ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

ОТЧЁТ

по курсовой работе по дисциплине
«Объектно-ориентированное программирование»
на тему
«Программа с использованием объектно-ориентированных технологий вариант №2 “калькулятор”»

Выполнил студент _____
Стрельников Александр Максимович

Группы _____
ИС-241

Работу принял _____ ассистент кафедры ПМиК С.М. Анатольевна
подпись

Защищена _____ Оценка _____

Новосибирск – 2023

СОДЕРЖАНИЕ

Постановка Задачи.

Написать программу с использованием объектно-ориентированных технологий. Количество созданных классов – не менее трёх. Какие объекты должна описывать иерархия классов, выбирается по таблице согласно своему варианту (вариант определяется по последней цифре зачетной книжки).

Постановка задачи, Содержимое классов – Ваше творческое решение. В таблице к каждой теме приводится пример возможных полей данных и действий. Таким образом, результатом работы будет: иерархия классов и программа с каким-либо примером работы с объектами этих классов. Язык и Среда разработки: C++ (Dev-C++, Visual Studio и др.)

Использовать простые алгоритмы, позволяющие понять применяемую технологию.

Теория.

1. Полиморфизм - – это принцип объектно-ориентированного программирования, который описывает возможность объектов различных классов использовать одинаковые интерфейсы или методы, но давать различные реализации этих методов.

Полиморфизм позволяет обрабатывать объекты производных классов как объекты базового класса, что делает код более гибким и удобным для использования. Принцип полиморфизма включает в себя переопределение методов (полиморфизм через наследование) и полиморфизм через интерфейсы.

Полиморфизм через наследование позволяет производным классам переопределить методы базового класса с собственной реализацией, при этом код, который использует базовый класс, может вызывать эти методы, не зависимо от фактического типа объекта. Это позволяет программисту работать с объектами различных классов, используя общие интерфейсы или абстрактные классы.

Полиморфизм через интерфейсы предполагает, что разные классы могут реализовать один и тот же интерфейс, но давать различные реализации. Это позволяет использовать объекты разных классов через общий интерфейс, обеспечивая единые способы взаимодействия с ними.

Пример в коде:

1. Абстрактный класс «Operation»:

```
// Абстрактный класс для операций
class Operation {
public:
    // Чисто виртуальная функция для выполнения операции
    virtual double calculate(double operand1, double operand2) const = 0;
    // Виртуальный деструктор для правильного удаления объектов производных классов
    virtual ~Operation() = default;
};
```

Здесь ***“calculate”*** - это виртуальная функция, которая объявлена в базовом классе ***“Operation”*** и сделана чисто виртуальной с использованием = 0. Это означает, что каждый производный класс должен предоставить свою реализацию этой функции.

2. Производные классы(например, “Addition”,”Substraction”):

```
// Класс для сложения
class Addition : public Operation {
public:
```

```
    // Переопределение виртуальной функции
    double calculate(double operand1, double operand2) const override {
        return operand1 + operand2;
    }
};
```

Каждый из этих классов наследует от ***“Operation”*** и предоставляет свою собственную реализацию функции calculate. Это позволяет объектам этих классов быть использованными через указатель или ссылку на базовый класс.

3. Использование виртуальных функций в “TextCalculator”:

```
// Класс для текстового калькулятора
class TextCalculator {
private:
    Operation* operation; // Текущая операция

public:
    // Конструктор, принимающий указатель на операцию
    TextCalculator(Operation* op) : operation(op) {}
    // Выполнение операции через текущую операцию
    double performOperation(double operand1, double operand2) const {
        return operation->calculate(operand1, operand2);
    }
};
```

Здесь ***“operation”*** - это указатель на базовый класс ***“Operation”***. Когда вы вызываете calculate через ***“operation”***, будет вызвана конкретная реализация соответствующего производного класса, что и является проявлением полиморфизма.

2. ***Абстракция*** - это принцип объектно-ориентированного программирования, который предоставляет возможность создания упрощенных, концептуальных моделей объектов и их свойств, игнорируя лишние детали реализации.

В контексте ООП, абстракция позволяет определить общие характеристики и поведение объектов, выделяя их отличительные черты и роли в системе. Она позволяет разработчику сконцентрироваться на существенных аспектах объектов и игнорировать малозначительные детали.

