МИНИCTEPCTBO НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Кафедра прикладной информатики**

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

ДИСЦИПЛИНЫ «Объектно – ориентированное программирование»

НА ТЕМУ:

**Основы объектно – ориентированного программирования на ЯП Python**

**Выполнил:**

студент группы ПИН-б-о-21-1

Стороженко Артем Владимирович

Проверил: А. А. Щеголев

Ставрополь, 2023

**Цель работы:** изучить базовые понятия (классы, подклассы и методы)

Реализовать фундаментальные принципы объектно-ориентированного

программирования.

**Задание 1**

В задании 1 было необходимо создать класс, работающий с римскими цифрами на основе шаблона.

Римское число может быть инициализировано как на основе целочисленного, так и строкового значения (римское или арабское число). При инициализации также проверяется, входит ли заданное число в допустимый промежуток. Код инициализатора класса представлен на листинге 1.1.

def \_\_init\_\_(self, value):  
  
 *"""Инициализация класса.  
  
 Параметры:  
 value (str): римское число, например, X.  
 или  
 value (int): арабское число, например, 5.  
 или  
 value (другой тип): возбудить исключение TypeError.  
 """* if not isinstance(value, (int, str)):  
 raise TypeError("Не могу создать римское число из {0}".  
 format(type(value)))  
  
 if isinstance(value, int):  
 self.\_\_check\_arabic(value)  
 self.\_arabic = value  
 elif isinstance(value, str):  
 self.\_\_check\_roman(value)  
 self.\_arabic = self.to\_arabic(value)

Листинг 1.1 – Инициализатор класса Roman

Проверка допустимости инициализируемых значений определяется с помощью методов \_\_check\_arabic и \_\_check\_roman. Их реализация представлена на листинге 1.2.

@staticmethod  
def \_\_check\_arabic(value):  
 *"""Возбудить исключение ValueError, если 'value' не принадлежит  
 [ARABIC\_MIN; ARABIC\_MIN]."""* if not Roman.ARABIC\_MIN <= value <= Roman.ARABIC\_MAX:  
 raise ValueError  
  
@staticmethod  
def \_\_check\_roman(value):  
 *"""Возбудить исключение ValueError, если 'value' содержит  
 недопустимые символы (не входящие в LETTERS)."""* for input\_letter in value:  
 for valid\_letter in Roman.LETTERS:  
 if input\_letter == valid\_letter:  
 break  
 elif valid\_letter == Roman.LETTERS[len(Roman.LETTERS) - 1]:  
 raise ValueError

Листинг 1.2 – методы \_\_check\_arabic и check\_roman

Внутри класса, римские цифры хранятся в виде своих арабских представлений в свойстве класса \_arabic, и преобразуются обратно только для вывода. Для перевода римских цифр в арабские используется функция to\_arabic, которая последовательно преобразует каждую римскую цифру в числе в арабскую. Реализация метода представлена на листинге 1.3.

@staticmethod  
def to\_arabic(roman):  
 *"""Преобразовать римское число 'roman' в арабское.  
  
 Параметры:  
 roman (str): римское число, например, "X".  
  
 Возвращает:  
 int: арабское число.  
 """* def letter\_to\_number(letter):  
 *"""Вернуть арабское значение римской цифры 'letter'.  
  
 Регистр не учитывается."""* for letter\_position in range(0, len(Roman.LETTERS)):  
 if letter == Roman.LETTERS[letter\_position]:  
 return Roman.NUMBERS[letter\_position]  
  
 Roman.\_\_check\_roman(roman)  
  
 i = 0 # Позиция в строке roman  
 value = 0 # Преобразованное число  
  
 while i < len(roman):  
  
 number = letter\_to\_number(roman[i])  
  
 i += 1  
  
 if i == len(roman):  
 # В строке roman больше не осталось символов, добавляем number  
 value += number  
 else:  
 # Если символы остались, необходимо посмотреть на следующий.  
 # Если следующий символ "больше", считаем их за одну цифру.  
 # Это необходимо, например, для того,  
 # чтобы IV преобразовать в 4, а не 15.  
 next\_number = letter\_to\_number(roman[i])  
 if next\_number > number:  
 # Комбинируем цифры и перемещаем i к следующей  
 value += next\_number - number  
 i += 1  
 else:  
 # Просто добавляем следующую цифру  
 value += number  
  
 Roman.\_\_check\_arabic(value)  
 return value

Листинг 1.3 – метод to\_arabic

Преобразование арабских цифр в римские производится аналогичным образом. Код метода представлен на листинге 1.4.

@staticmethod  
def to\_roman(arabic):  
 *"""Преобразовать арабское число 'arabic' в римское.  
  
 Параметры:  
 arabic (int): арабское число, например, 5.  
  
 Возвращает:  
 str: римское число.  
 """* Roman.\_\_check\_arabic(arabic)  
  
 roman = ""  
 # n - часть arabic, которую осталось преобразовать  
 n = arabic  
  
 for i, number in enumerate(Roman.NUMBERS):  
 while n >= number:  
 roman += Roman.LETTERS[i]  
 n -= Roman.NUMBERS[i]  
  
 return roman

Листинг 1.4 – метод to\_roman

Для строкового представления римского числа используется метод to\_roman. Код представлен на листинге 1.5.

def \_\_str\_\_(self):  
 *"""Вернуть строковое представление класса."""* return Roman.to\_roman(self.\_arabic)

Листинг 1.5 – Строковое представление римского числа

Перегрузка арифметических операций для римского числа заключается в применении арифметической операции к арабскому представлению числа, хранящемуся в свойстве \_arabic и вторым операндом. Арифметические операции также работают, если второй операнд представляет собой римское число. В таком случае, используется его арабское представление. Код для методов арифметических операций представлен на листинге 1.6.

def \_\_add\_\_(self, other):  
 *"""Создать новый объект как сумму 'self' и 'other'.  
  
 Параметры:  
 other (Roman): ...  
 или  
 other (int): арабское число, добавить к self.  
 или  
 other (другой тип): возбудить исключение TypeError.  
 """* if isinstance(other, int):  
 result = self.arabic + other  
 else:  
 result = self.arabic + other.arabic  
 return self.to\_roman(result)  
  
def \_\_sub\_\_(self, other):  
 *"""Создать новый объект как разность self и other.  
  
 Параметры:  
 other (Roman): ...  
 или  
 other (int): арабское число, добавить к self.  
 или  
 other (другой тип): возбудить исключение TypeError.  
 """* if isinstance(other, int):  
 result = self.arabic - other  
 else:  
 result = self.arabic - other.arabic  
 return self.to\_roman(result)  
  
def \_\_mul\_\_(self, other):  
 *"""Создать новый объект как произведение self и other.  
  
 Параметры:  
 other (Roman): ...  
 или  
 other (int): арабское число, добавить к self.  
 или  
 other (другой тип): возбудить исключение TypeError.  
 """* if isinstance(other, int):  
 result = self.arabic \* other  
 else:  
 result = self.arabic \* other.arabic  
 return self.to\_roman(result)  
  
def \_\_floordiv\_\_(self, other):  
 *"""Создать новый объект как частное self и other.  
  
 Параметры:  
 other (Roman): ...  
 или  
 other (int): арабское число, добавить к self.  
 или  
 other (другой тип): возбудить исключение TypeError.  
 """* if isinstance(other, int):  
 result = self.arabic / other  
 else:  
 result = self.arabic / other.arabic  
 return self.to\_roman(result)  
  
def \_\_truediv\_\_(self, other):  
 *"""Создать новый объект как частное self и other.  
  
 Параметры:  
 other (Roman): ...  
 или  
 other (int): арабское число, добавить к self.  
 или  
 other (другой тип): возбудить исключение TypeError.  
 """* # Любое деление для римского числа считается делением нацело,  
 # поэтому необходимо передать "работу" реализованному методу  
 # целочисленного деления  
 return self.\_\_floordiv\_\_(other)

Листинг 1.6 – Арифметические операции

UML – диаграмма класса Roman представлена на рисунке 1.1.

