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

Лабораторна робота № 5

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконала:
студентка групи ІА-32
Глущенко Анастасія
Сергіївна

Перевірив:
Мягкий Михайло
Юрійович

Київ 2025

Зміст

Вступ

Метою роботи є вивчення структури шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати «Builder» 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. «Adapter» (Адаптер)

Перетворює інтерфейс одного класу на інтерфейс, який очікує інший. Адаптер дозволяє об'єктам із несумісними інтерфейсами працювати разом, не змінюючи їхнього оригінального коду.

Це корисно, коли:

- Ви використовуєте сторонню бібліотеку з класами, які мають потрібну функціональність, але їхні методи називаються інакше, ніж це прийнято у вашій системі.
- Ви створюєте систему, яка має підтримувати кілька несумісних класів (наприклад, роботу з різними форматами даних, як-от XML, JSON).

Структура та реалізація

Адаптер може бути реалізований двома основними способами:

- Об'єктний Адаптер (через композицію): Адаптер містить посилання на об'єкт, який потрібно адаптувати (Adaptee), і викликає його методи зсередини, перекладаючи виклики клієнта на очікуваний Target інтерфейс.
- Класовий Адаптер (через успадкування): Адаптер успадковує як цільовий інтерфейс (Target), так і клас, який потрібно адаптувати (Adaptee). Це вимагає підтримки множинного успадкування в мові програмування.

Ключові елементи:

- Target (Ціль): Інтерфейс, який очікує клієнт.
- Adaptee (Адаптуючий): Існуючий клас із несумісним інтерфейсом.
- Adapter (Адаптер): Реалізує інтерфейс Target і містить логіку виклику методів Adaptee.

2. «Builder» (Будівельник)

Відокремлює процес конструювання складного об'єкта від його представлення, дозволяючи використовувати один і той самий процес конструювання для створення різних представлень.

Основна мета: Вирішення проблеми "телескопічного конструктора" (коли клас має багато необов'язкових полів, і доводиться створювати безліч конструкторів із різною кількістю параметрів). Замість цього, клієнт покроково збирає об'єкт.

Застосування:

- Створення складних об'єктів (наприклад, звітів, документів, конфігурацій).
- Коли об'єкт має багато необов'язкових параметрів, і його конструювання вимагає певної послідовності кроків.

Структура та реалізація

- Product (Продукт): Складний об'єкт, що створюється.
- Builder (Інтерфейс Будівельника): Визначає кроки, необхідні для створення Product.
- ConcreteBuilder (Конкретний Будівельник): Реалізує інтерфейс Builder, надаючи специфічні реалізації для кожного кроку конструювання та метод для отримання готового Product (getResult()).
- Director (Директор, Необов'язковий): Керує послідовністю кроків конструювання. Клієнт може працювати з Director для спрощення збірки або викликати методи Builder безпосередньо.

3. «Command» (Команда)

Інкапсулює запит як об'єкт, дозволяючи параметризувати клієнтів різними запитами, поставити запити в чергу або зберігати їх у журналі, а також підтримувати операції скасування (Undo).

Ключова ідея: Перетворює виклик методу на окремий об'єкт, який містить всю інформацію про виклик: об'єкт-отримувач, метод, що викликається, та аргументи.

Застосування:

- Реалізація кнопок, меню та гарячих клавіш (Команда не знає, що вона робить, а лише викликає execute() на отримувачі).
- Підтримка історії операцій (журналювання та скасування).
- Системи керування чергою завдань.

Структура та реалізація

- Command (Інтерфейс Команди): Оголошує метод для виконання дії, як правило, execute().
- ConcreteCommand (Конкретна Команда): Зберігає зв'язок між Receiver (Отримувачем) та його операцією. Реалізує execute(), викликаючи відповідний метод на Receiver.
- Receiver (Отримувач): Клас, який виконує корисну роботу (наприклад, редактор тексту, лампочка, які отримують інструкцію від Команди).
- Invoker (Ініціатор/Викликач): Об'єкт, який ініціює запит, викликаючи метод execute() на об'єкті Command. Не знає, як виконується запит.

