Лабораторная работа 4: Реализация принципов ООП на примере системы учета сотрудников

Цель работы: Освоить базовые принципы объектно-ориентированного программирования (ООП) на языке Python. Получить практические навыки создания иерархии классов, применения принципов инкапсуляции, наследования, полиморфизма и композиции для моделирования предметной области.

Стек технологий:

- **Язык программирования:** Python 3.x
- **Инструменты:** Любая IDE (PyCharm, VSCode и т.д.), Git для контроля версий (если работа ведется в рамках общего проекта).

Теоретическая часть (краткое содержание из лекции): Объектно-ориентированное программирование — это парадигма, использующая объекты, содержащие данные и методы для их обработки. Ключевыми принципами ООП являются:

- **Инкапсуляция:** Сокрытие внутренней реализации объекта и предоставление строго определенного интерфейса для взаимодействия с ним.
- **Наследование:** Создание нового класса (потомка) на основе существующего (родителя), что позволяет повторно использовать код и выстраивать иерархии.
- **Полиморфизм:** Возможность объектов с одинаковой спецификацией (интерфейсом) иметь различную реализацию.
- **Композиция:** Построение сложных объектов из более простых, где один класс включает в себя объекты других классов.

Предметная область: Система учета сотрудников компании.

Задание на практическую реализацию:

Работа разбита на 4 логически связанные части. Рекомендуется выполнять их последовательно, постепенно расширяя и модифицируя код.

Часть 1: Инкапсуляция

Цель: Реализовать базовый класс `Employee` (Сотрудник), инкапсулирующий данные о сотруднике.

Задание:

1. Создайте класс `Employee`.
2. В конструкторе класса (`__init__`) определите следующие **приватные (`private`)** атрибуты:
    - `__id` (уникальный идентификатор сотрудника, целое число)
    - `__name` (имя сотрудника, строка)
    - `__department` (отдел, строка)
    - `__base_salary` (базовая зарплата, вещественное число)
3. Для каждого из этих атрибутов реализуйте **методы доступа (геттеры и сеттеры)**, используя декоратор `@property` и `@<attribute_name>.setter`.
    - Геттер возвращает значение атрибута.

    - Сеттер устанавливает значение атрибута, проводя базовые проверки (например, `id` и `base_salary` должны быть положительными числами, `name` не должно быть пустой строкой). В случае невалидных данных необходимо возбуждать исключение `ValueError` с описанием ошибки.
4. Реализуйте метод `__str__` для возврата строкового представления объекта в формате: `"Сотрудник [id: {id}, имя: {name}, отдел: {department}, базовая зарплата: {base_salary}]"`.
5. **Проверка:** В основной программе (`if __name__ == "__main__":`) создайте несколько объектов класса `Employee`. Продемонстрируйте:
    - Корректную установку и получение значений через свойства.
    - Попытку установки невалидных значений (должно вызывать исключение).
    - Вывод объектов на экран.

Критерии оценки части 1:

- Создан класс с приватными атрибутами.
- Корректно реализованы свойства для доступа к атрибутам с валидацией.
- Продемонстрирована работа с объектами, включая обработку ошибок.
- Составлена UML-диаграмма классов https://plantuml.com/ru/

(Последующие части — Наследование, Полиморфизм, Композиция — будут представлены в следующих заданиях и логически продолжат развитие этой же кодобазы.)

Часть 2: Наследование и Абстракция

Цель: Расширить систему учета сотрудников, создав иерархию классов на основе наследования. Ввести абстрактные классы для определения общего интерфейса.

Задание:

2.1. Создание абстрактного базового класса

1. Создайте **абстрактный класс** `AbstractEmployee`, который наследуется от `ABC` (Abstract Base Class) из модуля `abc`.
2. Перенесите в него общие атрибуты и методы из класса `Employee` (теперь он станет потомком этого класса). В `AbstractEmployee` объявите следующие **абстрактные методы** с помощью декоратора `@abstractmethod`:
   - `def calculate_salary(self) -> float:` - для расчета итоговой заработной платы. Реализация будет разной для разных типов сотрудников.
   - `def get_info(self) -> str:` - для получения полной информации о сотруднике.
3. Сделайте класс `Employee` из Части 1 наследником `AbstractEmployee`. Реализуйте в нем эти абстрактные методы:
   - `calculate_salary`: для обычного сотрудника итоговая зарплата равна базовой.
   - `get_info`: должен возвращать строку, включающую всю информацию из `__str__`, а также рассчитанную зарплату.

2.2. Создание иерархии классов сотрудников Создайте три новых класса, наследующихся от `Employee`:

- `Manager` (Менеджер)
- `Developer` (Разработчик)
- `Salesperson` (Продавец)

Каждый класс должен иметь **дополнительные приватные атрибуты** и **переопределять** метод `calculate_salary` по своему.

