Отчёт по лабораторной работе

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

Сведения о студенте

Дата: 2025-12-21 **Семестр:** 3 **Группа:** ПИН-б-о-24-1 **Дисциплина:** Технологии программирования
Студент: Макаров Роман Дмитриевич

Оглавление

Введение

Цель работы

Разработка комплексной системы учета сотрудников компании с применением принципов объектно-ориентированного программирования, паттернов проектирования и современных практик тестирования.

Используемые технологии

- **Язык программирования:** Python 3.x
- **Фреймворки:** pytest, unittest
- **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
│   │   └── company_factory.py    # AbstractFactory для компаний
│   │
│   ├── patterns/                 # Реализации паттернов проектирования
│   │   ├── __init__.py
│   │   ├── singleton.py          # Singleton для DatabaseConnection
│   │   └── builder.py            # EmployeeBuilder
│   │
│   ├── utils/                    # Вспомогательные модули
│   │   ├── __init__.py
│   │   ├── comparators.py        # Компараторы
│   │   └── exceptions.py         # Кастомные исключения
│   │
│   └── database/                 # Работа с базой данных
│       ├── __init__.py
│       └── connection.py         # Singleton для подключения к БД
│
├── tests/                        # Тесты для всех частей ЛР
│   ├── __init__.py
│   ├── conftest.py               # Фикстуры pytest
│   │
│   ├── test_core/                # Тесты основных классов
│   │   ├── __init__.py
│   │   ├── test_employee.py      # Тесты Part 1: Инкапсуляция
│   │   ├── test_department.py    # Тесты Part 3: Полиморфизм
│   │   └── test_company.py       # Тесты Part 4: Композиция
│   │
│   ├── test_employees/           # Тесты классов сотрудников
│   │   ├── __init__.py
│   │   ├── test_employees_hierarchy.py  # Тесты Part 2: Наследование
│   │   ├── test_manager.py
│   │   ├── test_developer.py
│   │   └── test_salesperson.py
│   │
│   └── test_patterns/            # Тесты паттернов
│       ├── __init__.py
│       ├── test_singleton.py
│       ├── test_factory.py
│       └── test_builder.py
```

```
│
├── data/                          # Данные для тестирования
│   ├── json/                      # JSON файлы для сериализации
│   └── csv/                       # CSV отчеты
│
├── examples/                      # Примеры использования
│   └── demo_part5.py              # Демо Part 5: Паттерны
│
├── README.md                     # Описание проекта
└── main.py                       # Основной скрипт для запуска
```

Лабораторная работа 8: Тестирование программного обеспечения

Цель

Обеспечение качества кода через комплексное тестирование.

Структура тестов

```
├── tests/                         # Тесты для всех частей ЛР
│   ├── __init__.py
│   ├── conftest.py                # Фикстуры pytest
│   │
│   ├── test_core/                 # Тесты основных классов
│   │   ├── __init__.py
│   │   ├── test_employee.py       # Тесты Part 1: Инкапсуляция
│   │   ├── test_department.py     # Тесты Part 3: Полиморфизм
│   │   └── test_company.py        # Тесты Part 4: Композиция
│   │
│   ├── test_employees/            # Тесты классов сотрудников
│   │   ├── __init__.py
│   │   ├── test_employees_hierarchy.py  # Тесты Part 2: Наследование
│   │   ├── test_manager.py
│   │   ├── test_developer.py
│   │   └── test_salesperson.py
│   │
│   └── test_patterns/             # Тесты паттернов
│       ├── __init__.py
│       ├── test_singleton.py
│       ├── test_factory.py
│       └── test_builder.py
```

Метрики тестирования

- **Покрытие кода:** 96%
- **Количество unit-тестов:** 54

- **Интеграционные тесты:** 12

Часть 1: Тестирование инкапсуляции и базового класса `Employee`

Пример теста

```
class TestEmployeeCreation:
    def test_employee_creation_valid_data(self):
        # Arrange
        emp = Employee(1, "Alice", "IT", 5000.0)