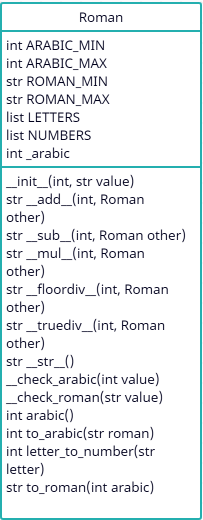


Рисунок 1.1 – UML – диаграмма класса Roman

Для проверки функционала класса был использован файл main.py. Его содержание представлено на листинге 1.7.

from roman import Roman  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 r1 = Roman("X")  
 r2 = Roman(5)  
  
 print(" Числа:", r1, r2, r1.arabic, r2.arabic)  
 print(" Сумма:", r1 + r2)  
 print(" Разность:", r1 - r2)  
 print("Произведение:", r1 \* r2)  
 print(" Частное:", r1 // r2)  
  
 print("\nПреобразование без создания объекта:")  
 print(2016, "=", Roman.to\_roman(2016))  
 print("MMXVI", "=", Roman.to\_arabic("MMXVI"))

Листинг 1.7 – Код файла main.py

Результат выполнения этого кода представлен на рисунке 1.2.

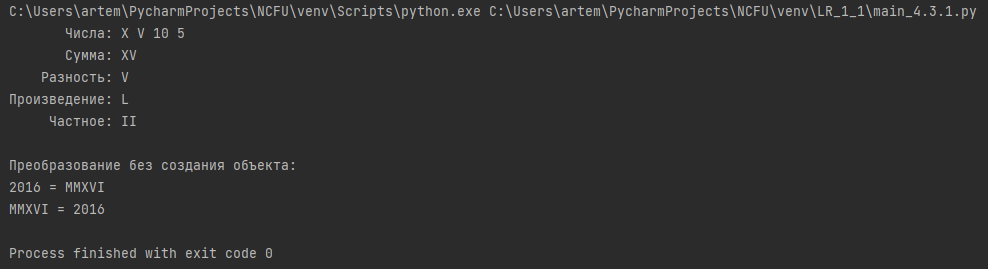


Рисунок 1.2 – Результат выполнения файла main.py

**Задание 2.**

В задании 2 необходимо было создать классы для обеспечения работы терминала по заказу пиццы.

В первую очередь были реализованы классы, описывающие виды пиццы. Все поля и методы классов определены в родительском классе Пицца, остальные классы отличаются лишь значениями при инициализации. Реализация класса Пицца представлена на листинге 2.1.

class Пицца:  
   
 *"""Класс Пицца содержит общие атрибуты для пиццы.  
  
 Дочерние классы будут их конкретизировать.  
 """*

def \_\_init\_\_(self):  
 self.название = "Заготовка"  
 self.тесто = "тонкое"  
 self.соус = "кечтуп"  
 self.начинка = []  
  
 self.цена = 0  
  
 def \_\_str\_\_(self):  
 res = ""  
 res += f"Пицца: {self.название} | Цена: {self.цена} р.\n"  
 res += f"Тесто: {self.тесто}\n"  
 res += f"Соус: {self.соус}\n"  
 res += f"Начинка: {self.начинка}\n"  
 return res  
  
 def подготовить(self):  
 print(f"Начинаю готовить пиццу {self.название}...")  
 print(f" -замешиваю {self.тесто} тесто...")  
 print(f" -добавляю соус {self.соус}...")  
 print(f" -и конечно: {self.начинка}")  
  
 def испечь(self):  
 print(f"Начинаю выпекать пиццу {self.название}... Готово!")  
  
 def нарезать(self):  
 print("Нарезаю на аппетитные кусочки...")  
  
 def упаковать(self):  
 print("Упаковываю в фирменную упаковку и готово!")

Листинг 2.1 – Реализация класса Пицца

На листингах 2.2, 2.3 и 2.4 представлены реализации классов ПиццаПепперони, ПиццаБарбекю и ПиццаДарыМоря соответственно.

class ПиццаПепперони(Пицца):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.цена = 500.00  
 self.название = "Пепперони"  
 self.начинка = ["пепперони", "моцарелла", "томаты"]

Листинг 2.2 – Реализация класса ПиццаПепперони

class ПиццаБарбекю(Пицца):  
  
 def \_\_init\_\_(self):  
 self.название = "Барбекю"  
 self.цена = 800.00  
 self.соус = "барбекю"  
 self.тесто = "тонкое"  
 self.начинка = ["цыпленок", "бекон", "красный лук", "моцарелла"]

Листинг 2.3 – Реализация класса ПиццаБарбекю

class ПиццаДарыМоря(Пицца):  
  
 def \_\_init\_\_(self):  
 self.название = "Дары моря"  
 self.цена = 740.00  
 self.соус = "белый"  
 self.тесто = "тонкое"  
 self.начинка = ["кальмар", "креветки", "мидии", "моцарелла", "оливки", "болгарский перец", "красный лук"]

Листинг 2.4 – Реализация класса ПиццаДарыМоря

Далее на основе шаблона был создан класс Заказ, содержащий все заказанные пиццы и рассчитывающий собственную стоимость.

class Заказ:  
 счетчик\_заказов = 0  
  
 def \_\_init\_\_(self):  
 self.заказанные\_пиццы = []  
 Заказ.счетчик\_заказов += 1  
  
 def \_\_str\_\_(self):  
 res = ""  
 res += f"Заказ №{Заказ.счетчик\_заказов}\n"  
 for пицца in self.заказанные\_пиццы:  
 res += f"{self.заказанные\_пиццы.index(пицца)}. Пицца: {пицца} | Цена: {пицца.цена} р.\n"  
 res += f"Тесто: {пицца.тесто} Соус: {пицца.соус}\n"  
 res += f"Начинка: {пицца.начинка}\n"  
 return res  
  
 def добавить(self, пицца):  
  
 *"""Метод добавления пиццы к заказу"""* self.заказанные\_пиццы.append(пицца)  
 print(f"Пицца {пицца.название} добавлена к заказу")  
  
 def сумма(self):  
  
 *"""Метод рассчета стиомости заказанных пицц"""* order\_sum = 0  
 for пицца in self.заказанные\_пиццы:  
 order\_sum += пицца.цена  
 return order\_sum  
  
 def выполнить(self):  
  
 *"""Метод, служащий для приготовления всех пицц в заказе"""* for пицца in self.заказанные\_пиццы:  
 for действие in [пицца.подготовить, пицца.испечь, пицца.нарезать, пицца.упаковать]:  
 time.sleep(1)  
 действие()

Листинг 2.5 – Код класса Заказ

Далее был реализован класс Терминал, обеспечивающий взаимодействие пользователя с пиццерией (создание заказов, добавление пицц, оплата заказов).

Вначале в классе были указаны константы, а именно версия программы и компания, обслуживающая клиента и команды отмены и подтверждения заказа. Они показаны на листинге 2.6

*"""Константы класса"""*ВЕРСИЯ = 1  
КОМПАНИЯ = "Pronto"  
КОМАНДА\_ОТМЕНА\_ЗАКАЗА = -1  
КОМАНДА\_ПОДТВЕРЖДЕНИЕ\_ЗАКАЗА = 0

Листинг 2.6 – Константы класса Терминал

Далее были реализованы конструктор класса и его строковое представление. В конструкторе класса инициализируется переменная для заказа и поле, определяющее отображение меню и самое меню, которое состоит из трех объектов пицц. Код представлен на листинге 2.7.

def \_\_init\_\_(self):  
 self.меню = [ПиццаПепперони(), ПиццаБарбекю(), ПиццаДарыМоря()]  
 self.заказ = None  
 self.отображать\_меню = True  
  
def \_\_str\_\_(self):  
 return f"{self.КОМПАНИЯ} #{self.ВЕРСИЯ}\nДобро пожаловать!\n"

Листинг 2.7 – Код конструктора и строкового представления класса Терминал

В начале работы программы отображается меню, что реализовано в методе класса «показать\_меню». Код метода представлен на листинге 2.8.

def показать\_меню(self):  
  
 *"""Метод для отображения меню пиццерии. Вызывается один раз при начале работы программы"""* if not self.отображать\_меню:  
 return  
 else:  
 for пицца in self.меню:  
 print(пицца)  
 print("Для выбора заказа укажите цифру через <Enter>")  
 print("Для отмены заказа введите -1")  
 print("Для подтверждения заказа введите 0\n")  
 self.отображать\_меню = False

Листинг 2.8 – Код метода «показать\_меню»

Основой функционала класса является метод «обработать\_команду», определяющий действия при вводе пользователем в терминал определенных в класса команд. Этот метод обеспечивает добавление пицц в заказ, подтверждение заказа для оплаты и отмену заказа. Реализация этого метода представлена на листинге 2.9.

def обработать\_команду(self, пункт\_меню):  
  
 *"""Метод, определяющий действия для каждой предопределенной команды терминала"""* try:  
  
 """Отмена заказа"""  
  
