ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция № 1 / **12**
13.05.2019 г.

ПРИНЦИП РАЗДЕЛЕНИЯ ИНТЕРФЕЙСА

- Interface Segregation Principle (ISP).

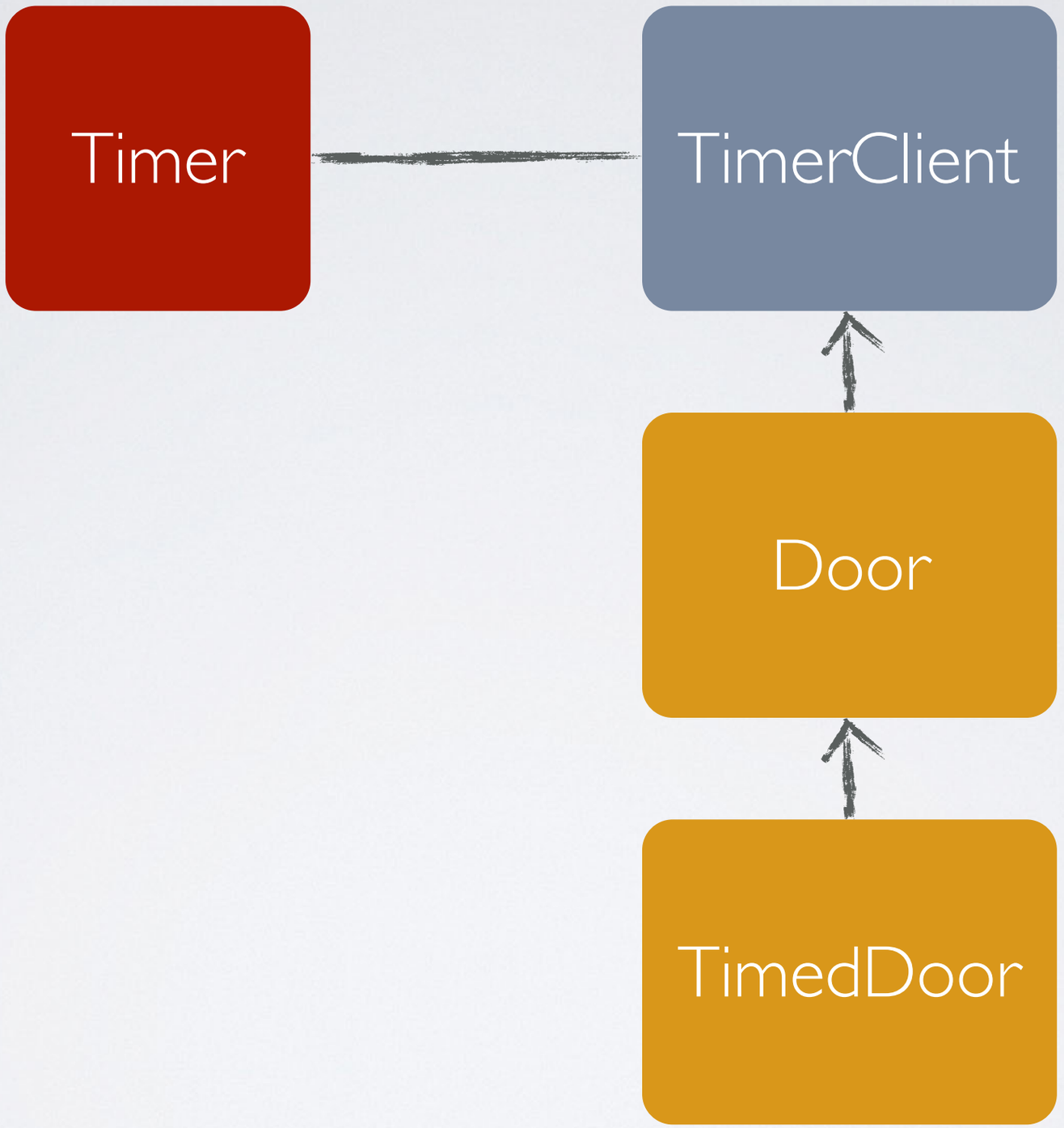
- *Клиенты не должны попадать в зависимость от методов, которыми они не пользуются.*

```
class Door {
public:
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual bool isOpen() = 0;
};
```

Как сделать класс **TimedDoor**?

