**Задание №1 гит** <https://github.com/Aleksey4ik6/Alekseyka>

1.Постановка задачи.

Представим, что вы программист автоматической системы создания

пицц в пиццерии. Вам поручили разработать настройщик пицц, который

позволит выбирать пользователю пиццу по своему вкусу, выбирая различные

добавки и начинки. Для этого вы используете паттерн «Строитель».

Дополнительное задание:

Данный заготовщик пицц даёт выбрать пользователю только один соус

и только один сыр. Расширьте приложение, добавив возможность выбирать

несколько соусов и видов сыра. Создайте пиццу «4 сыра».

2.Листинг программы:

from enum import Enum

# Класс пиццы

class Pizza:

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

self.name = None # название пиццы

self.dough = None # тесто

self.sauces = [] # соусы

self.cheeses = [] # сыры

self.toppings = [] # добавки

# Метод для строкового представления объекта

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

sauces\_str = ", ".join(sauce.value for sauce in self.sauces)

cheeses\_str = ", ".join(cheese.value for cheese in self.cheeses)

toppings\_str = ", ".join(topping.value for topping in self.toppings)

return (

f"Пицца {self.name}: "

f"(тесто={self.dough.value}, соусы=[{sauces\_str}], сыры=[{cheeses\_str}], добавки=[{toppings\_str}])"

)

# Перечисления

class Dough(Enum):

THIN = "тонкое"

THICK = "толстое"

GLUTEN\_FREE = "безглютеновое"

class Sauce(Enum):

TOMATO = "томатный"

CREAM = "сливочный"

PESTO = "песто"

GARLIC = "чесночный"

class Cheese(Enum):

MOZZARELLA = "моцарелла"

PARMESAN = "пармезан"

CHEDDAR = "чеддер"

GORGONZOLA = "горгонзола"

RICOTTA = "рикотта"

class Topping(Enum):

MUSHROOMS = "грибы"

PEPPERONI = "пепперони"

OLIVES = "оливки"

BACON = "бекон"

PINEAPPLE = "ананасы"

ARUGULA = "руккола"

# Класс постройщика пицц

class PizzaBuilder:

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

self.pizza = Pizza() # пицца для настройки

def set\_name(self, name): # установить название пиццы

self.pizza.name = name

return self

def build\_dough(self, dough): # выбрать тесто

self.pizza.dough = dough

return self

def build\_sauces(self, sauces): # выбрать соусы

self.pizza.sauces.extend(sauces)

return self

def build\_cheeses(self, cheeses): # выбрать сыры

self.pizza.cheeses.extend(cheeses)

return self

def build\_toppings(self, toppings): # прибавить добавки

self.pizza.toppings.extend(toppings)

return self

def bake(self): # запечь пиццу (вернуть объект пиццы)

return self.pizza

# Создаем пиццу "4 сыра"

builder = PizzaBuilder()

four\_cheese\_pizza = (

builder.set\_name("4 сыра")

.build\_dough(Dough.THIN)

.build\_sauces([Sauce.TOMATO, Sauce.CREAM])

.build\_cheeses([Cheese.MOZZARELLA, Cheese.PARMESAN, Cheese.CHEDDAR, Cheese.GORGONZOLA])

.build\_toppings([Topping.OLIVES, Topping.ARUGULA])

.bake()

)

# Создаем другую пиццу с уникальными ингредиентами

custom\_pizza = (

builder.set\_name("Гурманская")

.build\_dough(Dough.GLUTEN\_FREE)

.build\_sauces([Sauce.PESTO, Sauce.GARLIC])

.build\_cheeses([Cheese.RICOTTA, Cheese.GORGONZOLA])

.build\_toppings([Topping.BACON, Topping.PINEAPPLE, Topping.ARUGULA])

.bake()

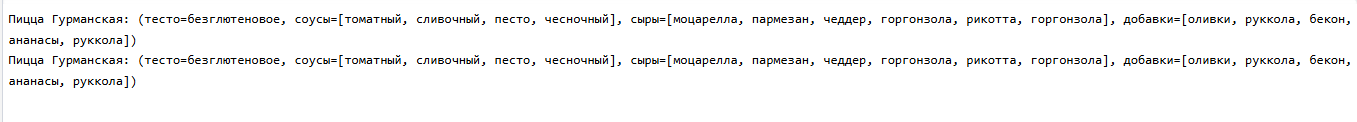
)

# Выводим созданные пиццы

print(four\_cheese\_pizza)

print(custom\_pizza)

3.Результаты



**Задание №2**

1.Постановка задачи.

