ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

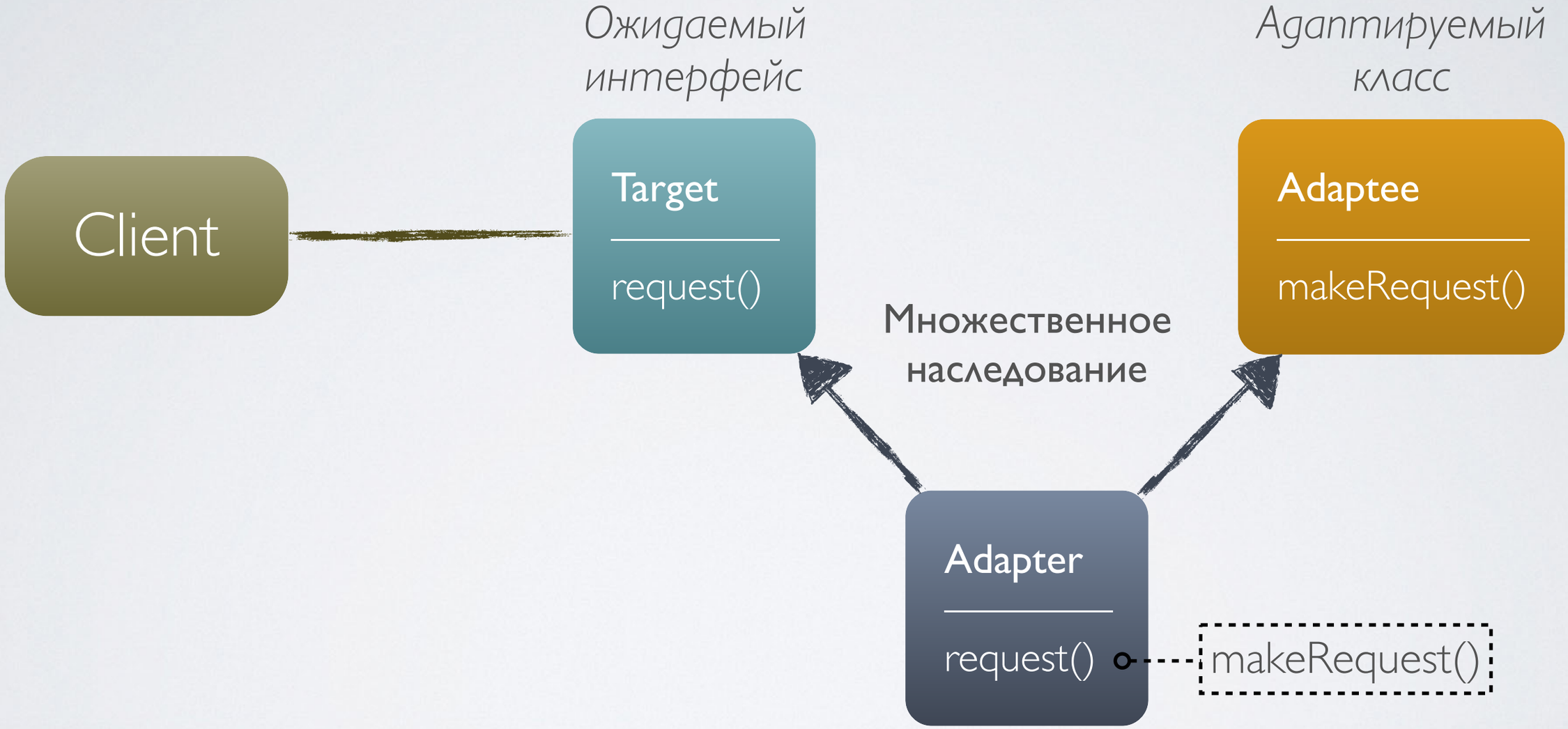
СТРУКТУРНЫЕ ПАТТЕРНЫ

ADAPTER

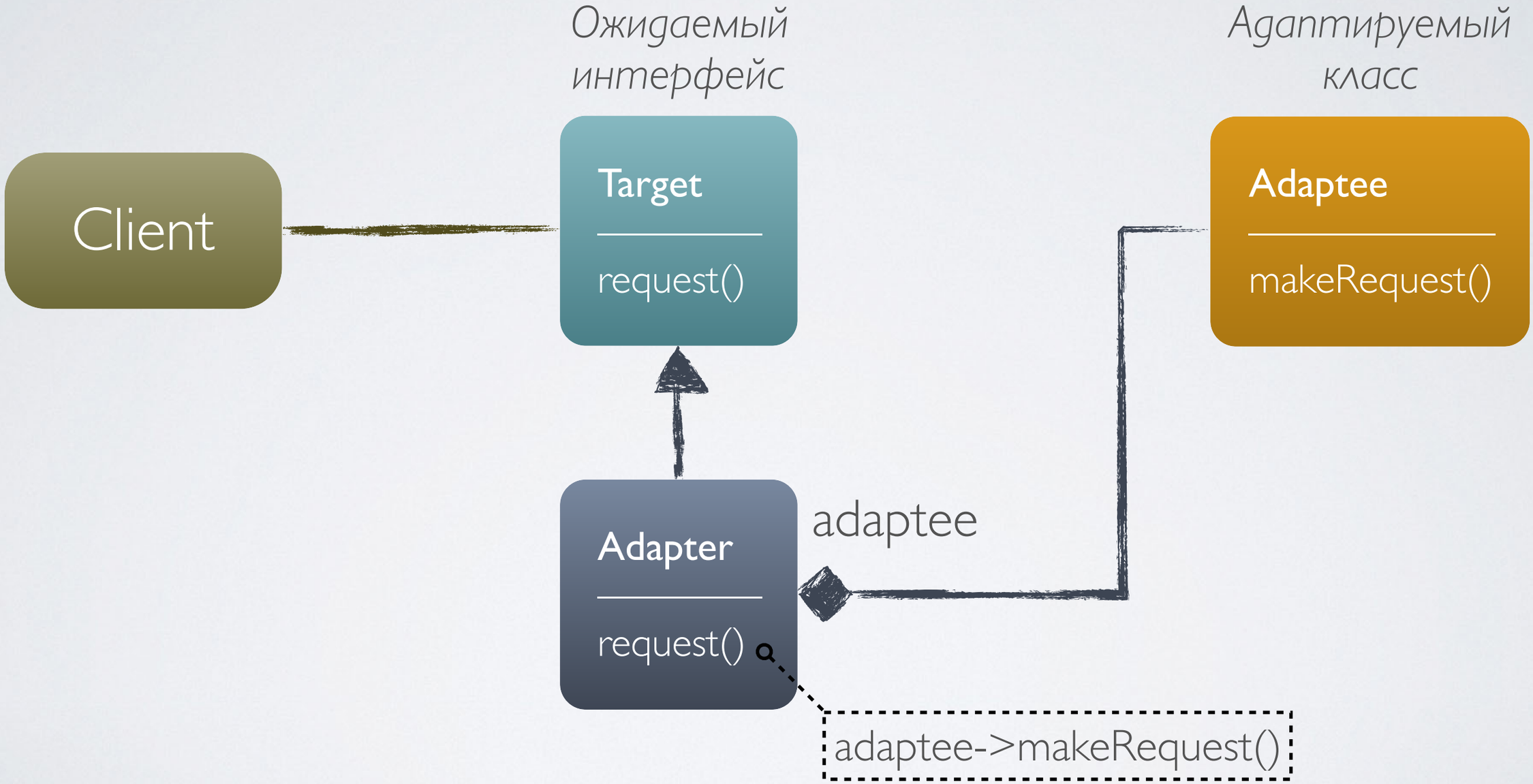
- *Адаптер* преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты.

- Другое название — Wrapper (обёртка).

АДАПТЕР КЛАССА



АДАПТЕР ОБЪЕКТОВ



```
class Circle {
public:
    virtual void draw();
    virtual double area();

    // ...
};

void reallyCoolThing(Circle *);
void amazingStuff(Circle *);
```

```
struct OldAndRustyCircle {
    void draw_me();
    void get_area_and_circumference(
        double *result,
        double *circum);
};
```

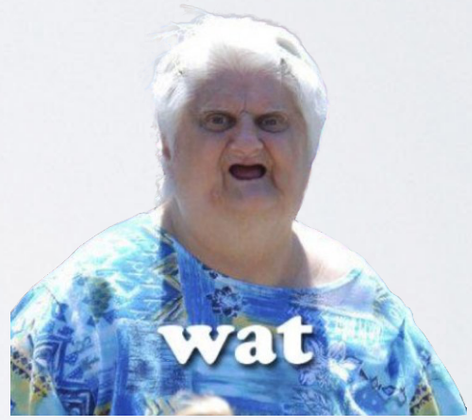
```
class NewAndShinyCircle : public Circle {
public:
    void draw() { old_circle.draw_me(); }
    // ....

private:
    OldAndRustyCircle old_circle;
};
```

```
struct Person {
    // ...
};

struct GetIQ {
    GetIQ(const Person *p);
    int get_iq();
};
```

Адаптер для класса GetIQ

```
struct IQCompare {
    bool operator()(const Person &p1, const Person &p2) {
        return GetIQ(&p1).get_iq() <
               GetIQ(&p2).get_iq();
    }
};

vector<Person> people;

sort(people.begin(), people.end(), IQCompare());
```

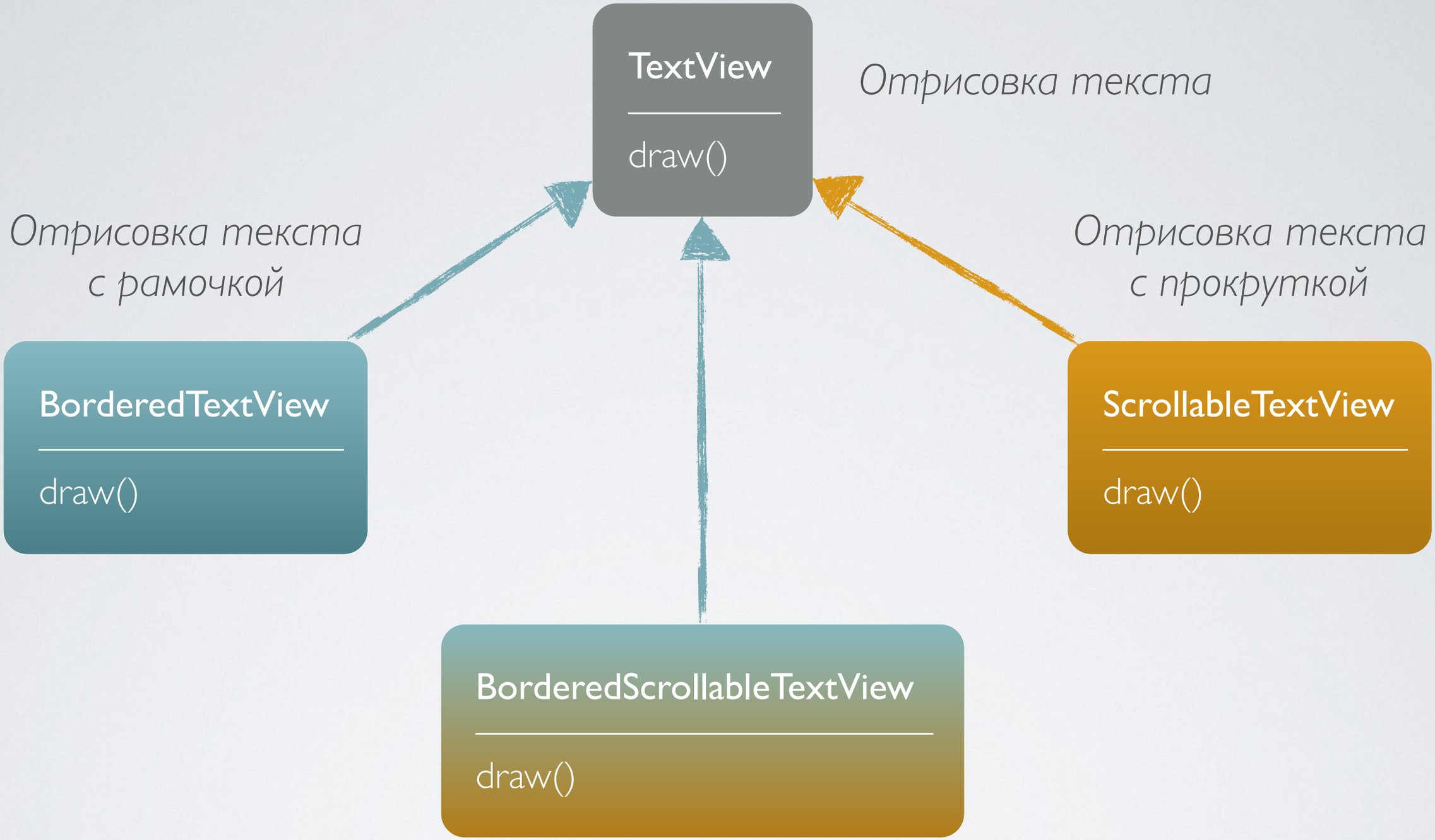
- *Адаптер класса:* можно переопределить какие-то методы **Adaptee**.
- *Адаптер объектов*: можно приспособить целую иерархию классов, наследованных от **Adaptee**.
- Двусторонние адаптеры класса.



