**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 №4
# ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»
## Варіант: 25. Installer generator

Виконав:

Студент групи ІА-22

Птачик Р.С.

Перевірив:

Мягкий М.Ю.

**Київ 2024**

**Зміст:**

**Тема:** шаблони «singleton», «iterator», «proxy», «state», «strategy».

**Мета:** Ознайомитися з основними паттернами проектування «singleton», «iterator», «proxy», «state», «strategy». Реалізувати частину функціоналу робочої програми за допомогою одного з розглянутих шаблонів.

## 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

Шаблони проектування — це перевірені часом та широко використовувані формалізовані рішення для типових задач, що виникають у процесі розробки програмного забезпечення. Вони містять опис проблеми, оптимальне рішення та рекомендації щодо його застосування в різних контекстах. Кожен шаблон має загальноприйняту назву, що полегшує його ідентифікацію та повторне використання.

Правильно сформульовані шаблони дозволяють розробникам уникати повторення тих самих рішень, зменшувати витрати часу на проектування та стандартизувати підходи до вирішення завдань. Важливим етапом роботи з шаблонами є коректне моделювання предметної області, яке допомагає чітко сформулювати задачу та обрати найбільш відповідний шаблон.

Застосування шаблонів проектування надає низку переваг:

- **Структурованість**: моделі, створені за допомогою шаблонів, є зрозумілішими для аналізу, розробки та розширення.
- **Гнучкість**: шаблони забезпечують можливість адаптації системи до змін.
- **Спрощення інтеграції**: використання шаблонів полегшує інтеграцію з іншими системами.

- **Стандартизація**: шаблони є своєрідним словником проектування, що забезпечує спільне розуміння архітектури всередині команди.

### Чому важливо використовувати шаблони проектування?

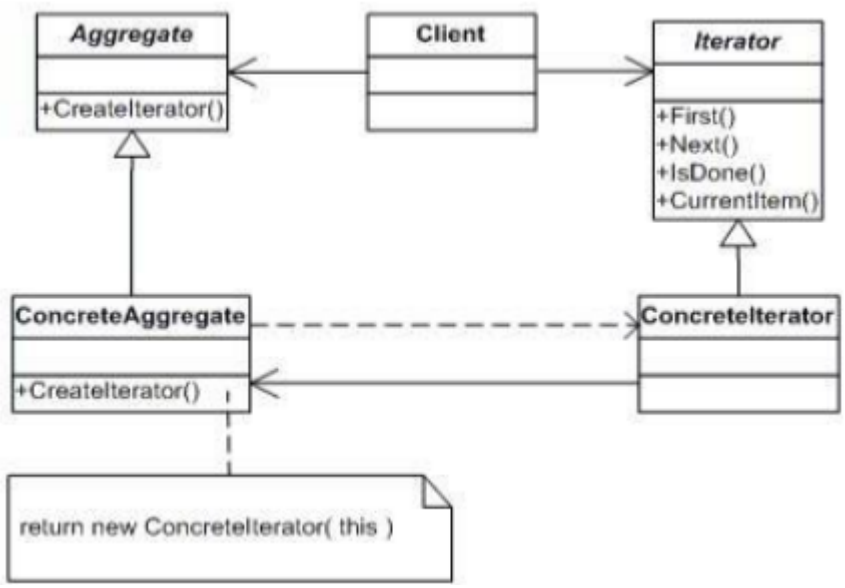
Шаблони проектування не гарантують абсолютної простоти реалізації, але у відповідних ситуаціях вони дозволяють досягти таких результатів:

- Зменшення витрат часу та трудовитрат на побудову архітектури.
- Забезпечення гнучкості та адаптованості системи.
- Зниження витрат на підтримку та розвиток проекту.
- Стандартизація підходів у межах команди розробників.

Однак слід пам’ятати, що шаблони проектування не є догмою і не завжди виправдані. Їх використання повинно бути усвідомленим та відповідати специфіці конкретного проекту.

## Шаблон «Iterator»

**Структура:**




Шаблон **Iterator** використовується для організації послідовного доступу до елементів складної колекції без необхідності розкривати її внутрішню структуру. Цей шаблон дозволяє здійснювати обхід колекції різними способами (наприклад, прямий, зворотний, обхід тільки певних елементів) і зберігати при цьому єдиний інтерфейс для взаємодії.

**Основні компоненти шаблону Iterator:**

- **First()** – встановлює покажчик ітерації на перший елемент колекції.
- **Next()** – переміщує покажчик ітерації на наступний елемент колекції.
- **IsDone** – булеве поле, що повертає true, коли досягнуто кінця колекції.
- **CurrentItem** – повертає поточний елемент, на який вказує ітератор.

