Отчёт по лабораторной работе

Тема: Реализация принципов объектно-ориентированного программирования на Python

Сведения о студенте

Дата: 2025-11-17 **Семестр:** 2 курс 1 семестр **Группа:** ПИН-б-о-24-1 **Дисциплина:** Технологии программирования **Студент:** Губжоков Роман Русланович

Оглавление

1. Введение
2. Лабораторная работа 4.1: Инкапсуляция
3. Лабораторная работа 4.2: Наследование и абстракция
4. Лабораторная работа 4.3: Полиморфизм и магические методы
5. Лабораторная работа 4.4: Композиция и агрегация
6. Заключение

Введение

Цель работы

Разработка комплексной системы учета сотрудников компании с применением принципов объектно-ориентированного программирования: инкапсуляции, наследования, полиморфизма, композиции и агрегации.

Лабораторная работа 4.1: Инкапсуляция

Цель

Реализация базового класса `Employee` с инкапсуляцией данных и валидацией.

Выполненные задачи

- Создан класс `Employee` с приватными атрибутами (`__id` , `__name` , `__department` , `__base_salary`)
- Реализованы свойства (property) для доступа к данным с валидацией
- Добавлена валидация входных параметров (проверка типов и значений)

- Реализован метод `__str__` для строкового представления
- Реализованы абстрактные методы `calculate_salary()` и `get_info()`

Ключевые элементы реализации

```
class Employee:
    def __init__(self, id: int, name: str, department: str, base_salary: float):
        self.id = id
        self.name = name
        self.department = department
        self.base_salary = base_salary

    def __str__(self):
        return f"Сотрудник [id: {self.id}, имя: {self.name}, отдел: {self.department}, базова

    @property
    def id(self):
        return self.__id

    @id.setter
    def id(self, value):
        if not isinstance(value, int) or value <= 0:
            raise ValueError("ID должен быть положительным целым числом.")
        self.__id = value

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        if not isinstance(value, str) or value.strip() == "":
            raise ValueError("Имя не может быть пустой строкой.")
        self.__name = value

    @property
    def department(self):
        return self.__department

    @department.setter
    def department(self, value):
        if not isinstance(value, str):
            raise ValueError("Название отдела должно быть строкой.")
        self.__department = value

    @property
    def base_salary(self):
        return self.__base_salary

    @base_salary.setter
    def base_salary(self, value):
        if not isinstance(value, (int, float)):
```

```
            raise ValueError("Зарплата должна быть неотрицательным числом.")
        self.__base_salary = value
```

Пример использования

```
from lab0201employee_class import Employee

if __name__ == "__main__":
    emp = Employee(165, "Shipazu", "Отдел Дизайна", 0)
    print(emp.id)
    print(emp.name)
    print(emp.department)
    print(emp.base_salary)

    emp.department = "Графический Отдел"
    print(emp.department)

    try:
        emp.base_salary = -100500
    except ValueError as e:
        print(f"Error: {e}")

    emp.base_salary = 65.050
    print(emp.base_salary)

    print(emp)
```

Результат выполнения кода

```
165
Shipazu
Отдел Дизайна
0
Графический Отдел
65.05
Сотрудник [id: 165, имя: Shipazu, отдел: Графический Отдел, базовая зарплата: 65.05]
```

Результаты тестирования

- Протестирована корректная установка и получение значений через свойства
- Проверена обработка невалидных данных (отрицательные ID, пустые строки)
- Убедились в корректности строкового представления
- Проверена работа абстрактных методов

Лабораторная работа 4.2: Наследование и абстракция

Цель

Создание иерархии классов сотрудников на основе наследования и абстракции.

Выполненные задачи

- Создан абстрактный класс `AbstractEmployee` с абстрактными методами
- Реализованы классы-наследники: `Manager`, `Developer`, `Salesperson`
- Переопределены методы расчета зарплат для каждого типа сотрудника
- Реализована фабрика сотрудников `EmployeeFactory`
- Добавлены специфичные методы для каждого типа (например, `add_skill()` для Developer)

Abstract Employee

```
import json
from typing import Dict, Any
from abc import ABC, abstractmethod

class AbstractEmployee(ABC):

    def __init__(self, id: int, name: str, department: str, base_salary: float):
        self.id = id  # @id.setter
        self.name = name  # @name.setter
        self.department = department  # @department.setter
        self.base_salary = base_salary  # @base_salary.setter

    @abstractmethod
    def calculate_salary(self) -> float:
        pass

    @abstractmethod
    def get_info(self) -> str:
        pass



    @staticmethod
    def from_dict(data: dict) -> 'AbstractEmployee':
        class_name = data.get('type') or data.get('class_name')

        if class_name == 'Employee':
            from .lab0202_employee import Employee
            return Employee.from_dict(data)
        elif class_name == 'Manager':
            from .lab0202_manager import Manager
            return Manager.from_dict(data)
        elif class_name == 'Developer':
            from .lab0202_developer import Developer
            return Developer.from_dict(data)
        elif class_name == 'Salesperson':
            from .lab0202_salesperson import Salesperson
```

```
            return Salesperson.from_dict(data)
        else:
            raise ValueError(f"Неизвестный класс сотрудника: {class_name}")

    def __eq__(self, other) -> bool:
        if not isinstance(other, AbstractEmployee):
            return False
        return self.id == other.id

    def __lt__(self, other) -> bool:
        if not isinstance(other, AbstractEmployee):
            return NotImplemented
        return self.calculate_salary() < other.calculate_salary()

    def __add__(self, other) -> float:
        if isinstance(other, AbstractEmployee):
            return self.calculate_salary() + other.calculate_salary()
        elif isinstance(other, (int, float)):
            return self.calculate_salary() + other
        return NotImplemented

    def __radd__(self, other) -> float:
        if other == 0:
            return self.calculate_salary()
        elif isinstance(other, (int, float)):
            return other + self.calculate_salary()
        return NotImplemented

    def __str__(self) -> str:
        return f"{self.name} (ID: {self.id}, Отдел: {self.department})"

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(id={self.id}, name='{self.name}', salary={self.bas

    # === СВОЙСТВА ===

    @property
    def id(self) -> int:
        return self.__id

    @id.setter
    def id(self, value: int) -> None:
        if not isinstance(value, int) or value <= 0:
            raise ValueError("ID должен быть положительным целым числом.")
        self.__id = value

    @property
    def name(self) -> str:
        return self.__name

    @name.setter
    def name(self, value: str) -> None:
        if not isinstance(value, str) or value.strip() == "":
```

```
            raise ValueError("Имя не может быть пустой строкой.")
        self.__name = value.strip()

    @property
    def department(self) -> str:
        return self.__department

    @department.setter
    def department(self, value: str) -> None:
        if not isinstance(value, str) or value.strip() == "":
            raise ValueError("Название отдела не может быть пустой строкой.")
        self.__department = value.strip()

    @property
    def base_salary(self) -> float:
        return self.__base_salary

    @base_salary.setter
    def base_salary(self, value: float) -> None:
        if not isinstance(value, (int, float)):
            raise ValueError("Зарплата должна быть числом.")
        if value < 0:
            raise ValueError("Зарплата не может быть отрицательной.")
        self.__base_salary = float(value)

    @property
    def salary(self) -> float:
        return self.base_salary

    # === МЕТОД СЕРИАЛИЗАЦИИ ===

    def to_dict(self) -> Dict[str, Any]:
        """Полная сериализация сотрудника в словарь"""
        return {
            'type': self.__class__.__name__,
            'id': self.id,
            'name': self.name,
            'department': self.department,
            'base_salary': float(self.base_salary),
            'calculated_salary': float(self.calculate_salary())
        }

    @classmethod
    @abstractmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'AbstractEmployee':
        pass
```

Manager

```
from .lab0202_employee import Employee
from typing import Dict, Any
```

```
class Manager(Employee):

    def __init__(self,
                 id: int,
                 name: str,
                 department: str,
                 base_salary: float,
                 bonus: float):
        super().__init__(id, name, department, base_salary)
        self.bonus = bonus

    @property
    def bonus(self) -> float:
        return self.__bonus

    @bonus.setter
    def bonus(self, value: float) -> None:
        if not isinstance(value, (int, float)):
            raise ValueError("Бонус должен быть числом.")
        if value < 0:
            raise ValueError("Бонус должен быть неотрицательным числом.")
        self.__bonus = float(value)

    def get_info(self) -> str:
        return (f"Менеджер [id: {self.id}, имя: {self.name}, отдел: {self.department}, "
                f"базовая зарплата: {self.base_salary:.2f}, бонус: {self.bonus:.2f}, "
                f"итоговая зарплата: {self.calculate_salary():.2f}]")

    def calculate_salary(self) -> float:
        return self.base_salary + self.bonus

    def to_dict(self) -> Dict[str, Any]:
        data = super().to_dict()
        data['bonus'] = self.bonus
        return data

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Manager':
        id_val = int(data['id'])
        name_val = str(data['name'])
        department_val = data.get('department', '')

        base_salary_val = data.get('base_salary', data.get('salary', 0))
        base_salary_val = float(base_salary_val)

        bonus_val = data.get('bonus', 0)
        bonus_val = float(bonus_val)

        manager = cls(
            id=id_val,
            name=name_val,
            department=department_val,
            base_salary=base_salary_val,
```

```
            bonus=bonus_val
        )

        return manager

    def __str__(self) -> str:
        return f"Manager {self.name} (Bonus: {self.bonus:.2f})"

    def __repr__(self) -> str:
        return f"Manager(id={self.id}, name='{self.name}', department='{self.department}', bo
```

Developer

```
from .lab0202_employee import Employee
from typing import Dict, Any, List

class Developer(Employee):

    def __init__(self,
                 id: int,
                 name: str,
                 department: str,
                 base_salary: float,
                 tech_stack: List[str],
                 seniority_level: str):
        super().__init__(id, name, department, base_salary)
        self.tech_stack = tech_stack
        self.seniority_level = seniority_level

    @property
    def tech_stack(self) -> List[str]:
        return self.__tech_stack

    @tech_stack.setter
    def tech_stack(self, value: List[str]) -> None:
        if not isinstance(value, list):
            raise ValueError("Компетенции должны быть списком")
        if len(value) == 0:
            raise ValueError("Компетенции должны быть непустым списком")
        if not all(isinstance(item, str) for item in value):
            raise ValueError("Все элементы tech_stack должны быть строками")
        self.__tech_stack = [item.strip() for item in value if item.strip()]

    @property
    def seniority_level(self) -> str:
        return self.__seniority_level

    @seniority_level.setter
    def seniority_level(self, value: str) -> None:
        if not isinstance(value, str):
            raise ValueError("Уровень разработчика должен быть строкой")
        value = value.lower().strip()
```

```
        valid_levels = ["junior", "middle", "senior"]
        if value not in valid_levels:
            raise ValueError(f'Уровень разработчика должен быть одним из: {", ".join(valid_le
        self.__seniority_level = value

    @property
    def seniority_coefficient(self) -> float:
        table = {"junior": 1.0, "middle": 1.5, "senior": 2.0}
        return table[self.seniority_level]

    def __iter__(self):
        return iter(self.tech_stack)

    def add_skill(self, new_skill: str) -> None:
        if not isinstance(new_skill, str) or new_skill.strip() == "":
            raise ValueError("Технология должна быть непустой строкой")
        new_skill = new_skill.strip()
        if new_skill not in self.tech_stack:
            self.tech_stack.append(new_skill)

    def calculate_salary(self) -> float:
        return self.base_salary * self.seniority_coefficient

    def get_info(self) -> str:
        return (f"Разработчик [id: {self.id}, имя: {self.name}, отдел: {self.department}, "
                f"базовая зарплата: {self.base_salary}, компетенции: {', '.join(self.tech_sta
                f"уровень: {self.seniority_level}, итоговая зарплата: {self.calculate_salary(

    def to_dict(self) -> Dict[str, Any]:
        """Полная сериализация Developer в словарь"""
        data = super().to_dict()
        data.update({
            'tech_stack': self.tech_stack.copy(),
            'seniority_level': self.seniority_level,
            'seniority_coefficient': self.seniority_coefficient
        })
        return data

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Developer':
        """Десериализует Developer из словаря"""
        id_val = data['id']
        name_val = data['name']
        department_val = data.get('department', '')
        base_salary_val = data.get('base_salary', data.get('salary', 0))

        tech_stack_val = data.get('tech_stack', [])
        seniority_level_val = data.get('seniority_level', 'junior')  # Значение по умолчанию

        developer = cls(
            id=id_val,
            name=name_val,
            department=department_val,
```

```
            base_salary=base_salary_val,
            tech_stack=tech_stack_val,
            seniority_level=seniority_level_val
        )

        return developer

    def __str__(self) -> str:
        return f"Developer {self.name} ({self.seniority_level}) - {', '.join(self.tech_stack[

    def __repr__(self) -> str:
        return f"Developer(id={self.id}, name='{self.name}', level='{self.seniority_level}',
```

