моделей складних систем різного цільового призначення. Ця мова увібрала в себе найкращі якості та досвід методів програмної інженерії, які з успіхом використовувалися протягом останніх років при моделюванні великих та складних систем.

З погляду методології ООАП (об'єктно-орієнтованого аналізу та проєктування) досить повна модель складної системи є певною кількістю взаємопов'язаних уявлень (views), кожне з яких відображає аспект поведінки або структури системи. У цьому найзагальнішими уявленнями складної системи прийнято вважати статичне і динамічне, які у своє чергу можуть поділятися інші більш приватні.

Принцип ієрархічної побудови моделей складних систем передбачає розгляд процес побудови моделей на різних рівнях абстрагування або деталізації в рамках фіксованих уявлень.

Рівень представлення (layer) – спосіб організації та розгляду моделі на одному рівні абстракції, що представляє горизонтальний зріз архітектури моделі, тоді як розбиття представляє її вертикальний зріз.

При цьому вихідна або початкова модель складної системи має найбільш загальне уявлення та відноситься до концептуального рівня. Така модель, що отримала назву концептуальної, будується на початковому етапі проєктування і може не містити багатьох деталей та аспектів системи, що моделюється. Наступні моделі конкретизують концептуальну модель, доповнюючи її уявленнями логічного та фізичного рівня.

Загалом процес ООАП можна розглядати як послідовний перехід від розробки найбільш загальних моделей та уявлень концептуального рівня до більш приватних і детальних уявлень логічного та фізичного рівня. У цьому кожному етапі ООАП дані моделі послідовно доповнюються дедалі більше деталей, що дозволяє їм адекватно відбивати різні аспекти конкретної реалізації складної системи.

В рамках мови UML уявлення про модель складної системи фіксуються у вигляді спеціальних графічних конструкцій, що отримали назву діаграм.

Діаграма (diagram) – графічне уявлення сукупності елементів моделі у формі зв'язкового графа, вершинам і ребрам (дугам) якого приписується певна семантика. Нотація канонічних діаграм є основним засобом розробки моделей мовою UML.

У нотації мови UML визначено такі види діаграм:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);
- компонентів (component diagram);
- розгортання (deployment diagram).

Перелічені діаграми є невід'ємною частиною графічної нотації мови UML. Понад те, процес ООАП нерозривно пов'язаний з процесом побудови цих діаграм. При цьому сукупність побудованих таким чином діаграм є самодостатньою в тому сенсі, що в них міститься вся інформація, яка потрібна для реалізації проєкту складної системи.

Кожна з цих діаграм деталізує та конкретизує різні уявлення про модель складної системи у термінах мови UML. При цьому діаграма варіантів використання являє собою найбільш загальну концептуальну модель складної системи, яка є вихідною для побудови інших діаграм. Діаграма класів, за своєю суттю, логічна модель, що відбиває статичні аспекти структурної побудови складної системи.

Діаграми кооперації та послідовностей є різновидами логічної моделі, які відображають динамічні аспекти функціонування складної системи. Діаграми станів та діяльності призначені для моделювання поведінки системи. І, нарешті, діаграми компонентів і розгортання служать уявлення фізичних компонентів складної системи і тому представляють її фізичну модель.

Хід роботи

Поставлене завдання

Обрали тему «Flexible automation tool».

Патерни які потрібно використати: strategy, command, abstract factory, facade, interpreter, SOA.

Опис теми: Інструмент автоматизації повинен забезпечувати найпростіші автоматичні дії для зручності користувача: завантаження нових фільмів / книг / файлів при випуску (наприклад, щоп'ятниці з'являються нові серії улюблених серіалів); встановити статуси в комунікаторах (discord – idle при нульовій активності на тривалий час) і т.д. Автоматизація забезпечується шляхом введення правил (на зразок IFTTT.com сервісу), запису макросів (натискання клавіш, дії миші), планувальника завдань (о 5 ранку – початок роздачі торрент-файлів).

