МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
“БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

Кафедра интеллектуальных информационных технологий

Отчёт по лабораторной работе №3
Специальность ПО-11

Выполнил:
А. А. Билялова
студент группы ПО11

Проверил:
А. А. Крощенко,
ст. преп. кафедры ИИТ,
10.03.2025

Брест 2025

Цель работы: приобрести навыки применения паттернов проектирования при решении практических задач с использованием языка Python.

Общее задание. Прочитать задания, взятые из каждой группы, соответствующей одному из трех основных типов паттернов;

• Определить паттерн проектирования, который может использоваться при реализации задания. Пояснить свой выбор;

• Реализовать фрагмент программной системы, используя выбранный паттерн. Реализовать все необходимые дополнительные классы.

Задание 1. Проект «Бургер-закусочная». Реализовать возможность формирования заказа из определенных позиций (тип бургера (веганский, куриный и т.д.)), напиток (холодный – пепси, кока-кола и т.д.; горячий – кофе, чай и т.д.), тип упаковки – с собой, на месте. Должна формироваться итоговая стоимость заказа.

Выполнение:

Код программы:

```
class Burger:
    def __init__(self, name: str, price: float):
        self.name = name
        self.price = price

    def __str__(self):
        return f"{self.name} (${self.price})"

class VeganBurger(Burger):
    def __init__(self):
        super().__init__("Веганский бургер", 5.0)

class ChickenBurger(Burger):
    def __init__(self):
        super().__init__("Куриный бургер", 6.0)

class Drink:
    def __init__(self, name: str, price: float):
        self.name = name
        self.price = price

    def __str__(self):
        return f"{self.name} (${self.price})"


class ColdDrink(Drink):
    def __init__(self):
        super().__init__("Холодный напиток (Пепси)", 2.0)

class HotDrink(Drink):
    def __init__(self):
        super().__init__("Горячий напиток (Кофе)", 3.0)

class Packaging:
    def __init__(self, name: str, price: float):
        self.name = name
        self.price = price

    def __str__(self):
        return f"{self.name} (${self.price})"
```

```
class TakeawayPackaging(Packaging):
    def __init__(self):
        super().__init__("Упаковка с собой", 0.5)

class OnSitePackaging(Packaging):
    def __init__(self):
        super().__init__("Упаковка на месте", 0.0)

class Order:
    def __init__(self):
        self.burger = None
        self.drink = None
        self.packaging = None

    def set_burger(self, burger: Burger):
        self.burger = burger

    def set_drink(self, drink: Drink):
        self.drink = drink

    def set_packaging(self, packaging: Packaging):
        self.packaging = packaging

    def calculate_total(self) -> float:
        total = 0
        if self.burger:
            total += self.burger.price
        if self.drink:
            total += self.drink.price
        if self.packaging:
            total += self.packaging.price
        return total

    def __str__(self):
        return (f"Заказ:\n"
                f"  Бургер: {self.burger}\n"
                f"  Напиток: {self.drink}\n"
                f"  Упаковка: {self.packaging}\n"
                f"Итого: ${self.calculate_total()}")

class OrderBuilder:
    def __init__(self):
        self.order = Order()

    def add_burger(self, burger: Burger):
        self.order.set_burger(burger)
        return self

    def add_drink(self, drink: Drink):
        self.order.set_drink(drink)
        return self

    def add_packaging(self, packaging: Packaging):
        self.order.set_packaging(packaging)
        return self

    def build(self) -> Order:
        return self.order

if __name__ == "__main__":
    # Создаем строителя
    builder = OrderBuilder()

    # Формируем заказ
    order1 = (builder
              .add_burger(VeganBurger())
              .add_drink(ColdDrink())
              .add_packaging(TakeawayPackaging())
```

```
            .build())

    order2 = (builder
              .add_burger(ChickenBurger())
              .add_drink(HotDrink())
              .add_packaging(OnSitePackaging())
              .build())

    # Выводим информацию о заказах
    print("Заказ 1:")
    print(order1)

    print("\nЗаказ 2:")
    print(order2)
```

Рисунки с результатами работы программы

```
Заказ 1:
Заказ:
  Бургер: Куриный бургер ($6.0)
  Напиток: Горячий напиток (Кофе) ($3.0)
  Упаковка: Упаковка на месте ($0.0)
Итого: $9.0

Заказ 2:
Заказ:
  Бургер: Куриный бургер ($6.0)
  Напиток: Горячий напиток (Кофе) ($3.0)
  Упаковка: Упаковка на месте ($0.0)
Итого: $9.0
```

Задание 2. Проект «IT-компания». В проекте должен быть реализован класс «Сотрудник» с субординацией (т.е. должна быть возможность определения кому подчиняется сотрудник и кто находится в его подчинении). Для каждого сотрудника помимо сведений о субординации хранятся другие данные (ФИО, отдел, должность, зарплата). Предусмотреть возможность удаления и добавления сотрудника.