4. «Chain of Responsibility» (Ланцюг Обов'язків)

Дозволяє передавати запити послідовно через ланцюг обробників. Кожен обробник у ланцюгу вирішує, чи може він обробити запит, чи повинен передати його наступному обробнику в ланцюгу.

Основна ідея: Уникнення жорсткої прив'язки відправника запиту до його отримувача. Запит рухається по ланцюгу, поки не буде оброблений або дійде до кінця.

Застосування:

- Системи обробки подій або запитів (наприклад, HTTP-фільтри, middleware у веб-фреймворках).
- Системи затвердження (наприклад, запит на відпустку, який має пройти через керівника відділу, потім HR, потім директора).
- Обробка винятків (спроба обробити виняток на різних рівнях).

Структура та реалізація

- Handler (Інтерфейс Обробника): Визначає інтерфейс для обробки запиту та, зазвичай, метод для встановлення наступника (setNext()).
- ConcreteHandler (Конкретний Обробник): Реалізує логіку обробки. Якщо обробник не може (або не повинен) обробити запит, він делегує його наступному обробнику в ланцюгу.
- Client (Клієнт): Ініціює запит, передаючи його першому обробнику в ланцюгу.

5. «Prototype» (Прототип)

Дозволяє створювати нові об'єкти шляхом клонування (копіювання) наявного об'єкта, відомого як прототип. Це дозволяє уникнути необхідності створення підкласів фабрик для кожного типу продукту.

Ключова ідея: Створення об'єкта через копіювання існуючого об'єкта, а не через виклик конструктора new. Це особливо корисно, коли:

- Створення об'єкта є складним або ресурсомістким (наприклад, завантаження даних із бази).
- Необхідно динамічно визначати, який тип об'єкта буде створений під час виконання.

Структура та реалізація

- Prototype (Інтерфейс Прототипу): Оголошує операцію клонування, як правило, clone().
- ConcretePrototype (Конкретний Прототип): Реалізує операцію клонування. Це може бути поверхневе копіювання (shallow copy — копіюються лише посилання на об'єкти) або глибоке копіювання (deep copy — копіюються також і об'єкти, на які вказують посилання). Вибір залежить від вимог.
- Client (Клієнт): Створює новий об'єкт, викликаючи метод clone() на об'єкті-прототипі.

Хід роботи

1. Шаблон проектування «Builder»

Призначення:

Шаблон Builder використовується для створення складних об’єктів з різними варіантами конфігурацій. Він дозволяє поетапно побудувати об’єкт, абстрагуючи створення від кінцевого представлення. Цей шаблон особливо корисний, коли об'єкт може бути створений з багатьох частин, а кількість можливих варіантів параметрів занадто велика для одного конструктора.

Проблема, яку вирішує шаблон:

У процесі створення складних об’єктів може виникати потреба у наданні кількох варіантів конфігурацій і параметрів, що застосовуються до об’єкта. Якщо цей процес реалізовувати без використання шаблону Builder, це призведе до складності у підтримці коду, до дублювання логіки створення об’єкта і складності тестування через жорстке зв'язування конструкторів із різними параметрами.

У моєму проекті, де реалізовано генератор інсталяторів, я створюю об’єкт конфігурації для Launch4j і генерую .exe файл з .jar. Параметри цієї конфігурації, такі як шлях до .jar, шлях до .exe, іконка, головний клас, параметри JRE тощо, є складними і численними. Без шаблону Builder кожен раз довелося б вручну передавати параметри в конфігурацію, що ускладнює процес налаштування та збільшення кількості параметрів.

Як шаблон вирішує проблему:

Шаблон Builder дозволяє чітко розділити процес створення складного об’єкта на окремі етапи та використовувати гнучку конфігурацію. У моїй реалізації клас Launch4jConfigBuilder виступає в ролі будівельника. Він відповідає за поетапне налаштування конфігурації для Launch4j, надаючи методи для задання шляху до .jar файлу, шляху до .exe файлу, головного

класу, мінімальної версії JRE, створення ярлика на робочому столі та інші параметри.

