Объектно - ориентированное проектирование и паттерны проектирования

Объектно ориентированное проектирование

Объектно-ориентированное проектирование (ООП) — это метод разработки программного обеспечения, основанный на концепции **объектов**, которые представляют данные и действия над этими данными. ООП помогает структурировать и организовывать код, делая его более гибким, расширяемым и легким для поддержки.

Почему объекты?

- Фокусирование на объектах упрощает для нас понимание сложных вещей.
- Уделяем внимание лишь важным аспектам

Пример : преподаватель - студент

Объектно - ориентированное программирование - парадигма программирования, основанная на представлении предметной области в виде взаимосвязанных абстрактных объектов и их реализаций.

Основные принципы ООП включают:

- Инкапсуляцию
- Наследование
- Полиморфизм
- Абстракцию

Концепция

Класс и Объект
Класс - принципиально новый тип данных.

Класс представляет собой множество объектов:
- имеющих общую структуру
- обладающих одинаковым поведением.

Класс является дальнейшим развитием типа структура (запись)

Концепция

Объект является представителем (экземпляром) какого-либо класса.

Объект обладает:
- состоянием,
- поведением,
- идентичностью.

Состояние объекта характеризуется
- набором его свойств (атрибутов),
- текущими значениями каждого из этих свойств.

Поведение объекта - выполнения определенной последовательности характерных для него действий.

Идентичность объекта – это свойство (или набор свойств) объекта, которое позволяет отличить его от всех прочих объектов того же типа (класса).

Концепция

Свойства – перечень параметров объекта, которые определяют внешний вид и поведение объекта, выделяют уникальные особенности каждого экземпляра.

К свойствам относятся: имя, тип, значение, цвет, размер и др.

Состояние – совокупность всех свойств данного объекта.

Метод - это некоторое действие (операция), которое можно выполнять над данным объектом. В результате этого действия в объекте что-нибудь меняется (например, местоположение, цвет и др.).

Другими словами можно еще сказать, методом называется команда, которую может выполнять объект.

Концепция

События – сигналы, формируемые внешней средой, на которые объект должен отреагировать соответствующим образом.
Средой взаимодействия объектов являются сообщения, генерируемые в результате наступления различных событий.

События наступают в результате действий пользователя – перемещение курсора пользователя, нажатия кнопок мыши или клавиш на клавиатуре, а также в результате работы самих объектов.
Для каждого объекта определено множество событий, на которые он может реагировать.

Объекты – это «существительные»,
Свойства объекта – «прилагательные»,
Методы объекта – это «глаголы».

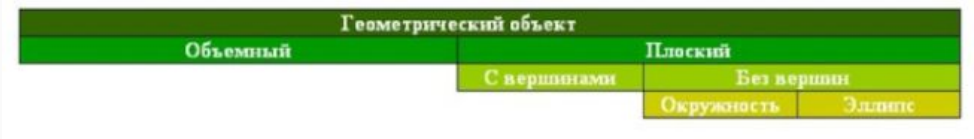
Классы объектов

Классом называют особую структуру, которая может иметь в своем составе поля, методы и свойства.

Класс представляет собой множество объектов
- имеющих общую структуру,
- обладающих одинаковым поведением.

Класс выступает в качестве объектного типа данных, а объект – это конкретный экземпляр класса.

Каждый конкретный класс имеет свои особенности поведения и характеристики, определяющие этот класс.



Наивысший уровень – самый общий и самый простой. Каждый последующий уровень более специфический и менее общий. На самом последнем уровне можно определить цвет, стиль заполнения, величину радиуса окружности и т.п.

Пример

```
class Point
{
public:
double x, y;
};

class Triangle
{
public:
double GetArea();
double GetPerimeter();
Point GetCenter();

void Move(double dx, double dy);
void Scale(double sx, double sy);
void Rotate(Point center, double angle);
Point p0, p1, p2;
};
```

Принцип абстракции

Выделение абстракций, то есть анализ предметной области, для которой составляется программа, с целью определения основных объектов этой предметной области, их свойств, отношений между объектами, возможных операций над объектами или их составляющими.

