Порождающие шаблоны проектирования

Паттерны проектирования - это общие решения для типичных проблем, которые возникают при разработке программного обеспечения. Они представляют собой набор лучших практик, разработанных опытными разработчиками, и помогают упростить архитектуру приложения, сделать его более гибким и легко поддерживаемым.

Паттерны проектирования можно разделить на три основные группы:

1. Структурные паттерны: эти паттерны касаются состава классов и объектов, и они помогают в создании больших систем из более мелких частей. Они упрощают архитектуру приложения, делая ее более гибкой и легко поддерживаемой. Примерами структурных паттернов являются Адаптер, Декоратор, Фасад, Композит, Мост, Приложение, Прокси.
2. Порождающие паттерны: эти паттерны касаются создания объектов, и они помогают в обеспечении гибкости и упрощении процесса создания объектов. Они также помогают в обеспечении того, что объекты создаются в соответствии с определенными критериями. Примерами порождающих паттернов являются Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка.
3. Поведенческие паттерны: эти паттерны касаются взаимодействия между объектами, и они помогают в создании гибких и легко расширяемых систем. Они также помогают в обеспечении того, что объекты взаимодействуют друг с другом в соответствии с определенными критериями. Примерами поведенческих паттернов являются Команда, Интерпретатор, Итератор, Посредник, Наблюдатель, Государство, Стратегия, Шаблонный метод, Посетитель.

Строитель
Паттерн проектирования "Строитель" (Builder) относится к порождающим паттернам и предназначен для решения задач, связанных с созданием сложных объектов.

Метафора: Представьте, что вы строите дом. Вы можете сами выполнять все работы, но это будет очень сложно и займет много времени. Вместо этого, вы нанимаете бригаду строителей, каждый из которых отвечает за свою часть работы: один закладывает фундамент, другой возводит стены, третий устанавливает крышу и так далее. В итоге, вы получаете готовый дом, не тратя лишних усилий и времени. Паттерн "Строитель" работает похоже. Он позволяет создавать сложные объекты, разделяя процесс их создания на более мелкие подзадачи, и выполняя эти подзадачи независимо друг от друга.

Когда стоит использовать паттерн "Строитель":

- Когда процесс создания объекта сложен и требует много шагов.
- Когда требуется создавать различные варианты объекта, и процесс их создания отличается только несколькими шагами.
- Когда требуется разделить процесс создания объекта и его представление.

Когда не стоит использовать паттерн "Строитель":

- Когда процесс создания объекта прост и состоит из нескольких шагов.
- Когда требуется создавать только один вариант объекта.
- Когда требуется создать объект сразу целиком, без разделения на подзадачи.

Принципиальная схема паттерна "Строитель" в языке PlantUML:

```
@startuml

abstract class Builder {
  + buildPart1()
  + buildPart2()
  + ...
  + buildPartN()
}

class ConcreteBuilder1 {
  + buildPart1()
  + buildPart2()
  + ...
  + buildPartN()
}

class ConcreteBuilder2 {
  + buildPart1()
  + buildPart2()
  + ...
  + buildPartN()
}

class Director {
  + Builder builder
  + construct(builder: Builder)
  + build()
}

Builder <|-- ConcreteBuilder1
Builder <|-- ConcreteBuilder2
Director o-- Builder

@enduml
```

Код абстрактной реализации паттерна "Строитель" на Python:

```python
from abc import ABC, abstractmethod

# Абстрактный класс Строитель
class Builder(ABC):
    @abstractmethod
    def build_part1(self):
        pass

    @abstractmethod
    def build_part2(self):
        pass

    # ...

    @abstractmethod
    def build_part_n(self):
        pass

# Конкретный класс Строитель1
class ConcreteBuilder1(Builder):
    def __init__(self):
        self.product = Product()

    def build_part1(self):
        # Конкретная реализация шага 1
        pass

    def build_part2(self):
        # Конкретная реализация шага 2
        pass

    # ...

    def build_part_n(self):
        # Конкретная реализация шага N
        pass

    def get_product(self):
        return self.product

# Конкретный класс Строитель2
class ConcreteBuilder2(Builder):
    def __init__(self):
        self.product = Product()

    def build_part1(self):
        # Конкретная реализация шага 1
        pass
```

```
    def build_part2(self):
        # Конкретная реализация шага 2
        pass

    # ...

    def build_part_n(self):
        # Конкретная реализация шага N
        pass

    def get_product(self):
        return self.product

# Класс Продукт
class Product:
    def __init__(self):
        self.parts = []

    def add_part(self, part):
        self.parts.append(part)

    def list_parts(self):
        return self.parts

# Класс Директор
class Director:
    def __init__(self):
        self.builder = None

    def set_builder(self, builder):
        self.builder = builder

    def build_product(self):
        self.builder.build_part1()
        self.builder.build_part2()
        # ...
        self.builder.build_part_n()
```