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

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



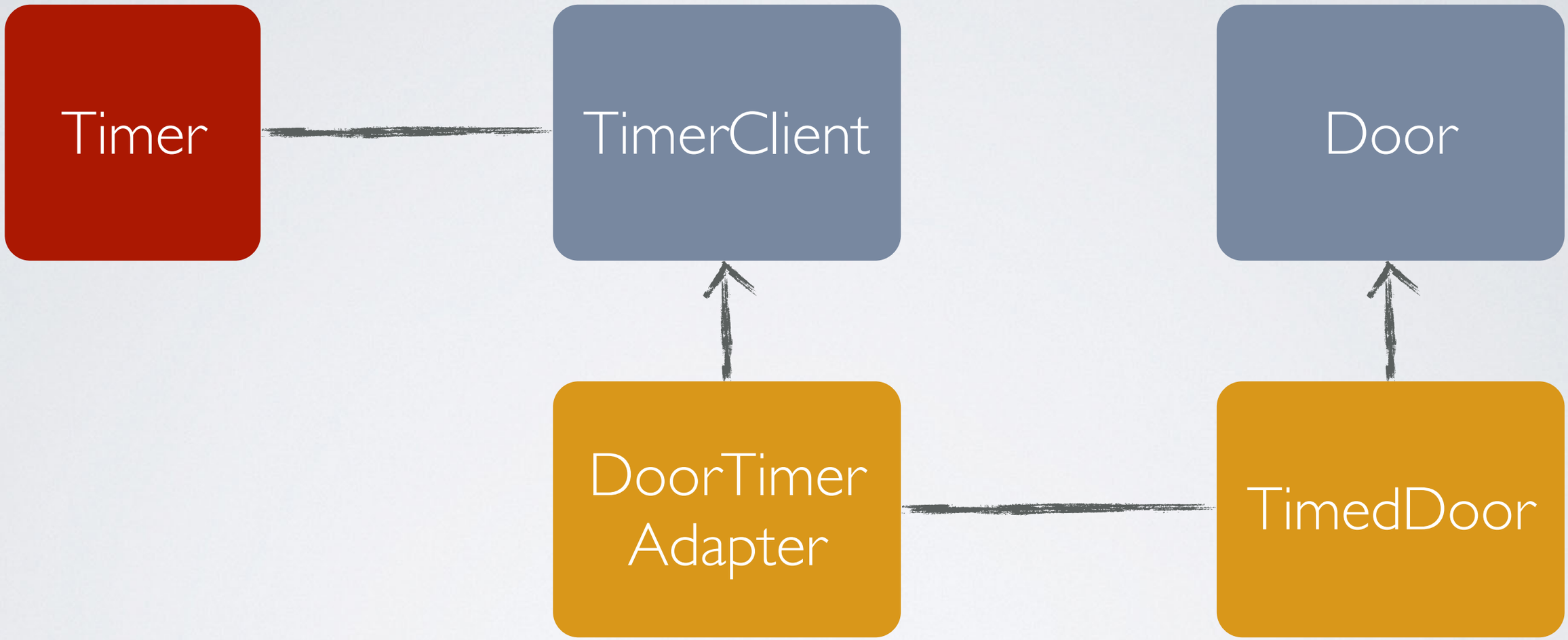
Вариант решения: наследуем Door от TimerClient

ПРОБЛЕМЫ

- **Door** не имеет никакого отношения к таймеру! (SRP)