Вы – программист в туристическом агентстве. Ваша задача

спланировать туристу путешествие, используя разные виды транспорта.

Пользователь должен указывать название желаемого транспорта и пункт

назначения. На данный момент туристическое агентство может предложить

только поездку на автобусе и рейс на самолёт. Для удобной смены вида

транспорта в реальном времени – вы используете стратегии, которые

представлены в виде транспорта.

2.Листинг программы:

from abc import ABC, abstractmethod

# Абстрактный базовый класс всех стратегий

class TravelStrategy(ABC):

@abstractmethod

def travel(self, transportName, destination):

pass # Абстрактный метод отправки в путешествие

# Путешествие на автобусе

class BusTravel(TravelStrategy):

def travel(self, transportName, destination):

print(f"Еду на автобусе: {transportName} до {destination}")

# Путешествие на самолете

class AirplaneTravel(TravelStrategy):

def travel(self, transportName, destination):

print(f"Лечу на самолете: {transportName} до {destination}")

# Путешествие на поезде

class TrainTravel(TravelStrategy):

def travel(self, transportName, destination):

print(f"Еду на поезде: {transportName} до {destination}")

# Путешествие на корабле

class ShipTravel(TravelStrategy):

def travel(self, transportName, destination):

print(f"Плыву на корабле: {transportName} до {destination}")

# Туристическое агентство

class TravelAgency:

def \_\_init\_\_(self, strategy: TravelStrategy):

self.strategy = strategy # Задаём начальную стратегию (транспорт)

def set\_strategy(self, strategy: TravelStrategy):

self.strategy = strategy # Задаём новый транспорт

def plan\_trip(self, transportName, destination):

self.strategy.travel(transportName, destination)

# Функция для выбора стратегии

def choose\_transport():

print("\nДоступные виды транспорта:")

print("1 - Автобус")

print("2 - Самолёт")

print("3 - Поезд")

print("4 - Корабль")

choice = int(input("Выберите номер транспорта: "))

if choice == 1:

return BusTravel()

elif choice == 2:

return AirplaneTravel()

elif choice == 3:

return TrainTravel()

elif choice == 4:

return ShipTravel()

else:

print("Неверный выбор. Попробуйте снова.")

return choose\_transport()

# Основная программа

def main():

agency = TravelAgency(None) # Создаём агентство без начальной стратегии

while True:

# Пользователь выбирает вид транспорта

strategy = choose\_transport()

agency.set\_strategy(strategy)

# Пользователь указывает транспортное средство и пункт назначения

transport\_name = input("Введите название транспорта: ")

destination = input("Введите пункт назначения: ")

# Планируем поездку

agency.plan\_trip(transport\_name, destination)

# Проверяем, хочет ли пользователь продолжить

cont = input("Запланировать ещё одну поездку? (да/нет): ").strip().lower()

if cont != 'да':

print("Спасибо за использование туристического агентства!")

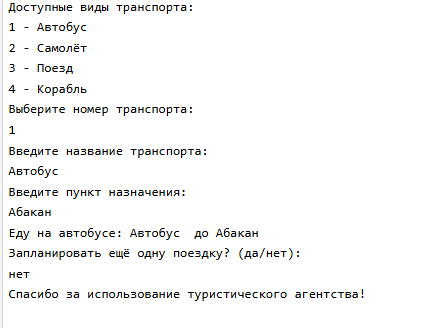
break

# Запуск программы

if \_\_name\_\_ == "\_\_main\_\_":

main()

3.Результаты



**Задание №3**

1.Постановка задачи. <https://github.com/Aleksey4ik6/Alekseyka>

Создайте класс ComputerBuilder, который будет строить объект

Computer шаг за шагом. Класс Computer должен иметь следующие атрибуты:

• Центральный процессор;

• Оперативная память;

• Накопитель;

• Видеокарта;

• Комплектующие.