Salesperson

```
from .lab0202_employee import Employee
from typing import Dict, Any

class Salesperson(Employee):

    def __init__(self,
                 id: int,
                 name: str,
                 department: str,
                 base_salary: float,
                 commission_rate: float,
                 sales_volume: float):
        super().__init__(id, name, department, base_salary)
        self.commission_rate = commission_rate
        self.sales_volume = sales_volume

    @property
    def commission_rate(self) -> float:
        return self.__commission_rate

    @commission_rate.setter
    def commission_rate(self, value: float) -> None:
        if not isinstance(value, (int, float)):
            raise ValueError("Процент комиссии должен быть числом.")
        value = float(value)
        if value < 0 or value > 1:
            raise ValueError("Процент комиссии должен быть в диапазоне от 0 до 1 включительно
        self.__commission_rate = value

    @property
    def sales_volume(self) -> float:
        return self.__sales_volume

    @sales_volume.setter
    def sales_volume(self, value: float) -> None:
        if not isinstance(value, (int, float)):
            raise ValueError("Объем продаж должен быть числом.")
```

```
        value = float(value)
        if value < 0:
            raise ValueError("Объем продаж должен быть неотрицательным числом.")
        self.__sales_volume = value

    def update_sales(self, amount: float) -> None:
        if not isinstance(amount, (int, float)):
            raise ValueError("Изменение объёма продаж должно быть числом.")
        new_volume = self.sales_volume + amount
        if new_volume < 0:
            raise ValueError(
                f"Объём продаж не может быть отрицательным. "
                f"Текущий: {self.sales_volume:.2f}, изменение: {amount:.2f}"
            )
        self.sales_volume = new_volume

    def calculate_salary(self) -> float:
        return self.base_salary + (self.sales_volume * self.commission_rate)

    def get_info(self) -> str:
        return (f"Продавец [id: {self.id}, имя: {self.name}, отдел: {self.department}, "
                f"базовая зарплата: {self.base_salary:.2f}, процент комиссии: {self.commissio
                f"объём продаж: {self.sales_volume:.2f}, итоговая зарплата: {self.calculate_s

    def to_dict(self) -> Dict[str, Any]:
        data = super().to_dict()
        data.update({
            'commission_rate': self.commission_rate,
            'sales_volume': self.sales_volume,
            'commission_earned': self.sales_volume * self.commission_rate  # Добавляем для ин
        })
        return data

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Salesperson':
        id_val = int(data['id'])
        name_val = str(data['name'])
        department_val = data.get('department', '')

        base_salary_val = data.get('base_salary', data.get('salary', 0))
        base_salary_val = float(base_salary_val)

        commission_rate_val = data.get('commission_rate', 0.0)
        commission_rate_val = float(commission_rate_val)

        sales_volume_val = data.get('sales_volume', 0.0)
        sales_volume_val = float(sales_volume_val)

        salesperson = cls(
            id=id_val,
            name=name_val,
            department=department_val,
            base_salary=base_salary_val,
```

```
            commission_rate=commission_rate_val,
            sales_volume=sales_volume_val
        )

        return salesperson

    def __str__(self) -> str:
        return f"Salesperson {self.name} (Sales: {self.sales_volume:.2f}, Commission: {self.c

    def __repr__(self) -> str:
        return (f"Salesperson(id={self.id}, name='{self.name}', "
                f"commission={self.commission_rate}, sales={self.sales_volume})")
```

Employee Factory

```
from .lab0202_employee import Employee
from .lab0202_manager import Manager
from .lab0202_developer import Developer
from .lab0202_salesperson import Salesperson

class EmployeeFactory:

    @staticmethod
    def create_employee(emp_type: str, **kwargs) -> Employee:
        emp_type = emp_type.lower()
        if emp_type == "employee":
            return EmployeeFactory._create_employee(**kwargs)
        elif emp_type == "manager":
            return EmployeeFactory._create_manager(**kwargs)
        elif emp_type == "developer":
            return EmployeeFactory._create_developer(**kwargs)
        elif emp_type == "salesperson":
            return EmployeeFactory._create_salesperson(**kwargs)
        else:
            raise ValueError(f'Неизвестный тип сотрудника: {emp_type}.'
                             f'Доступные типы: employee, manager, developer, salesperson')

    @staticmethod
    def _create_employee(**kwargs) -> Employee:
        params = ['id', 'name', 'department', 'base_salary']
        EmployeeFactory._check_params(params, kwargs)

        return Employee(
            id = kwargs['id'],
            name = kwargs['name'],
            department = kwargs['department'],
            base_salary = kwargs['base_salary']
            )

    @staticmethod
    def _create_manager(**kwargs) -> Manager:
        params = ['id', 'name', 'department', 'base_salary', 'bonus']
```

```
        EmployeeFactory._check_params(params, kwargs)

        return Manager(
            id = kwargs['id'],
            name = kwargs['name'],
            department = kwargs['department'],
            base_salary = kwargs['base_salary'],
            bonus = kwargs['bonus']
        )

    @staticmethod
    def _create_developer(**kwargs) -> Developer:
        params = ['id', 'name', 'department', 'base_salary', 'tech_stack', 'seniority_level']
        EmployeeFactory._check_params(params, kwargs)

        return Developer(
            id = kwargs['id'],
            name = kwargs['name'],
            department = kwargs['department'],
            base_salary = kwargs['base_salary'],
            tech_stack = kwargs['tech_stack'],
            seniority_level=kwargs['seniority_level']
        )

    @staticmethod
    def _create_salesperson(**kwargs) -> Salesperson:
        params = ['id', 'name', 'department', 'base_salary', 'commission_rate', 'sales_volume
        EmployeeFactory._check_params(params, kwargs)

        return Salesperson(
            id = kwargs['id'],
            name = kwargs['name'],
            department = kwargs['department'],
            base_salary = kwargs['base_salary'],
            commission_rate = kwargs['commission_rate'],
            sales_volume = kwargs['sales_volume']
        )

    @staticmethod
    def _check_params(req_params: list, params: dict):
        missing = [p for p in req_params if p not in params]
        if missing:
            raise ValueError(f"Отсутствуют обязательные параметры: {', '.join(missing)}")
```

Пример использования

```
from .lab0202_employee_factory import EmployeeFactory

def main():
    print("=" * 70)
    print("ДЕМОНСТРАЦИЯ РАБОТЫ СИСТЕМЫ СОТРУДНИКОВ")
    print("=" * 70)
```

```
# 1. Создание экземпляров каждого типа сотрудника через фабрику
print("\n1. СОЗДАНИЕ СОТРУДНИКОВ ЧЕРЕЗ ФАБРИКУ")
print("-" * 50)

try:
    # Создаем сотрудников разных типов
    employee = EmployeeFactory.create_employee(
        "employee",
        id=100,
        name="Иван Иванов",
        department="Общий отдел",
        base_salary=50000
    )

    manager = EmployeeFactory.create_employee(
        "manager",
        id=200,
        name="Петр Петров",
        department="Отдел управления",
        base_salary=80000,
        bonus=20000
    )

    developer = EmployeeFactory.create_employee(
        "developer",
        id=300,
        name="Анна Программистова",
        department="IT отдел",
        base_salary=70000,
        tech_stack=["Python", "Django", "PostgreSQL"],
        seniority_level="senior"
    )

    salesperson = EmployeeFactory.create_employee(
        "salesperson",
        id=400,
        name="Мария Продажникова",
        department="Отдел продаж",
        base_salary=40000,
        commission_rate=0.1,
        sales_volume=500000
    )

    print("Все сотрудники успешно созданы через фабрику")

except ValueError as e:
    print(f" Ошибка при создании сотрудников: {e}")
    return

# 2. Демонстрация работы с каждым экземпляром
print("\n2. ДЕМОНСТРАЦИЯ РАБОТЫ С КАЖДЫМ СОТРУДНИКОМ")
print("-" * 50)
```

```
# Для каждого сотрудника показываем calculate_salary() и get_info()
employees_list = [employee, manager, developer, salesperson]

for emp in employees_list:
    print(f"\n--- {emp.__class__.__name__} ---")
    print(f"calculate_salary(): {emp.calculate_salary():.2f} руб.")
    print(f"get_info(): {emp.get_info()}")

# 3. Демонстрация работы сеттеров
print("\n3. ДЕМОНСТРАЦИЯ РАБОТЫ СЕТТЕРОВ")
print("-" * 50)

# Меняем данные через сеттеры
print("До изменения:")
print(f"Менеджер: {manager.get_info()}")

manager.bonus = 30000  # Увеличиваем бонус
manager.department = "Высшее руководство"

print("\nПосле изменения бонуса и отдела:")
print(f"Менеджер: {manager.get_info()}")

# 4. Демонстрация полиморфизма
print("\n4. ДЕМОНСТРАЦИЯ ПОЛИМОРФИЗМА")
print("-" * 50)

print("Итерация по списку разных типов сотрудников:")
all_employees = [employee, manager, developer, salesperson]

for i, emp in enumerate(all_employees, 1):
    print(f"{i}. {emp.get_info()}")

# 5. Создание дополнительных сотрудников через фабрику
print("\n5. ДОПОЛНИТЕЛЬНОЕ СОЗДАНИЕ ЧЕРЕЗ ФАБРИКУ")
print("-" * 50)

try:
    # Создаем еще сотрудников разных типов
    junior_dev = EmployeeFactory.create_employee(
        "developer",
        id=501,
        name="Сергей Начинающий",
        department="IT отдел",
        base_salary=40000,
        tech_stack=["Python"],
        seniority_level="junior"
    )

    successful_sales = EmployeeFactory.create_employee(
        "salesperson",
        id=502,
        name="Ольга Успешная",
```

```
            department="Отдел продаж",
            base_salary=45000,
            commission_rate=0.15,
            sales_volume=1000000
        )

        print("Дополнительно созданные сотрудники:")
        print(f"- {junior_dev.get_info()}")
        print(f"- {successful_sales.get_info()}")

        # Добавляем в общий список
        all_employees.extend([junior_dev, successful_sales])

    except ValueError as e:
        print(f"Ошибка при создании дополнительных сотрудников: {e}")

    # 6. Итоговая демонстрация полиморфного поведения
    print("\n6. ИТОГОВАЯ ДЕМОНСТРАЦИЯ ПОЛИМОРФИЗМА")
    print("-" * 50)

    print(f"Всего сотрудников в системе: {len(all_employees)}")
    print("\nПолная информация о всех сотрудниках:")

    for i, emp in enumerate(all_employees, 1):
        print(f"\n{i}. {emp.get_info()}")
        print(f"   Расчитанная зарплата: {emp.calculate_salary():.2f} руб.")

    # 7. Демонстрация обработки ошибок фабрики
    print("\n7. ДЕМОНСТРАЦИЯ ОБРАБОТКИ ОШИБОК")
    print("-" * 50)

    try:
        # Неизвестный тип сотрудника
        invalid_emp = EmployeeFactory.create_employee("director", id=999, name="Тест")
    except ValueError as e:
        print(f" Корректная обработка неизвестного типа: {e}")

    try:
        # Недостаточно параметров
        incomplete_emp = EmployeeFactory.create_employee("employee", id=999, name="Тест")
    except ValueError as e:
        print(f" Корректная обработка недостающих параметров: {e}")

    print("\n" + "=" * 70)
    print("ДЕМОНСТРАЦИЯ ЗАВЕРШЕНА УСПЕШНО!")
    print("=" * 70)

if __name__ == "__main__":
    main()
```

