Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ
И РАДИОЭЛЕКТРОНИКИ (ТУСУР)
Кафедра компьютерных систем в управлении и проектировании (КСУП)

ИСПОЛЬЗОВАНИЕ И ПРИЁМЫ РАЗРАБОТКИ КЛАССОВ

Отчёт по лабораторной работе №4 по дисциплине
«Объектно-ориентированное программирование»

Студент группы 588-1
_____ / Чан Хыу Тхай
«_» _____2022 г.

Руководитель старший научный
сотрудник, доцент каф. КСУП
_____ / Горяинов А.Е.
«___» _____ 2022 г.

Томск 2022

1. Цель работы

Целью данной лабораторной работы является использование и разработка классов, а также приобретение практических навыков решения актуальных задач объектно-ориентированного программирования.

2. Теоретические основы

Класс – это просто независимая подпрограмма, в которой есть свой набор переменных и функций (обязательно изучи урок про функции). В основной программе мы можем создать экземпляр класса (он же называется объект) и пользоваться теми инструментами, которые имеются в классе. Использование классов позволяет:

- Разделить сложную программу на отдельные независимые части
- Создавать удобные библиотеки
- Использовать свои “наработки” в другом проекте, не переписывая один и тот же код
- Облегчить и упростить программу, если в ней используются повторяющиеся конструкции и алгоритмы

Классы очень похожи на структуры (читай урок по структурам), как по объявлению, так и по использованию, но класс является гораздо более мощной единицей языка благодаря механизмам наследования и прочим ООП-штукам.

Класс объявляется при помощи ключевого слова class и содержит внутри себя члены класса – переменные и функции:

```
class Имя_класса {

  член1;

  член2;

};
```

Важное отличие от структуры: содержимое класса делится на области: публичные и приватные. Они определяются при помощи ключевых слов public и private, область действует до начала следующей области или до закрывающей фигурной скобки класса:

- `public` – члены класса в этой области доступны для взаимодействия из основной программы (скетча), в которой будет создан объект. Например `write()` у серво.
- `private` – члены класса в этой области доступны только **внутри класса**, то есть из программы к ним обратиться нельзя.

```
class Имя_класса {

  public:

  // список членов, доступных в программе

  private:

  // список членов для использования внутри класса

};
```

Ранее было рассмотрено агрегирование – взаимодействие двух классов, где объекты одного класса явля-ются частью состояния другого класса. Связь агрегирования предполагает, что один объект предметной об-ласти является частью другого, например, двигатель как часть автомобиля, или песня как часть музыкаль-ного альбома. Однако в реальном мире объекты не обязательно связаны агрегированием.

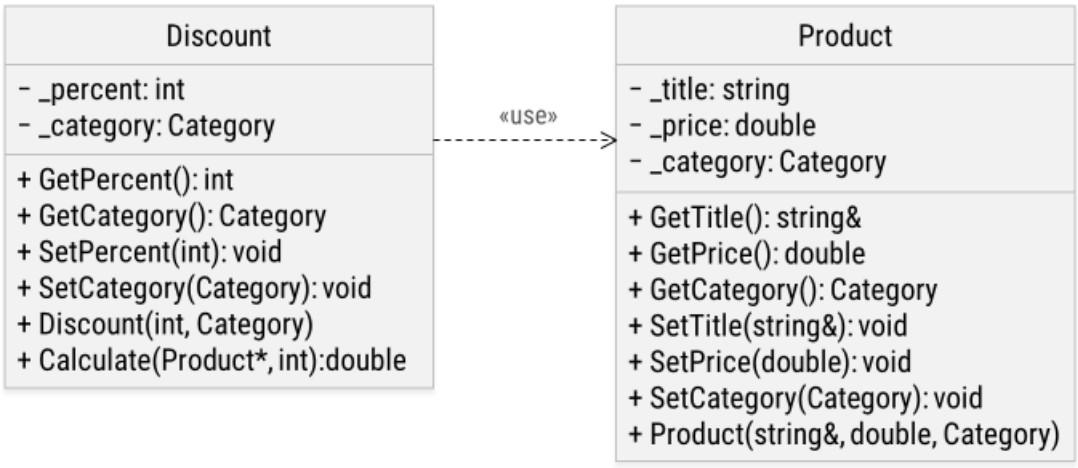
Например, покупатель и продавец в магазине взаимодействуют для совершения покупки, но ни в коем случае нельзя сказать, что покупатель является частью продавца или продавец – частью покупателя. Фактически, покупатель – это объект, который существует в среде самостоятельно, однако время от времени, для совер-шения действия (которое является частью его поведения) ему нужен объект продавца. Покупатель обраща-ется к продавцу, передавая деньги и запрос на товары, а продавец принимает деньги и возвращает запро-шенные товары. После завершения действия покупатель никак не связан с объектом продавца. Такой вид взаимодействия называется использованием.

