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 №7

«Патерни проектування»

Виконав
студент групи ІА-34:
Сльота Максим

Перевірив:
Мягкий М. Ю.

Київ - 2025

Тема: Патерни проектування.

Мета роботи: Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

Вихідні дані:

17. **System activity monitor** (iterator, command, abstract factory, bridge, visitor, SOA)

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

Теоретичні відомості:

Хід роботи:

1. Діаграма класів, яка представляє використання шаблону Bridge

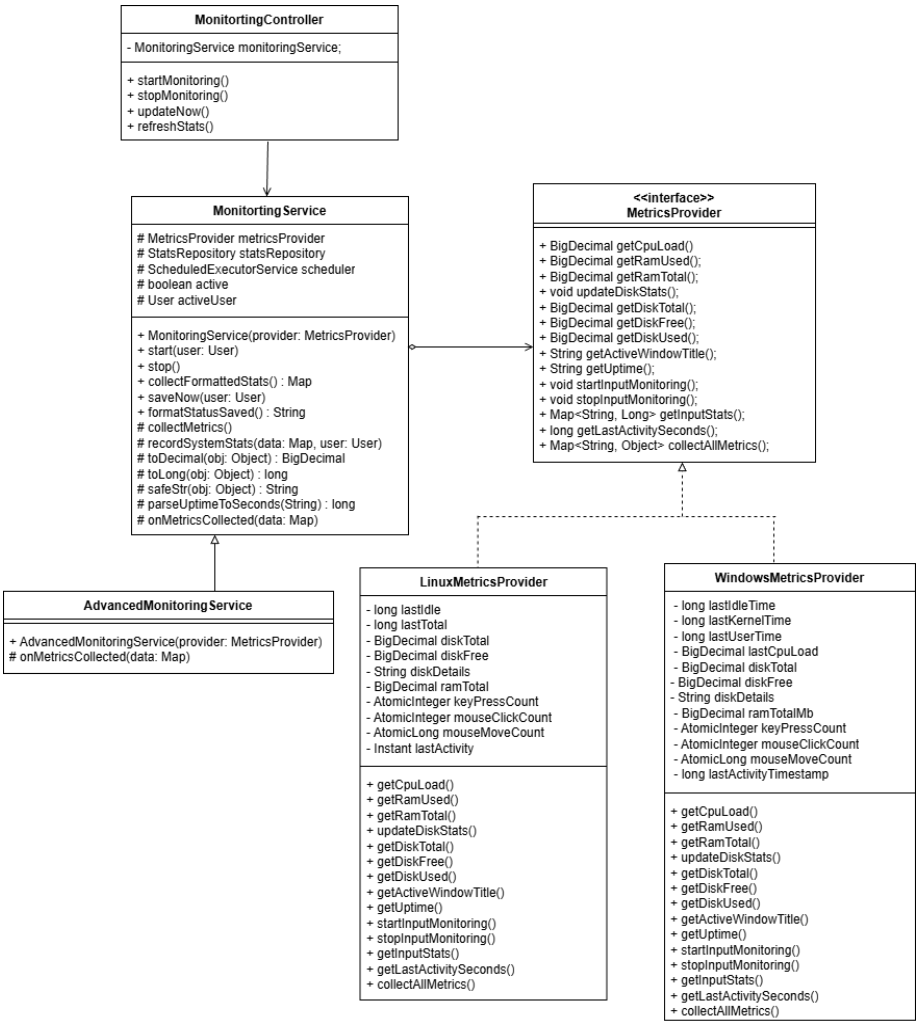


Рис 1. Діаграма класів

У проєкті System Activity Monitor застосовано шаблон «Міст» для того, щоб розділити загальну логіку моніторингу системи і конкретні способи отримання системних метрик на різних операційних системах. Такий підхід дозволяє будувати гнучку й масштабовану архітектуру, де високорівнева функціональність не залежить від платформи, на якій працює застосунок. Основна ідея полягає в тому, що логіка «що робити» і логіка «як це робиться» існують у різних класах і розвиваються незалежно одна від одної.

Абстрактна частина шаблона реалізована у вигляді класу MonitoringService. Він визначає загальний алгоритм роботи системи моніторингу: запуск і зупинку процесу збору даних, періодичне отримання метрик, збереження статистики та основні допоміжні операції. При цьому MonitoringService не знає нічого про специфіку отримання даних на Windows чи Linux. Він має лише посилання на інтерфейс MetricsProvider, через який отримує потрібну інформацію.