**Призначення:**

Шаблон **Iterator** дозволяє спростити доступ до елементів складних структур даних, таких як дерева чи графи, без необхідності розуміти їхню реалізацію. Він розділяє відповідальність між колекцією (збереження даних) та ітератором (обхід елементів).

**Проблема:**

Деякі колекції мають складну структуру, для обходу якої потрібні спеціалізовані алгоритми. Додавання цих алгоритмів безпосередньо до класу колекції ускладнює код і порушує принципи проектування.

**Рішення:**

Винести логіку обходу в окремий клас ітератора, який забезпечує необхідний доступ до елементів. Це дозволяє легко створювати нові способи обходу без зміни коду самої колекції.

**Переваги:**

- Дозволяє використовувати різні алгоритми обходу для однієї колекції.
- Зберігає інкапсуляцію, приховуючи внутрішню структуру колекції.
- Зменшує залежність між клієнтом і колекцією.

**Недоліки:**

- Може бути надмірним, якщо колекція проста і достатньо базового циклу для її обходу.
- Додає додатковий рівень складності для невеликих програм.

**Реалізувати не менше 3-х класів відповідно до обраної теми**

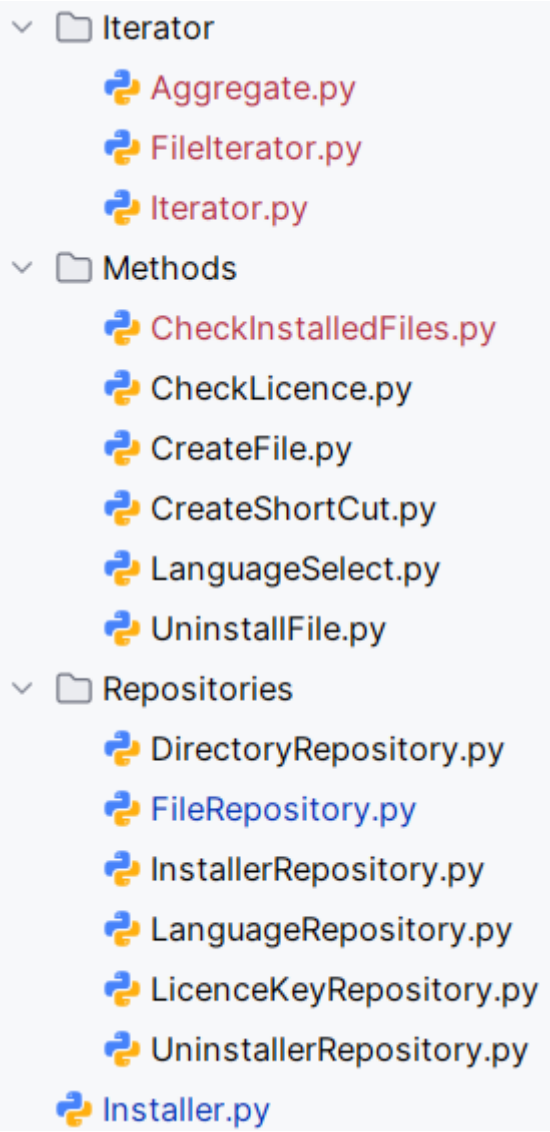



Рис. 1 – Структура проекту

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

**1. FileIterator:** реалізує логіку ітерації по файлах у репозиторії.

**2. Iterator:** загальний клас-обгортка для ітератора. Абстрагує конкретну реалізацію ітератора (в даному випадку, FileIterator).

**3. Aggregate:** відповідає за створення екземпляра ітератора. Є проміжним шаром, який відокремлює логіку репозиторію від конкретної реалізації ітератора.

**4. FileRepository:** репозиторій для файлів, який: зберігає дані про файли; містить метод для створення ітератора використовуючи Aggregate.

**5. CheckInstalledFiles:** Метод, який використовує ітератор для перевірки всіх встановлених файлів у репозиторії. Реалізує ітерацію через список файлів і повідомляє, коли всі файли перевірені.

**6. Installer:** забезпечує взаємодію між репозиторієм файлів та процесом перевірки.

## Реалізувати один з розглянутих шаблонів за обраною темою

Згідно з завданням, було використано та реалізовано шаблон Iterator.