Процес створення об’єкта виглядає так:

1. Launch4jConfigBuilder надає методи для налаштування окремих частин конфігурації.
2. Кожен метод повертає сам Builder, що дозволяє зручно викликати методи в ланцюговому виклику (Chaining).
3. Коли всі параметри налаштовані, викликається метод build(), який формує остаточний XML файл конфігурації для Launch4j.

Цей шаблон дозволяє створити об'єкт конфігурації поетапно і з мінімальними зусиллями, усуваючи необхідність передачі численних параметрів через конструктори чи методи.

Як реалізовано в проекті:

1. Клас **Launch4jConfigBuilder** надає методи для поетапного створення конфігурації Launch4j.
2. Клас **ExeGenerator** використовує Builder для створення конфігурації і передає її в Launch4j, щоб сформувати файл інсталятора.
3. Клас **InstallerBuilder** використовує об’єкт **Launch4jConfigBuilder** для створення конфігурації, а потім викликає метод **generateExe** для генерації .exe файлу.

Таким чином, шаблон Builder забезпечує:

- Інкапсуляцію процесу створення конфігурації,
- Зручність налаштування і розширення параметрів конфігурації,
- Гнучкість в налаштуванні складних об’єктів з багатьма параметрами,
- Зменшення кількості помилок при створенні об’єкта.