Діаграма варіантів використання

Проаналізувавши тему, проєктуємо діаграму варіантів використання відповідно до обраної теми лабораторного циклу (рис. 1).

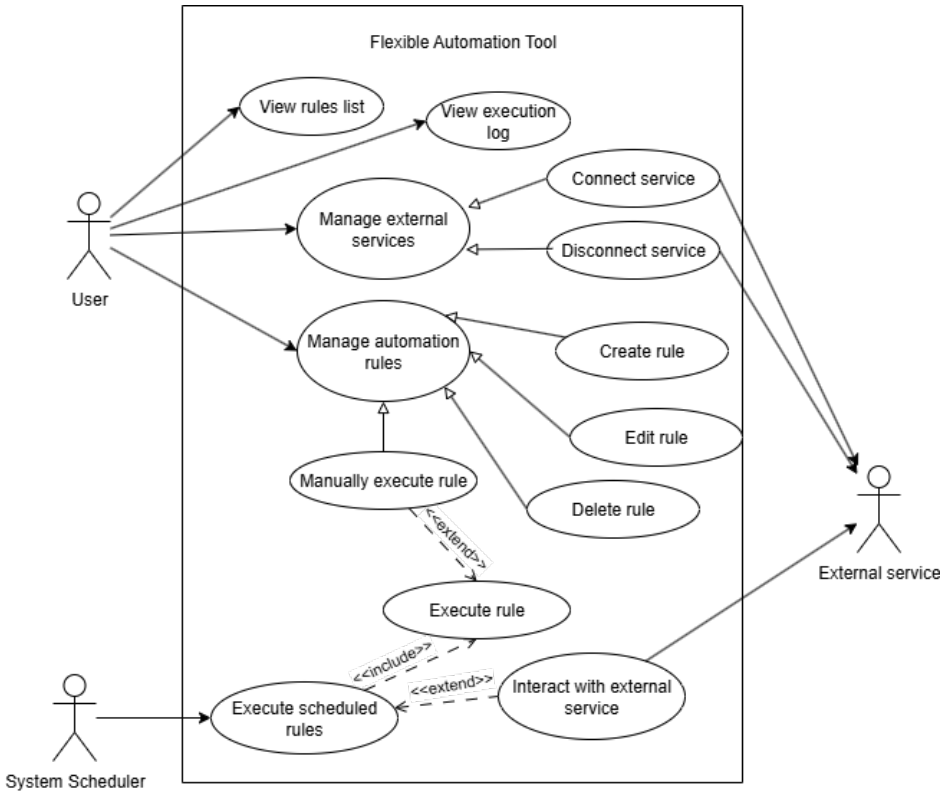

Рисунок 1. Use case діаграма

Діаграма класів системи

Проєктуємо діаграму класів предметної області (рис.2).