 пункт\_меню = int(пункт\_меню)  
 if пункт\_меню == Терминал.КОМАНДА\_ОТМЕНА\_ЗАКАЗА:  
 self.отмена\_заказа()  
 return  
  
 """Подтверждение заказа"""  
  
 elif пункт\_меню == Терминал.КОМАНДА\_ПОДТВЕРЖДЕНИЕ\_ЗАКАЗА:  
 self.принять\_оплату()  
 print("Заказ подтвержден.")  
 print(f"Заказ №{Заказ.счетчик\_заказов}")  
 self.заказ.выполнить()  
 print(f"Заказ №{Заказ.счетчик\_заказов} готов! Приятного аппетита!")  
 self.заказ = None  
  
 """Выбор пункта меню (выбор пиццы для добавления в заказ)"""  
  
 elif 1 <= пункт\_меню <= len(self.меню):  
 if self.заказ == None:  
 self.заказ = Заказ()  
 self.заказ.добавить(self.меню[пункт\_меню - 1])  
 else:  
 raise ValueError  
  
 """Действия при вводе неопределенной команды"""  
  
 except ValueError:  
 print("Не могу распознать команду! Проверьте ввод.")  
 if self.заказ != None:  
 self.отмена\_заказа()  
  
 """Действия при возникновении иных ошибок"""  
  
 except Exception:  
 print("Во время работы терминала произошла ошибка...")  
 self.отмена\_заказа()

Листинг 2.9. – Реализация метода «обработать\_команду»

Для проведения операции оплаты служат методы «принять\_оплату», позволяющий пользователю ввести сумму оплаты и «рассчитать\_сдачу», проверяющий валидность оплаты и определяющий размер сдачи. Код методов представлен на листинге 2.10.

def рассчитать\_сдачу(self, оплата):  
  
 *"""Метод для рассчеты выдаваемой сдачи. При вводе оплаты меньше цены возникает исключение ValueError"""* if оплата >= self.заказ.сумма():  
 сдача = оплата - self.заказ.сумма()  
 print(f"Вы ввели сумму: {self.заказ.сумма()} р. Ваша сдача: {сдача} р.")  
 else:  
 raise ValueError  
  
def принять\_оплату(self):  
  
 *"""Метод, принимающий оплату от клиента"""* try:  
 print(f"Сумма оплаты: {self.заказ.сумма()}")  
 сумма\_оплаты = input("Введите сумму оплаты: ")  
 сумма\_оплаты = float(сумма\_оплаты)  
 self.рассчитать\_сдачу(сумма\_оплаты)  
 except Exception:  
 print("Оплата не удалась. Заказ будет отменен.")  
 raise

Листинг 2.10 – Методы «принять\_оплату» и «рассчитать\_сдачу»

Для отмены заказа используется метод «отмена\_заказа». Он делает значение поля «заказ» объекта терминала равным NULL, и уменьшает счетчик заказов на 1. Код представлен на листинге 2.11.

def отмена\_заказа(self):  
  
 *"""Метод для отмены заказа. В поле заказ устанавливается значение по умолчанию, а счетчик заказов сбрасывается на 1"""* self.заказ = None  
 Заказ.счетчик\_заказов -= 1  
 print("Заказ отменен")

Листинг 2.11 – Код метода «отмена\_заказа»

UML – диаграмма класса представлена на рисунке 2.1.

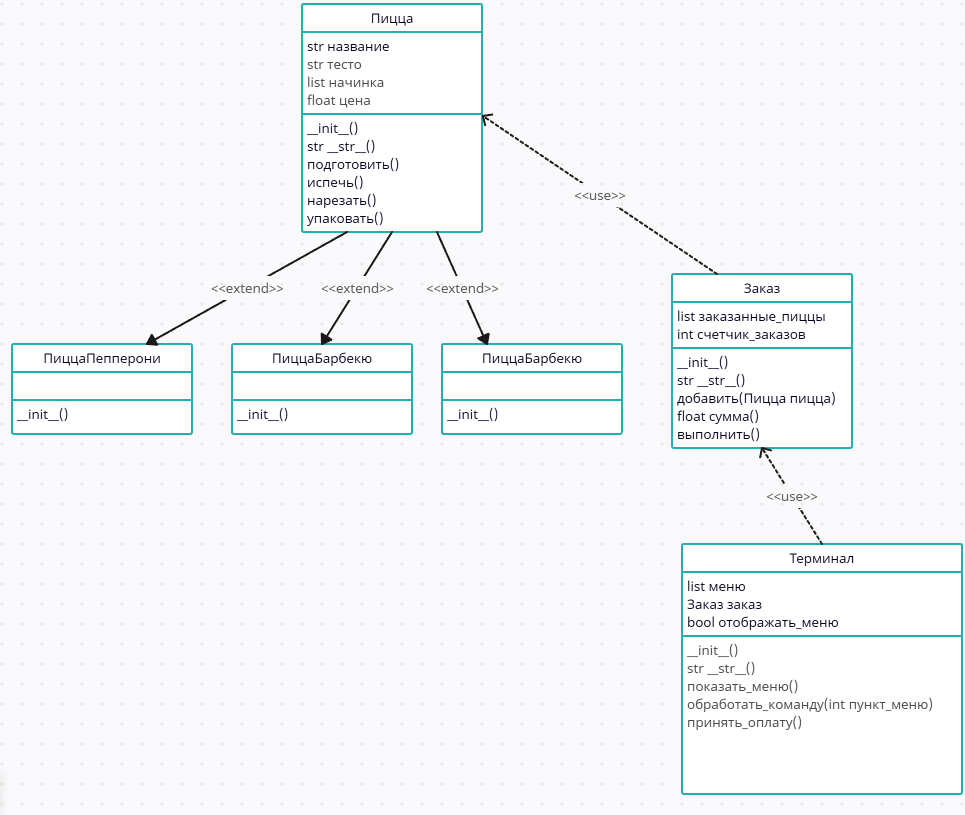


Рисунок 2.1 – UML – диаграмма классов задания 2

Для тестирования функционала класса была создана программа main.py. Ее код представлен на листинге 2.12.

from терминал import Терминал  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 терминал1 = Терминал()  
 print(терминал1)  
 while True:  
 терминал1.показать\_меню()  
 пункт\_меню = input()  
 терминал1.обработать\_команду(пункт\_меню)

Листинг 2.12 – Код программы main.py

Пример ее использования представлен на рисунках 2.2 и 2.3.

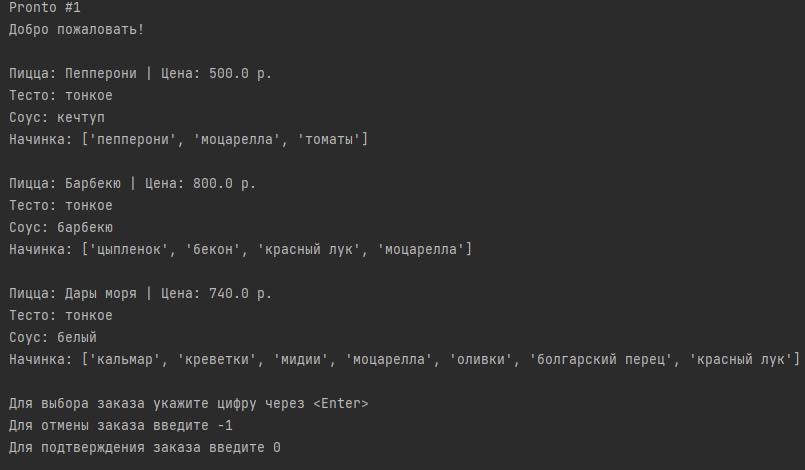


Рисунок 2.2 – Вывод меню

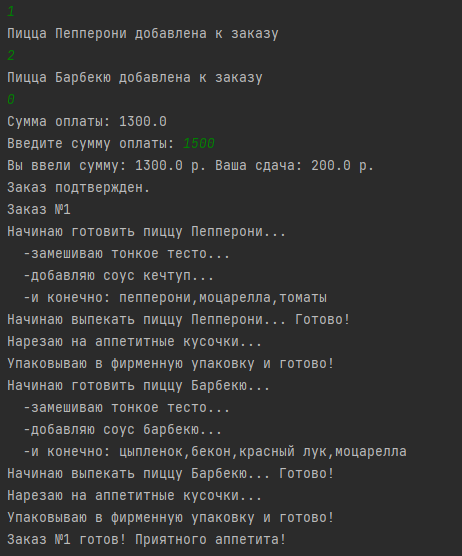


Рисунок 2.3 – Создание и исполнение заказа

**Задание 3**

В задании 3 необходимо было создать классы, представляющие собой различные предложения по вкладам.

