Отчёт по лабораторной работе

Тема: Организация рабочего окружения и работа с Git. Стратегии ветвления и автоматизация контроля качества.

Сведения о студенте

Дата: 2025-10-24 **Семестр:** 3 **Группа:** ПИН-б-о-24-1 **Дисциплина:** Технологии программирования
Студент: Макаров Роман Дмитриевич

Оглавление

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

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.x
- **Инструменты:** Стандартная библиотека Python
- **Система контроля версий:** Git

Структура проекта

```
employee_management_system/
├── 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 и порождающие паттерны
│   │   ├── __init__.py
│   │   └── employee_factory.py   # EmployeeFactory
│   │
│   └── utils/                    # Вспомогательные модули
│       ├── __init__.py
│       ├── comparators.py        # Компараторы
│       └── exceptions.py         # Кастомные исключения
│
├── data/                         # Данные для тестирования
│   ├── json/                     # JSON файлы для сериализации
│   └── csv/                      # CSV отчеты
│
├── 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
│
├── 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
- Реализованы свойства (property) для доступа к данным
- Добавлена валидация входных параметров

- Реализован метод `__str__` для строкового представления

Ключевые элементы реализации

```
class Employee:
    """Обычный сотрудник без дополнительных параметров."""
    def __init__(self, id: int, name: str, department: str, base_salary: float):
        self.__id = id
        self.__name = name
        self.__departament = department
        self.__base_salary = base_salary

    # --- Свойства (property) с проверкой корректности данных ---

    @property
    def id(self):
        """Возвращает идентификатор сотрудника."""
        return self.__id

    @id.setter
    def id(self, value):
        """Устанавливает идентификатор сотрудника с проверкой."""
        if not isinstance(value, int):
            raise ValueError("Идентификатор должен быть целым числом!")
        if value <= 0:
            raise ValueError("Идентификатор должен быть положительным числом!")
        self.__id = value
```

Пример использования

```
    # Создание объекта из класса Employee
    employee = Employee(
    id=1,
    name="Виктор",
    department="IT",
    base_salary=60000.0
    )
    print(f'  {employee}')

    # Работа со свойствами
    print(employee.id)  # 1 (до изменения)
    employee.id = 100
    print(employee.id)  # 100 (после изменения)
    print(emp1.calculate_salary())  # 60000.0

    # Демонстрация валидации
    try:
        employee.id = '123'  # Невалидный ID
    except ValueError as e:
        print(f"  Ошибка при установке невалидного ID: {e}")
```

Результаты тестирования

- Протестирована корректная установка и получение значений
- Проверена обработка невалидных данных
- Убедились в корректности строкового представления

Лабораторная работа 4.2: Наследование и абстракция

Цель

Создание иерархии классов сотрудников на основе наследования и абстракции.

Выполненные задачи

- Создан абстрактный класс `AbstractEmployee`
- Реализованы классы-наследники: `Manager`, `Developer`, `Salesperson`
- Реализована фабрика сотрудников `EmployeeFactory`
- Переопределены методы расчета зарплат

Диаграмма классов

```
AbstractEmployee (ABC)
    |
    └── Employee
        |
        ├── Manager (наследует Employee)
        |   └── calculate_salary(): base_salary + bonus
        |
        ├── Developer (наследует Employee)
        |   ├── calculate_salary(): base_salary * coefficient  (junior/middle/senior)
        |   └── add_skill(new_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(2, "Максим", "MANAGEMENT", 60000.0, 10000.0)
    developer = Developer(3, "Даниил", "DEV", 80000.0, ["JS", "GO", "C"], "senior")
    salesperson = Salesperson(4, "Павел", "SALES", 30000.0, 0.2, 200000.0)

    # Получение основной информации о сотрудниках
    print(f"  {developer.get_info()}")  # Менеджер [id: 2, имя: Максим, отдел: MANAGEMENT, ба
```

```
# Демонстрация расчета зарплат
print(manager.calculate_salary()) # 65000.0
print(developer.calculate_salary()) # 120000.0
print(salesperson.calculate_salary()) # 67500.0

# Использование фабрики
dev = EmployeeFactory.create_employee(
    "developer",
    id=1,
    name="Олег",
    department="DEV",
    base_salary=70000.0,
    tech_stack=["GO", "C++"],
    seniority_level="senior"
)

