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-24-2 **Дисциплина:** Технологии программирования **Студент:** Пахомов Давид Вадимович

Оглавление

Введение

Цель работы

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

Используемые технологии

- **Язык программирования:** Python 3.8+
- **Инструменты:** Стандартная библиотека Python (abc, json, csv, datetime)
- **Система контроля версий:** Git

Структура проекта

```
python-lab3/
├── src/                          # Исходный код системы
│   ├── core/                     # Основные классы системы
│   │   ├── __init__.py
│   │   ├── abstract_employee.py  # Абстрактный класс AbstractEmployee
│   │   ├── employee.py           # Базовый класс Employee
│   │   ├── department.py         # Класс Department
│   │   ├── company.py            # Класс Company
│   │   └── project.py            # Класс Project
│   ├── employees/                # Классы сотрудников
│   │   ├── __init__.py
│   │   ├── manager.py            # Класс Manager
│   │   ├── developer.py         # Класс Developer
│   │   └── salesperson.py       # Класс Salesperson
│   ├── factories/                # Фабрики
│   │   ├── __init__.py
│   │   └── employee_factory.py   # EmployeeFactory
│   └── utils/                    # Вспомогательные модули
│       ├── __init__.py
│       ├── exceptions.py        # Кастомные исключения
│       └── comparators.py        # Компараторы для сортировки
├── examples/                     # Примеры использования
│   ├── demo_part1.py             # Демо Part 1: Инкапсуляция
│   ├── demo_part2.py             # Демо Part 2: Наследование
│   ├── demo_part3.py             # Демо Part 3: Полиморфизм
│   └── demo_part4.py             # Демо Part 4: Композиция
├── data/                         # Данные
│   ├── json/                     # JSON файлы
│   └── csv/                      # CSV отчеты
├── requirements.txt              # Зависимости проекта
├── README.md                     # Описание проекта
└── main.py                       # Основной скрипт для запуска
```

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(AbstractEmployee):
    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
        # Валидация при инициализации
        self._validate_id(id)
        self._validate_name(name)
        self._validate_base_salary(base_salary)

    @property
    def id(self) -> int:
        """Получить ID сотрудника."""
        return self.__id

    @id.setter
    def id(self, value: int) -> None:
        """Установить ID сотрудника."""
        self._validate_id(value)
        self.__id = value
```

Пример использования

```
 # Создание сотрудника
emp1 = Employee(1, "Иван Иванов", "IT", 50000.0)
print(emp1)  # Сотрудник [id: 1, имя: Иван Иванов, отдел: IT, базовая зарплата: 50000.0]

# Работа с свойствами
print(emp1.name)  # Иван Иванов
emp1.name = "Иван Петров"
print(emp1.calculate_salary())  # 50000.0

# Демонстрация валидации
try:
    emp1.id = -5  # ValueError: ID должен быть положительным целым числом
except ValueError as e:
    print(e)
```

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)

Диаграмма классов

```
AbstractEmployee (ABC)
    │
    ├── Employee
    │
    ├── Manager (наследует Employee)
    │   └── calculate_salary(): base_salary + bonus
    │
    ├── Developer (наследует Employee)
    │   ├── calculate_salary(): base_salary * coefficient
    │   └── add_skill(skill: str)
    │
    └── Salesperson (наследует Employee)
        ├── calculate_salary(): base_salary + (sales_volume * commission_rate)
        └── update_sales(new_sales: float)
```

Пример использования

```
 # Создание сотрудников разных типов
manager = Manager(1, "Алиса", "MANAGEMENT", 70000.0, 20000.0)
developer = Developer(2, "Боб", "DEV", 50000.0, ["Python", "Java"], "senior")
salesperson = Salesperson(3, "Чарли", "SALES", 40000.0, 0.15, 100000.0)

# Расчет зарплат (полиморфизм)
print(manager.calculate_salary())    # 90000.0 (70000 + 20000)
print(developer.calculate_salary())  # 100000.0 (50000 * 2.0 для senior)
print(salesperson.calculate_salary()) # 55000.0 (40000 + 100000 * 0.15)

# Использование фабрики
emp = EmployeeFactory.create_employee(
    "developer",
    id=10,
    name="Фабричный Разработчик",
    department="DEV",
    base_salary=45000.0,
    tech_stack=["Python", "JavaScript"],
    seniority_level="middle"
)
print(emp.calculate_salary())  # 67500.0 (45000 * 1.5 для middle)
```

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__`
- Добавлена поддержка сериализации/десериализации в JSON
- Реализована итерация по объектам (отдел и стек технологий разработчика)
- Созданы компараторы для сортировки сотрудников

Примеры реализации

```
# Перегрузка операторов для сотрудников
emp1 = Employee(1, "Анна", "IT", 50000.0)
emp2 = Manager(2, "Борис", "MANAGEMENT", 70000.0, 20000.0)

print(emp1 == emp2)        # False (сравнение по ID)
print(emp1 < emp2)         # True (сравнение по зарплате)
print(emp1 + emp2)         # 140000.0 (сумма зарплат)

# Полиморфизм в коллекциях
department = Department("Разработка")
department.add_employee(emp1)
department.add_employee(emp2)
department.add_employee(Developer(3, "Виктор", "DEV", 50000.0, ["Python"], "senior"))

total_salary = department.calculate_total_salary()  # Работает с разными типами
print(total_salary)  # 240000.0