- **Для `Manager`:**
  - Доп. атрибут: `__bonus` (бонус, вещественное число).
  - Реализация `calculate_salary`: `базовая_зарплата + бонус`.
  - Переопределите также метод `get_info`, чтобы он включал информацию о бонусе.
- **Для `Developer`:**
  - Доп. атрибуты: `__tech_stack` (список технологий, которыми владеет разработчик, list[str]), `__seniority_level` (уровень Seniority, строка, например, "junior", "middle", "senior").
  - Реализация `calculate_salary`: Базовая зарплата умножается на коэффициент уровня:
    - "junior": x1.0
    - "middle": x1.5
    - "senior": x2.0
  - Реализуйте метод `add_skill(self, new_skill: str) -> None` для добавления новой технологии в стек.

    - Переопределите `get_info`, чтобы он включал уровень и стек технологий.

- **Для `Salesperson`:**

    - Доп. атрибуты: `__commission_rate` (процент комиссии, вещественное число, например, 0.1 для 10%), `__sales_volume` (объем продаж, вещественное число).
    - Реализация `calculate_salary`: `базовая_зарплата + (объем_продаж * процент_комиссии)`.
    - Реализуйте метод `update_sales(self, new_sales: float) -> None`, который добавляет сумму к текущему объему продаж.
    - Переопределите `get_info`, чтобы он включал процент комиссии и объем продаж.

2.3. Фабричный метод (Дополнительная сложность)

1. Создайте класс `EmployeeFactory`.
2. Реализуйте в нем статический метод `create_employee(emp_type: str, **kwargs) -> AbstractEmployee`.
3. Метод должен на основе строкового параметра `emp_type` ("manager", "developer", "salesperson", "employee") и переданных именованных аргументов создавать и возвращать соответствующий объект сотрудника.
4. Продемонстрируйте работу фабрики, создавая разные типы сотрудников.

2.4. Тестирование и демонстрация В основной программе создайте и продемонстрируйте:

1. По одному экземпляру каждого типа сотрудника (`Manager`, `Developer`, `Salesperson`, `Employee`).
2. Для каждого экземпляра:
    - Установите значения атрибутов (в т.ч. через сеттеры).
    - Вызовите метод `calculate_salary()` и выведите результат.
    - Вызовите метод `get_info()` и выведите результат.
3. Продемонстрируйте работу фабричного метода, создав с его помощью несколько сотрудников.
4. Создайте список (`list`) или словарь (`dict`) с объектами разных типов и итерируйтесь по нему, выводя для каждого информацию через `get_info()`, демонстрируя полиморфное поведение (которое будет полностью реализовано в Части 3).

Требования к коду:

- Все новые атрибуты должны быть приватными и иметь соответствующие свойства.
- Использовать `super()` для вызова методов родительского класса в конструкторах.
- Код должен быть документирован (docstrings для классов и методов).
- Для проверок в сеттерах использовать утверждения `assert` или возбуждать исключения `ValueError`.

Критерии оценки части 2:

- **Удовлетворительно:** Реализованы и работают классы `Manager`, `Developer`, `Salesperson` с корректным наследованием и переопределением метода `calculate_salary`.
- **Хорошо:** Реализован абстрактный базовый класс `AbstractEmployee`. Все классы реализуют абстрактные методы. Продемонстрирована работа системы.
- **Отлично:** Реализован класс `EmployeeFactory` с фабричным методом. Код хорошо структурирован, задокументирован и использует `super()`. Продемонстрирована работа с

коллекцией объектов. Составлена UML-диаграмма классов https://plantuml.com/ru/

Часть 3: Полиморфизм и Магические методы

Цель: Реализовать полиморфное поведение объектов различных классов сотрудников. Освоить перегрузку операторов и использование магических методов в Python для создания более интуитивного и удобного интерфейса работы с объектами.

Задание:

3.1. Углубленный полиморфизм в коллекциях

1. Создайте класс `Department` (Отдел).
2. Реализуйте в нем следующие методы:
    - `__init__(self, name: str)`: конструктор, инициализирует название отдела и пустой список сотрудников.
    - `add_employee(self, employee: AbstractEmployee) -> None`: добавляет сотрудника в отдел.
    - `remove_employee(self, employee_id: int) -> None`: удаляет сотрудника по ID.
    - `get_employees(self) -> list[AbstractEmployee]`: возвращает список всех сотрудников.
    - `calculate_total_salary(self) -> float`: вычисляет общую зарплату всех сотрудников отдела (демонстрирует полиморфизм - метод работает с разными типами сотрудников).
    - `get_employee_count(self) -> dict[str, int]`: возвращает словарь с количеством сотрудников каждого типа (Manager, Developer, etc.). Для этого используйте функцию `isinstance()` или поле `__class__.__name__`.
    - `find_employee_by_id(self, employee_id: int) -> Optional[AbstractEmployee]`: находит сотрудника по ID.

