

Шаблоны проектирования



Глуценко

Содержание

□ Шаблоны

- Прокси
- Адаптер
- Мост

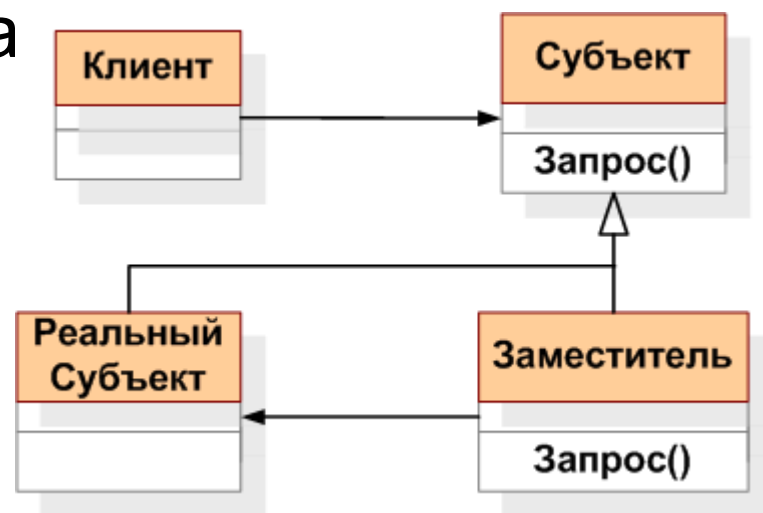
□ Примеры в Python

Прокси

- Структурный шаблон
- Проблема
 - Максимально отсрочить создание объекта
 - Для этого создается «кто-то» кто будет следить за тем, действительно ли необходимо создать объект на данном этапе или дать к нему доступ

Прокси. UML

- ❑ Создать «Заместитель» - суррогат громоздкого объекта ,хранит ссылку на реальный субъект
- ❑ Объект «Заместитель» может обращаться к объекту «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы)
- ❑ так, что «Заместителя» можно подставить везде вместо «Реального Субъекта»



Прокси. Пример

- Пусть сложной личностью будет Продюсер – и в какой-то момент может существовать их ограниченное число
- В такой «задаче» в роли прокси выступает Секретарь – он будет проверять доступен ли Продюсер
- Для нашего Клиента

Прокси. Решение

- В данной модели Клиент большую часть времени «общается» с Секретарем, пока не возникает необходимость в Продюсере
 - Поэтому именно Секретарь будет ответственный за создание ресурсоемкого Продюсера
-
- Сродненные шаблоны:
 - Adapter - обеспечивает отличающийся интерфейс к объекту
 - Decorator - обеспечивает расширенный интерфейс
 - Proxu - обеспечивает тот же самый интерфейс

```
class Producer:
    def produce(self):
        print("Producer is working hard!")

    def meet(self):
        print("You have 2 seconds!")

class Proxy:
    def __init__(self):
        self.occupied = 'No'
        self.producer = None

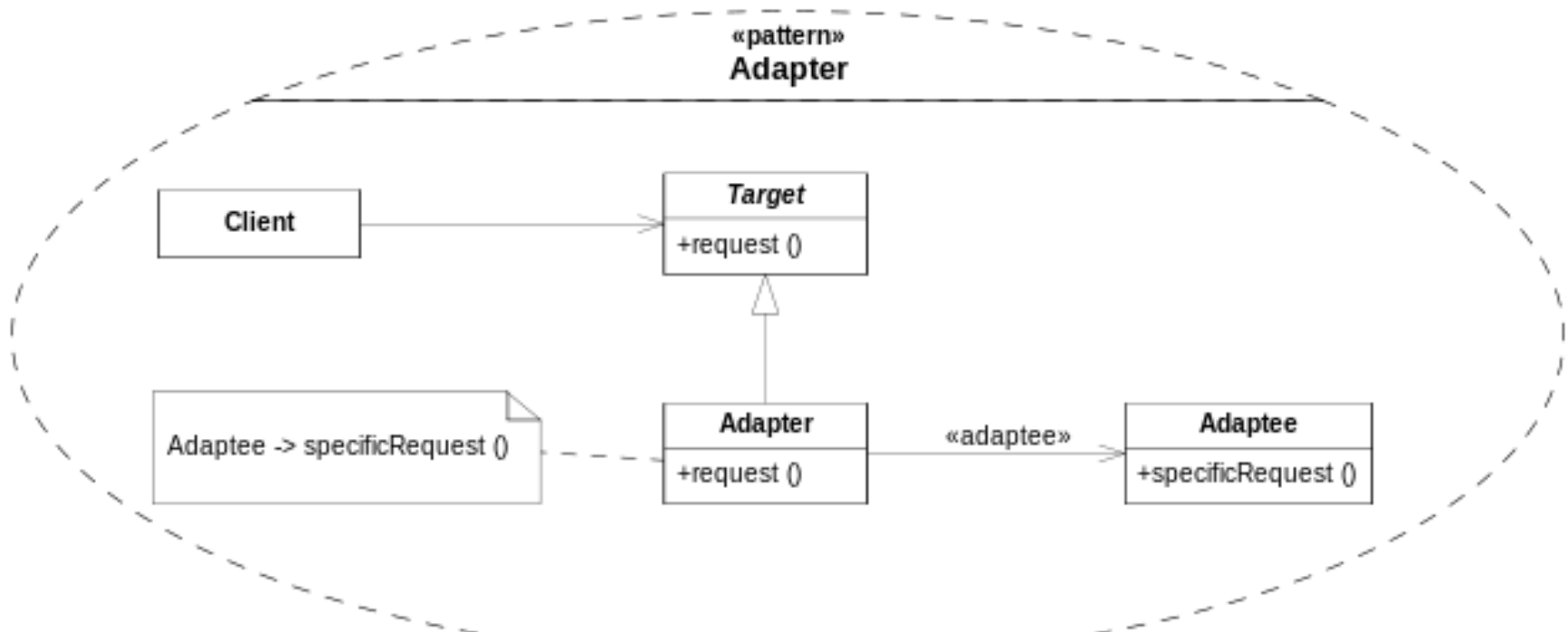
    def produce(self):
        print("Secretary checking if Producer is available ...")
        if self.occupied == 'No':
            print("He will take you")
            self.occupied = 'Yes'
            self.producer = Producer()
            self.producer.meet()
            time.sleep(2)
        else:
            #Otherwise, don't instantiate a producer
            print("Producer is busy! What can I do?")
            time.sleep(2)

    def meet(self):
        print("Secretary is working")
        time.sleep(2)

p = Proxy() #Instantiate a Proxy
p.produce() #Secretary produce until Producer is available
p.occupied = 'Yes' #Change the state to 'occupied'
p.produce()
```