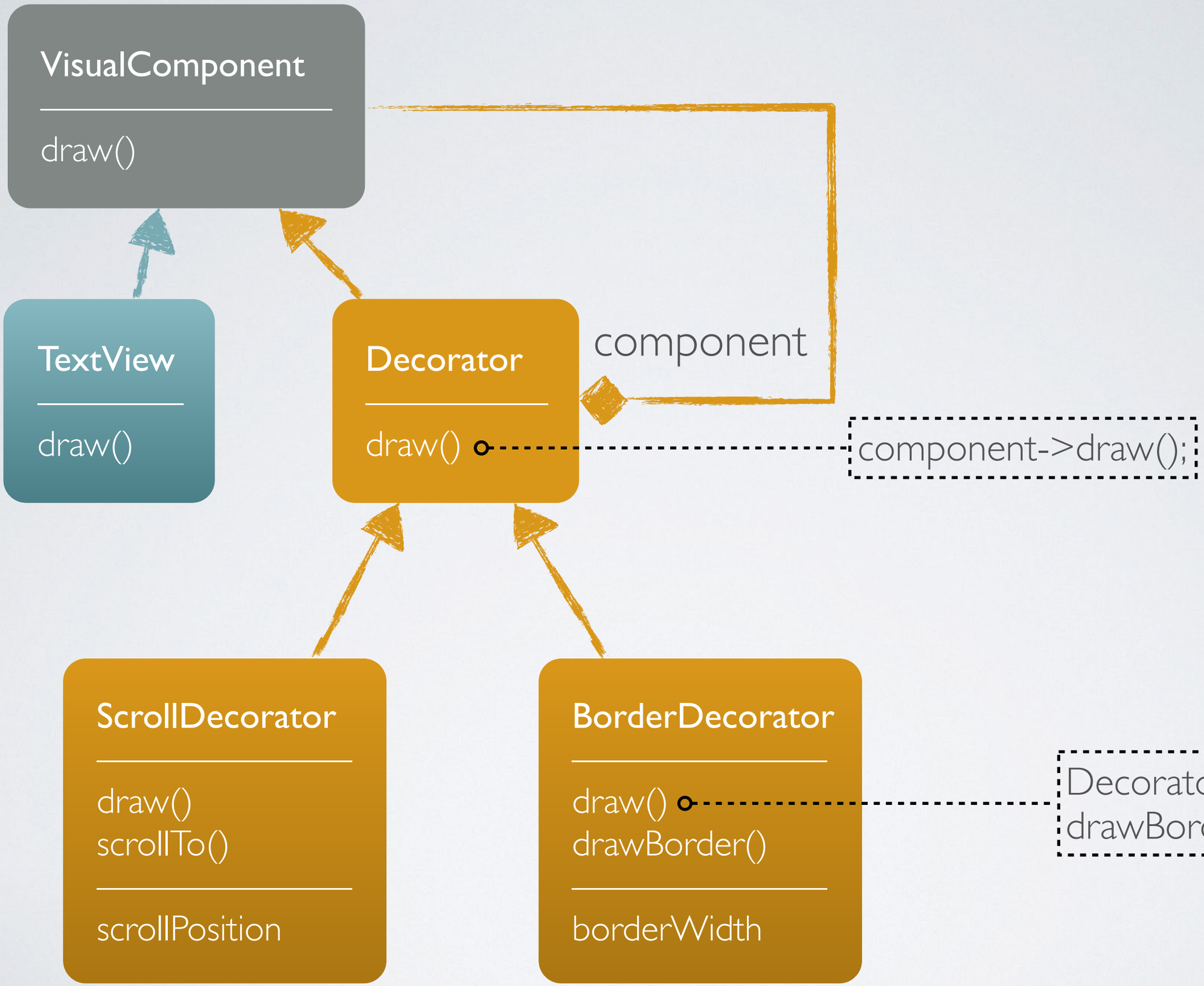
DECORATOR

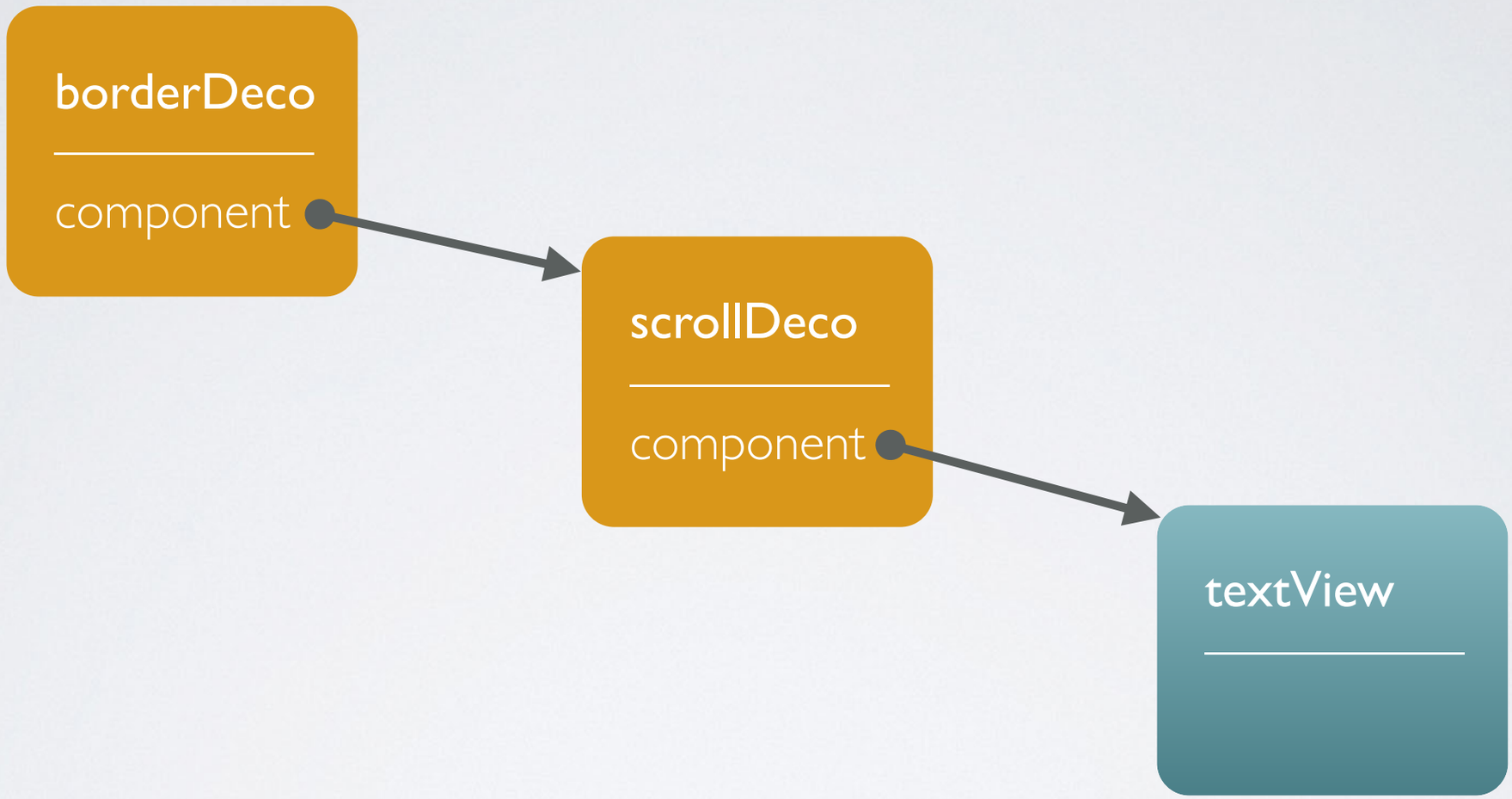
- *Декоратор* динамически добавляет объекту новые обязанности.

- Альтернатива порождению подклассов.


TextView
draw()
Отрисовка текста
Отрисовка текста
с рамочкой
Отрисовка текста
с прокруткой
BorderedTextView
draw()
ScrollableTextView
draw()
BorderedScrollableTextView
draw()
Отрисовка текста
с рамочкой и прокруткой


BORDEREDSCROLLABLEMOVABLECOPYABLEMENUABLETEXTVIEW
NAILED IT
memegenerator.net


VisualComponent
draw()
TextView
draw()
Decorator
draw()
component
component->draw();
ScrollDecorator
draw()
scrollTo()
scrollPosition
BorderDecorator
draw()
drawBorder()
borderWidth
Decorator::draw()
drawBorder()


borderDeco
component
scrollDeco
component
textView

```
struct VisualComponent {
    virtual void draw() = 0;
};

class TextView : public VisualComponent {
    // ....
};

class Decorator : public VisualComponent {
    VisualComponent *component;
public:
    void draw() { component->draw(); }
};

class BorderDecorator : public Decorator {
    int borderWidth;
    void drawBorder();
public:
    void draw() {
        component->draw();
        drawBorder();
    }
};

class ScrollDecorator : public Decorator {
    int scrollPosition;
public:
    void draw();
    void scrollTo();
};
```

- Динамическое добавление/удаление обязанностей.
- Декораторы прозрачны для компонента.
- Альтернатива порождению подклассов, когда это неудобно *(комбинаторный взрыв)* или невозможно *(код закрыт)*.
- Альтернатива перегруженному функциями базовому классу.
- Можно добавить свойство дважды (двойная рамка — два **BorderDecorator** в цепочке).
- Хорошо бы сделать базовый класс (**VisualComponent**) лёгким.
- Декораторы дают много мелких классов, в которых можно запутаться.
- Альтернатива декоратору — шаблон *Стратегия* (создание отдельного класса **Border**, отвечающего за отрисовку рамки).