3. Реализованный проект отправьте в свой GitHub-репозиторий

4. В отчёте предоставьте задание, ссылку на GitHub и скриншоты

выполнения программы.

2.Листинг программы:

class Computer:

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

self.cpu = None # Центральный процессор

self.ram = None # Оперативная память

self.storage = None # Накопитель

self.gpu = None # Видеокарта

self.components = [] # Другие комплектующие

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

return (

f"Computer Specifications:\n"

f"CPU: {self.cpu}\n"

f"RAM: {self.ram}\n"

f"Storage: {self.storage}\n"

f"GPU: {self.gpu}\n"

f"Additional Components: {', '.join(self.components)}"

)

class ComputerBuilder:

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

self.computer = Computer()

def set\_cpu(self, cpu):

self.computer.cpu = cpu

return self

def set\_ram(self, ram):

self.computer.ram = ram

return self

def set\_storage(self, storage):

self.computer.storage = storage

return self

def set\_gpu(self, gpu):

self.computer.gpu = gpu

return self

def add\_component(self, component):

self.computer.components.append(component)

return self

def build(self):

return self.computer

# Создание компьютера с использованием ComputerBuilder

builder = ComputerBuilder()

custom\_pc = (

builder.set\_cpu("Intel Core i9")

.set\_ram("32GB DDR4")

.set\_storage("1TB SSD")

.set\_gpu("NVIDIA RTX 4090")

.add\_component("Wi-Fi Card")

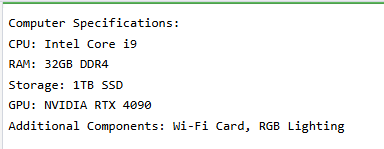
.add\_component("RGB Lighting")

.build()

)

print(custom\_pc)

3.Результаты



1.Постановка задачи. <https://github.com/Aleksey4ik6/Alekseyka>

2. \*Создайте базовый класс Animal и наследуйте от него, как

минимум, 2 конкретных вида животных. Используя паттерн «Стратегия» и

пример его реализации выше – создайте любые поведения животных

(например, MeowStrategy – мяукать, FlyStrategy – взлететь) и присвойте

подходящее для созданного вами животное поведение.

3. Реализованный проект отправьте в свой GitHub-репозиторий

4. В отчёте предоставьте задание, ссылку на GitHub и скриншоты

выполнения программы.

2.Листинг программы:

from abc import ABC, abstractmethod

# Базовый класс животных

class Animal:

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

self.name = name

self.behavior\_strategy = behavior\_strategy

def perform\_behavior(self):

self.behavior\_strategy.behave(self.name)

# Абстрактный базовый класс стратегии поведения

class BehaviorStrategy(ABC):

@abstractmethod

def behave(self, animal\_name):

pass

# Конкретные стратегии поведения

class MeowStrategy(BehaviorStrategy):

def behave(self, animal\_name):

print(f"{animal\_name} мяукает: 'Мяу-мяу!'")

class FlyStrategy(BehaviorStrategy):

def behave(self, animal\_name):

print(f"{animal\_name} взлетает в воздух!")

class BarkStrategy(BehaviorStrategy):

def behave(self, animal\_name):

print(f"{animal\_name} гавкает: 'Гав-гав!'")

# Конкретные классы животных

class Cat(Animal):

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

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

class Bird(Animal):

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

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

class Dog(Animal):

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

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

# Тестирование

cat = Cat("Мурзик")

bird = Bird("Чижик")

dog = Dog("Шарик")

cat.perform\_behavior() # Мурзик мяукает

bird.perform\_behavior() # Чижик взлетает

dog.perform\_behavior() # Шарик гавкает

3.Результаты

