

# Шаблоны проектирования: практические примеры

Проектирование программного обеспечения сегодня стало инженерной наукой.

Олег Федоров ([bishop3000@rambler.ru](mailto:bishop3000@rambler.ru))

Первых проектировщиков ПО можно было назвать вольными творцами, художниками от программирования, создававшими все с нуля, однако современное проектирование — это, в основном, использование готовых решений, шаблонов проектирования, так называемых «паттернов» (pattern — шаблон, в дальнейшем эти слова будут употребляться как синонимы). «Изобретение велосипедов» в архитектуре ПО стало делом неблагодарным, и любой проектировщик сейчас обязан знать как минимум базовый набор стандартных решений, а хороший — десятки, если не сотни шаблонов проектирования (а заодно

же цели служат паттерны проектирования в программировании, они рассматриваются как стандартные строительные блоки для архитекторов программного обеспечения и позволяют им легко находить общий язык.

Рассмотрим первые 23 паттерна, с которых все началось и которые сейчас должен знать каждый программист. Они разделены на три группы (для каждого паттерна приводится английское название из книги GoF и устоявшийся русский перевод).

«Порождающие шаблоны». В этой группе собраны паттерны, описывающие разные способы создания объектов. Прежде всего это «Фабричный метод» (Factory Method), прием определения интерфейса создания объектов, при этом выбранный класс воплощается в подклассах. Шаблон «Абстрактная фабрика» (Abstract Factory) определяет интерфейс для создания семейств, связанных между собой или независимых объектов, конкретные классы которых неизвестны. С помощью шаблона «Строитель» (Builder) можно отделить процесс конструирования сложного объекта от его конкретного представления и при этом использовать один и тот же процесс для создания различных представлений. «Прототип» (Prototype) описывает виды разрабатываемых объектов с помощью прототипа и создает новые путем его копирования. Применение шаблона «Одиночка» (Singleton) гарантирует, что некоторый класс может иметь только один экземпляр (и предоставляет глобальную точку доступа к нему).

«Структурные паттерны». В этой группе собраны паттерны, которые позволяют менять структуру взаимодействия классов. «Адаптер» (Adapter) позволяет адаптировать интерфейс класса к конкретной ситуации, средствами шаблона «Мост» (Bridge) можно отделить интерфейс класса и его реализацию, «Компоновщик» (Composite) объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам единообразно обращаться к отдельным объектам и группам объектов. Паттерн «Оформитель» (Decorator, также известен как Wrapper, «Оболочка») позволяет динамически добавлять новое поведение к объекту, «Фасад» (Facade) — скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. Шаблон

*Известность термин «паттерн»*

*получил после публикации книги*

*«Приемы объектно-ориентированного проектирования»*

столько же «антипаттернов», примеров того, «как делать не надо», причем осознавая, почему не следует поступать именно так).

## Что такое шаблоны проектирования

Всемирную известность термин «паттерн» получил после публикации книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования». Авторы книги, Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес больше известны, как «Банда четырех» (Gang of Four, часто сокращается до GoF). В этой книге было описано всего 23 шаблона, но она дала толчок к появлению новых работ на эту тему.

Можно сказать, что паттерны — это стандартные решения типовых задач, возникающих в объектно-ориентированном проектировании, некий набор готовых решений, пригодных для большинства задач.

Здесь можно привести сравнение со строительством дома. Все знают, что такое колонна, окно, дверь, труба, и зачем они нужны в здании. Наличие множества общих терминов позволяет упростить проектирование здания, а также общаться с коллегами, специализирующимися в других областях, на одном всем понятном языке. Этой

«Приспособленец» (Flyweight) используется для облегчения работы с большим числом мелких объектов, а «Заместитель» (Proxy) — контролировать доступ к объекту, перехватывая все вызовы к нему.

В группе «Паттерны поведения» собраны шаблоны, ответственные за реализацию поведения объектов. «Цепочка ответов» (Chain of Response) позволяет пропустить запрос через цепочку объектов, «Команда» (Command) инкапсулирует команду в объект, «Интерпретатор» (Interpreter) позволяет создать общее декларативное решение для часто изменяющихся условий задачи. В шаблоне «Итератор» (Iterator) организуется последовательный доступ к коллекции, «Посредник» (Mediator) определяет упрощенный механизм взаимодействия классов, «Напоминание» (Memento) задает принципы, позволяющие записывать и восстанавливать внутреннее состояние объекта. Средствами шаблона «Наблюдатель» (Observer) можно оповещать об изменениях множества объектов, «Состояние» (State) — менять поведение объекта при изменении его состояния. «Стратегия» (Strategy) инкапсулирует алгоритм внутри класса.