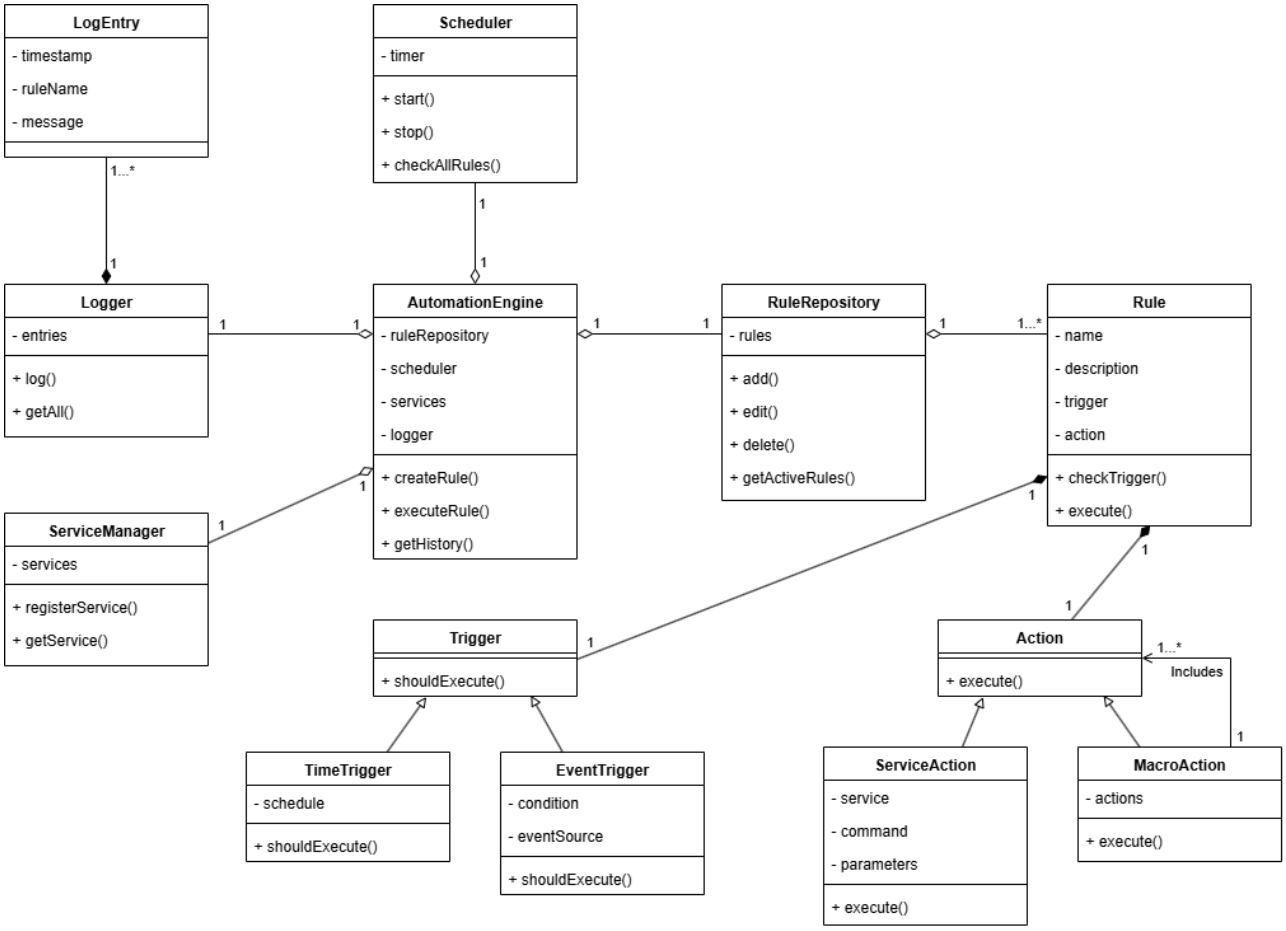

Рисунок 2. Діаграма класів системи

Зв’язки:

- AutomationEngine агрегує RuleRepository, Scheduler, ServiceManager, Logger (агрегація)
- Rule композиційно містить Trigger і Action (композиція — тригери й дії існують лише в межах правила)
- Trigger — абстрактний клас; TimeTrigger і EventTrigger наслідують його (наслідування)
- Action —клас; ServiceAction і MacroAction наслідують його (наслідування)
- MacroAction включає множину Action (асоціація — макрос включає кілька дій).
- RuleRepository композиційно містить колекцію Rule (композиція, зберігає правила, керує їхнім життєвим циклом)

Сценарії використання

Вибираємо 3 варіанти використання та напишемо за ними сценарії використання:

Варіант використання 1: «Створення правила автоматизації»

Передумови: Користувач запустив систему.

Постумови:

- Нове правило збережене у сховищі правил (RuleRepository).
- Правило готове до виконання при виконанні тригера.

Взаємодіючі сторони: Користувач, AutomationEngine, RuleRepository.

Короткий опис: Користувач створює нове правило автоматизації, що складається з тригера (умови запуску) та дії (виконуваної команди/сервісу).

