Лекция 8. Объектно-ориентированное программирование

Александр Смаль

CS центр

17 октября 2017

Санкт-Петербург

Ещё раз об ООП

Объектно-ориентированное программирование — концпеция программирования, основанная на понятиях объектов и классов.

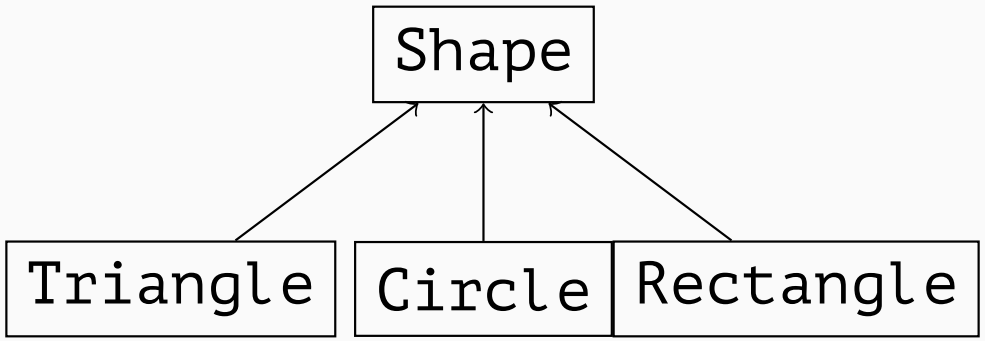
Основные принципы:

- инкапсуляция,
- наследование,
- полиморфизм,
- абстракция.

Подробнее о принципах проектирования ООП-программ можно узнать по ключевым слову „шаблоны проектирования”.

Как правильно построить иерархию?

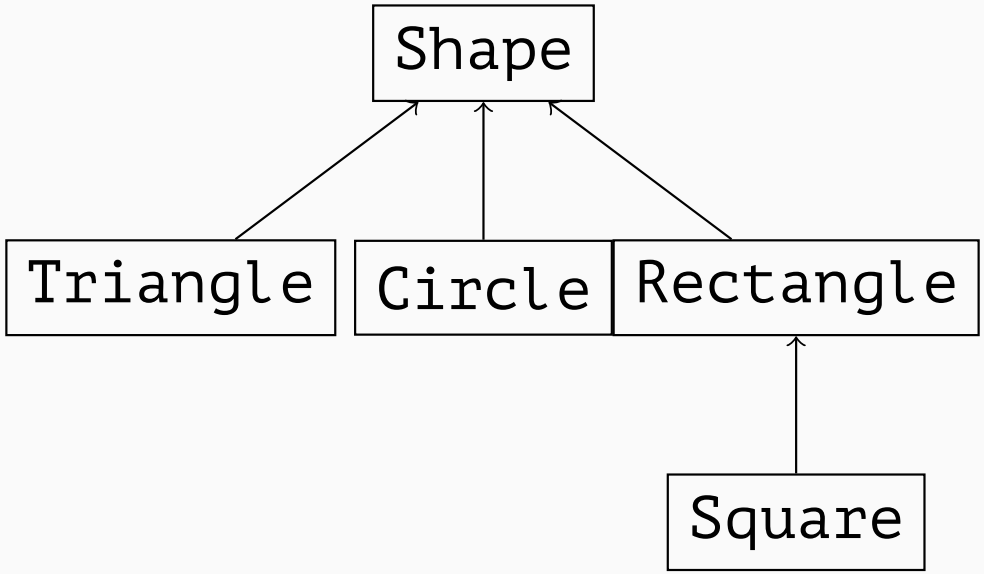
Иерархия геометрических фигур:



Куда добавить класс Square?

Как правильно построить иерархию?