Конкретна поведінка, пов'язана зі збором системних характеристик, винесена в окремі реалізації інтерфейсу MetricsProvider. У проєкті існують два класи:

WindowsMetricsProvider і LinuxMetricsProvider. Кожен із них відповідає за роботу зі своєю платформою й містить логіку доступу до системних ресурсів тієї ОС, на якій виконується програма.

Клас AdvancedMonitoringService представляє собою розширену абстракцію. Він доповнює базову логіку моніторингу додатковими аналітичними можливостями: визначенням критичних навантажень, виявленням нестачі пам’яті або дискового простору та формуванням попереджень. Завдяки тому, що AdvancedMonitoringService успадковує MonitoringService, він зберігає повний доступ до всіх стандартних механізмів.

Роль клієнта виконує MonitoringController, який ініціює роботу моніторингу, використовуючи абстракцію, а не конкретну реалізацію. Контролер отримує відповідний сервіс, визначаючи операційну систему динамічно під час виконання програми. Таким чином, клієнт взаємодіє лише з абстрактним інтерфейсом і повністю ізольований від платформних деталей.

Доданий вихідний код системи:

MetricsProvider.java - описує інтерфейс, який визначає набір методів для отримання системних метрик.

```
public interface MetricsProvider {
   BigDecimal getCpuLoad();
   BigDecimal getRamUsed();
   BigDecimal getRamTotal();
   void updateDiskStats();
   BigDecimal getDiskTotal();
   BigDecimal getDiskFree();
   BigDecimal getDiskUsed();
   String getActiveWindowTitle();
   String getUptime();
   void startInputMonitoring();
   void stopInputMonitoring();
   Map<String, Long> getInputStats();
   long getLastActivitySeconds();
   Map<String, Object> collectAllMetrics();
}
```

LinuxMetricsProvider.java - реалізує інтерфейс MetricsProvider для операційної системи Linux.

```
public class LinuxMetricsProvider implements MetricsProvider {
    private long lastIdle = 0;
    private long lastTotal = 0;
    private BigDecimal diskTotal = BigDecimal.ZERO;
    private BigDecimal diskFree = BigDecimal.ZERO;
    private String diskDetails = "Unknown";
    private BigDecimal ramTotal = BigDecimal.ZERO;
    private final AtomicInteger keyPressCount = new AtomicInteger(0);
    private final AtomicInteger mouseClickCount = new AtomicInteger(0);
    private final AtomicLong mouseMoveCount = new AtomicLong(0);
    private volatile Instant lastActivity = Instant.now();
    private Thread inputThread;
    private volatile boolean inputActive = false;
    @Override
    public BigDecimal getCpuLoad() {
    }
    @Override
    public BigDecimal getRamUsed() {
    }
    @Override
    public BigDecimal getRamTotal() {
        return ramTotal;
    }
    @Override
    public void updateDiskStats() {
    }
    @Override
    public BigDecimal getDiskTotal() {
        return diskTotal;
    }
    @Override
    public BigDecimal getDiskFree() {
        return diskFree;
    }
    @Override
    public BigDecimal getDiskUsed() {
        return diskTotal.subtract(diskFree);
    }
```

```
    @Override
    public String getActiveWindowTitle() {
    }
    @Override
    public String getUptime() {
    }
    @Override
    public void startInputMonitoring() {
    }
    @Override
    public void stopInputMonitoring() {
    }
    @Override
    public Map<String, Long> getInputStats() {
    }
    @Override
    public long getLastActivitySeconds() {
    }
    @Override
    public Map<String, Object> collectAllMetrics() {
}
```