Основний потік подій:

1. Користувач обирає опцію «Створити нове правило».
2. Система запитує назву та опис правила.
3. Користувач задає тип тригера (наприклад, «за розкладом» або «подія системи») та налаштовує параметри.
4. Користувач задає дію (ServiceAction чи MacroAction).
5. AutomationEngine перевіряє коректність введених даних.
6. RuleRepository зберігає створене правило.

Винятки:

- Якщо параметри тригера некоректні → система повідомляє про помилку, користувач виправляє введення.
- Якщо збереження у сховище неможливе → система виводить повідомлення про помилку, правило не створюється.

Варіант використання 2: «Автоматичне виконання правила за розкладом»

Передумови: У системі вже існують активні правила з типом TimeTrigger. Scheduler запущений.

Постумови:

- Виконана дія, визначена в правилі.
- У Logger збережений запис про виконання.

Взаємодіючі сторони: Scheduler, AutomationEngine, Rule, Logger, ServiceManager.

Короткий опис: Планувальник перевіряє правила з тригером «за часом» і запускає відповідні дії.

Основний потік подій:

1. Scheduler перевіряє всі активні правила у RuleRepository.

2. Знаходить правило, чий TimeTrigger відповідає поточному часу.
3. AutomationEngine викликає Rule.execute().
4. Rule виконує дію (наприклад, ServiceAction для запуску сервісу).
5. ServiceManager знаходить потрібний сервіс і виконує команду.
6. Logger записує інформацію про виконання правила.

Винятки:

- Якщо сервіс не знайдено → система повідомляє у Logger про помилку, дія не виконується.
- Якщо виникла внутрішня помилка виконання → Logger зберігає відповідний запис.

Варіант використання 3: «Перегляд історії виконаних правил»

Передумови: У Logger є записи про виконання правил.

Постумови:

- Користувач бачить список подій (час, правило, результат).

Взаємодіючі сторони: Користувач, AutomationEngine, Logger.

Короткий опис: Користувач переглядає журнал виконаних правил.

Основний потік подій:

1. Користувач обирає опцію «Переглянути історію».
2. AutomationEngine звертається до Logger.getAll().
3. Logger надає список LogEntry.
4. Система відображає історію виконання з усіма параметрами (дата/час, назва правила, результат).

Винятки:

- Якщо історія порожня → система повідомляє: «Записів не знайдено».

Вихідні коди класів системи

Лістинг 1 – Клас AutomationEngine

```
public class AutomationEngine
{
    private readonly RuleRepository _ruleRepository;
    private readonly Scheduler _scheduler;
    private readonly ServiceManager _serviceManager;
    private readonly Logger _logger;
```

```
    public AutomationEngine(RuleRepository ruleRepository, Scheduler scheduler,
ServiceManager serviceManager, Logger logger)
    {
        _ruleRepository = ruleRepository;
        _scheduler = scheduler;
        _serviceManager = serviceManager;
        _logger = logger;
    }

    public void CreateRule(Rule rule)
    {
        _ruleRepository.Add(rule);
        _logger.Log(rule.Name, "Rule created.");
    }

    public void ExecuteRule(Rule rule)
    {
        rule.Execute();
        _logger.Log(rule.Name, "Rule executed.");
    }

    public IEnumerable<LogEntry> GetHistory()
    {
        return _logger.GetAll();
    }

    public void Start()
    {
        _scheduler.Start();
        _logger.Log("SYSTEM", "Scheduler started.");
    }

    public void Stop()
    {
        _scheduler.Stop();
        _logger.Log("SYSTEM", "Scheduler stopped.");
    }
}
```

Лістинг 2 – Клас RuleRepository

```
public class RuleRepository
{
    private readonly List<Rule> _rules = new List<Rule>();

    public void Add(Rule rule) => _rules.Add(rule);

    public void Edit(Rule rule)
    {
        var existing = _rules.FirstOrDefault(r => r.Id == rule.Id);
        if (existing != null)
        {
            existing.Name = rule.Name;
            existing.Description = rule.Description;
            existing.Trigger = rule.Trigger;
            existing.Action = rule.Action;
        }
    }

    public void Delete(int id)
    {
        var rule = _rules.FirstOrDefault(r => r.Id == id);
        if (rule != null) _rules.Remove(rule);
    }
```

```
    public IEnumerable<Rule> GetActiveRules() => _rules;
}
```