Результат выполнения кода

```
======================================================================
ДЕМОНСТРАЦИЯ РАБОТЫ СИСТЕМЫ СОТРУДНИКОВ
======================================================================

1. СОЗДАНИЕ СОТРУДНИКОВ ЧЕРЕЗ ФАБРИКУ
--------------------------------------------------

Все сотрудники успешно созданы через фабрику

2. ДЕМОНСТРАЦИЯ РАБОТЫ С КАЖДЫМ СОТРУДНИКОМ
--------------------------------------------------

--- Employee ---
calculate_salary(): 50000.00 руб.
get_info(): Сотрудник [id: 100, имя: Иван Иванов, отдел: Общий отдел, базовая зарплата: 50000

--- Manager ---
calculate_salary(): 100000.00 руб.
get_info(): Менеджер [id: 200, имя: Петр Петров, отдел: Отдел управления, базовая зарплата: 8

--- Developer ---
calculate_salary(): 140000.00 руб.
get_info(): Разработчик [id: 300, имя: Анна Программистова, отдел: IT отдел, базовая зарплата

--- Salesperson ---
calculate_salary(): 90000.00 руб.
get_info(): Продавец [id: 400, имя: Мария Продажникова, отдел: Отдел продаж, базовая зарплата

3. ДЕМОНСТРАЦИЯ РАБОТЫ СЕТТЕРОВ
--------------------------------------------------

До изменения:
Менеджер: Менеджер [id: 200, имя: Петр Петров, отдел: Отдел управления, базовая зарплата: 800

После изменения бонуса и отдела:
Менеджер: Менеджер [id: 200, имя: Петр Петров, отдел: Высшее руководство, базовая зарплата: 8

4. ДЕМОНСТРАЦИЯ ПОЛИМОРФИЗМА
--------------------------------------------------

Итерация по списку разных типов сотрудников:
1. Сотрудник [id: 100, имя: Иван Иванов, отдел: Общий отдел, базовая зарплата: 50000]
2. Менеджер [id: 200, имя: Петр Петров, отдел: Высшее руководство, базовая зарплата: 80000, б
3. Разработчик [id: 300, имя: Анна Программистова, отдел: IT отдел, базовая зарплата: 70000,
4. Продавец [id: 400, имя: Мария Продажникова, отдел: Отдел продаж, базовая зарплата: 40000.0

5. ДОПОЛНИТЕЛЬНОЕ СОЗДАНИЕ ЧЕРЕЗ ФАБРИКУ
--------------------------------------------------

Дополнительно созданные сотрудники:
- Разработчик [id: 501, имя: Сергей Начинающий, отдел: IT отдел, базовая зарплата: 40000, ком
- Продавец [id: 502, имя: Ольга Успешная, отдел: Отдел продаж, базовая зарплата: 45000.00, пр
```

```
6. ИТОГОВАЯ ДЕМОНСТРАЦИЯ ПОЛИМОРФИЗМА
--------------------------------------------------

Всего сотрудников в системе: 6

Полная информация о всех сотрудниках:

1. Сотрудник [id: 100, имя: Иван Иванов, отдел: Общий отдел, базовая зарплата: 50000]
   Расчитанная зарплата: 50000.00 руб.

2. Менеджер [id: 200, имя: Петр Петров, отдел: Высшее руководство, базовая зарплата: 80000, б
   Расчитанная зарплата: 110000.00 руб.

3. Разработчик [id: 300, имя: Анна Программистова, отдел: IT отдел, базовая зарплата: 70000,
   Расчитанная зарплата: 140000.00 руб.

4. Продавец [id: 400, имя: Мария Продажникова, отдел: Отдел продаж, базовая зарплата: 40000.0
   Расчитанная зарплата: 90000.00 руб.

5. Разработчик [id: 501, имя: Сергей Начинающий, отдел: IT отдел, базовая зарплата: 40000, ко
   Расчитанная зарплата: 40000.00 руб.

6. Продавец [id: 502, имя: Ольга Успешная, отдел: Отдел продаж, базовая зарплата: 45000.00, п
   Расчитанная зарплата: 195000.00 руб.

7. ДЕМОНСТРАЦИЯ ОБРАБОТКИ ОШИБОК
--------------------------------------------------

 Корректная обработка неизвестного типа: Неизвестный тип сотрудника: director. Доступные типы
 Корректная обработка недостающих параметров: Недостаточно параметров для создания сотрудника

======================================================================
ДЕМОНСТРАЦИЯ ЗАВЕРШЕНА УСПЕШНО!
======================================================================
```

Результаты тестирования

- Все классы корректно наследуются от `AbstractEmployee`
- Абстрактные методы реализованы во всех классах
- Полиморфизм работает корректно в коллекциях
- Фабрика создает объекты правильных типов

Лабораторная работа 4.3: Полиморфизм и магические методы

Цель

Реализация полиморфного поведения объектов различных классов сотрудников. Освоение перегрузки операторов и использования магических методов.

Выполненные задачи

- Создан класс `Department` для управления сотрудниками
- Реализованы магические методы для сотрудников: `__eq__`, `__lt__`, `__add__`, `__radd__`
- Реализованы магические методы для отдела: `__len__`, `__getitem__`, `__contains__`, `__iter__`
- Реализована итерация по объектам (отдел и стек технологий разработчика)
- Созданы компараторы для сортировки сотрудников

Department

```
import sys
import json
import os
from typing import Optional, List, Dict, Any

current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)
if parent_dir not in sys.path:
    sys.path.insert(0, parent_dir)

from lab0202.lab0202_abstract_employee import AbstractEmployee

class Department:

    def __init__(self, name: str):
        if not name or not isinstance(name, str) or name.strip() == "":
            raise ValueError("Название отдела не может быть пустой строкой")
        self.name = name.strip()
        self.emp_list: List[AbstractEmployee] = []

    def __iter__(self):
        return iter(self.emp_list)

    def __len__(self) -> int:
        return len(self.emp_list)

    def __getitem__(self, key) -> AbstractEmployee:
        if isinstance(key, int):
            return self.emp_list[key]
        elif isinstance(key, slice):
            return self.emp_list[key]
        else:
            raise TypeError("Индекс должен быть целым числом или срезом")

    def __contains__(self, employee: AbstractEmployee) -> bool:
        if not isinstance(employee, AbstractEmployee):
            return False
```

```
        return any(emp.id == employee.id for emp in self.emp_list)

    def __str__(self) -> str:
        return f"Department '{self.name}' ({len(self)} сотрудников)"

    def __repr__(self) -> str:
        return f"Department(name='{self.name}', employees={len(self)})"

    def add_employee(self, employee: AbstractEmployee) -> None:
        """Добавить сотрудника в отдел"""
        if not isinstance(employee, AbstractEmployee):
            raise TypeError("Можно добавлять только объекты AbstractEmployee")

        # Проверяем уникальность ID
        if any(emp.id == employee.id for emp in self.emp_list):
            raise ValueError(
                f'Сотрудник {employee.name} (ID: {employee.id}) '
                f'уже находится в отделе "{self.name}"'
            )

        # Устанавливаем отдел сотруднику
        employee.department = self.name
        self.emp_list.append(employee)

    def remove_employee(self, employee_id: int) -> None:
        """Удалить сотрудника по ID"""
        initial_count = len(self.emp_list)
        employee_to_remove = None

        # Находим сотрудника
        for emp in self.emp_list:
            if emp.id == employee_id:
                employee_to_remove = emp
                break

        if employee_to_remove:
            self.emp_list.remove(employee_to_remove)
            print(f"Сотрудник {employee_to_remove.name} (ID: {employee_id}) удален из отдела
        else:
            raise ValueError(f'В отделе "{self.name}" нет сотрудника с ID = {employee_id}')

    def get_employees(self) -> List[Dict[str, Any]]:
        """Получить список сотрудников в виде словарей"""
        return [
            {
                "Name": emp.name,
                "ID": emp.id,
                "Department": emp.department,
                "Base Salary": emp.base_salary,
                "Calculated Salary": emp.calculate_salary(),
                "Type": emp.__class__.__name__
            }
            for emp in self.emp_list
```

```
    ]

def calculate_total_salary(self) -> float:
    """Рассчитать общую зарплату всех сотрудников отдела"""
    return sum(emp.calculate_salary() for emp in self.emp_list)

def get_employee_count(self) -> Dict[str, int]:
    """Получить статистику по типам сотрудников"""
    count_dict = {}
    for emp in self.emp_list:
        emp_type = emp.__class__.__name__
        count_dict[emp_type] = count_dict.get(emp_type, 0) + 1
    return count_dict

def find_employee_by_id(self, employee_id: int) -> Optional[AbstractEmployee]:
    """Найти сотрудника по ID"""
    for emp in self.emp_list:
        if emp.id == employee_id:
            return emp
    return None

def save_to_file(self, filename: str) -> None:
    """Сохранить отдел в файл"""
    data = self.to_dict()

    # Создаем директорию если не существует
    os.makedirs(os.path.dirname(os.path.abspath(filename)), exist_ok=True)

    with open(filename, 'w', encoding='utf-8') as f:
        json.dump(data, f, ensure_ascii=False, indent=2)

    print(f"Отдел '{self.name}' сохранен в файл: {filename}")

@classmethod
def load_from_file(cls, filename: str) -> 'Department':
    """Загрузить отдел из файла"""
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            data = json.load(f)

        return cls.from_dict(data)

    except FileNotFoundError:
        raise FileNotFoundError(f"Файл {filename} не найден")
    except KeyError as e:
        raise ValueError(f"Некорректный формат файла: отсутствует ключ {e}")
    except json.JSONDecodeError as e:
        raise ValueError(f"Некорректный JSON формат: {e}")
    except Exception as e:
        raise ValueError(f"Ошибка при загрузке файла: {e}")

def to_dict(self) -> Dict[str, Any]:
    """Полная сериализация отдела в словарь"""
```

```
        return {
            'type': 'Department',
            'name': self.name,
            'employee_count': len(self.emp_list),
            'total_salary': self.calculate_total_salary(),
            'employees': [emp.to_dict() for emp in self.emp_list]
        }

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Department':
        """Десериализация отдела из словаря"""
        if 'type' not in data or data['type'] != 'Department':
            raise ValueError("Некорректный формат данных отдела")

        if 'name' not in data:
            raise ValueError("Отсутствует название отдела")

        department = cls(data['name'])

        # Импортируем здесь, чтобы избежать циклических импортов
        from lab0202.lab0202_employee import Employee
        from lab0202.lab0202_manager import Manager
        from lab0202.lab0202_developer import Developer
        from lab0202.lab0202_salesperson import Salesperson

        employee_classes = {
            'Employee': Employee,
            'Manager': Manager,
            'Developer': Developer,
            'Salesperson': Salesperson
        }

        # Восстанавливаем сотрудников
        employees_data = data.get('employees', [])
        successful = 0
        failed = 0

        for emp_data in employees_data:
            try:
                emp_type = emp_data.get('type', 'Employee')

                if emp_type in employee_classes:
                    employee_class = employee_classes[emp_type]
                    employee = employee_class.from_dict(emp_data)
                    department.add_employee(employee)
                    successful += 1
                else:
                    print(f"⚠ Неизвестный тип сотрудника: {emp_type}")
                    failed += 1
            except Exception as e:
                print(f"⚠ Ошибка создания сотрудника: {e}")
                failed += 1
```

```
        if failed > 0:
            print(f"⚠ Успешно загружено: {successful}, не удалось: {failed}")

        return department

    def get_statistics(self) -> Dict[str, Any]:
        """Получить детальную статистику отдела"""
        total_salary = self.calculate_total_salary()
        avg_salary = total_salary / len(self.emp_list) if self.emp_list else 0

        return {
            'department_name': self.name,
            'total_employees': len(self.emp_list),
            'employee_types': self.get_employee_count(),
            'total_monthly_salary': total_salary,
            'average_salary': avg_salary,
            'min_salary': min((emp.calculate_salary() for emp in self.emp_list), default=0),
            'max_salary': max((emp.calculate_salary() for emp in self.emp_list), default=0)
        }

    def clear(self) -> None:
        """Очистить отдел (удалить всех сотрудников)"""
        count = len(self.emp_list)
        self.emp_list.clear()
        print(f"Отдел '{self.name}' очищен. Удалено сотрудников: {count}")

    def transfer_employee_to(self, employee_id: int, target_department: 'Department') -> None
        """Перевести сотрудника в другой отдел"""
        employee = self.find_employee_by_id(employee_id)
        if not employee:
            raise ValueError(f"Сотрудник с ID {employee_id} не найден в отделе '{self.name}'"

        # Удаляем из текущего отдела
        self.remove_employee(employee_id)

        # Добавляем в целевой отдел
        try:
            target_department.add_employee(employee)
            print(f"Сотрудник {employee.name} переведен из '{self.name}' в '{target_departmen
        except Exception as e:
            # Если не удалось добавить в целевой отдел, возвращаем обратно
            self.add_employee(employee)
            raise ValueError(f"Не удалось перевести сотрудника: {e}")
```

