

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

Лабораторна робота № 7

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконала:
студентка групи ІА-32
Глущенко Анастасія
Сергіївна

Перевірив:
Мякий Михайло
Юрійович

Київ 2025

Зміст

Вступ	3
Теоретичні відомості	4
Хід роботи.....	7
1. Шаблон проектування «Bridge».....	7
2. Діаграма класів, яка представляє використання шаблону «Bridge»	10
3. Фрагменти коду реалізації шаблону «Bridge»	12
Висновки	18
Контрольні запитання.....	19

Вступ

Метою роботи є вивчення структури шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати «Bridge» в реалізації програмної системи.

Було обрано тему:

Installer generator (iterator, builder, factory method, bridge, interpreter, client-server)

Генератор інсталяційних пакетів повинен мати якийсь спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка – створити ярлик на робочому столі; ввести в текстове поле деякі дані, наприклад, ліцензійний ключ і т.д.). Генератор повинен вивести один файл .exe або .msi

Теоретичні відомості

1. «Mediator» (Посередник)

Визначає об'єкт, який інкапсулює спосіб взаємодії множини об'єктів. Посередник забезпечує слабку зв'язність, позбавляючи об'єкти необхідності явно посилатися один на одного, і дозволяє вам незалежно змінювати їхню взаємодію. Ключова ідея: Переміщення хаотичних зв'язків між об'єктами («багато до багатьох») в один центральний вузол, перетворюючи структуру на «один до багатьох». Це нагадує диспетчера в аеропорту: літаки спілкуються лише з ним, а не один з одним. Застосування:

- Коли складна логіка взаємодії між об'єктами призводить до заплутаних зв'язків, які важко зрозуміти та підтримувати.
- Коли неможливо повторно використовувати об'єкт, оскільки він обмінюється інформацією та залежить від багатьох інших об'єктів.
- Для налаштування поведінки, розподіленої між кількома класами, без створення безлічі підкласів.

Структура та реалізація

- Mediator (Посередник): Оголошує інтерфейс для обміну інформацією з об'єктами-колегами.
- ConcreteMediator (Конкретний Посередник): Реалізує кооперативну поведінку, координуючи дії об'єктів Colleague. Він знає про всі конкретні колеги та керує ними.
- Colleague (Колега): Класи компонентів системи. Кожен колега знає про свій об'єкт Mediator. Замість того, щоб спілкуватися з іншими колегами напряму, вони спілкуються з Посередником.

2. «Facade» (Фасад)

Надає уніфікований (спрощений) інтерфейс до набору інтерфейсів у підсистемі. Фасад визначає інтерфейс вищого рівня, який полегшує використання підсистеми. Основна мета: Приховати складність системи за

простим "фасадом". Клієнту не потрібно знати про десятки класів бібліотеки, він викликає лише один метод фасаду.Застосування:

- Коли потрібно надати простий інтерфейс до складної підсистеми (наприклад, бібліотеки відеокодеків, де клієнту потрібно лише натиснути "Конвертувати").
- Для структурування підсистеми в шари (layers). Фасад може стати точкою входу для кожного рівня.
- Щоб зменшити кількість залежностей між клієнтом та складною системою.

Структура та реалізація

- Facade (Фасад): Знає, яким класам підсистеми адресувати запит. Делегує запити клієнта відповідним об'єктам у підсистемі.
- Subsystem classes (Класи підсистеми): Реалізують функціональність підсистеми. Виконують роботу, доручену об'єктом Facade. Вони не знають про існування Фасаду і не посилаються на нього.
- Client (Клієнт): Взаємодіє з підсистемою, надсилаючи запити Фасаду.

3. «Bridge» (Міст)

Відокремлює абстракцію від її реалізації таким чином, щоб вони могли змінюватися незалежно одна від одної.Ключова ідея: Перехід від успадкування до композиції. Замість того, щоб створювати підкласи для кожної комбінації властивостей (наприклад, RedCircle, BlueCircle, RedSquare, BlueSquare), ми розділяємо їх на дві ієрархії: Форма та Колір, і поєднуємо їх через посилання ("міст").Застосування:

- Коли ви хочете уникнути постійної прив'язки абстракції до реалізації (наприклад, вибір реалізації може відбуватися під час виконання).
- Коли зміни в реалізації не повинні впливати на клієнтів (клієнтський код не потрібно перекомпілювати).
- Для уникнення "вибухового" зростання кількості класів через множинне варіювання аспектів об'єкта.