        # Assert
        assert emp.id == 1
        assert emp.name == "Alice"
        assert emp.department == "IT"
        assert emp.base_salary == 5000.0

    def test_employee_invalid_id_in_init_raises_error(self):
        # Assert
        with pytest.raises(ValueError):
            Employee(-1, "Alice", "IT", 5000.0)

        with pytest.raises(ValueError):
            Employee(0, "Alice", "IT", 5000.0)

    def test_employee_invalid_base_salary_in_init_raises_error(self):
        # Assert
        with pytest.raises(ValueError):
            Employee(1, "Alice", "IT", -100.0)


class TestEmployeeMethods:
    def test_employee_str_representation(self):
        emp = Employee(1, "Alice", "IT", 5000.0)

        result = str(emp)

        expected = "Сотрудник [id: 1, имя: Alice, отдел: IT, базовая зарплата: 5000.0]"
        assert result == expected

    def test_employee_calculate_salary_returns_base_salary(self):
        emp = Employee(1, "Alice", "IT", 5000.0)

        salary = emp.calculate_salary()

        assert salary == 5000.0
```

Результаты

- Написаны тесты для конструктора и методов класса `Employee`.

- Добавлены тесты для проверки валидации данных в сеттерах.
- Тесты покрывают все методы класса

Часть 2: Тестирование наследования и абстрактных классов

Тестирование абстрактного класса

```
class TestAbstractEmployee:
    def test_cannot_instantiate_abstract_employee(self):
        with pytest.raises(TypeError):
            AbstractEmployee(1, "John", "IT", 5000.0)

    def test_subclasses_are_subtypes_of_abstract_employee(self):
        assert issubclass(Employee, AbstractEmployee)
        assert issubclass(Manager, AbstractEmployee)
        assert issubclass(Developer, AbstractEmployee)
        assert issubclass(Salesperson, AbstractEmployee)
```

Тестирование класса Manager

```
class TestManager:
    def test_manager_salary_calculation(self):
        manager = Manager(1, "John", "Management", 5000.0, 1000.0)
        salary = manager.calculate_salary()
        assert salary == 6000.0

    def test_manager_bonus_validation(self):
        manager = Manager(1, "John", "Management", 5000.0, 1000.0)

        with pytest.raises(ValueError):
            manager.bonus = -1
        with pytest.raises(ValueError):
            manager.bonus = 0

    def test_manager_str_contains_bonus(self):
        manager = Manager(1, "John", "Management", 5000.0, 1000.0)
        text = str(manager)

        assert "бонус" in text.lower()
        assert "1000.0" in text
```

Тестирование класса Developer

```
class TestDeveloper:
    @pytest.mark.parametrize("level, coef", [
        ("junior", 1.0),
        ("middle", 1.5),
        ("senior", 2.0),
```

```
    ])
    def test_developer_salary_by_level(self, level, coef):
        base_salary = 5000.0
        dev = Developer(1, "Alice", "DEV", base_salary, ["Python"], level)

        assert dev.calculate_salary() == base_salary * coef

    def test_add_skill_appends_to_tech_stack(self):
        dev = Developer(1, "Alice", "DEV", 5000.0, ["Python"], "senior")

        dev.add_skill("SQL")

        assert "SQL" in dev.tech_stack
        assert dev.tech_stack == ["Python", "SQL"]

    def test_tech_stack_validation(self):
        dev = Developer(1, "Alice", "DEV", 5000.0, ["Python"], "senior")

        with pytest.raises(ValueError):
            dev.tech_stack = "not_a_list"  # не список

        with pytest.raises(ValueError):
            dev.tech_stack = ["Python", ""]  # пустая строка

    def test_developer_iterates_over_skills(self):
        dev = Developer(1, "Alice", "DEV", 5000.0, ["Python", "SQL"], "senior")

        skills = [s for s in dev]

        assert skills == ["Python", "SQL"]
```

Тестирование класса Salesperson

```
class TestSalesperson:
    def test_salesperson_salary_with_commission(self):
        sp = Salesperson(1, "Bob", "Sales", 3000.0, 0.2, 200000.0)

        salary = sp.calculate_salary()

        assert salary == 3000.0 + 0.2 * 200000.0

    def test_update_sales_increases_volume(self):
        sp = Salesperson(1, "Bob", "Sales", 3000.0, 0.1, 10000.0)

        sp.update_sales(5000.0)

        assert sp.sales_volume == 15000.0

    def test_commission_and_sales_validation(self):
        with pytest.raises(ValueError):
            Salesperson(1, "Bob", "Sales", 3000.0, -0.1, 10000.0)
```

```
        sp = Salesperson(1, "Bob", "Sales", 3000.0, 0.1, 10000.0)

        with pytest.raises(ValueError):
            sp.commission_rate = 0

        with pytest.raises(ValueError):
            sp.sales_volume = 0
```

Тестирование фабрики сотрудников