Примеры реализации

```
# lab0205_demo.py
# Демонстрация и тестирование всех возможностей системы сотрудников и отделов

import sys
import os
import functools
```

```python
# Добавляем пути для импорта
current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)
sys.path.insert(0, parent_dir)

from lab0202.lab0202_abstract_employee import AbstractEmployee
from lab0202.lab0202_employee import Employee
from lab0202.lab0202_manager import Manager
from lab0202.lab0202_developer import Developer
from lab0202.lab0202_salesperson import Salesperson
from lab0203.lab0203_department import Department

# Функции-компараторы для сортировки
def compare_by_name(emp1: AbstractEmployee, emp2: AbstractEmployee) -> int:
    """Компаратор для сортировки по имени (алфавитный порядок)"""
    if emp1.name < emp2.name:
        return -1
    elif emp1.name > emp2.name:
        return 1
    else:
        return 0

def compare_by_salary(emp1: AbstractEmployee, emp2: AbstractEmployee) -> int:
    """Компаратор для сортировки по зарплате (по убыванию)"""
    salary1 = emp1.calculate_salary()
    salary2 = emp2.calculate_salary()
    if salary1 > salary2:
        return -1
    elif salary1 < salary2:
        return 1
    else:
        return 0

def compare_by_department_then_name(emp1: AbstractEmployee, emp2: AbstractEmployee) -> int:
    """Компаратор для сортировки по отделу, а затем по имени"""
    if emp1.department < emp2.department:
        return -1
    elif emp1.department > emp2.department:
        return 1
    else:
        return compare_by_name(emp1, emp2)

def main():
    """Основная демонстрационная функция"""

    print("=" * 70)
    print("ДЕМОНСТРАЦИЯ СИСТЕМЫ СОТРУДНИКОВ И ОТДЕЛОВ")
    print("=" * 70)

    # 1. СОЗДАНИЕ ОТДЕЛА И ДОБАВЛЕНИЕ СОТРУДНИКОВ
    print("\n1. СОЗДАНИЕ ОТДЕЛА И ДОБАВЛЕНИЕ СОТРУДНИКОВ")
    print("-" * 50)
```

```
# Создаем отдел
it_department = Department("IT Департамент")
print(f"Создан отдел: {it_department}")

# Создаем сотрудников разных типов
employee1 = Employee(id=100, name="Иван Иванов", department="", base_salary=50000)
employee2 = Employee(id=101, name="Анна Петрова", department="", base_salary=45000)
manager1 = Manager(id=102, name="Петр Сидоров", department="", base_salary=70000, bonus=1
developer1 = Developer(id=103, name="Дмитрий Орлов", department="", base_salary=60000,
                       tech_stack=["Python", "Django", "PostgreSQL"], seniority_level="mid
developer2 = Developer(id=104, name="Екатерина Белова", department="", base_salary=65000,
                       tech_stack=["Java", "Spring", "Hibernate"], seniority_level="senior
salesperson1 = Salesperson(id=105, name="Мария Смирнова", department="", base_salary=4000
                           commission_rate=0.1, sales_volume=500000)

# Добавляем сотрудников в отдел
employees = [employee1, employee2, manager1, developer1, developer2, salesperson1]
for emp in employees:
    it_department.add_employee(emp)
    print(f"Добавлен: {emp.name}")

print(f"\nВ отделе теперь {len(it_department)} сотрудников")

# 2. ВЫЗОВ calculate_total_salary() ДЛЯ ОТДЕЛА
print("\n2. РАСЧЕТ ОБЩЕЙ ЗАРПЛАТЫ ОТДЕЛА")
print("-" * 50)

total_salary = it_department.calculate_total_salary()
print(f"Общая зарплата отдела '{it_department.name}': {total_salary:.2f} руб.")

# 3. ИСПОЛЬЗОВАНИЕ ПЕРЕГРУЖЕННЫХ ОПЕРАТОРОВ
print("\n3. ИСПОЛЬЗОВАНИЕ ПЕРЕГРУЖЕННЫХ ОПЕРАТОРОВ")
print("-" * 50)

# 3.1 Сравнение двух сотрудников (==, <)
print("\na) Сравнение сотрудников:")
print(f"employee1 == employee2: {employee1 == employee2}")
print(f"employee1 == employee1: {employee1 == employee1}")
print(f"employee1 < manager1 (по зарплате): {employee1 < manager1}")
print(f"manager1 < developer1 (по зарплате): {manager1 < developer1}")

# 3.2 Суммирование зарплат сотрудников (employee1 + employee2)
print("\nб) Суммирование зарплат:")
sum_salaries = employee1 + employee2
print(f"employee1 + employee2 = {sum_salaries:.2f}")
sum_complex = manager1 + developer1 + salesperson1
print(f"manager1 + developer1 + salesperson1 = {sum_complex:.2f}")

# 3.3 Суммирование списка сотрудников через sum()
print("\nв) Суммирование списка сотрудников:")
total_via_sum = sum(employees)
print(f"sum(employees) = {total_via_sum:.2f}")
```

```
print(f"Проверка: совпадает с общей зарплатой? {abs(total_via_sum - total_salary) < 0.01}")

# 3.4 Проверка вхождения сотрудника в отдел (in)
print("\nг) Проверка вхождения сотрудника в отдел:")
print(f"employee1 в отделе: {employee1 in it_department}")
print(f"manager1 в отделе: {manager1 in it_department}")

# Создаем сотрудника, которого нет в отделе
outsider = Employee(id=999, name="Чужой", department="", base_salary=10000)
print(f"outsider в отделе: {outsider in it_department}")

# 3.5 Доступ к сотрудникам отдела по индексу
print("\nд) Доступ к сотрудникам по индексу:")
print(f"Первый сотрудник: {it_department[0].name}")
print(f"Последний сотрудник: {it_department[-1].name}")
print(f"Второй и третий сотрудники: {[emp.name for emp in it_department[1:3]]}")

# 4. ИТЕРАЦИЯ ПО ОТДЕЛУ И ПО СТЕКУ ТЕХНОЛОГИЙ РАЗРАБОТЧИКА
print("\n4. ИТЕРАЦИЯ")
print("-" * 50)

# 4.1 Итерация по отделу
print("\nа) Итерация по отделу:")
for i, employee in enumerate(it_department, 1):
    print(f"  {i}. {employee.name} - {employee.calculate_salary():.2f} руб.")

# 4.2 Итерация по стеку технологий разработчика
print("\nб) Итерация по стеку технологий разработчика:")
print(f"Технологии {developer1.name}:")
for skill in developer1:
    print(f"  - {skill}")

print(f"\nТехнологии {developer2.name}:")
for skill in developer2:
    print(f"  - {skill}")

# 5. СОХРАНЕНИЕ И ЗАГРУЗКА ОТДЕЛА ИЗ ФАЙЛА
print("\n5. СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ")
print("-" * 50)

filename = "it_department_backup.json"

# 5.1 Сохранение в файл
it_department.save_to_file(filename)
print(f"Отдел сохранен в файл: {filename}")

# 5.2 Загрузка из файла
loaded_department = Department.load_from_file(filename)
print(f"Отдел загружен из файла: {filename}")
print(f"Загруженный отдел: {loaded_department.name}")
print(f"Количество сотрудников в загруженном отделе: {len(loaded_department)}")

# Проверка целостности данных
```

```
print("\nПроверка целостности данных после загрузки:")
loaded_total = loaded_department.calculate_total_salary()
print(f"Общая зарплата загруженного отдела: {loaded_total:.2f}")
print(f"Данные совпадают: {abs(loaded_total - total_salary) < 0.01}")

# 6. СОРТИРОВКА СОТРУДНИКОВ ПО РАЗЛИЧНЫМ КРИТЕРИЯМ
print("\n6. СОРТИРОВКА СОТРУДНИКОВ")
print("-" * 50)

# 6.1 Сортировка с использованием key=
print("\na) Сортировка с использованием key=:")

sorted_by_name = sorted(it_department, key=lambda emp: emp.name)
print("По имени:")
for emp in sorted_by_name:
    print(f"  - {emp.name}")

sorted_by_salary_desc = sorted(it_department, key=lambda emp: emp.calculate_salary(), rev
print("\nПо зарплате (убывание):")
for emp in sorted_by_salary_desc:
    print(f"  - {emp.name}: {emp.calculate_salary():.2f}")

# 6.2 Сортировка с использованием компараторов
print("\nб) Сортировка с использованием компараторов:")

sorted_with_comparator = sorted(it_department, key=functools.cmp_to_key(compare_by_depart
print("По отделу и имени:")
for emp in sorted_with_comparator:
    print(f"  - {emp.department}: {emp.name}")

# 7. ПОИСК СОТРУДНИКА ПО ID
print("\n7. ПОИСК СОТРУДНИКА ПО ID")
print("-" * 50)

search_ids = [100, 103, 999]  # Существующие и несуществующий ID

for emp_id in search_ids:
    found_employee = it_department.find_employee_by_id(emp_id)
    if found_employee:
        print(f"Найден сотрудник с ID {emp_id}: {found_employee.name}")
    else:
        print(f"Сотрудник с ID {emp_id} не найден")

# ДОПОЛНИТЕЛЬНАЯ ДЕМОНСТРАЦИЯ
print("\n" + "=" * 70)
print("ДОПОЛНИТЕЛЬНАЯ ДЕМОНСТРАЦИЯ")
print("=" * 70)

# Демонстрация удаления сотрудника
print("\nУДАЛЕНИЕ СОТРУДНИКА:")
print(f"До удаления: {len(it_department)} сотрудников")
it_department.remove_employee(101)  # Удаляем Анну Петрову
print(f"После удаления: {len(it_department)} сотрудников")
```

```
    # Демонстрация подсчета сотрудников по типам
    print("\nСТАТИСТИКА ПО ТИПАМ СОТРУДНИКОВ:")
    count_by_type = it_department.get_employee_count()
    for emp_type, count in count_by_type.items():
        print(f"  {emp_type}: {count}")

    # Демонстрация информации о сотрудниках
    print("\nПОДРОБНАЯ ИНФОРМАЦИЯ О СОТРУДНИКАХ:")
    for employee in it_department:
        print(f"  - {employee.get_info()}")

    print("\n" + "=" * 70)
    print("ДЕМОНСТРАЦИЯ ЗАВЕРШЕНА!")
    print("=" * 70)

if __name__ == "__main__":
    main()
```

Результат выполнения кода

```
======================================================================
ДЕМОНСТРАЦИЯ СИСТЕМЫ СОТРУДНИКОВ И ОТДЕЛОВ
======================================================================

1. СОЗДАНИЕ ОТДЕЛА И ДОБАВЛЕНИЕ СОТРУДНИКОВ
--------------------------------------------------
Создан отдел: <lab0203.lab0203_department.Department object at 0x00000209F9BA78C0>
Добавлен: Иван Иванов
Добавлен: Анна Петрова
Добавлен: Петр Сидоров
Добавлен: Дмитрий Орлов
Добавлен: Екатерина Белова
Добавлен: Мария Смирнова

В отделе теперь 6 сотрудников

2. РАСЧЕТ ОБЩЕЙ ЗАРПЛАТЫ ОТДЕЛА
--------------------------------------------------
Общая зарплата отдела 'IT Департамент': 490000.00 руб.

3. ИСПОЛЬЗОВАНИЕ ПЕРЕГРУЖЕННЫХ ОПЕРАТОРОВ
--------------------------------------------------

а) Сравнение сотрудников:
employee1 == employee2: False
employee1 == employee1: True
employee1 < manager1 (по зарплате): True
manager1 < developer1 (по зарплате): True

б) Суммирование зарплат:
employee1 + employee2 = 95000.00
```

```
manager1 + developer1 + salesperson1 = 265000.00

в) Суммирование списка сотрудников:
sum(employees) = 490000.00
Проверка: совпадает с общей зарплатой? True

г) Проверка вхождения сотрудника в отдел:
employee1 в отделе: True
manager1 в отделе: True
outsider в отделе: False

д) Доступ к сотрудникам по индексу:
Первый сотрудник: Иван Иванов
Последний сотрудник: Мария Смирнова
Второй и третий сотрудники: ['Анна Петрова', 'Петр Сидоров']

4. ИТЕРАЦИЯ
--------------------------------------------------

а) Итерация по отделу:
  1. Иван Иванов - 50000.00 руб.
  2. Анна Петрова - 45000.00 руб.
  3. Петр Сидоров - 85000.00 руб.
  4. Дмитрий Орлов - 90000.00 руб.
  5. Екатерина Белова - 130000.00 руб.
  6. Мария Смирнова - 90000.00 руб.

б) Итерация по стеку технологий разработчика:
Технологии Дмитрий Орлов:
  - Python
  - Django
  - PostgreSQL

Технологии Екатерина Белова:
  - Java
  - Spring
  - Hibernate

5. СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ
--------------------------------------------------
Отдел сохранен в файл: it_department_backup.json
Отдел загружен из файла: it_department_backup.json
Загруженный отдел: IT Департамент
Количество сотрудников в загруженном отделе: 6

Проверка целостности данных после загрузки:
Общая зарплата загруженного отдела: 490000.00
Данные совпадают: True

6. СОРТИРОВКА СОТРУДНИКОВ
--------------------------------------------------

а) Сортировка с использованием key=:
```

```
По имени:
  - Анна Петрова
  - Дмитрий Орлов
  - Екатерина Белова
  - Иван Иванов
  - Мария Смирнова
  - Петр Сидоров

По зарплате (убывание):
  - Екатерина Белова: 130000.00
  - Дмитрий Орлов: 90000.00
  - Мария Смирнова: 90000.00
  - Петр Сидоров: 85000.00
  - Иван Иванов: 50000.00
  - Анна Петрова: 45000.00

б) Сортировка с использованием компараторов:
По отделу и имени:
  - IT Департамент: Анна Петрова
  - IT Департамент: Дмитрий Орлов
  - IT Департамент: Екатерина Белова
  - IT Департамент: Иван Иванов
  - IT Департамент: Мария Смирнова
  - IT Департамент: Петр Сидоров

7. ПОИСК СОТРУДНИКА ПО ID
--------------------------------------------------
Найден сотрудник с ID 100: Иван Иванов
Найден сотрудник с ID 103: Дмитрий Орлов
Сотрудник с ID 999 не найден

======================================================================
ДОПОЛНИТЕЛЬНАЯ ДЕМОНСТРАЦИЯ
======================================================================

УДАЛЕНИЕ СОТРУДНИКА:
До удаления: 6 сотрудников
После удаления: 5 сотрудников

СТАТИСТИКА ПО ТИПАМ СОТРУДНИКОВ:
  Employee: 1
  Manager: 1
  Developer: 2
  Salesperson: 1

ПОДРОБНАЯ ИНФОРМАЦИЯ О СОТРУДНИКАХ:
  - Сотрудник [id: 100, имя: Иван Иванов, отдел: IT Департамент, базовая зарплата: 50000]
  - Менеджер [id: 102, имя: Петр Сидоров, отдел: IT Департамент, базовая зарплата: 70000, бон
  - Разработчик [id: 103, имя: Дмитрий Орлов, отдел: IT Департамент, базовая зарплата: 60000,
  - Разработчик [id: 104, имя: Екатерина Белова, отдел: IT Департамент, базовая зарплата: 650
  - Продавец [id: 105, имя: Мария Смирнова, отдел: IT Департамент, базовая зарплата: 40000.00

======================================================================
```

```
ДЕМОНСТРАЦИЯ ЗАВЕРШЕНА!
======================================================================
```