Структура та реалізація

- **Abstraction (Абстракція):** Визначає інтерфейс абстракції та зберігає посилання на об'єкт типу `Implementor`. Делегує всю реальну роботу цьому об'єкту.
- **RefinedAbstraction (Уточнена Абстракція):** Розширює інтерфейс, визначений `Abstraction` (наприклад, конкретні фігури).
- **Implementor (Реалізатор):** Визначає інтерфейс для класів реалізації. Цей інтерфейс не обов'язково має збігатися з інтерфейсом `Abstraction`; зазвичай він надає лише примітивні операції.
- **ConcreteImplementor (Конкретний Реалізатор):** Містить конкретну реалізацію інтерфейсу `Implementor` (наприклад, конкретні кольори або драйвери для ОС).

4. «Template Method» (Шаблонний метод)

Визначає скелет алгоритму в операції, залишаючи реалізацію деяких кроків підкласам. Патерн дозволяє підкласам перевизначати певні етапи алгоритму без зміни його структури. Основна мета: Усунення дублювання коду в схожих алгоритмах. Загальна частина залишається в батьківському класі, а відмінності виносяться в підкласи. Застосування:

- Коли підкласи повинні реалізовувати алгоритм, який має незмінну послідовність кроків, але деталі кожного кроку можуть відрізнятися.
- Коли потрібно виокремити спільну поведінку класів у батьківський клас, щоб уникнути дублювання коду.
- Для контролю точок розширення (hooks) - дозволяє підкласам розширювати алгоритм тільки в певних місцях.

Структура та реалізація

- **AbstractClass (Абстрактний Клас):** Визначає абстрактні примітивні операції, які заміщуються в конкретних підкласах, та реалізує сам шаблонний метод, який визначає скелет алгоритму (викликаючи ці примітивні операції).

- ConcreteClass (Конкретний Клас): Реалізує примітивні операції для виконання кроків алгоритму, специфічних для підкласу.

Хід роботи

1. Шаблон проектування «Bridge»

Призначення: Шаблон Bridge (Міст) використовується для відокремлення абстракції від її реалізації, що дозволяє змінювати їх незалежно одна від одної. Цей шаблон запобігає жорсткому зв'язуванню високорівневої логіки з низькорівневими деталями реалізації, розділяючи один великий клас або ієрархію на дві окремі ієрархії - «Абстракцію» (що робити) та «Реалізацію» (як робити), які з'єднані між собою посиланням (мостом).

Проблема, яку вирішує шаблон: У процесі розробки генератора інсталяторів виникла потреба підтримувати різні режими роботи інсталятора (графічний інтерфейс GUI або консольний режим) та використовувати різні рушії збірки (поточний Launch4j, а в майбутньому - архіватори, JSmooth або нативні засоби Java). Якщо використовувати класичне успадкування, це призведе до «вибуху» кількості класів (проблема декартового добутку): для кожної комбінації режиму та рушія довелося б створювати окремий клас (наприклад, Launch4jGui, Launch4jConsole, ZipGui, ZipConsole тощо). Це ускладнює підтримку коду і порушує принцип єдиної відповідальності, оскільки логіка налаштування параметрів (header type) змішується з логікою фізичного запуску процесу збірки (ProcessBuilder).

Як шаблон вирішує проблему: Шаблон Bridge вирішує цю проблему шляхом розбиття системи на дві незалежні ієрархії:

1. **Абстракцію** (логіка налаштування інсталятора: GuiInstaller, ConsoleInstaller), яка відповідає за підготовку даних.

2. **Реалізацію** (фізичний рушій збірки: BuildEngine), яка відповідає за технічну генерацію файлу. Абстракція містить посилання (міст) на інтерфейс реалізації та делегує йому виконання низькорівневих операцій. Це дозволяє додавати нові режими інсталяторів або нові рушії збірки, не змінюючи код іншої частини системи.

Процес створення об'єкта виглядає так:

1. Існує інтерфейс BuildEngine (Implementor), який визначає контракт для інструментів збірки: отримати конфігурацію та шлях виводу, і створити файл.

2. Існує абстрактний клас InstallerGeneratorBridge (Abstraction), який містить посилання на об'єкт BuildEngine. Він реалізує високорівневу логіку валідації та підготовки шляхів.