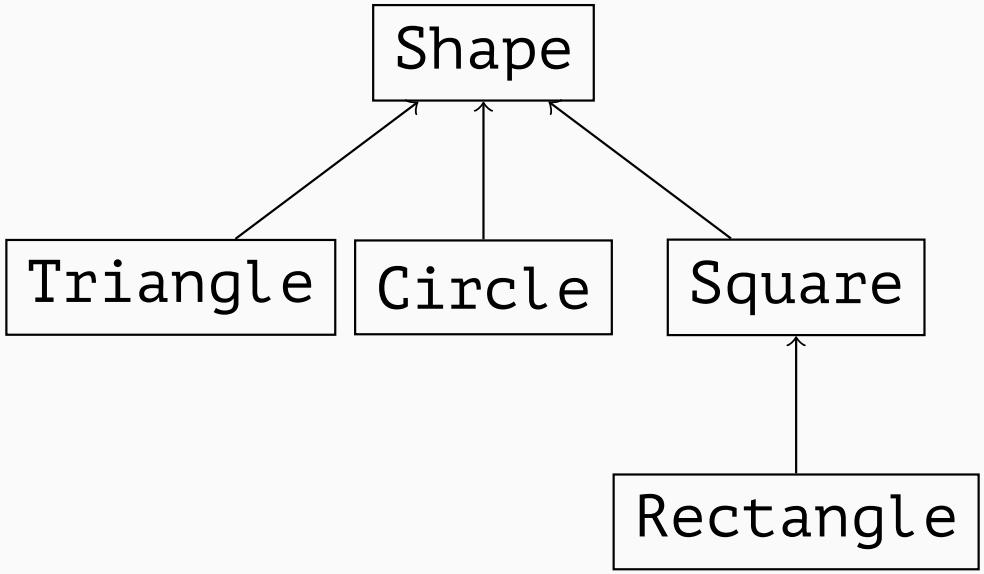
Квадрат — это прямоугольник, у которого все стороны равны.




```
void double_width(Rectangle & r) {
    r.set_width(r.width() * 2);
}
```

Как правильно построить иерархию?

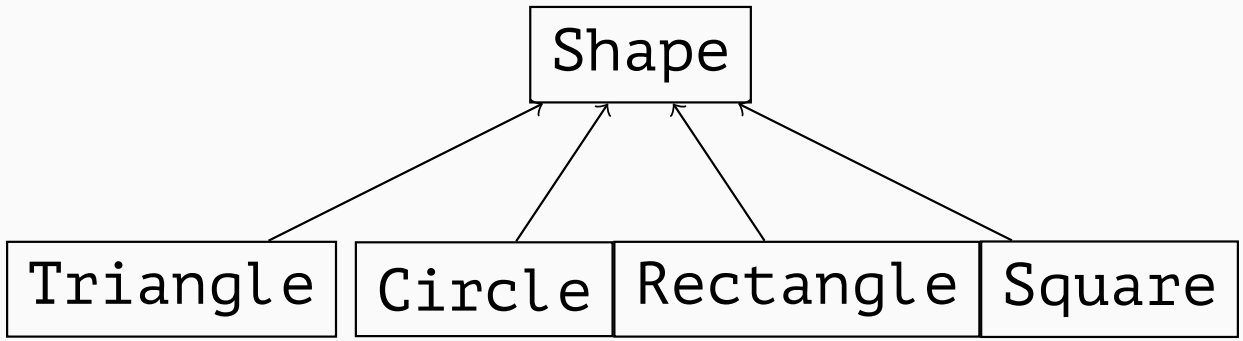
Прямоугольник задаётся двумя сторонами, а квадрат — одной.




```
double area(Square const& s) {
    return s.width() * s.width();
}
```

Как правильно построить иерархию?

Правильное решение — сделать эти классы независимыми:



Агрегирование vs наследование

- *Агрегирование* — это включение объекта одного класса в качестве поля в другой.
- Наследование устанавливает более сильные связи между классами, нежели агрегирование:
  - приведение между объектами,
  - доступ к `protected` членам.
- Если наследование можно заменить легко на агрегирование, то это нужно сделать.