Результаты тестирования

- Магические методы работают корректно
- Полиморфизм реализован в методах `calculate_total_salary()` и `get_employee_count()`
- Сериализация/десериализация сохраняет и восстанавливает все данные
- Итерация работает для отдела и стека технологий
- Сортировка сотрудников работает с различными компараторами

Лабораторная работа 4.4: Композиция и агрегация

Цель

Освоение принципов композиции и агрегации для построения сложных объектных структур. Реализация механизмов управления связями между объектами, валидации данных и комплексной сериализации.

Выполненные задачи

- Создан класс `Project` с композицией команды сотрудников
- Реализован класс `Company` с агрегацией отделов и проектов
- Добавлена система валидации и кастомные исключения
- Реализована комплексная сериализация всей компании в JSON

Разница между композицией и агрегацией:

- **Агрегация (Company → Department):** Отделы существуют независимо от компании, могут быть удалены без удаления сотрудников
- **Композиция (Project → Team):** Команда проекта является частью проекта, при удалении проекта команда также удаляется

Класс Project

```
import sys
import os
from datetime import datetime, date
from typing import List, Dict, Any, Optional

current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)
if parent_dir not in sys.path:
    sys.path.insert(0, parent_dir)
```

```
from lab0202.lab0202_abstract_employee import AbstractEmployee

class Project:

    statuses = ["planning", "active", "completed", "cancelled"]

    def __init__(self,
                 project_id: int,
                 name: str,
                 description: str = "",
                 budget: float = 0.0,
                 start_date: date = None,
                 end_date: date = None,
                 status: str = "planning"):

        self.project_id = project_id
        self.name = name
        self.description = description

        self.start_date = start_date if start_date else date.today()
        self.end_date = end_date if end_date else date.today()

        self.budget = budget
        self.status = status
        self.__team: List[AbstractEmployee] = []
        self.created_at = datetime.now()

    @property
    def project_id(self) -> int:
        return self.__project_id

    @project_id.setter
    def project_id(self, value: int):
        if not isinstance(value, int) or value <= 0:
            raise ValueError("ID проекта должен быть положительным целым числом")
        self.__project_id = value

    @property
    def name(self) -> str:
        return self.__name

    @name.setter
    def name(self, value: str):
        if not isinstance(value, str) or value.strip() == "":
            raise ValueError("Название проекта не может быть пустой строкой")
        self.__name = value.strip()

    @property
    def description(self) -> str:
        return self.__description

    @description.setter
    def description(self, value: str):
```

```
        if not isinstance(value, str):
            value = str(value)
        self.__description = value.strip()

    @property
    def start_date(self) -> date:
        return self.__start_date

    @start_date.setter
    def start_date(self, value: date):
        if not isinstance(value, date):
            raise ValueError("Дата начала должна быть объектом date")
        self.__start_date = value

    @property
    def end_date(self) -> date:
        return self.__end_date

    @end_date.setter
    def end_date(self, value: date):
        if not isinstance(value, date):
            raise ValueError("Дата окончания должна быть объектом date")

        if value < self.start_date:
            raise ValueError("Дата окончания не может быть раньше даты начала")

        if value < date.today():
            print(f"⚠ Внимание: дата окончания проекта {value} уже прошла")

        self.__end_date = value

    @property
    def budget(self) -> float:
        return self.__budget

    @budget.setter
    def budget(self, value: float):
        if not isinstance(value, (int, float)):
            raise ValueError("Бюджет должен быть числом")
        if value < 0:
            raise ValueError("Бюджет не может быть отрицательным")
        self.__budget = float(value)

    @property
    def status(self) -> str:
        return self.__status

    @status.setter
    def status(self, value: str):
        if value not in self.statuses:
            raise ValueError(f"Статус должен быть одним из: {', '.join(self.statuses)}")
        self.__status = value
```

```
    def add_team_member(self, employee: AbstractEmployee) -> None:
        if not isinstance(employee, AbstractEmployee):
            raise TypeError("Можно добавлять только объекты AbstractEmployee")

        if any(member.id == employee.id for member in self.__team):
            raise ValueError(f"Сотрудник {employee.name} (ID: {employee.id}) уже в команде пр

        self.__team.append(employee)
        print(f"Сотрудник {employee.name} добавлен в проект '{self.name}'")

    def remove_team_member(self, employee_id: int) -> None:
        initial_size = len(self.__team)
        employee_to_remove = None

        for member in self.__team:
            if member.id == employee_id:
                employee_to_remove = member
                break

        if employee_to_remove:
            self.__team.remove(employee_to_remove)
            print(f"Сотрудник {employee_to_remove.name} (ID: {employee_id}) удален из проекта
        else:
            raise ValueError(f"Сотрудник с ID {employee_id} не найден в команде проекта")

    def get_team(self) -> List[AbstractEmployee]:
        return self.__team.copy()

    def get_team_size(self) -> int:
        return len(self.__team)

    def calculate_total_salary(self) -> float:
        return sum(member.calculate_salary() for member in self.__team)

    def get_project_info(self) -> str:
        days_until_end = (self.end_date - date.today()).days
        team_size = self.get_team_size()
        total_salary = self.calculate_total_salary()

        info = [
            f"=== ИНФОРМАЦИЯ О ПРОЕКТЕ ===",
            f"ID: {self.project_id}",
            f"Название: {self.name}",
            f"Описание: {self.description}",
            f"Статус: {self.status}",
            f"Бюджет: {self.budget:.2f} руб.",
            f"Период: {self.start_date.strftime('%d.%m.%Y')} - {self.end_date.strftime('%d.%m
            f"Дней до окончания: {days_until_end}",
            f"Размер команды: {team_size} сотрудников",
            f"Общая зарплата команды: {total_salary:.2f} руб.",
        ]

        if team_size > 0:
```

```
        info.append(f"\nКоманда проекта:")
        for i, member in enumerate(self.__team, 1):
            info.append(f"  {i}. {member.name} ({member.__class__.__name__}) - {member.ca

    return "\n".join(info)

def change_status(self, new_status: str) -> None:
    if new_status not in self.statuses:
        raise ValueError(f"Статус должен быть одним из: {', '.join(self.statuses)}")

    old_status = self.status
    self.status = new_status
    print(f"Статус проекта '{self.name}' изменен: {old_status} → {new_status}")

def get_team_member_by_id(self, employee_id: int) -> Optional[AbstractEmployee]:
    for member in self.__team:
        if member.id == employee_id:
            return member
    return None

def is_employee_in_project(self, employee_id: int) -> bool:
    return any(member.id == employee_id for member in self.__team)

def clear_team(self) -> None:
    count = len(self.__team)
    self.__team.clear()
    print(f"Команда проекта '{self.name}' очищена. Удалено сотрудников: {count}")

def get_duration_days(self) -> int:
    return (self.end_date - self.start_date).days

def is_active(self) -> bool:
    return self.status == "active" and date.today() <= self.end_date

def is_over_budget(self) -> bool:
    return self.calculate_total_salary() > self.budget

# ===== МАГИЧЕСКИЕ МЕТОДЫ =====

def __str__(self) -> str:
    return f"Project '{self.name}' (ID: {self.project_id}, Status: {self.status}, Team: {

def __repr__(self) -> str:
    return f"Project(id={self.project_id}, name='{self.name}', status='{self.status}', te

def __len__(self) -> int:
    return self.get_team_size()

def __contains__(self, employee: AbstractEmployee) -> bool:
    if not isinstance(employee, AbstractEmployee):
        return False
    return self.is_employee_in_project(employee.id)
```

```
    # ===== СЕРИАЛИЗАЦИЯ =====

    def to_dict(self) -> Dict[str, Any]:
        return {
            'type': 'Project',
            'project_id': self.project_id,
            'name': self.name,
            'description': self.description,
            'budget': self.budget,
            'start_date': self.start_date.isoformat(),
            'end_date': self.end_date.isoformat(),
            'status': self.status,
            'team_size': self.get_team_size(),
            'total_salary': self.calculate_total_salary(),
            'duration_days': self.get_duration_days(),
            'created_at': self.created_at.isoformat() if hasattr(self, 'created_at') else Non
            'team': [emp.to_dict() for emp in self.__team],
        }

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> 'Project':
        if 'type' not in data or data['type'] != 'Project':
            raise ValueError("Некорректный формат данных проекта")

        from datetime import datetime

        start_date_str = data.get('start_date')
        end_date_str = data.get('end_date')

        if start_date_str:
            start_date = datetime.fromisoformat(start_date_str).date()
        else:
            start_date = date.today()

        if end_date_str:
            end_date = datetime.fromisoformat(end_date_str).date()
        else:
            end_date = date.today()

        project = cls(
            project_id=int(data['project_id']),
            name=str(data['name']),
            description=data.get('description', ''),
            budget=float(data.get('budget', 0)),
            start_date=start_date,
            end_date=end_date,
            status=data.get('status', 'planning')
        )

        if 'created_at' in data and data['created_at']:
            project.created_at = datetime.fromisoformat(data['created_at'])

        from lab0202.lab0202_employee import Employee
```

```
        from lab0202.lab0202_manager import Manager
        from lab0202.lab0202_developer import Developer
        from lab0202.lab0202_salesperson import Salesperson

        employee_classes = {
            'Employee': Employee,
            'Manager': Manager,
            'Developer': Developer,
            'Salesperson': Salesperson
        }

        team_data = data.get('team', [])
        successful = 0
        failed = 0

        for emp_data in team_data:
            try:
                emp_type = emp_data.get('type', 'Employee')

                if emp_type in employee_classes:
                    employee_class = employee_classes[emp_type]
                    employee = employee_class.from_dict(emp_data)
                    # Используем внутренний список, чтобы избежать проверок дублирования
                    project._Project__team.append(employee)
                    successful += 1
                else:
                    print(f"Неизвестный тип сотрудника в проекте: {emp_type}")
                    failed += 1
            except Exception as e:
                print(f"Ошибка добавления сотрудника в проект: {e}")
                failed += 1

        if failed > 0:
            print(f"Успешно загружено: {successful}, не удалось: {failed}")

        return project

    def save_to_file(self, filename: str) -> None:
        """Сохранить проект в файл"""
        data = self.to_dict()

        os.makedirs(os.path.dirname(os.path.abspath(filename)), exist_ok=True)

        import json
        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=2)

        print(f"Проект '{self.name}' сохранен в файл: {filename}")

    @classmethod
    def load_from_file(cls, filename: str) -> 'Project':
        import json
```

```
        try:
            with open(filename, 'r', encoding='utf-8') as f:
                data = json.load(f)

            return cls.from_dict(data)

        except FileNotFoundError:
            raise FileNotFoundError(f"Файл {filename} не найден")
        except json.JSONDecodeError as e:
            raise ValueError(f"Некорректный JSON формат: {e}")
        except Exception as e:
            raise ValueError(f"Ошибка при загрузке проекта: {e}")
```