WindowsMetricsProvider.java - реалізація MetricsProvider для Windows.

```
public class WindowsMetricsProvider implements MetricsProvider {
    private ScheduledExecutorService inputScheduler;
    private static final int VK_LBUTTON = 0x01;
    private static final int VK_RBUTTON = 0x02;
    private int lastX = -1;
    private int lastY = -1;
    private static final int MOVE_THRESHOLD = 3;
    private long lastIdleTime = 0;
    private long lastKernelTime = 0;
    private long lastUserTime = 0;
    private volatile BigDecimal lastCpuLoad = BigDecimal.ZERO;
    private final ScheduledExecutorService cpuScheduler = Executors.newSingleThreadScheduledExecutor();
    private BigDecimal diskTotal = BigDecimal.ZERO;
    private BigDecimal diskFree = BigDecimal.ZERO;
    private String diskDetails = "Unknown";
    private BigDecimal ramTotalMb = BigDecimal.ZERO;
```

```
private final AtomicInteger keyPressCount = new AtomicInteger(0);
private final AtomicInteger mouseClickCount = new AtomicInteger(0);
private final AtomicLong mouseMoveCount = new AtomicLong(0);
private volatile long lastActivityTimestamp = System.currentTimeMillis();
private volatile boolean inputMonitoringActive = false;
public WindowsMetricsProvider() {
}
private long filetimeToLong(WinBase.FILETIME ft) {
   return ((long) ft.dwHighDateTime << 32) | (ft.dwLowDateTime & 0xffffffffL);
}
private void updateCpuLoad() {
}
@Override
public BigDecimal getCpuLoad() {
   return lastCpuLoad;
}
@Override
public BigDecimal getRamUsed() {
}
@Override
public BigDecimal getRamTotal() {
   return ramTotalMb;
}
@Override
public void updateDiskStats() {
   }
   diskTotal = BigDecimal.valueOf(total).setScale(2, RoundingMode.HALF_UP);
   diskFree = BigDecimal.valueOf(free).setScale(2, RoundingMode.HALF_UP);
   diskDetails = sb.length() == 0 ? "Unknown" : sb.toString().replaceAll("\\| $", "");
}
@Override
public BigDecimal getDiskTotal() {
   return diskTotal;
}
@Override
public BigDecimal getDiskFree() {
   return diskFree;
}
```

```
    @Override
    public BigDecimal getDiskUsed() {
        return diskTotal.subtract(diskFree);
    }
    @Override
    public String getActiveWindowTitle() {
    }
    @Override
    public String getUptime() {
            }
    @Override
    public void startInputMonitoring() {
    }
    private void checkInput() {
    }
    @Override
    public void stopInputMonitoring() {
        inputMonitoringActive = false;
        if (inputScheduler != null && !inputScheduler.isShutdown()) {
            inputScheduler.shutdownNow();
            inputScheduler = null;
        }
    }
    @Override
    public Map<String, Long> getInputStats() {
    }
    @Override
    public long getLastActivitySeconds() {
    }
    @Override
    public Map<String, Object> collectAllMetrics() {
}
```

