ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

- Нет смысла решать схожие задачи каждый раз заново.

- Ограниченное количество «сюжетов» порождает безграничное количество «историй».

- *Узнать* шаблон.

ОПИСАНИЕ ПАТТЕРНОВ

Паттерн состоит из 4 основных элементов:

1. *Имя.* Представляет собой уникальное смысловое имя, однозначно определяющее данную задачу или проблему и ее решение.

2. *Задача.* Описание того, когда следует применять паттерн.

3. *Решение.* Описание элементов дизайна, отношений между ними, функций каждого элемента.

4. *Результаты.* Следствия применения паттерна и разного рода компромиссы.

«Описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте».

— Что такое паттерн проектирования?

ВИДЫ ПАТТЕРНОВ

- **Порождающие.** Создание объектов.
- **Структурные.** Композиция объектов и классов.
- **Поведенческие.** Взаимодействие объектов и классов.

SINGLETON

- Паттерн проектирования *Одиночка* гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

- Примеры:
  - Логирование действий (log). Один-единственный журнал.
  - Одна-единственная оконная система.
  - Один-единственный системный таймер.
  - …

РЕАЛИЗАЦИЯ ПАТТЕРНА SINGLETON

```
// Классическая реализация
class S {
public:
   static S* getInstance() {
      if (!p_instance)
         p_instance = new S();
      return p_instance;
   }

private:
   static S* p_instance = nullptr;
   S();
   S(S const&) = delete;
   void operator=(S const&) = delete;
};
```

Недостаток - за удаление ответственны клиенты.

РЕАЛИЗАЦИЯ ПАТТЕРНА SINGLETON

```
// Singleton Мэйерса
class S {
public:
    static S& getInstance() {
        static S instance;  // Гарантированно будет уничтожен.
                            // Конструируется при первом обращении.
        return instance;
    }



private:
    S();
    S(S const&) = delete;
    void operator=(S const&) = delete;
};
```

РЕАЛИЗАЦИЯ ПАТТЕРНА SINGLETON

```
// Улучшенная версия классической реализации
class S;

class Destroyer {
    S* p_instance;
public:
    ~Destroyer();
    void initialize(S* p);
};


class S {
    static S* p_instance = nullptr;
    static Destroyer destroyer{};
public:
    static S& getInstance();
protected:
    friend class Destroyer;
    S(){}
    ~S(){}
    S(S const&) = delete;
    void operator=(S const&) = delete;
};
```

```
Destroyer::~Destroyer() {
    delete p_instance;
}
void Destroyer::initialize(S* p) {
    p_instance = p;
}
S& S::getInstance() {
    if(!p_instance) {
        p_instance = new S();
        destroyer.initialize( p_instance);
    }
    return *p_instance;
}
```

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ SINGLETON

Достоинства:

- Гарантирует наличие единственного экземпляра класса.
- Предоставляет к нему глобальную точку доступа.
- Реализует отложенную инициализацию Singleton объекта.

Недостатки:

- Нарушает принцип единственной ответственности класса.
- Маскирует плохой дизайн.

BUILDER

- Паттерн *Строитель* позволяет конструировать сложные объекты пошагово.
- Конструирование сложного объекта отделяется от его представления.
- Один и тот же процесс конструирования может порождать различные представления.

НАЗНАЧЕНИЕ ПАТТЕРНА BUILDER

Паттерн Builder может помочь в решении следующих задач:

- В системе могут существовать сложные объекты, создание которых за одну операцию затруднительно или невозможно.

- Данные должны иметь несколько представлений.

Распорядитель

Director

construct()

builder

Строитель

Builder

buildPartA()
buildPartB()

вызывает
builder.buildPartA() *и*
builder.buildPartB()

Client

ConcreteBuilder1

buildPartA()
buildPartB()

