Урок 3

Порождающие шаблоны

Обзор. Реализация. Недостатки. Singleton. Factory Method. Abstract Fabric. Builder

Что такое паттерн проектирования

Паттерны проектирования – это подходы к решению практических задач, **выявленные** при анализе полученных решений и применяемые многократно. Паттерны не открывают или изобретают – они выявляются как повторяющиеся конструкции в коде, структуре или архитектуре при разработке программ.

Кристофер Александер писал, «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново» [7].

Паттерны дают направление решения проблемы, но не дают окончательного результата – это не входит в их задачу. Ценность паттерна состоит в том, что будучи идентифицированным однажды, он позволяет решить большое количество схожих проблем с меньшими усилиями.

Чаще всего одну задачу можно решить разными способами с применением разных паттернов: некоторые подходят лучше, некоторые – нет. Неоправданное приведение решения к использованию неподходящего в данном контексте паттерна само по себе является антипаттерном. Паттерны – не панацея, а лишь крупные строительные блоки, которые не заменят алгоритм или конкретную логику вашего приложения.

Обзор паттернов проектирования

В начале 1990-х Эрих Гамма, вдохновленный книгой К.Александера «A Pattern Language – Towns, Buildings, Construction», обдумывает каталог паттернов проектирования программного обеспечения как тему своей докторской. В дальнейшем к работе над каталогом присоединяются Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Результатом их работы стала ныне широко известная книга [1], включающая в себя каталог и 23 основных шаблона проектирования, более известных как GoF («Банда четырех»).

В 1996 году группа инженеров Simens опубликовала свой набор паттернов, известный как POSA.

Мартин Фаулер в 2001 году в книге «Шаблоны корпоративных приложений» описал каталог паттернов проектирования корпоративных приложений.

В 2001 году в книге [4] был представлен каталог паттернов для платформы J2EE, который включает в себя паттерны уровня представления, слоя доступа к данным, сервисного слоя и слоя для разработки веб-приложений.

Существует каталог паттернов параллельного программирования, впервые описанный в [6] в 2000 году.

Классический каталог паттернов GoF

Классический каталог паттернов GoF описывает основу основ: базовые паттерны создания, структурирования и взаимодействия объектов на самом нижнем уровне проектирования – на уровне классов и реальных объектов программы.

Другие паттерны более высокого уровня, например, паттерн MVC, могут использовать эти базовые паттерны для своего построения. Например, паттерн MVC может использовать паттерн Фабричный метод для определения конкретного класса Контроллера и Декоратор для добавления к представлению возможности прокрутки – основные отношения описываются паттернами Наблюдатель, Компоновщик и Стратегия.

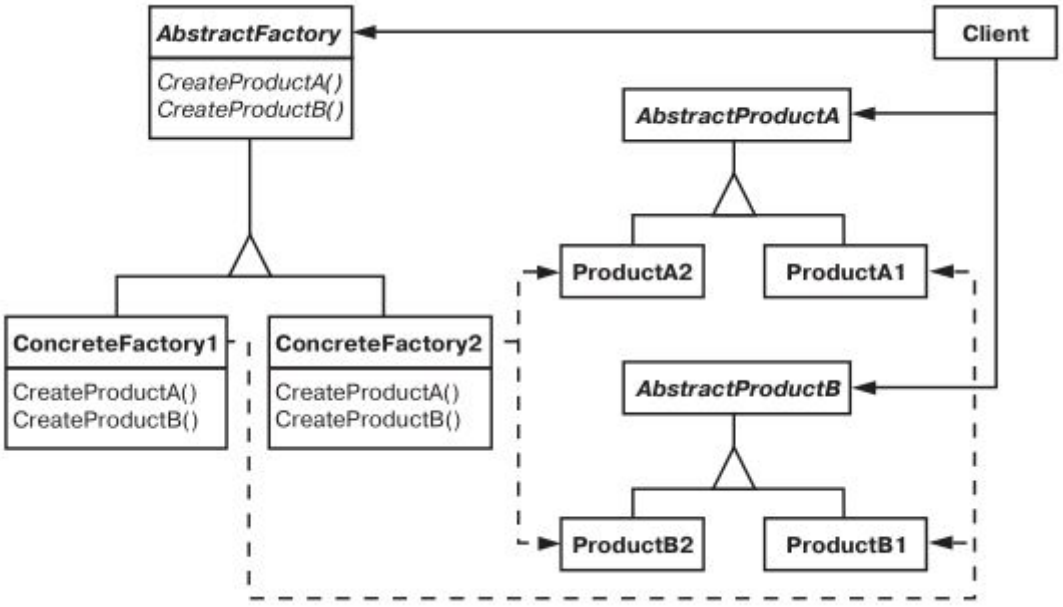
Каталог GoF классифицирует паттерны по трем целям

