Peter the Great
Saint-Petersburg Polytechnic University

Наука Программирования

Занятие №2
«Принцип единственной обязанности. Адаптер. Принцип разделения интерфейсов»

Осенний семестр 2017
Преподаватель: асс. каф. Чуканов В.С

11.09.17

Содержание

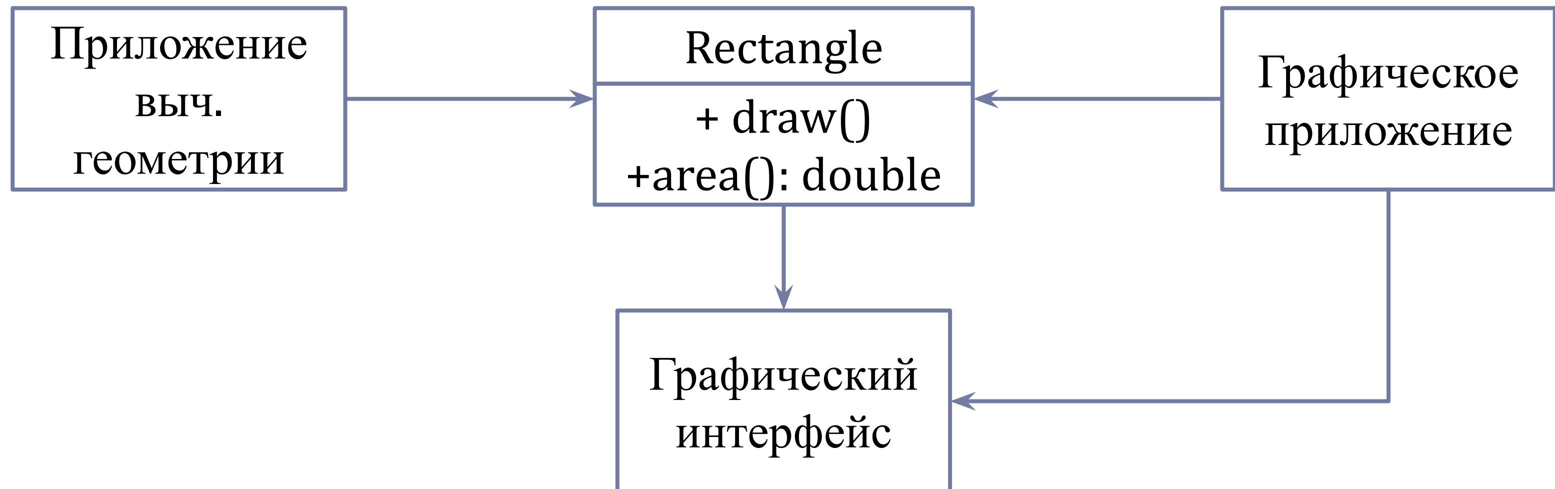
- Принцип единственной обязанности
- Адаптер
  - Соединение интерфейсов
  - Адаптер класса и адаптер объекта
  - Применение адаптера
- Принцип разделения интерфейсов
- Заключение

Принцип Единственной Обязанности

- АТД – абстрактный тип данных
  - Замкнутое множество данные + методы
- Single Responsibility Principle (SRP)
  - Класс должен иметь лишь одну причину для изменения
- Обязанность = ось изменения
  - Атомарный набор методы + данные = АТД
  - Принцип SRP: каждый класс реализует 1 АТД