3.2. Перегрузка операторов (магические методы) Для классов сотрудников реализуйте следующие магические методы:

- `__eq__(self, other) -> bool`: сравнение сотрудников по `id`.
- `__lt__(self, other) -> bool`: сравнение сотрудников по итоговой зарплате (для сортировки).
- `__add__(self, other) -> float`: сложение двух сотрудников возвращает сумму их зарплат.
- `__radd__(self, other) -> float`: поддержка суммирования в списке через `sum()` (например, `total = sum(employees_list)`).
- Для класса `Department` реализуйте:
    - `__len__(self) -> int`: возвращает количество сотрудников в отделе.
    - `__getitem__(self, key) -> AbstractEmployee`: доступ к сотруднику по индексу (например, `dept[0]`).
    - `__contains__(self, employee: AbstractEmployee) -> bool`: проверка принадлежности сотрудника отделу.

3.3. Итерация по объектам

1. Реализуйте для класса `Department` метод `__iter__`, чтобы сделать его итерируемым (можно возвращать итератор списка сотрудников).
2. Реализуйте для класса `Developer` метод `__iter__`, который будет позволять итерироваться по его стеку технологий (например, `for skill in developer:`).

3.4. Сериализация и десериализация

1. Реализуйте для всех классов сотрудников метод `to_dict(self) -> dict`, который возвращает словарь с данными сотрудника.
2. Реализуйте метод класса `from_dict(cls, data: dict) -> AbstractEmployee` для создания объекта из словаря.
3. Реализуйте для класса `Department` методы:
    - `save_to_file(self, filename: str) -> None`: сохраняет всех сотрудников отдела в JSON файл.
    - `load_from_file(cls, filename: str) -> 'Department'`: загружает отдел из JSON файла.

3.5. Компараторы и сортировка

1. Создайте несколько функций-компараторов для сортировки сотрудников:
    - По имени
    - По зарплате
    - По отделу и затем по имени
2. Продемонстрируйте сортировку списка сотрудников с использованием:
    - `sorted()` с ключом (`key=`))
    - `sorted()` с компаратором (`cmp=` в Python 3.2+ через `functools.cmp_to_key`)

3.6. Тестирование и демонстрация В основной программе продемонстрируйте:

1. Создание отдела и добавление в него сотрудников разных типов.
2. Вызов `calculate_total_salary()` для отдела.
3. Использование перегруженных операторов:
    - Сравнение двух сотрудников (`==`, `<`)
    - Суммирование зарплат сотрудников (`employee1 + employee2`)
    - Суммирование списка сотрудников через `sum()`
    - Проверка вхождения сотрудника в отдел (`in`)
    - Доступ к сотрудникам отдела по индексу
4. Итерацию по отделу и по стеку технологий разработчика.
5. Сохранение и загрузку отдела из файла.
6. Сортировку сотрудников по различным критериям.
7. Поиск сотрудника по ID.

Требования к коду:

- Все магические методы должны быть реализованы корректно
- Обрабатывать крайние случаи (пустые значения, неверные типы)
- Использовать полиморфизм везде, где это возможно
- Код должен быть хорошо документирован

Критерии оценки части 3:

- **Удовлетворительно:** Реализованы основные магические методы и полиморфное поведение в коллекциях
- **Хорошо:** Полностью реализована перегрузка операторов, итерация и сериализация
- **Отлично:** Реализованы все методы, включая компараторы для сортировки, обработку крайних случаев и полноценную демонстрацию работы. Составлена UML-диаграмма классов https://plantuml.com/ru/

Часть 4: Композиция, Агрегация и Работа со сложными структурами

Цель: Освоить принципы композиции и агрегации для построения сложных объектных структур. Реализовать механизмы управления связями между объектами, валидации данных и комплексной сериализации/десериализации системы.

