

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



Лекция № 1 / 09
10.04.2017 г.

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

- Наследование
- Композиция
- Агрегация
- Ассоциация

```

namespace Expression {
    class Node {
    public:
        virtual double evaluate() const = 0;
    };

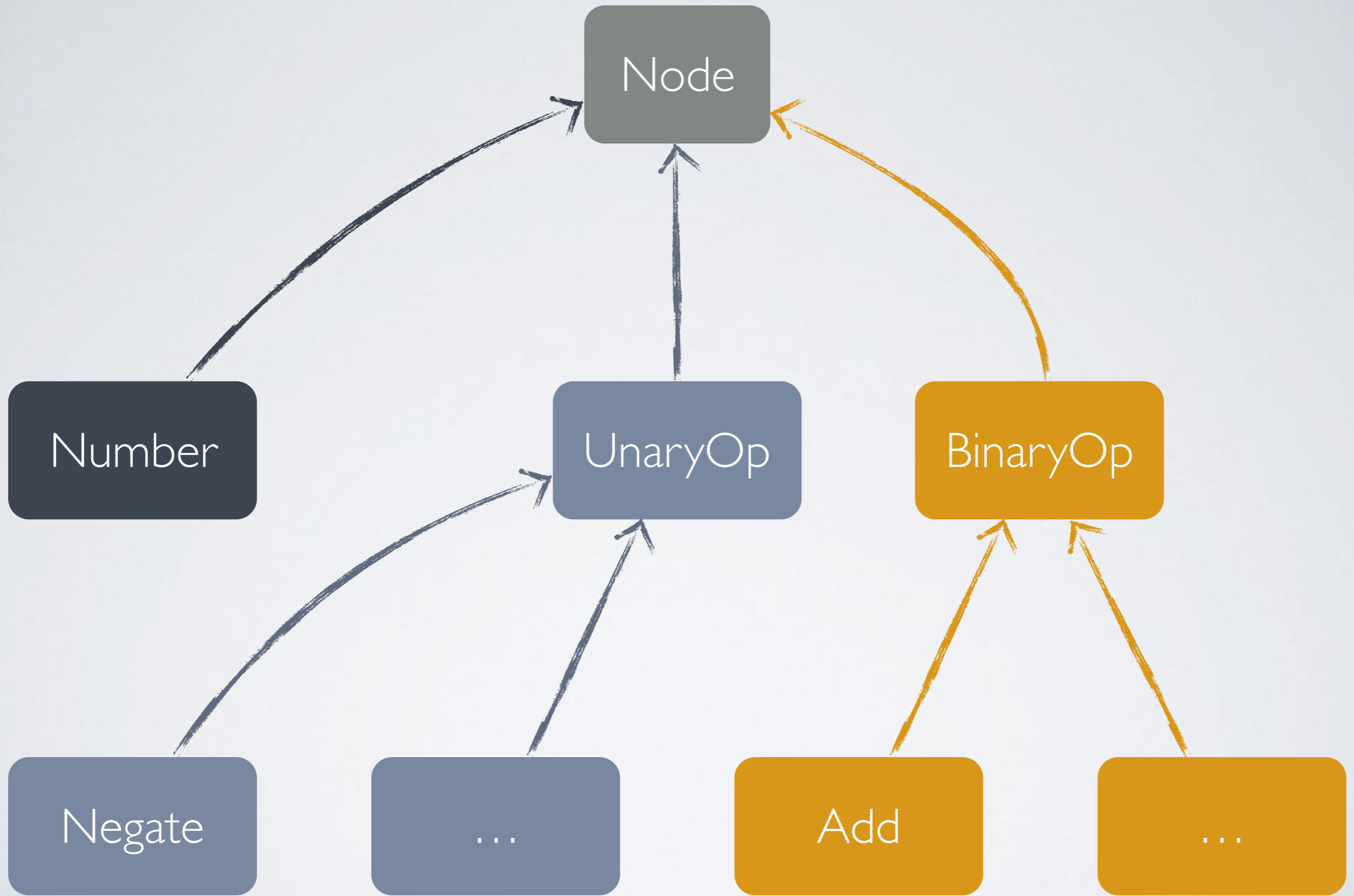
    class Number: public Node {
        double number;
    public:
        virtual double evaluate() const { return number; }
    };

    class UnaryOp: public Node { Node *arg; /* ... */ };
    class BinaryOp: public Node { Node *left, *right; /* ... */ };

    class Negate: public UnaryOp { /* ... */ };
    class Add: public BinaryOp { /* ... */ };
}

```

Наследование



ЧЕМ ХОРОШО НАСЛЕДОВАНИЕ?