3. Конкретні класи абстракції (GuiInstaller, ConsoleInstaller) уточнюють параметри (наприклад, встановлюють тип заголовка headerType) і викликають метод збірки через міст, не знаючи, який саме інструмент виконує роботу.

Як реалізовано в проекті:

- Інтерфейс BuildEngine визначає низькорівневий контракт для генерації файлів.

- Клас Launch4jBuildEngine є конкретною реалізацією (Concrete Implementor), яка інкапсулює роботу з зовнішньою утилітою Launch4j, запуск процесів та обробку логів.

- Клас InstallerGeneratorBridge виступає базовою абстракцією, що керує білдером конфігурації Launch4jConfigBuilder.

- Класи GuiInstaller та ConsoleInstaller є уточненими абстракціями (Refined Abstraction), які налаштовують інсталятор під конкретний режим відображення.

- Клас `ExeInstallerFactory` (Клієнт) зв'язує ці дві ієрархії, використовуючи потрібну абстракцію (`GuiInstaller`) для виконання завдання.

Таким чином, шаблон Bridge забезпечує:

- **Відокремлення інтерфейсу від реалізації:** Логіка налаштування інсталятора (GUI/Console) не залежить від інструменту, який його збирає (Launch4j).

- **Запобігання розростанню ієрархії класів:** Кількість класів зростає лінійно, а не експоненціально при додаванні нових параметрів.

- **Гнучкість архітектури:** Можливість змінювати реалізацію (рушій збірки) під час виконання або конфігурації, не змінюючи код абстракції.

- **Дотримання принципу єдиної відповідальності:** Класи абстракції відповідають за бізнес-логіку (параметри), а класи реалізації - за системні операції (запис файлів, запуск процесів).

2. Діаграма класів, яка представляє використання шаблону «Bridge»