Лістинг 3 – Клас Rule

```
[Table("rules")]
public class Rule
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [Required]
    public string Name { get; set; } = string.Empty;

    public string? Description { get; set; }

    // Зв'язок з Trigger
    public int TriggerId { get; set; }
    [ForeignKey("TriggerId")]
    public virtual Trigger Trigger { get; set; }

    // Зв'язок з Action
    public int ActionId { get; set; }
    [ForeignKey("ActionId")]
    public virtual Action Action { get; set; }

    public void CheckTrigger()
    {
        if (Trigger.ShouldExecute())
        {
            Execute();
        }
    }

    public void Execute()
    {
        Action.Execute();
    }
}
```

Лістинг 4 – Абстрактний клас Trigger і його нащадки

```
[Table("triggers")]
public abstract class Trigger
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public abstract bool ShouldExecute();
    public abstract bool Validate();
}

[Table("time_triggers")]
public class TimeTrigger : Trigger
{
    public string Schedule { get; set; } = string.Empty;

    public override bool ShouldExecute()
    {
        // Тут можна вставити логіку: перевірка дати/часу
        return false;
```

```
    }

    public override bool Validate() => !string.IsNullOrEmpty(Schedule);
}

[Table("event_triggers")]
public class EventTrigger : Trigger
{
    public string EventSource { get; set; } = string.Empty;
    public string Condition { get; set; } = string.Empty;

    public override bool ShouldExecute()
    {
        // Логіка: перевірка умови події
        return false;
    }

    public override bool Validate() => !string.IsNullOrEmpty(EventSource);
}
```

Лістинг 5 – Абстрактний клас Action і його нащадки

```
[Table("actions")]
public abstract class Action
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public abstract void Execute();
    public abstract bool Validate();
}

[Table("service_actions")]
public class ServiceAction : Action
{
    public string ServiceType { get; set; } = string.Empty;
    public string Command { get; set; } = string.Empty;
    public string Parameters { get; set; } = string.Empty;

    public override void Execute()
    {
        // Виконання команди для зовнішнього сервісу
    }

    public override bool Validate() => !string.IsNullOrEmpty(Command);
}

[Table("macro_actions")]
public class MacroAction : Action
{
    public virtual ICollection<Action> Actions { get; set; } = new List<Action>();

    public override void Execute()
    {
        foreach (var action in Actions)
        {
            action.Execute();
        }
    }

    public override bool Validate() => Actions.Count > 0;
}
```

Лістинг 6 – Клас Scheduler

```
public class Scheduler
{
    private Timer? _timer;
    private readonly RuleRepository _ruleRepository;
    private readonly AutomationEngine _engine;

    public Scheduler(RuleRepository ruleRepository, AutomationEngine engine)
    {
        _ruleRepository = ruleRepository;
        _engine = engine;
    }

    public void Start()
    {
        _timer = new Timer(CheckAllRules, null, 0, 1000 * 60);
    }

    public void Stop()
    {
        _timer?.Dispose();
    }

    private void CheckAllRules(object? state)
    {
        foreach (var rule in _ruleRepository.GetActiveRules())
        {
            rule.CheckTrigger();
        }
    }
}
```

Лістинг 7 – Клас ServiceManager

```
public class ServiceManager
{
    private readonly Dictionary<string, object> _services = new();

    public void RegisterService(string name, object service)
    {
        if (!_services.ContainsKey(name))
            _services[name] = service;
    }

    public object? GetService(string name)
    {
        return _services.TryGetValue(name, out var service) ? service : null;
    }
}
```