1. Порождающие – абстрагируют процесс инстанцирования объектов.
2. Структурные – относятся к вопросам создания более крупных структур из классов и объектов.
3. Поведенческие – характеризуют то, как классы или объекты взаимодействуют между собой.

Порождающие шаблоны

Создание объектов в банальной манере оператором new может привести к проблемам дизайна или к дополнительной сложности. Порождающие паттерны проектирования решают эту проблему, так или иначе управляя процессом создания объектов. Если оператор **new** – это арифметика чисел, то Порождающие паттерны – «алгебра оператора **new**».

Abstract Factory // Абстрактная фабрика



Идиома – фабрика, порождающая различные реализации объектов предопределенных интерфейсов. Позволяет легко масштабировать вширь реализации связанных семейств классов при неизменной логике клиентского кода.

- Client – клиентский код, который использует исключительно интерфейсы, объявленные в классах AbstractFactory и AbstractProduct.
- AbstractFactory – объявляет интерфейс для операций, создающих абстрактные объекты Продукты.
- ConcreteFactory – конкретная фабрика, имплементирует операции создания конкретных экземпляров классов объектов Продуктов.

- AbstractProduct – абстрактный продукт, описывает интерфейс продукта, которым пользуется клиент.
- ConcreteProduct – конкретный продукт, конкретная реализация класса имплементирует интерфейс AbstractProduct. Объекты этого класса создаются соответствующей реализацией конкретной фабрики и клиенту не видны.

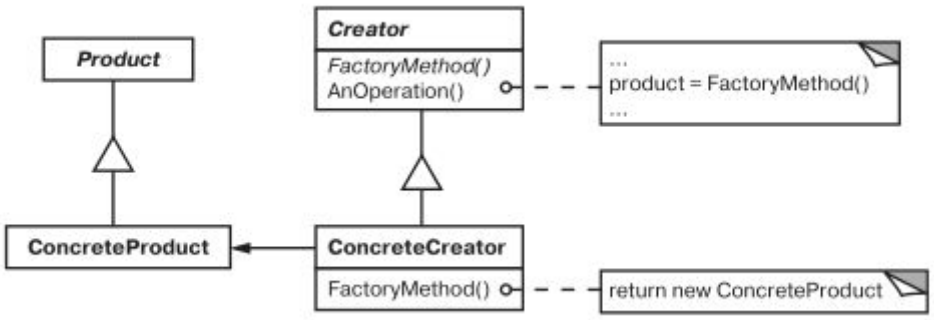
Достоинства, особенности

- Открытая расширяемость при неизменном поведении клиентского кода.
- Клиентский код не знает ничего о классах конкретной фабрики и конкретных продуктах, код развязывается по зависимостям.
- Конкретный объект фабричного класса обычно инстациируется в единственном экземпляре, при этом часто используется паттерн Factory Method (см. ниже), в тоже время возможно использование паттерна Singleton.
- Право инстанциации конкретных классов продуктов остается за клиентским кодом – создает только то, что потребно.

Недостатки

- Жесткий нерасширяемый фабричный интерфейс и интерфейс продуктов. При модификации интерфейса фабрики или продуктов изменения потребуется вносить во все, возможно, многочисленные, реализации.

Factory Method // Фабричный метод // уровень класса



- Product – интерфейс объектов, создаваемых фабричным методом.
- ConcreteProduct – конкретный продукт, имплементирует интерфейс Product.
- Creator – абстрактный создатель (чаще всего, каркас), объявляет Абстрактный метод, возвращающий объект реализующий тип Product, а также вызывает Фабричный метод, реализованный в потомках для создания объекта имплементирующего интерфейс Product.
- ConcreteCreator – конкретный создатель, замещает Фабричный метод, возвращающий объект.