MonitoringController.java - контролер JavaFX, який взаємодіє з користувачем.

```
public class MonitoringController {
    @FXML private Label cpuLabel, ramLabel, osLabel, windowLabel;
    @FXML private Label keysLabel, clicksLabel, movesLabel, uptimeLabel;
    @FXML private Label diskLabel, statusLabel;
    @FXML private Button startButton, stopButton;
```

```
private User activeUser;
private MonitoringService monitoringService;
private ScheduledExecutorService uiUpdater;
private boolean isMonitoring = false;
@FXML
public void initialize() {
    activeUser = Session.getCurrentUser();
    stopButton.setDisable(true);
    statusLabel.setText("Готово до запуску моніторингу");
}
@FXML
private void startMonitoring() {
    if (isMonitoring) {
        showAlert("Моніторинг уже запущено.");
        return;
    }
    try {
        SystemEnvironmentFactory factory = EnvironmentFactoryProducer.getFactory();
        monitoringService = factory.createMonitoringService();
        monitoringService.start(Session.isGuest() ? null : activeUser);
        isMonitoring = true;
        startButton.setDisable(true);
        stopButton.setDisable(false);
        startAutoUIUpdate();
        statusLabel.setText("Моніторинг активний");
    } catch (Exception e) {
        handleMonitoringError(e);
    }
}
@FXML
private void stopMonitoring() {
    try {
        if (!isMonitoring) return;
        monitoringService.stop();
        stopAutoUIUpdate();
        isMonitoring = false;
        startButton.setDisable(false);
        stopButton.setDisable(true);
```

```
            statusLabel.setText("Моніторинг зупинено.");
        } catch (Exception e) {
            handleMonitoringError(e);
        }
    }
    private void startAutoUIUpdate() {
        uiUpdater = Executors.newSingleThreadScheduledExecutor();
        uiUpdater.scheduleAtFixedRate(() ->
                        Platform.runLater(this::refreshStats),
                0, 5, TimeUnit.SECONDS);
    }
    private void stopAutoUIUpdate() {
        if (uiUpdater != null && !uiUpdater.isShutdown()) {
            uiUpdater.shutdownNow();
            uiUpdater = null;
        }
    }
    private void refreshStats() {
        try {
            Map<String, Object> data = monitoringService.collectFormattedStats();
            if (data == null || data.isEmpty()) return;
            cpuLabel.setText(data.get("cpuLoad") + " %");
            ramLabel.setText(
                    String.format("%s / %s MB",
                            data.get("ramUsed"),
                            data.get("ramTotal"))
            );
            osLabel.setText((String) data.get("osName"));
            windowLabel.setText((String) data.get("activeWindow"));
            uptimeLabel.setText("Uptime: " + data.get("uptime"));
            diskLabel.setText(
                    String.format("%.2f / %.2f GB  (%s)",
                            data.get("diskUsed"),
                            data.get("diskTotal"),
                            data.get("diskDetails"))
            );
            keysLabel.setText("Keys: " + data.get("keys"));
            clicksLabel.setText("Clicks: " + data.get("clicks"));
```

```
        movesLabel.setText("Moves: " + data.get("moves"));
    } catch (Exception e) {
        handleMonitoringError(e);
    }
}
@FXML
private void updateNow() {
    try {
        if (!Session.isGuest() && activeUser != null) {
            monitoringService.saveNow(activeUser);
            statusLabel.setText(monitoringService.formatStatusSaved());
        } else {
            statusLabel.setText("Гість: дані не збережено.");
        }
        refreshStats();
    } catch (Exception e) {
        handleMonitoringError(e);
    }
}
private void handleMonitoringError(Exception e) {
    if (monitoringService != null) monitoringService.stop();
    stopAutoUIUpdate();
    isMonitoring = false;
    startButton.setDisable(false);
    stopButton.setDisable(true);
    statusLabel.setText("Помилка: " + e.getMessage());
    showAlert("Помилка: " + e.getMessage());
    e.printStackTrace();
}
@FXML
private void goBack() {
    stopAutoUIUpdate();
    if (monitoringService != null) monitoringService.stop();
    switchScene("/fxml/main.fxml", "Main Menu");
}
private void switchScene(String fxml, String title) {
    try {
        FXMLLoader loader = new FXMLLoader(getClass().getResource(fxml));
```

```
            Scene scene = new Scene(loader.load());
            Stage stage = (Stage) cpuLabel.getScene().getWindow();
            stage.setScene(scene);
            stage.setTitle(title);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void showAlert(String text) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Повідомлення");
        alert.setHeaderText(null);
        alert.setContentText(text);
        alert.showAndWait();
    }
}
```