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

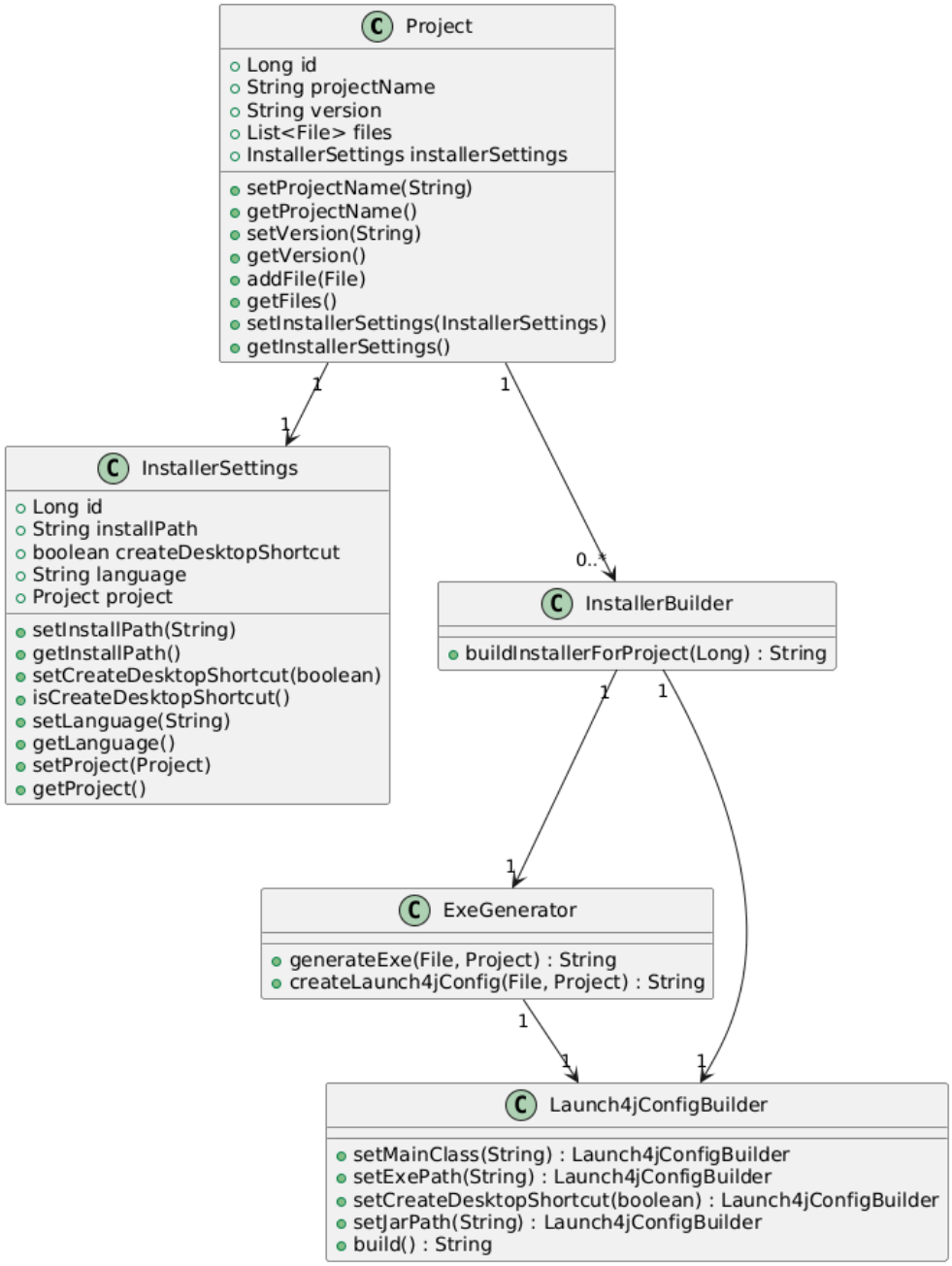


Рисунок 1 - Діаграма класів

Опис діаграми класів реалізації шаблону Builder

На діаграмі зображено структуру класів, що беруть участь у реалізації шаблону проектування Builder, який використовується для поетапного створення об'єкта конфігурації для генератора інсталяторів.

Клас Project є контейнером, який містить основну інформацію про проект, таку як його назва, версія та налаштування інсталятора

(InstallerSettings). Крім того, цей клас має список файлів, пов’язаних з проектом. Клас Project має залежність один-до-одного до InstallerSettings.

Клас InstallerSettings містить конфігурацію інсталятора для проекту, зокрема шлях до інсталяції, чи потрібно створювати ярлик на робочому столі, та мову інсталятора. Цей клас надає методи для налаштування та отримання цих параметрів. Зв'язок між класом InstallerSettings та Project є один-до-одного, оскільки кожен проект має тільки одне налаштування інсталятора.

Клас InstallerBuilder виконує роль будівельника, відповідаючи за поетапне створення інсталятора для конкретного проекту. Використовуючи клас Launch4jConfigBuilder, він налаштовує параметри інсталятора (шляхи до файлів, головний клас, параметри JRE тощо) та викликає метод generateExe в класі ExeGenerator для генерації інсталятора. Клас InstallerBuilder використовує всі компоненти для створення остаточного інсталятора.

Клас Launch4jConfigBuilder є частиною шаблону Builder і надає поетапні методи для налаштування параметрів конфігурації Launch4j. Він дозволяє налаштувати шлях до .jar файлу, головний клас, шлях до .exe файлу, мінімальну версію JRE та інші параметри, що використовуються для створення конфігурації для запуску через Launch4j.

Клас ExeGenerator є класом, який відповідає за генерацію .exe файлу за допомогою Launch4j. Використовуючи конфігурацію, створену класом Launch4jConfigBuilder, цей клас ініціює процес генерації .exe файлу за допомогою зовнішнього інструменту Launch4j, що дозволяє перетворювати .jar файли в .exe.

Ця діаграма відображає повну реалізацію шаблону Builder, де кожен клас відповідає за конкретну частину процесу створення інсталятора для проекту, а Builder забезпечує поетапне налаштування та формування фінального інсталяційного файлу.