В этом примере паттерна "Строитель" абстрактный класс `Builder` определяет интерфейс для создания продукта, разделяя его на отдельные шаги. Конкретные классы строителей `ConcreteBuilder1` и `ConcreteBuilder2` предоставляют конкретные реализации для этих шагов, создавая различные варианты продукта. Класс `Director` управляет процессом создания продукта, вызывая конкретные методы, определенные в конкретном классе строителя. В итоге, мы получаем гибкую систему, которая позволяет создавать сложные объекты, разделяя процесс их создания на более мелкие подзадачи.

Код на Python, связанный с реальностью:

```python
from abc import ABC, abstractmethod

# Абстрактный класс Строитель
class HouseBuilder(ABC):
    @abstractmethod
    def build_foundation(self):
        pass

    @abstractmethod
    def build_walls(self):
        pass

    @abstractmethod
    def build_roof(self):
        pass

    def get_house(self):
        return self.house

# Конкретный класс Строитель1
class ConcreteHouseBuilder(HouseBuilder):
    def __init__(self):
        self.house = House()

    def build_foundation(self):
        self.house.add_part("Фундамент")

    def build_walls(self):
        self.house.add_part("Стены")

    def build_roof(self):
        self.house.add_part("Крыша")

# Класс Продукт (Дом)
class House:
    def __init__(self):
        self.parts = []

    def add_part(self, part):
        self.parts.append(part)

    def list_parts(self):
        return self.parts

# Класс Директор
class Director:
    def __init__(self):
        self.builder = None
```

```
    def set_builder(self, builder):
        self.builder = builder

    def build_house(self):
        self.builder.build_foundation()
        self.builder.build_walls()
        self.builder.build_roof()
        return self.builder.get_house()

# Создаем объект класса Директор
director = Director()

# Создаем объект класса Конкретный Строитель1
builder = ConcreteHouseBuilder()

# Устанавливаем объект класса Конкретный Строитель1 в качестве строителя
director.set_builder(builder)

# Строим дом
house = director.build_house()

# Выводим список частей дома
print(house.list_parts())
```

Этот код создает объект класса "Дом", разделяя процесс его создания на три подзадачи: закладка фундамента, возведение стен и установка крыши. Класс "Директор" управляет процессом создания объекта, вызывая методы класса "Строитель". В итоге, мы получаем готовый объект класса "Дом", не тратя лишних усилий и времени.

Фабричный метод

Фабричный метод (Factory Method) является одним из порождающих паттернов проектирования, который предоставляет интерфейс для создания объектов в суперклассе, позволяя подклассам решать, какой конкретный объект будет создан.

Вот пример принципиальной схемы паттерна фабричный метод на языке PlantUML:

```
@startuml

abstract class Product {
  - name: string
  + get_name(): string
}

class ConcreteProduct1 {
  + get_name(): string
}

class ConcreteProduct2 {
```

```
  + get_name(): string
}

abstract class Creator {
  + factory_method(): Product
  + some_operation(): string
}

class ConcreteCreator1 {
  + factory_method(): Product
}

class ConcreteCreator2 {
  + factory_method(): Product
}

Creator <|-- ConcreteCreator1
Creator <|-- ConcreteCreator2
Product <|-- ConcreteProduct1
Product <|-- ConcreteProduct2

@enduml
```