Паттерн «Шаблонный метод» (Template Method) выделяет конкретные шаги в алгоритме и опирается на подклассы для их реализации. Средствами паттерна «Посетитель» (Visitor) в класс добавляются новые операции без его изменения.

В этой статье мы рассмотрим паттерны из первой группы — «Порождающие паттерны».

### «Фабричный метод»

Паттерн Factory Method позволяет реализовать идею «виртуального конструктора», то есть создания объектов без явного указания их типа. Обычно он применяется, когда базовый класс делегирует потомкам право принятия решений о типе создаваемого объекта. Предположим, что нам надо написать программу для сохранения документа на диск, причем в разных форматах: XML, .txt, .doc и даже бинарном. Создадим базовый класс, где будет код открытия файла и фабричный метод для создания

конкретного объекта, который будет сохранять данные в этот файл.

```
class DocumentWriter
{
public: virtual void writeDoc(const Document& doc, FILE& file) = 0;
};
class SaveDocument
{
protected:
    virtual DocumentFileWriter* createDocumentWriter() const = 0;
public:
    void save(const char* fileName)
    {
        FILE file = openFile(fileName);
        // Фабричный метод
        DocumentFileWriter* writer = createDocumentWriter();
        writer->writeDoc();
        closeFile(file);
    }
};
```

Обратите внимание, что теперь не надо менять базовые классы для добавления любого нового формата выходного файла. Нужно только создать два простых дополнительных класса, например для вывода в XML (см. листинг 1).

Главный недостаток этого шаблона — необходимость создания двух дополнительных классов при добавлении новой функции.

### «Абстрактная фабрика»

Шаблон Abstract Factory предоставляет интерфейс для создания семейств связанных или зависимых объектов, позволяя не указывать их конкретные классы. Обычно он применяется, когда нужно создавать составные объекты из конкретных частей (объектов). Классический пример — автомобиль. Если есть классы для разных частей автомобиля, то конкретный автомобиль будет состоять из множества таких частей, причем в каждой марке автомобиля будут конкретные части-детали. Для окончательной сборки автомобиля можно использовать «Абстрактную фабрику», классы-потомки которой будут создавать конкретные автомобили.

```
class AbstractCarFactory
{
public:
    virtual Car* createCar() = 0;
    virtual Engine* createEngine() = 0;
    virtual Wheel* createWheel() = 0;
    ...
};
```

### Листинг 1

```
class DocumentWriterXml : public DocumentWriter
{
public:
    virtual void writeDoc(const Document& doc, FILE& file)
    {
        // код для сохранения документа в формате XML в файл file
    }
};

class SaveDocumentXml : public SaveDocument
{
protected:
    virtual DocumentFileWriter* createDocumentWriter() const {return new DocumentWriterXml();}
};
```

## Листинг 2

```

class Engine
{
public:
    virtual void setSpeedLimit(int kmph);
    virtual void addOil(float amount);
    ...
};
class Wheel
{
    float mRadius;
public:
    Wheel(float radius) : mRadius(radius) {};
    virtual float getRadius() const {return mRadius;}
    ...
};
class Car
{
public:
    virtual const char* getName() const = 0;
    virtual void setEngine(Engine* engine);
    virtual void setWheels(Wheel* FR, Wheel* FL, Wheel* BR, Wheel* BL);
    virtual void setFrontWheels(Wheel* FR, Wheel* FL);
    virtual void setBackWheels(Wheel* BR, Wheel* BL);
    ...
};

```

Нам также понадобятся иерархии классов для каждой из частей автомобиля (см. листинг 2).

Процедура сборки итогового объекта приводится в листинге 3.

Для добавления конкретных моделей автомобилей нужно просто создать новый класс, породив его от AbstractCarFactory. Задача этого класса — создать конкретные детали автомобиля. Понадобится также создать классы этих конкретных деталей, если их еще нет.

```

class BMW5CarFactory : public AbstractCarFactory
{
public:
    virtual Car* createCar() {return new CarBMW5();}
    virtual Engine* createEngine() {return new EngineBMW5();}
    virtual Wheel* createWheel() {return new WheelNokian(16);}

```

```

...
};
class CarBMW5 : public Car
{
public:
    virtual const char* getName() const {return "BMW5";}
    ...
};

```

Главный недостаток этого паттерна состоит в том, что со временем становится труднее вносить изменения в структуру классов. Например, если у нас уже имеется 100 конкретных моделей автомобилей, а мы решили добавить в AbstractCarFactory новую функцию, создающую фары, то нам придется изменить все 100 классов-потомков, создающих конкретные модели.