Абстракция часто реализуется через абстрактные классы или интерфейсы. Абстрактный класс определяет общие свойства и методы для группы связанных классов, но не предоставляет конкретную реализацию для всех методов, оставляя их на уровне производных классов. Интерфейс определяет набор методов, которые должен реализовать класс, но не предписывает, как эти методы должны быть реализованы.

Пример в коде:

Использование абстрактного класса “Operation”:

```
// Абстрактный класс для операций
class Operation {
public:
    // Чисто виртуальная функция для выполнения операции
    virtual double calculate(double operand1, double operand2) const = 0;
    // Виртуальный деструктор для правильного удаления объектов производных классов
    virtual ~Operation() = default;
};
```

Здесь ***“Operation”*** - это абстрактный класс, который предоставляет общий интерфейс для различных операций. Функция calculate является абстрактной (чисто виртуальной), что означает, что каждый конкретный подкласс (например, ***“Addition”***, ***“Subtraction”***) обязан предоставить свою собственную реализацию этой функции.

3. **Инкапсуляция** - это принцип объектно-ориентированного программирования, который объединяет данные (поля) и методы, оперирующие этими данными, внутри класса, и скрывает их от прямого доступа извне.

Основная идея инкапсуляции заключается в том, что объекты должны иметь четко определенный интерфейс, через который можно выполнять операции с объектами, в то время как внутренняя структура и детали реализации остаются скрытыми.

Принцип инкапсуляции помогает обеспечить контролируемый доступ к данным и методам класса, предотвращая модификацию или неправильное использование данных извне.

Для этого используются модификаторы доступа, такие как публичный (**public**), приватный (**private**), защищенный (**protected**), которые определяют, какие члены класса могут быть доступны извне, какие могут быть доступны только внутри класса или его подклассов.

Важными понятиями, связанными с инкапсуляцией, являются сокрытие информации и согласованность интерфейса.

Пример в коде:

Инкапсуляция присутствует в классе ***"TextCalculator"***:

```
// Класс для текстового калькулятора
class TextCalculator {
private:
    Operation* operation; // Текущая операция

public:
    // Конструктор, принимающий указатель на операцию
    TextCalculator(Operation* op) : operation(op) {}
    // Выполнение операции через текущую операцию
    double performOperation(double operand1, double operand2) const {
        return operation->calculate(operand1, operand2);
    }
};
```

В этом классе переменная ***"operation"*** объявлена как закрытый (**private**) член класса. Это означает, что эта переменная не доступна извне класса, и её состояние может быть изменено и использовано только внутри самого класса ***"TextCalculator"***.

4. **Наследование** - это принцип объектно-ориентированного программирования, который позволяет создавать новые классы на основе существующих классов.

В рамках наследования, существующий класс, называемый родительским или базовым классом, передает свои свойства (поля и методы) новому классу, который называется дочерним или производным классом.

Дочерний класс наследует все свойства родительского класса и может расширять их, добавлять новые свойства и методы, или переопределять унаследованные методы для своих нужд. Это позволяет создавать иерархию классов, группируя их по общим характеристикам и поведению.

Принцип наследования позволяет повторно использовать код, минимизировать дублирование и упростить проектирование иерархии классов. Он также способствует созданию более абстрактной и гибкой архитектуры программы, позволяя работать с объектами различных классов, производных от одного базового класса, с помощью общего интерфейса.

Пример в коде:

В коде наследование проявляется в том, что классы ***"Addition"***, ***"Subtraction"***, ***"Multiplication"***, ***"Division***" и ***"Power"*** являются производными (или подклассами) от абстрактного класса ***"Operation"***. Это выражается ключевым словом (**public**) после двоеточия при объявлении этих классов.

