Шаблони классів

Шаблони проектування (Design Patterns) - це загальні рекомендації та архітектурні концепції, які допомагають розробникам створювати програмне забезпечення, яке є ефективним, легким для розуміння, та легко піддається розширенню та змінам.

Шаблони проектування розбивають завдання розробки програмного забезпечення на певні структури та взаємозв'язки між ними. Це дозволяє використовувати вже перевірені рішення для типових проблем.

Деякі популярні шаблони проектування включають:

1. **Singleton (Одинак):** Гарантує, що клас має тільки один екземпляр та надає глобальний доступ до нього.
2. **Factory Method (Фабричний метод):** Визначає метод для створення об'єктів, але залишає вибір конкретного класу-потомка до підкласів.
3. **Observer (Спостерігач):** Дозволяє одному об'єкту (subject) повідомляти інших об'єктів (observers) про зміни свого стану.
4. **Decorator (Декоратор):** Дозволяє динамічно приєднувати нові обов'язки до об'єкта без його модифікації.
5. **Adapter (Адаптер):** Дозволяє інтерфейсу одного класу працювати з іншим інтерфейсом.
6. **Strategy (Стратегія):** Визначає сімейство схожих алгоритмів та дозволяє їх замінювати один одним під час виконання.
7. **Command (Команда):** Інкапсулює запит як об'єкт, дозволяючи параметризувати клієнта зміною команд.
8. **State (Стан):** Дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану.
9. **MVC (Модель-Вид-Контролер):** Розбиває програму на три компоненти: модель (дані), вид (інтерфейс) та контролер (логіка).
10. **Composite (Компонувальник):** Дозволяє клієнтам обробляти окремі об'єкти та композити (комбінації об'єктів) однаковим способом.

Це лише кілька прикладів шаблонів проектування. Кожен шаблон вирішує конкретну задачу та має свої власні приклади використання. Обираючи шаблон для використання, важливо враховувати контекст та потреби вашого проекту.

Singleton

Шаблон проектування "Одинак" (Singleton) гарантує, що клас має тільки один екземпляр і надає глобальний доступ до цього екземпляра.

Одинак корисний, коли потрібно мати єдиний об'єкт, який керує об'єктом певного класу. Наприклад, це може бути клас, який керує підключеннями до бази даних або налаштуваннями програми.

У багатьох мовах програмування, включаючи Python, Одинак можна реалізувати за допомогою модулю (в Python, модуль - це об'єкт, який створюється лише один раз при імпортуванні).

Отже, ось приклад реалізації Одинака в Python:

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self, value):
        self.value = value

# Використання:

singleton1 = Singleton(1)
print(singleton1.value)  # Виведе 1

singleton2 = Singleton(2)
print(singleton2.value)  # Виведе 1, бо це той самий екземпляр
```

У цьому прикладі, за допомогою перевизначення методу `__new__`, ми гарантуємо, що клас має тільки один екземпляр. При кожному створенні нового екземпляру, ми перевіряємо, чи вже існує екземпляр, і повертаємо його, якщо так.

Отже, `singleton1` і `singleton2` вказують на один і той самий екземпляр класу `Singleton`.

Цей приклад демонструє базовий підхід до реалізації Одинака. У реальних проектах можуть бути використані різні підходи для забезпечення потокобезпеки, лінивого ініціалізації, та інших аспектів.

Шаблон проектування "Одинак" (Singleton) використовується для того, щоб гарантувати, що клас має тільки один екземпляр і надає глобальний доступ до цього екземпляра.

Ось декілька ситуацій, коли використання Одинака може бути корисним:

1. **Менеджер налаштувань:** Якщо у вас є клас, який керує налаштуваннями програми (наприклад, параметри підключення до бази даних), ви хочете, щоб у вас був тільки один екземпляр цього менеджера, щоб уникнути можливих конфліктів.
2. **Підключення до бази даних:** Якщо ви маєте клас, який керує підключеннями до бази даних, вам потрібен тільки один екземпляр цього класу, щоб ефективно керувати підключеннями.
3. **Логгер:** Одинак може бути використаний для створення глобального логгера, який дозволяє легко записувати події в журнал у всіх частинах програми.
4. **Пул об'єктів:** Якщо у вас є потреба в обмеженій кількості екземплярів певного класу (наприклад, пул з'єднань до сервера), Одинак може допомогти забезпечити, щоб ця кількість була додержана.
5. **Кешування даних:** Одинак може використовуватися для створення глобального кешу, який дозволяє ефективно зберігати та отримувати дані з будь-якої частини програми.

Загалом, Одинак допомагає забезпечити, що об'єкт класу буде створений тільки один раз і що усі інші виклики до цього класу будуть використовувати той самий екземпляр. Це дозволяє краще керувати глобальними ресурсами та уникнути надмірного використання пам'яті.

Factory Method

Шаблон проектування "Фабричний метод" (Factory Method) відноситься до породжувальних шаблонів. Він використовується для створення об'єктів певного

типу, але дозволяє субкласам вибирати клас для створення.

Головна ідея в тому, що ми визначаємо інтерфейс для створення об'єктів в базовому класі, а конкретна реалізація цього методу залишається на вузьких класах.

Ось приклад реалізації Фабричного методу:

```
class Product:
    def create(self):
        raise NotImplementedError

class ConcreteProductA(Product):
    def create(self):
        return "Product A"

class ConcreteProductB(Product):
    def create(self):
        return "Product B"

class Creator:
    def factory_method(self):
        raise NotImplementedError

    def get_product(self):
        product = self.factory_method()
        return f"Creator got a {product}"

class ConcreteCreatorA(Creator):
    def factory_method(self):
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
    def factory_method(self):
        return ConcreteProductB()
```

У цьому прикладі, `Product` є базовим класом, який визначає метод `create`. Класи `ConcreteProductA` та `ConcreteProductB` є конкретними реалізаціями цього методу для різних типів продуктів.

`Creator` є ще одним базовим класом, який визначає метод `factory_method` і метод `get_product`. Класи `ConcreteCreatorA` та `ConcreteCreatorB` визначають конкретні реалізації для `factory_method`.

Використання цих класів:

```
creator_a = ConcreteCreatorA()
product_a = creator_a.get_product()
print(product_a)  # Виведе: "Creator got a Product A"

creator_b = ConcreteCreatorB()
product_b = creator_b.get_product()
print(product_b)  # Виведе: "Creator got a Product B"
```

Цей паттерн дозволяє змінювати тип створеного продукту, не змінюючи коду Creator. Це особливо корисно, коли у вас є багато різних типів продуктів, і ви хочете мати можливість їх легко замінювати або додавати нові.

Шаблон проектування "Фабричний метод" (Factory Method) використовується для створення об'єктів певного типу, але дозволяє субкласам вибирати конкретну реалізацію цього об'єкта.

Ось декілька ситуацій, коли використання Фабричного методу може бути корисним:

1. **Інкапсуляція створення об'єкта:** Фабричний метод дозволяє відокремити процес створення об'єкта від його використання. Це дозволяє змінювати тип створеного об'єкта без зміни виклику коду, який його використовує.
2. **Можливість вибору підкласу для створення:** Фабричний метод надає можливість субкласам вибрати, який саме клас вони хочуть створити. Це особливо корисно, коли у вас є кілька різних класів, які відповідають за різні типи обробки або дії.
3. **Керування екземплярами об'єктів:** Фабричний метод може використовуватися для забезпечення, щоб в одному класі були створені тільки певні типи об'єктів. Наприклад, при створенні пулу з'єднань до бази даних.
4. **Легке розширення та зміна:** Фабричний метод дозволяє легко додавати нові типи об'єктів без необхідності зміни вихідного коду.
5. **Тестування та налагодження:** Фабричний метод спрощує тестування, оскільки можна замінювати фабричний метод на підроблений об'єкт для тестування.

Загалом, Фабричний метод є потужним інструментом для створення об'єктів з урахуванням потреб програми та легкості їх майбутнього розширення та зміни.

MVC

MVC (Model-View-Controller) - це шаблон архітектури програмного забезпечення, який використовується для розділення відповідальностей у програмі на три основні компоненти: Model (Модель), View (Вид) і Controller (Контролер).

1. **Модель (Model):** Модель представляє собою даний компонент і відповідає за обробку даних, логіку бізнес-логіки та взаємодію з базою даних або іншими джерелами даних. Вона не залежить від інтерфейсу користувача та представляє незалежну абстракцію даних.
2. **Вид (View):** Вид представляє собою компонент, який відповідає за відображення даних користувачу та інтерактивну взаємодію з ним. Вид не має власної бізнес-логіки і отримує дані від моделі для відображення.
3. **Контролер (Controller):** Контролер відповідає за прийняття вхідних даних від користувача, взаємодію з моделлю для оновлення даних та керування відображенням. Він вирішує, які дані використовувати та який вигляд показати.

Розділення програми на ці компоненти дозволяє забезпечити чистоту коду, покращити його читабельність та підтримуваність, а також спрощує тестування.

Приклад використання MVC можна побачити в веб-розробці, де модель представляє базу даних та бізнес-логіку, вид - веб-інтерфейс користувача, а контролер - серверну логіку, яка обробляє запити від клієнта та взаємодіє з моделлю для надсилання та отримання даних.

Цей шаблон сприяє розподілу відповідальностей у програмі та полегшує розвиток та підтримку складних програм.

Шаблон архітектури програмного забезпечення "Model-View-Controller" (MVC) потрібен для розділення відповідальностей у програмі та поліпшення її структури.

Ось декілька причин, чому використання MVC є корисним:

1. **Розділення відповідальностей:** MVC дозволяє розділити програму на три основні компоненти: модель, вид і контролер. Це допомагає зберігати код організованим та легко читабельним.
2. **Покращення підтримуваності:** Кожен компонент (модель, вид, контролер) відповідає лише за свої власні функціональні обов'язки, що полегшує розуміння та редагування кожного компонента окремо.

3. **Тестування:** Розділення логіки програми на окремі компоненти спрощує тестування кожного з них окремо. Наприклад, можна легко тестувати модель, не запускаючи інтерфейс користувача.
4. **Можливість змінювати інтерфейс без впливу на бізнес-логіку:** Вид (інтерфейс) та контролер можна змінювати без впливу на модель, що дозволяє змінювати або модернізувати інтерфейс користувача без необхідності перероблення логіки додатку.
5. **Підтримка багатомовності:** MVC спрощує роботу з багатомовними програмами, оскільки дозволяє легко замінювати частини відображення (вигляду) для різних мов.
6. **Полегшення роботи над великими проектами:** Великі проекти з великою кількістю коду стають більш керованими та структурованими за допомогою MVC.

Отже, використання шаблону архітектури MVC допомагає зробити програму більш організованою, підтримуваною та тестируваною.

Повна енциклопедія -

Патерни/шаблони проектування

Патерни проектування описують типові способи вирішення поширених проблем при проектуванні програм.

https://refactoring.guru/uk/design-patterns