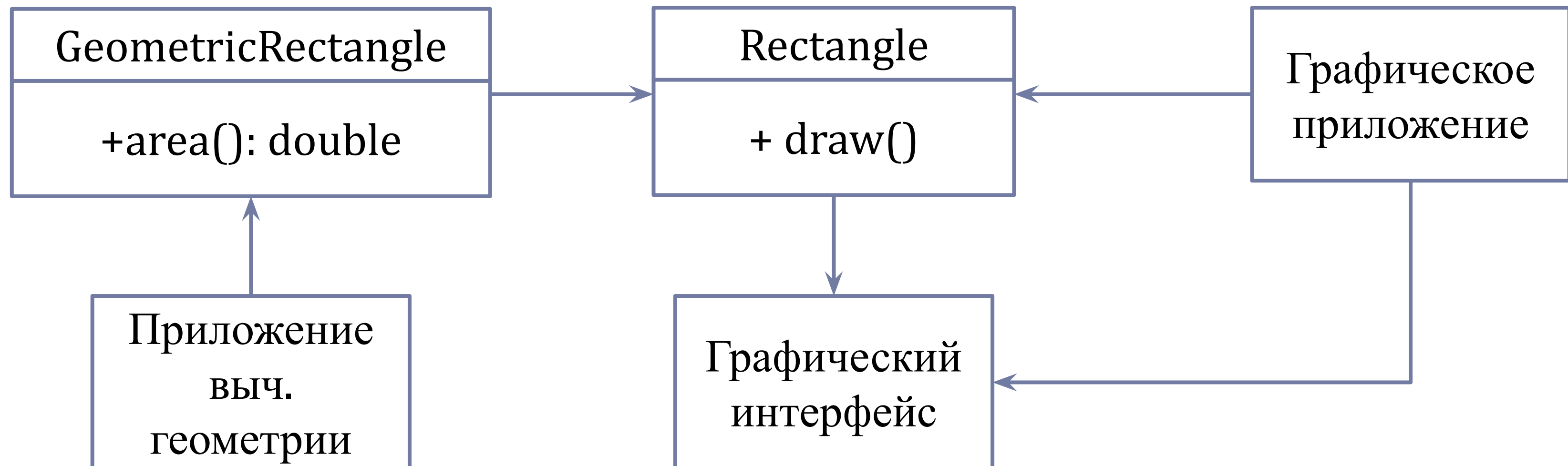
SRP: Пример Rectangle

- Rectangle
  - Используется для расчета площади и визуализации
  - Две обязанности = 2 АТД
    - Какие проблемы это может вызвать?



SRP: Решение Примера Rectangle

- Избыточная связь между приложениями выч. геом и визуализации
  - Изменение в модуле выч. геом могло привести к необходимости пересобирать модуль визуализации
- Решение: разделить обязанности Rectangle по двум классам



