Министерство образования Республики Беларусь  
Учреждение образования  
«Брестский государственный технический университет»  
Кафедра интеллектуальных информационных технологий

Отчет

По дисциплине «Объектное моделирование интеллектуальных систем»

По лабораторной работе № 3

«Иерархии наследования, динамический полиморфизм в приложениях на языках С++, С##»

Выполнил:  
Студент 3 курса  
Группы ИИ-22  
Копанчук Е. Р.

Проверил:  
Монтик Н. С.

Брест 2023

**Цель работы:** изучить особенности организации и реализации множественного наследования классов. Изучить варианты организации иерархии классов с использованием отношений агрегации, композиции, использования. Изучить проектирование иерархий классов с использованием динамического полиморфизма.

**Ход работы:**

**Множественное наследование**

Анализировать предметную область и разработать классы для демонстрации всех видов наследования. Для этого:

- описать исходный класс. Состав минимальный, включая конструкторы, деструктор, перегруженный оператор. Включить в классы члены-данные, требующие динамического распределения памяти;

- для заданной предметной области разработать дополнительные классы для демонстрации всех видов наследования;

- описать классы на UML (язык - русский);

- описать диаграмму классов

(уровень ассоциаций) на UML (язык - русский) без раскрытия классов;

- уточнить диаграмму классов (типы отношений) на UML (язык - русский) без раскрытия классов;

- описать уточненную диаграмму классов UML с раскрытием классов;

- специфицировать, прокомментировать при необходимости данные и методы класса.

**Организация иерархий классов**

Для указанной предметной области:

– реализовать иерархии классов, используя наследование и используя агрегацию (состав минимальный, включая конструкторы, деструктор, перегруженный оператор):

- для каждого описать диаграмму UML (язык - русский);

- описать диаграмму UML (язык - английский);

- специфицировать данные и методы класса;

- разработать диаграмму классов (уровень ассоциаций);

- уточнить диаграмму классов (типы отношений). Программировать.

Реализовать приложение для обработки объектов предметной области MIX (целое и вещественное), использовав агрегацию и наследование для организации иерархии классов.

**Динамический полиморфизм**

1. Продемонстрировать для своей предметной области (без применения виртуальных функций) использование собственных указателей классов и указателей базовых классов для доступа к членам классов иерархии со стороны объектов.
2. Повторить предыдущий пункт с применением виртуальных функций.
3. Реализовать приложение с использованием динамического полиморфизма. Вариант реализации – консольный с управлением через простейшее меню.

**Предметная область: "Университет"**

**Множественное наследование**

Описание исходного класса "Университет":

class University {

private:

string name; // Название университета

int numStudents; // Количество студентов

Student\* students; // Указатель на массив объектов класса Student

public:

// Конструкторы

University() {

name = "";

numStudents = 0;

students = nullptr;}

University(string \_name, int \_numStudents) {

name = \_name;

numStudents = \_numStudents;

students = new Student[numStudents];}

// Деструктор

~University() {

delete[] students;}

// Перегруженный оператор присваивания

University& operator=(const University& other) {

if (this != &other) {

name = other.name;

numStudents = other.numStudents;

delete[] students;

students = new Student[numStudents];

for (int i = 0; i < numStudents; ++i) {

students[i] = other.students[i];}}

return \*this;}

};

Класс "Университет" содержит данные о названии университета, количестве студентов и массиве объектов класса "Студент". Требуется динамическое распределение памяти для хранения массива студентов.

Дополнительные классы для демонстрации всех видов наследования:

a) Класс "Студент":

class Student {

private:

string name; // Имя студента

int age; // Возраст студента

public:

// Конструкторы

Student() {

name = "";

age = 0;}

Student(string \_name, int \_age) {

name = \_name;

age = \_age;}

// Деструктор

~Student() {}

// Геттеры и сеттеры

string getName() const {

return name;}

void setName(string \_name) {

name = \_name;}

int getAge() const {

return age;}

void setAge(int \_age) {

age = \_age;}

};

б) Класс "Преподаватель":

class Teacher {

private:

string name; // Имя преподавателя

string specialization; // Специализация преподавателя

public:

// Конструкторы

Teacher() {

name = "";

specialization = "";}

Teacher(string \_name, string \_specialization) {

name = \_name;

specialization = \_specialization;}

// Деструктор

~Teacher() {}

// Геттеры и сеттеры

string getName() const {

return name;}

void setName(string \_name) {

name = \_name;}

string getSpecialization() const {

return specialization;}

void setSpecialization(string \_specialization) {

specialization = \_specialization;}

};

в) Класс "Курс":

class Course {

private:

string name; // Название курса

Teacher teacher; // Указатель на объект класса Teacher

public:

// Конструкторы

Course() {

name = "";

teacher = null;}

Course(string \_name, Teacher \_teacher) {

name = \_name;

teacher = \_teacher;}

// Деструктор

~Course() {}

// Геттеры и сеттеры

string getName() const {

return name;}

void setName(string \_name) {

name = \_name;}

Teacher\* getTeacher() {

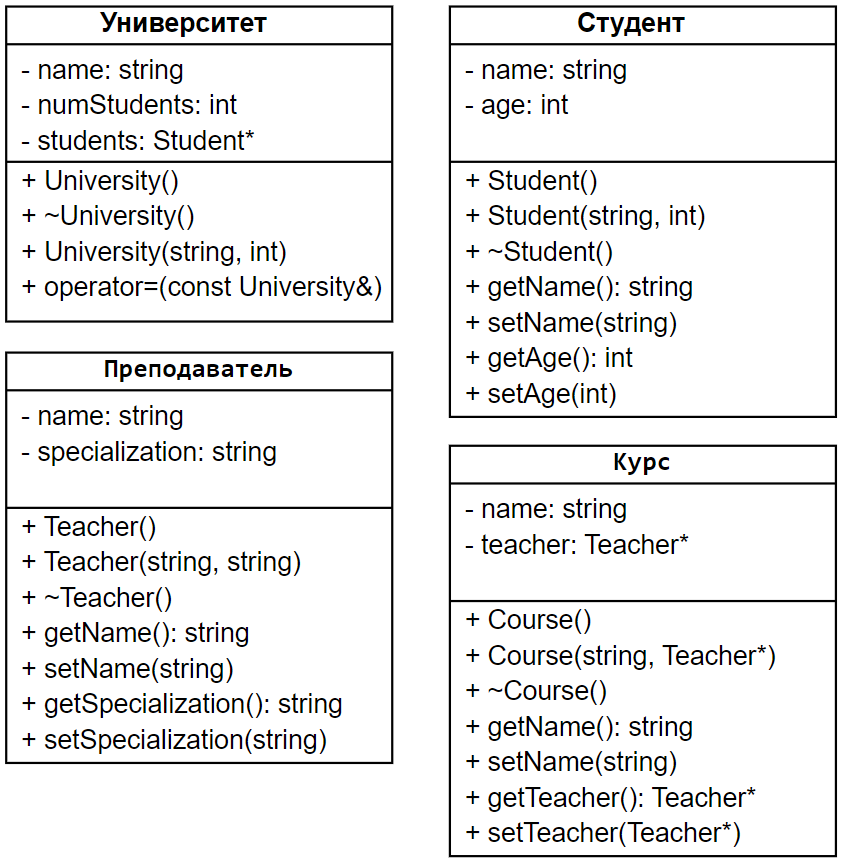
return teacher;}

void setTeacher(Teacher\* \_teacher) {

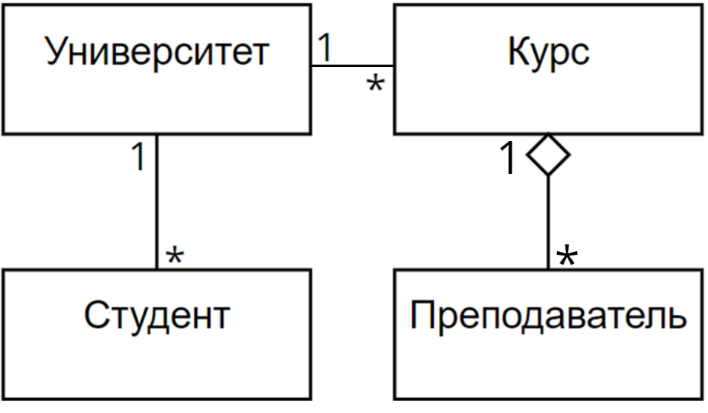
teacher = \_teacher;}

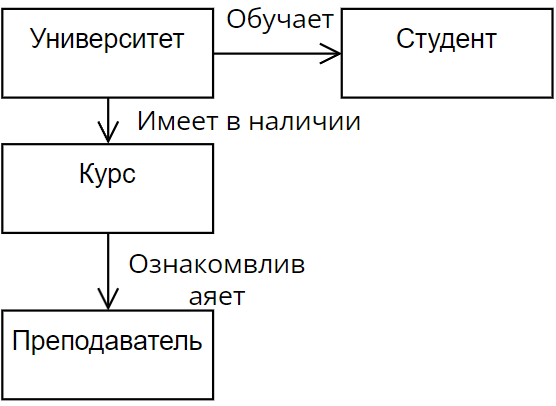
};

