Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №5

Технології розроблення програмного забезпечення

«Патерни проектування. Паттерн «Adapter»»

«System activity monitor»

Виконав
студент групи ІА–32:
Феклістов Д.С.

Київ 2025

Зміст

Варіант

Варіант №10: VCS all-in-one (Проєкт: Система моніторингу системної активності).

Мета роботи: Ознайомитися з призначенням структурного патерну проектування **Адаптер (Adapter)**, вивчити його структуру та застосувати для інтеграції несумісного інтерфейсу зовнішньої бібліотеки (LegacyMetricsLib) у клас моніторингу системи (SystemMonitor).

Теоретичні відомості

Патерн Адаптер (Adapter)

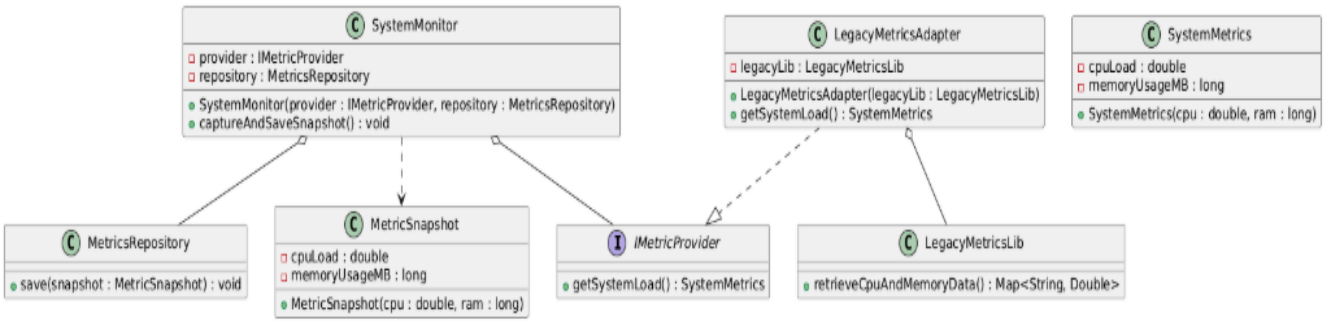
Призначення: Адаптер (Перехідник) — це **структурний** патерн, який дозволяє об'єктам з несумісними інтерфейсами працювати разом. Він створює обгортку навколо класу-джерела (Adaptee), перетворюючи його інтерфейс на той, який очікує клас-споживач (Client).

Ключові учасники:

1. **Client (SystemMonitor)**: Клас, який використовує функціональність і очікує **Target**-інтерфейс.
2. **Target (IMetricProvider)**: Інтерфейс, який Adapter повинен реалізувати. Це стандартизований інтерфейс.
3. **Adaptee (LegacyMetricsLib)**: Існуючий клас, який має потрібні дані, але його інтерфейс несумісний з Target.
4. **Adapter (LegacyMetricsAdapter)**: Клас, який реалізує **Target** і містить екземпляр **Adaptee**, виконуючи перетворення даних.

Діаграма Класів

Діаграма класів відображає архітектурну роль патерну **Адаптер**, показуючи, як клас **SystemMonitor** (Client) взаємодіє виключно з інтерфейсом **IMetricProvider** (Target), не знаючи про існування несумісної бібліотеки **LegacyMetricsLib** (Adaptee).



Хід роботи

Робота полягала в реалізації патерну **Адаптер на рівні об'єктів** для забезпечення сумісності існуючого класу моніторингу SystemMonitor з новою (застарілою) бібліотекою збору метрик LegacyMetricsLib.

Основні етапи:

1. **Створення Target-інтерфейсу:** Визначено **IMetricProvider** та уніфікований об'єкт даних **SystemMetrics** для стандартизації взаємодії.
2. **Створення Adaptee:** Реалізовано клас **LegacyMetricsLib**, що імітує несумісне зовнішнє джерело даних.
3. **Створення Adapter:** Клас **LegacyMetricsAdapter** реалізував **IMetricProvider** та отримав посилання на **LegacyMetricsLib**.
4. **Адаптація даних:** В методі getSystemLoad() **Адаптер** виконав конвертацію одиниць вимірювання (байти RAM -> мегабайти RAM) та перепакування даних у формат **SystemMetrics**.
5. **Інверсія залежностей:** Клас **SystemMonitor** був модифікований для прийому будь-якого об'єкта, що реалізує **IMetricProvider**, дотримуючись принципу DIP.
6. **Зв'язування:** У Main.java відбулася ін'єкція залежності: створений **LegacyMetricsAdapter** був переданий у **SystemMonitor**.

```
=== System Monitor: Ініціалізовано ===

--- Запуск збору метрик ---
-> [Legacy] Отримання сирих даних з застарілої бібліотеки...
-> [Adapter] Дані успішно адаптовані та конвертовані.
Знімок системи збережено: [Snapshot] CPU: 42,0%, RAM: 8192 MB
--- Збір завершено ---
```