- Расширяемость. Полиморфизм. Имея указатель на объект базового класса, код может работать с любым его наследником.
- Отсутствие дублирования. В идеале:

*Новый функционал = Старый функционал +
Переопределение некоторых методов*

ПРОБЛЕМНЫЕ ТОЧКИ НАСЛЕДОВАНИЯ

- Нельзя отказаться от «наследства». Нельзя нарушать «контракт» базового класса.
- Неудачный интерфейс базового класса делает переопределение неудобным или невозможным.
- В большой и развесистой иерархии сложно разобраться.
- Трудности с наследованием сразу от нескольких классов из одной иерархии.

```
class Text {
public:
    virtual void draw() = 0;
};
```

```
class BoldText:           public Text { /* ... */ };
class ItalicText:        public Text { /* ... */ };
class UnderlinedText:   public Text { /* ... */ };
class StrikeText:       public Text { /* ... */ };
class BorderedText:     public Text { /* ... */ };
class BigText:           public Text { /* ... */ };
class SmallText:        public Text { /* ... */ };
```

Иллюстрация трудностей с наследованием

```
class BoldItalicUnderlinedBorderedBigText; // WTF?
```



```
class InternalFrameInternalFrameTitlePaneInternalFrameTitlePaneMaximizeButtonWindowNotFocusedState extends State {  
    // ...  
};
```

Это не шутка 😞



КОМПОЗИЦИЯ

- Членами одного класса (*хозяина*) являются объекты других классов (*рабы?*).
- Класс-хозяин управляет порождением и уничтожением этих объектов.

```
class GuiRectangle {  
public:  
    // нарисовать прямоугольник  
    void draw();  
  
private:  
    Rectangle rect;  
};
```

Уже знакомый пример

Попытка наследования

```
class Object {
public:
    virtual void update() {}
    virtual void draw() {}
    virtual void collide(vector<Object *> const &) {}
};
```

```
class Visible : public Object {
public:
    virtual void draw() { /* нарисовать модель на месте объекта */ };
private:
    Model *model;
};
```

```
class Solid : public Object {
public:
    virtual void collide
    (vector<Object *> const &)
    { /* обработка столкновений */ }
};
```

```
class Movable : public Object {
public:
    virtual void update()
    { /* изменение положения */ };
};
```

- **Player:** Visible, Solid, Movable.
- **Cloud:** Movable, Visible, но не Solid.
- **Building:** Solid, Visible, но не Movable.
- **Trap:** Solid, но не Visible и не Movable.

КОМПОЗИЦИЯ

```
class Object {
    VisibilityDelegate *_v;
    UpdateDelegate     *_u;
    CollisionDelegate  *_c;
public:
    Object(VisibilityDelegate *v, UpdateDelegate *u, CollisionDelegate *c)
        : _v(v), _u(u), _c(c) {}

    void update() { _u->update(); }
    void draw()   { _v->draw(); }
    void collide(vector<Object *> const &objects) {
        _c->collide(objects);
    }
};
```

```
class VisibilityDelegate {
public:
    virtual void draw() = 0;
};
```

```
class Invisible : public VisibilityDelegate {
public:
    virtual void draw() {}
};
```

```
class Visible: public VisibilityDelegate {
public:
    virtual void draw() { /* отрисовка модели */ }
};
```

```
class CollisionDelegate {
public:
    virtual void collide(vector<Object *> const &) = 0;
};

class Solid : public CollisionDelegate {
public:
    virtual void collide(vector<Object *> const &) { /* ... */ }
};

class NotSolid : public CollisionDelegate {
public:
    virtual void collide(vector<Object *> const &) {}
};

class UpdateDelegate {
public:
    virtual void update() = 0;
};

class Movable : public UpdateDelegate {
public:
    virtual void update() { /* перемещаем объект */ };
};

class NotMovable : public UpdateDelegate {
public:
    virtual void update() {}
};
```

```
class Player : public Object {  
public:  
    Player() : Object(new Visible(), new Movable(), new Solid()) {}  
    // ...  
};
```

«Набираем» объект из делегатов

ПРЕИМУЩЕСТВА КОМПОЗИЦИИ

- Одна большая иерархия заменяется на несколько меньших, не связанных между собой иерархий.
Более простая система!
- Не нужно тянуть всё из базового класса. Объект-делегат вообще можно создавать при необходимости.
- Интерфейс класса-хозяина независим.