**Описание классов:**

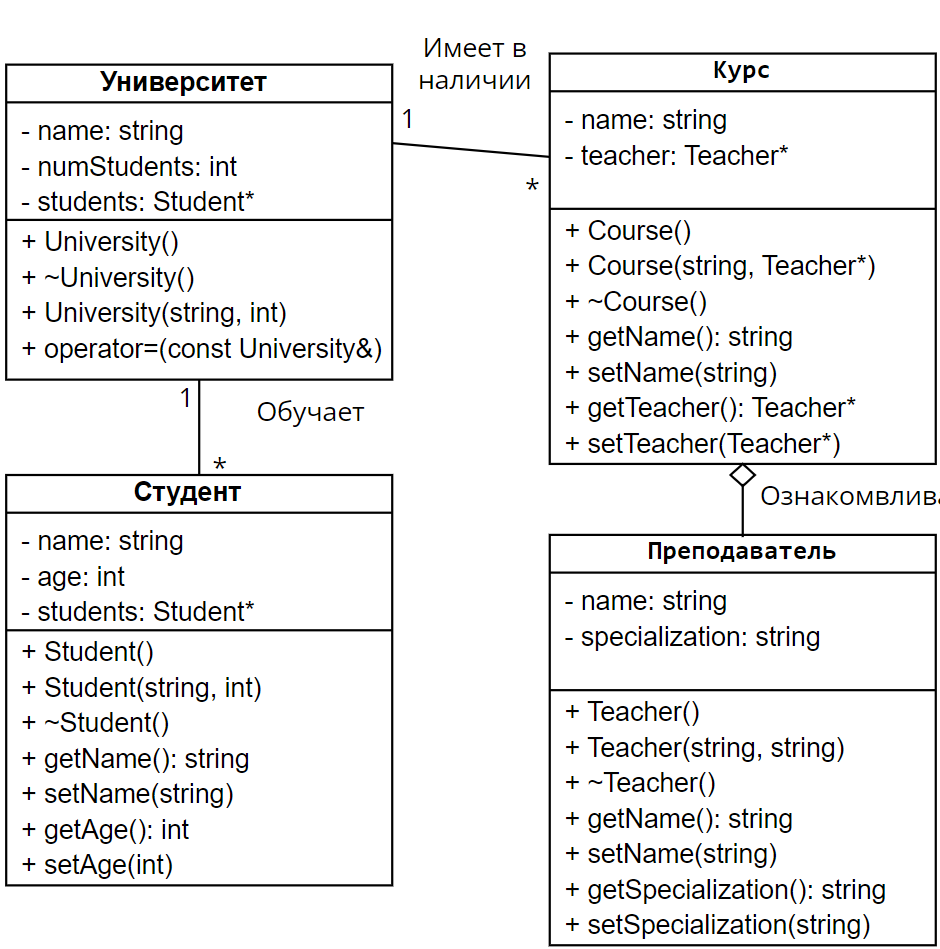


**Диаграмма классов (уровень ассоциаций и типы отношений):**





**Уточненная диаграмма классов:**



**Организация иерархий классов**

Так как структура всех UML будет идентична предыдущему пункту, нет нужды повторно их повторно рисовать.

**Код программы:**

#include <iostream>

#include <string>

using namespace std;

class Person {

protected:

string name;

int age;

public:

Person() : name(""), age(0) {}

Person(string \_name, int \_age) : name(\_name), age(\_age) {}

string getName() const {return name;}

void setName(string \_name) {name = \_name;}

int getAge() const {return age;}

void setAge(int \_age) {age = \_age;}};

class Student : public Person {private: double grade;

public:

Student() : Person(), grade(0.0) {}

Student(string \_name, int \_age, double \_grade) : Person(\_name, \_age), grade(\_grade) {}

double getGrade() const {return grade;}

void setGrade(double \_grade) {grade = \_grade;}};

class Teacher : public Person {

private:

string specialization;

public:

Teacher() : Person(), specialization("") {}

Teacher(string \_name, string \_specialization) : Person(\_name, 0), specialization(\_specialization) {}

string getSpecialization() const {return specialization;}

void setSpecialization(string \_specialization) {specialization = \_specialization;}};

class Course {

private:

string name;

Teacher\* teacher;

Student\* students;

int numStudents;

public:

Course() : name(""), teacher(nullptr), students(nullptr), numStudents(0) {}

Course(string \_name, Teacher\* \_teacher, Student\* \_students, int \_numStudents)

: name(\_name), teacher(\_teacher), students(\_students), numStudents(\_numStudents) {}

string getName() const {return name;}

void setName(string \_name) {name = \_name;}

Teacher\* getTeacher() {return teacher;}

void setTeacher(Teacher\* \_teacher) {teacher = \_teacher;}

Student\* getStudents() {return students;}

void setStudents(Student\* \_students, int \_numStudents) {students = \_students; numStudents = \_numStudents;}};

class University {

private:

string name;

int numStudents;

