Министерство науки и образования РФ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Заполярный государственный университет им. Н. М. Федоровского»
Факультет электроэнергетики, экономики и управления
Кафедра информационных систем и технологий

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Тема: *“Архитектура, шаблоны и антипаттерн”*

Проблема терминологии

- Иногда одно и то же решение называют архитектурой, стилем или паттерном — в зависимости от масштаба
- Одно и то же решение может применяться на разном уровне ИС
- Решения могут сосуществовать в рамках одной ИС или быть взаимоисключающими
- Решения могут “наследовать идеи” друг от друга

Паттерны GoF

Приёмы объектно-ориентированного проектирования. Паттерны проектирования

Порождающие

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Структурные

- Adapter
- Bridge
- Composite
- Decorator
- Facad
- Flyweight
- Proxy

Поведенческие

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Порождающие Паттерны

Singleton

- **Singleton** - у класса есть только один экземпляр, и он предоставляет к нему глобальную точку доступа. При попытке создания данного объекта он создаётся только в том случае, если ещё не существует, в противном случае возвращается ссылка на уже существующий экземпляр и нового выделения памяти не происходит

Примеры singleton

Python:

```python
class Singleton:
    _instance = None
    def __new__(cls):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance
```

JS:

```javascript
function Singleton() {
 const instance = Singleton.instance;
 if (instance) return instance;
Singleton.instance = this;
}
```

** примеры являются “шуточными” и не рекомендуется к повсеместному использованию!*

Factory Method -> Abstract factory

Factory Method – метод, предоставляющий подклассам (дочерним классам, субклассам) интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать

Abstract factory - предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов

Примеры Factory Method на Python

```python
class Transport(ABC):
    @abstractmethod
    def deliver(self, cargo: str) ->
str:
        pass
```

```python
class Truck(Transport):
    def deliver(self, cargo: str) ->
str:
        return f"Везу '{cargo}' по
дороге на грузовике"
```

```python
class Ship(Transport):
    def deliver(self, cargo: str) ->
str:
        return f"Везу '{cargo}' по
морю на корабле"
```

```python
class Logistics(ABC):
    @abstractmethod
    def factory_method(self) ->
Transport:
        """Создает объект транспорта"""
        pass

    def plan_delivery(self, cargo: str)
-> str:
        """Бизнес-логика, которая не
знает конкретный класс транспорта"""
        transport =
self.factory_method()
        return transport.deliver(cargo)
```

```python
class RoadLogistics(Logistics):
    def factory_method(self) ->
Transport:
        return Truck()
```

```python
class SeaLogistics(Logistics):
    def factory_method(self) ->
Transport:
        return Ship()
```

```python
def client_code(logistics: Logistics, cargo: str) ->
None:
    print(logistics.plan_delivery(cargo))
client_code(RoadLogistics(), "телевизоры")
client_code(SeaLogistics(), "контейнеры")
```

Abstract factory

```
class Transport(ABC):
    @abstractmethod
    def deliver(self, cargo: str) ->
str:
        pass

class Packaging(ABC):
    @abstractmethod
    def pack(self, cargo: str) ->
str:
        pass
```

```
class Logistics(ABC):
    @abstractmethod
    def transport_factory_method(self) -> Transport:
        """Создает объект транспорта"""
        pass

    @abstractmethod
    def packaging_factory_method(self) -> Packaging:
        pass


    def plan_delivery(self, cargo: str) -> str:
        """Бизнес-логика, которая не знает
конкретный класс транспорта"""
        transport = self.factory_method()
        return transport.deliver(cargo)
```

Builder

- **Builder** - способ пошагового создания составного объекта

```
class House:
    def __init__(self, walls, roof, garage,
pool, garden):
        self.walls = walls
        self.roof = roof
        self.garage = garage
        self.pool = pool
        self.garden = garden

# делаем дом по условию
if(1==1):
    walls = "Кирпичные стены"
else:
    walls = None
if(1==1):
    roof = "Черепичная крыша"
else:
    roof = None
if(1==1):
    garage = "Гараж на 2 машины"
else:
    garage = None
if(1==1):
    pool = "Бассейн 5м"
else:
    pool = None
if(1==1):
    garden = "Сад с яблонями"
else:
    garden = None
house = House(walls,roof,garage,pool,garden)
```