# Демонстрация полиморфного поведения
print(f'\n{' Демонстрация полиморфного поведения ':-^60}\n')
employees_list = [manager, developer, salesperson, dev]
for emp in employees_list:
    print(emp.name) # Максим  Даниил  Павел  Олег
```

Результаты тестирования

- Все классы корректно наследуются от `AbstractEmployee`
- Абстрактные методы реализованы и работают корректно во всех классах-наследниках
- Полиморфизм работает корректно в коллекциях
- Фабрика работает корректно

Лабораторная работа 4.3: Полиморфизм и магические методы

Цель

Реализация полиморфного поведения и перегрузки операторов.

Выполненные задачи

- Создан класс `Department` для управления сотрудниками
- Реализованы магические методы для сотрудников и отделов
- Добавлена поддержка сериализации/десериализации
- Реализована итерация по объектам

Примеры реализации

```
# Создание сотрудников разных типов
employee = Employee(1, "Евгений", "ADMIN", 30000.0)
manager = Manager(2, "Максим", "MANAGEMENT", 60000.0, 10000.0)

# Создание отдела
department = Department("IT")
department.add_employee(employee)
department.add_employee(manager)
print(department.name) # IT

# Суммирование через sum()
employees = [employee, manager]
total_salary = sum(employees)
print(total_salary) # 100000.0

# Использование перегруженных операторов
print(employee == manager) # False
print(employee < manager)  # True
print(employee + manager)  # 100000.0

# Магические методы для отдела
print(len(department))        # 2
print(department[0].name)     # Евгений
print(employee in department) # True

# Итерация по отделу
for emp in department:
    print(emp.name) # Евгений  Максим

    # Сериализация и десериализация
# Сохранение отдела в файл .json
department.save_to_file('department.json')

# Загрузка отдела из файла .json
test_department = Department.load_from_file('department_load.json')
```

Результаты тестирования

- Созданный класс `Department` и его методы работают корректно
- Магические методы для сотрудников и отделов работают корректно
- 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

Построение сложных объектных структур с использованием композиции и агрегации.

Выполненные задачи

- Создан класс `Project` с композицией сотрудников
- Реализован класс `Company` с агрегацией отделов и проектов
- Добавлена система валидации и кастомные исключения
- Реализована комплексная сериализация системы

Пример реализации

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

# Создание отделов
dev_department = Department("Development")
sales_department = Department("Sales")

# Добавление отделов в компанию
company.add_department(dev_department)
company.add_department(sales_department)

# Создание сотрудников
manager = Manager(1, "Alice Johnson", "DEV", 7000, 2000)
developer = Developer(2, "Bob Smith", "DEV", 5000, ["Python", "SQL"], "senior")
salesperson = Salesperson(3, "Charlie Brown", "SAL", 4000, 0.15, 50000)

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

# Создание проектов
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"

# Добавление проектов в компанию
company.add_project(ai_project)
company.add_project(web_project)

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

# Валидация и обработка ошибок
try:
    company.add_project(Project(101, "Duplicate", "Test", "2024-12-31", "planning"))
except DuplicateIdError as e:
    print(f"    Ошибка при добавлении проекта с дублирующимся ID: {e}")
```

```
    # Сериализация компании
# Сохранение компании в company.json
company.save_to_file('company.json')

# Загрузка компании из company_load.json
test_company = Company.load_from_file('company_load.json')

    # Экспорт отчетов
# Экспорт отчета по сотрудникам
company.export_employees_csv('employees.csv')

# Экспорт отчета по проектам
company.export_projects_csv('projects.csv')
```

Архитектурная схема

```
Company
    ├── departments: list[Department]
    │       │
    │       └── Department
    │            └── employees: list[AbstractEmployee]
    │                   └── AbstractEmployee → Employee → {Manager, Developer, Salesperson}
    │
    └── projects: list[Project]
            │
            └── Project
                 └── team: list[AbstractEmployee]  (те же Manager / Developer / Salesperson)
```

Заключение

Достигнутые результаты

1. Разработана функциональная система учета сотрудников
2. Применены все принципы ООП: инкапсуляция, наследование, полиморфизм
3. Обеспечена полная сериализация/десериализация системы

Преимущества реализованного решения

- **Гибкость:** Легкое добавление новых типов сотрудников
- **Масштабируемость:** Поддержка большого количества сотрудников и отделов
- **Поддерживаемость:** Чистая архитектура и документация

Возможности дальнейшего развития

- Интеграция с веб-интерфейсом

- Добавление модуля отчетности
- Поддержка распределенной архитектуры
- Интеграция с системами аутентификации
- Реализация паттернов проектирования

Приложения

Приложение А: Примеры использования

`employee_management_system/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

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. Документация pytest: https://docs.pytest.org/