Создатель «полагается» на свои подклассы в определении Фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта. Паттерн избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с

интерфейсом класса Product, поэтому может работать с любыми определенными пользователями классов конкретных продуктов.

Основная мысль такова: ConcreteCreator (или реже – не абстрактный Creator с реальным методом) инстанциируют объекты классов, реализующих известный клиентскому коду интерфейс, не завязывая клиентский код на эти конкретные реализации.

Пример использования паттернов: Абстрактная фабрика и Фабричный метод

Проектируем систему планирующую закупки у расширяющегося (и сужающегося) списка поставщиков плюс документооборот (накладные), плюс отчеты для маркетинговых мероприятий поставщиков. Построить всех поставщиков под одну гребенку вряд ли получится, а вот инкапсулировать логику разнообразных их информационных систем можно, применив два рассмотренных паттерна.

Определяем интерфейсы, описывающие функционал информационного обмена с поставщиками.

Очевидно, что поставщик обычно предоставляет цену на товары:

```
public interface PriceProvider {
    Money getPrice(String article);
}
```

И некоторый документооборот:

```
public interface DocProvider {
    Doc getDoc(int id);
    void sendPayment(Payment payment);
}
```

А также отбирает данные о продажах и выплачивает бонусы особо талантливым реселлерам.

```
public interface MarketingProvider {
    void claimSales();
    Money getBonus();
}
```

Эти три интерфейса описывают интерфейс продуктов. См. AbstractProduct на диаграмме классов.

Кроме того, нам потребуется описать интерфейс Абстрактной фабрики:

```
public interface ExchangeFactory {
    PriceProvider createPriceProvider();
    DocProvider createDocProvider();
    MarketingProvider createMarketingProvider();
}
```

Различные реализации интерфейсов конкретными поставщиками учитывают их специфику. При рассмотрении паттернов – это второстепенная деталь. Важно, что они имплементируют общее, заданное заранее поведение, но каждый по-своему. Пример реализации одного из провайдеров:

```
public class CitilinkPriceProvider implements PriceProvider {
    @Override
    public Money getPrice(String article) {
        return catalog.findByArticle(article).getPrice();
    }
    ...
}
```

Имплементация Конкретной фабрики достаточно банальна:

```
public class CitilinkExchangeFactory implements ExchangeFactory {

    @Override
    public PriceProvider createPriceProvider() {
        return new CitilinkPriceProvider();
    }

    @Override
    public DocProvider createDocProvider() {
        return new CitilinkDocProvider();
    }

    @Override
    public MarketingProvider createMarketingProvider() {
        return new CitilinkMarketingProvider();
    }
}
```

Теперь в игру вступает паттерн Фабричный метод. Ответственность класса Fabric – знать о конкретных реализациях Абстрактной фабрики и, соответственно, задача метода createFactory(...) – вернуть конкретную реализацию конкретной фабрики на основе внешней конфигурационной информации: например, строки.

```
public class Fabric {
    // определяем конфигурационные константы
    public static final String SUPPLIER_ONE = "Citilink";
    public static final String SUPPLIER_TWO = "Ulmart";

    // создать объект, реализующий известный интерфейс на основе внешней информации
    public ExchangeFactory createFactory(String name) throws Exception {
        switch (name) {
            case SUPPLIER_ONE:
    // 1) вариант – с жесткой зависимостью
                return new CitilinkExchangeFactory();
            default:
    // 2) вариант – позднее связывание, с интроспекцией
                Package aPackage = Fabric.class.getPackage();
                Reflections reflections = new Reflections(aPackage);
                Set<Class<? extends ExchangeFactory>> typesOf =
                    reflections.getSubTypesOf(ExchangeFactory.class);

                for (Class<? extends ExchangeFactory> aClass : typesOf) {
                    if (aClass.getSimpleName().contains(SUPPLIER_TWO)) {
                        return aClass.newInstance();
                    }
```

```
            }
        }
        return null;
    }

    private static Fabric instance = new Fabric();

    public static Fabric getInstance() {
        return instance;
    }

    private Fabric(){};
}
```

Реализация Fabric использует один из простейших вариантов Одиночки.