Адаптер

- Структурный шаблон
- Изменяет интерфейс сервера на тот, который ожидает клиент
- Фактически это класс-оболочка



Адаптер. Пример

- У нас есть два объекта с разными методами:
один говорит по-корейски(speak_korean())
другой по-английски(speak_english())
- Но клиент не хочет разбираться, он хочет пользоваться методом speak() И быть довольным
- Сродненные шаблоны:
 - Bridge
 - Decorator – подключает дополнительные обязанности к объекту

```

class Korean:
    def __init__(self):
        self.name = "Korean"

    def speak_korean(self):
        return "An-neyong?"

class British:
    def __init__(self):
        self.name = "British"
        #Note the different method name here!

    def speak_english(self):
        return "Hello!"

class Adapter:
    def __init__(self, object, **adapted_method):
        self._object = object #Change the name of THIS object
        #Add a new dictionary item that establishes the mapping
        #between the generic method name: speak() and the concrete method
        self.__dict__.update(adapted_method)

    def __getattr__(self, attr):
        """Simply return the rest of attributes!"""
        return getattr(self._object, attr)

objects = []
korean = Korean()
british = British()
objects.append(Adapter(korean, speak=korean.speak_korean)) #key=value
objects.append(Adapter(british, speak=british.speak_english))
for obj in objects:
    print("{} says '{}' \n".format(obj.name, obj.speak()))

```

Мост

- Структурный шаблон проектирования
- Проблема:
 - Отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо.
 - При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию.
- Сродненные шаблоны:
 - Абстрактная фабрика
 - Адаптер
 - Стратегия

Мост

□ Зачем?

- Групповая разработка
- Любые изменения в интерфейсе абстракции тут же должны быть имплементированы в реализации.

□ Когда?

- Если использовать чисто абстракцию, придется реализовывать N разновидностей * M реализаций

□ Пример: геометрические фигуры

разновидность: круг, треугольник, квадрат

реализация: рисовать себя в Windows, OS, Linux

```

class DrawingAPI:# Implementor
    def drawCircle(self, x, y, radius):
        pass

class DrawingAPI1(DrawingAPI):# ConcreteImplementor 1/2
    def drawCircle(self, x, y, radius):
        print ("API1.circle at %f:%f radius %f" % (x, y, radius))

class DrawingAPI2(DrawingAPI):# ConcreteImplementor 2/2
    def drawCircle(self, x, y, radius):
        print ("API2.circle at %f:%f radius %f" % (x, y, radius))

class Shape:# Abstraction
    def draw(self):# Low-level
        pass
    def resizeByPercentage(self, pct):# High-level
        pass

class CircleShape(Shape):# Refined Abstraction
    def __init__(self, x, y, radius, drawingAPI):
        self.__x = x
        self.__y = y
        self.__radius = radius
        self.__drawingAPI = drawingAPI

    def draw(self):# low-level i.e. Implementation specific
        self.__drawingAPI.drawCircle(self.__x, self.__y, self.__radius)
    def resizeByPercentage(self, pct):# high-level i.e. Abstraction specific
        self.__radius *= pct

shapes = [CircleShape(1, 2, 3, DrawingAPI1()),
          CircleShape(5, 7, 11, DrawingAPI2())]
for shape in shapes:
    shape.resizeByPercentage(2.5)
    shape.draw()

```