ПРОБЛЕМНЫЕ ТОЧКИ КОМПОЗИЦИИ

- Объект-хозяин не входит в иерархию классов делегатов — полиморфизм не работает.
- Нужно явно добавлять методы, вызывающие методы делегатов.

- Наследование: **A** является **B** (*A is a B*).
- Композиция: **A** содержит **B** (*A has a B*).
- **Apple** и **Fruit**. Яблоко является фруктом.
Наследование уместно.
- **Employee** и **Person**. Вроде бы сотрудник является человеком. Но в действительности сотрудник — это должность, замещаемая человеком (*который может вообще потерять работу или совмещать несколько должностей*). Лучше использовать композицию.

АГРЕГАЦИЯ

- Аналог композиции, но без *владения*.

```

class Professor;

class Department {
    // ...
private:
    // Агрегация
    std::vector<Professor *> members;
    // ...
};

class University {
    // ...
private:
    // Композиция
    std::vector<Department> faculty;
    // ...

    void create_dept() {
        // ...
        faculty.push_back(Department(/* ... */));
        faculty.push_back(Department(/* ... */));
        // ...
    }
};

```

// Если университет (University) уничтожается, то факультеты (Department) тоже
// должны быть уничтожены. Но профессора (Professor) останутся! Кроме того,
// один профессор может работать на нескольких факультетах, но факультет не может
// принадлежать нескольким университетам.

В ЧЁМ РАЗНИЦА МЕЖДУ

`vector<Object>`

и

`vector<Object *>` ?

- Композиция
- Хранятся сами объекты
- Никакого полиморфизма
- Агрегация
- Объекты хранятся где-то ещё (где?)
- Возможен полиморфизм по **Object**.

АССОЦИАЦИЯ

- Самая свободная форма связи между классами.
- Один класс содержит указатель или ссылку на объект другого класса и вызывает его методы.
- Порождение и уничтожение объекта происходит где-то ввне.

```
class Lamp {
public:
    void on();
    void off();
};

class ToggleButton {
    Lamp lamp;
    bool is_on;
public:
    ToggleButton() : is_on(false) {}

    void toggle() {
        is_on = !is_on;

        if (is_on)
            lamp.on();
        else
            lamp.off();
    }
};
```

Композиция (бессмысленная притом)

```

class Switchable {
public:
    virtual void on() {}
    virtual void off() {}
};

class Lamp: public Switchable { /* ... */ };

class ToggleButton {
    Switchable *object;
    bool is_on;
public:
    ToggleButton(Switchable *o)
        : object(o), is_on(false) {}

    void toggle() {
        is_on = !is_on;

        if (is_on)
            object->on();
        else
            object->off();
    }
};

```

Ассоциация


```
class MultiSwitch : public Switchable {
    vector<Switchable *> objects;
public:
    MultiSwitch(const vector<Switchable *> &objs)
        : objects(objs) {}

    void on() {
        for (auto object: objects)
            object->on();
    }

    void off() {
        for (auto object: objects)
            object->off();
    }
};
```

Вот что можно делать, когда нет жёстких связей

СИЛА СВЯЗЕЙ МЕЖДУ ОБЪЕКТАМИ

Композиция

Агрегация

Ассоциация

Обычно чем слабее, тем лучше!

ПРИНЦИП ОТКРЫТИЯ-ЗАКРЫТИЯ

(OPEN-CLOSED PRINCIPLE)

Программные объекты:
(классы, модули, методы, ...)



Открыты для
расширения

НО ...