Код Програми

Нижче наведено код для нових та модифікованих файлів, що реалізують патерн Адаптер.

Модифікація файлу Main.java

```
import monitor.SystemMonitor;
import monitor.LegacyMetricsLib;
import monitor.LegacyMetricsAdapter;
import service.MetricsRepository;

public class Main {
  public static void main(String[] args) {

    // 0. Ініціалізація існуючих компонентів
    MetricsRepository repository = new MetricsRepository();

    // 1. Створення Adaptee
    LegacyMetricsLib legacyLib = new LegacyMetricsLib();

    // 2. Створення Adapter, обгортаючи Adaptee
    LegacyMetricsAdapter metricsProvider = new
LegacyMetricsAdapter(legacyLib);

    // 3. Створення Client (SystemMonitor), передача йому Adapter
    SystemMonitor monitor = new SystemMonitor(metricsProvider, repository);

    // Запуск функціоналу Client
    monitor.captureAndSaveSnapshot();
  }
}
```

Модифіковані та Нові Файли у папці `monitor/`

`monitor/SystemMonitor.java` (Client)

```
package monitor;
import service.MetricsRepository;
import data.MetricSnapshot;

// SystemMonitor тепер залежить лише від IMetricProvider (Target)
public class SystemMonitor {

  // ДОДАНО: Залежність від інтерфейсу Target
  private final IMetricProvider provider;
  private final MetricsRepository repository;
  // ЗМІНЕНО: Конструктор приймає IMetricProvider
  public SystemMonitor(IMetricProvider provider, MetricsRepository repository) {
      this.provider = provider;
      this.repository = repository;
      System.out.println("=== System Monitor: Ініціалізовано ===");
  }
  // ЗМІНЕНО: Виклик методу через IMetricProvider
  public void captureAndSaveSnapshot() {
      System.out.println("\n--- Запуск збору метрик ---");
      // Виклик через уніфікований інтерфейс (це може бути Adapter або будь-який інший провайдер)
      SystemMetrics metrics = provider.getSystemLoad();

      // Створення MetricSnapshot (потребує імпорту data.MetricSnapshot)
      MetricSnapshot snapshot = new MetricSnapshot(metrics.getCpuLoad(), metrics.getMemoryUsageMB());
      repository.save(snapshot);

      System.out.println("Знімок системи збережено: " + snapshot.toString());
```

```
        System.out.println("--- Збір завершено ---");
    }

    // ... Інші методи (generateReport, що використовує ітератор)
```

monitor/IMetricProvider.java (Target)

```
package monitor;

// Цільовий інтерфейс, який буде використовувати SystemMonitor
public interface IMetricProvider {
    SystemMetrics getSystemLoad();
}
```

monitor/SystemMetrics.java (Target Data Object)

```
package monitor;

// Уніфікований об'єкт даних, який очікує SystemMonitor
public class SystemMetrics {
    private final double cpuLoad;
    private final long memoryUsageMB;

    public SystemMetrics(double cpuLoad, long memoryUsageMB) {
        this.cpuLoad = cpuLoad;
        this.memoryUsageMB = memoryUsageMB;
    }

    public double getCpuLoad() {
        return cpuLoad;
    }
    public long getMemoryUsageMB() {
        return memoryUsageMB;
```

```
    }

    @Override
    public String toString() {
        return String.format("CPU: %.1f%%, RAM: %d MB", cpuLoad, memoryUsageMB);
    }
}
```