Объекты представляют неполную информацию о реальных сущностях предметной области Абстракция позволяет оперировать с объектом на уровне, адекватном решаемой задаче.

Высокоуровневые обращения к объекту могут обрабатываться с помощью вызова функций и методов низкого уровня.

Принцип инкапсуляции

Инкапсуляция - способность объекта скрывать внутреннее устройство своих свойств и методов.

Согласно данному принципу, класс должен рассматриваться как черный ящик
Внешний пользователь не знает детали реализации объекта и работает с ним только путем предоставленного объектом интерфейса.

Следование данному принципу может уменьшить число связей между классами и упростить их независимую реализацию, модификацию и тестирование.

```
class IntStack
{
public:
void Push(int value);
int Pop();
bool IsEmpty()const;
private:
// здесь располагаются данные
// необходимые для реализации стека целых чисел
};
```

Инкапсуляция и ограничение доступа

Инкапсуляция предполагает возможность ограничения доступа к данным (полям) класса.

Это позволяет

- упростить интерфейс класса, показав наиболее существенные для внешнего пользователя данные и методы,
- обеспечить возможность внесения изменений в реализацию класса без изменения других классов (важно для дальнейшего сопровождения и модернизации программного кода).

При сокрытии полей объекта доступ к ним осуществляется только посредством методов класса. Это защищает данные от внешнего вмешательства или неправильного использования.

Управление доступом

Ключи доступа:
private - элементы данных могут использоваться только функциями-методами класса, к которому принадлежат эти элементы данных

public- элементы данных могут использоваться любыми функциями программы

protected- элементы данных могут использоваться функциями-методами того же класса, к которому принадлежат эти элементы данных, а также функциями- методами производных классов (классов-потомков)

По умолчанию ключ доступа private. Т.е. если ключи доступа не указаны, то все элементы класса являются скрытыми (недоступными).

Попытка обратиться в программе к скрытым данным или методам вызывает сообщение:
is not accessible

Принцип наследования

Наследование позволяет описать новый класс на основе уже существующего родительского (базового) класса.

Класс-потомок может добавить свои собственные свойства и методы, пользоваться методами и свойствами базового класса.

Наследование позволяет строить иерархии классов

```
class Plane
{
public:
void TakeOff();
void Fly();
void Land();
private:
double m_fuel;
};

class MilitaryPlane : public Plane
{
public:
void Attack();
private:
int m_ammo;
```

Принцип полиморфизм

Это возможность заменить в классе потомке метод класса родителя, сохранив при этом имя метода.
Это свойство классов решать схожие по смыслу проблемы разными способами.

Цель полиморфизма - использование одного имени для задания общих для класса действий.

Для изменения метода необходимо перекрыть его в потомке, т.е. объявить в потомке одноименный метод и реализовать в нем нужные действия.

В результате объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу.

Концепция полиморфизма - идея «один интерфейс - множество методов».
Полиморфизм позволяет манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

Принцип полиморфизм

```
class Shape
{
public:
virtual double GetArea()=0;
};

class Rectangle : public Shape
{
public:
virtual double GetArea()
{
return width * height;
}
private:
double width, height;
};

class Circle : public Shape
{
public:
virtual double GetArea()
{
return 3.1415927 * radius * radius;
}
```

Паттерны проектирования

Шаблоны проектирования - это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками, как решать определенные проблемы в определенных ситуациях.

Википедия: Шаблон проектирования, или паттерн, в разработке программного обеспечения - повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

Однако:

- шаблоны проектирования не являются решением всех ваших проблем;
- не пытайтесь использовать их в обязательном порядке — это может привести к негативным последствиям. Шаблоны — это подходы к решению проблем, а не решения для поиска проблем;
- если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

Типы паттернов

Паттерны бывают следующих трех видов:

- **Порождающие** - эти паттерны связаны с процессом создания объектов и обеспечивают гибкость и повторное использование кода. Примеры таких паттернов: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
- **Структурные** - Они определяют способы композиции классов и объектов для создания новых структур. Примеры структурных паттернов: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- **Поведенческие** - Эти паттерны определяют взаимодействие между объектами и облегчают обмен информацией между ними. Примеры поведенческих паттернов: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Если говорить простыми словами, то это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов.

