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 №5
ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»
Варіант: 25. Installer generator

Виконав:

Студент групи ІА-22

Птачик Р.С.

Перевірив:

Мягкий М.Ю.

Київ 2024

Зміст:

Тема: шаблони «adapter», «builder», «command», «chain of responsibility», «prototype».

Мета: ознайомитися з шаблонами проектування «adapter», «builder», «command», «chain of responsibility», «prototype». Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами

Хід роботи:

Тема:

25. 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.

Короткі теоретичні відомості

Анти-патерни — це типові помилки у проектуванні, управлінні чи реалізації програмного забезпечення. Їхнє розпізнавання дозволяє уникати поширених проблем і створювати більш стійкі та зрозумілі системи. Вони є протилежністю шаблонів проектування, які демонструють ефективні рішення.

Анти-патерни в управлінні розробкою ПЗ:

- **Дим і дзеркала**: Демонстрація непрацюючих функцій для отримання схвалення проекту.
- **Роздування ПЗ**: Додавання непотрібних функцій, що збільшують вимоги до ресурсів.
- **Функції для галочки**: Включення функцій, які використовуються лише для маркетингових заяв.

Анти-патерни в розробці ПЗ:

- **Великий клубок бруду**: Система без чіткої структури, важка в обслуговуванні.

- **Інверсія абстракції**: Приховування частини функціоналу від зовнішнього використання.
- **Бензинова фабрика**: Створення занадто складного дизайну без реальної потреби.
- **Затичка на введення даних**: Відсутність перевірок на некоректний ввід.
- **Роздування інтерфейсу**: Виготовлення інтерфейсу дуже потужним і дуже важким для реалізації.

Анти-патерни в ООП:

- **Базовий клас-утиліта**: Спадкування замість делегування.
- **Божественний об'єкт**: Клас, який виконує забагато функцій, що порушує принцип єдиного обов'язку.
- **Самотність**: Надмірне використання патерну Singleton.
- **Об'єктна клоака**: перевикористання об'єктів, що знаходяться в непридатному для перевикористання стан.

Анти-патерни в програмуванні:

- **Непотрібна складність**: Ускладнення рішень без потреби.
- **Дія на відстані**: Несподівані взаємодії між частинами системи.
- **Спагеті-код**: Код з хаотичною структурою, важкий для розуміння та підтримки.
- **Таємничий код**: Використання абревіатур замість мнемонічних імен.

Методологічні анти-патерни:

- **Копіювання-вставка**: Дублювання коду замість використання спільних рішень.
- **Передчасна оптимізація**: Оптимізація без достатньої інформації.
- **Винахід колеса**: Створення рішень з нуля, ігноруючи існуючі.
- **Передчасна оптимізація**: Оптимізація на основі недостатньої інформації.

Шаблони проектування

Шаблон "Adapter" (Адаптер)

Призначення: Узгоджує інтерфейси несумісних об'єктів, дозволяючи їм працювати разом.

Проблема: Потрібно інтегрувати бібліотеки з різними форматами (наприклад, XML і JSON).

Рішення: Адаптер перетворює інтерфейс одного об'єкта в інтерфейс, зрозумілий іншому.

Приклад: Зарядка ноутбука через адаптер до розетки іншого стандарту.

Шаблон "Builder" (Будівельник)

Призначення: Розділяє процес створення об'єкта від його представлення. Підходить для складних об'єктів із багатьма етапами створення.

Проблема: Складність створення об'єкта через багатоетапний процес (наприклад, формування HTTP-відповіді).

Рішення: Реалізувати окремі методи для кожного етапу створення об'єкта.

Приклад: Будівництво будинку, де кожен етап (фундамент, стіни, дах) виконується поетапно.

Шаблон "Command" (Команда)

Призначення: Інкапсулює виклик методу у вигляді об'єкта, дозволяючи гнучко управляти діями.

Проблема: Як організувати обробку кліків у текстовому редакторі без дублювання коду.

Рішення: Створення класів-команд, які викликають методи бізнес-логіки через об'єкт-інтерфейс.

Шаблон "Chain of Responsibility" (Ланцюг відповідальності)

Призначення: Передає запит через ланцюг об'єктів, поки його не обробить відповідний об'єкт.