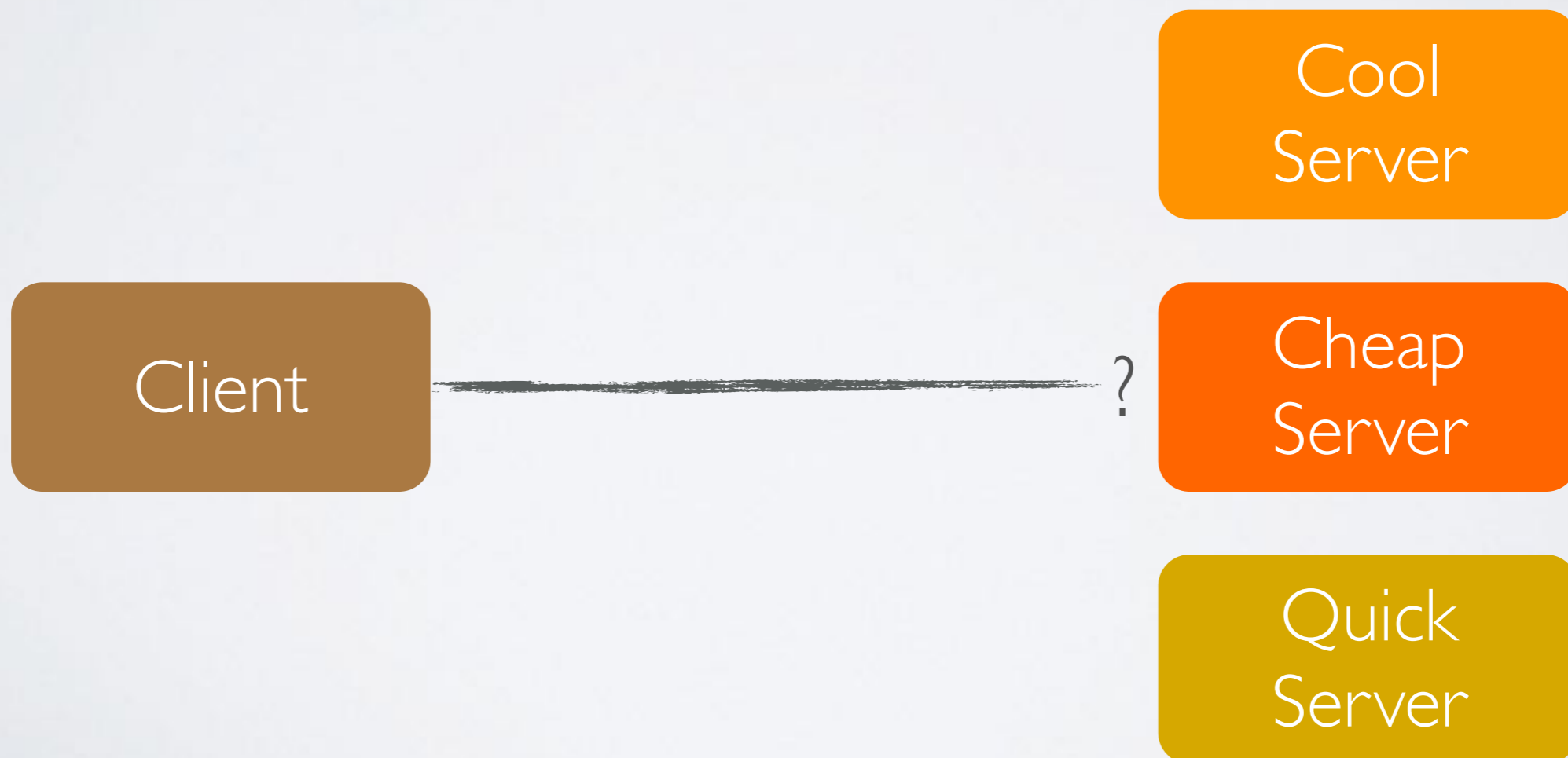


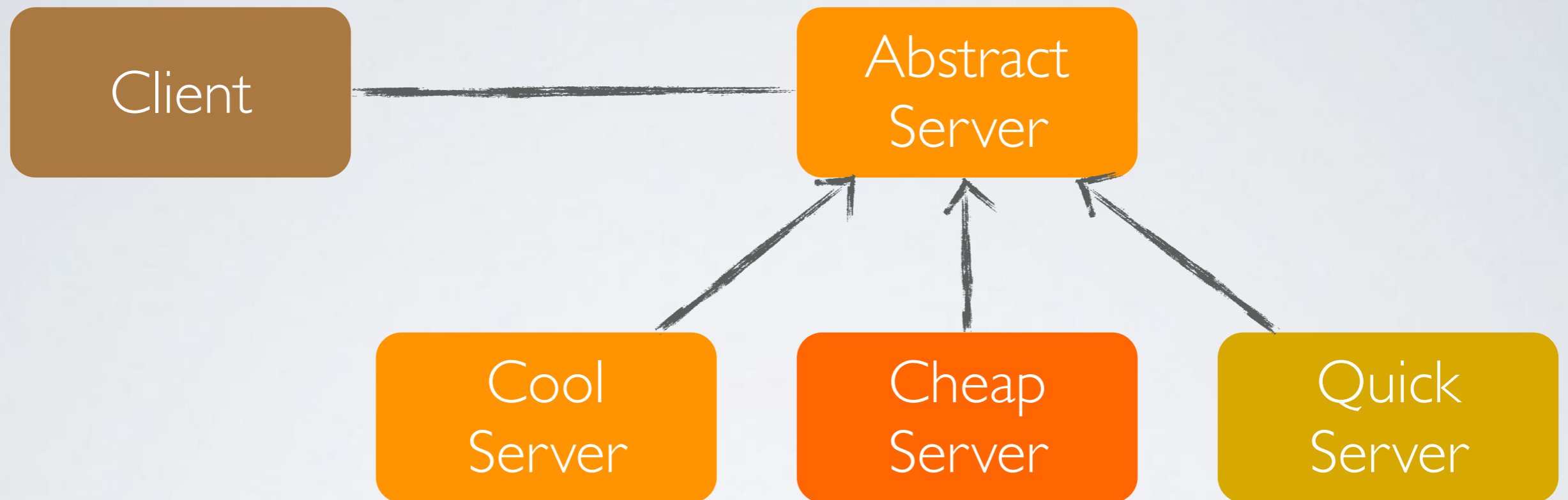
Закреты
для модификации

БЫЛО:



Теперь хочется так:





Конечно же, выносим абстракцию в базовый класс!

Мега-задача

Есть набор объектов (круги и квадраты),
нужно уметь отрисовать этот набор

```
/* shape.h */
enum ShapeType { CIRCLE, SQUARE };

struct Shape {
    ShapeType type;
};

/* circle.h */
struct Circle {
    ShapeType type;
    double radius;
    Point center;
};

void drawCircle(struct Circle *);

/* square.h */
struct Square {
    ShapeType type;
    double side;
    Point topLeft;
};

void drawSquare(struct Square *);
```

Процедурный style

```
/* drawallshapes.c */
typedef struct Shape *ShapePointer;

void drawAllShapes(ShapePointer *list, int n) {
    int i;
    for (i = 0; i < n; ++i) {
        ShapePointer *s = list[i];
        switch (s->type) {
            case SQUARE: drawSquare((struct Square *)s); break;
            case CIRCLE: drawCircle((struct Circle *)s); break;
        }
    }
}
```



```
class Shape {  
public:  
    virtual void draw() const = 0;  
};
```

```
class Square : public Shape {  
public:  
    // ...  
    virtual void draw() const;  
};
```

```
class Circle : public Shape {  
public:  
    // ...  
    virtual void draw() const;  
};
```

```
void drawAllShapes(Shape **list, int n) {  
    for (int i = 0; i < n; ++i)  
        list[i]->draw();  
}
```

OO style

Закрытость для расширения

```
switch (s->type) {  
    case SQUARE: drawSquare((struct Square *)s); break;  
    case CIRCLE: drawCircle((struct Circle *)s); break;  
}
```