# Магические методы для отдела
print(len(department))      # 3
print(department[0].name)   # Анна
print(emp1 in department)   # True

# Итерация по отделу
for emp in department:
    print(emp.get_info())

# Сериализация
department.save_to_file("data/json/department.json")
loaded_dept = Department.load_from_file("data/json/department.json")
```

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
- Реализован экспорт данных в CSV форматы
- Добавлены комплексные бизнес-методы для анализа данных

Архитектурная схема

```
Company (агрегация)
    ├── Department[] (агрегация)
    │   └── AbstractEmployee[] (агрегация)
    └── Project[] (агрегация)
        └── AbstractEmployee[] (композиция - команда проекта)
```

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

Пример использования

```
 # Создание компании
company = Company("TechInnovations")

# Создание отделов (агрегация)
dev_department = Department("Development")
sales_department = Department("Sales")
company.add_department(dev_department)
company.add_department(sales_department)

# Создание сотрудников
manager = Manager(1, "Алиса Джонсон", "DEV", 70000.0, 20000.0)
developer = Developer(2, "Боб Смит", "DEV", 50000.0, ["Python", "SQL"], "senior")
salesperson = Salesperson(3, "Чарли Браун", "SAL", 40000.0, 0.15, 50000.0)

# Добавление в отделы
dev_department.add_employee(manager)
dev_department.add_employee(developer)
sales_department.add_employee(salesperson)

# Создание проектов (агрегация)
ai_project = Project(101, "AI Platform", "Разработка AI системы",
                     "2024-12-31", "active")
web_project = Project(102, "Web Portal", "Создание веб-портала",
                      "2024-09-30", "planning")

company.add_project(ai_project)
company.add_project(web_project)

# Формирование команд проектов (композиция)
ai_project.add_team_member(developer)
ai_project.add_team_member(manager)
web_project.add_team_member(developer)

# Финансовые показатели
total_cost = company.calculate_total_monthly_cost()
print(f"Общие месячные затраты: {total_cost}")

# Статистика
stats = company.get_department_stats()
budget_analysis = company.get_project_budget_analysis()

# Сериализация
company.save_to_json("data/json/company.json")
company.export_employees_csv("data/csv/employees.csv")
company.export_projects_csv("data/csv/projects.csv")

# Загрузка
loaded_company = Company.load_from_json("data/json/company.json")
```

Результаты тестирования

- ✅ Композиция и агрегация реализованы корректно
- ✅ Валидация предотвращает невалидные операции
- ✅ Кастомные исключения обрабатываются правильно
- ✅ Сериализация сохраняет все связи между объектами
- ✅ Экспорт в CSV работает корректно
- ✅ Бизнес-методы возвращают корректные результаты

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

Выводы

В ходе выполнения лабораторных работ были успешно освоены все основные принципы объектно-ориентированного программирования на языке Python. Реализованная система демонстрирует правильное применение инкапсуляции, наследования, полиморфизма, композиции и агрегации. Код структурирован, документирован и готов к дальнейшему развитию.

Приложения

Приложение A: Примеры использования

Все примеры использования находятся в папке `examples/`:

- `demo_part1.py` - Демонстрация инкапсуляции
- `demo_part2.py` - Демонстрация наследования и абстракции
- `demo_part3.py` - Демонстрация полиморфизма и магических методов
- `demo_part4.py` - Демонстрация композиции и агрегации

Приложение B: Структура классов

Основные классы:

- `AbstractEmployee` - Абстрактный базовый класс
- `Employee` - Базовый класс сотрудника
- `Manager` - Менеджер (наследник Employee)
- `Developer` - Разработчик (наследник Employee)
- `Salesperson` - Продавец (наследник Employee)
- `Department` - Отдел компании
- `Project` - Проект компании
- `Company` - Компания

Вспомогательные классы:

- `EmployeeFactory` - Фабрика для создания сотрудников
- Кастомные исключения: `EmployeeNotFoundError`, `DepartmentNotFoundError`, `ProjectNotFoundError`, `InvalidStatusError`, `DuplicateIdError`

Приложение C: Ключевые методы

Employee:

- `calculate_salary()` - Расчет зарплаты
- `get_info()` - Полная информация о сотруднике
- `to_dict()` / `from_dict()` - Сериализация

Department:

- `add_employee()` / `remove_employee()` - Управление сотрудниками
- `calculate_total_salary()` - Общая зарплата отдела
- `get_employee_count()` - Статистика по типам
- `save_to_file()` / `load_from_file()` - Сериализация

Company:

- `add_department()` / `remove_department()` - Управление отделами
- `add_project()` / `remove_project()` - Управление проектами
- `calculate_total_monthly_cost()` - Общие затраты
- `get_department_stats()` - Статистика по отделам
- `get_project_budget_analysis()` - Анализ бюджетов
- `save_to_json()` / `load_from_json()` - Сериализация
- `export_employees_csv()` / `export_projects_csv()` - Экспорт отчетов

Список использованных источников

1. Роберт Мартин. "Чистый код. Создание, анализ и рефакторинг"
2. Мартин Фаулер. "Рефакторинг. Улучшение существующего кода"
3. Эрик Гамма и др. "Паттерны объектно-ориентированного проектирования"
4. Документация Python: https://docs.python.org/3/
5. PEP 8 -- Style Guide for Python Code: https://pep8.org/
6. Python Data Model: https://docs.python.org/3/reference/datamodel.html

---

Дата завершения работы: 2025-11-17