В этой схеме:

- `Product` - это абстрактный класс, который определяет интерфейс для объектов, которые создаются фабричным методом.
- `ConcreteProduct1` и `ConcreteProduct2` - это конкретные классы, которые реализуют интерфейс `Product`.
- `Creator` - это абстрактный класс, который содержит фабричный метод `factory_method`, который возвращает объект продукта, и метод `some_operation`, который использует фабричный метод для создания объекта продукта и выполнения некоторой операции с ним.
- `ConcreteCreator1` и `ConcreteCreator2` - это конкретные классы, которые реализуют интерфейс `Creator` и переопределяют фабричный метод для создания конкретных объектов продукта.

Стрелки с пустым ромбом указывают на то, что один класс реализует интерфейс или наследуется от другого класса.

В этом паттерне есть четыре основных компонента:

1. Продукт (Product): это объект, который создается фабричным методом.
2. Создатель (Creator): это абстрактный класс, который содержит фабричный метод, который возвращает объект продукта.
3. Конкретный создатель (Concrete Creator): это подкласс создателя, который переопределяет фабричный метод и возвращает конкретный объект продукта.

4. Клиент (Client): это объект, который использует фабричный метод для создания объектов продукта.

Вот пример абстрактной реализации паттерна фабричный метод на языке Python:

```
from abc import ABC, abstractmethod

# Продукт
class Product(ABC):
    @abstractmethod
    def operation(self):
        pass

# Создатель
class Creator(ABC):
    @abstractmethod
    def factory_method(self):
        pass

    def some_operation(self):
        product = self.factory_method()
        result = product.operation()
        return result

# Конкретный создатель 1
class ConcreteCreator1(Creator):
    def factory_method(self):
        return ConcreteProduct1()

# Конкретный создатель 2
class ConcreteCreator2(Creator):
    def factory_method(self):
        return ConcreteProduct2()

# Конкретный продукт 1
class ConcreteProduct1(Product):
    def operation(self):
        return "Result of ConcreteProduct1"

# Конкретный продукт 2
class ConcreteProduct2(Product):
    def operation(self):
        return "Result of ConcreteProduct2"

# Клиент
def client_code(creator: Creator) -> str:
    return creator.some_operation()

if __name__ == "__main__":
```

```
    print("Client: Testing client code with the first creator type:")
    creator1 = ConcreteCreator1()
    print(f"Creator1: {client_code(creator1)}")

    print("\n")

    print("Client: Testing the same client code with the second creator type:")
    creator2 = ConcreteCreator2()
    print(f"Creator2: {client_code(creator2)}")
```

В этом примере абстрактный класс `Product` определяет интерфейс для объектов, которые создаются фабричным методом. Абстрактный класс `Creator` содержит фабричный метод `factory_method`, который возвращает объект продукта, и метод `some_operation`, который использует фабричный метод для создания объекта продукта и выполнения некоторой операции с ним.

Конкретные классы `ConcreteCreator1` и `ConcreteCreator2` переопределяют фабричный метод и возвращают конкретные объекты продукта `ConcreteProduct1` и `ConcreteProduct2` соответственно.

Клиентский код `client_code` использует фабричный метод для создания объектов продукта и выполнения некоторой операции с ними. В этом примере клиентский код вызывает метод `some_operation` создателя, который в свою очередь использует фабричный метод для создания объекта продукта и выполнения операции с ним.

Вот пример реализации паттерна фабричный метод, связанного с реальностью, на языке Python:

```
from abc import ABC, abstractmethod

# Продукт
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

# Создатель
class AnimalFactory(ABC):
    @abstractmethod
    def create_animal(self):
        pass

    def get_animal_sound(self):
        animal = self.create_animal()
        return animal.make_sound()

# Конкретный создатель 1
class DogFactory(AnimalFactory):
    def create_animal(self):
        return Dog()
```

```python
# Конкретный создатель 2
class CatFactory(AnimalFactory):
    def create_animal(self):
        return Cat()

# Конкретный продукт 1
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Конкретный продукт 2
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Клиент
def client_code(factory: AnimalFactory) -> str:
    return factory.get_animal_sound()

if __name__ == "__main__":
    print("Client: Testing client code with the first factory type:")
    factory1 = DogFactory()
    print(f"Factory1: {client_code(factory1)}")

    print("\n")

    print("Client: Testing the same client code with the second factory type:")
    factory2 = CatFactory()
    print(f"Factory2: {client_code(factory2)}")
```