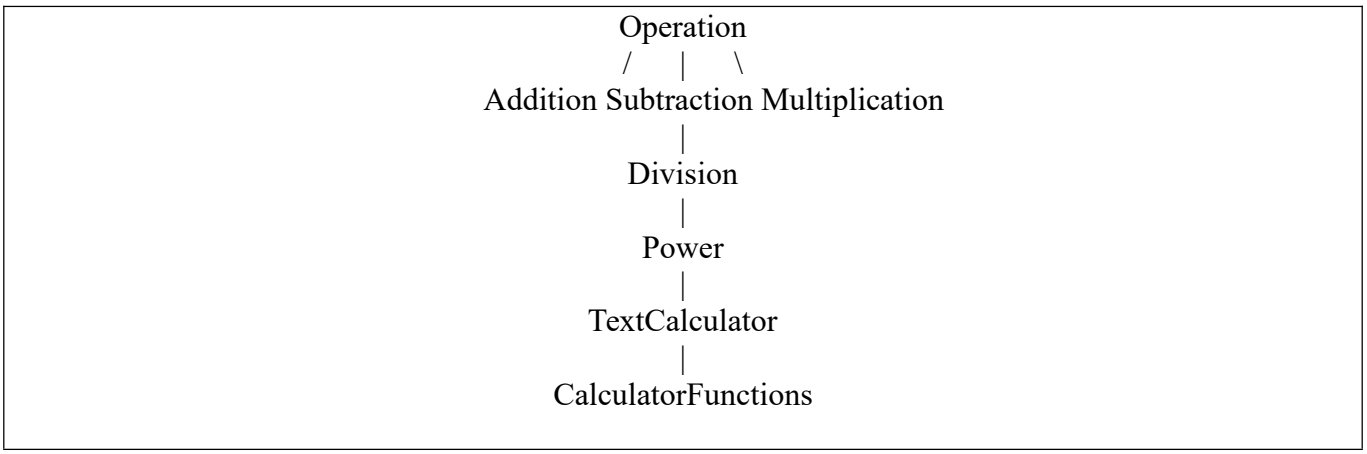
```
// Абстрактный класс для операций
class Operation {
public:
    // Чисто виртуальная функция для выполнения операции
    virtual double calculate(double operand1, double operand2) const = 0;
    // Виртуальный деструктор для правильного удаления объектов производных классов
    virtual ~Operation() = default;
};

// Класс для сложения
class Addition : public Operation {
public:
    // Переопределение виртуальной функции
    double calculate(double operand1, double operand2) const override {
        return operand1 + operand2;
    }
};
// Класс для вычитания
class Subtraction : public Operation {
public:
    double calculate(double operand1, double operand2) const override {
```

```
        return operand1 - operand2;
    }
};
//… аналогично для других классов
```

Здесь классы ***“Addition”***, ***“Subtraction”***, и другие наследуют виртуальные функции от ***“Operation”***, что является проявлением наследования в объектно-ориентированном программировании.

3. Иерархия классов.



Объяснение иерархии:

1. ***"Operation"***: Это абстрактный класс для операций. У него есть чисто виртуальная функция calculate, которую должны реализовывать производные классы. Он также имеет виртуальный деструктор.

2. ***"Addition"***, ***"Subtraction***", ***"Multiplication"***, ***"Division"***, ***"Power"***: Эти классы являются производными от класса "Operation" и реализуют конкретные операции (сложение, вычитание, умножение, деление, возведение в степень). Они переопределяют чисто виртуальную функцию calculate.

3. ***"TextCalculator"***: Этот класс содержит указатель на объект типа ***"Operation"*** и предоставляет метод "performOperation", который использует текущую операцию для выполнения математической операции.

4. ***"CalculatorFunctions"***: Это класс с набором статических методов, представляющих различные математические функции (сложение, вычитание и так далее). В вашем коде этот класс не использует наследование.

Таким образом, иерархия объектов включает абстрактный класс ***"Operation"*** и его производные классы, которые реализуют конкретные операции. Класс ***"TextCalculator"*** использует объект типа ***"Operation"*** для выполнения математических операций, и ***"CalculatorFunctions"*** предоставляет функции для вывода результатов операций.