Student\* students;

public:

University() : name(""), numStudents(0), students(nullptr) {}

University(string \_name, int \_numStudents, Student\* \_students)

: name(\_name), numStudents(\_numStudents), students(\_students) {}

~University() {

delete[] students;}

University& operator=(const University& other) {

if (this != &other) {

name = other.name;

numStudents = other.numStudents;

delete[] students;

students = new Student[numStudents];

for (int i = 0; i < numStudents; ++i) {

students[i] = other.students[i];}}

return \*this;}};

int main() {

Teacher\* teacher = new Teacher("Иванов", "Математика");

Student students[2] = {Student("Иванов", 20, 4.5), Student("Петров", 19, 4.2)};

Course\* course = new Course("Алгебра", teacher, students, 2);

University university("МГУ", 2, students);

cout<<course->getName<<" "<<course->getStudents<<" "<<course->getTeacher<<endl;

cout<<students[0].getAge<<" "<<students[0].getGrade<<students[0].getName<<endl;

cout<<students[1].getAge<<" "<<students[1].getGrade<<students[1].getName<<endl;

delete course;

delete teacher;

return 0;

}

**Динамический полиморфизм**

1. Демонстрация использования собственных указателей классов и указателей базовых классов для доступа к членам классов иерархии со стороны объектов (без применения виртуальных функций):

**Код программы:**

#include <iostream>

using namespace std;

// Базовый класс

class BaseClass {

public:

int baseVariable;

void baseMethod() {

cout << "Вызван метод базового класса" << endl;}};

// Производный класс

class DerivedClass : public BaseClass {

public:

int derivedVariable;

void derivedMethod() {

cout << "Вызван метод производного класса" << endl;}};

int main() {

// Создание объекта производного класса

DerivedClass derivedObj;

derivedObj.baseVariable = 42;

derivedObj.derivedVariable = 10;

// Использование указателей базового класса для доступа к объекту производного класса

BaseClass\* basePtr = &derivedObj;

basePtr->baseVariable = 21;

cout << "Значение baseVariable: " << derivedObj.baseVariable << endl;

cout << "Значение derivedVariable: " << derivedObj.derivedVariable << endl;

// Использование указателя производного класса для доступа к объекту производного класса

DerivedClass\* derivedPtr = &derivedObj;

derivedPtr->baseVariable = 33;

derivedPtr->derivedVariable = 99;

cout << "Значение baseVariable: " << derivedObj.baseVariable << endl;

cout << "Значение derivedVariable: " << derivedObj.derivedVariable << endl;

return 0;

}

2. Демонстрация использования виртуальных функций:

**Код программы:**

#include <iostream>

using namespace std;

// Базовый класс

class BaseClass {

public:

int baseVariable;

virtual void method() {

cout << "Вызвана виртуальная функция базового класса" << endl;}};

// Производный класс

class DerivedClass : public BaseClass {

public:

int derivedVariable;

void method() override {

cout << "Вызвана виртуальная функция производного класса" << endl;}};

int main() {

// Создание объекта производного класса

DerivedClass derivedObj;

derivedObj.baseVariable = 42;

derivedObj.derivedVariable = 10;

// Использование указателей базового класса для вызова виртуальной функции

BaseClass\* basePtr = &derivedObj;

basePtr->method();

return 0;

}

3. Реализовать приложение с использованием динамического полиморфизма через консольное меню:

**Код программы:**

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() = 0;};

class Circle : public Shape {

public:

void draw() override {

cout << "Рисуем круг" << “◯” <<endl;}};

class Square : public Shape {

public:

void draw() override {

cout << "Рисуем квадрат" << “☐” endl;}};

int main() {

Shape\* shape;

int option;

do {

cout << "Выберите фигуру для отрисовки:" << endl;

cout << "1. Круг" << endl;

cout << "2. Квадрат" << endl;

cout << "0. Выход" << endl;

cout << "Введите номер: ";

cin >> option;

switch (option) {

case 1:

shape = new Circle();

shape->draw();

delete shape;

break;

case 2:

shape = new Square();

shape->draw();

delete shape;

break;

case 0:

cout << "Выход из программы." << endl;

break;

default:

cout << "Ошибка: некорректный выбор." << endl;

break;}}

while (option != 0);

return 0;}

**Вывод:** изучили особенности организации и реализации множественного наследования классов. Изучили варианты организации иерархии классов с использованием отношений агрегации, композиции, использования. Изучили проектирование иерархий классов с использованием динамического полиморфизма.