## «Строитель»

Суть этого паттерна — отделить процесс создания сложного объекта от его представления. Таким образом, можно получать различные представления объекта, используя один и тот же технологический процесс его создания. На первый взгляд этот паттерн кажется похожим на «Абстрактную фабрику», но отличия есть. Можно сказать, что «Абстрактная фабрика» концентрируется на том, *что создается*, а «Строитель» на том, *как создается*.

Рассмотрим тот же самый пример с автомобилями. Для создания автомобиля нам теперь понадобится специальный класс CarBuilder.

```

class CarBuilder
{
    Car* mCar;
    virtual Car* _createNewCar() const {return Car();}
public:
    Car* getCar() const {return mCar;}
    void createNewCar() {mCar = _createNewCar();}

    virtual void addEngine() = 0;
    virtual void addWheels() = 0;
    ...
};

```

## Листинг 3

```

Car* createCar(AbstractCarFactory* carFactory)
{
    Car* car = carFactory->createCar();
    Engine* engine = carFactory->createEngine();
    car->setEngine(engine);
    car->setWheels(carFactory->createWheel(), carFactory->createWheel(), carFactory->createWheel(), carFactory->createWheel());
    // дальше может идти сколько угодно сложный код для "сборки" готового автомобиля из частей
    return car;
}

```

#### Листинг 4

```
class BMW5CarBuilder : public CarBuilder
{
public:
    virtual void addEngine()
    {
        Engine* engine = new EngineBMW5();
        engine->setSpeedLimit(250);
        engine->addOil(10);
        getCar()->setEngine(engine);
    }
    virtual void addWheels()
    {
        getCar()->setFrontWheels(new WheelNokian(16), new WheelNokian(16));
        getCar()->setBackWheels(new WheelBridestone(16), new WheelBridestone(16));
    }
    ...
};
```

Необходимо также создать класс Director, управленец, в котором будет реализован алгоритм сборки автомобиля. Этот класс не создает конкретных автомобилей, а использует для этого CarBuilder. Его задача — воплотить общий алгоритм конструирования автомобиля, тогда как детали должны быть реализованы в потомках класса CarBuilder. Таким образом достигается важная задача: общий алгоритм создания автомобиля остается неизменным, но при этом конкретный автомобиль может создаваться по сколь угодно сложной схеме, так как CarDirector делегирует CarBuilder всю работу по созданию конкретного автомобиля.

```
class CarDirector
{
    carBuilder* mBuilder;
public:
    void setCarBuilder(CarBuilder* builder)
    {
        mBuilder = builder;
    }
    Car* getCar() const
    {
        return mBuilder->getCar();
    }
    void constructCar()
    {
        mBuilder->createNewCar();
        mBuilder->addEngine();
        mBuilder->addWheels();
        ...
    }
};
```

Итак, теперь у нас есть класс CarDirector, который задает правила создания автомобилей, и класс CarBuilder, потомки которого создают конкретные

детали автомобиля по правилам, описанным в CarDirector. Например, для сборки BMW5 нам понадобится новый класс, порожденный от CarBuilder (см. листинг 4).

Обратим внимание на отличие в конструировании объектов от паттерна «Абстрактная фабрика». Класс BMW5CarBuilder имеет больше возможностей конструирования, чем BMW5CarFactory, поскольку он нацелен на процесс сборки. А BMW5CarFactory просто создает детали и не может влиять на процесс конструирования. В этом основное различие между ними.

Главная проблема этого паттерна та же, что и у «Абстрактной фабрики» — сложно вносить изменения в систему, если она уже наполнена конкретными классами, так как придется изменять их все.

Достоинства паттерна «Строитель» проявляются при создании сложных и больших иерархий объектов, поскольку у конкретных классов-строителей (потомков CarBuilder) есть возможность влиять на алгоритм конструирования объекта. Например, если когда-либо придется создавать автомобиль без двигателя, то метод addEngine для его разработчика просто будет пуст, тогда как логика работы CarDirector сохранится, поскольку все изменения будут локализованы в новом классе — потомке CarBuilder.

#### «Прототип»

Паттерн Prototype обычно применяется, когда можно заранее создать объекты всех нужных типов, просто найти среди них подходящий и клонировать его, т. е. создать новый объект — копию эталонного. Второе применение — это операции, аналогичные копированию-вставке (Ctrl-C, Ctrl-V). Например, если пользователь «выделил» автомобиль, скопировал его и хочет создать его дубликат — проще всего использовать данный паттерн и клонировать выделенный объект.