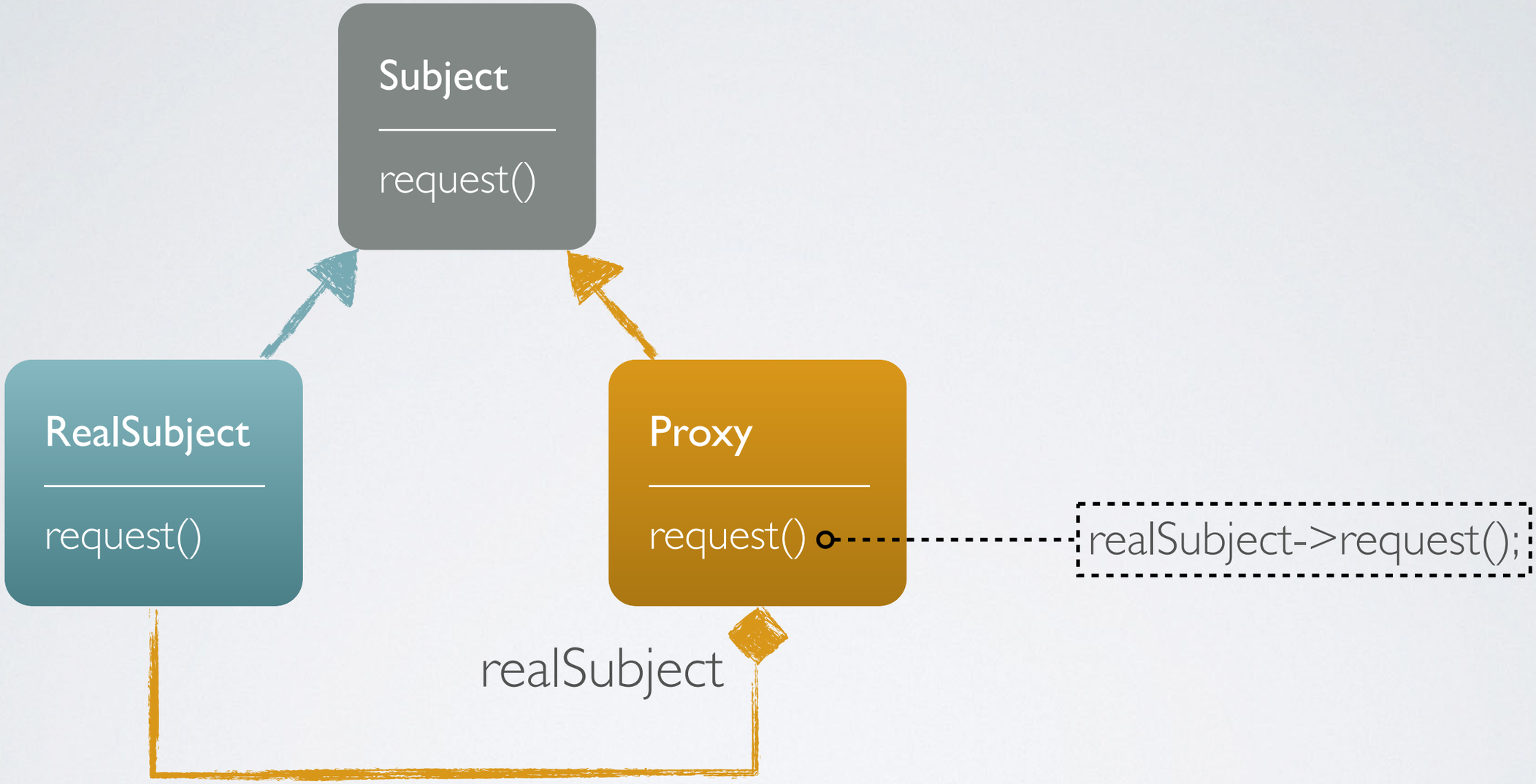
PROXY

- *Заместитель* является суррогатом другого объекта и контролирует доступ к нему.
- Полезен, когда в системе есть «тяжёлые» объекты.

```
class Graphic {
public:
    virtual void draw();
    virtual Rect getExtent() const;

    // ...
};

class Image : public Graphic {
    Image(const string &fileName) {
        // load image
    }

    // ...
};
```

Проблема: есть 100 изображений по 5 мегабайт, а нужен только размер – getExtent()

```
class ImageProxy : public Graphic {
    std::string fileName;
    Rect extent;
    bool extentLoaded;
public:
    ImageProxy(const string &fn)
        : fileName(fn), extentLoaded(false) {}

    void draw() {
        if (!image)
            image = loadImage(fileName);
        image->draw();
    }

    Rect getExtent() const {
        if (image)
            return image->getExtent();
        else {
            if (!extentLoaded) {
                extent = loadExtentFromImageHeader();
                extentLoaded = true;
            }

            return extent;
        }
    }

    Rect loadExtentFromHeader();
};
```


Subject
request()
RealSubject
request()
Proxy
request()
realSubject->request();
realSubject

- *Удалённый заместитель* — когда сам объект находится в другом адресном пространстве.
- *Виртуальный заместитель* откладывает создание «тяжёлых» объектов или использует кэширование.
- *Защищающий заместитель* контролирует доступ к исходному объекту.
- Smart Pointer – тоже заместитель!
- В C++ можно переопределить **operator->** и **operator*** для контролируемого, но прозрачного доступа к исходному объекту.

Proxy

Не изменяя интерфейс, управляет доступом к объекту.