Родительский класс TimeDeposit содержит следующие поля: имя, процент, ограничения по времени вклада, ограничения по сумме вклада. При инициализации проводится проверка инициализированных значений. Код конструктора представлен на листинге 3.1

def \_\_init\_\_(self, name, interest\_rate, period\_limit, sum\_limit):  
 self.name = name  
 self.\_interest\_rate = interest\_rate  
 self.\_period\_limit = period\_limit  
 self.\_sum\_limit = sum\_limit  
 # Проверить значения  
 self.\_check\_self()

Листинг 3.1 – Код конструктора класса TimeDeposit

Перегруженный метод строкового представления выводит инфомрацию о наименовании, валюте, процентной ставке, сроке и сумме вклада. Код перегруженного метода представлен на листинге 3.2.

def \_\_str\_\_(self):  
  
 *"""Вернуть строкое представление депозита.  
  
 К информации о родителе добавляется информация о капитализации.  
  
 Формат вывода:  
  
 Наименование: Вклад с Капитализацией  
 Валюта: руб.  
 Процентная ставка: 5  
 Срок (мес.): [6; 18)  
 Сумма: [1,000; 100,000)  
 Капитализация % : Да  
 """* res = f"Наименование: {self.name}\n"  
 res += f"Валюта: {self.currency}\n"  
 res += f"Процентная ставка: {self.\_interest\_rate}\n"  
 res += f"Срок (мес.): {self.\_period\_limit}\n"  
 res += f"Сумма: {self.\_sum\_limit}\n"  
 return res

Листинг 3.2 – Перегрузка строкового метода класса TimeDeposit

Метод check\_self проверяет корректность введенных ограничений по вкладам и процента (минимальная сумма вклада не может быть больше максимальной, процент не может быть меньше 1 и больше 100 и т.д.).

Метод \_check\_user\_params проверяет соблюдены ли условия вклада вкладчиком, т.е. соответствует ли введенный им период и сумма условиям вклада.

Реализация двух этих методов представлена на листинге 3.3.

def \_check\_self(self):  
  
 *"""Проверяет корректность ограничений по вкладу"""* assert 0 < self.\_interest\_rate <= 100, \  
 "Неверно указан процент по вкладу!"  
 assert 1 <= self.\_period\_limit[0] < self.\_period\_limit[1], \  
 "Неверно указаны ограничения по сроку вклада!"  
 assert 0 < self.\_sum\_limit[0] <= self.\_sum\_limit[1], \  
 "Неверно указаны ограничения по сумме вклада!"  
  
def \_check\_user\_params(self, initial\_sum, period):  
 is\_sum\_ok = self.\_sum\_limit[0] <= initial\_sum < self.\_sum\_limit[1]  
 is\_period\_ok = self.\_period\_limit[0] <= period < self.\_period\_limit[1]  
 assert is\_sum\_ok and is\_period\_ok, "Условия вклада не соблюдены!"

Листинг 3.3 – Реализация методов check\_self и \_check\_user\_params

Метод get\_profit вычисляет прибыль, которую получит клиент в результате вклада. Он используется для того, чтобы рассчитать сумму вклада по окончанию срока с помощью метода get\_sum. Код этих методов представлен на листинге 3.4.

def get\_profit(self, initial\_sum, period):  
  
 *"""Вернуть прибыль по вкладу вклада клиента."""* self.\_check\_user\_params(initial\_sum, period)  
 return initial\_sum \* self.\_interest\_rate / 100 \* period / 12  
  
def get\_sum(self, initial\_sum, period):  
 return initial\_sum + self.get\_profit(initial\_sum, period)

Листинг 3.4 – Код методов get\_profit и get\_sum

Класс BonusDeposit расширяет родительский класс. Его отличия от родительского класса заключаются в наличии поля, описывающего бонусный процент и ограничения по сумме для его начисления, отображение этих полей в строковом представлении класса, изменненую формулу рассчета прибыли и измененный метод проверки собственных полей. Код класса представлен на листинге 3.5.

class BonusTimeDeposit(TimeDeposit):  
   
 *"""Cрочный вклад c получением бонуса к концу срока вклада.  
  
 Бонус начисляется как % от прибыли, если вклад больше определенной суммы.  
  
 Атрибуты:  
 - self.\_bonus (dict ("percent"=int, "sum"=float)):  
 % от прибыли, мин. сумма;  
 """* def \_\_init\_\_(self, name, interest\_rate, period\_limit, sum\_limit, bonus):  
 self.\_bonus = bonus  
 super().\_\_init\_\_(name, interest\_rate, period\_limit, sum\_limit)  
  
 def \_\_str\_\_(self):  
 res = super().\_\_str\_\_()  
 res += f"Бонус (%): {self.\_bonus['percent']}"  
 res += f"Бонус (мин. сумма): {self.\_bonus['sum']}"  
 return res  
  
 def \_check\_self(self):  
 super().\_check\_self()  
 assert 0 < self.\_bonus['sum'], \  
 "Неверно указаны ограничения по минимальной сумме бонуса по вкладу!"  
 assert 0 < self.\_bonus['percent'] <= 100, \  
 "Неверно указаны ограничения по проценту бонуса по вкладу!"  
  
 def get\_profit(self, initial\_sum, period):  
 self.\_check\_user\_params(initial\_sum, period)  
 profit = initial\_sum \* self.\_interest\_rate / 100 \* period / 12  
 if initial\_sum > self.\_bonus["sum"]:  
 profit += profit \* self.\_bonus["percent"] / 100  
 return profit

Листинг 3.6 – Код класса CompoundTimeDeposit

Также, в соответствии с заданием были созданы отдельные вклады (объекты классов вкладов). Их список показан на листинге 3.7.

deposits\_data = dict(interest\_rate=5, period\_limit=(6, 18),  
 sum\_limit=(1000, 100000))  
  
deposits = (  
 TimeDeposit("Сохраняй", interest\_rate=5,  
 period\_limit=(6, 18),  
 sum\_limit=(1000, 100000)),  
 BonusTimeDeposit("Бонусный 2", \*\*deposits\_data,  
 bonus=dict(percent=5, sum=2000)),  
 CompoundTimeDeposit("С капитализацией", \*\*deposits\_data),  
 CompoundTimeDeposit("Долгосрочный", interest\_rate=10, period\_limit=(36, 60), sum\_limit=(1000, 100000)),  
 BonusTimeDeposit("Бонусный MAX", \*\*deposits\_data, bonus=dict(percent=15, sum=45000))  
 )

Листинг 3.7 – Возможные вклады

На рисунке 3.1 представлена UML – диаграмма классов для этого задания.

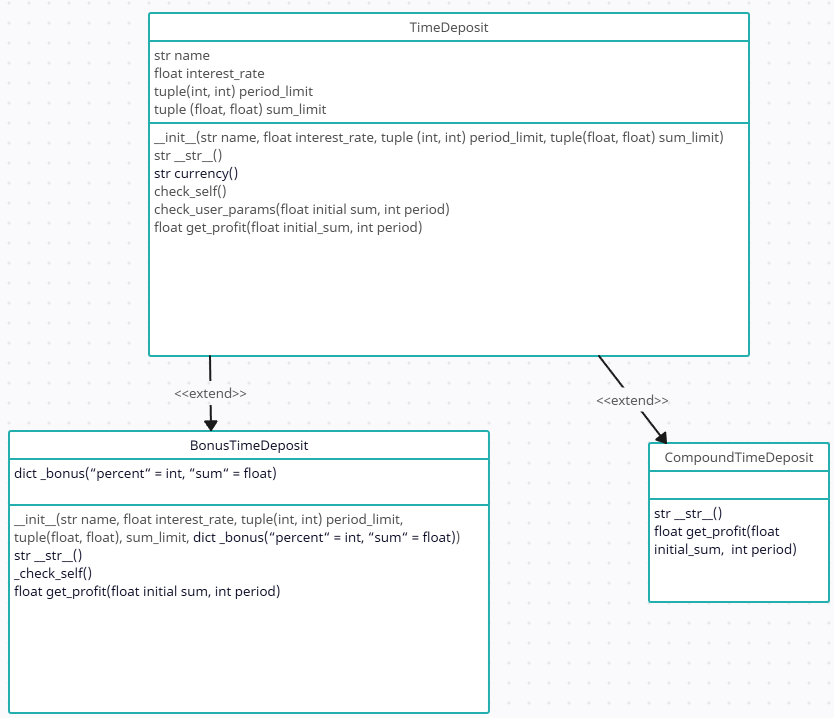


Рисунок 3.1 – UML – диаграмма классов для задания 3

Для проверки функционала классов была создана программа main.py. Пример ее работы представлен на рисунке 3.2.