Сводим воедино. Клиентский код:

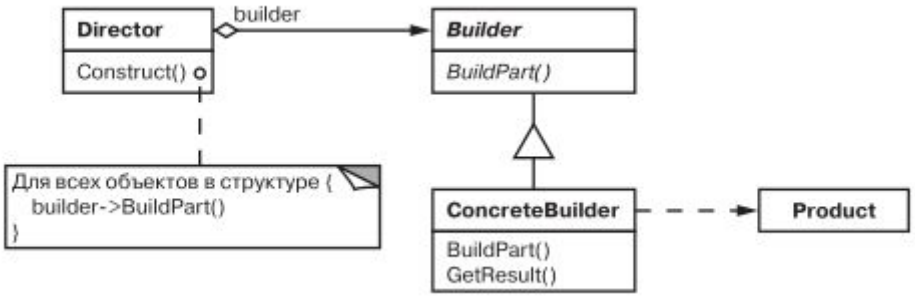
```
public Money getSupplierPrice(String supplierName, String article) throws Exception {
  // создать Абстрактную фабрику сервисов конкретного поставщика
  ExchangeFactory exchangeFactory = Fabric.getInstance().createFactory(supplierName);

  // создать фабрикой конкретного поставщика его провайдер некоторых услуг
  PriceProvider priceProvider = exchangeFactory.createPriceProvider();

  // получить услугу
  Money price = priceProvider.getPrice(article);

  ...
```

Builder // Строитель



Паттерн Строитель дает гибкость при построении сложных объектов, когда заранее не известны возможные опции построения, которые могут быть расширены без переделки кода собственно построителя.

Сегодня на практике чаще используется более упрощенная схема, а именно: интерфейс *Builder* становится реальным классом, а вариации ConcreteBuilder реализованы как множество функций, принимающих аргументы для «постройки» сложного продукта и возвращающих указатель на «строителя». Построение происходит при вызове цепочки методов, крайний из которых возвращает потребный клиенту продукт стройки. Данный паттерн часто используется в современном коде и вы легко его узнаете по цепочке вызовов.

Пример

Создание сообщения электронной почты. Есть обязательные опции (кому), а есть и множество факультативных, факт использования которых, скорее всего, будет различным в каждой конкретной ситуации.

```
package gof.creational.builder;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.internet.*;

/**
 *        Построитель сообщения электронной почты
 */

public class MimeMessageBuilder {
    private final MimeMessage message;

    public MimeMessageBuilder(Session session) {
        message = new MimeMessage(session);
    }

    public MimeMessageBuilder from(String address) throws MessagingException {
        message.setFrom(address);
        return this;
    }

    public MimeMessageBuilder to(String address) throws MessagingException {
        message.setRecipients(Message.RecipientType.TO, address);
        return this;
    }

    public MimeMessageBuilder cc(String address) throws MessagingException {
        message.setRecipients(Message.RecipientType.CC, address);
        return this;
    }

    public MimeMessageBuilder subject(String subject) throws MessagingException {
        message.setSubject(subject);
        return this;
    }

    public MimeMessageBuilder body(String body) throws MessagingException {
        message.setText(body);
        return this;
    }

    public MimeMessage build() {
        return message;
    }
}
```

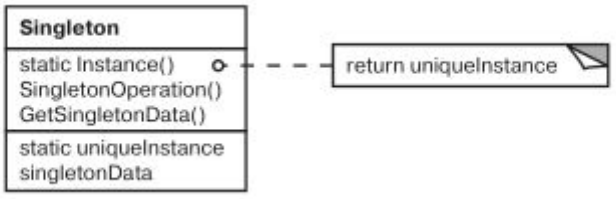
Клиентский код:

```
public class Client {
    public void sendMail(Session session) throws MessagingException {
        MimeMessage message = (new MimeMessageBuilder(session))
            .from("from@example.com")
            .to("to@example.com")
            .subject("hello")
            .build();

        // send it
        Transport.send(message);
    }
}
```

Интересный пример использования данного паттерна описан тут https://habrahabr.ru/post/244521/

Singleton // Одиночка



Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Класс Одиночка объявляет свой конструктор приватным, что, с одной стороны, не позволяет клиентскому коду самовольно порождать инстансы Одиночки, с другой – предоставляет статический метод getInstance, возвращающий инстанс объекта Одиночки, инстанциация которого полностью контролируется классом Одиночкой.