Decorator

Добавляет новое поведение

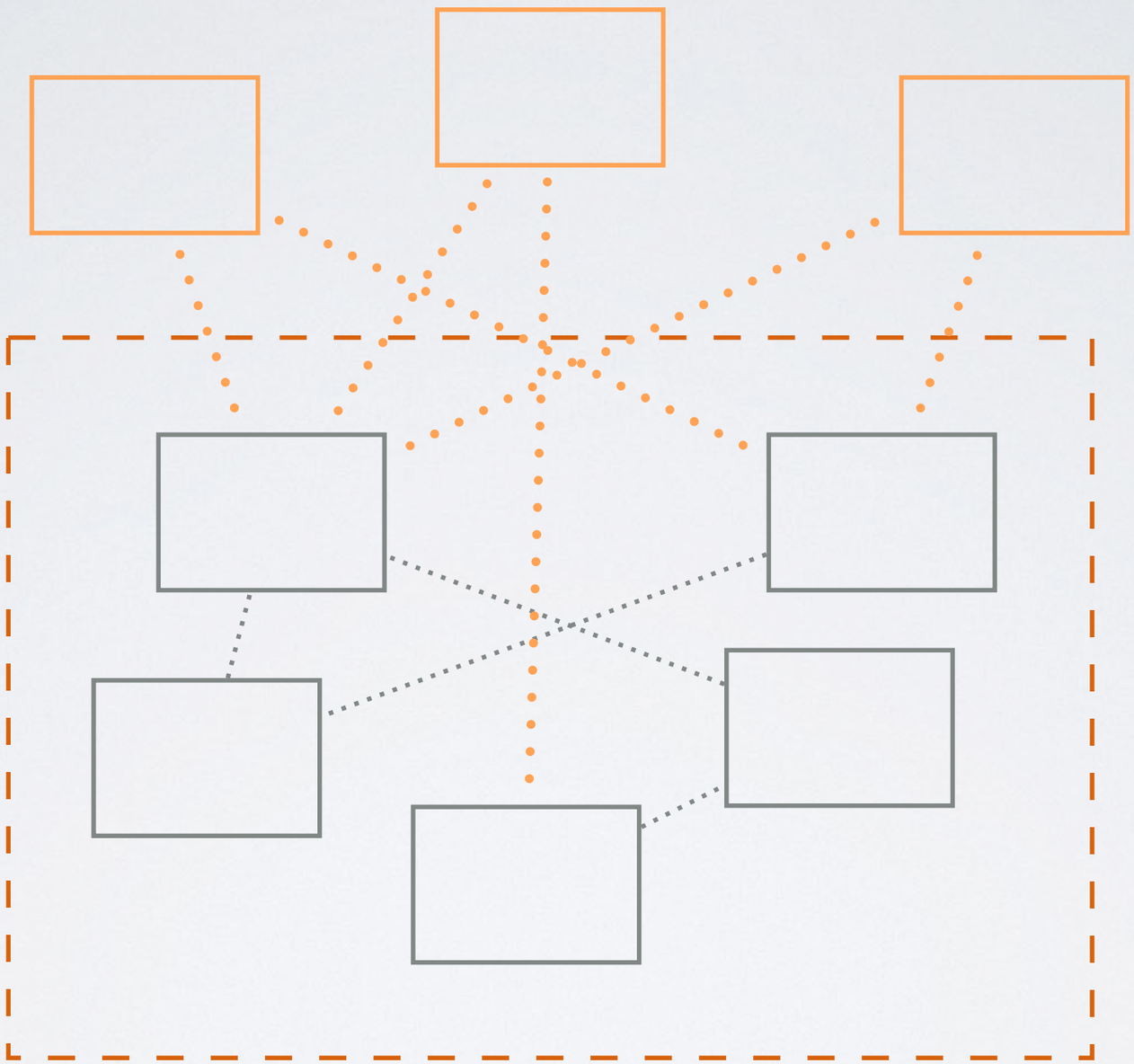
Adapter

Изменяет интерфейс адаптируемых объектов

FACADE

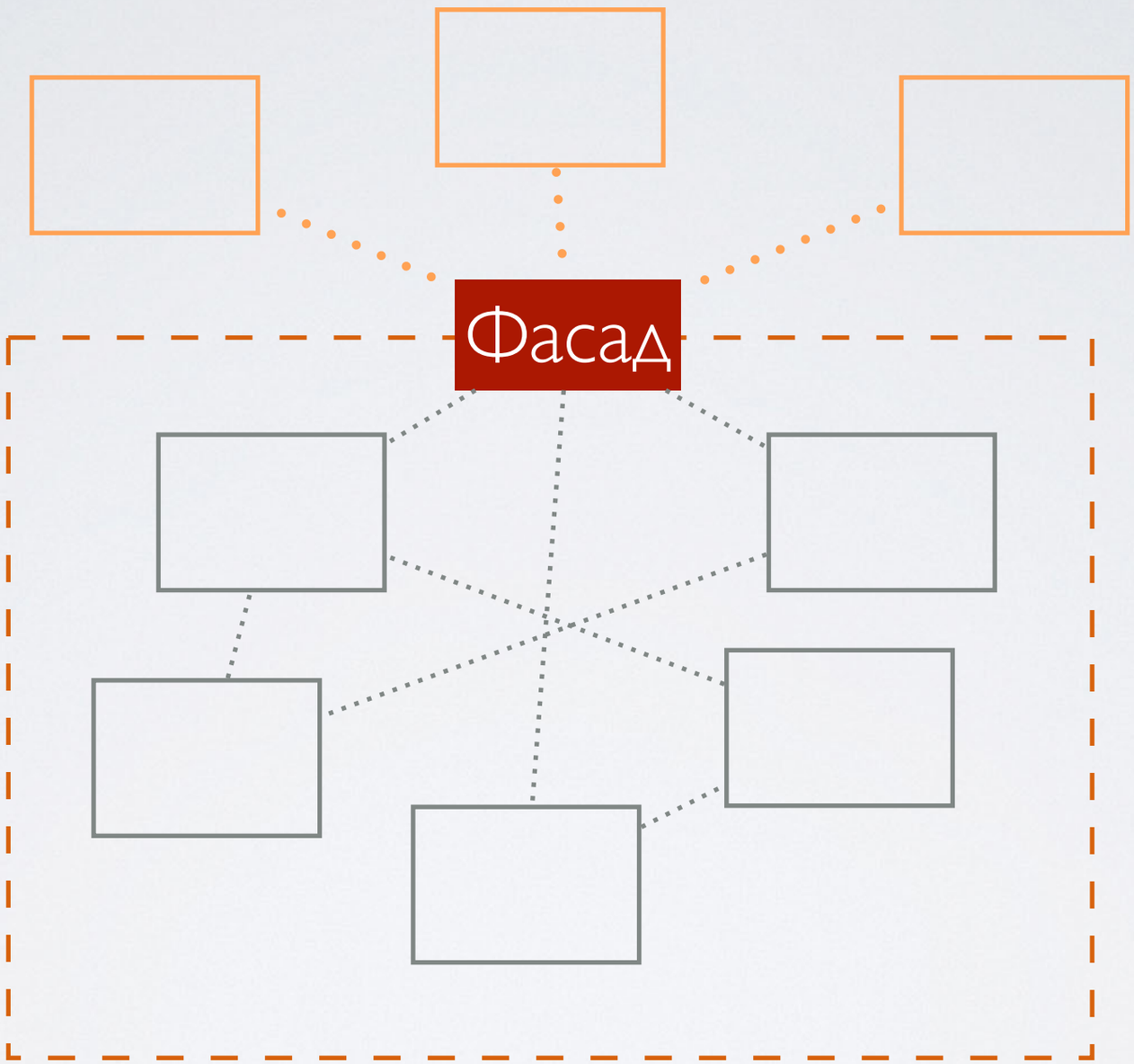
- *Фасад* предоставляет унифицированный интерфейс (более высокого уровня) вместо набора интерфейсов подсистемы, упрощая её использование.