3. Фрагменти коду реалізації шаблону «Builder»

Launch4jConfigBuilder

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

public class Launch4jConfigBuilder {

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

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

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

        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();
    }
}
```

InstallerBuilder

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

import com.example.installer.File;
import com.example.installer.Project;
import com.example.installer.repository.ProjectRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;

@Service
public class InstallerBuilder {

    private final JarValidator jarValidator;
    private final ExeGenerator exeGenerator;
    private final ProjectRepository projectRepository;

    public InstallerBuilder(JarValidator jarValidator,
                            ExeGenerator exeGenerator,
                            ProjectRepository projectRepository) {
        this.jarValidator = jarValidator;
        this.exeGenerator = exeGenerator;
        this.projectRepository = projectRepository;
    }

    @Transactional
    public String buildInstallerForProject(Long projectId) throws Exception
{
        Project project = projectRepository.findById(projectId)
                .orElseThrow(() -> new IllegalArgumentException("Project
not found"));
```

```
        ProjectFileIterator it = new
ProjectFileIterator(project.getFiles());
        List<File> jars = new ArrayList<>();

        while (it.hasNext()) {
            File f = it.next();
            if (jarValidator.isValidJar(f)) {
                jars.add(f);
            }
        }

        if (jars.isEmpty())
            throw new IllegalStateException("No JAR file found in
project.");
        if (jars.size() > 1)
            throw new IllegalStateException("Only one JAR file allowed.");

        File jarFile = jars.get(0);

        String installDir =
project.getInstallerSettings().getInstallPath();
        Path exePath = Paths.get(installDir,
jarFile.getFileName().replace(".jar", ".exe"));

        String xml = new Launch4jConfigBuilder()
                .setJarPath(jarFile.getPath())
                .setExePath(exePath.toString())

.setMainClass("com.example.testjarproject.HelloApplication")
                .setMinJreVersion("16")

.setCreateDesktopShortcut(project.getInstallerSettings().isCreateDesktopSho
rtcut())
                .build();

        return exeGenerator.generate(xml, exePath.toString());
    }
}
```

ExeGenerator

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

import org.springframework.stereotype.Service;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;

@Service
public class ExeGenerator {

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

    public String generate(String xmlConfig, String exeOutputPath) throws
Exception {

        Path tempConfig = Paths.get("generated/launch4j-config.xml");
        try (FileWriter writer = new FileWriter(tempConfig.toFile())) {
```

```
            writer.write(xmlConfig);
        }

        ProcessBuilder pb = new ProcessBuilder(
                LAUNCH4J_EXE,
                tempConfig.toAbsolutePath().toString()
        );

        pb.redirectErrorStream(true);
        Process process = pb.start();

        int exitCode = process.waitFor();
        if (exitCode != 0) {
            throw new RuntimeException("Launch4j error. Exit code = " +
exitCode);
        }