monitor/LegacyMetricsLib.java (Adaptee)

```
package monitor;


import java.util.Map;


// Стороння/Застаріла бібліотека з несумісним інтерфейсом
public class LegacyMetricsLib {


    // Несумісний метод: повертає Map<String, Double>
    public Map<String, Double> retrieveCpuAndMemoryData() {
        System.out.println("-> [Legacy] Отримання сирих даних з застарілої бібліотеки...");
        // Імітація даних з нестандартними ключами
        return Map.of(
                "cpu_usage_percent", 42.0,
                "memory_total_bytes", 8589934592.0 // 8 GB в байтах
        );
    }
```

```
}
```

monitor/LegacyMetricsAdapter.java (Adapter)

```
package monitor;

import java.util.Map;

// Адаптер: реалізує Target і обгортає Adaptee
public class LegacyMetricsAdapter implements IMetricProvider {

    // Композиція: містить екземпляр Adaptee
    private final LegacyMetricsLib legacyLib;

    public LegacyMetricsAdapter(LegacyMetricsLib legacyLib) {
        this.legacyLib = legacyLib;
    }

    @Override
    public SystemMetrics getSystemLoad() {
        // 1. Виклик несумісного методу Adaptee
        Map<String, Double> rawData = legacyLib.retrieveCpuAndMemoryData();

        // 2. Логіка адаптації:
        double cpu = rawData.get("cpu_usage_percent");

        // Конвертація RAM з байтів у мегабайти
        long ramBytes = rawData.get("memory_total_bytes").longValue();
        long ramMB = ramBytes / (1024 * 1024);

        System.out.println("-> [Adapter] Дані успішно адаптовані та конвертовані.");

        // 3. Повернення об'єкта у форматі Target
        return new SystemMetrics(cpu, ramMB);
    }
}
```

Висновок

В ході виконання лабораторної роботи було успішно реалізовано структурний патерн **Адаптер (Adapter)**, що дозволило інтегрувати несумісний зовнішній компонент (LegacyMetricsLib) у модуль моніторингу (SystemMonitor).

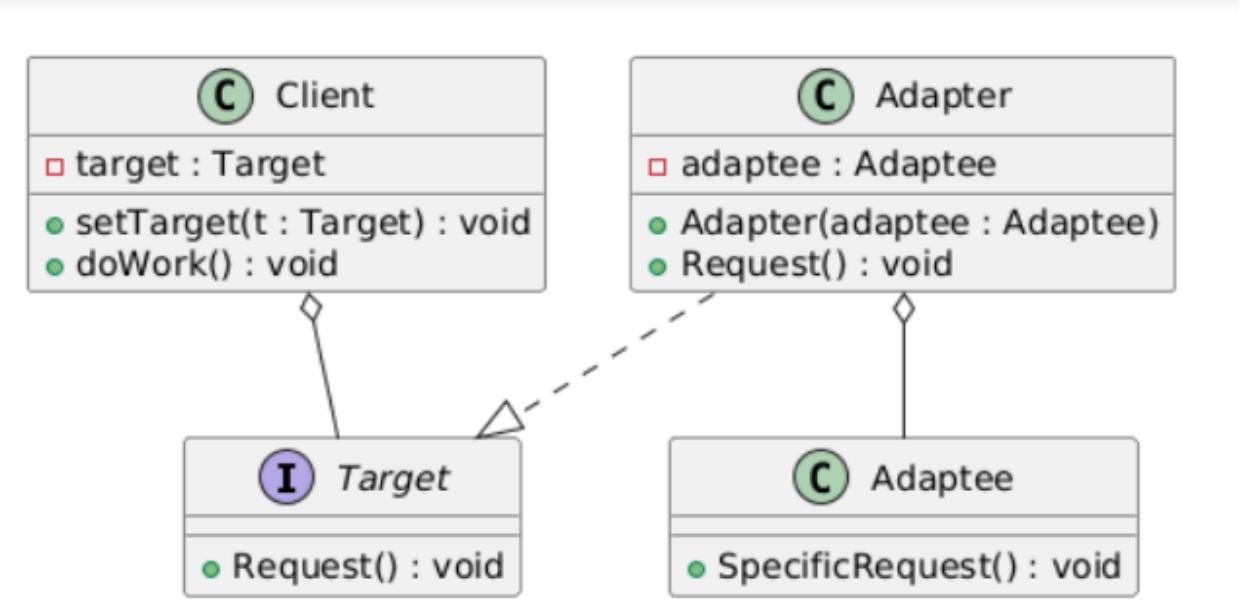
1. Створено клас **LegacyMetricsAdapter**, який виступив як посередник, перетворивши несумісний формат даних на стандартизований об'єкт **SystemMetrics**.
2. Клас **SystemMonitor** (Client) був успішно абстрагований від деталей реалізації збору метрик і залежить лише від інтерфейсу **IMetricProvider** (Target).
3. Таким чином, досягнуто підвищення гнучкості системи та дотримано **Принципу інверсії залежностей (DIP)**.