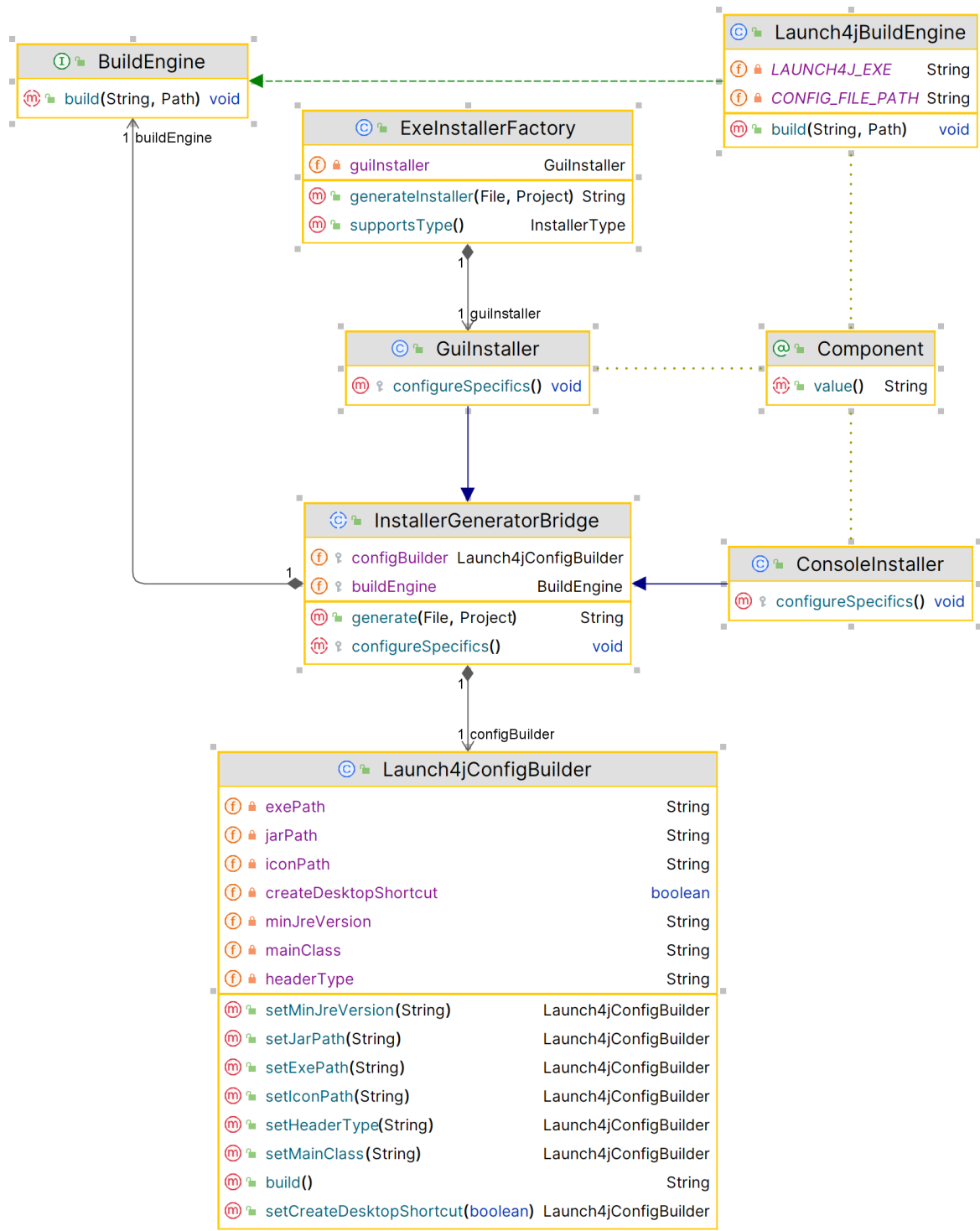


Рисунок 1 - Діаграма класів

Опис діаграми класів реалізації шаблону Bridge

На діаграмі зображено структуру класів, що беруть участь у реалізації шаблону проектування Bridge (Міст), який використовується для

розділення високорівневої логіки налаштування інсталятора (Абстракція) та низькорівневого механізму його фізичної збірки (Реалізація).

Абстрактний клас **InstallerGeneratorBridge** виступає в ролі Абстракції (Abstraction). Він визначає базову логіку підготовки процесу генерації, включаючи валідацію шляхів та керування білдером конфігурації. Цей клас містить поле (міст) типу BuildEngine, що дозволяє йому делегувати виконання фізичної збірки об'єкту реалізації, не прив'язуючись до конкретного інструменту.

Класи **GuiInstaller** та **ConsoleInstaller** є Уточненими Абстракціями (Refined Abstractions). Вони успадковуються від InstallerGeneratorBridge та розширюють його функціонал, встановлюючи специфічні параметри для конкретного режиму інсталятора (графічний інтерфейс або консольний режим). Вони використовують методи батьківського класу для передачі сформованої конфігурації через міст.

Інтерфейс **BuildEngine** виступає в ролі Реалізації (Implementor). Він визначає спільний контракт для всіх рушіїв збірки, декларуючи метод build, який приймає контент конфігурації та шлях виводу. Цей інтерфейс не залежить від типів інсталяторів (GUI чи Console).

Клас **Launch4jBuildEngine** є Конкретною Реалізацією (Concrete Implementor). Він імплементує інтерфейс BuildEngine та містить специфічний код для взаємодії із зовнішньою утилітою Launch4j, включаючи запис файлів, запуск процесів та обробку логів.

Зв'язок агрегації між класом InstallerGeneratorBridge та інтерфейсом BuildEngine є ключовим елементом патерну (власне, "мостом"). Це дозволяє змінювати реалізацію (рушій збірки) незалежно від абстракції (режиму інсталятора).

Клас **ExeInstallerFactory** виступає клієнтом, який ініціює процес генерації. Він використовує об'єкт уточненої абстракції (GuiInstaller) для виконання завдання, покладаючись на вже налаштований міст.

Клас **Launch4jConfigBuilder** є допоміжним компонентом, який використовується в ієрархії абстракції для формування вмісту XML-конфігурації перед передачею його в рушій збірки.

Ця діаграма відображає архітектуру, що дозволяє уникнути комбінаторного вибуху класів. Завдяки патерну Bridge, додавання нових режимів роботи (наприклад, "Service Mode") або нових рушіїв збірки (наприклад, "JSmooth") відбувається шляхом розширення відповідних ієрархій, не зачіпаючи іншу частину системи.