Реалізація шаблону Iterator:

**1. FileIterator (Ітератор):**

```
class FileIterator:  2 usages
    def __init__(self, file_repository):
        self.file_repository = file_repository
        self.index = 0

    def first(self):  2 usages (2 dynamic)
        self.index = 0

    def next(self):  2 usages (2 dynamic)
        self.index += 1

    def is_done(self):  3 usages (2 dynamic)
        return self.index >= len(self.file_repository.files)

    def current_item(self):  1 usage (1 dynamic)
        if not self.is_done():
            return self.file_repository.files[self.index]
        else:
            raise StopIteration("No more files in the repository.")
```

Рис. 2 – Клас FileIterator

- Відповідає за послідовний обхід файлів у сховищі.
- Містить методи для управління ітерацією: **first** (установка початкового стану), **next** (перехід до наступного елемента), **is_done** (перевірка завершення обходу), **current_item** (отримання поточного елемента).

- Інкапсулює логіку обходу колекції файлів.

**2. Iterator:**

```
class Iterator:  2 usages
    def __init__(self, iterator_imp):
        self.iterator_imp = iterator_imp

    def first(self):  2 usages (2 dynamic)
        return self.iterator_imp.first()

    def next(self):  2 usages (2 dynamic)
        return self.iterator_imp.next()

    def is_done(self):  2 usages (2 dynamic)
        return self.iterator_imp.is_done()

    def current_item(self):  1 usage (1 dynamic)
        return self.iterator_imp.current_item()
```

Рис. 3 – Клас Iterator

- Абстрактний загальний клас для ітераторів, який делегує виконання базових методів (**first**, **next**, **is_done**, **current_item**) конкретній реалізації, переданій як залежність (**iterator_imp**).
- Забезпечує узгоджений інтерфейс для взаємодії з різними реалізаціями ітераторів.

**3. Aggregate:**

```
class Aggregate:  2 usages
    def __init__(self, repository):
        self.repository = repository

    def create_iterator(self):  2 usages (1 dynamic)
        return Iterator(FileIterator(self.repository))
```

Рис. 4 – Клас Aggregate

- Шаблонний клас, що відповідає за створення ітератора для об'єкта, зокрема екземпляру **FileIterator**.
- Інкапсулює створення об'єкта ітератора, дозволяючи клієнтському коду не залежати від конкретних реалізацій.

**4. FileRepository (Агрегатор):**

```
def create_iterator(self):  1 usage (1 dynamic)  new *
    aggregate = Aggregate(self)
    return aggregate.create_iterator()
```

Рис. 5 – Клас FileRepository

- Клас, що представляє сховище файлів.
- Містить методи для додавання файлів (**add_file**) та створення ітератора для обходу файлів (**create_iterator**).
- Зберігає файли у внутрішньому списку (**files**), де кожен файл представлений у вигляді словника з інформацією про його ID, ім'я, шлях та тип.

**5. CheckInstalledFiles:**

```
def check_installed_files(file_repository):  2 usages
    print("Checking installed files...")
    iterator = file_repository.create_iterator()
    iterator.first()

    while not iterator.is_done():
        iterator.next()
    print("All files checked.")
```

Рис. 6 – Клас CheckInstalledFiles

- Функція для перевірки всіх файлів у сховищі, використовуючи ітератор.
- Ініціалізує ітерацію з першого елемента (**first**) та обходить всі елементи, перевіряючи їх послідовно.

**6. Installer (Клієнт):**

```
def check_installed_files(self):  1 usage  new *
    check_installed_files(self._installer_repository.get_file_repo())
```

Рис. 7 – Клас Installer

- Клас для забезпечення взаємодії між репозиторієм файлів та функцією перевірки файлів.
- У конструкторі отримує сховище файлів (**installer_repository**) та викликає функцію **check_installed_files** для перевірки встановлених файлів, використовуючи ітератор, створений репозиторієм.

## Діаграма класів для шаблону Iterator

Діаграма класів для шаблону Iterator:

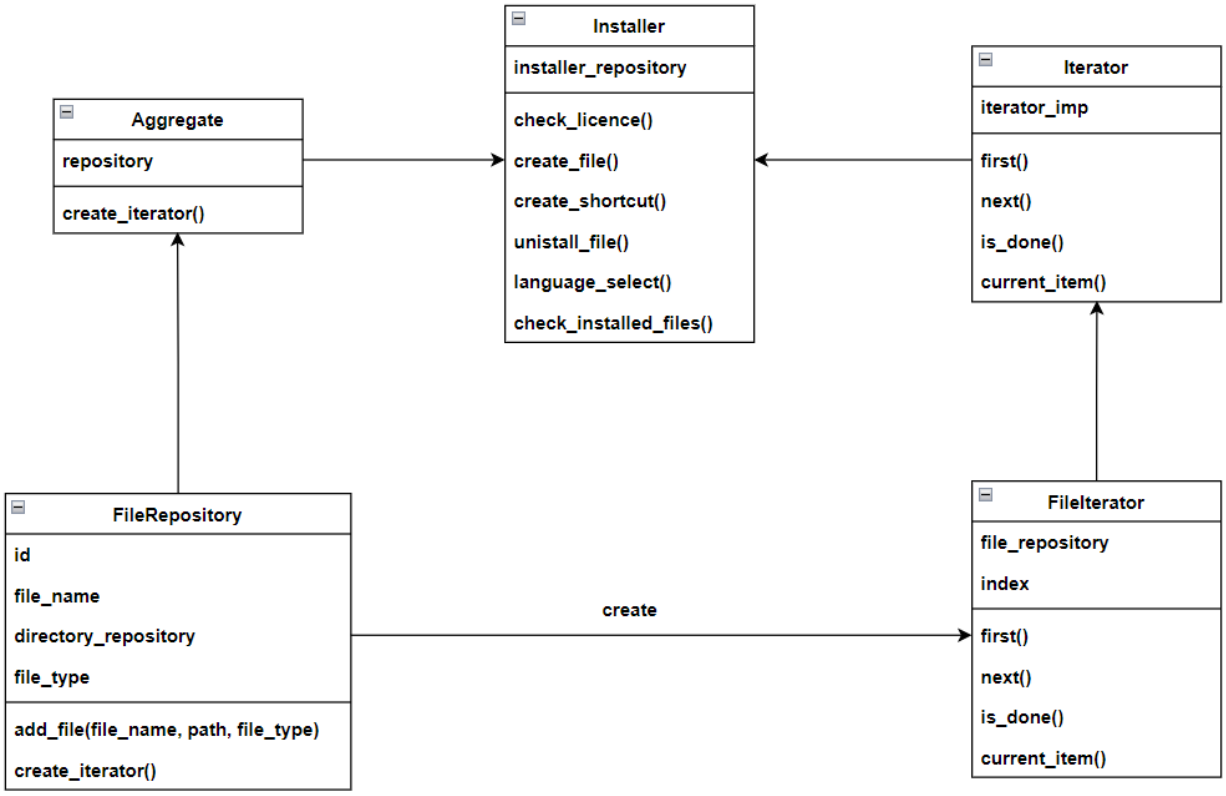



Рис. 8 – Діаграма класів для шаблону Iterator

## Проблема, яку допоміг вирішити шаблон Iterator

До впровадження шаблону **Iterator**, доступ до елементів колекції (наприклад, файлів у сховищі) був тісно пов'язаний із реалізацією внутрішньої структури колекції. Це створювало такі труднощі:

- Необхідність створення унікальної логіки ітерації для кожного випадку.
- Дублювання коду, що ускладнювало його підтримку та модифікацію.
- Порушення принципів інкапсуляції через прямий доступ до внутрішніх елементів колекції.

Реалізація шаблону **Iterator** дозволила:

1. **Інкапсулювати логіку обходу елементів у класі ітератора.** Це зробило код структурованішим та зручнішим для читання.
2. **Стандартизувати доступ до елементів.** Незалежно від способу зберігання, ітератор забезпечує єдиний інтерфейс для обходу колекції.
3. **Підтримувати різні способи обходу.** Наприклад, ітерація у прямому або зворотному порядку стала можливою без зміни структури даних.

## Переваги використання шаблону Iterator

1. **Інкапсуляція логіки обходу:** Вся логіка ітерації винесена в окремий клас, що підвищує розділення відповідальностей.
2. **Гнучкість:** Нові способи доступу до елементів можна додавати, модифікуючи лише ітератор.
3. **Спрощена підтримка:** Завдяки стандартизації, зміни в структурі колекції не впливають на код, що використовує ітератор.
4. **Масштабованість:** Дозволяє легко розширювати функціонал, наприклад, додавати фільтри для відбору певних елементів.

## Код та висновок

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

**Висновок:** в цій лабораторній роботі я реалізував шаблон Iterator для генератора інсталяційних пакетів (Installer generator), а саме було реалізовано перевірку файлів у репозиторії завдяки ітератору. Застосування шаблону Iterator дозволило: винести логіку обходу колекції в окремий клас, спрощуючи підтримку та модифікацію коду; підвищити масштабованість системи завдяки можливості додавання нових алгоритмів обходу; зробити систему зручнішою для використання та підтримки у майбутньому. Таким чином, реалізація цього шаблону для перевірки файлів у Installer Generator підтвердила його ефективність для роботи зі складними структурами даних, забезпечуючи чистоту та структурованість коду.