Проблема: Як організувати перевірки прав доступу в системі з мінімальною залежністю.

Рішення: Реалізувати клас для кожної перевірки та об'єднати їх у ланцюг.

Шаблон "Prototype" (Прототип)

Призначення: Створює об'єкти шляхом клонування існуючих шаблонів.

Проблема: Як скопіювати об'єкт зі складною внутрішньою структурою, не порушуючи інкапсуляцію.

Рішення: Додати метод clone для об'єктів, які потрібно копіювати.

Реалізувати не менше 3-х класів відповідно до обраної теми

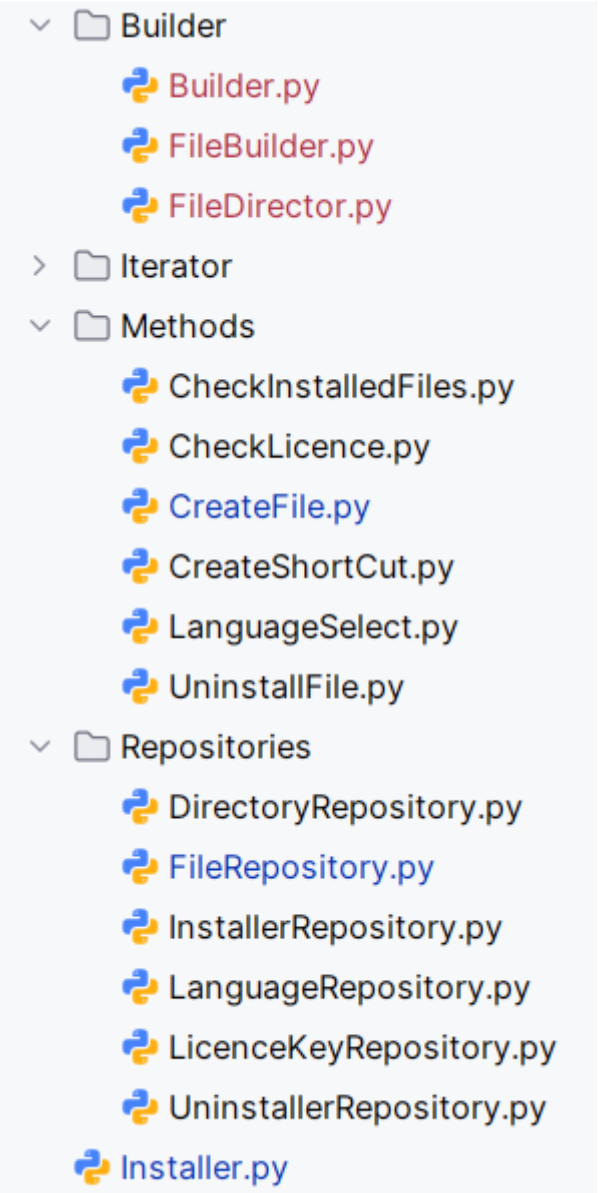

Рис. 1 – Структура проекту

В цій лабораторній роботі було реалізовано частину генератора інсталяційних пакетів з використанням шаблону Iterator, а саме збереження файлу. В результаті було створено та оновлено наступні класи:

1. Builder: Визначає методи для налаштування параметрів об'єкта та метод buildPart() для його створення. Призначення цього класу – стандартизувати

процес побудови файлів із різними параметрами (ім'я файлу, шлях, тип файлу), забезпечуючи модульність і гнучкість.

2. FileBuilder: Реалізує клас Builder та відповідає за створення об'єкта файлу. Логіка класу побудована таким чином, що для кожного параметра (ім'я, шлях, тип файлу) використовується загальний метод buildPart() з класу Builder, що дозволяє централізувати процес збереження даних.

3. FileDirector: Відповідає за координацію процесу створення об'єкта файлу, використовуючи FileBuilder. Його метод construct_file() забезпечує чіткий порядок виклику методів set_name(), set_path(), set_type() у FileBuilder, гарантуючи послідовність і правильність створення об'єкта.

4. createFile(): Метод, який інтегрує всі компоненти шаблону "Builder". Використовує FileDirector для створення файлів і FileRepository для їхнього збереження. Забезпечує зручний спосіб роботи з файлами, абстрагуючи клієнтський код від деталей реалізації процесу створення та зберігання файлів.