4. **Объяснение алгоритма**

1. Запрос ввода данных от пользователя:
    * В начале программы запрашиваются числовые значения ***“operand1”*** и ***“operand2”*** с помощью функции ***“getDoubleInput”***.
2. Основной цикл программы:
    * Программа находится в цикле, который выполняется до тех пор, пока переменная ***“exitProgram”*** не станет равной ***“true”***.
3. Создание объектов операций:
    * Создаются объекты различных операций: ***“Addition”***, ***“Subtraction”***, ***“Multiplication”***, ***“Division”***, ***“Power”***.
4. Создание объектов ***“TextCalculator”***:
    * Создаются объекты ***“TextCalculator”*** с различными операциями. Каждый объект ***“TextCalculator”*** содержит указатель на объект операции.
5. Выбор операции:
    * Пользователю предлагается выбрать операцию, введя соответствующий номер.
    * Если введен “0”, программа завершает свою работу.
    * Если введен “9”, пользователю предоставляется возможность ввести новые значения для ***“operand1”*** и ***“operand2”***.
6. Выполнение выбранной операции:
    * Создается объект ***“CalculatorFunctions”***.
    * Вызывается функция ***“Print_Result”***, передаются выбор пользователя, ***“operand1”***, ***“operand2”*** и объект ***“CalculatorFunctions”***.
    * В ***“Print_Result”*** в зависимости от выбора пользователя вызывается соответствующий метод объекта ***“CalculatorFunctions”***, который выводит результат операции.
7. Вывод результата:
    * Выводится результат выбранной операции.
8. Повторение цикла:
    * Программа возвращается в начало цикла и повторяет процесс до тех пор, пока пользователь не решит завершить работу.

В этом алгоритме основное внимание уделено использованию полиморфизма для выполнения различных операций с помощью общего интерфейса ***“Operation”***. Ввод данных, выбор операции и вывод результата обобщены, что делает программу удобной для расширения новыми операциями.

5. Результат работы.

```
PS D:\VS.main()\CourceWorkOOP> g++ -Wall -o res calculate.cpp
PS D:\VS.main()\CourceWorkOOP> ./res.exe
Введите значение operand1: 10
Введите значение operand2: 20
Выберите операцию:
 1 - Сложение
 2 - Вычитание
 3 - Умножение
 4 - Деление
 5 - Возведение в степень (operand1 в степени operand'a2)
 0 - Завершить программу
 9 - Ввод новых значений
Ваш выбор: 2

Вычитание: -10

Выберите операцию:
 1 - Сложение
 2 - Вычитание
 3 - Умножение
 4 - Деление
 5 - Возведение в степень (operand1 в степени operand'a2)
 0 - Завершить программу
 9 - Ввод новых значений
Ваш выбор: 0

PS D:\VS.main()\CourceWorkOOP>
```