Использование – взаимодействие двух объектов, при котором один объект обращается к полям или методам другого объекта во время выполнения собственных методов. Рассмотрим на более конкретном примере.

Допустим у нас есть класс Товара в магазине, у которого есть название, цена и категория (перечисление):

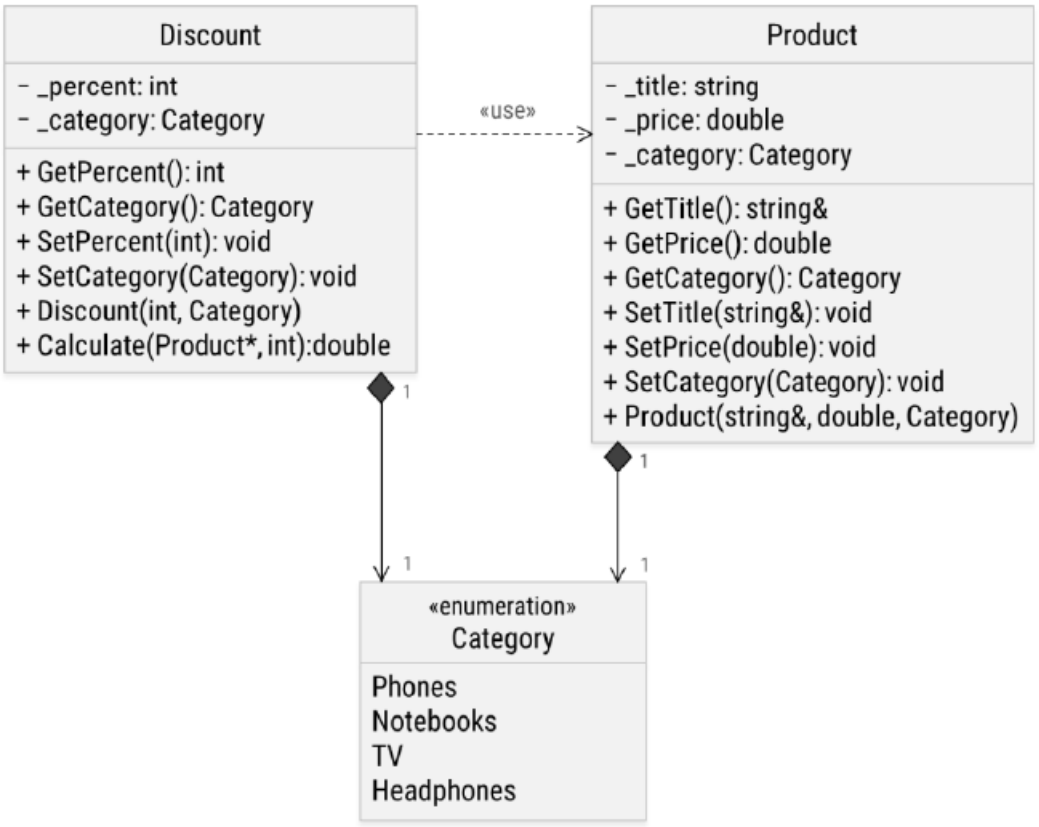
```
enum Category
{
    Phones,
    Notebooks,
    TV,
    Headphones
};

class Product
{
    string _title;
    double _price;
    Category _category;

public:
    void SetTitle(string& title);
    void SetPrice(double price);
    void SetCategory(Category category);

    string& GetTitle();
    double GetPrice();
    Category GetCategory();

    Product(string& title, double price, Category category);
};
```

На UML-диаграммах классов использованием обозначается направленной пунктирной стрелкой со стереоти-пом (подписью) «use». Стрелка направляется от использующего класса к используемому:



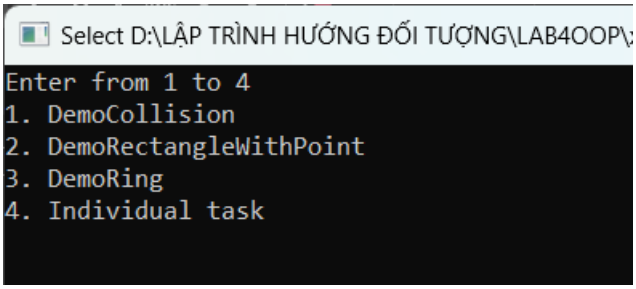
Для связи использования кратность не указывается. Отдельно стоит пояснить, что если два класса связаны агрегированием, то, логично, они связаны и использованием (например, потому что тип данных используе-мого класса явно указан в сеттерах и геттерах). Однако, если между двумя классами существует более одной связи, то на диаграммах указывается более сильная связь. То есть из двух связей агрегирования или исполь-зования на диаграмме рисуют только агрегирование.