ConcreteBuilder2

buildPartA()
buildPartB()

```
struct Wheel {
    int size;
};

struct Engine {
    int horsepower;
};

struct Body {
    enum { SUV, HATCHBACK, SEDAN } shape;
    const string &toString();
};
```

```
class Builder {
public:
    virtual Wheel  *getWheel()  = 0;
    virtual Engine *getEngine() = 0;
    virtual Body   *getBody()   = 0;
};
```

```
struct Car {
    Wheel  *wheels[4];
    Engine *engine;
    Body   *body;

    void specifications() {
        cout << "body:" << body->toString() << endl;
        cout << "engine horsepower:" << engine->horsepower << endl;
        cout << "tire size:" << wheels[0]->size << "'" << endl;
    }
};
```

```
class JeepBuilder : public Builder {
public:
    Wheel *getWheel() {
        Wheel *wheel = new Wheel();
        wheel->size = 22;
        return wheel;
    }

    Engine *getEngine() {
        Engine *engine = new Engine();
        engine->horsepower = 400;
        return engine;
    }

    Body *getBody() {
        Body *body = new Body();
        body->shape = Body::SUV;
        return body;
    }
};
```

```
class NissanBuilder : public Builder {
public:
    Wheel *getWheel() {
        Wheel* wheel = new Wheel();
        wheel->size = 16;
        return wheel;
    }

    Engine *getEngine() {
        Engine* engine = new Engine();
        engine->horsepower = 85;
        return engine;
    }

    Body *getBody() {
        Body *body = new Body();
        body->shape = Body::HATCHBACK;
        return body;
    }
};
```

```
class Director {
    Builder *builder;
public:
    void setBuilder(Builder *newBuilder) {
        builder = newBuilder;
    }

    Car *getCar() {
        Car *car = new Car();

        car->body = builder->getBody();
        car->engine = builder->getEngine();

        car->wheels[0] = builder->getWheel();
        car->wheels[1] = builder->getWheel();
        car->wheels[2] = builder->getWheel();
        car->wheels[3] = builder->getWheel();

        return car;
    }
};
```

```
int main() {
    Car car;
    Director director;

    JeepBuilder jeepBuilder;
    NissanBuilder nissanBuilder;

    cout << "Jeep" << endl;
    director.setBuilder(&jeepBuilder);
    car = director.getCar();
    car->specifications();

    cout << endl;

    cout << "Nissan" << endl;
    director.setBuilder(&nissanBuilder);
    car = director.getCar();
    car->specifications();

    return 0;
}
```

- Внутреннее представление продукта можно легко изменять (создав новый подкласс *Builder*).

- Конструирование отдельно, представление отдельно. Представление (внутренняя структура) скрыто.

- Тонкий контроль над процессом конструирования (*Director* инкапсулирует алгоритм конструирования).

FACTORY METHOD

- *Фабричный метод* — паттерн, порождающий объекты.
- Определяет интерфейс для создания объекта. Какой объект какого именно класса будет создан — деталь реализации (определяется подклассами).

НАЗНАЧЕНИЕ ПАТТЕРНА FACTORY METHOD

Когда надо применять:

- Когда заранее неизвестно, объекты каких типов необходимо создавать.
- Когда система должна быть независимой от процесса создания новых объектов и расширяемой.
- Когда создание новых объектов необходимо делегировать из базового класса классам наследникам.

Product

ConcreteProduct

Creator

createProduct()
doSomething()

ConcreteCreator

createProduct()

Product *product = createProduct();
…

return new ConcreteProduct();

```
struct Product {
    virtual string getName() = 0;
};

struct ConcreteProductA : Product {
    string getName() { return "ConcreteProductA"; }
};

struct ConcreteProductB : Product {
    string getName() { return "ConcreteProductB"; }
};
```

```
struct Creator {
    virtual Product *factoryMethod() = 0;
};

struct ConcreteCreatorA : Creator {
    Product *factoryMethod() { return new ConcreteProductA(); }
};

struct ConcreteCreatorB : Creator {
    Product *factoryMethod() { return new ConcreteProductB(); }
};
```

```
int main() {
    ConcreteCreatorA CreatorA;
    ConcreteCreatorB CreatorB;

    Creator *creators[] = { &CreatorA, &CreatorB };

    for (auto creator : creators) {
        Product *product = creator->factoryMethod();
        cout << product->getName() << endl;
        delete product;
    }

    return 0;
}
```

Параметризованное создание объектов

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

    static Shape *create(string type);
};

class Circle : public Shape {
public:
    void draw() { cout << "I am circle" << endl; }
};
```

```
class Square : public Shape {
public:
    void draw() { cout << "I am square" << endl; }
};

Shape *Shape::create(string type) {
    if (type == "circle") return new Circle();
    if (type == "square") return new Square();
    return 0;
}
```

```
void main() {
    // Круг хочу
    Shape *obj1 = Shape::create("circle");

    // Квадрат хочу
    Shape *obj2 = Shape::create("square");

    obj1->draw();
    obj2->draw();

    delete obj1;
    delete obj2;
}
```

```
struct Creator {
    virtual Product *Create(int id) {
        if (id == MINE)
            return new MyProduct();
        if (id == YOURS)
            return new YourProduct();

        return 0;
    }
};

struct MyCreator : Creator {
    Product *Create(int id) {
        if (id == THEIRS)
            return new TheirProduct();