MonitotingService.java - організовує роботу моніторингу: періодичний збір метрик, збереження даних у базу, контроль потоків, обробку помилок.

```
public abstract class MonitoringService {
    protected final StatsRepository statsRepository = new StatsRepositoryImpl();
    protected final MetricsProvider metricsProvider;
    protected ScheduledExecutorService scheduler;
    protected volatile boolean active = false;
    protected User activeUser;
    private final ThreadFactory threadFactory = runnable -> {
        Thread t = new Thread(runnable);
        t.setName("MonitoringScheduler-" + t.getId());
        t.setDaemon(true);
        t.setUncaughtExceptionHandler((thr, ex) ->
                System.err.println("Uncaught exception in " + thr.getName() + ": " + ex));
        return t;
    };
    public MonitoringService(MetricsProvider provider) {
        this.metricsProvider = provider;
    }
    public synchronized void start(User user) {
        if (active) return;
        active = true;
```

```
    this.activeUser = user;
    scheduler = Executors.newScheduledThreadPool(2, threadFactory);
    scheduler.scheduleAtFixedRate(() -> safeGuard(this::collectMetrics),
            0, 5, TimeUnit.SECONDS);
    metricsProvider.startInputMonitoring();
    System.out.println("MonitoringService: моніторинг запущено.");
}
public synchronized void stop() {
    active = false;
    metricsProvider.stopInputMonitoring();
    if (scheduler != null && !scheduler.isShutdown()) {
        scheduler.shutdownNow();
    }
    System.out.println("Моніторинг зупинено.");
}
protected void safeGuard(Runnable task) {
    try {
        task.run();
    } catch (Exception e) {
        System.err.println("Exception in scheduled task: " + e.getMessage());
    }
}
protected void collectMetrics() {
    if (!active) return;
    Map<String, Object> data = metricsProvider.collectAllMetrics();
    onMetricsCollected(data);
    if (activeUser != null) {
        recordSystemStats(data, activeUser);
    }
}
public Map<String, Object> collectFormattedStats() {
    return metricsProvider.collectAllMetrics();
}
protected abstract void onMetricsCollected(Map<String, Object> data);
protected void recordSystemStats(Map<String, Object> data, User user) {
    try {
        BigDecimal cpu = toDecimal(data.get("cpuLoad"));
        BigDecimal ramUsed = toDecimal(data.get("ramUsed"));
```

```
        BigDecimal ramTotal = toDecimal(data.get("ramTotal"));
        BigDecimal diskTotal = toDecimal(data.get("diskTotal"));
        BigDecimal diskFree = toDecimal(data.get("diskFree"));
        BigDecimal diskUsed = toDecimal(data.get("diskUsed"));
        long uptimeSec = parseUptimeToSeconds(safeStr(data.get("uptime")));
        long keys = toLong(data.get("keys"));
        long clicks = toLong(data.get("clicks"));
        long moves = toLong(data.get("moves"));
        String window = safeStr(data.get("activeWindow"));
        SystemStats stats = new SystemStats();
        stats.setUser(user);
        stats.setCpuLoad(cpu);
        stats.setRamUsedMb(ramUsed);
        stats.setRamTotalMb(ramTotal);
        stats.setActiveWindow(window);
        stats.setKeyboardPresses((int) keys);
        stats.setMouseClicks((int) clicks);
        stats.setMouseMoves(moves);
        stats.setSystemUptimeSeconds(uptimeSec);
        stats.setDiskTotalGb(diskTotal);
        stats.setDiskFreeGb(diskFree);
        stats.setDiskUsedGb(diskUsed);
        statsRepository.save(stats);
    } catch (Exception e) {
        System.err.println("[MonitoringService] Помилка збереження метрик: " + e.getMessage());
    }
}
protected BigDecimal toDecimal(Object obj) {
    if (obj instanceof BigDecimal bd) return bd;

    if (obj instanceof Number n) {
        return BigDecimal.valueOf(n.doubleValue());
    }
    try {
        return new BigDecimal(String.valueOf(obj));
    } catch (Exception e) {
        return BigDecimal.ZERO;
    }
```

```
}
protected long toLong(Object obj) {
    if (obj instanceof Number n) return n.longValue();
    try {
        return Long.parseLong(String.valueOf(obj));
    } catch (Exception e) {
        return 0L;
    }
}
protected String safeStr(Object obj) {
    return obj == null ? "" : obj.toString();
}
protected long parseUptimeToSeconds(String uptime) {
    if (uptime == null || uptime.equalsIgnoreCase("Unknown")) return 0;
    uptime = uptime.trim();
    long days = 0, hours = 0, minutes = 0;
    String[] parts = uptime.split(" ");
    for (int i = 0; i < parts.length - 1; i++) {
        if (parts[i].matches("\\d+")) {
            switch (parts[i + 1].toLowerCase()) {
                case "d" -> days = Long.parseLong(parts[i]);
                case "h" -> hours = Long.parseLong(parts[i]);
                case "m" -> minutes = Long.parseLong(parts[i]);
            }
        }
    }
    return days * 86400 + hours * 3600 + minutes * 60;
}

public void saveNow(User user) {
    if (user == null) {
        System.out.println("Guest mode — не зберігаємо.");
        return;
    }
    try {
        Map<String, Object> data = collectFormattedStats();
        recordSystemStats(data, user);
    } catch (Exception e) {
```

```
            System.err.println("[MonitoringService] Помилка при ручному збереженні: " + e.getMessage());
        }
    }
    public String formatStatusSaved() {
        return "Статистика вручну оновлена (" + LocalTime.now().withNano(0) + ")";
    }
}
```