Лістинг 8 – Класи Logger та LogEntry

```
[Table("log_entries")]
public class LogEntry
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    public DateTime Timestamp { get; set; } = DateTime.Now;

    public string RuleName { get; set; } = string.Empty;
```

```
    public string Message { get; set; } = string.Empty;
}

public class Logger
{
    private readonly List<LogEntry> _entries = new();

    public void Log(string ruleName, string message)
    {
        _entries.Add(new LogEntry
        {
            RuleName = ruleName,
            Message = message,
            Timestamp = DateTime.Now
        });
    }

    public IEnumerable<LogEntry> GetAll() => _entries;
}
```

Лістинг 9 – DDL скрипт БД

```
CREATE TABLE Actions (
    Id AUTOINCREMENT PRIMARY KEY,
    ActionType TEXT(50),
    ServiceType TEXT(100),
    Command TEXT(100),
    Parameters MEMO
);

CREATE TABLE Rules (
    Id AUTOINCREMENT PRIMARY KEY,
    Name TEXT(100),
    Description MEMO,
    TriggerId LONG,
    ActionId LONG,
    IsActive YESNO
);

CREATE TABLE Triggers (
    Id AUTOINCREMENT PRIMARY KEY,
    TriggerType TEXT(50),
    Schedule TEXT(255),
    EventSource TEXT(255),
    Condition TEXT(255)
);

CREATE TABLE MacroActionItems (
    Id AUTOINCREMENT PRIMARY KEY,
    MacroActionId LONG,
    ChildActionId LONG,
    OrderIndex INTEGER
);

CREATE TABLE Services (
```

```
    Id AUTOINCREMENT PRIMARY KEY,
    Name TEXT(100),
    ServiceType TEXT(100),
    Config MEMO
);

CREATE TABLE ExecutionHistory (
    Id AUTOINCREMENT PRIMARY KEY,
    RuleId LONG,
    ExecutedAt DATETIME,
    Status TEXT(50),
    Message MEMO
);
```