Преимущества:

- Только один объект данного класса.
- Глобальная область видимости, аналог глобальной переменной.
- В зависимости от реализации инстанциация «тяжелого» объекта может быть отложена по времени до момента первого использования – Lazy initialization.

Недостатки:

- Сложность **правильной** потокобезопасной реализации в многопоточном приложении.

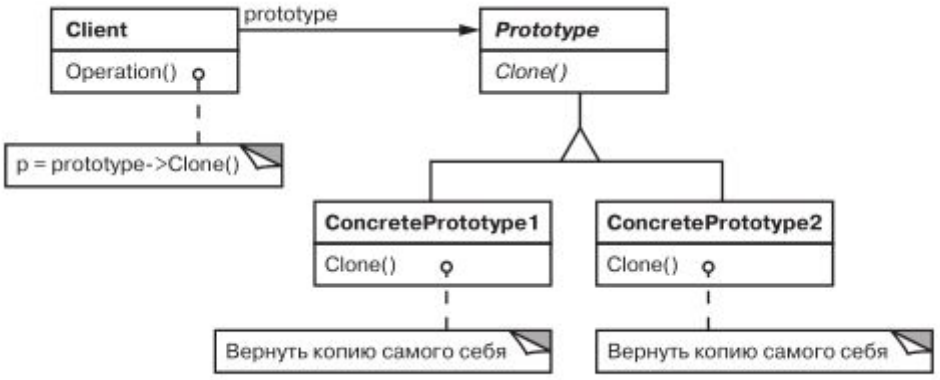
Пример

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        Singleton localInstance = instance;
        if (localInstance == null) {
            synchronized (Singleton.class) {
                localInstance = instance;
                if (localInstance == null) {
                    instance = localInstance = new Singleton();
                }
            }
        }
        return localInstance;
    }
}
```

Prototype // Прототип



Задает виды создаваемых объектов с помощью экземпляра Прототипа и новые объекты путем копирования этого Прототипа.

Проявление этого паттерна в реальной жизни – создание копии какого-либо объекта на основе имеющегося экземпляра. Например, в 1С есть часто используемая операция:


«Создать новый элемент копированием»

Для реализации этого паттерна для базового класса некоторой иерархии классов определяется операция создания новой копии объекта путем копирования – обычно, это метод clone().

В Java метод clone() определен в базовом классе Object. Поскольку операция копирования (например, потоки) может быть выполнена не для всех объектов, для применения этого метода по договоренности требуется, чтобы копируемый класс имплементировал маркер-интерфейс Cloneable. Однако это будет «поверхностное» копирование, реализуемое средствами JVM. В дальнейшем возникает два возможных сценария:

- Реализация метода clone(), учитывающая его особенности в каждом конкретном классе.
- Создание общего интерфейса: с одной стороны, имплементирующего интерфейс Cloneable, с другой – задающего набор общих для данной иерархии геттеров/сеттеров.

Практическое задание

1. Реализовать как минимум один Порождающий паттерн в своем приложении.

Дополнительные материалы

1. https://refactoring.guru/ru/design-patterns/creational-patterns

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://ru.wikipedia.org/wiki/Порождающие_шаблоны_проектирования
2. Приемы объектно-ориентированного проектирования. Паттерны проектирования., Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж., СПб.: Питер, 2015.
3. Шаблоны проектирования в Java, М. Гранд, М.: Новое знание, 2004.
4. Паттерны проектирования на платформе .NET. СПб.: Питер, 2015.
5. Образцы J2EE. Лучшие решения и стратегии проектирования, Алур Дипак, М.Лори, 2004.
6. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects Volume 2 Edition, Willy, 2000.
7. Кристофер Александер, Язык шаблонов … , М. Студия Артемия Лебедева, 2014.