В этом примере абстрактный класс `Animal` определяет интерфейс для объектов, которые создаются фабричным методом. Абстрактный класс `AnimalFactory` содержит фабричный метод `create_animal`, который возвращает объект животного, и метод `get_animal_sound`, который использует фабричный метод для создания объекта животного и выполнения некоторой операции с ним.

Конкретные классы `DogFactory` и `CatFactory` переопределяют фабричный метод и возвращают конкретные объекты животных `Dog` и `Cat` соответственно.

Клиентский код `client_code` использует фабричный метод для создания объектов животных и выполнения некоторой операции с ними. В этом примере клиентский код вызывает метод `get_animal_sound` фабрики, который в свою очередь использует фабричный метод для создания объекта животного и выполнения операции с ним.

Одиночка

Одиночка (Singleton) - это порождающий шаблон проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Принципиальная схема шаблона Одиночка на языке PlantUML:

```
@startuml

class Singleton {
  - instance: Singleton
  + Singleton()
  + get_instance(): Singleton
}

@enduml
```

В этой схеме:

- `Singleton` - это класс, для которого нужно обеспечить единственный экземпляр.
- `- instance: Singleton` - это статическое приватное поле, которое хранит ссылку на единственный экземпляр класса `Singleton`.
- `+ Singleton()` - это конструктор класса `Singleton`, который должен быть объявлен как приватный, чтобы предотвратить создание экземпляров класса извне.
- `+ get_instance(): Singleton` - это статический метод, который обеспечивает глобальную точку доступа к единственному экземпляру класса `Singleton`.

Код абстрактной реализации паттерна Одиночка на 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, name):
        self.name = name

    def __str__(s):
        return f"Singleton object with name {s.name}"

if __name__ == "__main__":
    # Создаем два объекта Singleton
    s1 = Singleton("Object 1")
    s2 = Singleton("Object 2")

    # Сравниваем объекты
    print(f"s1 == s2: {s1 == s2}")

    # Выводим объекты на экран
```

```
    print(f"s1: {s1}")
    print(f"s2: {s2}")

    # Меняем имя у объекта s1
    s1.name = "New name"

    # Выводим объекты на экран еще раз
    print(f"s1: {s1}")
    print(f"s2: {s2}")
```

Этот код определяет базовый класс `Singleton`, который можно использовать в качестве родительского класса для любых классов, для которых нужно обеспечить единственный экземпляр.

Здесь используется магический метод `__new__`, который вызывается при создании нового экземпляра класса. В этом методе проверяется, существует ли уже экземпляр класса `Singleton`. Если экземпляр еще не создан, он создается с помощью вызова `super().__new__(cls, *args, **kwargs)`, который вызывает реализацию метода `__new__` из родительского класса. Если экземпляр уже существует, он возвращается из метода `__new__`.

В этом примере мы создаем два объекта Singleton с разными именами. Затем мы сравниваем эти объекты и выводим их на экран. Мы видим, что объекты равны между собой, и что они имеют одинаковые имена, которые были установлены при создании первого объекта.

Затем мы меняем имя у объекта s1 и снова выводим объекты на экран. Мы видим, что имя изменилось для обоих объектов, потому что они являются одним и тем же объектом.

Этот пример показывает, что паттерн Одиночка гарантирует, что у класса есть только один экземпляр, и что все объекты, которые ссылаются на этот экземпляр, являются одним и тем же объектом.

Вот пример реального использования паттерна Одиночка в Python:

Представим, что мы разрабатываем приложение для работы с базой данных. Нам нужно создать объект, который будет отвечать за подключение к базе данных и выполнение запросов. При этом нам нужно гарантировать, что у нас будет только один экземпляр этого объекта, потому что создание нескольких подключений к базе данных может привести к проблемам с производительностью и безопасностью.

Решение этой задачи с помощью паттерна Одиночка может выглядеть следующим образом:

```
import sqlite3

class Database:
    _instance = None

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

```
        return cls._instance

    def __init__(self):
        self.conn = sqlite3.connect("mydatabase.db")
        self.cursor = self.conn.cursor()

    def execute_query(self, query, params=None):
        self.cursor.execute(query, params or ())
        self.conn.commit()
        return self.cursor.fetchall()

    def __del__(self):
        self.conn.close()

if __name__ == "__main__":
    # Создаем два объекта Database
    db1 = Database()
    db2 = Database()

    # Сравниваем объекты
    print(f"db1 == db2: {db1 == db2}")

    # Выполняем запрос к базе данных с помощью объекта db1
    result = db1.execute_query("SELECT * FROM users WHERE id = ?", (1,))
    print(f"Result of db1 query: {result}")

    # Выполняем запрос к базе данных с помощью объекта db2
    result = db2.execute_query("SELECT * FROM users WHERE id = ?", (1,))
    print(f"Result of db2 query: {result}")
```

В этом примере мы создаем класс `Database`, который отвечает за подключение к базе данных и выполнение запросов. Мы реализуем паттерн Одиночка с помощью магического метода `__new__`, который гарантирует, что у нас будет только один экземпляр этого класса.

Затем мы создаем два объекта `Database` и сравниваем их. Мы видим, что объекты равны между собой, потому что они являются одним и тем же экземпляром класса.

Затем мы выполняем запрос к базе данных с помощью объекта `db1` и запрос к базе данных с помощью объекта `db2`. Мы видим, что результаты запросов одинаковы, потому что они выполняются с помощью одного и того же подключения к базе данных.

Этот пример показывает, как паттерн Одиночка может быть использован для решения реальных задач в разработке приложений. Он гарантирует, что у нас будет только один экземпляр объекта, который отвечает за важную часть нашего приложения, и что этот объект будет доступен во всех частях приложения.

Прототип

Шаблон проектирования "Прототип" (Prototype) предоставляет механизм копирования объектов, не

зависящий от их конкретных классов. Этот паттерн полезен в ситуациях, когда создание объекта затратно по времени или ресурсам, либо когда требуется создавать множество похожих объектов с небольшими отличиями.

Принципиальная диаграмма для шаблона проектирования "Прототип" на языке PlantUML:

```
@startuml

interface Prototype {
  + clone(): Prototype
}

class ConcretePrototype1 implements Prototype {
  + clone(): ConcretePrototype1
}

class ConcretePrototype2 implements Prototype {
  + clone(): ConcretePrototype2
}

class Client {
  - prototype: Prototype
  + setPrototype(p: Prototype): void
  + createObject(): Prototype
}

Prototype <|-- ConcretePrototype1
Prototype <|-- ConcretePrototype2
Client o-- Prototype

@enduml
```

Описание:

- `Prototype` - интерфейс, который определяет операцию `clone()` для создания копии объекта.
- `ConcretePrototype1` и `ConcretePrototype2` - конкретные реализации интерфейса `Prototype`, которые реализуют операцию `clone()` для создания копии объекта своего класса.
- `Client` - класс, который использует прототип для создания новых объектов. Он имеет ссылку на прототип, операцию `setPrototype()` для установки прототипа и операцию `createObject()` для создания нового объекта путем клонирования прототипа.