3. Фрагменти коду реалізації шаблону «Bridge»

InstallerGeneratorBridge

```
package com.example.installer.service.installer.bridge;

import com.example.installer.File;
import com.example.installer.Project;
import com.example.installer.service.installer.Launch4jConfigBuilder;
import java.nio.file.Path;
import java.nio.file.Paths;

public abstract class InstallerGeneratorBridge {

    protected final BuildEngine buildEngine;

    protected final Launch4jConfigBuilder configBuilder;

    protected InstallerGeneratorBridge(BuildEngine buildEngine,
Launch4jConfigBuilder configBuilder) {
        this.buildEngine = buildEngine;
        this.configBuilder = configBuilder;
    }

    public String generate(File jarFile, Project project) throws Exception
    {
        String installDir =
project.getInstallerSettings().getInstallPath();
        if (installDir == null || installDir.trim().isEmpty()) {
            throw new IllegalArgumentException("Install Path is not set!
Please configure it in project settings.");
        }

        String projectName = project.getProjectName();
        Path exePath = Paths.get(installDir, projectName + ".exe");
        String mainClass = project.getInstallerSettings().getMainClass();

        configBuilder.setJarPath(jarFile.getPath())
            .setExePath(exePath.toString())
            .setMainClass(mainClass)
            .setMinJreVersion("16")
    }
}
```

```

.setCreateDesktopShortcut(project.getInstallerSettings().isCreateDesktopShortcut());

    configureSpecifics();

    String configContent = configBuilder.build();

    buildEngine.build(configContent, exePath);

    return exePath.toAbsolutePath().toString();
}

protected abstract void configureSpecifics();
}

```

GuiInstaller

```

package com.example.installer.service.installer.bridge;

import com.example.installer.service.installer.Launch4jConfigBuilder;
import org.springframework.stereotype.Component;

@Component
public class GuiInstaller extends InstallerGeneratorBridge {

    public GuiInstaller(BuildEngine buildEngine, Launch4jConfigBuilder configBuilder) {
        super(buildEngine, configBuilder);
    }

    @Override
    protected void configureSpecifics() {
        configBuilder.setHeaderType("gui");
    }
}

```

ConsoleInstaller

```

package com.example.installer.service.installer.bridge;

import com.example.installer.service.installer.Launch4jConfigBuilder;
import org.springframework.stereotype.Component;

@Component
public class ConsoleInstaller extends InstallerGeneratorBridge {

    public ConsoleInstaller(BuildEngine buildEngine, Launch4jConfigBuilder configBuilder) {
        super(buildEngine, configBuilder);
    }

    @Override
    protected void configureSpecifics() {
        configBuilder.setHeaderType("console");
    }
}

```

BuildEngine

```
package com.example.installer.service.installer.bridge;

import java.nio.file.Path;

public interface BuildEngine {
    void build(String configContent, Path outputExePath) throws Exception;
}
```

Launch4jBuildEngine

```
package com.example.installer.service.installer.bridge;

import org.springframework.stereotype.Component;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

@Component
public class Launch4jBuildEngine implements BuildEngine {

    private static final String LAUNCH4J_EXE = "C:\\Program Files (x86)\\Launch4j\\launch4j.exe";

    private static final String CONFIG_FILE_PATH = "D:\\Робочий стіл\\trpz\\InstallerGenerator\\generated\\launch4j-config.xml";

    @Override
    public void build(String configContent, Path outputExePath) throws Exception {
        if (!Files.exists(Paths.get(LAUNCH4J_EXE))) {
            throw new RuntimeException("Launch4j not found at " + LAUNCH4J_EXE);
        }

        Path configPath = Paths.get(CONFIG_FILE_PATH);

        if (configPath.getParent() != null) {
            Files.createDirectories(configPath.getParent());
        }

        try (FileWriter writer = new FileWriter(configPath.toFile())) {
            writer.write(configContent);
        }

        System.out.println("Config written to: " + configPath.toAbsolutePath());

        ProcessBuilder pb = new ProcessBuilder(
            LAUNCH4J_EXE,
            configPath.toAbsolutePath().toString()
        );
        pb.redirectErrorStream(true);
        Process process = pb.start();

        try (BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()))) {

```

```

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println("[Launch4j]: " + line);
        }
    }

    int exitCode = process.waitFor();

    if (exitCode != 0) {
        throw new RuntimeException("Launch4j failed with code " +
exitCode);
    }

    System.out.println("Build process finished. Checking result at: " +
outputExePath);
}
}

