|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |
| Институт кибербезопасности и цифровых технологий |
| Кафедра КБ-2 «Прикладные информационные технологии» |

**ПРАКТИЧЕСКАЯ РАБОТА №6 по дисциплине**

**«Технологии и методы программирования»**

***(наименование дисциплины)***

**Тема практической работы   
«Заместитель, Компоновщик, Инверсия управления»**

Студент группы: БИСО-01-20 Дутов А.С.

*(шифр учебной группы) (Фамилия И.О)*

Преподаватель: Лесько С.А.

*(Фамилия И.О)*

Москва, 2023

**Практическая работа №6**

**Заместитель**

class Subject:

def request(self):

pass

class RealSubject(Subject):

def request(self):

print("RealSubject: Handling request.")

class Proxy(Subject):

def \_\_init\_\_(self, real\_subject):

self.real\_subject = real\_subject

def request(self):

if self.check\_access():

self.real\_subject.request()

self.log\_access()

def check\_access(self):

print("Proxy: Checking access prior to firing a real request.")

return True

def log\_access(self):

print("Proxy: Logging the time of request.")

# Пример использования

real\_subject = RealSubject()

proxy = Proxy(real\_subject)

proxy.request()

В этом примере мы создали класс Subject, определяющий интерфейс, который должен быть реализован. Мы также создали класс RealSubject, который представляет реальный объект, реализующий этот интерфейс.

Затем мы создали класс Proxy, который наследуется от класса Subject, принимает объект RealSubject в качестве параметра в конструкторе, и реализует метод request(), который проверяет доступ перед выполнением запроса к реальному объекту.

Мы также реализовали методы check\_access() и log\_access(), которые проверяют доступ и логируют время выполнения запроса, соответственно.

В конце мы создали объект RealSubject, создали объект Proxy, передавая в него объект RealSubject, и вызвали метод request() для объекта Proxy.

**Компоновщик**

class Component:

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

self.name = name

def operation(self):

pass

class Leaf(Component):

def operation(self):

print(f"Leaf {self.name}: Handling operation.")

class Composite(Component):

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

super().\_\_init\_\_(name)

self.children = []

def add(self, component):

self.children.append(component)

def remove(self, component):

self.children.remove(component)

def operation(self):

print(f"Composite {self.name}: Handling operation.")

for child in self.children:

child.operation()

# Пример использования

leaf1 = Leaf("1")

leaf2 = Leaf("2")

leaf3 = Leaf("3")

composite1 = Composite("A")

composite1.add(leaf1)

composite1.add(leaf2)

composite2 = Composite("B")

composite2.add(leaf3)

composite1.add(composite2)

composite1.operation()

В этом примере мы создали класс Component, который определяет интерфейс для всех компонентов в дереве компоновщика. Мы также создали два класса-наследника: Leaf и Composite. Лист представляет конечный элемент дерева, а составной объект содержит другие компоненты.

Затем мы создали класс Composite, который содержит в себе список дочерних компонентов, и реализует методы добавления, удаления и обработки компонентов.

В нашем примере мы создали три листа leaf1, leaf2 и leaf3, а также две композитные группы: composite1 и composite2. Мы добавили leaf1 и leaf2 в группу composite1, а leaf3 в группу composite2. Затем мы добавили composite2 в composite1.

Наконец, мы вызвали метод operation() для объекта composite1. Внутри этого метода компоненты обрабатываются рекурсивно, пока все элементы дерева не будут обработаны.

**Инверсия управления**

class Service:

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

self.name = "Service"

def execute(self):

print("Executing Service")

class Client:

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

self.service = service

def do\_something(self):

print(f"Client calls {self.service.name}")

self.service.execute()

# Пример использования

service = Service()

client = Client(service)

client.do\_something()

Важным моментом здесь является то, что объект Client знает только об интерфейсе объекта Service, а не о его конкретной реализации. Это позволяет заменять объект Service на другой объект, который реализует тот же интерфейс, без изменения класса Client.

Таким образом, мы добиваемся инверсии управления. Класс Client не создает объект Service самостоятельно, а получает его извне. Это позволяет лучшей расширяемости и гибкости кода.