Классы клиента



Классы подсистемы

Классы клиента

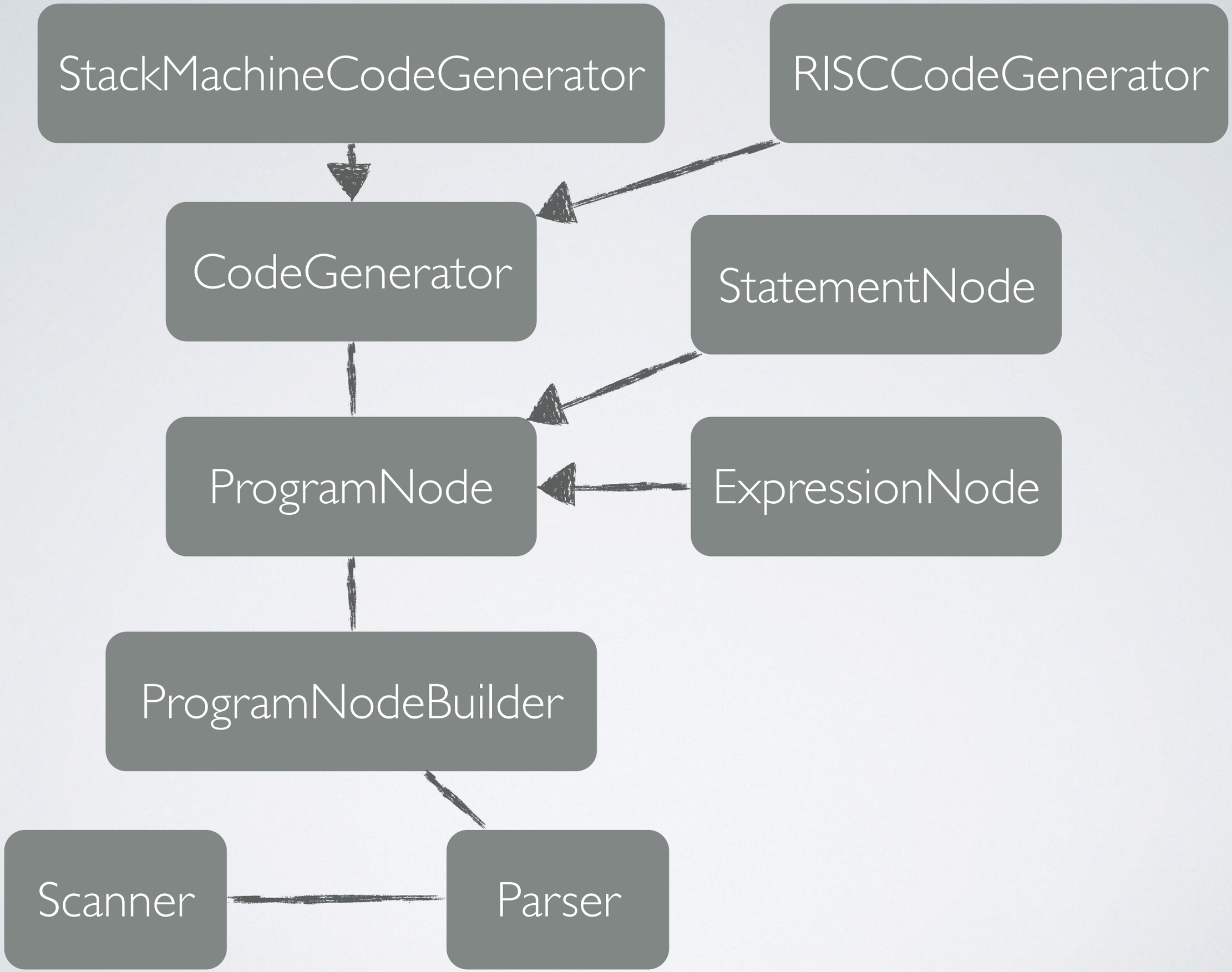


Классы подсистемы

```
class Scanner {
public:
    Scanner(istream &);
    virtual ~Scanner();

    virtual Token &scan();
    // ...
};

class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void parse(Scanner &, ProgramNodeBuilder &);
};

class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    virtual void ProgramNode *newVariable(const char *name) const;
    virtual void ProgramNode *newAssignment(ProgramNode *var, ProgramNode *expr) const;
    virtual void ProgramNode *newReturnStatement(ProgramNode *value) const;
    virtual void ProgramNode *newCondition(ProgramNode *cond, ProgramNode *truePart,
                                           ProgramNode *falsePart) const;

    ProgramNode *rootNode();

    // ...
};
```

```
class ProgramNode {
public:
    virtual void getSourcePosition(int &line, int &col);
    virtual void traverse(CodeGenerator &);

    // ...
};
```

```
class CodeGenerator {
public:
    virtual void visit(StatementNode *);
    virtual void visit(ExpressionNode *);

    CodeGenerator(BytecodeStream &);
};
```


StackMachineCodeGenerator
RISCCodeGenerator
CodeGenerator
StatementNode
ProgramNode
ExpressionNode
ProgramNodeBuilder
Scanner
Parser

```
class Compiler {
public:
    Compiler();

    virtual void compile(istream &input, BytecodeStream &output) {
        Scanner scanner(input);
        ProgramNodeBuilder builder;
        Parser parser;

        parser.parse(scanner, builder);

        RISCCodeGenerator generator(output);
        ProgramNode *parseTree = builder.rootNode();
        parseTree->traverse(generator);
    }
};
```

- Фасад предоставляет простой, задачно-ориентированный интерфейс к подсистеме.

- Объект-фасад сам создаёт все необходимые объекты подсистемы и устанавливает связи между ними.

- Классы подсистемы отделяются от клиентов и от других подсистем. Связанность ослабляется.

- Остаётся возможность низкоуровневого доступа к классам подсистемы (когда это действительно необходимо).

COMPOSITE

- *Компоновщик* компонует объекты в древовидные структуры для представления иерархий часть-целое.
- Группа объектов трактуется так же, как и один объект.

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

class Rectangle : public GraphicsObject {
    // ...
};

class Ellipse : public GraphicsObject {
    // ...
};

class CompositeObject : public GraphicsObject {
    std::vector<GraphicsObject *> objects;
public:
    void add(GraphicsObject *obj);

    void draw() {
        std::vector<GraphicsObject *>::iterator q;