** Код не использующий паттерн Builder*

Builder

```
class House:
    def __init__(self):
        self.walls = None
        self.roof = None
        self.garage = None
        self.pool = None
        self.garden = None

class HouseBuilder:
    def __init__(self):
        self._house = House()

    def add_walls(self, walls):
        self._house.walls = walls
        return self

    def add_roof(self, roof):
        self._house.roof = roof
        return self

    def add_garage(self, garage):
        self._house.garage = garage
        return self

    def add_pool(self, pool):
        self._house.pool = pool
        return self

    def add_garden(self, garden):
        self._house.garden = garden
        return self

    def build(self):
        return self._house

builder = HouseBuilder()
if 1 == 1:
    builder.add_walls("Кирпичные стены")

if 1 == 1 and 1 == 1:
    builder.add_roof("Черепичная крыша")
.add_garage("Гараж на 2 машины")
…
myHouse = builder.build()
```

** Код использующий паттерн Builder*

Prototype

- **Prototype –** паттерн при котором создание нового объекта базируется на клонировании существующего

```
class Config:
    def __init__(self, host, port,
cache, debug):
        self.host = host
        self.port = port
        self.cache = cache
        self.debug = debug

dev_config = Config("localhost", 5432,
True, True)
prod_config = Config("prod.db", 5432,
True, False)
```

```
class Config:
    def __init__(self, host, port, cache, debug):
        self.host = host
        self.port = port
        self.cache = cache
        self.debug = debug

    def clone(self):
        return copy.deepcopy(self)

base_config = Config("localhost", 5432, True, False)

dev_config = base_config.clone()
dev_config.debug = True

prod_config = base_config.clone()
prod_config.host = "prod.db"
```

** примеры являются "шуточными"*

Структурные Паттерны

Adapter

Adapter - структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс. Другими словами — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Facade

Facade - позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы

Bridge

Bridge - разделение абстракции(что делаем) и реализации(как/чем делаем) так, чтобы они могли изменяться независимо

Абстракция	Реализация		
A1	B1	=>	A1B1, A1B2, A2B1, A2B2 ...
A2	B2		
A3	B3		

Composite

Composite - шаблон, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково

```
class Unit(ABC):
    """Абстрактный компонент, в данном случае
это - отряд (отряд может
    состоять из одного солдата или более)"""

    @abstractmethod
    def print(self) -> None:
        """Вывод данных о компоненте"""
        pass

class Archer(Unit):
    """Лучник"""

    def print(self) -> None:
        print('лучник', end=' ')
```

```
class Knight(Unit):
    """Рыцарь"""

    def print(self) -> None:
        print('рыцарь', end=' ')

class Swordsman(Unit):
    """Мечник"""

    def print(self) -> None:
        print('мечник', end=' ')
```

Composite

```
class Squad(Unit):
    """Компоновщик - отряд, состоящий более чем из одного человека. Также
    может включать в себя другие отряды-компоновщики."""

    def __init__(self):
        self._units = []

    def print(self) -> None:
        print("Отряд {} (".format(self.__hash__()), end=' ')
        for u in self._units:
            u.print()
        print(')')

    def add(self, unit: Unit) -> None:
        """Добавление нового отряда"""
        self._units.append(unit)
        unit.print()
        print('присоединился к отряду {}'.format(self.__hash__()))
        print()

    def remove(self, unit: Unit) -> None:
        """Удаление отряда из текущего компоновщика"""
        for u in self._units:
            if u == unit:
                self._units.remove(u)
                u.print()
                print('покинул отряд {}'.format(self.__hash__()))
                print()
                break
        else:
            unit.print()
            print('в отряде {} не найден'.format(self.__hash__()))
            print()
```

Decorator

Декоратор позволяет динамически добавлять объекту новое поведение, не изменяя его

Flyweight

**Flyweight - ** шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

Proxy

Proxy – объект, который контролирует доступ к другому объекту, перехватывая все вызовы

Министерство науки и образования РФ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Заполярный государственный университет им. Н. М. Федоровского»
Факультет электроэнергетики, экономики и управления
Кафедра информационных систем и технологий

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Тема: *“Архитектура, шаблоны и антипаттерн”*