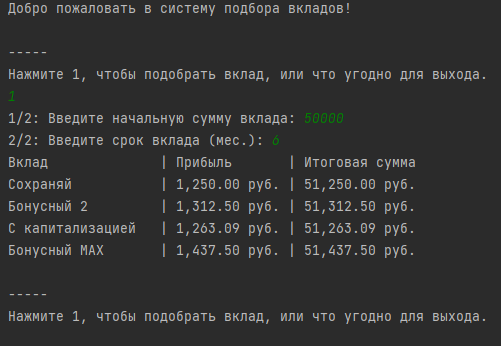


Рисунок 3.2 – Пример работы программы main.py

**Задание 4**

В задании 4 нужно было реализовать простой класс в соответствии с вариантом. Мне, в соответствии с 8 варинтом нужно было реализовать класс «Очередь».

Очередь представляет собой структуру данных оперирующую ими в соответствии с правилом FIFO (First In First Out), то есть сначала из нее выбывают элементы, вошедшие в нее первыми.

В классе определены следующие поля: name – имя очереди, \_items – список элементов очереди, front – первый элемент очереди, rear – последний элемент очереди. Конструктор класса представлен на листинге 4.1.

def \_\_init\_\_(self, items=[], name=None):  
 *"""  
 Инициализация атрибутов класса Очередь. Параметр имя является необязательным,  
  
 Первый и последний элементы очереди задаются, если в ней есть хотя бы один элемент.  
 """* self.\_items = items # Элементы очереди  
 self.name = name # Название объекта (необязательно)

Листинг 4.1 – Конструктор класса «Очередь»

Прямое обращение к элементам очереди не допускается, можно лишь получить копию их списка с помощью метода get\_items() и получить значение следующего элемента с помощью метода peek(). Эти методы представлены на листинге 4.2.

def get\_items(self):  
 *"""Возвращает список всех элементов очереди"""* return self.\_items

def peek(self):  
  
 *"""Возвращает значение следующего элемента в очереди"""* if self.count() > 0:  
 return self.\_items[0]  
 else:  
 raise IndexError("The queue is empty")

Листинг 4.2 – Код метода get\_items и свойств front и rear

Добавление и исключение элементов из очереди производится с помощью специальных методов enqueue и dequeue. Реализация этих методов представлена на листинге 4.3.

def enqueue(self, item):  
 *"""Добавить элемент в конец очереди"""* self.\_items.append(item)  
  
def dequeue(self):  
 *"""Вернуть и убрать из очереди первый элемент.  
 Если элементов в очереди нет, вызывается IndexError"""* if self.count() > 0:  
 return self.\_items.pop(0)  
 else:  
 raise IndexError("The queue is empty")

Листинг 4.3 – Методы enqueue и dequeue

Строковое представление класса – список его элементов. Реализация строкового метода представлена на листинге 4.4.

def \_\_str\_\_(self):  
  
 *"""Строковое представление класса. Возвращается список элементов"""* return f"{self.\_items}"

Листинг 4.4 – Строковое представление класса, методы isEmpty и clear

Для очереди также реализованы методы подсчета количества элементов и создание объекта очереди из строки. Код этих методов представлен на листинге 4.5.

def count(self):  
  
 *"""Получение количества элементов очереди"""* return len(self.\_items)

def from\_string(self, str\_value):  
  
 *"""Создание элемента из строки"""* return Queue(str\_value.split(","))

Листинг 4.5 – Методы проверки на вхождение и создания объекта из строки

В функционал класса также входит возможность сохранения объекта класса в JSON – файл и его последующая загрузка из него. Реализация этих методов представлена на листинге 4.6.

def save(self, filename):  
  
 *"""Сохранение объекта в JSON - файл"""* queue\_json = {'\_items' : self.\_items}  
 with open(f"{filename}.json", "w") as file:  
 json.dump(queue\_json, file, indent=2)  
  
@staticmethod  
def load(filename):  
  
 *"""Загрузка объекта из JSON - файла. Возвращает объект, находящийся в JSON - файле"""* with open(f"{filename}.json") as file:  
 obj\_dict = json.load(file)  
 queue = Queue([])  
 Queue.\_items = obj\_dict["\_items"]  
 return queue

Листинг 4.6 – Загрузка и выгрузка объекта класса из JSON – файла

Для класса «Очередь» также были реализованы перегрузки методов арифметических действий. Суть перегрузок заключаласю в том, чтобы провести арифметичскую операцию с первым элементом очереди и исключить его из нее после этого. Код этих методов представлен на листинге 4.7

def \_\_add\_\_(self, other):  
  
 *"""Перегрузка оператора сложения для очереди.  
 Сложение происходит с первым элементом очереди, который после этого из нее убирается"""* if ((isinstance(other, int) and isinstance(self.\_items[0], int)) or # Проверка соответствия типов  
 (isinstance(other, float) and isinstance(self.\_items[0], float))):  
 result = self.\_items[0] + other  
 self.dequeue()  
 return result  
 else:  
 raise ValueError  
  
def \_\_sub\_\_(self, other):  
  
 *"""Перегрузка оператора вычитания. Принцип работы аналогичен перегрузке метода сложения"""* if ((isinstance(other, int) and isinstance(self.\_items[0], int)) or # Проверка соответствия типов  
 (isinstance(other, float) and isinstance(self.\_items[0], float))):  
 result = self.\_items[0] - other  
 self.dequeue()  
 return result  
 else:  
 raise ValueError("Wrong data type")  
  
def \_\_mul\_\_(self, other):  
  
 *"""Перегрузка оператора умножения. Принцип работы аналогичен перегрузке метода сложения"""* if isinstance(other, (int, float)) and isinstance(self.\_items[0], (int, float)): # Проверка соответствия типов  
 result = self.\_items[0] \* other  
 self.dequeue()  
 return result  
 else:  
 raise ValueError("Wrong data type")  
  
def \_\_truediv\_\_(self, other):  
 *"""Перегрузка оператора деления. Принцип работы аналогичен перегрузке метода сложения"""* if isinstance(other, (int, float)) and isinstance(self.\_items[0], (int, float)): # Проверка соответствия типов  
 result = self.\_items[0] / other  
 self.dequeue()  
 return result  
 else:  
 raise ValueError("Wrong data type")  
  
def \_\_floordiv\_\_(self, other):  
  
 *"""Перегрузка оператора деления нацело. Принцип работы аналогичен перегрузке метода сложения"""* if isinstance(other, int) and isinstance(self.\_items[0], int):  
 result = self.\_items[0] // other  
 self.dequeue()  
 return result  
 else:  
 raise ValueError("Wrong data type")

Листинг 4.7 – Код перегруженных методов арифметических операций

UML – диграмма класса Queue представлена на рисунке 4.1.

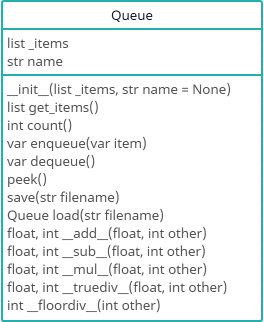


Рисунок 4.1 – UML – диаграмма класса Queue

Для тестирования функционала класса была создана программа main.py. Ее код представлен на листинге 4.8.