Вот пример реализации шаблона Прототип в Python:

```
from copy import deepcopy

class Prototype:
    def __init__(self, **attrs):
```

```
        self.__dict__.update(attrs)

    def clone(self):
        return deepcopy(self)

class ConcretePrototype1(Prototype):
    def __init__(self, **attrs):
        super().__init__(**attrs)
        self.attr1 = "ConcretePrototype1"

class ConcretePrototype2(Prototype):
    def __init__(self, **attrs):
        super().__init__(**attrs)
        self.attr2 = "ConcretePrototype2"

if __name__ == "__main__":
    # Создаем объекты-прототипы
    p1 = ConcretePrototype1(attr3="value3")
    p2 = ConcretePrototype2(attr4="value4")

    # Клонируем объекты-прототипы
    c1 = p1.clone()
    c2 = p2.clone()

    # Проверяем, что объекты-прототипы и их клоны разные
    print(f"p1 is c1: {p1 is c1}")
    print(f"p2 is c2: {p2 is c2}")

    # Проверяем, что объекты-прототипы и их клоны имеют одинаковые значения атрибутов
    print(f"p1.__dict__ == c1.__dict__: {p1.__dict__ == c1.__dict__}")
    print(f"p2.__dict__ == c2.__dict__: {p2.__dict__ == c2.__dict__}")

    # Проверяем, что клоны имеют те же типы, что и объекты-прототипы
    print(f"type(c1) == type(p1): {type(c1) == type(p1)}")
    print(f"type(c2) == type(p2): {type(c2) == type(p2)}")
```

В этом примере мы создаем абстрактный класс `Prototype`, который предоставляет метод `clone()` для копирования объектов. Затем мы создаем два конкретных класса-прототипа `ConcretePrototype1` и `ConcretePrototype2`, которые наследуются от `Prototype` и добавляют свои собственные атрибуты.

В основной части программы мы создаем объекты-прототипы `p1` и `p2`, затем клонируем их с помощью метода `clone()`. Мы проверяем, что объекты-прототипы и их клоны разные, но имеют одинаковые значения атрибутов и типы.

Пример реального использования шаблона Прототип:

Представим, что мы разрабатываем игру, в которой есть множество различных типов врагов. Каждый тип врага имеет свои собственные характеристики, такие как здоровье, скорость и урон.

Создание каждого врага вручную может быть затратно по времени и ресурсам, поэтому мы решаем использовать шаблон Прототип.

Вот пример реализации шаблона Прототип для этой задачи:

```
from copy import deepcopy

class EnemyPrototype:
    def __init__(self, **attrs):
        self.__dict__.update(attrs)

    def clone(self):
        return deepcopy(self)

class ConcreteEnemy1(EnemyPrototype):
    def __init__(self, **attrs):
        super().__init__(**attrs)
        self.name = "ConcreteEnemy1"
        self.health = 100
        self.speed = 5
        self.damage = 20

class ConcreteEnemy2(EnemyPrototype):
    def __init__(self, **attrs):
        super().__init__(**attrs)
        self.name = "ConcreteEnemy2"
        self.health = 200
        self.speed = 3
        self.damage = 30

class EnemyFactory:
    def __init__(self):
        self.prototypes = {
            "ConcreteEnemy1": ConcreteEnemy1(),
            "ConcreteEnemy2": ConcreteEnemy2(),
        }

    def create_enemy(self, name, **attrs):
        prototype = self.prototypes[name]
        enemy = prototype.clone()
        enemy.__dict__.update(attrs)
        return enemy

if __name__ == "__main__":
    # Создаем фабрику врагов
    factory = EnemyFactory()

    # Создаем врагов с помощью фабрики
    e1 = factory.create_enemy("ConcreteEnemy1", position=(10, 20))
```

```
    e2 = factory.create_enemy("ConcreteEnemy2", position=(30, 40))

    # Проверяем, что враги имеют ожидаемые значения атрибутов
    print(f"e1.__dict__: {e1.__dict__}")
    print(f"e2.__dict__: {e2.__dict__}")
```

В этом примере мы создаем абстрактный класс `EnemyPrototype`, который предоставляет метод `clone()` для копирования объектов-врагов. Затем мы создаем два конкретных класса-прототипа `ConcreteEnemy1` и `ConcreteEnemy2`, которые наследуются от `EnemyPrototype` и добавляют свои собственные характеристики.

В основной части программы мы создаем фабрику врагов `EnemyFactory`, которая хранит словарь прототипов врагов. Затем мы создаем врагов с помощью фабрики, передавая ей имя прототипа и дополнительные атрибуты, такие как позиция на карте. Мы проверяем, что враги имеют ожидаемые значения атрибутов.