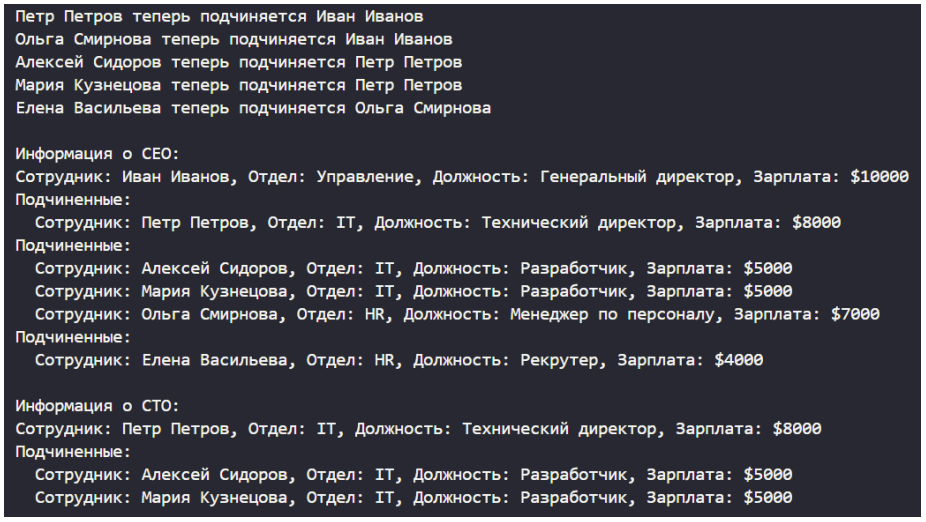
Выполнение:

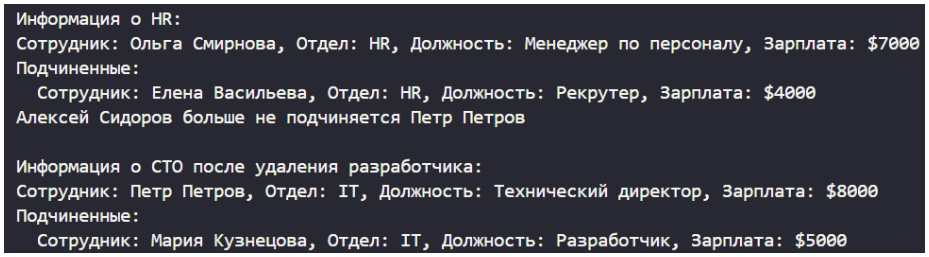
Код программы:

```
from typing import List

class Employee:
    def __init__(self, name: str, department: str, position: str, salary: float):
        self.name = name
        self.department = department
        self.position = position
        self.salary = salary

    def add_subordinate(self, employee):
        raise NotImplementedError("Этот метод должен быть переопределен в подклассе")

    def remove_subordinate(self, employee):
        raise NotImplementedError("Этот метод должен быть переопределен в подклассе")

    def get_subordinates(self) -> List['Employee']:
        raise NotImplementedError("Этот метод должен быть переопределен в подклассе")

    def __str__(self):
        return (f"Сотрудник: {self.name}, "
                f"Отдел: {self.department}, "
```

```
                f"Должность: {self.position}, "
                f"Зарплата: ${self.salary}")

class Manager(Employee):
    def __init__(self, name: str, department: str, position: str, salary: float):
        super().__init__(name, department, position, salary)
        self.subordinates: List[Employee] = []

    def add_subordinate(self, employee: 'Employee'):
        self.subordinates.append(employee)
        print(f"{employee.name} теперь подчиняется {self.name}")

    def remove_subordinate(self, employee: 'Employee'):
        if employee in self.subordinates:
            self.subordinates.remove(employee)
            print(f"{employee.name} больше не подчиняется {self.name}")
        else:
            print(f"{employee.name} не найден в подчинении у {self.name}")

    def get_subordinates(self) -> List['Employee']:
        return self.subordinates

    def __str__(self):
        subordinates_info = "\n  ".join([str(sub) for sub in self.subordinates])
        return (f"{super().__str__()}\n"
                f"Подчиненные:\n  {subordinates_info if subordinates_info else 'Нет
подчиненных'}")

class RegularEmployee(Employee):
    def __init__(self, name: str, department: str, position: str, salary: float):
        super().__init__(name, department, position, salary)

    def __str__(self):
        return super().__str__()

if __name__ == "__main__":
    # Создаем сотрудников
    ceo = Manager("Иван Иванов", "Управление", "Генеральный директор", 10000)
    cto = Manager("Петр Петров", "IT", "Технический директор", 8000)
    dev1 = RegularEmployee("Алексей Сидоров", "IT", "Разработчик", 5000)
    dev2 = RegularEmployee("Мария Кузнецова", "IT", "Разработчик", 5000)
    hr = Manager("Ольга Смирнова", "HR", "Менеджер по персоналу", 7000)
    recruiter = RegularEmployee("Елена Васильева", "HR", "Рекрутер", 4000)

    # Устанавливаем субординацию
    ceo.add_subordinate(cto)
    ceo.add_subordinate(hr)
    cto.add_subordinate(dev1)
    cto.add_subordinate(dev2)
    hr.add_subordinate(recruiter)

    # Выводим информацию о сотрудниках
    print("\nИнформация о CEO:")
    print(ceo)

    print("\nИнформация о CTO:")
    print(cto)

    print("\nИнформация о HR:")
    print(hr)

    # Удаляем подчиненного
    cto.remove_subordinate(dev1)
    print("\nИнформация о CTO после удаления разработчика:")
    print(cto)
```