# Программирование на языке высокого уровня (Python).  
# Задание №4.3.4. Вариант 8  
#  
# Выполнил: Стороженко А.В.  
# Группа: ПИН-б-о-21-1  
# E-mail: artem\_storozhenko\_2016@mail.ru  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 from queue import Queue  
 queue = Queue([1, 2.25, 3])  
  
 # Добавление в очередь  
 print(f"Добавление в очередь: {queue.get\_items()}", end=' --> ')  
 queue.enqueue("fourth")  
 print(queue.get\_items())  
  
 # Удаление из очереди  
 print(f"Удаление из очереди: {queue.get\_items()}", end=' --> ')  
 queue.dequeue()  
 print(queue.get\_items())  
  
 # Сохранение объекта в JSON - файл  
 print("Сохранение объекта в JSON - файл")  
 queue.save("object")  
 with open("object.json", 'r') as file:  
 for line in file:  
 print(line)  
 # Получение значения следующего элемента в очереди  
 print(f"Получение следующего элемента очереди {queue.get\_items()} --> {queue.peek()}")  
  
 # Строковое представление объекта  
 print(f"Строковое представление объекта: {queue}")  
  
 # Получение количества элементов в очереди  
 print(f"Количество элементов: {queue.get\_items()} --> {queue.count()}")  
  
 # Загрузка объекта из JSON - файла  
 print("До загрузки")  
 print(f"Элементы очереди: {queue.get\_items()}")  
 print("\nПосле загрузки")  
 queue.load("object")  
 print(f"Элементы очереди: {queue.get\_items()}")  
  
 # Арифметические операции над объектом  
 queue.enqueue(4)  
 print(f"queue / 2 = {queue / 2}")  
  
 # Создание очереди из строки  
 print(f"""Создание очереди из строки: {queue.from\_string("1,2,3,4,5")}""")

Листинг 4.8 – Код программы main.py

Результат работы этой программы показан на рисунке 4.2

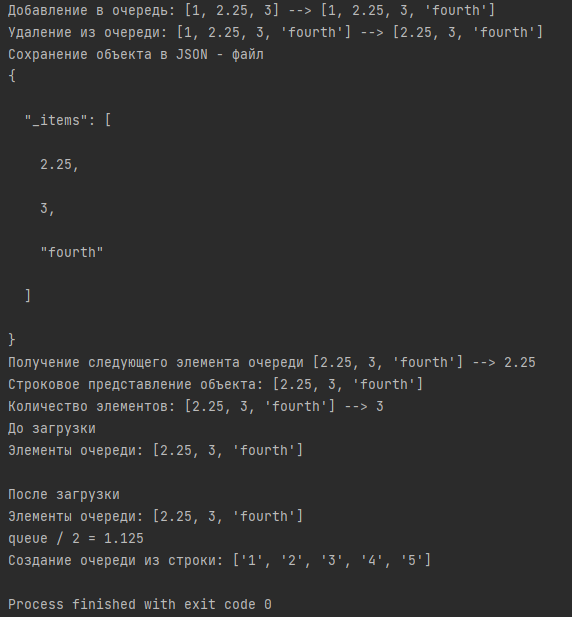


Рисунок 4.2 – Результат работы main.py для задания 4

**Задание 5**

В 5 задании необходимо было создать класс, представляющий собой контейнер для объектов класса, созданного в предыдущем задании, то есть мне нужно было реализовать класс, описывающий коллекцию очередей.

Конструктор класса инициализирует значение только одного поля - \_items. При этом, он также проверяет, относится ли каждый из элементов списка к объекту класса «Очередь». Если это не так, возникает исключение TypeError. Конструктор класса представлен на листинге 5.1.

class QueueCollection:  
 def \_\_init\_\_(self, items=[]):  
  
 *"""Инициализация атрибутов класса КоллекцияОчередей"""* for item in items:  
 if not isinstance(item, Queue):  
 raise TypeError  
 self.\_items = items

Лиситнг 5.1 – Конструктор класса QueueCollection

Так же, как и для класса «Очередь», в классе «КоллекцияОчередей» реализованы возможности сохранения и загрузки объектов класса из JSON – файла. Для правильной сериализации объектов класса используется метод toJSON, использующий метод \_\_dict\_\_ для всех элементов объекта, которые нельзя сериализовать обычным способом. Код представлен на листинге 5.2.

def toJSON(self):  
  
 *"""Преобразование объекта в JSON - строку. Для каждого объекта, который нельзя сериализировать,  
 берется его представление в виде словаря (\_\_dict\_\_)"""* return json.dumps(self, default=lambda o: o.\_\_dict\_\_,  
 sort\_keys=True, indent=4)

def save(self, filename):  
  
 *"""Сохранение объекта в JSON - файл"""* str\_json = self.toJSON()  
 with open(f"{filename}.json", "w") as file:  
 file.write(str\_json)  
  
@staticmethod  
def load(filename):  
  
 *"""Загрузка объекта из JSON - файла"""* with open(f"{filename}.json") as file:  
 newQueueCollection = QueueCollection([])  
 obj\_from\_json = json.load(file)  
 str\_items = []  
 for item in obj\_from\_json["\_items"]:  
 queue\_item = Queue(item["\_items"], name=item["name"])  
 newQueueCollection.add\_item(queue\_item)  
 return newQueueCollection

)

Листинг 5.2 – Методы преобразования в JSON – строку, сохранения и загрузки из JSON – файла

Для класса также реализованы методы очистки и проверки на пустоту. Они работают аналогично таковым в классе «Очередь». Код представлен на листинге 5.3.

def clear(self):  
  
 *"""Очистка коллекции"""* self.\_items = []  
  
def isEmpty(self):  
  
 *"""Проверка на пустоту коллекции"""* if len(self.\_items) < 1:  
 return True  
 else:  
 return False

Листинг 5.3 – Методы clear и isEmpty

Перегрузка метода строкового представления для класса «КоллекцияОчредей» выводит названия каждой очереди, находящейся в коллекции (если оно есть) и список элементов каждой коллекции. Реализация этого метода представлена на листинге 5.4.

def \_\_str\_\_(self):  
  
 *"""Строковое представление объекта. Возвращаются имя каждой очереди, если оно есть, и ее элементы"""* queues\_str = "["  
 if len(self.\_items) > 0:  
 queues\_str += "\n"  
 for queue in self.\_items:  
 if queue.name != None:  
 queues\_str += "\n" + queue.name + "\n"  
 queues\_str += str(queue) + "\n\n"  
 queues\_str += "]"  
 return queues\_str

Листинг 5.4 – Перегрузка метода строкового представления класса

В отличие от класса «Очередь» в классе «КоллекцияОчередей» предусмотрен прямой доступ к элементам, который реализуется с помощью перегрузки метода \_\_setitem\_\_. Для простого обращения по индексу предусмотрена перегрузка метода \_\_getitem\_\_. Код перегруженных методов представлен на листинге 5.5.

def \_\_getitem\_\_(self, item):  
  
 *"""Перегрузка оператора индексирования"""* return self.\_items[item]  
  
def \_\_setitem\_\_(self, item\_index, item):  
  
 *"""Перегрузка оператора записи значения по индексу"""* if isinstance(item, Queue):  
 self.\_items[item\_index] = item  
 else:  
 raise ValueError("Добавляемый элемент не является очередью!")

Листинг 5.5 – Код перегрузок методов \_\_getitem\_\_ и \_\_setitem\_\_

Помимо этого, класс содержит отдельные методы добавления и удаления элементов, не использующие обращение по индексу. Код этих методов представлен на листинге 5.6.

def add(self, queue):  
 *"""Добавление элемента в коллекцию"""* if isinstance(queue, Queue):  
 self.\_items.append(queue)  
 else:  
 raise ValueError

def remove(self, index):  
  
 *"""Удаление элемента из коллекции"""* self.\_items.pop(index)

Листинг 5.6 – Код методов add и remove

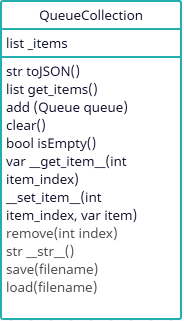


Рисунок 5.1 – UML – диаграмма класса QueueCollection

Для проверки работы класса была создана программа main.py. Ее код представлен на листинге 5.7.

if \_\_name\_\_ == "\_\_main\_\_":  
 from queue import Queue  
 from queueCollection import QueueCollection  
 queue\_1 = Queue(['15', '64', '88'], name="numbers")  
 queue\_2 = Queue(['list', 'dict', 'tuple'], name="data\_structures")  
 queue\_3 = Queue(["a", "b", "c"], name="characters")  
 queueCollection = QueueCollection([queue\_1, queue\_2, queue\_3])  
 queue\_4 = Queue(['blue', 'red', 'green'])  
  
 # Добавление элемента в коллекцию  
 queueCollection.add\_item(queue\_4)  
  
 # Строковое представление коллекции  
 print(queueCollection)  
  
 # Срез коллекции  
 print(queueCollection[1:3])  
  
 # Сохранение коллекции  
 queueCollection.save("qCol")  
  
 # Очистка коллекции  
 queueCollection.clear()  
 print(queueCollection)  
   
 # Загрузка коллекции из файла  
 new\_QCol = QueueCollection.load("qCol")  
 print(new\_QCol)  
 new\_QCol[0] = queue\_2  
 print(new\_QCol)

Листинг 5.7 – Программа maim.py для задания 5

Результат ее работы представлен на рисунке 5.2.