Для клонирования объектов нужно добавить в базовый класс метод клонирования, который обычно называют

#### Листинг 5

```
class Car
{
public:
    virtual Car* clone() const = 0;
    virtual const char* getName() const = 0;
    virtual void setEngine(Engine* engine);
    virtual Engine* getEngine() const;
    virtual void setWheels(Wheel* FR, Wheel* FL, Wheel* BR, Wheel* BL);
    virtual void setFrontWheels(Wheel* FR, Wheel* FL);
    virtual void setBackWheels(Wheel* BR, Wheel* BL);
    ...
};
```

**Листинг 6**

```
class CarBMW5 : public Car
{
public:
    virtual Car* clone() const
    {
        CarBMW5* car = new CarBMW5();
        car->setEngine(getEngine()->clone());
        car->setFrontWheels(getWheelFR()->clone(), getWheelFL()->clone());
        car->setBackWheels(getWheelBR()->clone(), getWheelBL()->clone());
        ...
        return car;
    }
    ...
};
```

clone. Продолжим пример с автомобилями — расширим интерфейс класса Car (листинг 5).

Теперь в потомках класса Car нужно перезагрузить и реализовать этот метод, он должен создавать абсолютно идентичную новую копию объекта (см. листинг 6).

Обратите внимание, что функция клонирования CarBMW5::clone() вызывает функции клонирования для всех своих составных частей. Это стандартная практика: если использовать функцию клонирования в составном объекте, то нужно добавить ее во все объекты, которые он содержит. Таким образом, автомобиль легко копирует себя, так как просто последовательно запускает процесс копирования всех составных частей и собирает из них новый автомобиль — свою полную копию.

Применений у этого паттерна может быть несколько, как уже было сказано выше. В случае с копированием объектов (Ctrl-C, Ctrl-V) мы будем иметь массив выделенных объектов и при нажатии на Ctrl-C просто создадим их копии, используя метод clone. А нажав на Ctrl-V, мы вставляем эти объекты в нужное место. Мы также можем заранее создать все возможные объекты (типы автомобилей) и поместить их в один массив. А потом легко создавать автомобили по имени: находим объект с нужным именем в массиве и вызываем его метод clone(). Простой пример.

```
class CarFactory
{
    std::vector<Car*> mCarTemplates;
public:
    CarCreator()
    {
        mCarTemplates.push_back(new CarBMW5());
        mCarTemplates.push_back(new CarToyotaHiace());
        mCarTemplates.push_back(new CarLadaKalina());
    }
    Car* create(const char* name)
    {
        for(size_t i = 0; i < mCarTemplates.size(); ++i)
            if (strcmp(mCarTemplates[i]->getName(), name) == 0)
```

```
        return mCarTemplates[i]->clone();
        return NULL;
    }
};
```

**«Одиночка»**

Паттерн Singleton гарантирует, что объект какого-то класса будет создан только один раз. Это, пожалуй, самый спорный из всех шаблонов проектирования. Многие профессионалы объектно-ориентированного проектирования не рекомендуют применять этот паттерн, поскольку считают его обычным аналогом глобальных переменных. Самая простая реализация паттерна Singleton — это так называемый «синглтон Мейерса», где «Одиночка» представляет собой статический локальный объект (это решение небезопасно при работе с нитями).

```
template<typename T> class Singleton
{
public:
    static T& instance()
    {
        // у класса T есть конструктор по умолчанию
        static T theSingleInstance;
        return theSingleInstance;
    }
};
```

Чтобы класс стал «Одиночкой», его достаточно породить от Singleton.

Обычно всякие фабрики и контейнеры должны присутствовать в системе в единственном экземпляре, поэтому их разумно порождать от Singleton. Расширим класс CarFactory из предыдущего примера.

```
class CarFactory : public Singleton<CarFactory>
{
    ... дальше то же, что и в CarFactory из паттерна "Прототип"
};
```

Все обращения к такому объекту ведутся через метод Instance(). Для создания автомобилей по названию нужно просто написать следующий код.

```
Car* bmw = CarFactory::instance().create("BMW5");
Car* hiace = CarFactory::instance().create("ToyotaHiace");
```

При этом объект типа CarFactory фактически будет создан в момент первого вызова метода instance(), а удален только при выходе из программы.

Итак, мы рассмотрели в этой статье первую группу классических паттернов — «Порождающие». Их нужно знать и понимать очень хорошо, они встречаются наиболее часто, ведь создание объектов — неотъемлемая часть любого алгоритма. ❗

*Продолжение следует*