        return Creator::Create(id);
    }
};
```

Параметризованное создание с наследованием

- В **Creator** можно задать поведение по умолчанию, а можно и не задавать.
- Параметризация: нужна, когда снаружи поступает ID или имя класса.

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ FACTORY METHOD

Достоинства:

- Избавляет класс от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует принцип открытости/закрытости.

Недостатки:

- Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

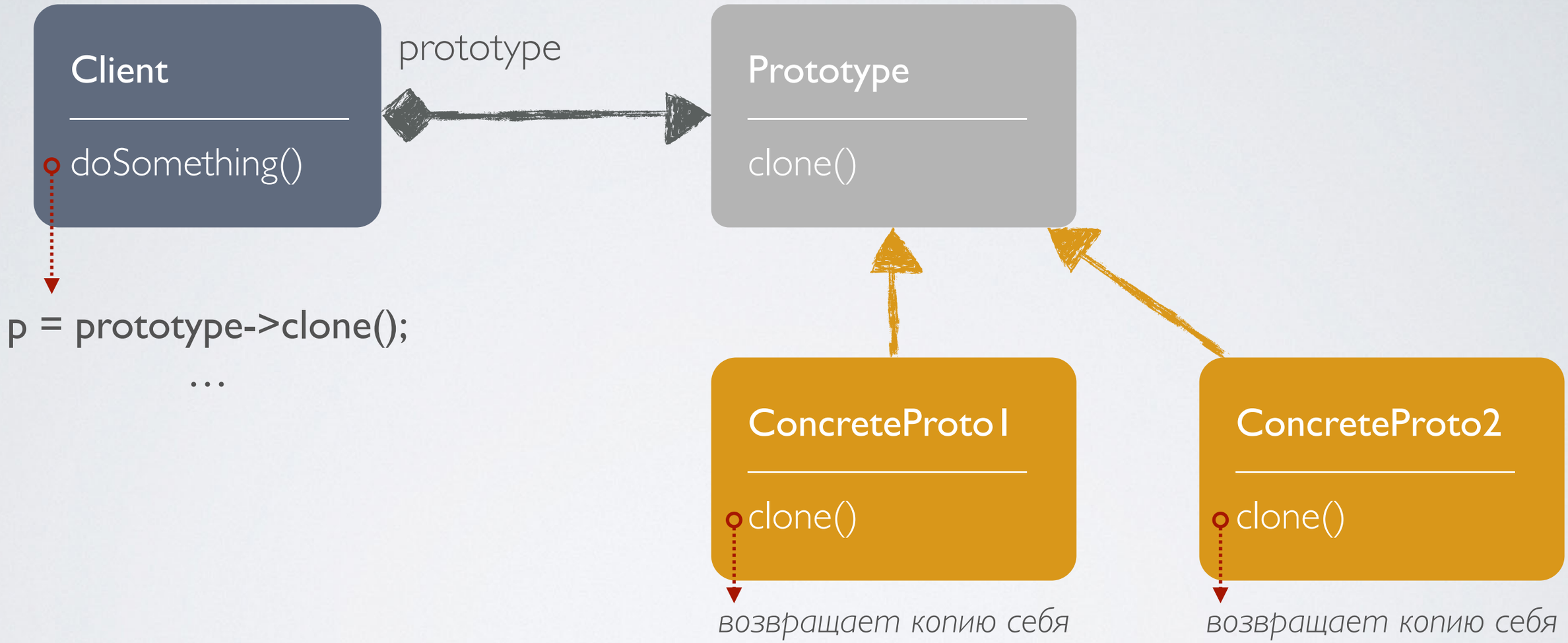
PROTOTYPE

- *Прототип* — паттерн, который позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.

НАЗНАЧЕНИЕ ПАТТЕРНА PROTOTYPE

Когда надо применять:

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения.
- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора.




```
p = prototype->clone();
...
```

```
class Prototype {
    std::string type;
    int value;

public:
    virtual Prototype *clone() = 0;

    std::string getType() { return type; }
    int getValue()        { return value; }
};

class ConcretePrototype1 : public Prototype {
public:
    ConcretePrototype1(int number)
        : type("Type1"), value(number) {

    Prototype *clone() {
        return new ConcretePrototype1(*this);
    }
};

class ConcretePrototype2 : public Prototype {
public:
    ConcretePrototype2(int number)
        : type("Type2"), value(number) {

    Prototype *clone() {
        return new ConcretePrototype2(*this);
    }
};
```

```
class ObjectFactory {
    static Prototype *type1value1;
    static Prototype *type1value2;
    static Prototype *type2value1;
    static Prototype *type2value2;

public:
    static void initialize() {
        type1value1 = new ConcretePrototype1(1);
        type1value2 = new ConcretePrototype1(2);
        type2value1 = new ConcretePrototype2(1);
        type2value2 = new ConcretePrototype2(2);
    }

    static Prototype *getType1Value1() {
        return type1value1->clone();
    }

    static Prototype *getType1Value2() {
        return type1value2->clone();
    }

    static Prototype *getType2Value1() {
        return type2value1->clone();
    }

    static Prototype *getType2Value2() {
        return type2value2->clone();
    }
};

Prototype *ObjectFactory::type1value1 = 0;
Prototype *ObjectFactory::type1value2 = 0;
Prototype *ObjectFactory::type2value1 = 0;
Prototype *ObjectFactory::type2value2 = 0;
```

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ PROTOTYPE

Достоинства:

- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Меньше повторяющегося кода инициализации объектов.
- Прототипы можно создавать и удалять динамически (в отличие от классов в Factory Method). Количество классов уменьшается.
- Различие поведения задается не созданием новых классов, а заданием атрибутов (значений членов).

Недостатки:

- Сложно клонировать составные объекты, имеющие ссылки на другие объекты.