|  |
| --- |
| 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 №5 по дисциплине**

**«Технологии и методы программирования»**

***(наименование дисциплины)***

**Тема практической работы   
«Абстрактная фабрика. Посредник. Строитель. Адаптер»**

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 №5**

**Абстрактная фабрика**

# Абстрактные классы, определяющие интерфейсы

class Button:

def paint(self):

pass

class Checkbox:

def paint(self):

pass

# Конкретные классы, реализующие интерфейсы

class WinButton(Button):

def paint(self):

print("Painting a Windows Button")

class LinuxButton(Button):

def paint(self):

print("Painting a Linux Button")

class WinCheckbox(Checkbox):

def paint(self):

print("Painting a Windows Checkbox")

class LinuxCheckbox(Checkbox):

def paint(self):

print("Painting a Linux Checkbox")

# Абстрактная фабрика

class GUIFactory:

def create\_button(self):

pass

def create\_checkbox(self):

pass

# Конкретные фабрики, наследуемые от абстрактной фабрики

class WinFactory(GUIFactory):

def create\_button(self):

return WinButton()

def create\_checkbox(self):

return WinCheckbox()

class LinuxFactory(GUIFactory):

def create\_button(self):

return LinuxButton()

def create\_checkbox(self):

return LinuxCheckbox()

# Пример использования

class Application:

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

self.factory = factory

def create\_interface(self):

button = self.factory.create\_button()

checkbox = self.factory.create\_checkbox()

button.paint()

checkbox.paint()

win\_app = Application(WinFactory())

win\_app.create\_interface()

linux\_app = Application(LinuxFactory())

linux\_app.create\_interface()

В этом примере мы создали два абстрактных класса, Button и Checkbox, которые определяют метод paint(). Мы также создали конкретные классы, WinButton, LinuxButton, WinCheckbox и LinuxCheckbox, реализующие эти интерфейсы. Далее мы создали абстрактную фабрику GUIFactory, определяющую методы create\_button() и create\_checkbox(). Эти методы возвращают объекты конкретных классов, которые соответствуют текущей платформе. Затем мы создали две конкретные фабрики, WinFactory и LinuxFactory, которые наследуются от абстрактной фабрики GUIFactory. Каждая из этих фабрик реализует методы create\_button() и create\_checkbox(), возвращая соответствующие объекты конкретных классов для Windows и Linux. Наконец, мы создали класс Application, который принимает объект абстрактной фабрики в качестве параметра и использует ее для создания пользовательского интерфейса с помощью методов create\_button() и create\_checkbox(). В конце мы создали два объекта Application, один для Windows и один для Linux, используя соответствующие фабрики, создали пользовательский интерфейс и вызвали метод paint() для каждого из созданных элементов интерфейса.

**Посредник**

class Mediator:

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

self.components = []

def add\_component(self, component):

self.components.append(component)

def notify(self, event, source):

for component in self.components:

if component != source:

component.handle\_notification(event)

class Component:

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

self.mediator = mediator

self.mediator.add\_component(self)

def trigger\_event(self, event):

self.mediator.notify(event, self)

def handle\_notification(self, event):

print(f"Component received event: {event}")

# Пример использования

mediator = Mediator()

component1 = Component(mediator)

component2 = Component(mediator)

component1.trigger\_event("Some event")

В этом примере мы создали класс Mediator, который хранит список компонентов. Мы также создали класс Component, который имеет ссылку на посредника и методы для отправки и обработки событий. Для отправки событий компоненты вызывают метод trigger\_event(), передавая в качестве параметра текстовое описание события. Посредник получает все события и оповещает всех компонентов, кроме отправителя, вызывая метод handle\_notification() для каждого из них. В конце мы создали посредника и два компонента, и отправили событие от компонента 1. Компонент 2 обрабатывает это событие, вызывая метод handle\_notification().

**Строитель**

class Product:

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

self.name = name

self.price = price

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

return f"{self.name} - ${self.price}"

class ProductBuilder:

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

self.name = None

self.price = None

def set\_name(self, name):

self.name = name

def set\_price(self, price):

self.price = price

def build(self):

return Product(self.name, self.price)

# Пример использования

builder = ProductBuilder()

builder.set\_name("Product 1")

builder.set\_price(10.50)

product = builder.build()

print(product)

В этом примере мы создали класс Product, который имеет два свойства: name и price. Мы также создали класс ProductBuilder, который имеет методы для установки каждого из свойств продукта и метод build(), который создает и возвращает объект Product.

В конце мы создаем объект ProductBuilder, устанавливаем свойства продукта с помощью методов set\_name() и set\_price(), и вызываем метод build() для создания объекта Product, который затем выводим на экран.

**Адаптер**

class Target:

def request(self):

print("Target: The default target's behavior.")

class Adaptee:

def specific\_request(self):

print("Adaptee: The specific request's behavior.")

class Adapter(Target):

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

self.adaptee = adaptee

def request(self):

self.adaptee.specific\_request()

# Пример использования

adaptee = Adaptee()

adapter = Adapter(adaptee)

adapter.request()

В этом примере мы создали класс Target, который определяет интерфейс, который должен реализовать клиент. Мы также создали класс Adaptee, который имеет определенный специфический метод.

Затем мы создали класс Adapter, который наследуется от класса Target, принимает объект Adaptee в качестве параметра в конструкторе, и реализует метод request(), который вызывает специфический метод объекта Adaptee.

В конце мы создаем объект Adaptee, создаем объект Adapter, передавая в него объект Adaptee, и вызываем метод request() для объекта Adapter.