AdvancedMonitoringService.java - додає аналітичні функції: виведення попереджень про перевантаження CPU, RAM або нестачу місця на диску.

```
public class AdvancedMonitoringService extends MonitoringService {
    public AdvancedMonitoringService(MetricsProvider provider) {
        super(provider);
    }
    @Override
    protected void onMetricsCollected(Map<String, Object> data) {
        BigDecimal cpu   = toDecimal(data.get("cpuLoad"));
        BigDecimal ramU  = toDecimal(data.get("ramUsed"));
        BigDecimal ramT  = toDecimal(data.get("ramTotal"));
        BigDecimal diskT = toDecimal(data.get("diskTotal"));
        BigDecimal diskF = toDecimal(data.get("diskFree"));
        // CPU warning
        if (cpu.compareTo(BigDecimal.valueOf(90)) > 0) {
            System.out.printf("Високе навантаження CPU — %.2f%%%n", cpu);
        }
        // RAM warning
        if (ramT.compareTo(BigDecimal.ZERO) > 0) {
            BigDecimal perc = ramU.divide(ramT, 4, RoundingMode.HALF_UP)
                    .multiply(BigDecimal.valueOf(100));
            if (perc.compareTo(BigDecimal.valueOf(85)) > 0) {
                System.out.printf("Високе використання RAM — %.2f%% (%s MB)%n",
                        perc, ramU);
            }
        }
        // Disk warning
        if (diskT.compareTo(BigDecimal.ZERO) > 0) {
            BigDecimal freePerc = diskF.divide(diskT, 4, RoundingMode.HALF_UP)
                    .multiply(BigDecimal.valueOf(100));
            if (freePerc.compareTo(BigDecimal.valueOf(10)) < 0) {
```

```
            System.out.printf("Мало місця — %.2f%% залишилось%n", freePerc);
        }
    }
}
```

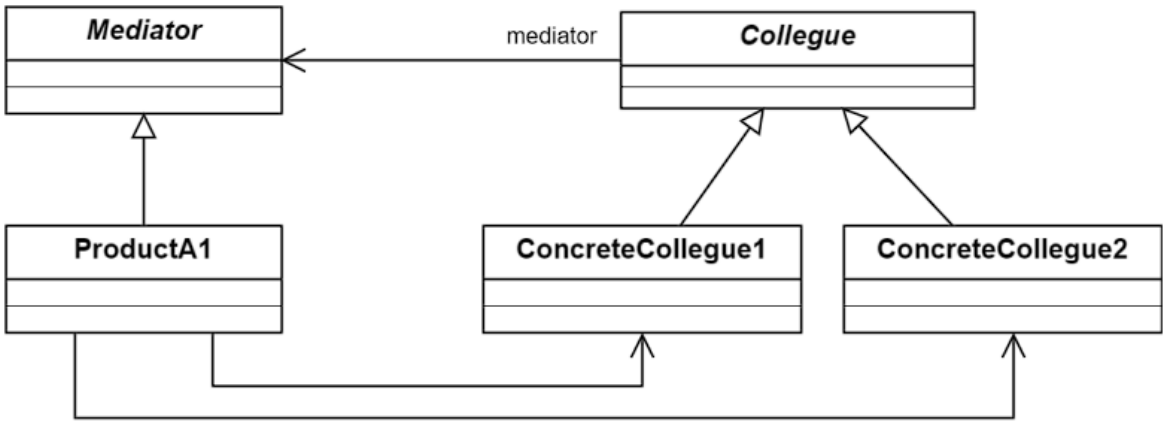
Висновок: У ході виконання лабораторної роботи було реалізовано механізм моніторингу системних ресурсів із застосуванням шаблону «Міст». Така архітектура дозволила чітко відокремити високорівневу логіку моніторингу від платформозалежних способів отримання системних метрик. Завдяки цьому рішення стало гнучким, зручним для підтримки та незалежним від операційної системи. Додавання нових реалізацій або зміна існуючих не впливає на роботу основної бізнес-логіки.

Відповідь на контрольні питання:

1. Яке призначення шаблону «Посередник»?

   Цей шаблон визначає об'єкт, який інкапсулює спосіб взаємодії множини об'єктів. Він сприяє слабкій зв'язаності, позбавляючи об'єкти необхідності явно посилатися один на одного, і дозволяє змінювати їхню взаємодію незалежно.

2. Нарисуйте структуру шаблону «Посередник».



3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?