Відповіді на питання

1. Призначення шаблону «Адаптер»

Адаптер перетворює інтерфейс одного класу на інший, з яким уміє працювати клієнт. Це «перехідник», що дозволяє з’єднати несумісні компоненти без змін у їхньому коді, забезпечуючи взаємодію через узгоджений інтерфейс.

2. Нарисуйте структуру шаблону «Адаптер»



3. Класи в «Адаптері» і їхня взаємодія

- **Client:** Використовує лише цільовий інтерфейс.
- **Target:** Інтерфейс, який очікує клієнт.
- **Adaptee:** Існуючий клас із «непідходящим» інтерфейсом, який ми не змінюємо.
- **Adapter:** Реалізує Target і містить посилання на Adaptee, транслюючи виклики.
- **Взаємодія:** Client звертається до Adapter як до Target; Adapter переформатовує виклик і делегує його Adaptee.

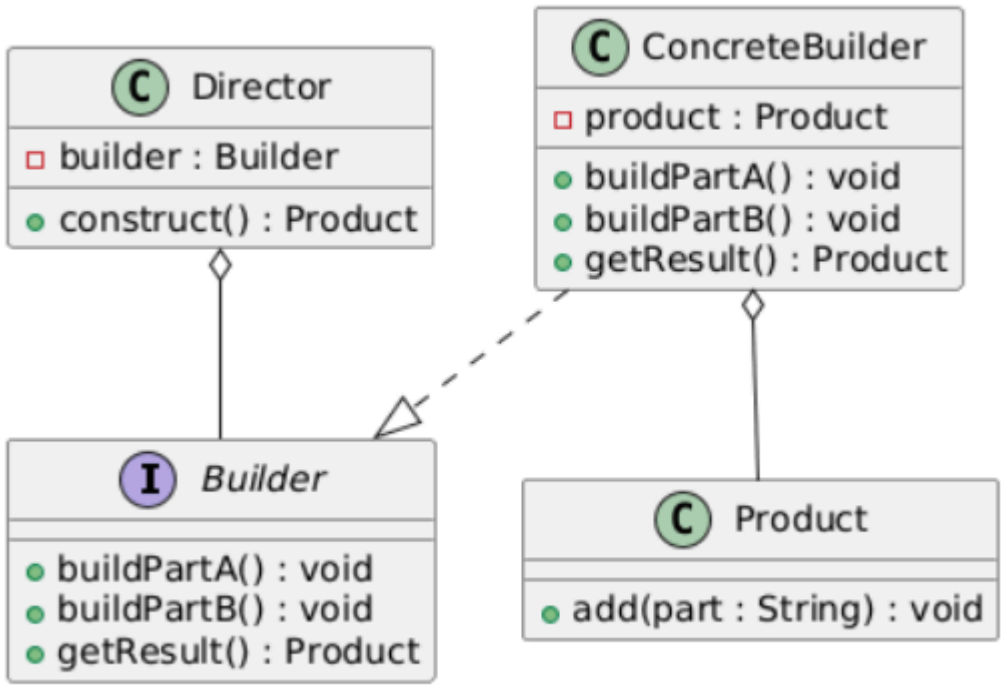
4. Різниця між адаптером об’єктів та класів

- **Об’єктний адаптер (композиція):** Тримає екземпляр Adaptee і реалізує Target. Гнучкий варіант для більшості мов (зокрема Java), працює навіть із final-класами.
- **Класовий адаптер (спадкування):** Успадковує Adaptee й одночасно реалізує Target. Жорсткіший, обмежений множинним спадкуванням і тісно прив’язаний до конкретного Adaptee.

5. Призначення шаблону «Будівельник»

Будівельник розділяє процес покрокового створення складного об’єкта від його представлення. Один і той самий сценарій побудови можна застосовувати для різних конфігурацій продукту.