Открытость для расширения

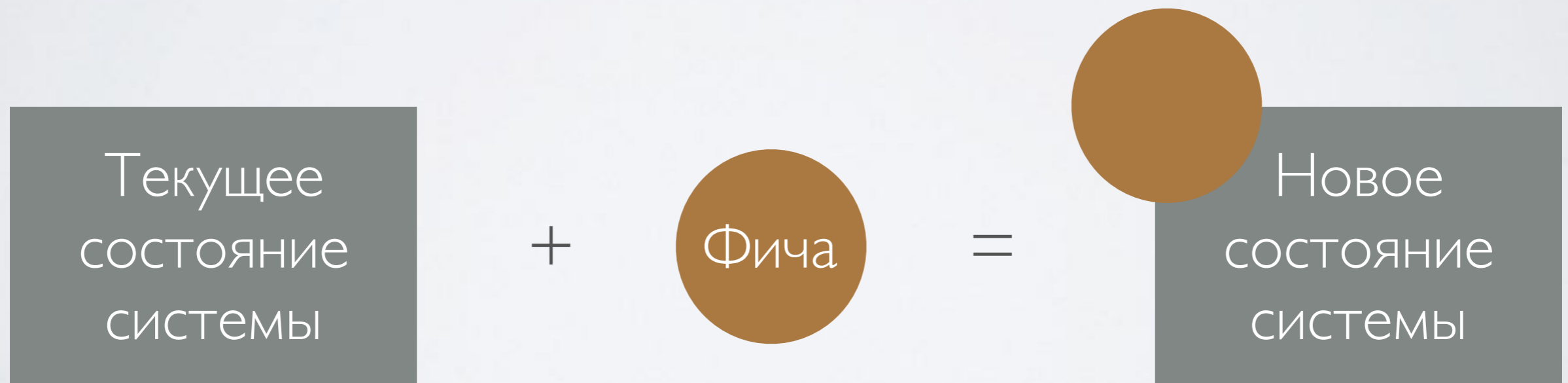
```
class MyNewLovelyShape : public Shape {  
public:  
    // ...  
    virtual void draw() const;  
};
```

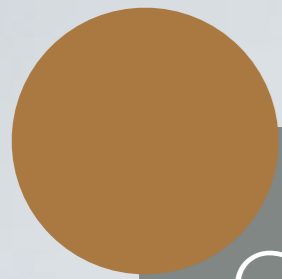
Усложнение

Сначала рисовать все
окружности, а **затем** все квадраты

КАК ВОООБЩЕ МОЖНО ВНОСИТЬ ИЗМЕНЕНИЯ?

Способ первый





Состояние
системы



ИНОЙ СПОСОБ

Дано:

Текущее
состояние
системы

и

Фича

Шаг первый

Фича

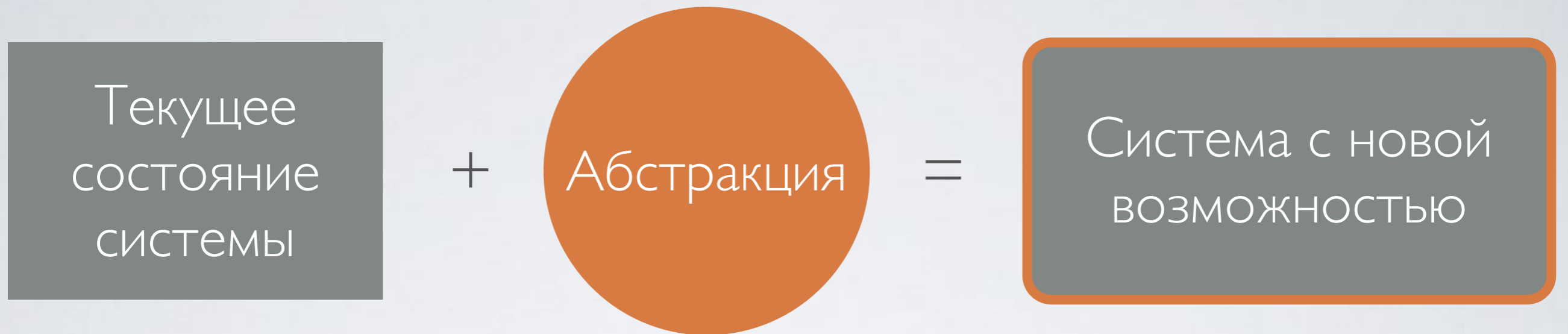
расщепляется на

Абстракция

Фича'



Шаг второй



Шаг третий



Какая абстракция соответствует тому,
чтобы рисовать сначала окружности,
а потом квадраты?


```
class Shape {  
public:  
    virtual void draw() const = 0;  
    virtual bool precedes(const Shape &another) const = 0;  
};
```

Задание порядка

КОНЕЦ ДЕВЯТОЙ ЛЕКЦИИ