Примеры некорректного наследования

- Класс `Circle` унаследовать от класса `Point`.
- Класс `LinearSystem` унаследовать от класса `Matrix`.

Принцип подстановки Барбары Лисков

Liskov Substitution Principle (LSP)

> *Функции, работающие с базовым классом, должны иметь возможность работать с подклассами не зная об этом.*

Этот принцип является важнейшим критерием при построении иерархий наследования.

Другие формулировки

- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.
- Подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс

Модификаторы при наследовании

При наследовании можно использовать модификаторы доступа:

```
struct A {};
struct B1 : public A {};
struct B2 : private A {};
struct B3 : protected A {};
```

Для классов, объявленных как `struct`, по-умолчанию используется `public`, для объявленных как `class` — `private`.

Важно: *отношение наследования* (в терминах ООП) задаётся только `public`-наследованием.

Использование `private`- и `protected`-наследований целесообразно, если необходимо не только агрегировать другой класс, но и переопределить его виртуальные методы.

Переопределение private виртуальных методов

```
struct NetworkDevice {
    void send(void * data, size_t size) {
        log("start sending");
        send_impl(data, size);
        log("stop sending");
    }
    ...
private:
    virtual void send_impl(void * data, size_t size)
    {...}
};

struct Router : NetworkDevice {
private:
    void send_impl(void * data, size_t size) {...}
};
```

Реализация чистых виртуальных методов

Чистые виртуальные методы могут иметь определения:

```
struct NetworkDevice {
    virtual void send(void * data, size_t size) = 0;
    ...
};

void NetworkDevice::send(void * data, size_t size) {
    ...
}

struct Router : NetworkDevice {
    void send(void * data, size_t size) {
        // невиртуальный вызов
        NetworkDevice::send(data, size);
    }
};
```

Интерфейсы

Интерфейс — это абстрактный класс, у которого отсутствуют поля, а все методы являются чистыми виртуальными.

```
struct IConvertibleToString {
    virtual ~IConvertibleToString() {}
    virtual string toString() const = 0;
};
```

```
struct IClonable {
    virtual ~IClonable() {}
    virtual IClonable * clone() const = 0;
};
```

```
struct Person : IClonable {
    Person * clone() {return new Person(*this);}
};
```

Множественное наследование

В C++ разрешено множественное наследование.

```
struct Person {};
struct Student : Person {};
struct Worker  : Person {};
struct WorkingStudent : Student, Worker {};
```

Стоит избегать *наследования реализаций* более чем от одного класса, вместо этого использовать интерфейсы.

```
struct IWorker {};
struct Worker : Person, IWorker {};
struct Student : Person {};
struct WorkingStudent : Student, IWorker {};
```

Множественное наследование — это отдельная большая тема.

Дружественные классы

```
struct String {
    ...
    friend struct StringBuffer;
private:
    char * data_;
    size_t len_;
};

struct StringBuffer {
    void append(String const& s) {
        append(s.data_);
    }
    void append(char const* s) {...}
    ...
};
```

Дружественные функции

Дружественные функции можно определять прямо внутри описания класса (они становятся `inline`).

```
struct String {
    ...

    friend void print(String const& s)
    {
        os << s.data_;
    }

private:
    char * data_;
    size_t len_;
};
```

Дружественные методы

```
struct String;
struct StringBuffer {
    void append(String const& s);
    void append(char const* s) {...}
    ...
};

struct String {
    ...
    friend
      void StringBuffer::append(String const& s);
};

void StringBuffer::append(String const& s) {
    append(s.data_);
}
```

Отношение дружбы

Отношение дружбы можно охарактеризовать следующими утверждениями:

- Отношение дружбы не симметрично.
- Отношение дружбы не транзитивно.
- Отношение наследования не задаёт отношение дружбы.
- Отношение дружбы сильнее, чем отношение наследования.

Вывод

Стоит избегать ключевого слова `friend`, так как оно нарушает инкапсуляцию.