6. Приложение.

```cpp
#include <iostream>
#include <cmath>

using namespace std;
// Абстрактный класс для операций
class Operation {
public:
    // Чисто виртуальная функция для выполнения операции
    virtual double calculate(double operand1, double operand2) const = 0;
    // Виртуальный деструктор для правильного удаления объектов производных классов
    virtual ~Operation() = default;
};
// Класс для сложения
class Addition : public Operation {
public:
    // Переопределение виртуальной функции
    double calculate(double operand1, double operand2) const override {
        return operand1 + operand2;
    }
};
// Класс для вычитания
class Subtraction : public Operation {
public:
    double calculate(double operand1, double operand2) const override {
        return operand1 - operand2;
    }
};
// Класс для умножения
class Multiplication : public Operation {
public:
    double calculate(double operand1, double operand2) const override {
        return operand1 * operand2;
    }
};
// Класс для деления
class Division : public Operation {
public:
    double calculate(double operand1, double operand2) const override {
        // Проверка деления на ноль
        if (operand2 != 0) {
            return operand1 / operand2;
```

```
        } else {
            // Вывод сообщения об ошибке
            cerr << "Ошибка: деление на ноль!\n";
            return 0.0; // Можно выбрать другое значение по умолчанию
        }
    }
};
// Класс для возведения в степень
class Power : public Operation {
public:
    double calculate(double operand1, double operand2) const override {
        return std::pow(operand1, operand2);
    }
};
// Класс для текстового калькулятора
class TextCalculator {
private:
    Operation* operation; // Текущая операция
public:
    // Конструктор, принимающий указатель на операцию
    TextCalculator(Operation* op) : operation(op) {}
    // Выполнение операции через текущую операцию
    double performOperation(double operand1, double operand2) const {
        return operation->calculate(operand1, operand2);
    }
};
// Класс с функциями-членами
class CalculatorFunctions {
public:
    // Функция сложения
    static void add(double operand1, double operand2) {
        cout << "Сложение: " << operand1 + operand2 << std::endl;
        }
    // Функция вычитания
    static void subtract(double operand1, double operand2) {
        cout << "Вычитание: " << operand1 - operand2 << std::endl;
    }
    // Функция умножения
    static void multiply(double operand1, double operand2) {
        cout << "Умножение: " << operand1 * operand2 << std::endl;
    }
    // Функция деления
    static void divide(double operand1, double operand2) {
        if (operand2 != 0) {
            cout << "Деление: " << operand1 / operand2 << std::endl;
```

```
        } else {
            cerr << "Ошибка: деление на ноль!\n";
        }
    }
    // Функция возведения в степень
    static void power(double operand1, double operand2) {
        cout << "Возведение в степень: " << std::pow(operand1, operand2) << std::endl;
    }
};
// Прототип функции
double getDoubleInput(const std::string& prompt);
void Print_Result(int choice,double operand1, double operand2,  CalculatorFunctions&
calculatorFunctions);
int main() {
    bool exitProgram = false;
    // Запрос числовых значений от пользователя
        double operand1 = getDoubleInput("Введите значение operand1: ");
        double operand2 = getDoubleInput("Введите значение operand2: ");
    do {

        // Пример использования калькулятора
        Addition addition;
        Subtraction subtraction;
        Multiplication multiplication;
        Division division;
        Power power;
        // Создание объектов TextCalculator с разными операциями
        TextCalculator calculatorAddition(&addition);
        TextCalculator calculatorSubtraction(&subtraction);
        TextCalculator calculatorMultiplication(&multiplication);
        TextCalculator calculatorDivision(&division);
        TextCalculator calculatorPower(&power);
        // Выбор функции-члена на основе ввода
        int choice;
        std::cout << "Выберите операцию:\n"
                     " 1 - Сложение\n"
                     " 2 - Вычитание\n"
                     " 3 - Умножение\n"
                     " 4 - Деление\n"
                     " 5 - Возведение в степень (operand1 в степени operand'a2)\n"
                     " 0 - Завершить программу\n"
                     " 9 - Ввод новых значений\n"
                     "Ваш выбор: ";
        std::cin >> choice;
        std::cout << endl;
```

```
        if (choice == 0) {
            // Пользователь выбрал завершение программы
            exitProgram = true;
        }
        else if(choice == 9)
        {
            // Повторный запрос числовых значений от пользователя
            operand1 = getDoubleInput("Введите значение operand1: ");
            operand2 = getDoubleInput("Введите значение operand2: ");
        }
        else {
            // Создание объекта CalculatorFunctions
            CalculatorFunctions calculatorFunctions;
            // Вывод результата
            Print_Result(choice, operand1, operand2, calculatorFunctions);
        }
    } while (!exitProgram);
    return 0;
}
// Функция для вывода выбранного типа математического действия
void Print_Result(int choice,double operand1, double operand2,  CalculatorFunctions&
calculatorFunctions)
{
    switch (choice) {
        case 1:
            calculatorFunctions.add(operand1, operand2);
            cout << "\n";
            break;
        case 2:
            calculatorFunctions.subtract(operand1, operand2);
            cout << "\n";
            break;
        case 3:
            calculatorFunctions.multiply(operand1, operand2);
            cout << "\n";
            break;
        case 4:
            calculatorFunctions.divide(operand1, operand2);
            cout << "\n";
            break;
        case 5:
            calculatorFunctions.power(operand1, operand2);
            cout << "\n";
            break;
        default:
```

```
            cerr << "Некорректный ввод.\n";
            break;
    }
}
// Определение функции getDoubleInput
double getDoubleInput(const std::string& prompt) {
    double value;
    while (true) {
        std::cout << prompt;
        if (std::cin >> value) {
            // Проверка, что ввод - число
            break;
        } else {
            cin.clear(); // Сброс ошибки ввода
            cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка
буфера
            cout << "Ошибка! Введите корректное число.\n";
        }
    }
    return value;
}
```