```

ExeInstallerFactory

```

package com.example.installer.service.installer;

import com.example.installer.File;
import com.example.installer.Project;
import com.example.installer.service.installer.bridge.GuiInstaller;
import org.springframework.stereotype.Service;

@Service
public class ExeInstallerFactory implements InstallerFactory {

    private final GuiInstaller guiInstaller;

    public ExeInstallerFactory(GuiInstaller guiInstaller) {
        this.guiInstaller = guiInstaller;
    }

    @Override
    public InstallerType supportsType() {
        return InstallerType.EXE;
    }

    @Override
    public String generateInstaller(File jarFile, Project project) throws
Exception {

        return guiInstaller.generate(jarFile, project);
    }
}

```

Launch4jConfigBuilder

```

package com.example.installer.service.installer;
import org.springframework.stereotype.Service;

@Service
public class Launch4jConfigBuilder {

    private String jarPath;
    private String exePath;
}

```

```

private String mainClass;
private String minJreVersion;
private String iconPath;
private boolean createDesktopShortcut;

private String headerType = "gui";

public Launch4jConfigBuilder setJarPath(String jarPath) {
    this.jarPath = jarPath;
    return this;
}

public Launch4jConfigBuilder setExePath(String exePath) {
    this.exePath = exePath;
    return this;
}

public Launch4jConfigBuilder setMainClass(String mainClass) {
    this.mainClass = mainClass;
    return this;
}

public Launch4jConfigBuilder setMinJreVersion(String minJreVersion) {
    this.minJreVersion = minJreVersion;
    return this;
}

public Launch4jConfigBuilder setIconPath(String iconPath) {
    this.iconPath = iconPath;
    return this;
}

public Launch4jConfigBuilder setCreateDesktopShortcut(boolean val) {
    this.createDesktopShortcut = val;
    return this;
}

public Launch4jConfigBuilder setHeaderType(String headerType) {
    this.headerType = headerType;
    return this;
}

public String build() {
    StringBuilder xml = new StringBuilder();

    xml.append(""""
        <launch4jConfig>
            <dontWrapJar>false</dontWrapJar>
        """);

    xml.append("<headerType>").append(headerType).append("</headerType>\n");

    xml.append("    <jar>").append(jarPath).append("</jar>\n");
    xml.append("    <outfile>").append(exePath).append("</outfile>\n");

    if (iconPath != null && !iconPath.isEmpty()) {
        xml.append("    <icon>").append(iconPath).append("</icon>\n");
    }

    xml.append(""""
        <errTitle>Installer Generator</errTitle>
        <classPath>

```

```

        """);

xml.append("<mainClass>").append(mainClass).append("</mainClass>\n");
xml.append("    </classPath>\n");

xml.append("""
    <jre>
""");

xml.append("<minVersion>").append(minJreVersion).append("</minVersion>\n");

xml.append("""
    </jre>
    </launch4jConfig>
""");

return xml.toString();
    }
}

```

У проєкті реалізовано структурний шаблон Bridge, який дозволив розділити високорівневу логіку налаштувань від низькорівневої логіки генерації файлів:

- **Відокремлення абстракції від реалізації:** Базовий клас `InstallerGeneratorBridge` (абстракція) визначає процес підготовки даних, але не залежить від конкретних інструментів збірки (реалізації), взаємодіючи з ними виключно через інтерфейс `BuildEngine`.
- **Незалежне розширення:** Архітектура дозволяє додавати нові режими інсталяторів (наприклад, `ConsoleInstaller`) або нові рушії збірки (наприклад, для архівації в ZIP), не змінюючи код іншої частини системи та уникаючи комбінаторного зростання кількості класів.
- **Делегування виконання:** Абстракція не виконує фізичну роботу зі створення файлу самотійно, а делегує це завдання об'єкту реалізації (`Launch4jBuildEngine`) через посилення-міст, що забезпечує гнучкість і слабку зв'язність компонентів.

Висновки

На даному етапі реалізації проєкту «Installer Generator» було успішно впроваджено шаблон проектування «Bridge», який дозволив розділити абстракцію налаштування інсталятора та реалізацію його фізичної збірки на дві незалежні ієрархії, що значно підвищило гнучкість архітектури.

Було створено абстрактний клас `InstallerGeneratorBridge`, який визначає базову логіку підготовки даних, та інтерфейс `BuildEngine`, що відповідає за низькорівневі операції генерації. Розроблено уточнені абстракції `GuiInstaller` та `ConsoleInstaller` для підтримки різних режимів роботи інсталятора, а також конкретну реалізацію `Launch4jBuildEngine`, яка інкапсулює взаємодію з утилітою `Launch4j`.