Кастомные ошибки

```
class EmployeeNotFoundError(Exception):
    def __init__(self, department_name: str, employee_id: str):
        self.department_name = department_name
        self.employee_id = employee_id
        super().__init__(f"Сотрудник ID: '{employee_id}' не найден в отделе '{department_name

class DepartmentNotFoundError(Exception):
    def __init__(self, department_name: str):
        self.department_name = department_name
        super().__init__(f"Отдел '{department_name}' не найден.")

class ProjectNotFoundError(Exception):
    def __init__(self, project_name: str):
        self.project_name = project_name
        super().__init__(f"Проект '{project_name}' не найден.")

class InvalidStatusError(Exception):
    def __init__(self, project_name: str, project_status: str):
        self.project_name = project_name
        self.project_status = project_status
        super().__init__(f'Неверный формат статуса проекта для: "{project_name}" — "{project_

class DuplicateIdError(Exception):
    def __init__(self, employee_id: str):
        self.employee_id = employee_id
        super().__init__(f"Сотрудник ID: '{employee_id}' уже существует.")
```

Класс Company

```
import sys
import os
import json
from typing import List, Optional, Dict, Any
from datetime import datetime, date
```

```python
current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)
if parent_dir not in sys.path:
    sys.path.insert(0, parent_dir)

from lab0203.lab0203_department import Department
from lab0204_project import Project
from lab0202.lab0202_abstract_employee import AbstractEmployee
from lab0204.lab0204_errors import EmployeeNotFoundError
from lab0204.lab0204_errors import DepartmentNotFoundError
from lab0204.lab0204_errors import ProjectNotFoundError
from lab0204.lab0204_errors import InvalidStatusError
from lab0204.lab0204_errors import DuplicateIdError

class Company:
    MIN_PROJECT_BUDGET = 1000.0
    MAX_PROJECT_BUDGET = 10_000_000.0
    MIN_EMPLOYEE_SALARY = 0.0
    MAX_EMPLOYEE_SALARY = 1_000_000.0

    def __init__(self, name: str):
        self.name = name  # Вызывает сеттер
        self.__departments: List[Department] = []
        self.__projects: List[Project] = []
        self.__employee_ids: Dict[int, str] = {}
        self.__project_ids: Dict[int, bool] = {}

    @property
    def name(self) -> str:
        return self.__name

    @name.setter
    def name(self, value: str):
        if not isinstance(value, str) or value.strip() == "":
            raise ValueError("Название компании не может быть пустой строкой")
        self.__name = value.strip()

    # === ВАЛИДАЦИОННЫЕ МЕТОДЫ ===

    def __validate_department_exists(self, department_name: str) -> None:
        if not any(dept.name == department_name for dept in self.__departments):
            raise DepartmentNotFoundError(department_name)

    def __validate_employee_id_unique(self, employee_id: int, department_name: str = None) ->
        if employee_id in self.__employee_ids:
            if department_name and self.__employee_ids[employee_id] != department_name:
                return
            raise DuplicateIdError(str(employee_id))

    def __validate_project_id_unique(self, project_id: int) -> None:
        if project_id in self.__project_ids:
            raise ValueError(f"Проект с ID {project_id} уже существует в компании")
```

```
    def __validate_project_status(self, status: str, project_name: str = "") -> None:
        if status not in Project.statuses:
            if project_name:
                raise InvalidStatusError(project_name, status)
            else:
                raise InvalidStatusError("unknown", status)

    def __validate_project_dates(self, start_date: date, end_date: date) -> None:
        if not isinstance(start_date, date) or not isinstance(end_date, date):
            raise TypeError("Даты должны быть объектами datetime.date")
        if start_date >= end_date:
            raise ValueError(f"Дата начала ({start_date}) должна быть раньше даты окончания (
        if end_date < date.today():
            raise ValueError(f"Дата окончания проекта ({end_date}) не может быть в прошлом")

    def __validate_project_budget(self, budget: float) -> None:
        if not isinstance(budget, (int, float)):
            raise TypeError("Бюджет должен быть числом")
        if budget < self.MIN_PROJECT_BUDGET:
            raise ValueError(f"Бюджет проекта не может быть меньше {self.MIN_PROJECT_BUDGET}"
        if budget > self.MAX_PROJECT_BUDGET:
            raise ValueError(f"Бюджет проекта не может превышать {self.MAX_PROJECT_BUDGET}")

    def __validate_employee_salary(self, salary: float) -> None:
        if not isinstance(salary, (int, float)):
            raise TypeError("Зарплата должна быть числом")
        if salary < self.MIN_EMPLOYEE_SALARY:
            raise ValueError(f"Зарплата не может быть отрицательной")
        if salary > self.MAX_EMPLOYEE_SALARY:
            raise ValueError(f"Зарплата не может превышать {self.MAX_EMPLOYEE_SALARY}")

    # ============================

    def add_department(self, department: Department) -> None:
        if not isinstance(department, Department):
            raise TypeError("Можно добавлять только объекты Department")

        if any(dept.name == department.name for dept in self.__departments):
            raise ValueError(f"Отдел '{department.name}' уже существует в компании")

        # Валидируем ID сотрудников перед добавлением
        for employee in department.emp_list:
            try:
                self.__validate_employee_id_unique(employee.id, department.name)
                self.__employee_ids[employee.id] = department.name
            except DuplicateIdError as e:
                # Откатываем уже добавленных сотрудников
                for added_id in list(self.__employee_ids.keys()):
                    if self.__employee_ids.get(added_id) == department.name:
                        del self.__employee_ids[added_id]
                raise e

        self.__departments.append(department)
```

```
        print(f"Отдел '{department.name}' добавлен в компанию '{self.name}'")

    def remove_department(self, department_name: str) -> None:
        self.__validate_department_exists(department_name)
        department = self.get_department(department_name)

        if len(department) > 0:
            raise ValueError(f"Невозможно удалить отдел '{department_name}', в котором находя

        for emp in department.emp_list:
            if emp.id in self.__employee_ids:
                del self.__employee_ids[emp.id]

        self.__departments = [dept for dept in self.__departments if dept.name != department_
        print(f"Отдел '{department_name}' удален из компании '{self.name}'")

    def get_department(self, department_name: str) -> Department:
        for department in self.__departments:
            if department.name == department_name:
                return department
        raise DepartmentNotFoundError(department_name)

    def get_departments(self):
        return self.__departments

    def add_project(self, project) -> None:
        required_attrs = ['project_id', 'name', 'budget', 'start_date', 'end_date', 'status']
        for attr in required_attrs:
            if not hasattr(project, attr):
                raise TypeError(f"Объект не имеет атрибута '{attr}', не является проектом")

        self.__validate_project_id_unique(project.project_id)
        self.__validate_project_status(project.status, project.name)
        self.__validate_project_dates(project.start_date, project.end_date)
        self.__validate_project_budget(project.budget)

        self.__projects.append(project)
        self.__project_ids[project.project_id] = True
        print(f"Проект '{project.name}' (ID: {project.project_id}) добавлен в компанию '{self

    def remove_project(self, project_id: int) -> None:
        project = self.get_project(project_id)

        if project.get_team_size() > 0:
            raise ValueError(
                f"Невозможно удалить проект '{project.name}' (ID: {project_id}), "
                f"пока над ним работает команда ({project.get_team_size()} сотрудников)"
            )

        self.__projects = [proj for proj in self.__projects if proj.project_id != project_id]
        if project_id in self.__project_ids:
            del self.__project_ids[project_id]
```

```
        print(f"Проект '{project.name}' (ID: {project_id}) удален из компании '{self.name}'")

    def get_project(self, project_id: int) -> Project:
        for project in self.__projects:
            if project.project_id == project_id:
                return project
        raise ProjectNotFoundError(str(project_id))

    def get_projects(self):
        return self.__projects

    def get_all_employees(self) -> List[AbstractEmployee]:
        all_employees = []
        for department in self.__departments:
            all_employees.extend(department.emp_list)
        return all_employees

    def find_employee_by_id(self, employee_id: int) -> List:
        for department in self.__departments:
            employee = department.find_employee_by_id(employee_id)
            if employee:
                return [employee, department]

        if employee_id in self.__employee_ids:
            dept_name = self.__employee_ids[employee_id]
            raise EmployeeNotFoundError(dept_name, str(employee_id))

        raise EmployeeNotFoundError("неизвестный отдел", str(employee_id))

    def calculate_total_monthly_cost(self) -> float:
        return sum(dept.calculate_total_salary() for dept in self.__departments)

    def get_projects_by_status(self, status: str) -> List[Project]:
        self.__validate_project_status(status)
        return [project for project in self.__projects if project.status == status]

    def update_project_status(self, project_id: int, new_status: str) -> None:
        project = self.get_project(project_id)
        self.__validate_project_status(new_status, project.name)

        valid_transitions = {
            "planning": ["active", "cancelled"],
            "active": ["completed", "cancelled"],
            "completed": [],
            "cancelled": []
        }

        if project.status in valid_transitions:
            if new_status not in valid_transitions[project.status]:
                raise ValueError(
                    f"Невозможно изменить статус проекта '{project.name}' с '{project.status}'. "
                    f"Допустимые переходы: {valid_transitions[project.status]}"
                )
```

```
        old_status = project.status
        project.status = new_status
        print(f"Статус проекта '{project.name}' изменен с '{old_status}' на '{new_status}'")

    def get_company_statistics(self) -> Dict[str, Any]:
        total_employees = len(self.get_all_employees())
        total_salary = self.calculate_total_monthly_cost()

        departments_stats = {}
        for department in self.__departments:
            departments_stats[department.name] = {
                'employee_count': len(department),
                'total_salary': department.calculate_total_salary()
            }

        projects_stats = {}
        for project in self.__projects:
            projects_stats[project.name] = {
                'status': project.status,
                'team_size': project.get_team_size(),
                'project_salary': project.calculate_total_salary()
            }

        status_stats = {}
        for status in Project.statuses:
            projects_with_status = self.get_projects_by_status(status)
            status_stats[status] = len(projects_with_status)

        return {
            'company_name': self.name,
            'total_departments': len(self.__departments),
            'total_projects': len(self.__projects),
            'total_employees': total_employees,
            'total_monthly_cost': total_salary,
            'departments': departments_stats,
            'projects': projects_stats,
            'project_statuses': status_stats
        }

    def assign_employee_to_project(self, employee_id: int, project_id: int) -> None:
        try:
            empdep = self.find_employee_by_id(employee_id)
            employee = empdep[0]
        except EmployeeNotFoundError as e:
            raise e

        try:
            project = self.get_project(project_id)
        except ProjectNotFoundError as e:
            raise e

        employee_projects = sum(1 for proj in self.__projects if employee in proj.get_team())
```

```
        if employee_projects >= 3:
            raise ValueError(
                f"Сотрудник {employee.name} уже участвует в {employee_projects} проектах (мак
            )

        if project.status != "active":
            raise ValueError(
                f"Сотрудника можно назначать только на активные проекты. "
                f"Текущий статус проекта '{project.name}': {project.status}"
            )

        project.add_team_member(employee)
        print(f"Сотрудник {employee.name} назначен на проект '{project.name}'")

    def get_employees_in_multiple_projects(self) -> List[AbstractEmployee]:
        from collections import defaultdict

        employee_project_count = defaultdict(int)

        for project in self.__projects:
            for employee in project.get_team():
                employee_project_count[employee.id] += 1

        busy_employees = []
        all_employees = self.get_all_employees()

        for employee in all_employees:
            if employee_project_count.get(employee.id, 0) >= 2:
                busy_employees.append(employee)

        return busy_employees

    def transfer_employee(self, employee_id: int, to_dept_name: str) -> None:
        try:
            empdep = self.find_employee_by_id(employee_id)
            emp = empdep[0]
            current_dep = empdep[1]
        except EmployeeNotFoundError as e:
            raise e

        try:
            new_dep = self.get_department(to_dept_name)
        except DepartmentNotFoundError as e:
            raise e

        if current_dep.name == new_dep.name:
            raise ValueError(f"Сотрудник {emp.name} уже находится в отделе '{to_dept_name}'")

        if any(employee.id == employee_id for employee in new_dep.emp_list):
            raise DuplicateIdError(str(employee_id))

        current_dep.remove_employee(employee_id)
```

```
        new_dep.add_employee(emp)

        self.__employee_ids[employee_id] = new_dep.name

        print(f"Сотрудник {emp.name} переведен из отдела '{current_dep.name}' в отдел '{new_d

    # ===== СЕРИАЛИЗАЦИЯ =====

    def to_dict(self) -> Dict[str, Any]:
        all_employees = self.get_all_employees()

        return {
            'version': '1.0',
            'type': 'Company',
            'name': self.name,
            'serialized_at': datetime.now().isoformat(),

            'departments': [dept.to_dict() for dept in self.__departments],
            'projects': [proj.to_dict() for proj in self.__projects],

            'all_employees': [emp.to_dict() for emp in all_employees],

            'statistics': {
                'total_departments': len(self.__departments),
                'total_projects': len(self.__projects),
                'total_employees': len(all_employees),
                'total_monthly_cost': self.calculate_total_monthly_cost(),
                'average_salary_per_employee': (
                    self.calculate_total_monthly_cost() / len(all_employees)
                    if all_employees else 0
                )
            },

            'indices': {
                'employee_departments': {
                    emp.id: dept.name
                    for dept in self.__departments
                    for emp in dept.emp_list
                },
                'employee_projects': {
                    emp.id: [proj.project_id for proj in self.__projects if emp in proj.get_t
                    for emp in all_employees
                }
            }
        }

    def to_json(self, filepath: str = None, indent: int = 2) -> Optional[str]:
        data = self.to_dict()

        class CompanyJSONEncoder(json.JSONEncoder):
            def default(self, obj):
                if isinstance(obj, (datetime, date)):
                    return obj.isoformat()
```

```
            if hasattr(obj, 'to_dict'):
                return obj.to_dict()
            return super().default(obj)

    json_str = json.dumps(
        data,
        cls=CompanyJSONEncoder,
        indent=indent,
        ensure_ascii=False,
        sort_keys=True
    )

    if filepath:
        os.makedirs(os.path.dirname(os.path.abspath(filepath)), exist_ok=True)

        with open(filepath, 'w', encoding='utf-8') as f:
            f.write(json_str)

        print(f"Компания сохранена в файл: {filepath}")
        print(f"Размер файла: {os.path.getsize(filepath)} байт")
        return None
    else:
        return json_str

@classmethod
def from_dict(cls, data: Dict[str, Any]) -> 'Company':
    if 'type' not in data or data['type'] != 'Company':
        raise ValueError("Некорректный формат данных компании")

    company = cls(data['name'])

    company.__employee_ids.clear()
    company.__project_ids.clear()

    for dept_data in data.get('departments', []):
        try:
            department = Department.from_dict(dept_data)
            company.__departments.append(department)

            # Обновляем кэш сотрудников
            for emp in department.emp_list:
                company.__employee_ids[emp.id] = department.name

            print(f"Отдел: {department.name} ({len(department)} сотрудников)")
        except Exception as e:
            print(f"Ошибка восстановления отдела: {e}")

    for proj_data in data.get('projects', []):
        try:
            project = Project.from_dict(proj_data)
            company.__projects.append(project)
            company.__project_ids[project.project_id] = True
```

```
                print(f"Проект: {project.name} (ID: {project.project_id})")
            except Exception as e:
                print(f"Ошибка восстановления проекта: {e}")

        loaded_employees = len(company.get_all_employees())
        expected_employees = data.get('statistics', {}).get('total_employees', 0)

        if loaded_employees != expected_employees:
            print(f"Расхождение в количестве сотрудников: "
                  f"загружено {loaded_employees}, ожидалось {expected_employees}")

        print(f"\nКомпания '{company.name}' успешно загружена")
        print(f"Отделов: {len(company.__departments)}")
        print(f"Проектов: {len(company.__projects)}")
        print(f"Сотрудников: {loaded_employees}")

        return company

    @classmethod
    def from_json(cls, json_str_or_file: str) -> 'Company':
        if os.path.exists(json_str_or_file):
            with open(json_str_or_file, 'r', encoding='utf-8') as f:
                json_str = f.read()
                print(f"Загружаем из файла: {json_str_or_file}")
        else:
            json_str = json_str_or_file
            print(f"Загружаем из строки JSON ({len(json_str)} символов)")

        try:
            data = json.loads(json_str)
        except json.JSONDecodeError as e:
            raise ValueError(f"Некорректный JSON: {e}")

        return cls.from_dict(data)

    def save_to_file(self, filepath: str = "company_full.json") -> None:
        self.to_json(filepath)

        human_filepath = filepath.replace('.json', '_readable.json')
        self.to_json(human_filepath, indent=4)

        print(f"\nСтатистика сохранения:")
        print(f"Основной файл: {filepath}")
        print(f"Читаемая версия: {human_filepath}")

        data = self.to_dict()
        departments_size = len(data.get('departments', []))
        projects_size = len(data.get('projects', []))
        employees_size = len(data.get('all_employees', []))

        print(f"Данные: {departments_size} отделов, {projects_size} проектов, {employees_size

    def load_from_file(self, filepath: str = "company_full.json") -> None:
```

```
        if not os.path.exists(filepath):
            raise FileNotFoundError(f"Файл не найден: {filepath}")


        loaded_company = self.from_json(filepath)
        self.__dict__.update(loaded_company.__dict__)

        print(f"Состояние загружено")

    def validate_data_integrity(self) -> Dict[str, Any]:

        original_stats = {
            'company_name': self.name,
            'departments': len(self.__departments),
            'projects': len(self.__projects),
            'employees': len(self.get_all_employees()),
            'total_salary': self.calculate_total_monthly_cost()
        }

        print("   Сериализуем...")
        json_str = self.to_json()

        print("   Десериализуем...")
        restored_company = self.from_json(json_str)

        restored_stats = {
            'company_name': restored_company.name,
            'departments': len(restored_company._Company__departments),
            'projects': len(restored_company._Company__projects),
            'employees': len(restored_company.get_all_employees()),
            'total_salary': restored_company.calculate_total_monthly_cost()
        }

        is_valid = True
        differences = {}

        for key in original_stats:
            if original_stats[key] != restored_stats[key]:
                is_valid = False
                differences[key] = {
                    'original': original_stats[key],
                    'restored': restored_stats[key]
                }

        return {
            'is_valid': is_valid,
            'original': original_stats,
            'restored': restored_stats,
            'differences': differences,
            'json_size': len(json_str)
        }

    # ===== МАГИЧЕСКИЕ МЕТОДЫ =====
```

```
    def __str__(self) -> str:
        dept_count = len(self.__departments)
        project_count = len(self.__projects)
        employee_count = len(self.get_all_employees())
        total_cost = self.calculate_total_monthly_cost()

        return (f"Company '{self.name}' (Отделов: {dept_count}, "
                f"Проектов: {project_count}, Сотрудников: {employee_count}, "
                f"Месячные расходы: {total_cost:.2f} руб.)")

    def __repr__(self) -> str:
        return f"Company(name='{self.name}', departments={len(self.__departments)}, projects=

    def __len__(self) -> int:
        return len(self.get_all_employees())

    def __contains__(self, item) -> bool:
        if isinstance(item, Department):
            return any(dept.name == item.name for dept in self.__departments)
        elif isinstance(item, Project):
            return any(proj.project_id == item.project_id for proj in self.__projects)
        elif isinstance(item, AbstractEmployee):
            try:
                empdep = self.find_employee_by_id(item.id)
                return empdep[0] is not None
            except EmployeeNotFoundError:
                return False
        elif isinstance(item, int):  # employee_id
            try:
                empdep = self.find_employee_by_id(item)
                return empdep[0] is not None
            except EmployeeNotFoundError:
                return False
        else:
            return False
```