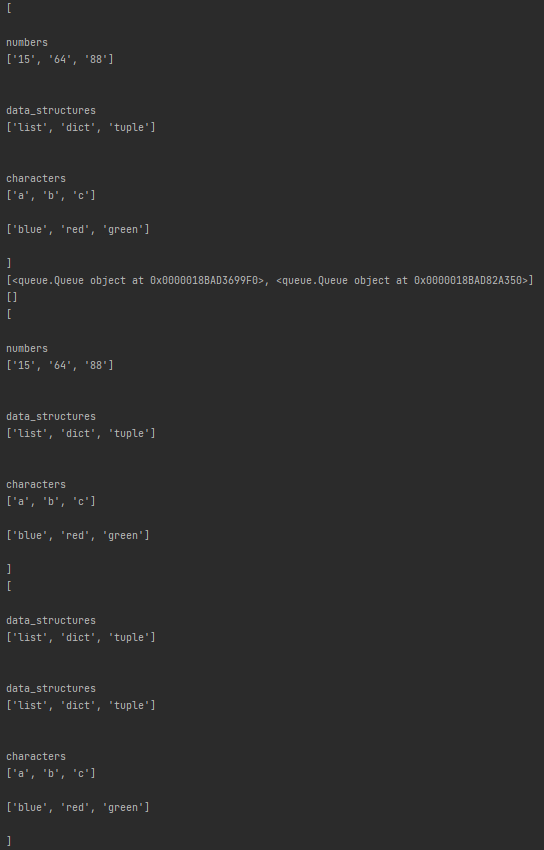


Рисунок 5.2 – Результат работы main.py для задания 5

**Задание 6**

В качестве 6 задания было предложено создать иерархию классов в соответствии с вариантом. Мой вариант – 8, поэтому из 5 представленных мной был выбран 3 – иерархия транспортных средств.

Абстрактный класс Vehicle содержит общие для сех транспортных средств поля и методы. К общим для всех ТС полям относятся:

* \_environment – определяет среду, в которой может двигаться транспортное средство (земля, вода, воздух)
* \_mechanical – определяет, является ли ТС механическим или нет
* speed – влияет на то, с какой скоростью ТС двигается по маршруту

Конструктор класса и метод \_check\_attributes представлены на листинге 6.1.

def \_\_init\_\_(self, speed, mechanical):  
 self.speed = speed  
 self.\_mechanical = mechanical  
 self.\_check\_atributes()  
  
def \_check\_attributes(self):  
  
 *"""Проверка правильности ввода атрибутов при инициализации"""* if not isinstance(self.speed, (int, float)):  
 raise ValueError("Incorrect speed value")  
 if not isinstance(self.\_mechanical, str):  
 raise ValueError("Incorrect environment value")

Листинг 6.1 – Конструктор класса и метод \_check\_attributes

Метод move абстрактного класса ТС служит для начала движения по заданному маршруту. К его параметрам относятся: среда, по которой планируется движение и длину пути. Метод движения абстрактного ТС не содержит никаких команд, поскольку создание и использование объекта абстрактного класса не предусматривается.

Для абстрактного класса реализованы методы, позволяющие получить доступ для чтения к защищенным полям класса:

* \_is\_mechanical – возвращает True, если ТС является механическим и False в обратном случае
* get\_env – возвращает копию поля \_environment транспортного средства

Также в абстрактном классе предусмотрен метод, позволяющий сравнить значение среды, передаваемое в качестве аргумента, со значением среды для движения транспортного средства.

Код всех этих методов представлен на листинге 6.2.

def \_isMechanical(self):  
  
 *"""Проверка механизированности ТС"""* if self.\_mechanical == "mechanical":  
 return True  
 else:  
 return False  
  
def get\_env(self):  
  
 *"""Метод, возвращающий среду, в которой ТС может двигаться"""* return f"The vehicle can move in this environment: {self.\_environment}"  
  
def \_check\_env(self, environment):  
   
 *"""Сравнение среды маршрута со средой возможного движения ТС"""* if self.\_environment != environment:  
 raise Exception("Wrong movement environment!")

Листинг 6.2 – Методы \_is\_mechanical

По ходу движения ТС у него возникают неисправности (реализовано в расширениях методах move дочерних классов). Для исправления этих неисправностей служат специальные методы, в абстрактном классе предусмотрено 2 таких метода: refuel и change\_oil. Эти методы могут работать только для механических ТС. В противном случае вызывается исключение. Их реализация представлена на листинге 6.3.

def \_refuel(self):  
  
 *"""Метод заправки ТС в случае обнаружения недостатка топлива. При вызове метода для немеханического ТС вызывается исключение.  
 Этой неисправности соответствует число 1"""* if self.\_isMechanical():  
 print("Refueling the vehicle...")  
 time.sleep(5)  
 print("The vehicle is fueled\n")  
 else:  
 raise Exception("Non-mechanical vehicle")  
  
def \_change\_oil(self):  
  
 *"""Метод заправки ТС в случае обнаружения устаревания масла. При вызове метода для немеханического ТС вызывается исключение.  
 Этой неисправности соответствует число 2"""* if self.\_isMechanical():  
 print("Changing oil...")  
 time.sleep(1.5)  
 print("Oil changed\n")  
 else:  
 raise Exception("Non-mechanical vehicle")

Листинг 6.3 – Методы refuel и change\_oil

Метод \_do\_service реализует поиск и исправление найденных неисправностей в ТС. Этот метод принимает в качестве аргумента целое число Для каждого класса транспортного средства предусмотрены расширения этого метода, добавляющие методы для устранения неполадок, связанным с конкретным классом транспортного средства (например, у водного ТС – метод закрытия течи). Метод абстрактного класса \_do\_service предусматривает исправление двух неполадок – заправка и смена масла. Код метода представлен на листинге 6.4.

def \_do\_service(self, malfunction):  
  
 *"""Метод, осуществляющий ремонт ТС в соответствии с неисправностью. Принимает на вход целочисленное значение,  
  
 соответствующее неисправности"""* if isinstance(malfunction, int):  
 if malfunction == 1:  
 print("\n\nRefuel required")  
 time.sleep(0.25)  
 self.\_refuel()  
 elif malfunction == 2:  
 print("\n\nOil change required")  
 self.\_change\_oil()  
 else:  
 raise ValueError("Wrong malfunction value type")

Листинг 6.4 – Метод \_do\_service

Класс WheeledVehicle расширяет класс Vehicle. В дополнение к полям родительского класса, в этом классе присутствует поле number\_of\_wheels, отвечающее за количество колес ТС, что необходимо для метода change\_wheels. Также, в этот класс был добавлен метод repair\_brakes, отвечающий за починку тормозов. Код этих методов представлен на листинге 6.5.

def \_change\_wheels(self):  
  
 *"""Метод для починки колес транспортного ТС. Наследуется дочерними классами класса WheeledVehicle"""* print("Changing the wheels...")  
 time.sleep(0.25)  
 for i in range(1, self.\_number\_of\_wheels + 1):  
 time.sleep(1)  
 print(f"Wheel {i} changed")  
 time.sleep(0.2)  
 print("Wheels changed\n")  
  
def \_repair\_brakes(self):  
  
 *"""Метод для ремонта тормозов транспортного ТС. Наследуется дочерними классами класса WheeledVehicle"""* print("Repairing the brakes...")  
 time.sleep(3)  
 time.sleep(0.2)  
 print("Brakes are repaired\n")

Листинг 6.5 – Методы change\_wheels и repair\_brakes

В этом классе метод move реализован полностью, в отличие от абстрактного. Он отображает часть проеханного пути и, с некоторой вероятностью, за каждый проеханный километр создает неисправность, которая определяется и устраняется. Код метода представлен на листинге 6.6.

def move(self, environment, travel\_length):  
  
 *"""Переопределенный метода для движения родительского класса"""* self.\_check\_env(environment)  
 time.sleep(0.5)  
 print("The wheeled vehicle is moving")  
 distance\_traveled = 0  
  
 """В зависимости от скорости ТС выбирается интервал изменения проеханного ТС расстояния.  
   