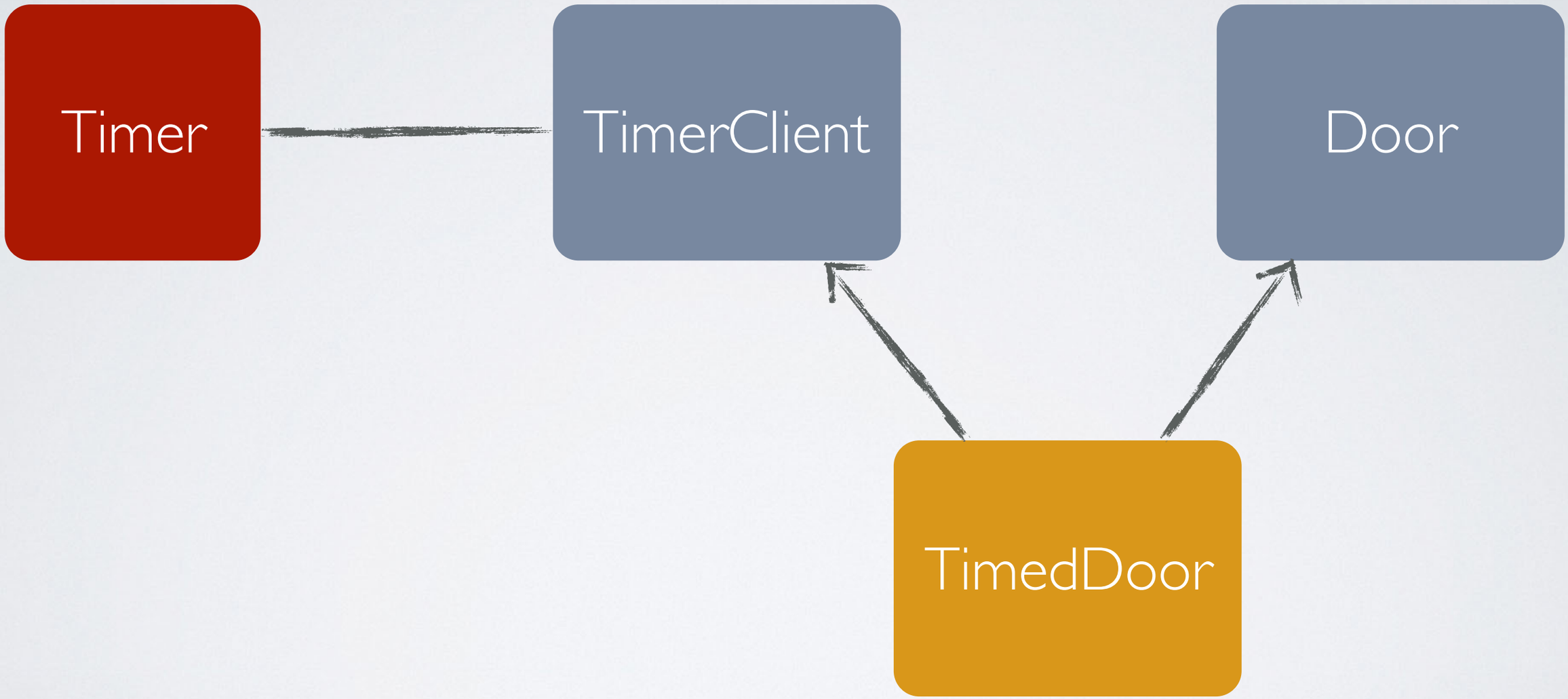
- Все пользователи **Door** должны тащить за собой **TimerClient** (и зависеть от изменений в нём).

- Возможное нарушение принципа LSP.



Вариант удачнее: посредством DoorTimeAdapter

Третий вариант: множественное наследование



Single **R**esponsibility **P**rinciple
Open-**C**losed **P**rinciple
Liskov **S**ubstitution **P**rinciple
Interface **S**egregation **P**rinciple
Dependency **I**nversion **P**rinciple

Solid – прочный, крепкий, надежный…

```
struct Base {};

struct Point : Base {
    double x, y;
};

struct Square : Base {
    Point top_left;
    double side;
};

struct Rectangle : Base {
    Point top_left;
    double height, width;
};

struct Circle : Base {
    Point center;
    double radius;
};
```

Структуры данных (POD — Plain Old Data)

```
#include <typeinfo>
#include <exception>

using namespace std;

namespace Geometry {
    const double PI = 3.141592653589793;

    class UnknownShape : public exception {};

    inline double area(Base *object) {
        if (typeid(*object) == typeid(Square)) {
            Square *s = static_cast<Square *>(object);
            return s->side * s->side;
        } else if (typeid(*object) == typeid(Rectangle)) {
            Rectangle *r = static_cast<Rectangle *>(object);
            return r->height * r->width;
        } else if (typeid(*object) == typeid(Circle)) {
            Circle *r = static_cast<Circle *>(object);
            return PI * c->radius * c->radius;
        }

        throw UnknownShape();
    }
};
```