Таким чином, на поточному етапі було:

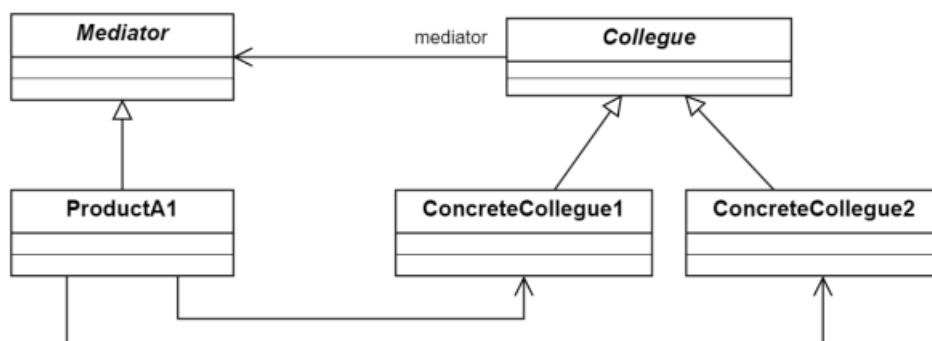
- реалізовано шаблон проектування Bridge для відокремлення високорівневої логіки конфігурації від технічних деталей реалізації;
- усунуто ризик комбінаторного зростання кількості класів (проблема декартового добутку) при додаванні нових параметрів;
- забезпечено можливість незалежного розширення ієрархій: нові режими інсталяторів та нові рушії збірки можна додавати, не змінюючи код один одного;
- підвищено чистоту коду шляхом чіткого розподілу відповідальності (Single Responsibility Principle) між класами налаштування та класами генерації.

Контрольні запитання

1. Яке призначення шаблону «Посередник»?

Призначення - визначити об'єкт, який інкапсулює спосіб взаємодії множини об'єктів, забезпечуючи слабку зв'язність і дозволяючи змінювати взаємодію між ними незалежно.

2. Нарисуйте структуру шаблону «Посередник».



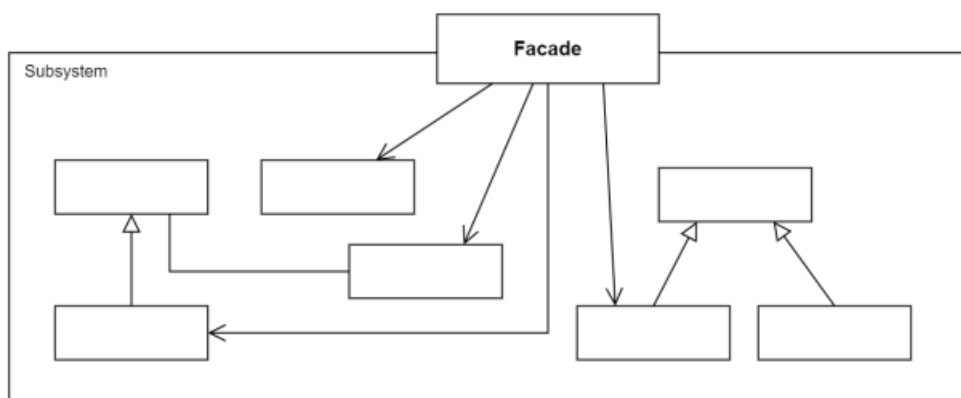
3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?

Входять: **Mediator** (інтерфейс), **ConcreteMediator** (координатор), **Colleague** (компоненти системи). Взаємодія: Колеги (**Colleagues**) не спілкуються напряду; вони надсилають повідомлення Посереднику (**Mediator**), який перенаправляє їх іншим Колегам або виконує відповідні дії.

4. Яке призначення шаблону «Фасад»?

Призначення - надати уніфікований і спрощений інтерфейс до набору інтерфейсів у складній підсистемі, приховуючи її складність від клієнта.

5. Нарисуйте структуру шаблону «Фасад».