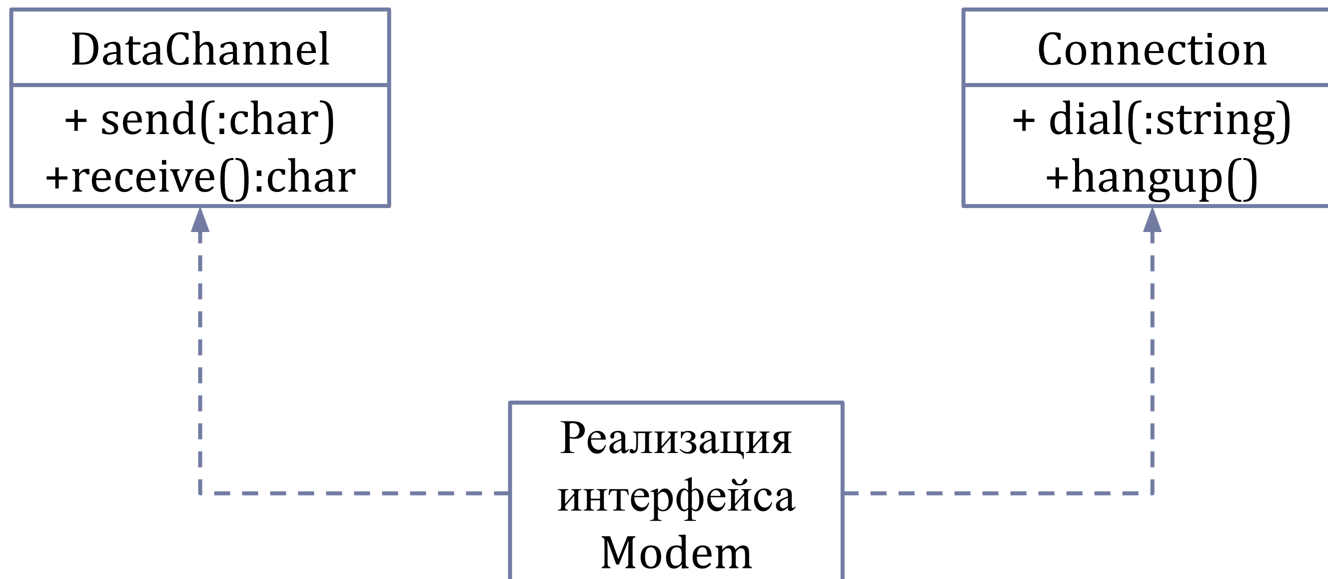
SRP: Пример Modem

- Modem
  - Интерфейс сетевого взаимодействия

```
class Modem
{
public:
    virtual void dial(std::string) = 0;
    virtual void hangup() = 0;
    virtual void send(char) = 0;
    virtual char receive() = 0;
};
```

SRP: Пример Modem

- Разделение обязанностей не всегда является необходимым
  - Особенности оборудования/ОС могут обуславливать слияние обязанностей в одном классе
  - Определяется постановкой задачи и возможностью изменения обязанностей независимо
  - Разделение обязанностей может быть реализовано с помощью паттернов Фасад (Facade) и Заместитель (Proxy)



Шаблон Проектирования: Адаптер

- Позволяет повторно использовать реализованную функциональность при несовместимых интерфейсах
- Технически – переадресация вызова от одного интерфейса к другому
- Пример
  - Имеется реализованный в библиотеке класс для генерации случайных, равномерно распределенных чисел в интервале [0, 1]
  - Необходимо написать класс для генерации чисел в интервале [0, 100]

Пример Адаптера: Код Библиотечных Классов

```
class ValueGenerator
{
public:
    virtual float getNormalizedValue() const = 0;
};

class ValueGeneratorStupid : public ValueGenerator
{
public:
    virtual float getNormalizedValue() const
    {
        return static_cast<float> (rand() % 10000) * 0.0001f;
    }
};
```

Пример Адаптера: Код Библиотечных Классов (2)

```
class ValueGeneratorUniform : public ValueGenerator
{
public:
    virtual float getNormalizedValue() const
    {
        //! C++11 stuff
        std::random_device device;
        std::mt19937 generator(device());
        std::uniform_real_distribution<float> distr(0.0f, 1.0f);
        return distr(generator);
    }
};
```

В C++11 существует множество генераторов случайных чисел, в т.ч. с равномерным распределением в заданном интервале

Пример Адаптера: Код Целевого Класса

- Решение
  - Объявляем интерфейс класса для генерации чисел в заданном диапазоне
    - Объявляем виртуальный метод getValue()
  - Создаем наследника с реализацией виртуального метода getValue()
- Реализация может адаптировать как интерфейсный метод, так и быть привязанной к одной выбранной реализации
  - Адаптер объекта VS адаптер класса

Адаптер: Решение

- Интерфейс класса

```
class Value100Generator
{
public:
    virtual float getValue() = 0;
};
```

Адаптер Класса

- Реализация адаптера

```
class Value100GeneratorAdapterClassBased:
    public Value100Generator,
    private ValueGeneratorUniform //Inherit implementation
{
public:
    //! Must return random value from range 1..100
    virtual float getValue()
    {
        return getNormalizedValue() * 100.0f;
    }
};
```

Адаптер Объекта

- Реализация адаптера

```
class Value100GeneratorAdapterObjectBased:
    public Value100Generator
{
public:
    Value100GeneratorAdapterObjectBased(ValueGenerator
*generator): m_generator(generator) {}
    //! Must return random value from range 1..100
    virtual float getValue()
    {
        return m_generator->getNormalizedValue() * 100.0f;
    }
private:
    const ValueGenerator *m_generator;
};
```