Процедурный стиль, работающий с POD

```
class Shape {
public:
    virtual double area() const = 0;
};

class Square : public Shape {
    Point top_left;
    double side;
public:
    double area() const override {
        return side*side;
    }
};

// Аналогично Rectangle и Circle
```

ОО стиль

Процедурный стиль

- Легко добавить новые функции. Структуры данных не меняются.
- Сложно добавить новые структуры данных — придётся менять все функции.

ОО стиль

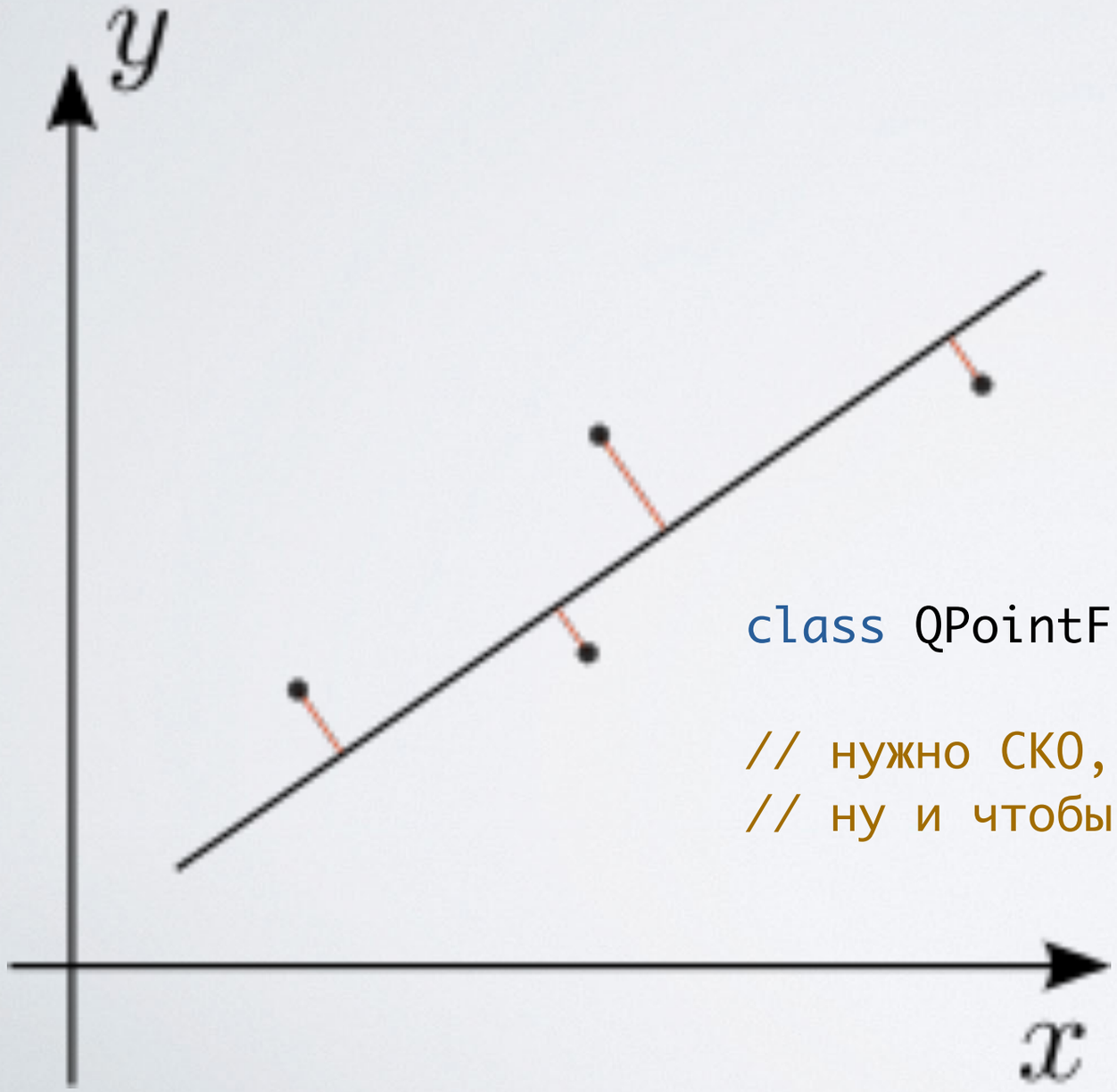
- Легко добавить новые классы. Существующие функции не меняются.
- Сложно добавить новые функции — придётся менять все классы.