 Во время движения у ТС случайным образом возникают неисправности,   
   
 для исправления которых вызывается метод \_do\_service"""  
  
 while distance\_traveled < travel\_length:  
 distance\_traveled += 1  
 time.sleep(1 / self.speed)  
 print("\r", end="")  
 print(f"{distance\_traveled} / {travel\_length} km", end='')  
 if random() < 0.01:  
 if self.\_isMechanical():  
 malfunction = randint(1, 4)  
 else:  
 malfunction = randint(1, 2)  
 self.\_do\_service(malfunction)  
 print()  
 print("The wheeled vehicle has arrived to the final destination")

Листинг 6.6 – Метод WheeledVehicle.move

Метод WheeledVehicle.\_do\_service включает в себя действия для определения и починки неисправностей колес и тормозов. Для выявления и исправления проблем с топливом и маслом использует вызов родительского \_do\_service. Код приведен в листинге 6.7.

def \_do\_service(self, malfunction):  
 super().\_do\_service(malfunction) # Для заправки и смены масла вызывается родительский метод  
 if malfunction == 3:  
 print("\nWheel damage detected")  
 time.sleep(0.25)  
 self.\_change\_wheels()  
 elif malfunction == 4:  
 print("\nBrakes malfunction detected")  
 time.sleep(0.25)  
 self.\_repair\_brakes()

Листинг 6.7 – Метод WheeledVehicle.\_do\_service

Класс WaterVehicle также расширяет класс Vehicle. К его отличиям относится наличие метода close\_leak и расширенные методы move и \_do\_service. Также, значение \_environment для водного ТС по умолчанию установлено как “water”. Реализация класса представлена на листинге 6.8.

class WaterVehicle(Vehicle):  
  
 *"""Класс WaterVehicle дополняет класс Vehicle"""* def \_\_init\_\_(self, speed, mechanical):  
 super().\_\_init\_\_(speed, mechanical)  
 self.\_environment = "water" # У водного транспорта среда движения всегда вода  
 self.\_mechanical = mechanical  
 self.\_check\_attributes()  
  
 def \_close\_leak(self):  
  
 *"""Метод для закрытия течи на водном ТС"""* print("Closing the leak...")  
 time.sleep(1)  
 print("The leak is closed")  
 time.sleep(0.5)  
 print("Removing the water...")  
 time.sleep(2)  
 print("Water is removed\n")  
  
 def \_do\_service(self, malfunction):  
 super().\_do\_service(malfunction)  
 if malfunction == 3:  
 print("\nLeak detected")  
 time.sleep(0.25)  
 self.\_close\_leak()  
  
 def move(self, environment, travel\_length):  
 self.\_check\_env(environment)  
 time.sleep(0.5)  
 print("\nThe water vehicle is moving")  
 distance\_traveled = 0  
 while distance\_traveled < travel\_length:  
 distance\_traveled += 1  
 time.sleep(1 / self.speed)  
 print("\r", end="")  
 print(f"{distance\_traveled} / {travel\_length} km", end='')  
 if random() < 0.01:  
 if self.\_isMechanical():  
 malfunction = randint(1, 3)  
 else:  
 malfunction = randint(1, 2)  
 self.\_do\_service(malfunction)  
 print()  
 print("The water vehicle has arrived to the final destination")

Листинг 6.8 – Реализация класса WaterVehicle

Класс Car расширяет класс WheeledVehicle и описывает автомобиль. Класс Car содержит метод для починки фар repair\_lights, измененные методы move и \_do\_service. Кроме того, поля \_environment, \_mechanical и number\_of\_wheels имеют установленные по умолчанию значения, поэтому при инициализации автомобиля нужно указать только скорость. Реализация этого класса представлена на листинге 6.9.

class Car(WheeledVehicle):  
  
 *"""Класс Car дополняет класс WheeledVehicle"""* def \_\_init\_\_(self, speed):  
 super().\_\_init\_\_(speed, number\_of\_wheels=4, mechanical="mechanical") # Автомобиль является механическим ТС и имеет 4 колеса по умолчанию  
 self.\_check\_attributes()  
  
 def \_repair\_lights(self):  
  
 *"""Метод для починки фар автомобиля"""* print("Repairing headlights...")  
 time.sleep(3)  
 print("Headlights functioning\n")  
 time.sleep(0.5)  
 print("Repairing back lights...")  
 time.sleep(3)  
 print("Back lights functioning\n")  
  
 def \_do\_service(self, malfunction):  
 super().\_do\_service(malfunction)  
 if malfunction == 3:  
 print("\nLights malfunction detected")  
 self.\_repair\_lights()  
 print("\nCar service is finished")  
  
 def move(self, environment, travel\_length):  
 self.\_check\_env(environment)  
 time.sleep(0.5)  
 print("\nThe car is moving")  
 distance\_traveled = 0  
 while distance\_traveled < travel\_length:  
 distance\_traveled += 1  
 time.sleep(1 / self.speed)  
 print("\r", end="")  
 print(f"{distance\_traveled} / {travel\_length} km", end='')  
 if random() < 0.01:  
 malfunction = randint(1, 3)  
 self.\_do\_service(malfunction)  
 print()  
 print("The car has arrived to the final destination")

Листинг 6.9 – Реализация класса Car

UML – диаграмма классов для задания 6 представлена на рисунке 6.1.

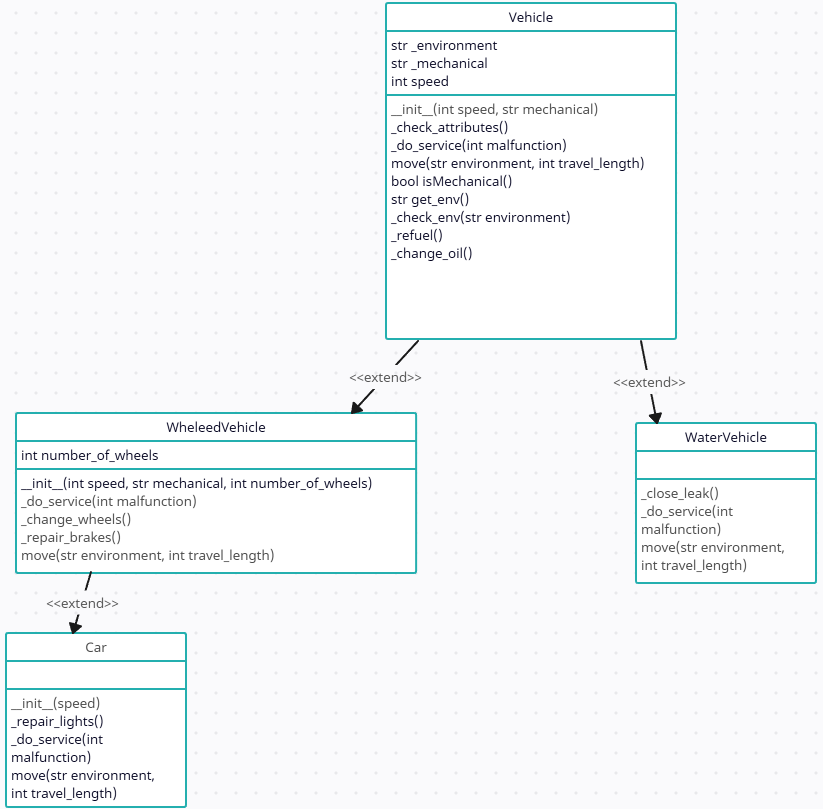


Рисунок 6.1 – UML – диаграмма иерархии классов транспортных средств

Для проверки функционала иерархии классов была создана программа main.py. Ее код представлен на листинге 6.10.

from Hierarchy import \*  
ship = WaterVehicle(80, "mechanical")  
print(ship.get\_env())  
ship.move("water", 300)  
bicycle = WheeledVehicle(30, "non-mechanical", 2)  
bicycle.move("ground", 100)  
car = Car(120)  
car.move("ground", 400)

Листинг 6.10 – main.py для задания 6

Результат работы программы показан на рисунке 6.2.

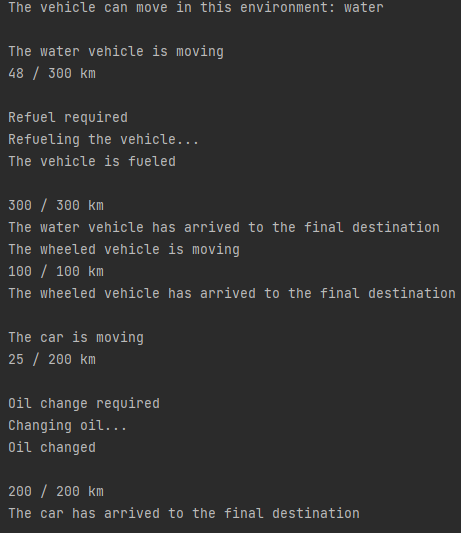


Рисунок 6.2 – Результат работы программы main.py для задания 6

**Вывод:** в ходе выполнения лабораторной работы я изучил реализацию объектно – ориентированной парадигмы программирования на языке программирования python, в частности механизм наследования и инкапсуляции.