```
class TestEmployeeFactories:
    def test_manager_factory_creates_manager(self):
        mgr = ManagerFactory.create_employee(
            id=1,
            name="John",
            department="MAN",
            base_salary=5000.0,
            bonus=1000.0,
        )

        assert isinstance(mgr, Manager)
        assert mgr.calculate_salary() == 6000.0

    def test_developer_factory_creates_developer(self):
        dev = DeveloperFactory.create_employee(
            id=1,
            name="Alice",
            department="DEV",
            base_salary=5000.0,
            tech_stack=["Python"],
            seniority_level="middle",
        )

        assert isinstance(dev, Developer)
        assert dev.calculate_salary() == 5000.0 * 1.5

    def test_salesperson_factory_creates_salesperson(self):
        sp = SalespersonFactory.create_employee(
            id=1,
            name="Bob",
            department="SAL",
            base_salary=3000.0,
            commission_rate=0.2,
            sales_volume=100000.0,
        )

        assert isinstance(sp, Salesperson)
        assert sp.calculate_salary() == 3000.0 + 0.2 * 100000.0
```

Тестирование полиморфного поведения

```
class TestPolymorphicEmployees:
    def test_polymorphic_calculate_salary(self):
        employees = [
            Employee(1, "Base", "IT", 4000.0),
            Manager(2, "Manager", "MAN", 5000.0, 1000.0),
            Developer(3, "Dev", "DEV", 5000.0, ["Python"], "senior"),
            Salesperson(4, "Sales", "SAL", 3000.0, 0.1, 50000.0),
        ]
        salaries = [e.calculate_salary() for e in employees]

        assert salaries[0] == 4000.0
        assert salaries[1] == 6000.0
        assert salaries[2] == 10000.0
        assert salaries[3] == 3000.0 + 0.1 * 50000.0
```

Результаты

- Написаны тесты для основных классов наследников
- Добавлены тесты для фабрики и параметризованные тесты
- Полное покрытие тестами всей иерархии классов, включая крайние случаи и проверку полиморфного поведения

Часть 3: Тестирование полиморфизма и магических методов

Тестирование класса Department

```
class TestDepartmentBasic:
    def test_add_and_get_employees(self):
        dept = Department("IT")
        emp = Employee(1, "John", "IT", 5000.0)

        dept.add_employee(emp)

        assert dept.get_employees() == [emp]

    def test_add_duplicate_employee_raises(self):
        dept = Department("IT")
        emp = Employee(1, "John", "IT", 5000.0)
        dept.add_employee(emp)

        with pytest.raises(ValueError):
            dept.add_employee(emp)

    def test_remove_employee_by_id(self):
        dept = Department("IT")
        emp = Employee(1, "John", "IT", 5000.0)
        dept.add_employee(emp)

        dept.remove_employee(1)
```

```
        assert len(dept) == 0

    def test_remove_non_existing_employee_raises(self):
        dept = Department("IT")

        with pytest.raises(ValueError):
            dept.remove_employee(999)

    def test_find_employee_by_id(self):
        dept = Department("IT")
        emp = Employee(1, "John", "IT", 5000.0)
        dept.add_employee(emp)

        found = dept.find_employee_by_id(1)

        assert found is emp
        assert dept.find_employee_by_id(999) is None
```

Тестирование магических методов сотрудников

```
class TestEmployeeMagicMethods:
    def test_employee_equality_by_id(self):
        emp1 = Employee(1, "John", "IT", 5000.0)
        emp2 = Employee(1, "Jane", "HR", 4000.0)
        emp3 = Employee(2, "Bob", "IT", 5000.0)

        assert emp1 == emp2    # одинаковый id
        assert emp1 != emp3

    def test_employee_salary_comparison(self):
        emp1 = Employee(1, "John", "IT", 5000.0)
        emp2 = Employee(2, "Jane", "HR", 6000.0)

        assert emp1 < emp2
        assert emp2 > emp1

    def test_employee_addition_with_employee_and_number(self):
        emp1 = Employee(1, "John", "IT", 5000.0)
        emp2 = Employee(2, "Jane", "HR", 6000.0)

        assert emp1 + emp2 == 11000.0
        assert emp1 + 1000 == 6000.0
        assert 1000 + emp1 == 6000.0
```

Тестирование магических методов Department

```
class TestDepartmentMagicMethods:
    def test_len_getitem_contains_iter(self):
        dept = Department("IT")
```