Принцип Разделения Интерфейсов

- «Жирные» интерфейсы
  - Состоят из множества несцепленных функций
  - Реализуют более 1 АТД
  - Перегруженные функциями интерфейсы приводят к жесткости, хрупкости и тд
- Рассмотрим класс Door

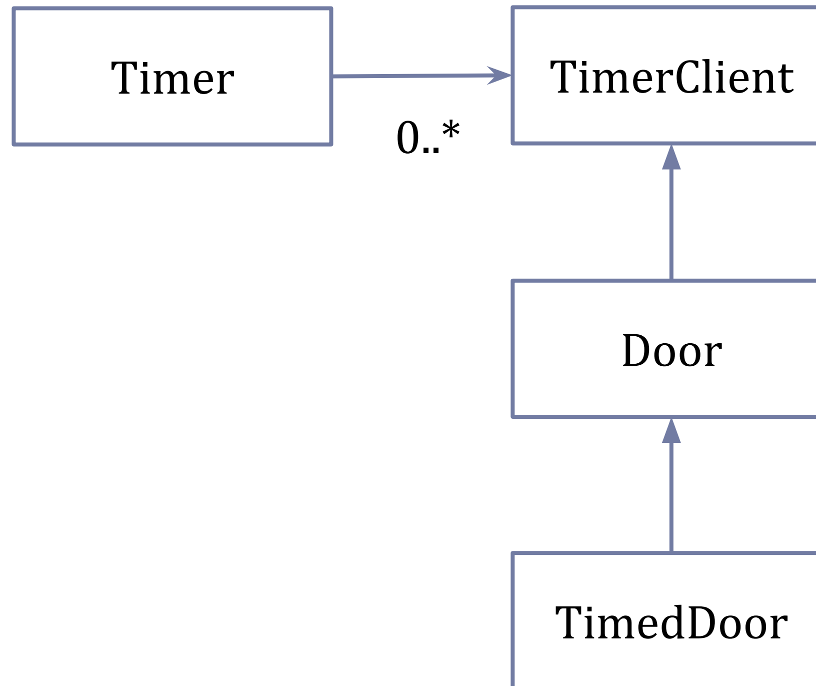
```
class Door
{
public:
    virtual void Lock() = 0;
    virtual void UnLock() = 0;
    virtual bool IsDoorOpen() = 0;
};
```

«Загрязнение» Интерфейса

- Новое требование
  - Новый тип дверей: вызывают сигнал тревоги, если слишком долго открыты
  - Класс TimedDoor
- Поддержка абстракции TimerClient
  - Класс, реагирующий на истечение времени таймера

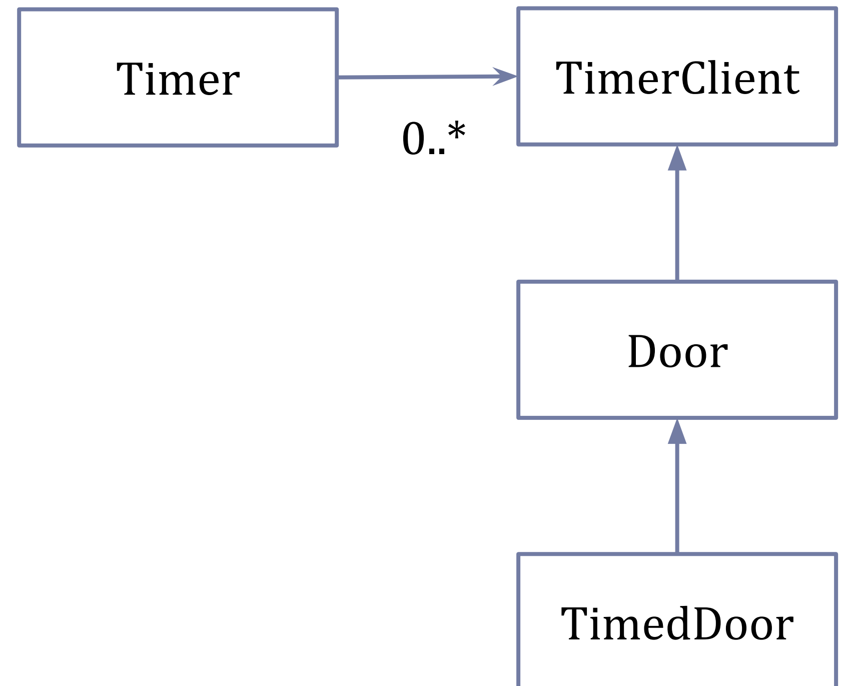
```
class TimerClient
{
public:
    virtual void TimeOut() = 0;
};

class Timer
{
public:
    void Register(int timeout, TimerClient *client);
};
```