Этот пример показывает, как шаблон Прототип может быть использован для создания множества похожих объектов с небольшими отличиями, не затратно по времени и ресурсам.

Абстрактная фабрика
Шаблон проектирования "Абстрактная фабрика" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Этот паттерн полезен в ситуациях, когда требуется создавать объекты, которые могут меняться в зависимости от конфигурации, окружения или других факторов, и при этом сохранять взаимосвязь между объектами.

Принципиальная диаграмма для шаблона проектирования "Абстрактная фабрика" на языке PlantUML:

```
@startuml

interface AbstractProductA {
  + operation(): void
}

interface AbstractProductB {
  + operation(): void
}

class ConcreteProductA1 implements AbstractProductA {
  + operation(): void
}

class ConcreteProductA2 implements AbstractProductA {
  + operation(): void
}

class ConcreteProductB1 implements AbstractProductB {
```

```
  + operation(): void
}

class ConcreteProductB2 implements AbstractProductB {
  + operation(): void
}

interface AbstractFactory {
  + createProductA(): AbstractProductA
  + createProductB(): AbstractProductB
}

class ConcreteFactory1 implements AbstractFactory {
  + createProductA(): ConcreteProductA1
  + createProductB(): ConcreteProductB1
}

class ConcreteFactory2 implements AbstractFactory {
  + createProductA(): ConcreteProductA2
  + createProductB(): ConcreteProductB2
}

Client o-- AbstractFactory
AbstractFactory <|-- ConcreteFactory1
AbstractFactory <|-- ConcreteFactory2
AbstractProductA <|-- ConcreteProductA1
AbstractProductA <|-- ConcreteProductA2
AbstractProductB <|-- ConcreteProductB1
AbstractProductB <|-- ConcreteProductB2

@enduml
```

Описание:

- `AbstractProductA` и `AbstractProductB` - интерфейсы, которые определяют операции для конкретных продуктов.
- `ConcreteProductA1`, `ConcreteProductA2`, `ConcreteProductB1` и `ConcreteProductB2` - конкретные реализации интерфейсов `AbstractProductA` и `AbstractProductB`.
- `AbstractFactory` - интерфейс, который определяет операции для создания конкретных продуктов.
- `ConcreteFactory1` и `ConcreteFactory2` - конкретные реализации интерфейса `AbstractFactory`, которые создают конкретные продукты, соответствующие данной фабрике.
- `Client` - класс, который использует интерфейс `AbstractFactory` для создания конкретных продуктов, не зная о конкретных классах продуктов и фабрик.

Вот пример реализации шаблона Абстрактная фабрика в Python:

```python
from abc import ABC, abstractmethod

class Button(ABC):
    @abstractmethod
    def render(self):
        pass

    @abstractmethod
    def onClick(self):
        pass

class WindowsButton(Button):
    def render(self):
        print("Rendering Windows button")

    def onClick(self):
        print("Clicked Windows button")

class MacButton(Button):
    def render(self):
        print("Rendering Mac button")

    def onClick(self):
        print("Clicked Mac button")

class GUIFactory(ABC):
    @abstractmethod
    def createButton(self):
        pass

class WindowsFactory(GUIFactory):
    def createButton(self):
        return WindowsButton()

class MacFactory(GUIFactory):
    def createButton(self):
        return MacButton()

if __name__ == "__main__":
    # Создаем фабрики для разных операционных систем
    windows_factory = WindowsFactory()
    mac_factory = MacFactory()

    # Создаем объекты с помощью фабрик
    windows_button = windows_factory.createButton()
    mac_button = mac_factory.createButton()

    # Проверяем, что объекты имеют ожидаемые типы
    print(f"type(windows_button) == WindowsButton: {type(windows_button) ==
```

```
WindowsButton}")
    print(f"type(mac_button) == MacButton: {type(mac_button) == MacButton}")

    # Проверяем, что объекты работают корректно
    windows_button.render()
    windows_button.onClick()
    mac_button.render()
    mac_button.onClick()
```

В этом примере мы создаем абстрактный класс `Button`, который предоставляет интерфейс для рендеринга и обработки нажатия кнопки. Затем мы создаем два конкретных класса-реализации `WindowsButton` и `MacButton`, которые реализуют этот интерфейс для разных операционных систем.