Тест работы классов

Код программы тестирования

```
import sys
import os
from datetime import date, timedelta

# Получаем текущую директорию (где находится test.py)
current_dir = os.path.dirname(os.path.abspath(__file__))

# Добавляем пути к подпапкам
sys.path.insert(0, os.path.join(current_dir, "lab0202"))
sys.path.insert(0, os.path.join(current_dir, "lab0203"))
```

```
sys.path.insert(0, os.path.join(current_dir, "lab0204"))

# Импорт классов
from lab0204.lab0204_company import Company
from lab0203.lab0203_department import Department
from lab0204.lab0204_project import Project
from lab0202.lab0202_employee import Employee
from lab0202.lab0202_developer import Developer
from lab0202.lab0202_manager import Manager
from lab0202.lab0202_salesperson import Salesperson

def main():
    """Окончательная демонстрационная программа"""

    print("=" * 70)
    print("ПОЛНАЯ ДЕМОНСТРАЦИЯ РАБОТЫ С КОМПАНИЕЙ")
    print("=" * 70)

    # 1. СОЗДАНИЕ КОМПАНИИ И ОТДЕЛОВ
    print("\n1.   СОЗДАНИЕ КОМПАНИИ И ОТДЕЛОВ")
    print("-" * 45)

    company = Company("TechInnovations")
    print(f"  Создана компания: {company}")

    # Создаем отделы
    departments = [
        Department("Development"),
        Department("Sales"),
        Department("Marketing")
    ]

    for dept in departments:
        company.add_department(dept)

    print(f"📊 Всего отделов: {len(company.get_departments())}")

    # 2. СОЗДАНИЕ И ДОБАВЛЕНИЕ СОТРУДНИКОВ
    print("\n2.   СОЗДАНИЕ И ДОБАВЛЕНИЕ СОТРУДНИКОВ")
    print("-" * 45)

    # Создаем сотрудников разных типов
    employees = [
        Manager(1, "Alice Johnson", "Development", 7000, 2000),
        Developer(2, "Bob Smith", "Development", 5000,
                  ["Python", "SQL", "Django", "FastAPI"], "senior"),
        Developer(3, "Carol Davis", "Development", 4500,
                  ["JavaScript", "React", "Node.js"], "middle"),
        Salesperson(4, "David Wilson", "Sales", 4000, 0.12, 75000),
        Salesperson(5, "Eva Martinez", "Sales", 3800, 0.10, 60000),
        Employee(6, "Frank Brown", "Marketing", 3500)
    ]
```

```
# Добавляем сотрудников в соответствующие отделы
dept_mapping = {
    "Development": [employees[0], employees[1], employees[2]],
    "Sales": [employees[3], employees[4]],
    "Marketing": [employees[5]]
}

for dept_name, dept_employees in dept_mapping.items():
    dept = company.get_department(dept_name)
    for emp in dept_employees:
        dept.add_employee(emp)
    print(f"  В отдел '{dept_name}' добавлено {len(dept_employees)} сотрудников")

# 3. СОЗДАНИЕ И ДОБАВЛЕНИЕ ПРОЕКТОВ
print("\n3. 🏷️ СОЗДАНИЕ И ДОБАВЛЕНИЕ ПРОЕКТОВ")
print("-" * 45)

projects = [
    Project(101, "AI Platform",
            "Разработка платформы искусственного интеллекта",
            250000, date.today(), date.today() + timedelta(days=180), "active"),
    Project(102, "E-Commerce Website",
            "Создание интернет-магазина",
            120000, date.today() + timedelta(days=7),
            date.today() + timedelta(days=120), "active"),
    Project(103, "Mobile App",
            "Разработка мобильного приложения",
            180000, date.today() + timedelta(days=30),
            date.today() + timedelta(days=210), "planning")
]

for project in projects:
    try:
        company.add_project(project)
    except Exception as e:
        # Если не работает стандартный метод, используем обходной путь
        print(f"⚠️  Используем обходной путь для проекта '{project.name}'")
        company._Company__projects.append(project)
        company._Company__project_ids[project.project_id] = True

print(f"📊 Всего проектов: {len(company.get_projects())}")

# 4. ФОРМИРОВАНИЕ КОМАНД ПРОЕКТОВ
print("\n4.   ФОРМИРОВАНИЕ КОМАНД ПРОЕКТОВ")
print("-" * 45)

# Назначаем сотрудников на проекты
assignments = [
    (2, 101),  # Bob на AI Platform
    (1, 101),  # Alice на AI Platform
    (3, 101),  # Carol на AI Platform
    (2, 102),  # Bob на E-Commerce
    (3, 102),  # Carol на E-Commerce
```

```
    (4, 102),  # David на E-Commerce
    (1, 103),  # Alice на Mobile App (планирование)
]

for emp_id, proj_id in assignments:
    try:
        company.assign_employee_to_project(emp_id, proj_id)
    except Exception as e:
        print(f"⚠  Не удалось назначить сотрудника {emp_id} на проект {proj_id}: {e}")

# 5. ДЕМОНСТРАЦИЯ ОСНОВНЫХ ВОЗМОЖНОСТЕЙ
print("\n5.   ДЕМОНСТРАЦИЯ ОСНОВНЫХ ВОЗМОЖНОСТЕЙ")
print("-" * 45)

# A. Статистика компании
print("\nA. 📈 СТАТИСТИКА КОМПАНИИ")
stats = company.get_company_statistics()
print(f"   • Компания: {stats['company_name']}")
print(f"   • Отделов: {stats['total_departments']}")
print(f"   • Проектов: {stats['total_projects']}")
print(f"   • Сотрудников: {stats['total_employees']}")
print(f"   • Месячные расходы: {stats['total_monthly_cost']:.2f} руб.")

# B. Сотрудники в нескольких проектах
print("\nB.   СОТРУДНИКИ В НЕСКОЛЬКИХ ПРОЕКТАХ")
busy_employees = company.get_employees_in_multiple_projects()
if busy_employees:
    for emp in busy_employees:
        print(f"   • {emp.name} (ID: {emp.id}) - участвует в нескольких проектах")
else:
    print("   Нет сотрудников в нескольких проектах")

# C. Перевод сотрудника
print("\nC.   ПЕРЕВОД СОТРУДНИКА МЕЖДУ ОТДЕЛАМИ")
try:
    company.transfer_employee(6, "Development")  # Frank из Marketing в Development
    print(f"     Frank Brown переведен из Marketing в Development")
except Exception as e:
    print(f"     Ошибка перевода: {e}")

# D. Изменение статуса проекта
print("\nD.   ИЗМЕНЕНИЕ СТАТУСА ПРОЕКТА")
try:
    company.update_project_status(103, "active")  # planning -> active
    print(f"     Статус проекта 'Mobile App' изменен на 'active'")
except Exception as e:
    print(f"     Ошибка: {e}")

# 6. ИНФОРМАЦИЯ О ПРОЕКТАХ
print("\n6. 📊 ПОДРОБНАЯ ИНФОРМАЦИЯ О ПРОЕКТАХ")
print("-" * 45)

for project in company.get_projects():
```

```
    print(f"\n {project.name} (ID: {project.project_id}):")
    print(f"   • Статус: {project.status}")
    print(f"   • Бюджет: {project.budget:.2f} руб.")
    print(f"   • Команда: {project.get_team_size()} сотрудников")
    print(f"   • Зарплаты команды: {project.calculate_total_salary():.2f} руб.")

    if project.get_team_size() > 0:
        print(f"   • Состав команды:")
        for member in project.get_team():
            print(f"     - {member.name} ({member.__class__.__name__})")

# 7. СЕРИАЛИЗАЦИЯ И СОХРАНЕНИЕ
print("\n7.   СЕРИАЛИЗАЦИЯ И СОХРАНЕНИЕ ДАННЫХ")
print("-" * 45)

try:
    # Сохраняем компанию в файл
    os.makedirs("output", exist_ok=True)
    company.save_to_file("output/techinnovations.json")

    # Загружаем обратно
    print("  Загружаем компанию из файла...")
    loaded_company = Company.from_json("output/techinnovations.json")

    print(f"  Данные успешно сохранены и загружены!")
    print(f"   Оригинальная компания: {company.name}")
    print(f"   Загруженная компания: {loaded_company.name}")
    print(f"   Совпадают: {company.name == loaded_company.name}")

except Exception as e:
    print(f"  Ошибка сериализации: {e}")

# 8. ИТОГИ
print("\n" + "=" * 70)
print("ИТОГИ ДЕМОНСТРАЦИИ")
print("=" * 70)

print(f"\n  КОМПАНИЯ: {company.name}")
print(f" ФИНАЛЬНАЯ СТАТИСТИКА:")
print(f"   • Отделов: {len(company.get_departments())}")
print(f"   • Проектов: {len(company.get_projects())}")
print(f"   • Сотрудников: {len(company.get_all_employees())}")
print(f"   • Месячный фонд зарплат: {company.calculate_total_monthly_cost():.2f} руб.")

# Сводка по отделам
print(f"\n  СТРУКТУРА ОТДЕЛОВ:")
for dept in company.get_departments():
    dept_stats = dept.get_statistics() if hasattr(dept, 'get_statistics') else {}
    emp_count = len(dept)
    print(f"   • {dept.name}: {emp_count} сотрудников")

# Сводка по проектам
print(f"\n АКТИВНЫЕ ПРОЕКТЫ:")
```

```
    for project in company.get_projects():
        if project.status == "active":
            print(f"   • {project.name}: {project.get_team_size()} сотрудников, "
                  f"бюджет: {project.budget:.2f} руб.")

    print(f"\n  ДЕМОНСТРАЦИЯ УСПЕШНО ЗАВЕРШЕНА!")
    print(f"  Данные сохранены в папке 'output/'")

if __name__ == "__main__":
    main()
```