Singleton

Singleton — это порождающий паттерн, который гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру.

```
class Settings:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

Здесь мы создали класс Settings и переопределили метод __new__. Он вызывается при создании нового объекта. Мы проверяем, существует ли уже экземпляр в _instance. Если нет - создаем. Если есть - возвращаем существующий.

Таким образом, сколько бы раз мы не вызывали Settings(), у нас будет только один объект.

```
s1 = Settings()
s2 = Settings()

print(s1 == s2) # True - объект один и тот же
```

Factory

Factory — это порождающий паттерн, нужен, когда у тебя много похожих классов, и ты хочешь упростить их создание.

```
class MonsterFactory:
    def create_monster(self, type):
        if type == "goblin":
            return Goblin()
        elif type == "skeleton":
            return Skeleton()
        elif type == "orc":
            return Orc()

factory = MonsterFactory()
monster = factory.create_monster("skeleton")
```

Например, в игре у тебя есть классы Goblin, Skeleton, Orc - разных монстров. Чтобы не писать в коде goblin = Goblin(), skeleton = Skeleton() и т.д., можно сделать фабрику.

Теперь, чтобы создать монстра, достаточно вызвать метод create_monster и указать тип. Код становится чище и проще для понимания.

Паттерн Factory - очень удобный способ создавать схожие объекты в одном месте, не засоряя код.

Adapter

Adapter — это структурный паттерн, который позволяет объектам с несовместимыми интерфейсами работать вместе.

```
interface EuropeanPlug {
    void useEuropeanPlug();
}

class BritishPlug {
    void useBritishPlug() {
        System.out.println("Using British plug");
    }
}

class PlugAdapter implements EuropeanPlug {
    private BritishPlug britishPlug;

    public PlugAdapter(BritishPlug britishPlug) {
        this.britishPlug = britishPlug;
    }

    @Override
    public void useEuropeanPlug() {
        britishPlug.useBritishPlug();
    }
}
```

Observer

Observer — это поведенческий паттерн, который определяет отношение «один-ко-многим» между объектами. Когда один объект (субъект) изменяется, все зависимые объекты (наблюдатели) уведомляются об этом и автоматически обновляются.

Представь, в игре у тебя есть юниты и здания. И юниты должны автоматически атаковать вражеские здания, когда те появляются.

Здесь Building наследует от Observable и может уведомлять наблюдателей о событиях через notify(). A Unit реализует интерфейс Observer и получает уведомления в методе update().

Таким образом юниты могут автоматически реагировать на появление зданий. Это и есть Observer - мощный паттерн для реакции объектов друг на друга.

```python
from __future__ import annotations
from abc import ABC, abstractmethod
from random import randint

class Observable(ABC):
    def __init__(self):
        self.observers = []

    def notify(self):
        for observer in self.observers:
            observer.update(self)

    def attach(self, observer):
        self.observers.append(observer)

class Observer(ABC):
  @abstractmethod
  def update(self, subject: Observable) -> None:
    pass

class Unit(Observer):
    def update(self, building: Building):
        print(f"Unit is attacking {building}")
        building.hp -= randint(1, 10)

class Building(Observable):
    def __init__(self):
        super().__init__()
        self.hp = 100

# Использование

building = Building()
unit = Unit()

building.attach(unit)

building.notify() # Unit attacks building
```

Другие паттерны

Command - помещает задачи в объекты-команды для многократного выполнения или отмены.

Strategy - позволяет легко переключать разные алгоритмы решения задачи.

Decorator - добавляет объектам новое поведение без изменения кода.

Все эти паттерны тоже пригодятся в больших проектах. Поэтому стоит разобраться с ними, когда появится время.

Литература

https://tproger.ru/translations/design-patterns-simple-words-1

https://youtu.be/ChEdFh7Q-Vw?si=kSmFHfSv1OSBipqo

https://youtu.be/a4fVPvDTYVQ?si=YC2o6kmwNVHZA59A

https://www.youtube.com/watch?v=-AaVsHkgWcQ

https://www.youtube.com/watch?v=VqgXn7wsPsc

https://www.youtube.com/watch?v=9HJ55_fCtGg