6. Нарисуйте структуру шаблону «Будівельник»



7. Класи в «Будівельнику» і їхня взаємодія

- **Product:** Те, що будуємо.
- **Builder:** Описує кроки конструювання частин продукту.
- **ConcreteBuilder:** Реалізує кроки і збирає продукт, надає метод отримання результату.
- **Director:** Визначає послідовність викликів Builder.
- **Взаємодія:** Клієнт створює Director і ConcreteBuilder, передає Builder у Director; Director викликає кроки, клієнт забирає готовий Product з Builder.

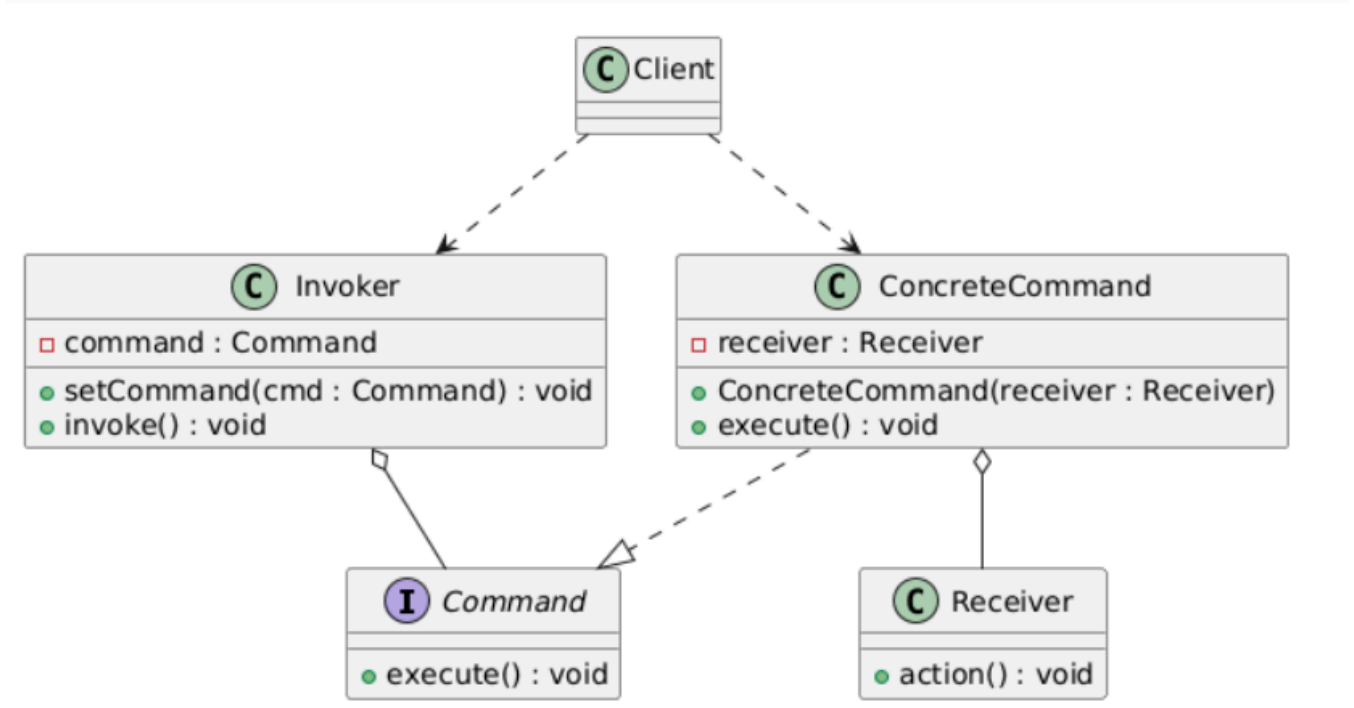
8. Коли застосовувати «Будівельник»

- **Багато параметрів:** Заміна громіздких конструкторів із купою опцій.
- **Складний процес:** Будівництво в кілька кроків із фіксованим порядком.
- **Різні варіанти:** Потреба створювати різні конфігурації одного продукту одним процесом.

9. Призначення шаблону «Команда»

Команда пакує дію і її контекст в окремий об’єкт, що дозволяє передавати, ставити в чергу, логувати операції та підтримувати Undo/Redo без знання викликачем деталей виконання.