Взаимодействие TimedDoor & Timer



Анализ

- Door теперь зависит от TimerClient
  - Изначальная абстракция Door не имела подобной зависимости
  - Реализации Door, не требующие отсчета времени, будут обязаны реализовать метод TimeOut()



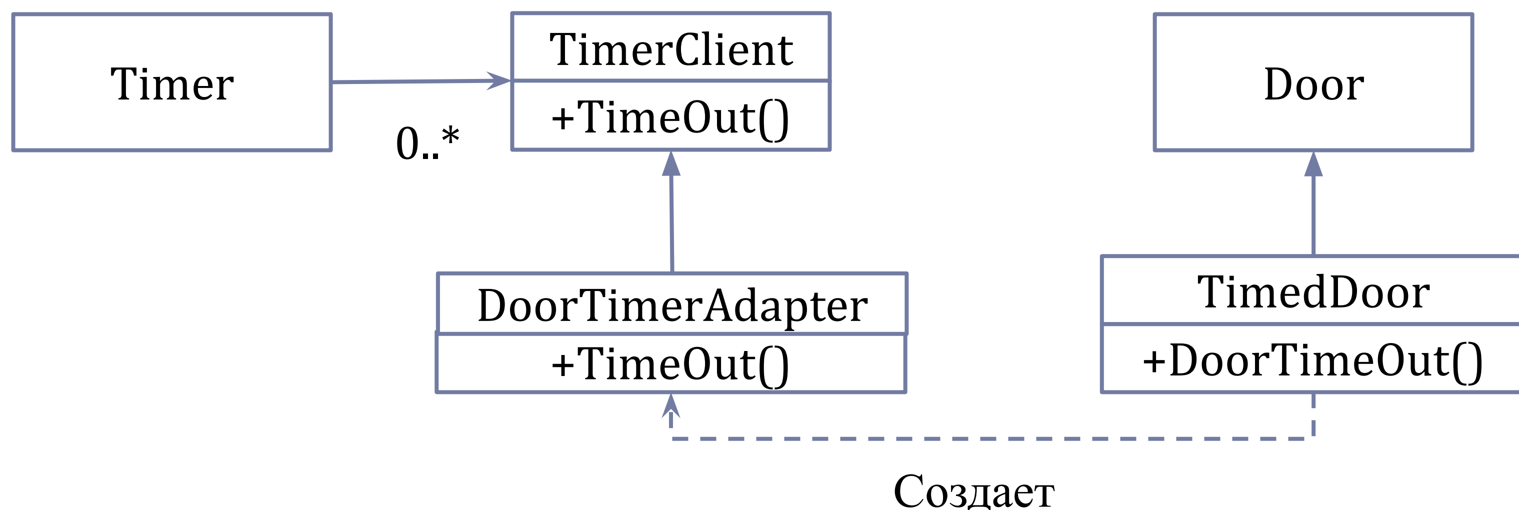
Жесткость и Вязкость Решения

- Новое требование – регистрация более одного запроса на истечение времени
- Любое изменение TimerClient повлечет изменения во всех объектах Door

```
class TimerClient
{
public:
    virtual void TimeOut(int timeOutId) = 0;
};

class Timer
{
public:
    void Register(int timeout, int timeOutId,
TimerClient *client);
};
```