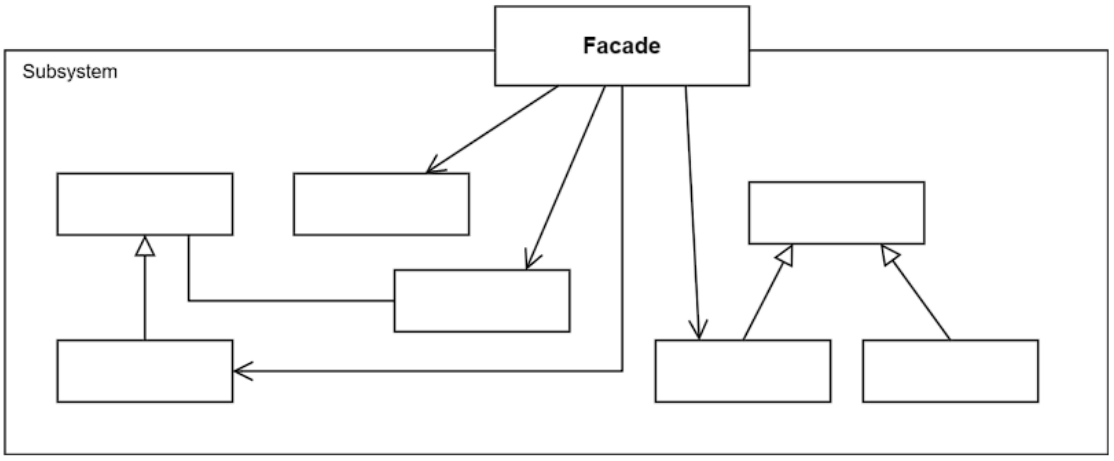
   Mediator: Інтерфейс, що визначає методи для комунікації з компонентами (колегами). ConcreteMediator: Реалізує інтерфейс Посередника та координує взаємодію між компонентами. Він знає про всі конкретні компоненти та керує ними. Colleague: Класи компонентів. Кожен колега знає свого посередника. Замість того щоб спілкуватися з іншими колегами напряму, вони викликають метод посередника.

4. Яке призначення шаблону «Фасад»?

   Надає уніфікований інтерфейс до набору інтерфейсів у складній підсистемі.

Фасад визначає інтерфейс вищого рівня, який полегшує використання підсистеми.

5. Нарисуйте структуру шаблону «Фасад».



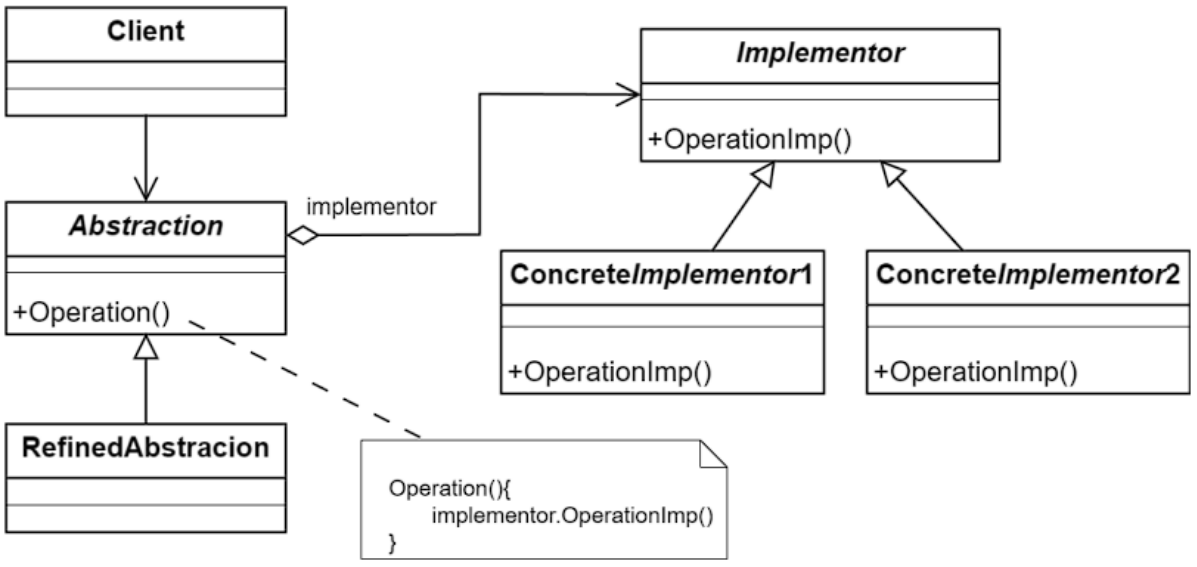
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?