10. Нарисуйте структуру шаблону «Команда»



11. Класи в «Команді» і їхня взаємодія

- **Command:** Інтерфейс із `execute()` (і за потреби `undo()`).
- **ConcreteCommand:** Тримає Receiver і реалізує виконання.
- **Receiver:** Містить бізнес-логіку, реально виконує дію.
- **Invoker:** Запускає команду, не знаючи деталей Receiver.
- **Взаємодія:** Клієнт пов’язує команду з отримувачем і передає її викликачу; Invoker викликає `execute()`, ConcreteCommand делегує в Receiver.

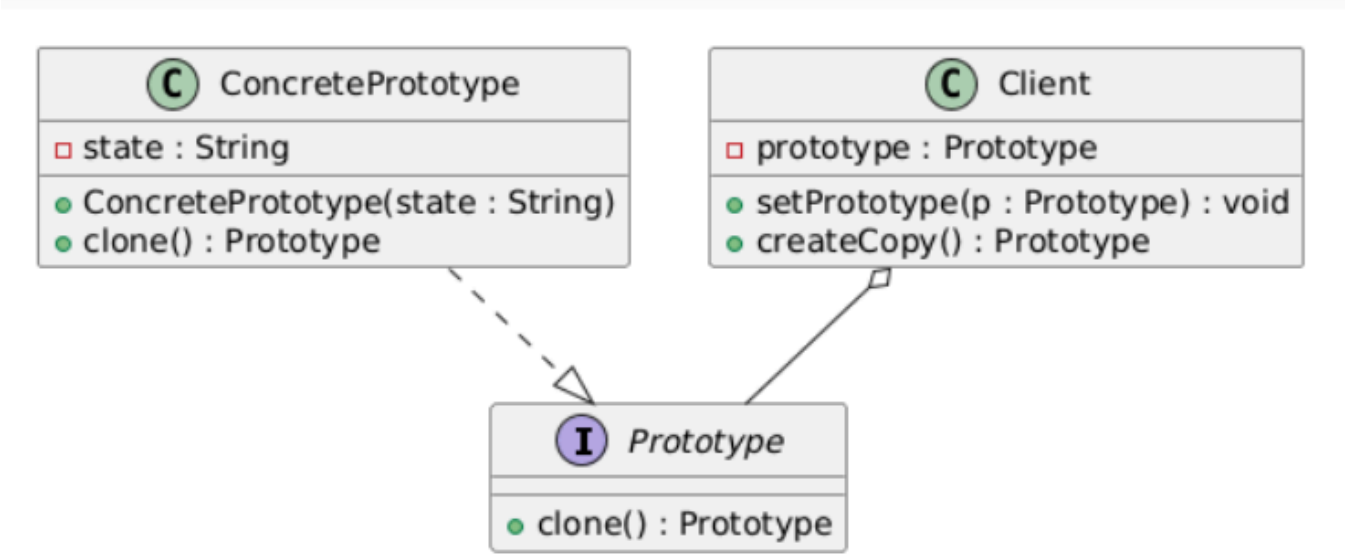
12. Як працює «Команда»

1. Клієнт створює ConcreteCommand з Receiver.
2. Передає команду Invoker’у.
3. Invoker викликає `execute()` у відповідь на подію.
4. Команда делегує дію конкретному методу Receiver (і може зберігати стан для `undo()`).

13. Призначення шаблону «Прототип»

Прототип дозволяє створювати нові об’єкти шляхом клонування наявного екземпляра, що економить час і ресурси ініціалізації, особливо для складних об’єктів.

14. Нарисуйте структуру шаблону «Прототип»



15. Класи в «Прототипі» і їхня взаємодія

- **Prototype:** Описує метод клонування.
- **ConcretePrototype:** Реалізує копіювання власного стану.
- **Взаємодія:** Клієнт створює новий об’єкт через `clone()` із наявного прототипу (поверхневе чи глибоке копіювання залежно від реалізації).

16. Приклади використання «Ланцюжка відповідальності»

- **HTTP middleware:** Послідовні обробники — автентифікація, авторизація, кеш, контролер.
- **Логування:** Рівні важливості (DEBUG, INFO, ERROR) послідовно обробляють повідомлення.
- **Техпідтримка:** Ескалація тікета від першої лінії до спеціалістів до вирішення.