        return exeOutputPath;
    }
}
```

Ці фрагменти коду представляють основні моменти реалізації патерну Builder, зокрема використання класу Launch4jConfigBuilder для створення конфігурації, ExeGenerator для генерації файлу .exe, і InstallerBuilder для обробки процесу створення інсталятора для проєкту.

Висновки

На даному етапі реалізації проєкту «Installer Generator» було впроваджено шаблон проектування «Builder», який став основою для формування конфігурацій інсталятора та автоматизації створення .exe файлів для проєктів.

Було створено класи Launch4jConfigBuilder, ExeGenerator та InstallerBuilder, які взаємодіють між собою для формування інсталятора з файлів .jar. Клас Launch4jConfigBuilder відповідає за побудову конфігураційного XML файлу для Launch4j, в той час як ExeGenerator здійснює генерацію самого .exe файлу, а InstallerBuilder виступає основним компонентом для керування процесом, перевірки .jar файлів та їх конвертації.

Таким чином, на поточному етапі було:

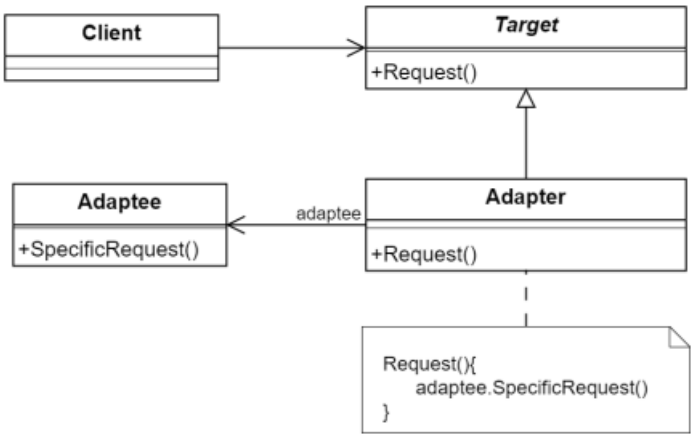
- реалізовано шаблон проектування Builder для створення інсталяційних конфігурацій;
- побудовано інтерфейс для автоматизованої генерації інсталятора;
- забезпечено механізм перевірки та конвертації .jar файлів у .exe;
- оптимізовано архітектуру для подальшого розвитку та масштабування.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Призначення «Адаптера» - дозволити об'єктам із несумісними інтерфейсами працювати разом, перетворюючи інтерфейс одного класу на інтерфейс, який очікує інший.

2. Нарисуйте структуру шаблону «Адаптер».



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

До шаблону входять: Target (цільовий інтерфейс, який очікує клієнт), Adaptee (існуючий несумісний клас), та Adapter (який реалізує Target і делегує запити об'єкту Adaptee). Клієнт взаємодіє лише з Adapter через інтерфейс Target.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Об'єктний Адаптер використовує композицію (тримає посилання на Adaptee). Це гнучкіше і дозволяє адаптувати будь-який підклас Adaptee.

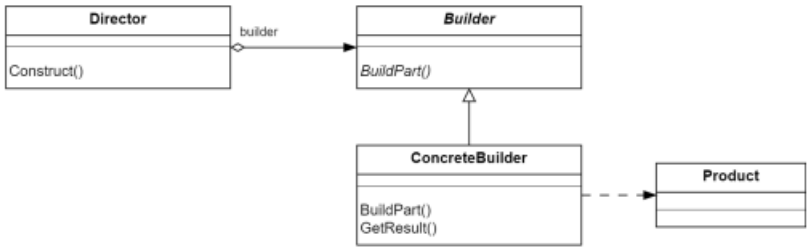
Класовий Адаптер використовує множинне успадкування (успадковує як Target, так і Adaptee). Цей підхід є менш гнучким і доступний не у всіх мовах.

5. Яке призначення шаблону «Будівельник»?

Призначення «Будівельника» - відокремити процес покрокового конструювання складного об'єкта від його фінального представлення,

дозволяючи використовувати один і той самий процес для створення різних варіантів продукту.

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