5. Installer: забезпечує взаємодію між репозиторієм файлів та процесом зберігання файлу.

Реалізувати один з розглянутих шаблонів за обраною темою

Згідно з завданням, було використано та реалізовано шаблон Builder.

Реалізація шаблону Builder:

1. Builder:

```
class Builder:  2 usages
    def __init__(self):
        self.data = {}

    def buildPart(self, key, value):
        self.data[key] = value

    def get(self):
        return self.data
```

Рис. 2 – Клас Builder

- Визначає метод buildPart(), який має реалізувати кожен конкретний будівельник (Builder).
- Гарантує, що будь-який клас, який реалізує цей інтерфейс, матиме однакову структуру створення об'єкта файлу.

2. FileBuilder:

```
class FileBuilder(Builder):  4 usages
    def __init__(self):
        super().__init__()

    def set_name(self, file_name):  1 usage
        self.buildPart( key: "file_name", file_name)
        return self

    def set_path(self, path):  1 usage (1 dynamic)
        self.buildPart( key: "file_path", path)
        return self

    def set_type(self, file_type):  1 usage (1 dynamic)
        self.buildPart( key: "file_type", file_type)
        return self

    def build(self):
        if not self.data:
            raise ValueError("File details are incomplete.")
        return self.get()
```

Рис. 3 – Клас FileBuilder

- Використовується для створення специфічного об'єкта файлу, який зберігає атрибути, такі як ім'я файлу, шлях до файлу та тип файлу.
- Реалізує метод build(), який повертає об'єкт файлу, створений із заданими параметрами.
- Логіка класу забезпечує централізоване управління атрибутами файлу за допомогою методу buildPart () інтерфейсу Builder.

3. FileDirector:

```
class FileDirector:  2 usages
    def __init__(self, builder: FileBuilder):
        self.builder = builder

    def construct_file(self, file_name, path, file_type)
        return (
            self.builder
            .set_name(file_name)
            .set_path(path)
            .set_type(file_type)
            .build()
        )
```

Рис. 4 – Клас FileDirector

- Відповідає за контроль процесу створення об'єкта файлу шляхом виклику методів set_name(), set_path(), set_type() у переданого FileBuilder.
- Забезпечує клієнтам зручний спосіб створення об'єкта без необхідності прямої взаємодії з конкретними будівельниками.
- Його метод construct_file() поетапно викликає методи будівельника, гарантуючи правильне налаштування параметрів файлу.

4. createFile():

```
def createFile(file_repository, file_name, path, file_type):  2
    builder = FileBuilder()
    director = FileDirector(builder)

    file = director.construct_file(file_name, path, file_type)

    file_repository.add_file(file)
```

Рис. 5 – Клас createFile

- Використовує FileDirector для управління процесом створення файлу та FileRepository для його збереження.

- Забезпечує спрощений і зручний інтерфейс для створення та збереження файлів, абстрагуючи клієнтський код від деталей реалізації.

5. Installer:

```
def create_file(self, file_name, path, file_type):  2 usages  new *
    createFile(self._installer_repository.get_file_repo(), file_name, path, file_type)
```

Рис. 6 – Клас Installer

- Клас для забезпечення взаємодії між репозиторієм файлів та функцією збереження файлів.
- У конструкторі отримує сховище файлів (**installer_repository**) та викликає метод **createFile** для збереження файлів, використовуючи FileBuilder та FileDirector.

Діаграма класів для шаблону Builder

Діаграма класів для шаблону Builder:

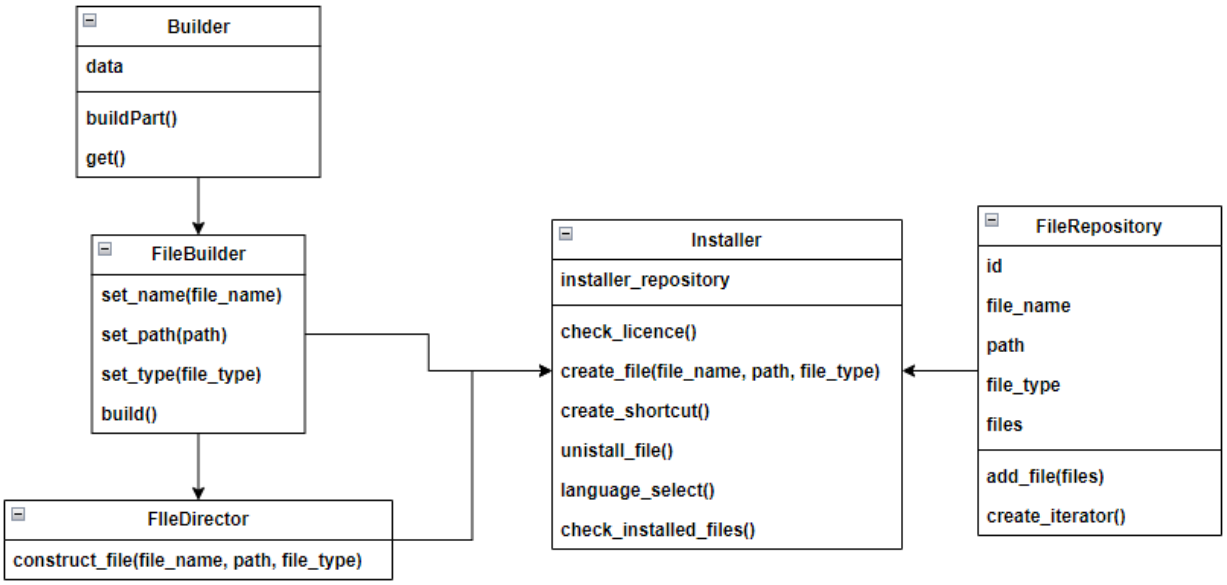

Рис. 7 – Діаграма класів для шаблона Builder

Проблема, яку допоміг вирішити шаблон Builder

Використання шаблону Builder у проекті "Installer Generator" вирішило проблему створення складних об'єктів файлів, які мають кілька параметрів, таких як ім'я, шлях і тип файлу.

Завдяки шаблону Builder вдалося:

- Спрощувати процес створення файлів, інкапсулюючи логіку побудови в окремому класі.
- Забезпечити чітку структуру та послідовність, мінімізуючи ймовірність помилок.
- Надавати гнучкість у створенні файлів із різними конфігураціями без дублювання коду.

Це рішення зробило процес побудови файлів модульним, читабельним і зручним для розширення, що спрощує підтримку проекту в довгостроковій перспективі.

Переваги використання шаблону Builder

Ізоляція процесу збереження об'єкта:

- Клієнтський код не потребує знань про порядок або специфіку ініціалізації параметрів об'єкта файлу.
- Усі деталі створення об'єкта інкапсулюються в класі Builder, що забезпечує простий і зрозумілий інтерфейс.

Гнучкість:

- Можна легко створювати файли з різними конфігураціями, налаштовуючи лише необхідні параметри.
- Можливість реалізовувати різні варіанти будівельників для інших типів об'єктів, наприклад, для директорій чи інших елементів інсталяційного пакета.

Принцип розділення обов’язків (SRP):

- Клас Builder відповідає лише за створення об'єкта, клас Director — за управління процесом побудови, а FileRepository — за зберігання. Це підвищує загальну структурованість системи.

Масштабованість:

- Додавання нових параметрів або змін у структурі файлу потребує лише оновлення будівельника, не зачіпаючи клієнтський код.
- Шаблон легко адаптується до нових вимог, таких як підтримка інших типів файлів або додаткових атрибутів.

Код та висновок

Код можна знайти за посиланням:
https://github.com/FryMondo/TRPZ/tree/master/InstallGenerator

Висновок: У цій лабораторній роботі реалізовано шаблон Builder для генератора інсталяційних пакетів. Завдяки йому створено гнучкий механізм побудови файлів із заданими параметрами, що спростило логіку створення об'єктів і підвищило масштабованість системи. Реалізація включала класи Builder, FileBuilder, FileDirector, FileRepository, а також метод createFile(), що забезпечили ефективну взаємодію між компонентами системи. Застосування шаблону Builder дозволило: інкапсулювати логіку створення об'єктів у спеціалізованих класах, спрощуючи підтримку та розширення функціональності; забезпечити чітку структуру та послідовність у процесі створення об'єктів; підвищити гнучкість системи, дозволяючи легко додавати нові параметри чи типи файлів.