Решение: Использование Адаптера

- Адаптер
  - Разделяет иерархии Door & TimerClient
  - «Транслирует» интерфейс TimerClient в TimedDoor



Решение: Использование Адаптера (2)

```
class TimedDoor : public Door
{
public:
    virtual void DoorTimeOut(int timeOutId);
};

class DoorTimerAdapter : public TimerClient
{
public:
    DoorTimerAdapter(TimedDoor &door) : m_door(&door) { }
    virtual void TimeOut(int timeOutId)
    {
        m_door->DoorTimeOut(timeOutId);
    }
private:
    TimedDoor *m_door;
};
```

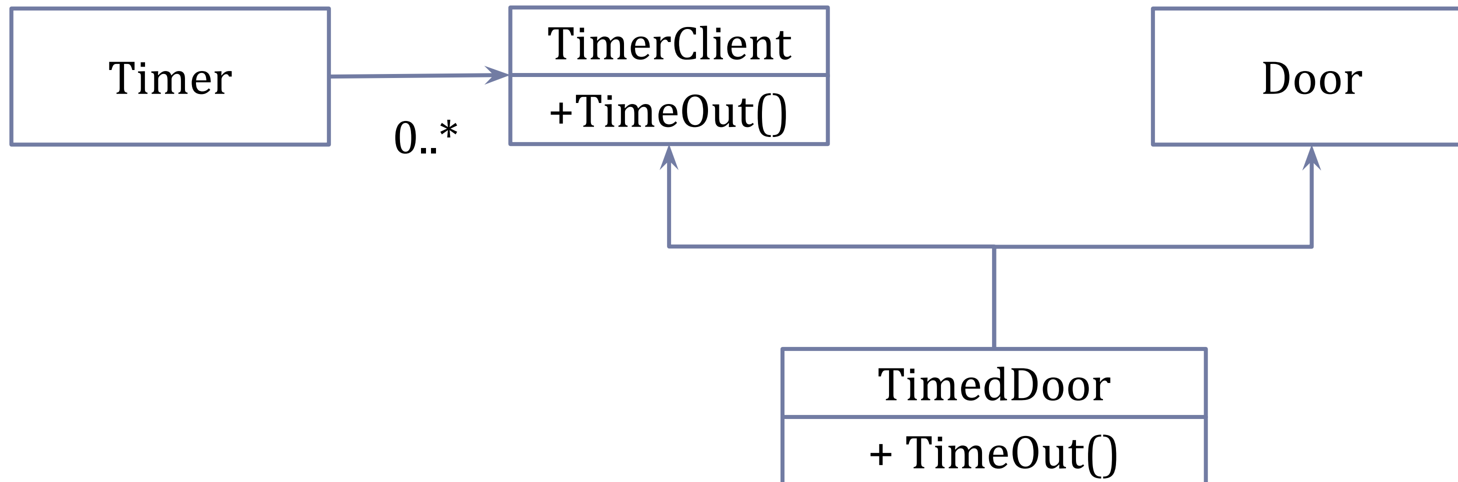
Анализ

- Каждый вызов регистрации запроса на таймер вынуждает создать объект-адаптер

```
DoorTimerAdapter doorAdapter(door);
timer->Register(timeOut, timeOutId,
&doorAdapter);
```

- Какое еще существует решение?

Решение: Множественное Наследование




```
class TimedDoor : public Door, public TimerClient
{
public:
    virtual void TimeOut(int timeOutId);
};
```

Заключение

- Принцип единственной обязанности
  - Каждый класс должен реализовывать лишь одну «ось изменения»
- Адаптер
  - Паттерн проектирования для улучшения коэф. повторного использования кода
  - Переадресовывает операции одного интерфейса в другой
- Принцип разделения интерфейсов
  - «Жирные» интерфейсы приводят к вязкости и жесткости
  - Интерфейсы могут быть разделены посредством адаптера или множественного наследования