Структура бази даних

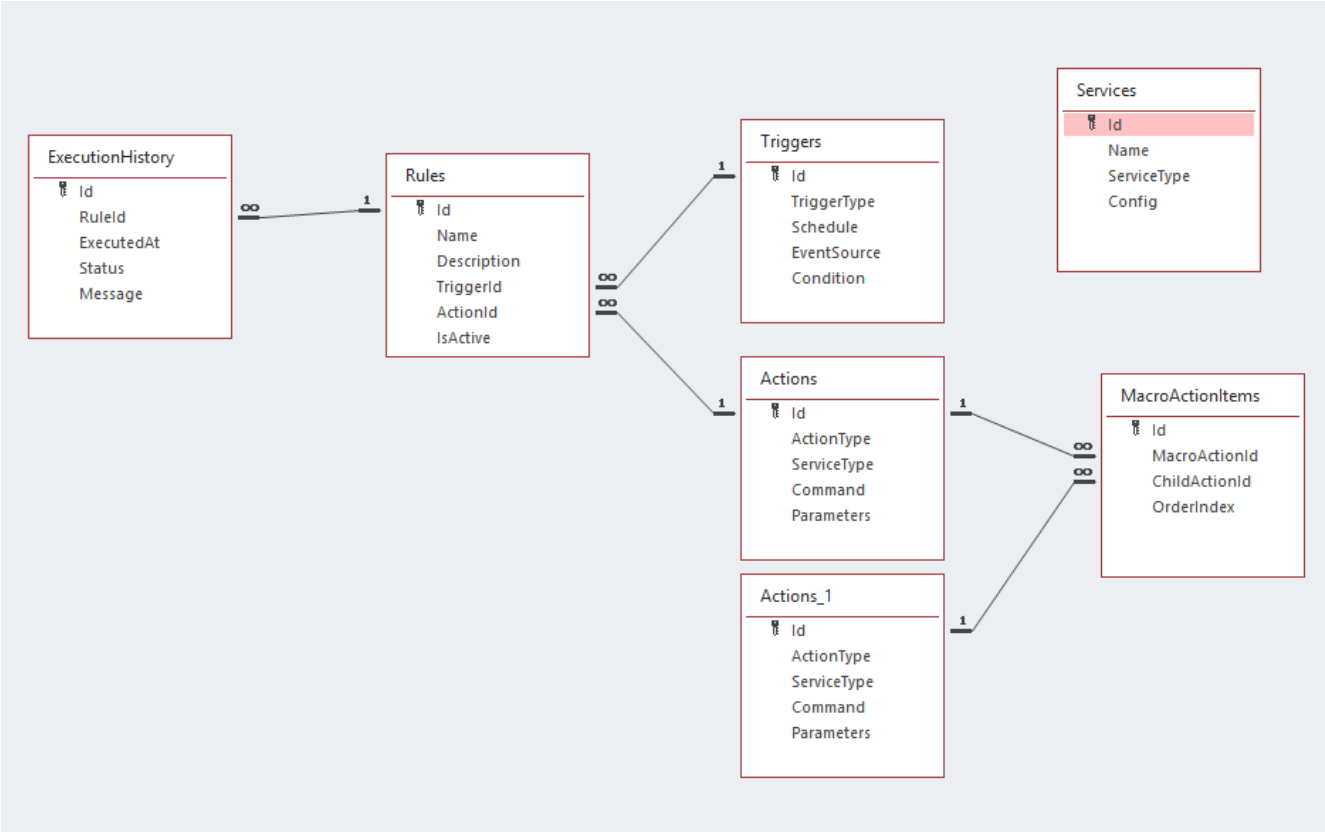

Рисунок 3. Зображення структури бази даних системи

Відповіді на контрольні запитання

1. Що таке UML?

   UML (Unified Modeling Language) — це уніфікована мова моделювання для опису структури та поведінки програмних систем.

2. Що таке діаграма класів UML?

Це діаграма, що показує класи системи, їхні атрибути, методи та зв’язки між ними.

3. Які діаграми UML називають канонічними?

   До канонічних належать основні типи діаграм UML: діаграми класів, варіантів використання, послідовностей, станів, діяльності та компонентів.

4. Що таке діаграма варіантів використання?

   Це діаграма, що показує взаємодію користувачів (акторів) із системою через варіанти використання (use cases).

5. Що таке варіант використання?

   Варіант використання — це опис певного сценарію взаємодії користувача із системою для досягнення конкретної мети.

6. Які відношення можуть бути відображені на діаграмі використання?

   Використовуються відношення асоціації, узагальнення, а також залежності типу *include* та *extend*.

7. Що таке сценарій?

   Сценарій — це послідовність кроків, що описує конкретний перебіг виконання варіанта використання.

8. Що таке діаграма класів?

   Це структурна UML-діаграма, яка показує статичні зв’язки між класами системи.

9. Які зв’язки між класами ви знаєте?

   Асоціація, агрегація, композиція, наслідування та залежність.

10. Чим відрізняється композиція від агрегації?

    Композиція означає сильну залежність — якщо головний об’єкт знищується, знищуються і його частини. Агрегація — слабша форма зв’язку, частини можуть існувати окремо.

11. Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів?

    Агрегація позначається порожнім ромбом, а композиція — зафарбованим ромбом біля класу-цілого.

12. Що являють собою нормальні форми баз даних?

    Це правила структуризації таблиць, які усувають надлишковість і забезпечують логічну цілісність даних.

13. Що таке фізична модель бази даних? Логічна?

Фізична модель описує, як дані зберігаються у конкретній СУБД; логічна — які сутності та зв’язки існують між ними незалежно від реалізації.

14. Який взаємозв'язок між таблицями БД та програмними класами?

Кожна таблиця зазвичай відповідає окремому класу, а рядки таблиці — це об’єкти цього класу.

Висновки

У ході лабораторної роботи було проаналізовано тему «Flexible Automation Tool», спроєктовано діаграму варіантів використання та діаграму класів, визначено основні зв’язки між компонентами системи. Реалізовано структуру класів із використанням шаблонів проєктування (Strategy, Command, Abstract Factory, Facade, Interpreter), а також розроблено модель бази даних для збереження правил, дій і тригерів. Отримана система є основою для створення гнучкого інструменту автоматизації користувацьких завдань.