Задание:

4.1. Система проектов и композиция

1. Создайте класс `Project` (Проект) с атрибутами:
    - `project_id: int` - уникальный идентификатор проекта
    - `name: str` - название проекта
    - `description: str` - описание проекта
    - `deadline: datetime` - срок выполнения проекта
    - `status: str` - статус проекта ("planning", "active", "completed", "cancelled")
    - `__team: list[AbstractEmployee]` - список сотрудников, работающих над проектом (**композиция**)
2. Реализуйте в классе `Project`:
    - `__init__()` с валидацией входных данных
    - `add_team_member(employee: AbstractEmployee) -> None` - добавление сотрудника в проект
    - `remove_team_member(employee_id: int) -> None` - удаление сотрудника по ID
    - `get_team() -> list[AbstractEmployee]` - получение списка команды
    - `get_team_size() -> int` - получение размера команды
    - `calculate_total_salary() -> float` - расчет суммарной зарплаты команды
    - `get_project_info() -> str` - полная информация о проекте
    - `change_status(new_status: str) -> None` - изменение статуса с валидацией

4.2. Класс Company и агрегация

1. Создайте класс `Company` (Компания) с атрибутами:
    - `name: str` - название компании
    - `__departments: list[Department]` - список отделов (**агрегация**)
    - `__projects: list[Project]` - список проектов (**агрегация**)
2. Реализуйте в классе `Company`:
    - Методы управления отделами: `add_department()`, `remove_department()`, `get_departments()`
    - Методы управления проектами: `add_project()`, `remove_project()`, `get_projects()`
    - `get_all_employees() -> list[AbstractEmployee]` - получение всех сотрудников компании
    - `find_employee_by_id(employee_id: int) -> Optional[AbstractEmployee]` - поиск сотрудника по ID
    - `calculate_total_monthly_cost() -> float` - расчет общих месячных затрат на зарплаты

- `get_projects_by_status(status: str) -> list[Project]` - фильтрация проектов по статусу

4.3. Система валидации и исключения

1. Создайте кастомные исключения:
   - `EmployeeNotFoundError`
   - `DepartmentNotFoundError`
   - `ProjectNotFoundError`
   - `InvalidStatusError`
   - `DuplicateIdError`
2. Реализуйте комплексную валидацию:
   - Проверка уникальности ID сотрудников и проектов
   - Валидация статусов проектов
   - Проверка корректности дат
   - Валидация финансовых показателей

4.4. Управление зависимостями и связями

1. Реализуйте механизм проверки перед удалением:
   - Нельзя удалить отдел, если в нем есть сотрудники
   - Нельзя удалить сотрудника, если он участвует в проектах
   - Нельзя удалить проект, если над ним работает команда
2. Реализуйте методы для переноса сотрудников между отделами

4.5. Расширенная сериализация и сохранение состояния

1. Реализуйте полную сериализацию всей компании в JSON:
   - Сохранение всех отделов, сотрудников и проектов
   - Сохранение связей между объектами
   - Обработка циклических ссылок
2. Реализуйте загрузку компании из JSON:
   - Восстановление объектной структуры
   - Восстановление связей между объектами
   - Валидация загружаемых данных
3. Реализуйте экспорт данных в различные форматы:
   - CSV отчет по сотрудникам с детальной информацией
   - CSV отчет по проектам с составом команд и бюджетами
   - Текстовый отчет по финансовым показателям компании

4.6. Комплексные бизнес-методы

1. Реализуйте методы для анализа данных:

- `get_department_stats() -> dict` - статистика по отделам
- `get_project_budget_analysis() -> dict` - анализ бюджетов проектов
- `find_overloaded_employees() -> list[AbstractEmployee]` - поиск перегруженных сотрудников

2. Реализуйте методы для планирования:

- `assign_employee_to_project(employee_id: int, project_id: int) -> bool`
- `check_employee_availability(employee_id: int) -> bool`

4.7. Тестирование и демонстрация В основной программе продемонстрируйте:

1. **Создание комплексной структуры:**

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

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

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

# Создание сотрудников разных типов
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)
```

2. **Работу с композицией и агрегацией:**
    - Покажите разницу в жизненном цикле объектов
    - Демонстрацию управления связями
3. **Валидацию и обработку ошибок:**
    - Попытку добавить дубликат ID
    - Попытку невалидного изменения статуса
    - Попытку удаления занятого отдела
4. **Сериализацию и экспорт:**

```
# Сохранение всей компании
company.save_to_json("company_data.json")

# Загрузка компании
loaded_company = Company.load_from_json("company_data.json")

# Экспорт отчетов
company.export_employees_csv("employees_report.csv")
company.export_projects_csv("projects_report.csv")
```

5. **Анализ данных:**
    - Получение статистики по компании
    - Расчет финансовых показателей
    - Поиск и анализ различных сущностей

Требования к коду:

- Четкое разделение композиции и агрегации
- Полноценная валидация всех операций
- Корректная обработка исключительных ситуаций
- Полный цикл сериализации/десериализации
- Генерация содержательных отчетов

Критерии оценки части 4:

- **Удовлетворительно:** Реализованы базовые классы Project и Company с основными методами
- **Хорошо:** Реализована валидация данных, основные механизмы управления связями
- **Отлично:** Полная сериализация/десериализация, комплексные бизнес-методы, генерация отчетов, демонстрация работы всей системы. Составлена UML-диаграмма классов https://plantuml.com/ru/