Рисунки с результатами работы программы





Задание 3. Проект «Расчет зарплаты». Для задания, указанного во втором пункте («IT- компания») реализовать расчет зарплаты с выводом полного отчета.

Порядок вывода сотрудников в отчете – по старшинству для каждого отдела.

Выполнение:

Код программы:

```
from typing import List

class Employee:
    def __init__(self, name: str, department: str, position: str, salary: float):
        self.name = name
        self.department = department
        self.position = position
        self.salary = salary

    def accept(self, visitor):
        raise NotImplementedError("Этот метод должен быть переопределен в подклассе")

class Manager(Employee):
    def __init__(self, name: str, department: str, position: str, salary: float):
        super().__init__(name, department, position, salary)
        self.subordinates: List[Employee] = []

    def add_subordinate(self, employee: 'Employee'):
        self.subordinates.append(employee)

    def accept(self, visitor):
        visitor.visit_manager(self)
        for subordinate in self.subordinates:
            subordinate.accept(visitor)

class RegularEmployee(Employee):
    def __init__(self, name: str, department: str, position: str, salary: float):
        super().__init__(name, department, position, salary)

    def accept(self, visitor):
        visitor.visit_regular_employee(self)
```

```
class SalaryReportVisitor:
    def __init__(self):
        self.report = []

    def visit_manager(self, manager: Manager):
        self.report.append(f"Руководитель: {manager.name}, "
                           f"Отдел: {manager.department}, "
                           f"Должность: {manager.position}, "
                           f"Зарплата: ${manager.salary}")

    def visit_regular_employee(self, employee: RegularEmployee):
        self.report.append(f"Сотрудник: {employee.name}, "
                           f"Отдел: {employee.department}, "
                           f"Должность: {employee.position}, "
                           f"Зарплата: ${employee.salary}")

    def generate_report(self) -> str:
        return "\n".join(self.report)

if __name__ == "__main__":
    # Создаем сотрудников
    ceo = Manager("Иван Иванов", "Управление", "Генеральный директор", 10000)
    cto = Manager("Петр Петров", "IT", "Технический директор", 8000)
    dev1 = RegularEmployee("Алексей Сидоров", "IT", "Разработчик", 5000)
    dev2 = RegularEmployee("Мария Кузнецова", "IT", "Разработчик", 5000)
    hr = Manager("Ольга Смирнова", "HR", "Менеджер по персоналу", 7000)
    recruiter = RegularEmployee("Елена Васильева", "HR", "Рекрутер", 4000)

    # Устанавливаем субординацию
    ceo.add_subordinate(cto)
    ceo.add_subordinate(hr)
    cto.add_subordinate(dev1)
    cto.add_subordinate(dev2)
    hr.add_subordinate(recruiter)

    # Создаем посетителя для расчета зарплаты и формирования отчета
    salary_report_visitor = SalaryReportVisitor()

    # Применяем посетителя к CEO
    ceo.accept(salary_report_visitor)

    # Выводим отчет
    print("Отчет по зарплатам:")
    print(salary_report_visitor.generate_report())
```

Рисунки с результатами работы программы

```
Отчет по зарплатам:
Руководитель: Иван Иванов, Отдел: Управление, Должность: Генеральный директор, Зарплата: $10000
Руководитель: Петр Петров, Отдел: IT, Должность: Технический директор, Зарплата: $8000
Сотрудник: Алексей Сидоров, Отдел: IT, Должность: Разработчик, Зарплата: $5000
Сотрудник: Мария Кузнецова, Отдел: IT, Должность: Разработчик, Зарплата: $5000
Руководитель: Ольга Смирнова, Отдел: HR, Должность: Менеджер по персоналу, Зарплата: $7000
Сотрудник: Елена Васильева, Отдел: HR, Должность: Рекрутер, Зарплата: $4000
```

Вывод: приобрела навыки применения паттернов проектирования при решении практических задач с использованием языка Python.