В основной части программы мы создаем абстрактный класс `GUIFactory`, который предоставляет интерфейс для создания объектов-компонентов графического интерфейса. Затем мы создаем два конкретных класса-реализации `WindowsFactory` и `MacFactory`, которые реализуют этот интерфейс для разных операционных систем.

Затем мы создаем фабрики для разных операционных систем, а затем с их помощью создаем объекты-компоненты графического интерфейса. Мы проверяем, что объекты имеют ожидаемые типы, и что они работают корректно.

Пример реального использования шаблона Абстрактная фабрика:

Представим, что мы разрабатываем приложение, которое должно работать на разных платформах (например, Windows, macOS, Linux). Приложение должно иметь единый графический интерфейс, но при этом использовать нативные компоненты для каждой платформы.

Вот пример реализации шаблона Абстрактная фабрика для этой задачи:

```
from abc import ABC, abstractmethod

class FileDialog(ABC):
    @abstractmethod
    def open(self):
        pass

    @abstractmethod
    def close(self):
        pass

    @abstractmethod
    def getSelectedFile(self):
        pass

class WindowsFileDialog(FileDialog):
    def open(self):
        print("Opening Windows file dialog")
```

```
    def close(self):
        print("Closing Windows file dialog")

    def getSelectedFile(self):
        print("Getting selected file from Windows file dialog")
        return "C:\\file.txt"

class MacFileDialog(FileDialog):
    def open(self):
        print("Opening Mac file dialog")

    class MacFileDialog(FileDialog):
    def open(self):
        print("Opening Mac file dialog")

    def close(self):
        print("Closing Mac file dialog")

    def getSelectedFile(self):
        print("Getting selected file from Mac file dialog")
        return "/Users/user/file.txt"

class GUIFactory(ABC):
    @abstractmethod
    def createFileDialog(self):
        pass

class WindowsFactory(GUIFactory):
    def createFileDialog(self):
        return WindowsFileDialog()

class MacFactory(GUIFactory):
    def createFileDialog(self):
        return MacFileDialog()

if __name__ == "__main__":
    # Определяем текущую операционную систему
    import sys
    if sys.platform.startswith("win"):
        factory = WindowsFactory()
    elif sys.platform.startswith("darwin"):
        factory = MacFactory()
    else:
        print("Unsupported platform")
        sys.exit(1)

    # Создаем объекты с помощью фабрики
    file_dialog = factory.createFileDialog()
```

```
    # Проверяем, что объекты имеют ожидаемые типы
    print(f"type(file_dialog) == WindowsFileDialog: {type(file_dialog) ==
WindowsFileDialog}")
    print(f"type(file_dialog) == MacFileDialog: {type(file_dialog) == MacFileDialog}")

    # Проверяем, что объекты работают корректно
    file_dialog.open()
    selected_file = file_dialog.getSelectedFile()
    file_dialog.close()
    print(f"Selected file: {selected_file}")
```

В этом примере мы создаем абстрактный класс `FileDialog`, который предоставляет интерфейс для открытия, закрытия и получения выбранного файла из диалога выбора файла. Затем мы создаем два конкретных класса-реализации `WindowsFileDialog` и `MacFileDialog`, которые реализуют этот интерфейс для разных операционных систем.

В основной части программы мы создаем абстрактный класс `GUIFactory`, который предоставляет интерфейс для создания объектов-компонентов графического интерфейса. Затем мы создаем два конкретных класса-реализации `WindowsFactory` и `MacFactory`, которые реализуют этот интерфейс для разных операционных систем.

Затем мы определяем текущую операционную систему, создаем фабрику для этой системы, а затем с ее помощью создаем объект-компонент графического интерфейса. Мы проверяем, что объект имеет ожидаемый тип, и что он работает корректно.

Этот пример показывает, как шаблон Абстрактная фабрика может быть использован для создания семейств взаимосвязанных объектов, не специфицируя их конкретных классов, и при этом сохранять взаимосвязь между объектами.