Результат тестирования

```
======================================================================
ПОЛНАЯ ДЕМОНСТРАЦИЯ РАБОТЫ С КОМПАНИЕЙ
======================================================================

1.   СОЗДАНИЕ КОМПАНИИ И ОТДЕЛОВ
------------------------------------------
  Создана компания: Company 'TechInnovations' (Отделов: 0, Проектов: 0, Сотрудников: 0, Месяч
Отдел 'Development' добавлен в компанию 'TechInnovations'
Отдел 'Sales' добавлен в компанию 'TechInnovations'
Отдел 'Marketing' добавлен в компанию 'TechInnovations'
📊 Всего отделов: 3

2.   СОЗДАНИЕ И ДОБАВЛЕНИЕ СОТРУДНИКОВ
------------------------------------------
  В отдел 'Development' добавлено 3 сотрудников
  В отдел 'Sales' добавлено 2 сотрудников
  В отдел 'Marketing' добавлено 1 сотрудников

3. 🏷️ СОЗДАНИЕ И ДОБАВЛЕНИЕ ПРОЕКТОВ
------------------------------------------
Проект 'AI Platform' (ID: 101) добавлен в компанию 'TechInnovations'
Проект 'E-Commerce Website' (ID: 102) добавлен в компанию 'TechInnovations'
Проект 'Mobile App' (ID: 103) добавлен в компанию 'TechInnovations'
📊 Всего проектов: 3

4.   ФОРМИРОВАНИЕ КОМАНД ПРОЕКТОВ
------------------------------------------
Сотрудник Bob Smith добавлен в проект 'AI Platform'
Сотрудник Bob Smith назначен на проект 'AI Platform'
Сотрудник Alice Johnson добавлен в проект 'AI Platform'
Сотрудник Alice Johnson назначен на проект 'AI Platform'
Сотрудник Carol Davis добавлен в проект 'AI Platform'
Сотрудник Carol Davis назначен на проект 'AI Platform'
Сотрудник Bob Smith добавлен в проект 'E-Commerce Website'
Сотрудник Bob Smith назначен на проект 'E-Commerce Website'
Сотрудник Carol Davis добавлен в проект 'E-Commerce Website'
Сотрудник Carol Davis назначен на проект 'E-Commerce Website'
Сотрудник David Wilson добавлен в проект 'E-Commerce Website'
Сотрудник David Wilson назначен на проект 'E-Commerce Website'
```

⚠ Не удалось назначить сотрудника 1 на проект 103: Сотрудника можно назначать только на акти

5. ДЕМОНСТРАЦИЯ ОСНОВНЫХ ВОЗМОЖНОСТЕЙ

A. 📈 СТАТИСТИКА КОМПАНИИ
- Компания: TechInnovations
- Отделов: 3
- Проектов: 3
- Сотрудников: 6
- Месячные расходы: 52050.00 руб.

B. СОТРУДНИКИ В НЕСКОЛЬКИХ ПРОЕКТАХ
- Bob Smith (ID: 2) - участвует в нескольких проектах
- Carol Davis (ID: 3) - участвует в нескольких проектах

C. ПЕРЕВОД СОТРУДНИКА МЕЖДУ ОТДЕЛАМИ
Сотрудник Frank Brown (ID: 6) удален из отдела 'Marketing'
Сотрудник Frank Brown переведен из отдела 'Marketing' в отдел 'Development'
Frank Brown переведен из Marketing в Development

D. ИЗМЕНЕНИЕ СТАТУСА ПРОЕКТА
Статус проекта 'Mobile App' изменен с 'planning' на 'active'
Статус проекта 'Mobile App' изменен на 'active'

6. 📊 ПОДРОБНАЯ ИНФОРМАЦИЯ О ПРОЕКТАХ

🏷 AI Platform (ID: 101):
- Статус: active
- Бюджет: 250000.00 руб.
- Команда: 3 сотрудников
- Зарплаты команды: 25750.00 руб.
- Состав команды:
  - Bob Smith (Developer)
  - Alice Johnson (Manager)
  - Carol Davis (Developer)

🏷 E-Commerce Website (ID: 102):
- Статус: active
- Бюджет: 120000.00 руб.
- Команда: 3 сотрудников
- Зарплаты команды: 29750.00 руб.
- Состав команды:
  - Bob Smith (Developer)
  - Carol Davis (Developer)
  - David Wilson (Salesperson)

🏷 Mobile App (ID: 103):
- Статус: active
- Бюджет: 180000.00 руб.
- Команда: 0 сотрудников
- Зарплаты команды: 0.00 руб.

```
7.   СЕРИАЛИЗАЦИЯ И СОХРАНЕНИЕ ДАННЫХ
------------------------------------------
Компания сохранена в файл: output/techinnovations.json
Размер файла: 8678 байт
Компания сохранена в файл: output/techinnovations_readable.json
Размер файла: 11206 байт

Статистика сохранения:
Основной файл: output/techinnovations.json
Читаемая версия: output/techinnovations_readable.json
Данные: 3 отделов, 3 проектов, 6 сотрудников
  Загружаем компанию из файла...
Загружаем из файла: output/techinnovations.json
Отдел: Development (4 сотрудников)
Отдел: Sales (2 сотрудников)
Отдел: Marketing (0 сотрудников)
Проект: AI Platform (ID: 101)
Проект: E-Commerce Website (ID: 102)
Проект: Mobile App (ID: 103)

Компания 'TechInnovations' успешно загружена
Отделов: 3
Проектов: 3
Сотрудников: 6
  Данные успешно сохранены и загружены!
   Оригинальная компания: TechInnovations
   Загруженная компания: TechInnovations
   Совпадают: True

======================================================================
ИТОГИ ДЕМОНСТРАЦИИ
======================================================================

  КОМПАНИЯ: TechInnovations
📊 ФИНАЛЬНАЯ СТАТИСТИКА:
   • Отделов: 3
   • Проектов: 3
   • Сотрудников: 6
   • Месячный фонд зарплат: 52050.00 руб.

  СТРУКТУРА ОТДЕЛОВ:
   • Development: 4 сотрудников
   • Sales: 2 сотрудников
   • Marketing: 0 сотрудников

🏷 АКТИВНЫЕ ПРОЕКТЫ:
   • AI Platform: 3 сотрудников, бюджет: 250000.00 руб.
   • E-Commerce Website: 3 сотрудников, бюджет: 120000.00 руб.
   • Mobile App: 0 сотрудников, бюджет: 180000.00 руб.

  ДЕМОНСТРАЦИЯ УСПЕШНО ЗАВЕРШЕНА!
  Данные сохранены в папке 'output/'
```

Итоги тестирования

- Композиция и агрегация реализованы корректно
- Валидация предотвращает невалидные операции
- Кастомные исключения обрабатываются правильно
- Сериализация сохраняет все связи между объектами

Заключение

Достигнутые результаты

1. Разработана полнофункциональная система учета сотрудников компании
2. Применены все принципы ООП:
    - **Инкапсуляция:** Приватные атрибуты и свойства с валидацией
    - **Наследование:** Иерархия классов с абстрактным базовым классом
    - **Полиморфизм:** Единый интерфейс для работы с разными типами сотрудников
    - **Композиция и агрегация:** Правильное управление жизненным циклом объектов
3. Реализованы магические методы для удобной работы с объектами
4. Обеспечена полная сериализация/десериализация системы
5. Создана расширяемая и поддерживаемая архитектура

Преимущества реализованного решения

- **Гибкость:** Легкое добавление новых типов сотрудников через наследование
- **Масштабируемость:** Поддержка большого количества сотрудников, отделов и проектов
- **Безопасность:** Валидация данных на всех уровнях
- **Удобство:** Магические методы делают работу с объектами интуитивной
- **Персистентность:** Полная поддержка сохранения и загрузки состояния

Возможности дальнейшего развития

- Интеграция с базой данных
- Добавление веб-интерфейса
- Реализация паттернов проектирования (Factory, Builder, Observer и др.)
- Добавление модуля отчетности с графиками
- Интеграция с системами аутентификации и авторизации
- Добавление логирования операций
- Реализация многопоточности для обработки больших объемов данных