Входять: Product (складний об'єкт), Builder (інтерфейс з кроками), ConcreteBuilder (реалізує кроки для конкретного продукту), та Director (керує послідовністю кроків). Director використовує Builder для конструювання Product, а клієнт отримує готовий Product від Builder.

8. У яких випадках варто застосовувати шаблон «Будівельник»?

Варто застосовувати, коли:

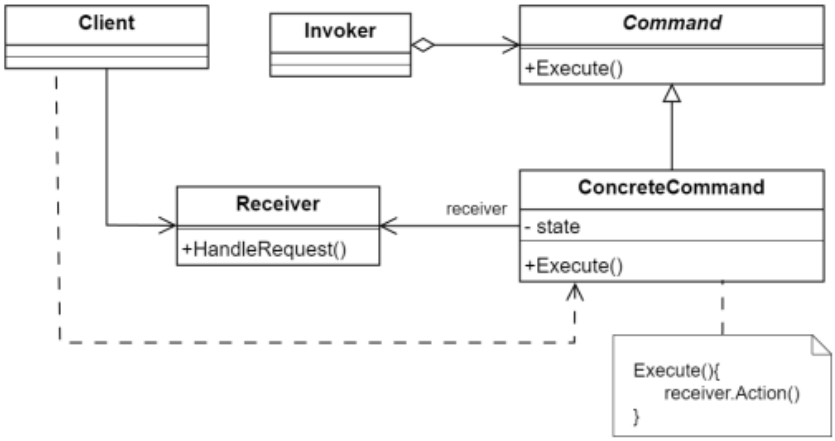
Необхідно створювати складні об'єкти, які можуть мати багато необов'язкових параметрів і різні конфігурації (щоб уникнути "телескопічного конструктора").

Процес конструювання об'єкта вимагає виконання певних кроків у фіксованій послідовності.

9. Яке призначення шаблону «Команда»?

Призначення «Команди» - інкапсулювати запит як об'єкт, що дозволяє параметризувати клієнтів, створювати черги запитів, підтримувати їх журналювання та реалізовувати операції скасування (Undo).

10. Нарисуйте структуру шаблону «Команда».



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

Входять: Command (інтерфейс з методом execute()), ConcreteCommand (зберігає зв'язок з Receiver), Receiver (об'єкт, який виконує роботу), та Invoker (ініціатор, викликає execute()). Invoker взаємодіє тільки з Command, не знаючи деталей виконання.

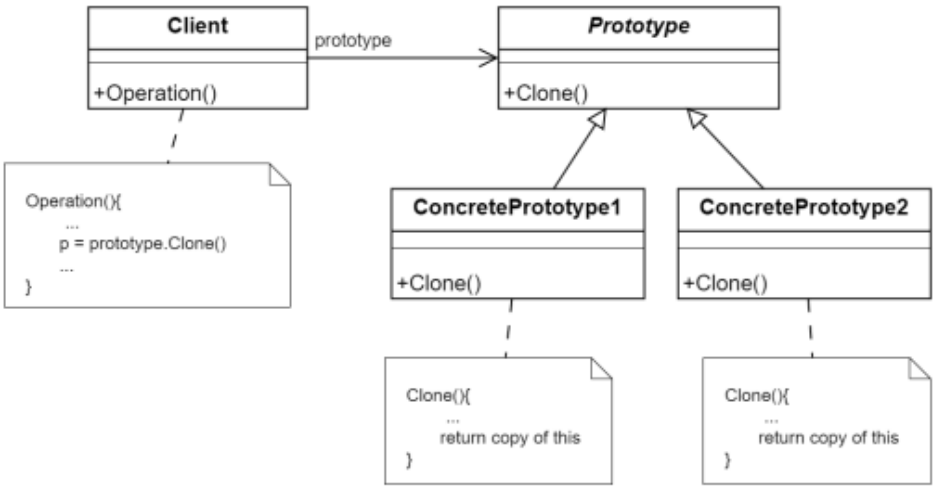
12. Розкажіть як працює шаблон «Команда».

Клієнт створює об'єкт ConcreteCommand, передаючи йому об'єкт-Receiver (отримувач, який знає, як виконати дію) і необхідні аргументи. Потім клієнт передає цю Command об'єкту Invoker. Коли Invoker готовий, він викликає метод execute() на об'єкті Command, який, у свою чергу, викликає відповідний метод на Receiver.

13. Яке призначення шаблону «Прототип»?

Призначення «Прототипу» — створювати нові об'єкти шляхом клонування (копіювання) наявного об'єкта-прототипу, замість використання конструктора. Це корисно, коли створення об'єкта є ресурсомістким або коли необхідно динамічно обирати тип створюваного об'єкта.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

Входять: Prototype (інтерфейс, який оголошує метод clone()), ConcretePrototype (реалізує логіку клонування), та Client (створює новий об'єкт, викликаючи метод clone() на вже існуючому об'єкті ConcretePrototype).

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Системи затвердження/узгодження: Запит (наприклад, про відпустку) проходить послідовно через різних посадових осіб (менеджер -> HR -> директор), поки не буде схвалений або відхилений.

Обробка подій/запитів (Middleware): HTTP-запит проходить через низку обробників (аутентифікація, логування, кешування) перед тим, як дійти до основного контролера.

Фільтри у графічних редакторах: Зображення послідовно обробляється різними фільтрами (розмиття, зміна кольору, додавання тексту), кожен з яких виконує свою частину роботи.