Каждый из шаблонов предназначен для решения своего круга практических задач:

* **Порождающие шаблоны**, предназначенные для создания новых объектов в системе;
* **Структурные шаблоны**, решающие задачи компоновки системы на основе классов и объектов;
* **Шаблоны поведения**, предназначенные для распределения обязанностей между объектами в системе.

**Структурные шаблоны**

**Адаптер (Adapter)**

**Проблема:**необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

**Фасад (Facade)**

Шаблон “фасад” ­ структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

**Проблема**: как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

**Заместитель (Proxy)**

Заместитель — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы.

**Проблема**: необходимо управлять доступом к объекту так, чтобы не создавать громоздкие объекты «по требованию».

**Декоратор**

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

**Поведенческие шаблоны**

**Стратегия (Strategy)**

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов­клиентов, которые его используют.

**Проблема:**по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

**Команда (Command)**

Команда — поведенческий шаблон проектирования, используемый при объектно­ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект­команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

**Порождающие шаблоны**

* **Шаблон «Фабричный метод» (Factory Method).** В данном шаблоне вводится полиморфный класс Factory, в котором определяется интерфейс фабричного метода, подобного CreateWarrior( ), а ответственность за создание объектов конкретных классов переносится на производные от Factory классы, в которых этот метод переопределяется;
* **Шаблон «Абстрактная фабрика» (Abstract Factory).** Использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов. Для приведённого выше примера, шаблон может предоставлять методы для создания всех типов воинов;
* **Шаблон «Строитель» (Builder).** Определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта;
* **Шаблон «Прототип» (Prototype).** Создаёт новые объекты с помощью прототипов (создавая копии самого себя);
* **Шаблон «Синглетон» (Singleton).** Позволяет создавать только один экземпляр класса.

Вот основные порождающие шаблоны проектирования с примерами на Python:

**1. Singleton (Одиночка)**

Шаблон «Одиночка» гарантирует, что у класса будет только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Пример:**

class Singleton:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.\_instance

# Использование:

singleton1 = Singleton()

singleton2 = Singleton()

print(singleton1 is singleton2) # True, оба переменные ссылаются на один и тот же объект

**2. Factory Method (Фабричный метод)**

Шаблон «Фабричный метод» позволяет классу делегировать создание объектов другим классам или методам, избегая прямого указания конкретного класса объекта.

**Пример:**

from abc import ABC, abstractmethod

class Transport(ABC):

@abstractmethod

def deliver(self):

pass

class Truck(Transport):

def deliver(self):

return "Delivery by Truck"

class Ship(Transport):

def deliver(self):

return "Delivery by Ship"

class TransportFactory:

def create\_transport(self, transport\_type):

if transport\_type == "truck":

return Truck()

elif transport\_type == "ship":

return Ship()

# Использование:

factory = TransportFactory()

transport = factory.create\_transport("truck")

print(transport.deliver()) # Delivery by Truck

**3. Abstract Factory (Абстрактная фабрика)**

Абстрактная фабрика предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

**Пример:**

from abc import ABC, abstractmethod

# Продукты

class Chair(ABC):

@abstractmethod

def sit\_on(self):

pass

class Sofa(ABC):

@abstractmethod

def lie\_on(self):

pass

# Конкретные продукты

class ModernChair(Chair):

def sit\_on(self):

return "Sitting on a modern chair"

class VictorianSofa(Sofa):

def lie\_on(self):

return "Lying on a Victorian sofa"

# Абстрактная фабрика

class FurnitureFactory(ABC):

@abstractmethod

def create\_chair(self):

pass

@abstractmethod

def create\_sofa(self):

pass

# Конкретная фабрика

class ModernFurnitureFactory(FurnitureFactory):

def create\_chair(self):

return ModernChair()

def create\_sofa(self):

return VictorianSofa()

# Использование:

factory = ModernFurnitureFactory()

chair = factory.create\_chair()

sofa = factory.create\_sofa()

print(chair.sit\_on()) # Sitting on a modern chair

print(sofa.lie\_on()) # Lying on a Victorian sofa

**4. Builder (Строитель)**

Шаблон «Строитель» позволяет пошагово создавать сложные объекты, разделяя процесс создания на этапы.

**Пример:**

class House:

def \_\_init\_\_(self):

self.walls = None

self.roof = None

self.windows = None

def \_\_str\_\_(self):

return f"House with {self.walls}, {self.roof}, and {self.windows}"

class HouseBuilder:

def \_\_init\_\_(self):

self.house = House()

def build\_walls(self):

self.house.walls = "wooden walls"

return self

def build\_roof(self):

self.house.roof = "shingled roof"

return self

def build\_windows(self):

self.house.windows = "4 windows"

return self

def get\_house(self):

return self.house

# Использование:

builder = HouseBuilder()

house = builder.build\_walls().build\_roof().build\_windows().get\_house()

print(house) # House with wooden walls, shingled roof, and 4 windows

**5. Prototype (Прототип)**

Шаблон «Прототип» позволяет создавать новые объекты, копируя существующие. Этот подход особенно полезен, когда создание объекта «с нуля» является дорогим.

**Пример:**

import copy

class Prototype:

def \_\_init\_\_(self, value):

self.value = value

def clone(self):

return copy.deepcopy(self)

# Использование:

original = Prototype(42)

clone = original.clone()

print(original.value) # 42

print(clone.value) # 42

print(original is clone) # False, это разные объекты