Полная диаграмма примера будет выглядеть следующим образом:



3. Ход работы

Ниже представлен интерфейс лабораторной работы:



В первой задаче необходимо исправить разработанные классы, если класс агрегирует массив значений, массив и количество элементов массива задаются двумя отдельными сеттерами, переделайте отдельные сеттеры массива в один сеттер для указать зависимые поля. Это должны быть классы Album и Band. (рис. 1), (рис. 2).

```
class Band
{
    string Name;
    string History;
    int countMembers;
    string* Members;
    int countAlbums;
    Album* Albums;
    Song searchSongResult;
    Album searchAlbumResult;
    int allSongsCount = 0;
    Song* AllSongsStorage;
    int AllRockCount = 0, AllMetallCount = 0, AllHipHopCount = 0, AllRapCount = 0, AllJazzCount = 0, AllClassicCount = 0;
    Song* RockStorage;
    Song* MetallStorage;
    Song* HipHopStorage;
    Song* RapStorage;
    Song* JazzStorage;
    Song* ClassicStorage;

public:
    void ReadBandFromConsole();
    void WriteBandFromConsole();
    Song* FindSong(string songTitle);
    Album* FindAlbum(Song* song);
    void GetAllSongs();
    void GetAllGenreSongs(Genre findingGenre);

    void DemoBand();
};
```

Рис. 1 – Создать класс Band

```
class Album
{
public:
    string Name;
    int Year;
    Song* Songs;
    int countSong = 0;
    int RockCount = 0, MetallCount = 0, HipHopCount = 0, RapCount = 0, JazzCount = 0, ClassicCount = 0;
    void ReadAlbumFromConsole();
    void WriteAlbumFromConsole();
};
```

Рис. 2 – Создать класс Album

Следующей задачей необходимо создать статический класс CollisionManager - класс, выполняющий проверку о пересечении/столк-новении геометрических фигур. Классы по проверке столкновений применяются в геометрических САПР, 2D- и 3D-редакторах и компьютерных играх. Класс должен реализовать два метода:

- bool IsCollision(Rectangle&, Rectangle&) – метод принимает два экземпляра прямоугольника и проверяет не пересекаются ли они. Прямоугольники считаются пересекающимися если разница их координат по X (по модулю) меньше суммы половин их ширин и разница их координат по Y (по модулю) меньше суммы половин их высот. Если прямоугольники пересекаются, метод возвращает true, иначе возвращает false. (рис. 3).

```
bool CollisionManager::IsCollision(RectangleClass firstRectangle, RectangleClass secondRectangle)
{
    double dX = abs(firstRectangle.GetPointX() - secondRectangle.GetPointX());
    double dY = abs(firstRectangle.GetPointY() - secondRectangle.GetPointY());
    double dWidth = abs(firstRectangle.GetWidth() - secondRectangle.GetWidth()) / 2;
    double dLenght = abs(firstRectangle.GetLength() - secondRectangle.GetLength()) / 2;

    if (dX < dWidth && dY < dLenght)
    {
        return true;
    }
    return false;
}

bool CollisionManager::IsCollision(Ring firstRing, Ring secondRing)
{
    double dX = abs(firstRing.GetPointX() - secondRing.GetPointX());
    double dY = abs(firstRing.GetPointY() - secondRing.GetPointY());
    double C = sqrt(pow(dX, 2) + pow(dY, 2));
    if (C < (firstRing.GetOuterRadius() + secondRing.GetOuterRadius()))
    {
        return true;
    }
    return false;
}
```

Рис. 3 – Создать класс CollisionManager

Следующей задачей необходимо создать сервисный (не статический) класс GeometricProgram, в который поместите все демонстра-ционные функции, связанные с геометрическими фигурами: DemoRing(), DemoCollision() и другие ра-нее разработанные функции. Класс GeometricProgram является чистой выдумкой пользовательского интерфейса для лабораторной работы №4.

Реализация пользовательского интерфейса в виде клас-сов также часто встречается на практике (рис. 4), (рис. 5).

```
void GeometricProgram::DemoCollision()
{
    RectangleClass rectangle1, rectangle2;
    rectangle1.ReadRectanglesFromConsole();
    rectangle2.ReadRectanglesFromConsole();

    switch (CollisionManager::IsCollision(rectangle1, rectangle2))
    {
    case true: cout << "Rectangles intersect" << endl; break;

    case false: cout << "Rectangles do not intersect" << endl; break;
    }

    Point point1, point2;
    point1 = Point(4, 4);
    point2 = Point(5, 5);

    Ring ring1, ring2;
    ring1 = Ring(point1, 10, 7);
    ring2 = Ring(point2, 5, 4);

    switch (CollisionManager::IsCollision(ring1, ring2))
    {
    case true: cout << "Rings intersect" << endl; break;

    case false: cout << "Rings do not intersect" << endl; break;
    }
}
```