1. Иногда процедурный код адекватен.

2. При необходимости процедурный код можно переделать в объектно-ориентированный.

ЗАДАЧА

Проектирование класса для линейной аппроксимации набора точек методом наименьших квадратов.




```
class QPointF;

// нужно СКО, дисперсия, коэффициенты A и B (y=Ax+B)
// ну и чтобы можно было удобно отрисовывать линию
```

- Входные данные — параметры конструктора *(во всех вариантах, чтобы было удобно)*.
- Выходные данные — публичные методы интерфейса *(во всех возможных вариантах, чтобы было удобно)*.
- «Бедный» интерфейс — предоставляет минимум данных (результат вычислений) одним способом (обычно совпадающим со способом хранения).
- «Богатый» интерфейс — идёт на шаг дальше и ориентирован на решение микрозадач с результатами.

```
class LinearRegression {
public:
    LinearRegression(const QVector<QPointF> &_points);

    // y=Ax+B
    double a();
    double b();

    double mse();
    double std();

protected:
    // ...
};
```

Минимум интерфейса

```
class LinearRegression {
public:
    LinearRegression(const QPointF *_points, int _size);
    LinearRegression(const QVector<QPointF> &_points);

    // y=Ax+B
    double a();
    double b();

    double y(double x)       { return a() * x + b(); }
    double x(double y)       { return (y - b()) / a(); }
    QPointF point(double x) { return QPointF(x, y(x)); }

    double mse();
    double std();

protected:
    // ...
};
```

Стало побогаче!

ГДЕ ВЫЧИСЛЯТЬ?

- Сразу в конструкторе («энергичный» подход).
- В отдельном методе (`run()`, `calculate()` и т.п.)
- При первом вызове метода интерфейса («ленивый» подход).

```
class LinearRegression {
public:
    // ...
    double a();
    // ...

protected:
    void calculate();
    /****/
    const QPointF *points;
    int size;
    bool ready;
    double _a, _b;
};
```

```
double LinearRegression::a() {
    if (!ready) {
        calculate();
    }
    return _a;
}

void LinearRegression::calculate() {
    // ...
    _a = ...;
    _b = ...;
    ready = true;
}
```

Реализация ленивого подхода

ЗАДАЧА’

Проектирование класса
для ~~линейной~~ экспоненциальной аппроксимации
набора точек.

```
class BaseRegression {
public:
    BaseRegression(const QPointF *_points, int _size);
    BaseRegression(const QVector<QPointF> &_points);

    virtual double y(double x)=0;

    QPointF point(double x) {
        return QPointF(x, y(x));
    }

    double mse();
    double std();

protected:
    /*****/
    const QPointF *points;
    int size;
};
```

```
class ExponentialRegression : public BaseRegression {
public:
    ExponentialRegression(const QVector<QPointF> &points);

    // y=A*exp(B*x)
    double a();
    double b();

    double y(double x) { return a()*exp(b()*x); }

protected:
    void calculate();
    /****/
    bool ready;
    double _a, _b;
};
```

```
void ExponentialRegression::calculate()
{
    QVector<QPointF> logpoints(size);

    for (int i = 0; i < size; ++i)
        logpoints[i] = QPointF(points[i].x(),
                               log(fabs(points[i].y())));

    LinearRegression lr(logpoints);

    _b = lr.a();
    _a = exp(lr.b());

    if (size > 0 && points[0].y() < 0)
        _a *= -1;

    ready = true;
}
```

КОНЕЦ ДВЕНАДЦАТОЙ ЛЕКЦИИ