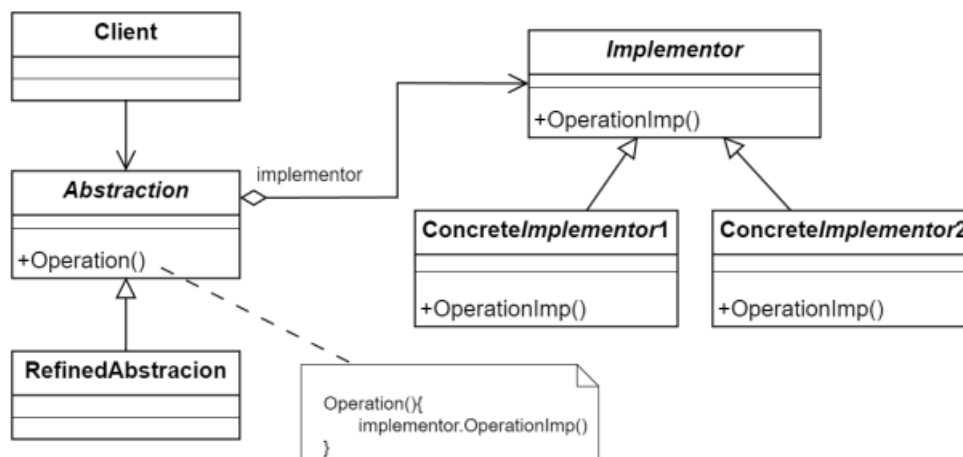
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?

Входять: Facade (спрощений інтерфейс), Subsystem Classes (складові підсистеми), Client. Взаємодія: Клієнт викликає методи Фасаду. Фасад делегує виконання роботи відповідним класам підсистеми. Класи підсистеми не знають про Фасад.

7. Яке призначення шаблону «Міст»?

Призначення - відокремити абстракцію від її реалізації так, щоб вони могли змінюватися незалежно одна від одної (перехід від успадкування до композиції).

8. Нарисуйте структуру шаблону «Міст».



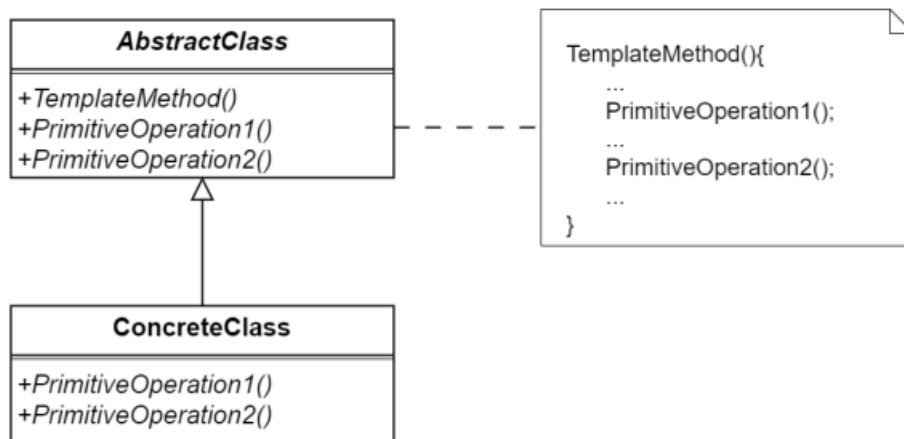
9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

Входять: Abstraction (інтерфейс високого рівня), RefinedAbstraction (уточнення), Implementor (інтерфейс реалізації), ConcreteImplementor (конкретна реалізація). Взаємодія: Abstraction містить посилання на об'єкт Implementor і делегує йому виконання низькорівневих операцій.

10. Яке призначення шаблону «Шаблонний метод»?

Призначення - визначити скелет алгоритму в суперкласі, дозволяючи підкласам перевизначати окремі кроки цього алгоритму без зміни його загальної структури.

11. Нарисуйте структуру шаблону «Шаблонний метод».



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

Входять: `AbstractClass` (містить шаблонний метод і абстрактні кроки), `ConcreteClass` (реалізує конкретні кроки). Взаємодія: Клієнт викликає шаблонний метод у `AbstractClass`, який послідовно викликає кроки алгоритму, реалізовані в `ConcreteClass`.

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

«Фабричний метод» - це породжувальний патерн, який спеціалізується на створенні об'єктів (делегує створення підкласам). «Шаблонний метод» - це поведінковий патерн, який керує потоком виконання алгоритму. Часто Фабричний метод виступає як один із кроків усередині Шаблонного методу.

14. Яку функціональність додає шаблон «Міст»?

Він додає можливість замінювати реалізацію об'єкта під час виконання програми, запобігає «вибуховому» зростанню кількості класів (через комбінації властивостей) та дозволяє розробляти абстракцію і реалізацію незалежно.