```
        employees = [Employee(i, f"Emp{i}", "IT", 5000.0) for i in range(3)]

        for emp in employees:
            dept.add_employee(emp)

        assert len(dept) == 3
        assert dept[0] is employees[0]
        assert employees[1] in dept

        names = [e.name for e in dept]
        assert names == ["Emp0", "Emp1", "Emp2"]
```

Тестирование полиморфизма

```
class TestDepartmentPolymorphism:
    def test_calculate_total_salary(self):
        dept = Department("IT")
        emp = Employee(1, "Base", "IT", 4000.0)
        mgr = Manager(2, "Manager", "IT", 5000.0, 1000.0)
        dev = Developer(3, "Dev", "IT", 5000.0, ["Python"], "middle")

        dept.add_employee(emp)
        dept.add_employee(mgr)
        dept.add_employee(dev)

        total = dept.calculate_total_salary()
        expected = (
            emp.calculate_salary()
            + mgr.calculate_salary()
            + dev.calculate_salary()
        )

        assert total == expected
```

Тестирование сериализации

```
class TestSerialization:
    def test_employee_to_from_dict(self):
        emp = Employee(1, "John", "IT", 5000.0)

        data = emp.to_dict()
        new_emp = Employee.from_dict(data)

        assert new_emp.id == emp.id
        assert new_emp.name == emp.name
        assert new_emp.department == emp.department
        assert new_emp.base_salary == emp.base_salary
```

Тестирование сортировки

```
class TestSorting:
    def test_sorting_by_name_and_salary(self):
        employees = [
            Employee(3, "Charlie", "IT", 7000.0),
            Employee(1, "Alice", "HR", 5000.0),
            Employee(2, "Bob", "IT", 6000.0),
        ]

        sorted_by_name = sorted(employees, key=lambda e: e.name)
        assert [e.name for e in sorted_by_name] == ["Alice", "Bob", "Charlie"]

        sorted_by_salary = sorted(employees, key=lambda e: e.calculate_salary())
        assert sorted_by_salary[0].calculate_salary() == 5000.0
```

Результаты

- Написаны тесты для основных магических методов и полиморфного поведения
- Добавлены тесты для сериализации и итерации

Часть 4: Тестирование композиции, агрегации и сложных структур

Тестирование методов управления командой проекта

```
class TestProject:
    def test_project_team_management(self):
        project = Project(1, "AI Platform", "Разработка AI системы", "2024-12-31", "planning"
        dev = Developer(1, "John", "DEV", 5000.0, ["Python"], "senior")

        project.add_team_member(dev)
        assert len(project.get_team()) == 1
        assert project.get_team_size() == 1

        project.remove_team_member(1)
        assert len(project.get_team()) == 0

    def test_project_total_salary(self):
        project = Project(1, "AI Platform", "Разработка AI системы", "2024-12-31", "planning"
        manager = Manager(1, "Alice", "DEV", 7000.0, 2000.0)
        developer = Developer(2, "Bob", "DEV", 5000.0, ["Python"], "senior")

        project.add_team_member(manager)
        project.add_team_member(developer)

        total = project.calculate_total_salary()
        expected = manager.calculate_salary() + developer.calculate_salary()

        assert total == expected

    @pytest.mark.parametrize("invalid_status", ["invalid", "done", "in_progress"])
    def test_project_invalid_status_raises_error(self, invalid_status):
```

```
        with pytest.raises(InvalidStatusError):
            Project(1, "Test", "Test", "2024-12-31", invalid_status)
```

Тестирование класса Company

```
class TestCompanyDepartmentsProjects:
    def test_company_department_management(self):
        company = Company("TechCorp")
        dept = Department("Development")

        company.add_department(dept)
        assert len(company.get_departments()) == 1

        company.remove_department("Development")
        assert len(company.get_departments()) == 0

    def test_cannot_delete_department_with_employees(self):
        company = Company("TechCorp")
        dept = Department("Development")
        emp = Employee(1, "John", "DEV", 5000.0)

        dept.add_employee(emp)
        company.add_department(dept)

        with pytest.raises(ValueError):
            company.remove_department("Development")

    def test_company_find_employee(self):
        company = Company("TechCorp")
        dept = Department("Development")
        emp = Employee(1, "John", "DEV", 5000.0)

        dept.add_employee(emp)
        company.add_department(dept)

        found = company.find_employee_by_id(1)

        assert found is not None
        assert found.name == "John"

    def test_company_total_monthly_cost(self):
        company = Company("TechCorp")
        dept = Department("Development")
        emp1 = Employee(1, "John", "DEV", 5000.0)
        emp2 = Employee(2, "Jane", "DEV", 6000.0)

        dept.add_employee(emp1)
        dept.add_employee(emp2)
        company.add_department(dept)

        total = company.calculate_total_monthly_cost()
        assert total == emp1.calculate_salary() + emp2.calculate_salary()
```

Тестирование кастомных исключений