Facade: Знає, яким класам підсистеми адресувати запит, і делегує запити клієнта відповідним об'єктам у підсистемі. Subsystem classes: Реалізують функціональність підсистеми. Виконують роботу, доручену об'єктом Facade. Вони не знають про існування Фасаду.

7. Яке призначення шаблону «Міст»?

Відокремлює абстракцію від її реалізації таким чином, щоб вони могли змінюватися незалежно одна від одної.

8. Нарисуйте структуру шаблону «Міст».



9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

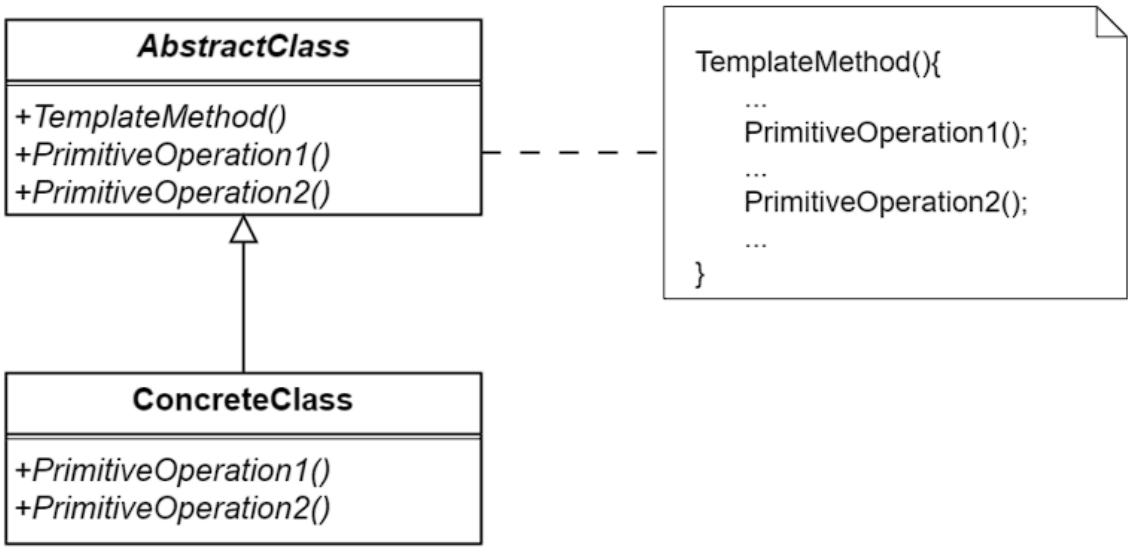
Abstraction: Визначає інтерфейс абстракції та зберігає посилання на об'єкт типу Implementor. RefinedAbstraction: Розширює інтерфейс, визначений в Abstraction. Implementor: Визначає інтерфейс для класів реалізації. Цей інтерфейс не обов'язково має

збігатися з інтерфейсом Abstraction. ConcreteImplementor: Реалізує інтерфейс Implementor.

10.Яке призначення шаблону «Шаблоний метод»?

Визначає скелет алгоритму в операції (методі), залишаючи реалізацію певних кроків підкласам. Це дозволяє підкласам перевизначати певні частини алгоритму без зміни його загальної структури.

11.Нарисуйте структуру шаблону «Шаблоний метод».



12.Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?

AbstractClass: Оголошує абстрактні методи (кроки алгоритму) і реалізує сам шаблонний метод, який викликає ці кроки у певному порядку. ConcreteClass: Реалізує абстрактні методи, виконуючи специфічні для даного підкласу кроки алгоритму.

13.Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

«Шаблонний метод» задає загальний алгоритм у базовому класі та дозволяє підкласам змінювати окремі кроки цього алгоритму. Тобто він керує тим, як виконується алгоритм. «Фабричний метод» визначає спосіб створення об’єктів і дозволяє підкласам вирішувати, який саме об’єкт буде створено. Тобто він керує тим, які об’єкти створюються.

14.Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» розділяє абстракцію та її реалізацію так, щоб їх можна було змінювати незалежно одна від одної. Він дозволяє будувати окремі ієрархії для

логічної частини (абстракції) і технічної частини (реалізації), що робить систему гнучкішою та зменшує кількість підкласів.