Рис. 4 – Создать функцию DemoCollision

```
void GeometricProgram::DemoRing()
{
    cout << "Number of rings before constructor call: " << Ring::GetAllRingsCount() << endl;

    Ring* ring = new Ring(Point(25.0, 25.0), 10.0, 5.0);
    cout << "Number of rings after constructor call: " << Ring::GetAllRingsCount() << endl;

    delete ring;
    cout << "Number of rings after destructor call: " << Ring::GetAllRingsCount() << endl;

    const int countRings = 3;
    Ring rings[countRings];

    for (int i = 0; i < countRings; i++)
    {
        double innerRadius, outerRadius;
        cout << "Enter inner radius " << i + 1 << "-th rings: ";
        CheckInput::CheckInputDouble(&innerRadius);
        cout << "\nEnter outer radius  " << i + 1 << "-th rings:";
        CheckInput::CheckInputDouble(&outerRadius);

        Point point;
        cout << "\nEnter Center  " << i + 1 << "-th rings: " << endl;
        double X, Y;
        cout << "Enter X: ";
        CheckInput::CheckInputDouble(&X);
        cout << "\nEnter Y: ";
        CheckInput::CheckInputDouble(&Y);

        point = Point(X, Y);

        rings[i] = Ring(point, outerRadius, innerRadius);
        cout << "Created: " << Ring::GetAllRingsCount() << endl;
        rings[i].WriteRingFromConsole();
```

Рис. 5 – Создать функцию DemoRing

На рисунке ниже показана функция DemoCollision (рис. 6).

```
Rectangles do not intersect
Left: 2
Left: 2
Left: 1
Left: 0
Rings intersect
Left: -1
Left: -2
Enter from 1 to 4
```

Рис. 6 – Результат демонстрационной функции

На рисунке ниже показана функция DemoRectangleWithPoint (рис. 7).

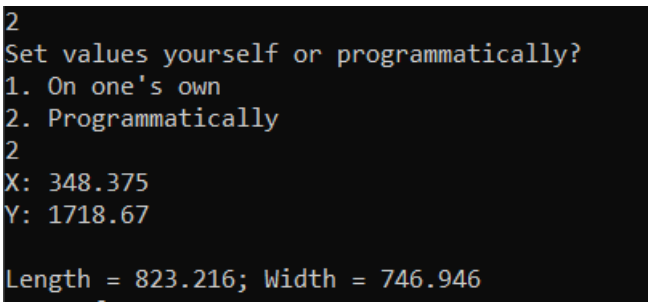


Рис. 7 – Результат демонстрационной функции

На рисунке ниже показана функция DemoRectangleWithPoint (рис. 8).

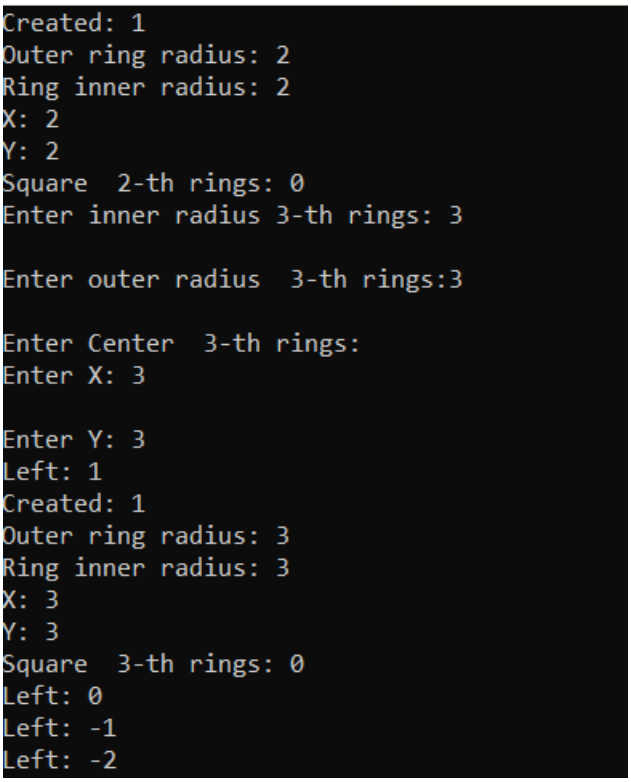


Рис. 8 – Результат демонстрационной функции

4. Вывод

В ходе работы в лаборатории были выполнены все задачи, разработана природа «Использование и приёмы разработки классов» в объектно-ориентированном программировании и набор функций для работы с ней.