```
class TestCustomExceptions:
    def test_duplicate_project_id_raises_error(self):
        company = Company("TechCorp")
        p1 = Project(1, "P1", "Desc", "2024-12-31", "planning")
        p2 = Project(1, "P2", "Desc2", "2024-12-31", "planning")

        company.add_project(p1)

        with pytest.raises(DuplicateIdError):
            company.add_project(p2)
```

Результаты

- Написаны тесты для основных методов Project и Company
- Добавлены тесты для кастомных исключений

Часть 5: Тестирование паттернов проектирования

Тестрирование Singleton-паттерна

```
class TestSingletonDatabaseConnection:
    def test_singleton_instance_is_same(self):
        db1 = DatabaseConnection()
        db2 = DatabaseConnection()

        assert db1 is db2
        assert id(db1) == id(db2)

    def test_get_connection_returns_same_object(self):
        db = DatabaseConnection()
        conn1 = db.get_connection()
        conn2 = db.get_connection()

        assert conn1 is conn2
```

Тестрирование Factory Method-паттерна

```
class TestEmployeeFactories:
    def test_manager_factory_creates_manager(self):
        mgr = ManagerFactory.create_employee(
            id=1,
            name="John",
            department="MAN",
            base_salary=5000.0,
```

```
            bonus=1000.0,
        )

        assert isinstance(mgr, Manager)
        assert mgr.calculate_salary() == 6000.0

    def test_developer_factory_creates_developer(self):
        dev = DeveloperFactory.create_employee(
            id=1,
            name="Alice",
            department="DEV",
            base_salary=5000.0,
            tech_stack=["Python"],
            seniority_level="middle",
        )

        assert isinstance(dev, Developer)
        assert dev.calculate_salary() == 5000.0 * 1.5

    def test_salesperson_factory_creates_salesperson(self):
        sp = SalespersonFactory.create_employee(
            id=1,
            name="Bob",
            department="SAL",
            base_salary=3000.0,
            commission_rate=0.2,
            sales_volume=100000.0,
        )

        assert isinstance(sp, Salesperson)
        assert sp.calculate_salary() == 3000.0 + 0.2 * 100000.0
```

Тестрирование Abstract Factory-паттерна

```
class TestCompanyFactories:
    def test_tech_company_factory_creates_company_with_departments_and_projects(self):
        factory = TechCompanyFactory()

        company = factory.create_company("TechCorp")

        assert isinstance(company, Company)
        deps = company.get_departments()
        projs = company.get_projects()

        assert any(d.name == "Development" for d in deps)
        assert any(d.name == "QA" for d in deps)
        assert any(p.name == "AI Platform" for p in projs)
        assert any(p.name == "Web Portal" for p in projs)

    def test_sales_company_factory_creates_company_with_departments_and_projects(self):
        factory = SalesCompanyFactory()
```

```
        company = factory.create_company("SalesCorp")

        deps = company.get_departments()
        projs = company.get_projects()

        assert any(d.name == "Sales" for d in deps)
        assert any(d.name == "Marketing" for d in deps)
        assert any(p.name == "Vending" for p in projs)
        assert any("Маркетплейс" in p.name for p in projs)
```

Тестирование Builder-паттерна

```
class TestEmployeeBuilder:
    def test_build_developer_with_builder(self):
        dev = (
            EmployeeBuilder()
            .set_type("developer")
            .set_id(101)
            .set_name("John Doe")
            .set_department("DEV")
            .set_base_salary(5000.0)
            .set_tech_stack(["Python", "Java"])
            .set_seniority_level("senior")
            .build()
        )

        assert dev.id == 101
        assert dev.name == "John Doe"
        assert dev.calculate_salary() == 5000.0 * 2.0  # senior
        assert "Python" in dev.tech_stack
```

Результаты

- Написаны тесты для 3+ паттернов с базовой функциональностью

Заключение

Достигнутые результаты

1. Разработана полнофункциональная система учета сотрудников
2. Применены все принципы ООП: инкапсуляция, наследование, полиморфизм
3. Реализованы 4+ паттернов проектирования
4. Обеспечено высокое качество кода через тестирование
5. Создана расширяемая и поддерживаемая архитектура

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

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/