Лабораторная работа №4
Использование и приёмы разработки классов

Использование

Ранее было рассмотрено агрегирование – взаимодействие двух классов, где объекты одного класса являются частью состояния другого класса. Связь агрегирования предполагает, что один объект предметной области является частью другого, например, двигатель как часть автомобиля, или песня как часть музыкального 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 совершения действия (которое является частью его поведения) ему нужен объект продавца. Покупатель обращается к продавцу, передавая деньги и запрос на товары, а продавец принимает деньги и возвращает запрошенные 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);
};
```

Предположим, что заказчик попросил вас разработать скидки на товары. Скидка действует во всём магазине для любых покупателей и предоставляется на определенную категорию товаров и определенным процентом. Скидка должна примениться после того, как пробиты все товары на кассе непосредственно перед оплатой – она должна проверить все товары в корзине и посчитать размер предоставляемой скидки в случае, если в ней есть товары соответствующей категории.

Категория товаров и процент будут полями нового класса. Но размер скидки – величина рассчитываемая и зависит от списка покупок конкретного покупателя. То есть и величина предоставленной скидки, и список товаров не будут храниться в виде полей внутри скидки. Это не логично и с точки зрения реальной предметной области, и с точки зрения программы – после проведения покупки данные о товарах и предоставленном размере скидки уже не нужны. В этом случае, добавим в класс скидки метод, который будет принимать на вход массив покупаемых товаров, и возвращать размер скидки для этих товаров. Для этого внутри метода необходимо проверить все товары массива и, если их категория совпадает с категорией скидки, то просуммировать размер скидки на каждый товар (стоимость товара, умноженная на процент скидки):

```
class Discount
{
    int _percent; // от 0 до 100
    Category _category;

public:
    void SetPercent(int percent);
    void SetCategory(Category category);

    int GetPercent();
    Category GetCategory();

    Discount(int percent, Category category);

    // Метод расчета размера скидки
    double Calculate(Product* products, int productsCount);
};

// Реализация метода расчета скидки
double Discount::Calculate(Product* products, int productsCount)
{
    // Извне в метод приходит массив товаров,
    // Процент скидки и категория хранятся в самом объекте скидки

    double discountSize = 0.0;
    for (int i = 0; i < productsCount; i++)
    {
            // Если категория товара совпадает с категорией скидки,
            // то предоставляем на него скидку
            if (products[i].GetCategory() == this->GetCategory())
            {
                    discountSize += products[i].GetPrice() * this->GetPercent();
            }
    }

    return discountSize;
}
```

В примере видно, что в метод приходит массив товаров, используемый во время расчета размера скидки, но после завершения метода, скидка ничего не знает о товарах – переменная products перестает существовать, а значение указателя на массив не сохранено ни в одном из полей объекта.

Использование может быть в одном из трёх вариантов:

- метод класса принимает объекты другого класса в качестве аргументов (как в примере выше);
- метод класса возвращает объекты другого класса (например, если бы метод в классе скидки возвращал новый массив товаров, на который была предоставлена скидка);
- метод класса создает внутри себя объекты другого класса, но объекты уничтожаются при завершении метода.

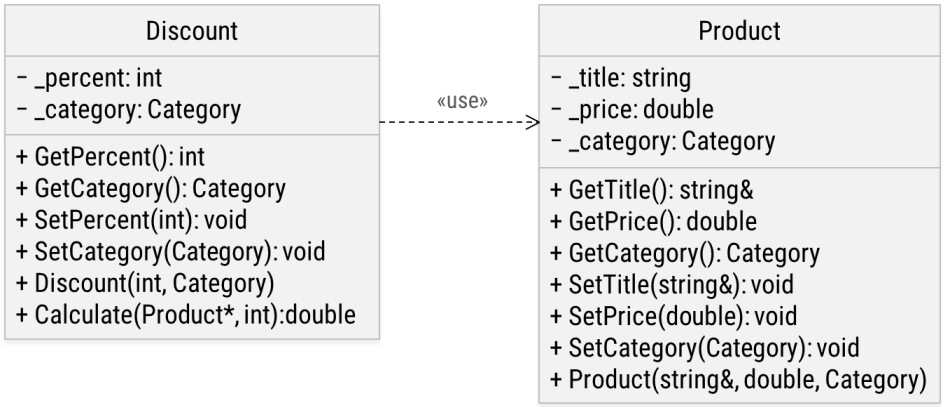
Простой признак использования – если в реализации методов одного класса встречается название другого класса, значит, эти классы связаны использованием.

Простой признак агрегирования – если в списке полей одного класса встречается название другого класса, эти классы связаны агрегированием.

Отметим, что использование является более слабой связью объектов по сравнению с агрегацией или композицией. Более слабой связью она является потому, что при использовании объекты взаимодействуют только во время работы метода, а при агрегировании один объект является постоянной частью другого объекта. Само понятие "слабая связь – сильная связь" часто упоминаются при анализе и проектировании архитектуры

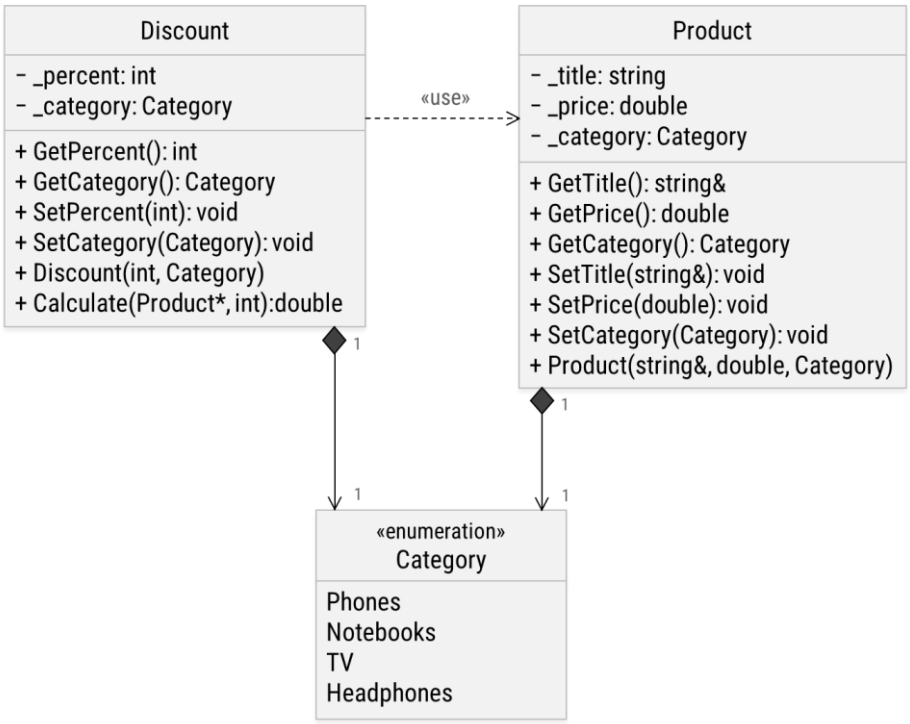
программы. Слабая связь всегда предпочтительнее сильной связи, так как подразумевает, что при изменении одного класса с меньшей вероятностью придется вносить изменения в другой класс. Если классы связаны слабо, их легко модифицировать независимо друг от друга. Если классы связаны сильно, то модификация одного класса приведет к модификации и другого. Это кажется мало критичным на примере двух классов товара и скидки, но программы состоят из сотен и тысяч классов, взаимодействующих друг с другом. Один класс может быть связан с десятком других. Таким образом, при появлении новых требований от заказчика, даже самых незначительных, при сильной связности классов в вашей программе изменения в одном классе приводят к дополнительным изменениям в десятке других, что в свою очередь может привести к изменениям в следующих классах и т.д. пока вся программа не будет переписана. Как проектировать программы для обеспечения наименьшей связи между классами – не входит в курс объектно-ориентированного программирования, и должно изучаться позже. На данном этапе отметим, что использование является самой слабой связью, которая может быть между классами.

На UML-диаграммах классов использованием обозначается направленной пунктирной стрелкой со стереотипом (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 используемого 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 использования на диаграмме рисуют только агрегирование.

Полная диаграмма примера будет выглядеть следующим образом:



Закрытые методы класса

Рассмотрим следующий пример:

```
class Student
{
    string _surname;
    string _name;
    string _group;

public:
    void SetSurname(string surname);
    void SetName(string name);
    void SetGroup(string group);

    string GetSurname();
    string GetName();
    string GetGroup();

    Student(string surname, string name, string group);
};

void Student::SetSurname(string surname)
{
    for (int i = 0; i < surname.length(); i++)
    {
        bool isLetter =
            (surname[i] > 'A' && surname[i] < 'Z')
            || (surname[i] > 'a' && surname[i] < 'z');
        // если символ в строке не является буквой, кидаем исключение
        if (!isLetter)
        {
            throw exception("Строка не должна содержать символов, кроме букв");
        }
    }

    _surname = surname;
}

void Student::SetName(string name)
{
    for (int i = 0; i < name.length(); i++)
    {
        bool isLetter =
            (name[i] > 'A' && name[i] < 'Z')
            || (name[i] > 'a' && name[i] < 'z');
        // если символ в строке не является буквой, кидаем исключение
        if (!isLetter)
        {
            throw exception("Строка не должна содержать символов, кроме букв");
        }
    }

    _name = name;
}
```

Представлен класс студента Student, которые хранит данные фамилии, имени и группы студента. Класс предоставляет геттеры и сеттеры для полей, а также конструктор класса. Под объявлением класса приведены реализации двух сеттеров SetSurname() и SetName(). Реализация других методов опущена, так как она не важна для данного примера.

Если посмотреть на реализацию сеттеров, то можно заметить, что их реализация полностью дублируется – отличаются только имена входной переменной и конечного поля. Дублирование кода может привести к проблемам поддержки. Например, в случае если заказчик попросит добавить поддержку двойных фамилий и имен, написанных через пробел или дефис, придется менять код в обоих методах.

Проблему дублирования можно решить, создав отдельную функцию проверки фамилии:

```
void CheckString(string value)
{
    for (int i = 0; i < value.length(); i++)
    {
            bool isLetter =
                    (value[i] > 'A' && value[i] < 'Z')
                    || (value[i] > 'a' && value[i] < 'z');
            // если символ в строке не является буквой, кидаем исключение
            if (!isLetter)
            {
                    throw exception("Строка не должна содержать символов, кроме букв");
            }
    }
}
```

Тогда реализация сеттеров заметно сократится:

```
void Student::SetSurname(string surname)
{
    CheckString(surname);
    _surname = surname;
}

void Student::SetName(string name)
{
    CheckString(name);
    _name = name;
}
```

Но где разместить функцию CheckString()? Функция нужна только для избавления от дублирования внутри конкретного класса. Она не должна находиться вне класса, и не должна вызываться со стороны клиентского кода. Как мы помним, чем меньше открытых методов есть у класса, тем проще использовать его объекты. Чем больше методов у класса, тем больше времени разработчик тратит на изучение сторонних классов вместо написания новой функциональности. Следовательно, функции, подобные CheckString(), должны быть закрыты для внешнего использования. Для этого функцию необходимо сделать закрытым (private) методом класса. Объявление класса в таком случае будет выглядеть следующим образом:

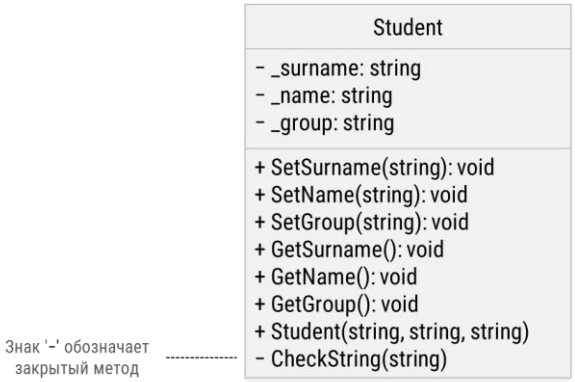
```
class Student
{
private:
    string _surname;
    string _name;
    string _group;

    // Метод под модификаторов private доступен только внутри класса
    void CheckString(string value);
```

```
public:
    void SetSurname(string surname);
    void SetName(string name);
    void SetGroup(string group);

    string GetSurname();
    string GetName();
    string GetGroup();

    Student(string surname, string name, string group);
};
```

Вызов такого метода невозможен со стороны клиентского кода:

```
void main()
{
    Student* student = new Student("Смирнов", "Юрий", "517");
    student->CheckString("Wrong1!%^&");
    // Ошибка компиляции - нельзя вызвать закрытый метод
}
```

На UML-диаграммах классов закрытые методы отображаются в области методов класса, но со знаком – в начале строки:



Помимо избавления от дублирования кода, закрытые методы класса также применяются для разбиения больших методов (более 50 строк кода) на более мелкие, и повышения читаемости кода. Важно, что закрытыми делают только те методы, которые должны использоваться только внутри данного класса. Если методы потенциально могут использоваться в разных классах, для этого применяется другой приём, который будет показан в последующих разделах.

Правила оформления закрытых методов не отличаются от правил именования открытых:

1) Именование от глагола, отвечая на вопрос "Что сделать?".
2) Именование в стиле Паскаль без нижнего подчеркивания.

Зависимые поля

Создавая класс, разработчик должен определить поля, описывающую данную сущность предметной области. Логично, что поля тесно связаны между собой по смыслу. Но в отдельных случаях, поля могут быть не просто связаны, а зависимы друг от друга – когда значение одного поля не может рассматриваться отдельно от другого поля.

Самый простой пример – это поля для хранения массива в классе. Для хранения массива необходимо завести два поля: в одном будет храниться указатель на выделенную под массив память, а во второй – количество элементов. Например, рассмотренный ранее класс группы Group с массивом студентов:

```
class Group
{
    string _number;
    int _entranceYear;
    Student* _students;
    int _studentsCount;

public:
    string GetNumber();
    int GetEntranceYear();
    Student* GetStudents();
    int GetStudentsCount();

    void SetNumber(string& number);
    void SetEntranceYear(int entranceYear);
    void SetStudents(Student* students);
    void SetStudentsCount(int studentsCount);

    Group(string& number, int entranceYear, Student* students, int studentsCount);
};
```

В этом классе есть два поля, который необходимы для работы с одним массивом. Эти поля зависимы. Каждое из этих полей имеет свой сеттер. Однако, два отдельных сеттера дают возможность вызова сеттеров отдельно друг от друга. Например, другой разработчик, используя класс Group, может поместить в поле _students указатель на новый массив через сеттер SetStudents(), но не обновить значение _studentsCount. А это в свою очередь приведет к ошибке выполнения, когда в другом участке кода произойдет обращение за пределы массива.

Так как эти поля зависимы друг от друга, они должны устанавливаться вместе, другими словами, через один сеттер:

```
void SetStudents(Student* students, int studentsCount)
{
    if (studentsCount < 0)
    {
            throw exception("Количество студентов в массиве не может быть отрицательным");
    }
    this->_students = students;
    this->_studentsCount = studentsCount;
}
```

Внутри такого сеттера переносятся проверки из двух предыдущих, а сами раздельные сеттеры удаляются. Объявление класса будет иметь следующий вид:

```
class Group
{
    string _number;
    int _entranceYear;
```

```
    Student* _students;
    int _studentsCount;

public:
    string GetNumber();
    int GetEntranceYear();
    Student* GetStudents();
    int GetStudentsCount();

    void SetNumber(string& number);
    void SetEntranceYear(int entranceYear);
    void SetStudents(Student* students, int studentsCount)

    Group(string& number, int entranceYear, Student* students, int studentsCount);
};
```

Теперь, если сторонний разработчик в клиентском коде захочет задать массив в объект класса Group, ему **придется** передать и указатель на массив, и количество элементов массива. Это может спасти клиентский код от нескольких часов отладки.

Обратите внимание, что геттеры зависимых полей объединять не нужно. Во-первых, средства Си++ не позволяют возвращать более одного значения из функции, а передача значений через ссылки или указатели в аргументы геттера не удобна в использовании. Во-вторых, во многих задачах требуется, например, знать только количество элементов в массиве. Вызывать геттер с двумя возвращаемыми значениями из функции, чтобы использовать только одно из них – нерационально. Более того, использование списков вместо обычных массивов решает проблему с передачей и обновлением актуального количества элементов.

Более сложный пример зависимых полей — это пример класса Треугольник, хранящего значения всех трёх сторон:

```
class Triangle
{
    double _a;
    double _b;
    double _c;

public:
    double GetA();
    double GetB();
    double GetC();

    void SetA(double a);
    void SetB(double b);
    void SetC(double c);
    Triangle(double a, double b, double c);
};
```

Реализация любого из сеттеров будет следующая:

```
void Triangle::SetA(double a)
{
    if (a < 0)
    {
           throw exception("Сторона треугольника не может быть отрицательной");
    }
    _a = a;
}
```

Однако такая реализация имеет "подводный камень" – она позволяет создавать "несуществующие" треугольники. Обязательным условием треугольника является необходимость, чтобы длина любой стороны была меньше суммы двух других сторон. Например, нельзя нарисовать треугольник со сторонами (10, 3, 3). Изменение длины любой стороны треугольника должно сверяться со двумя другими сторонами. Представленная выше реализация отсеивает только отрицательные значения, но создание треугольника со сторонами (10, 3, 3) никак не ограничено.

Разработчику может прийти идея добавить в сеттеры дополнительные проверки на суммы длин сторон:

```
void Triangle::SetA(double a)
{
    if (a < 0)
    {
           throw exception("Сторона треугольника не может быть отрицательной");
    }

    // Мы должны проверить все три комбинации сумм сторон
    if (a > (this->GetB() + this->GetC())
           || this->GetB() > (a + this->GetC())
           || this->GetC() > (a + this->GetB()))
    {
           throw exception("Сумма двух сторон нового треугольника меньше одной из
сторон");
    }

    _a = a;
}
```

Но эта реализация приведет к ошибке при первом же вызове конструктора. Разберем подробнее, почему это плохая реализация. При инициализации полей конструктор обязан использовать сеттеры, чтобы гарантировать присвоение значений полей строго через проверки. Для треугольника это конструктор будет реализован следующим образом:

```
Triangle::Triangle(double a, double b, double c)
{
    SetA(a);
    SetB(b);
    SetC(c);
}
```

При вызове конструктора, до того, как вызовется его первая строка (SetA(a);), произойдет выделение памяти под поля объекта: `_a`, `_b`, `_c`. В зависимости от настроек компилятора, значения полей будут равны нулю или случайных числам (зависящим от того, какие данные хранились ранее в выделенном под их хранение участке памяти). При выполнении первой строки SetA(a) произойдет проверка значения a на неотрицательность, а затем проверка на суммы длин сторон с текущими значениями. Но на момент вызова сеттера SetA(), значения в полях _b и _c еще не заданы и равны нулю (или случайным числам, в том числе и отрицательным). Следовательно, сумма _b и _c также будет меньше любого положительного числа в переменной a (или непредсказуема в случае случайных чисел). Перестановка порядка сеттеров ничего не поменяет – создание треугольника будет приводить к выбросу исключения.

В таких случаях вместо трёх отдельных сеттеров приходится создавать сеттеры для трёх сторон, которые задают все три стороны одновременно:

```
void SetSides(double a, double b, double c);
```

Однако такие сеттеры неудобны, когда нужно поменять только одну стороны. Вызов такого сеттера превращается в сложную составную конструкцию:

```
triangle.SetSides(a, triangle.GetB(), triangle.GetC());
```

На практике, при разработке САПР, 2D и 3D-редакторов, предпочитают представление фигур в виде точек координат. Треугольник, как примитивный полигон, может быть описан тремя произвольными точками. Преимущество хранения треугольника в виде точек в том, что точки никак не зависят друг от друга, и треугольник в плоскости можно провести через три любые произвольные точки.

Как показано выше – зависимые поля могут создать проблемы при использовании классов, а одним из способов борьбы с зависимыми полями, это представление объекта альтернативным набором полей. Однако зависимость полей может проявляться и иными способами.

Например, при создании пользовательского интерфейса используются элементы управления, такие как текстовые поля, кнопки, чекбоксы и т.д. Как правило, элементы управления имеют прямоугольную форму либо вписаны в прямоугольник, а также имеют координаты. Также важно знать, что элементы управления всегда располагаются под прямыми углами – крайне редко можно встретить элементы интерфейса не под прямым углом. Сами элементы управления в исходном коде также описываются классами с полями. Для хранения данных о расположении элементов в окне можно сделать одно из двух представлений:

```
class ControlUI
{
    Point _leftTop;
    Point _rightTop;
    Point _leftBottom;
    Point _rightBottom;

public:
    // сеттеры и геттеры
};
```

```
class ControlUI
{
    Point _location;
    int _width;
    int _height;

public:
    // сеттеры и геттеры
};
```

где класс Point хранит две целочисленные координаты X и Y.

По примеру треугольника может показаться, что представление геометрических фигур лучше в виде набора отдельных точек, как в примере слева: четыре точки описывают углы прямоугольника, в который вписан пользовательский элемент управления. Однако большей независимостью полей будет обладать вариант справа, описывающий прямоугольник одной точкой положения `_location` (как правило, верхний левый угол) и два целочисленных значения ширины и высоты элемента `_width` и `_height` соответственно.

Рассмотрим две простые операции, которые часто выполняются над пользовательскими элементами управления – это перемещение элемента с сохранением его размеров и это изменение размеров при сохранении расположения.

Чтобы переместить элемент управления с первой реализацией, необходимо поменять значения во всех четырех точках. Например, при перемещении элемента на 10 пикселей правее, необходимо прибавить 10 ко всем значениям X во всех четырех точках. В противном случае, изменятся размеры элементы.

Чтобы переместить элемент управления со второй реализацией, необходимо поменять значения только в одной точке, а поля длины и ширины останутся неизменными – одна и та же операция привела к меньшему изменению данных.

Аналогично для операции изменения размеров: если мы захотим увеличить по высоте элемент управления, то нам необходимо поменять координаты Y в двух точках в первой реализации, и всего лишь одно значение _height во второй реализации.

Таким образом, правильное представление объектов в виде полей может повлиять как на удобство использования классов, количество совершаемых операций и безопасность работы программы. Во многих случаях избежать зависимых полей невозможно, однако задача разработчика создать наиболее эффективную и удобную в использовании реализацию.

Поля, доступные только на чтение

При разработке программ часто требуются поля, для которых нужно гарантировать, что их значения не будут изменены. Простой пример – это любой идентификационный номер Id: номер паспорта, страховой номер или номер записи в журнале учета. Идентификационные номера должны быть уникальными и ни в коем случае не повторяться между объектами. Те, кто знаком с курсом баз данных, знают, что в каждой таблице БД как правило создаются собственные поля ключа Id. При каждой новой записи в таблице создается новый, следующий по номеру Id. При удалении записи, её Id больше не используется. Этот приём позволяет при создании запросов в БД определять уникальные не повторяющиеся записи и избегать коллизий.

Из структурного программирования читателю должны быть известны константы. ***Константы*** — это переменные, значение которых после инициализации не может быть изменено. В Си++ константы объявляются с помощью ключевого слова const. Константы могут быть глобальными, а могут быть объявлены внутри структур и классов, в том числе и скрытые под модификатором доступа private:

```
// глобальная константа, указывающая точность знаков после запятой
// для вывода чисел на экран во всей программе
const int DefaultPrecision = 3;

class Circle
{
private:
    // скрытая внутри класса константа Пи, используемая для расчета площади круга
    const double PI = 3.14159;

    double _radius;

public:
    void SetRadius(double radius);
    double GetRadius();

    Circle(double radius);

    void WriteAreaToConsole();
};

// Метод расчета и вывода площади круга на экран
void Circle::WriteAreaToConsole()
{
    // Расчет площади с использованием константы
    double area = PI * this->GetRadius() * this->GetRadius();

    // Задание точности вывода с помощью глобальной константы
    cout.precision(DefaultPrecision);

    // Вывод на экран с заданной точностью
    cout << area;
}
```

В примере выше показаны объявление глобальной (доступной в любой точке кода) и локальной (доступной только внутри класса) констант, а также их использование внутри метода WriteAreaToConsole().

Однако константы задаются один раз и являются общими для всех объектов программы. Логика же идентификационных номеров предполагает, что у ***каждого*** объекта будет свой собственный уникальный идентификационный номер.

Например, мы хотим написать класс Студент, но хотим избежать возможности полных однофамильцев, учащихся в одной группе (как ни странно, но такое случается). В такой ситуации, помимо полей фамилии, имени и номера группы, нам необходимо хранить уникальный Id, отличающийся для каждого студента. Таким Id может служить номер зачетной книжки:

```
class Student
{
    int _id;
    string _surname;
    string _name;
    string _group;

public:
    void SetId(int id);
    void SetSurname(string& surname);
    void SetName(string& name);
    void SetGroup(string& group);

    int GetId();
    string& GetSurname();
    string& GetName();
    string& GetGroup();

    Student(int id, string& surname, string& name, string& group);
};
```

После создания объекта Student, его _id может быть изменен с помощью сеттера SetId(). Отсюда вытекает простая идея – чтобы клиентский код не мог изменить _id студента, необходимо скрыть его сеттер:

```
class Student
{
    int _id;
    string _surname;
    string _name;
    string _group;

    void SetId(int id); // теперь сеттер под private!

public:
    void SetSurname(string& surname);
    void SetName(string& name);
    void SetGroup(string& group);

    int GetId();
    string& GetSurname();
    string& GetName();
    string& GetGroup();

    Student(int id, string& surname, string& name, string& group);
};
```

Таким образом, клиентской код может указать _id для объекта только один раз – в момент вызова конструктора. После этого он не сможет его изменить, но сможет его узнать с помощью геттера GetId(). Другими словами, поле _id стало ***доступным только на чтение***:

```
void Readonly_Example()
{
    Student* student = new Student(100101, "Смирнов", "Юрий", "517");
    cout << student->GetId();
    // student->SetId(100102); Ошибка компиляции ! Метод сокрыт
}
```

Примечание: в рамках данного раздела "доступный только на чтение" подразумевает поля с закрытым сеттером. При этом значение поля может изменяться внутри класса. В других языках программирования и в других контекстах "доступный только на чтение" может означать в том числе и поля, которые не могут изменяться даже внутри класса – то есть, их значения остаются без изменений после вызова конструктора.

С понятием доступных только на чтение полей связаны еще два важных понятия – изменяемые и неизменяемые объекты.

Изменяемый объект (mutable) – объект, состояние которого может быть изменено после его создания.

Неизменяемый объект (immutable) – объект, состояние которого не может быть изменено после его создания.

Данные определения не строгие, так как объекты могут быть неизменяемыми частично, а могут быть неизменяемыми полностью. Как отмечено выше, объекты могут быть неизменяемыми со стороны клиентского кода, а могут быть неизменяемыми даже внутри самого класса. Однако, несмотря на свою нестрогость, данные определения можно часто встретить в литературе по программированию. Почему данные определения важны?

Создание закрытых методов, полей доступных на чтение и даже полей недоступных на чтение даёт возможность разработчику создания так называемых ***конечных автоматов***. Данное понятие может быть известно читателю из курса дискретной математики. Говоря простыми словами, конечный автомат – это объект, который в один момент времени находится в одном из своих состояний (их конечное количество, из-за чего автомат и называется конечным). В зависимости от своего текущего состояния, объект может вести себя поразному. Рассмотрим на гипотетическом примере:

```
class StateMachine
{
    int _state;

    void DoSomething1();
    void DoSomething2();
    void DoSomething3();

public:
    StateMachine();
    void Do();
};

void StateMachine::Do()
{
    int action = _state % 3;
    switch (action)
    {
    case 1: DoSomething1(); break;
    case 2: DoSomething2(); break;
    case 3: DoSomething3(); break;
    }
    _state++;
}
```

В данном примере класс конечного автомата StateMachine хранит в поле своё текущее состояние в виде целочисленной переменной _state. Однако узнать текущее состояние объекта нельзя, так как нет соответствующего геттера. Для клиентского кода в этом классе есть только метод Do(), который в зависимости от текущего значения состояния _state выполняет одно из трёх действий DoSomething1(), DoSomething2(), DoSomething3() – под ними могут подразумеваться любые действия, это не важно для примера.

При каждом вызове метода Do() состояние конечного автомата меняется, при чем клиентский код не знает, как именно меняется состояние внутри объекта. И здесь возникает очевидная проблема использования подобных классов – нельзя достоверно знать, что сделает объект, если вызывать его методы. Поведение объекта становится непредсказуемым и неконтролируемым.

Нельзя сказать, что конечные автоматы – это плохо. Конечные автоматы успешно применяются в программировании, например, в тестировании программ, или в создании компиляторов и эмуляторов – в задачах для опытных разработчиков. Однако начинающие программисты часто излишне увлекаются новыми изученными техниками и приёмами, что приводит к созданию сложных громоздких классов с непредсказуемым поведением наподобие конечных автоматов.

Данный пример приведен как предостережение и рекомендация: программист должен писать настолько простые и предсказуемые классы, насколько это возможно для решения задачи.

Конечный автомат является одним из наиболее сложных вариантов изменяемого (mutable) объекта. Его противоположностью являются неизменяемые объекты (immutable) – объекты, чьё состояние не меняется в ходе жизни объекта, и чьё поведение предельно предсказуемо со стороны клиентского кода. Разработчик по возможности должен отдавать предпочтение реализации именно immutable-объектов – в той степени, насколько это не мешает оптимальному решению задачи.

Более подробно тема неизменяемых объектов будет изучена в объектно-ориентированном анализе и проектировании. На данном этапе читателю предлагается просто запомнить определения данных понятий.

Статические поля и методы класса

В предыдущем разделе был создан класс студента Student с защищенным от внешних изменений полем _id. Идентификационный номер задается с помощью конструктора. Это позволяет гарантировать, что _id не изменится в ходе выполнения программы. Однако, это не решает проблемы уникальности идентификационных номеров у объектов класса. Например, следующий код вполне корректен:

```
void SameIdsExample()
{
    Student* student1 = new Student(100101, "Смирнов", "Юрий", "517");
    Student* student2 = new Student(100101, "Нилов", "Михаил", "587");
}
```

К сожалению, такая реализация позволяет задавать одинаковые Id. Разумеется, идентификационные номера не присваиваются объектам напрямую. Для генерации уникальных номеров создают специальные функции. Примером простого генератора идентификационных номеров может служить следующая реализация функции с использованием глобальной переменной:

```
// Глобальная переменная, хранящая последний сгенерированный Id
int LastStudentId = 100000;

// Функция для получения нового Id
int MakeNewStudentId()
{
    LastStudentId++;
    return LastStudentId;
}

void IdsGenerationExample()
{
    Student* student1 = new Student(MakeNewStudentId(), "Смирнов", "Юрий", "517");
    Student* student2 = new Student(MakeNewStudentId(), "Нилов", "Михаил", "587");
    cout << student1->GetId() << endl;
    cout << student2->GetId() << endl;
}
```

Принцип работы данной реализации следующий: номер последнего сгенерированного Id хранится в глобальной переменной, фактически являясь счетчиком. Вызов функции MakeNewStudentId() приводит к увеличению счетчика на 1 и возвращению этого значения. При повторном вызове функции счетчик снова увеличится и вернет новое значение Id. Таким образом, гарантируется, что все новые Id не будут повторяться. Главное, это

всегда использовать данную функцию для генерации Id. В функции IdsGenerationExample() продемонстрировано использование данной функции. При запуске программы, функция выведет на экран:

```
100001
100002
```

Единственное, что остается реализовать разработчику в реальной программе – это сохранение значения глобальной переменной в файл при завершении программы. Действительно, если программу запустить снова, то значение глобальной переменной снова будет 100000, а это приведет к повторной генерации значений Id, которые уже могут быть у студентов. Чтобы этого избежать, необходимо написать две функции: одна функция будет сохранять значение глобальной переменной при завершении программы в файл, а вторая – загружать сохраненное значение из файла при запуске программы. Впрочем, в любой более-менее сложной программе необходимо создавать функции по сохранению и загрузке настроек программы.

Реализация с использованием глобальной переменной имеет очевидные недостатки:

1) Глобальная переменная, необходимая только для работы класса Student, будет доступна в любой точке программы – любой клиентский код может присвоить новое значение, нарушив правильную генерацию Id.
2) Функция MakeNewStudentId() также доступна в любой точке кода, может вызываться не только для генерации Id в конструкторах Student.

Исправить данные недостатки позволяет реализация на основе статических полей и методов.

Статические поля – это поля класса, общие для всех объектов данного класса.

Статические методы – это методы класса, вызываемые не от лица конкретного объекта, а вызываемые от лица класса, даже при условии отсутствия объектов данного класса.

Статические поля и методы обозначаются модификатором static, также могут быть закрытыми или открытыми. Статические методы также могут принимать аргументы любых типов данных и возвращать значения. Рассмотрим их применение на практике:

```
class Student
{
private:
    static int LastId = 100000;

    int _id;
    string _surname;
    string _name;
    string _group;

    static int MakeNewId();
    void SetId(int id);
    void CheckString(string value);

public:
    void SetSurname(string surname);
    void SetName(string name);
    void SetGroup(string group);

    int GetId();
    string GetSurname();
    string GetName();
    string GetGroup();

    Student(string surname, string name, string group);
};

int Student::MakeNewId()
```

```
{
    Student::LastId++;
    return Student::LastId;
}

Student::Student(string surname, string name, string group)
{
    SetId(Student::MakeNewId());
    SetSurname(surname);
    SetName(name);
    SetGroup(group);
}
```

Изменения в классе выделены жирным. Во-первых, это объявление статической переменной с инициализацией `static int LastId = 100000;` - объявление аналогично объявлению обычного поля с добавлением модификатора static.

Во-вторых, это объявление закрытого статического метода `static int MakeNewId()`. Опять же объявление аналогично объявлению обычных методов класса с добавлением модификатора static.

Определение метода MakeNewId() под объявлением класса аналогично определению обычных методов: указание возвращаемого типа, указание название класса, для метода которого происходит определение, и указание название самого метода.

Но меняется способ обращения к полю и методу. Пример обращения к статическому полю показан внутри определения метода MakeNewId(). Статическое поле не относится ни к одному из объектов класса Student, оно является общим для всех объектов. Поэтому обращение к полю происходит не с указанием имени объекта (переменной типа Student), а через имя самого класса:

```
Student::LastId++;
```

Аналогично, с указанием имени класса происходит вызов статического метода:

```
Student::MakeNewId()
```

Например, в строке конструктора результат вызова статического метода MakeNewId() сразу передается в закрытый метод SetId():

```
SetId(Student::MakeNewId());
```

При обращении к статическим полям или методам внутри самого класса можно не указывать часть спецификатора класса `Student::`, а обращаться к полю или методу напрямую. В коде спецификатор указан для наглядности вызова. При этом, в отличие от примера с глобальной переменной, из названия переменной и метода убрано слово Student – так как вызываться статические поля и методы в клиентском коде могут только с указанием имени класса, слово Student в имени самого поля или метода будет лишним.

При разработке статических методов есть ограничение – статические методы могут обращаться только к статическим полям класса, но не могут обращаться к обычным полям. Это логично, так как статический метод вызывается не от определенного объекта, а от класса в целом.

Статические поля и методы во многом схожи с глобальными переменными и обычными функциями, но имеют ряд преимуществ перед ними:

Статические поля и методы объявляются внутри определенного класса, что упрощает их поиск – если хочешь найти глобальные переменные, связанные с классом студентов, поищи их в классе студентов.

Статические поля могут быть не только простыми типами данных (int, double, char), но и составными, например, перечислениями, другими классами и структурами. Статические методы могут принимать входные аргументы различных типов, возвращать любые типы данных. Внутри статического метода можно, например, создавать объекты класса.

Статические поля и методы могут быть под модификатором доступа – если глобальная переменная нужна исключительно для внутренней работы класса, но должна быть недоступна вне класса, то лучше это значение хранить в закрытом статическом поле. Аналогично могут быть скрыты статические методы.

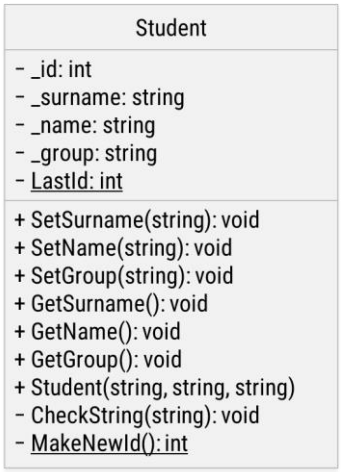
Благодаря статическим полю и методу мы смогли сделать реализацию класса Student, которая имеет следующие преимущества:

а) уникальные Id для каждого нового объекта генерируются непосредственно в классе.

б) клиентский код никак не участвует в генерации Id – класс проще использовать, и сложнее нарушить его правильную работу.

Впрочем, реализация с закрытым статическим полем не позволяет сделать сохранение и загрузку последнего использованного Id в файл (так как обратится к закрытому полю извне нет возможности), а потому не рекомендуется к применению на практике. Данный пример был представлен только для демонстрации статических полей и методов.

Статические члены класса на UML-диаграммах классов обозначаются с подчеркиванием:



Отдельно стоит сказать про именование статических членов классов:

1) статические поля именуются в стиле Паскаль без нижнего подчеркивания вне зависимости от модификатора доступа;
2) статические методы именуются как и обычные методы вне зависимости от модификатора доступа.

Сервисные и статические классы

Ранее в классе Student был создан закрытый метод CheckString(), выполняющий проверку строки на отсутствие иных символов, кроме букв. Метод используется в сеттерах SetSurname() и SetName() для проверки правильности фамилии и имени соответственно – закрытый метод помог избавиться от дублирования кода внутри одного класса.

Однако аналогичная проверка может понадобиться в другом классе. Например, в приложение необходимо добавить класс преподавателя Teacher, в котором также есть поля фамилии и имени. Логично, что их проверка будет подчиняться тем же правилам.

Первое решение, которое приходит на ум – это копирование метода CheckString() в класс Teacher. Решение плохое, так как приводит к дублированию кода. Во-первых, если проверку нужно будет модифицировать, её придется исправлять в обоих классах. Во-вторых, если в методе CheckString() уже есть ошибка, то эта ошибка копируется в другой класс.

Второе неправильное решение, которое можно встретить у начинающих разработчиков – это помещение метода CheckString() класса Student под модификатор public. Логика разработчика здесь следующая: чтобы не дублировать метод, надо предоставить доступ к уже существующему. Однако на практике использование такого открытого метода выглядит следующим образом:

```
void Teacher::SetSurname(string surname)
{
    Student* student = new Student("", "", "");
    student->CheckString(surname);
    _surname = surname;
}
```

Данное решение приводит к еще большему числу недостатков, чем дублирование кода в первом варианте. Фактически, чтобы вызвать метод класса Student, сеттер SetSurname() в классе Teacher должен создать объект студента. Объект создается с пустыми или случайными данными, и только после этого происходит проверка. Это приводит к следующим недостаткам:

1) С точки зрения реальной предметной области, преподаватель чтобы запомнить свою фамилию, вызывает студента и просит её проверить. Работа кода не соответствует реальной предметной области, что затрудняет изучение и поддержку кода новыми разработчиками в проекте.
2) Если в конструкторе класса Student создаются Id, то каждое создание студента будет приводить к генерации неиспользуемых номеров.
3) Если бы класс Student агрегировал другие классы, например, изучаемые дисциплины, то создание объекта студента могло бы занять большое количество кода. Большее, чем используемый метод проверки строки.
4) Появляется лишняя связь использования между классом Teacher и Student – это означает, что при модификации класса Student увеличивается вероятность изменений и в классе Teacher. Поэтому одна из целей проектирования программы заключается в минимизации связей между классами.
5) Очень легко забыть оператор delete для освобождения памяти созданного объекта (как в примере выше), что приводит к мелким утечкам памяти. Заметили ли вы отсутствие оператора delete в сеттере до того, как прочитали этот пункт?

Третье неправильное решение – это сделать закрытый метод CheckString() в классе Student открытым и статическим. Действительно, метод не использует полей класса Student и может быть легко сделан статическим. Это решает проблему создания не нужных объектов и утечек памяти, но остается проблема несоответствия реальной предметной области и лишней связи между классами. Это важные проблемы с точки зрения поддержки программы, и их нельзя игнорировать.

Есть и другие неправильные решения, допускаемые начинающими программистами, такие как передача экземпляра студента вместе с фамилией в сеттер SetSurname() преподавателя, или создание поля типа _fakeStudent внутри объекта преподавателя, чтобы не создавать новый объект студента при каждом вызове сеттера. Это очевидно неправильные решения.

Правильное решение – создание ***сервисного класса*** для проверки правильности строк. ***Сервисный класс*** – это класс, как правило, не имеющий полей, и предоставляющий вспомогательные, часто используемые методы для других классов.

Например, создадим сервисный класс NameChecker, предоставляющий метод для проверки правильности имен и фамилий:

```
class NameChecker
{
public:
    void CheckNameForLetters(string name);
};

void NameChecker::CheckNameForLetters(string name)
{
    for (int i = 0; i < name.length(); i++)
    {
            bool isLetter =
                    (name[i] > 'A' && name[i] < 'Z')
                    || (name[i] > 'a' && name[i] < 'z');
            // если символ в строке не является буквой, кидаем исключение
            if (!isLetter)
            {
                    throw exception("Строка не должна содержать символов, кроме букв");
            }
    }
}
```

Класс NameChecker не хранит никаких полей, сеттеров и геттеров, а хранит только один метод CheckNameForLetters(), аналогичный методу CheckString(). Метод переименован для того, чтобы лучше отражать решаемую задачу. Если для метода CheckString() класса Student это было не так важно – метод использовался внутри класса, где было сложно перепутать контекст применения, то метод CheckNameForLetters() может использоваться во многих классах и функциях – классах студента, учителя или в пользовательском интерфейсе для проверки введенных или загруженных значений. Метод CheckString() из класса Student при этом удаляется и заменяется на вызова метода из класса NameChecker:

```
void Student::SetSurname(string surname)
{
    NameChecker* checker = new NameChecker();
    checker->CheckNameForLetters(surname);
    _surname = surname;
    delete checker;
}
```

Аналогично реализуется и сеттер для имени студента, а также сеттеры преподавателя. Создав данный сервисный класс, мы избавились от несуществующей в реальной предметной области связи между студентом и фамилией преподавателя – это упростит читаемость и понимание кода другими разработчиками.

Так как метод CheckNameForLetters() для своей работы не использует никаких полей класса NameChecker, то можно упростить использование данного метода, сделав его статическим:

```
class NameChecker
{
public:
    static void CheckNameForLetters(string name);
};
```

Тогда не понадобится вызывать конструктор класса NameChecker и оператор delete в сеттерах:

```
void Student::SetSurname(string surname)
{
    NameChecker::CheckNameForLetters(surname);
    _surname = surname;
}
```

Так как единственный метод класса является статическим, то правильнее сделать статическим весь класс:

```
static class NameChecker
{
public:
    static void CheckNameForLetters(string name);
};
```

Статический класс – это класс, использование которого не предполагает создания объектов данного класса. Модификатор static перед объявлением класса указывает, что все члены этого класса обязаны быть статическими, а создание объектов класса невозможно.

На первый взгляд может показаться, что добавление модификатора static для класса ничего не меняет в использовании его статических методов. Но есть ряд существенных отличий:

1) Для нестатического класса создавался конструктор по умолчанию, с помощью которого можно создавать объекты NameChecker. Несмотря на то, что у объектов нет никаких полей или методов, давать возможность создавать экземпляры подобного класса будет неправильным. Статический класс не приводит к созданию конструктора по умолчанию, и не позволяет создавать экземпляры класса.
2) Статический класс невозможно агрегировать, его можно только использовать. Объекты любого нестатического класса могут быть статическими полями другого класса, таких объектов может быть любое количество. Для статического класса не существует его экземпляров. Грубо говоря, статический класс гарантирует существование только одного экземпляра класса, вызываемого по его имени.

Данные отличия статического класса от нестатического класса со статическими методами и полями позволяют создать более жесткие ограничения на использование класса, тем самым, в ряде случаев, сделать использование класса более безопасным для клиентского кода.

Не стоит путать сервисные классы и статические. Сервисные классы определяются по их назначению, а статические классы – по реализации в исходном коде. Статические классы, как правило, являются сервисными, но не все сервисные классы являются статическими.

В примере выше, сервисный класс содержит только один метод. Но сервисные классы могут содержать множество методов с различными проверками или другими алгоритмами. В предыдущих разделах говорилось, что классы описывают предметы из реального мира: студент, автомобиль, песня, фильм – всё это объекты, существующие в реальном мире, и которые могут быть описаны предметной областью. Однако в природе вряд ли существует такой объект как "Проверяльщик имён". Мы выдумали такой объект, наделив его обязанностями в нашей программе. Часто в программировании приходится создавать такие классы, которые являются так называемыми ***чистыми выдумками***. Данный приём используется в тех случаях, когда существование такого объекта делает архитектуру менее связной, а код – читаемее. Обязательным требованием к такому классу является выполнение строго одной задачи. Сортировщик, поисковик, валидатор, конвертер и т.д. – всё это примеры сервисных классов и чистых выдумок.

На самом деле, механизм статических полей, методов и классов (статических, но не сервисных!) имеет ряд недостатков, ограничивающих возможности разработчика (ограничения при наследовании и полиморфизме – механизмах, о которых будет рассказано позже), а потому их применение в разработке должно быть ограничено. Удобство вызова методов класса без создания его объектов может показаться преимуществом, и появляется соблазн написания статических методов и классов везде, где это возможно. Но это будет ошибкой, отталкивающей разработчика обратно в сторону процедурного программирования и не давая использовать механизмы объектно-ориентированной парадигмы. На практике статические классы и методы следует использовать для:

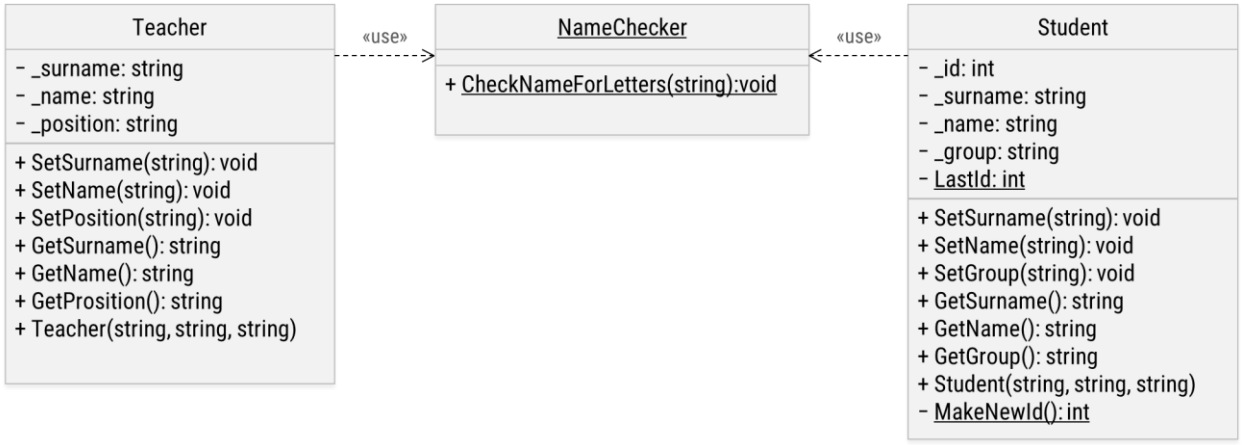
а) разного рода конвертеров (например, чисел в строки определенного формата);

б) небольших, но часто используемых в программе расчетных формул (например, перевода градусов в радианы, или расчета пересечения геометрических фигур);

в) валидаторов (для полей классов или значений, вводимых пользователем);

г) глобальных настроек приложения, где хранятся, например, разрешение экрана, языковые настройки и т.д. (но только в небольших приложениях).

На UML-диаграммах классов статические классы обозначаются как обычные классы, но с подчеркнутыми именем класса и всеми его членами:



Как было сказано ранее, статический класс может быть только использован, поэтому на диаграммах классов он может быть связан с другими классами верхнего уровня только использованием. С другой стороны, статический класс может агрегировать и использовать другие классы.

Также стоит сказать про именование методов для валидации данных, таких как CheckNameForLetters(). В зависимости от задачи, валидация может возвращать разные объекты. В зависимости от возвращаемого значения, используйте разное именование методов:

1) Используйте именование со слов Check ("проверить") или Is ("является") для методов, возвращающих логическое значение true или false (проверка на "Да, является правильным" или "Нет, является не правильным"). Например, CheckNameForLetters() или IsNameContainsLetters().
2) Используйте именование со слов Correct ("исправить") или To ("Привести к") для методов, возвращающих исправленное значение входного аргумента. Например, CorrectNameCase() или ToCorrectCase(), если метод принимает на вход имя из букв разного регистра, а возвращает имя в нижнем регистре с заглавной буквой.
3) Используйте именование со слова Assert ("утверждать") для методов, выкидывающих исключение в случае не прохождения проверки. Например, AssertNameForLetters().

Метод с выбросом исключения:

```
void NameChecker::AssertNameForLetters(string name)
{
    for (int i = 0; i < name.length(); i++)
    {
        bool isLetter =
            (name[i] > 'A' && name[i] < 'Z')
            || (name[i] > 'a' && name[i] < 'z');
        // если символ в строке не является буквой, кидаем исключение
        if (!isLetter)
        {
            throw exception("Строка не должна содержать символов, кроме букв");
        }
    }
```

```
}
```

Метод с возвращением логического значения:

```
bool NameChecker::IsNameContainsLetters(string name)
{
    for (int i = 0; i < name.length(); i++)
    {
           bool isLetter =
                  (name[i] > 'A' && name[i] < 'Z')
                  || (name[i] > 'a' && name[i] < 'z');
           // если символ в строке не является буквой, возвращаем false
           if (!isLetter)
           {
                  return false;
           }
    }
    // если все символы в строке прошли проверку, возвращаем true
    return true;
}
```

Применение данной рекомендации повышает читаемость кода. Так, по слову Is в названии метода разработчик сразу понимает, что метод возвращает логическое значение, и его можно использовать в условном операторе. А по слову Assert разработчик понимает, что метод может сгенерировать исключение, и его нужно обвернуть в блоки try-catch. Согласно описанной рекомендации, метод CheckNameOnLetters() в ранее представленном классе NameChecker правильнее переименовать в AssertNameOnLetters().

Задания

Зависимые поля

4.1.1 Исправьте классы, разработанные в предыдущей лабораторной работе: если класс агрегирует массив значений, а массив и количество элементов массива задаются двумя раздельными сеттерами, переделайте его раздельные сеттеры в один сеттер для задания зависимых полей. Это должны быть классы Album и Band.

4.1.2 Исправьте те места кода, где вызывались измененные сеттеры.

4.1.3 Скомпилируйте программу и убедитесь, что программа работает корректно после изменения сеттеров.

4.1.4 Создайте класс кольца Ring, хранящего значения внешнего и внутреннего радиусов, а также центра координат кольца (объект ранее созданного класса Point). Реализуйте сеттеры и геттеры для полей класса. Внешний и внутренний радиус должны быть вещественными положительными числами, внутренний радиус не может быть больше внешнего, а внешний – меньше внутреннего. В случае неправильных значений полей, сеттер должен генерировать исключение. Реализуйте конструктор класса.

4.1.5 Добавьте в класс Ring метод double GetArea(), возвращающий площадь кольца. Площадь кольца равна разности площади круга, образуемого внешним радиусом, и площади круга, образуемого внутренним радиусом.

4.1.6 Создайте функцию DemoRing(), в которой создайте несколько экземпляров колец, и выведите их площадь на экран. Убедитесь, что невозможно создать кольца с неправильными значениями радиусов: обверните в блок try-catch создание кольца с отрицательными радиусами и кольца, где внутренний радиус больше внешнего. В блоке catch выведите сообщение о пойманном исключении.

Закрытые методы класса

4.2.1 В сеттерах класса Ring реализованы проверки на неотрицательность значений радиусов. Вынесите проверки во внутренний закрытый метод AssertOnPositiveValue(double value). Метод принимает на вход проверяемое вещественное значение и, если оно отрицательное, генерирует исключение. Сеттер класса для радиусов должен использовать новый метод.

4.2.2 Запустите функцию DemoRing() и убедитесь, что программа работает как и прежде.

4.2.3 В функции DemoRing() попытайтесь обратиться к закрытому методу AssertOnPositiveValue() – убедитесь, что компилятор не позволяет вызывать закрытые методы.

Поля, доступные только на чтение

4.3.1 Переделайте класс Point таким образом, чтобы поля координат X и Y были доступны только на чтение. Значения координат могут задаваться только через конструктор и более не меняются. Другими словами, переделайте класс Point, чтобы его объекты были неизменяемыми (immutable). Небольшие структуры и классы типа Point часто делают неизменяемыми.

4.3.2 Перепишите те участки программ, где ранее использовались сеттеры класса Point (если использовались) – теперь вместо вызова сеттеров должно происходить создание нового объекта точки.

4.3.3 Убедитесь, что программа работает правильно.

Статические поля и методы класса

4.4.1 Создайте в классе Ring закрытое статическое целочисленное поле AllRingsCount. Поле предназначено для подсчета всех существующих объектов колец.

4.4.2 Создайте в классе Ring открытый статический метод GetAllRingsCount(), возвращающий значение статического поля AllRingsCount. Обратите внимание, что поле AllRingsCount фактически является доступным только на чтение.

4.4.3 Добавьте в конструктор класса Ring строку, увеличивающую значение статического поля AllRingsCount на единицу. То есть, при каждом вызове конструктора должно происходить увеличение счетчика.

4.4.4 Добавьте в класс Ring деструктор. Деструктор должен состоять из одной строки, в которой счетчик AllRingsCount уменьшается на единицу. То есть, при каждом вызове деструктора должно происходить уменьшение счетчика.

4.4.5 В функции DemoRing() после каждого создания объекта Ring добавьте вывод на экран значения AllRingsCount, используя статический метод класса.

4.4.6 Добавьте в функцию DemoRing() следующий код:

```
cout << "Количество колец до вызова конструктора: "<< Ring::GetAllRingsCount();

Ring* ring = new Ring(10.0, 5.0, Point(25.0, 25.0));
cout << "Количество колец после вызова конструктора: " << Ring::GetAllRingsCount();

delete ring;
cout << "Количество колец после вызова деструктора: " << Ring::GetAllRingsCount();
```

4.4.7 Убедитесь, что счетчик AllRingsCount показывает правильное количество объектов кругов в программе, и что количество уменьшается после вызова оператора delete.

Сервисные классы

4.5.1 Создайте статический класс DoubleValidator, хранящий методы по проверке вещественных чисел в программе. В классе должны быть следующие методы:

- bool IsValuePositive(double value) – метод принимает значение и, возвращает true, если оно положительное, и false, если оно отрицательное.
- bool IsValueInRange(double value, double min, double max) – метод принимает проверяемое значение и, если оно не попадает в диапазон между min и max, возвращает false, иначе возвращает true.
- void AssertPositiveValue(double value) – метод принимает значение и, если значение отрицательное, генерирует исключение. Для реализации метода вызывайте уже готовый метод IsValuePositive()
- void AssertValueInRange(double value, double min, double max) – метод принимает проверяемое значение и, если оно не попадает в диапазон между min и max, генерирует исключение. Для реализации метода вызывайте уже готовый метод IsValueInRange().

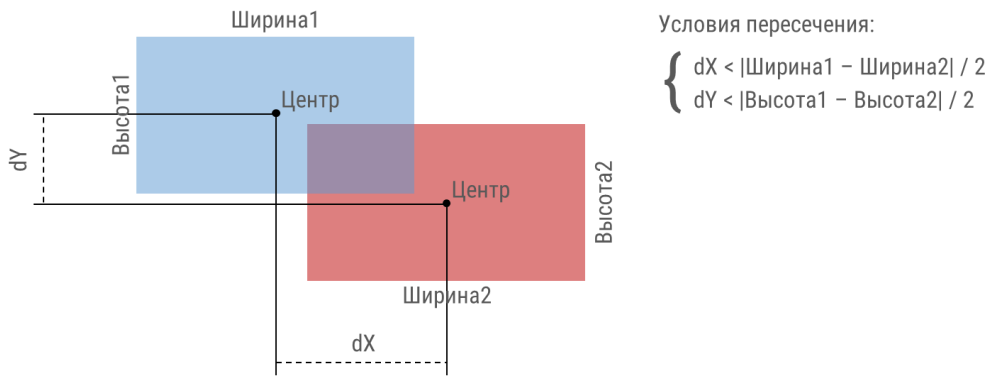
Методы проверки удобно делать в двух вариантах (Is и Assert), так как в пользовательском интерфейсе для проверки введенных пользователем значений удобно вызывать методы Is, возвращающие логические значения. В свою очередь в сеттерах классов для проверки входных значений удобнее использовать методы Assert.

4.5.2 Исправьте классы Ring и Rectangle на использование DoubleValidator в сеттерах (радиусы кольца, длина и ширина прямоугольника).

4.5.3 Запустите программу и убедитесь, что она работает правильно.

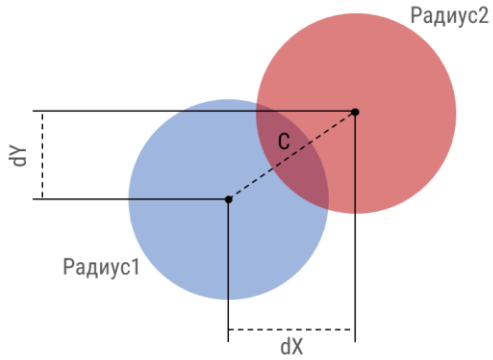
4.5.4 Создайте статический класс CollisionManager - класс, выполняющий проверку о пересечении/столкновении геометрических фигур. Классы по проверке столкновений применяются в геометрических САПР, 2D- и 3D-редакторах и компьютерных играх. Класс должен реализовать два метода:

- 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.



- bool IsCollision(Ring&, Ring&) – метод принимает два экземпляра колец и проверяет не пересекаются ли они. Кольца, как и круги, считаются пересекающимися, если гипотенуза треугольника, образующегося между координатами центров колец меньше суммы их внешних радиусов. По

желанию можете реализовать учет коллизии, при котором одно кольцо вписано внутрь другого кольца.



Условия пересечения:

C < (Радиус1 + Радиус2)

где C – гипотенуза треугольника, образованного катетами dX и dY

4.5.5 Создайте функцию DemoCollision для демонстрации расчета столкновений. Для демонстрации расчета столкновений создайте по две пары пересекающихся и не пересекающихся прямоугольников (координаты могут заданы в исходном коде) и выведите результат расчета столкновения на экран. Аналогично реализуйте демонстрацию расчета столкновения для колец.

4.5.6 Создайте сервисный (не статический) класс GeometricProgram, в который поместите все демонстрационные функции, связанные с геометрическими фигурами: DemoRing(), DemoCollision() и другие ранее разработанные функции. Класс GeometricProgram является чистой выдумкой пользовательского интерфейса для лабораторной работы №4. Реализация пользовательского интерфейса в виде классов также часто встречается на практике.

4.5.7 Экземпляр класса GeometricProgram должен создаваться в функции main() и, в зависимости от выбора пользователя, должны вызываться методы демонстрации прямоугольника, кольца или столкновений.

UML-диаграммы классов

4.6.1 Нарисуйте диаграмму классов по результатам лабораторной работы. На диаграмме отразить классы Point, Rectangle, Ring, DoubleValidator, CollisionManager, GeometricProgram. Проверьте связи между классами, правильность обозначенных модификаторов доступа и статических членов классов. Постарайтесь расположить классы иерархически, где более высокоуровневые классы отображены наверху, а низкоуровневые – внизу. Иерархическое расположение классов на диаграмме упрощает чтение диаграммы.