        for (q = objects.begin(); q != objects.end(); ++q)
            q->draw();
    }

    // ....
};
```

Client

Component

operation()
add(Component)
remove(Component)
getChild(int)

Leaf

operation()

Composite

operation()
add(Component)
remove(Component)
getChild(int)

потомки

вызвать g->operation()
для всех потомков g

- Единая работа как с составными, так и одиночными объектами.

- Базовый класс должен поддерживать все операции для составных объектов (пусть и в виде заглушек по умолчанию).

БОЛЬШЕ ПАТТЕРНОВ

OBJECT POOL (ПУЛ ОБЪЕКТОВ)

- Вместо вызова **new** и **delete** берём объект из пула. Когда не нужен — возвращаем.

- **+**: не теряется время на конструирование и уничтожение.

- **–**: если пула недостаточно?

- **–**: невозвращение приводит к быстрому исчерпанию пула.

NULL OBJECT (ОБЪЕКТ-ЗАГЛУШКА)

- Полная реализация какого-нибудь интерфейса, которая ничего не делает (все методы с пустыми телами).

- Вместо **nullptr** можно передавать указатель на объект такого класса — избавившись тем самым от **nullptr**!

SERVANT (СЛУГА)

- Разновидность паттерна *Command*.
- Некоторое поведение из иерархии классов выносится в отдельный класс *(слугу)*. Слуга оперирует объектами, которые ему дали, «своего» у него ничего нет.
- Лёгкий и недорогой способ соблюдать *Single Responsibility Principle*.

```cpp
struct Point {
    int x = 0;
    int y = 0;
};

class Movable {
public:
    virtual void setPosition(const Point &p) = 0;
    virtual Point position() const = 0;
};

class Triangle : public Movable {
    int A, B, C;
    Point pos;
public:
    // ...
    void setPosition(const Point &p) override { pos = p; }
    Point position() const override { return pos; }
};

struct Mover {
    static void moveTo(Movable &obj, const Point &newPos) {
        Point prevPos = obj.getPosition();

        std::cout << "Moving from " << prevPos << " to " << newPos << std::endl;

        obj.setPosition(newPos);
    }

    static void moveBy(Movable &obj, int dx, int dy) {
        Point prevPos = obj.getPosition();
        Point newPos { prevPos.x + dx, prevPos.y + dy };

        moveTo(obj, newPos);
    }
};
```

MVC (МОДЕЛЬ-ПРЕДСТАВЛЕНИЕ-КОНТРОЛЛЕР)

- **Модель:** оперирует данными в логике задачи. Ничего не знает о способах их представления и о взаимодействии с пользователем.

- **Представление:** отображает данные модели, не оперируя с ними и не взаимодействуя с пользователем.

- **Контроллер:** получает запросы пользователя и посылает их модели и представлению.


Модель
обновляет
манипулирует
Представление
Контроллер
видит
Пользователь
использует