Professor Hans Noodles

41 уровень

18.11.2019 16187 4

Паттерны проектирования: AbstractFactory

Привет! Сегодня мы продолжим изучать паттерны проектирования и поговорим об **абстрактной фабрике**.

Чем займемся на лекции:

- обсудим, что такое абстрактная фабрика и какую проблему данный паттерн решает;
- создадим каркас кроссплатформенного приложения для заказа кофе с пользовательским интерфейсом;
- изучим инструкцию по применению данного паттерна с диаграммой и кодом;
- в качестве бонуса, в лекции спрятана пасхалка, благодаря которой ты научишься определять имя операционной системы с помощью Java и в зависимости от результата выполнять то или иное действие.

Для полного понимания данного паттерна тебе необходимо хорошо разбираться в таких темах:

- наследование в Java;
- абстрактные классы и методы в Java.

Какие проблемы решает паттерн абстрактная фабрика?

Абстрактная фабрика, как и все фабричные паттерны, помогает нам правильно организовать создание новых объектов. С ее помощью мы управляем “выпуском” различных семейств взаимосвязанных объектов.

Различные семейства взаимосвязанных объектов…Что это? Не переживай: на практике все проще, чем может показаться.

Начнем с того, что может быть семейством взаимосвязанных объектов? Предположим, мы разрабатываем с тобой стратегию,

- кавалерия;
- лучники.

Эти типы боевых единиц связаны между собой, ведь они несут службу в одной армии. Мы можем сказать, что перечисленные выше категории — это семейство взаимосвязанных объектов. С этим разобрались.

Но паттерн абстрактная фабрика применяют для организации создания **различных** семейств взаимосвязанных объектов.

Здесь тоже ничего сложного. Продолжим пример со стратегией. В них, как правило, есть несколько различных противоборствующих сторон. У разных сторон боевые единицы могут существенно различаться внешне. Пехотинцы, всадники и лучники римской армии — не то же самое, что пехотинцы, всадники и лучники викингов.

В рамках стратегии, солдаты разных армий — это различные семейства взаимосвязанных объектов.

Было бы забавно, если по ошибке программиста в рядах римских пехотинцев разгуливал солдат во французском мундире времен Наполеона, с мушкетом наперевес.

Именно для решения такой проблемы нужен шаблон проектирования абстрактная фабрика. Нет, не проблемы конфузов путешествий во времени, а создания различных групп взаимосвязанных объектов.

Абстрактная фабрика предоставляет интерфейс создания всех имеющихся продуктов (объектов семейства). У абстрактной фабрики, как правило, есть несколько реализаций. Каждая из них отвечает за создание продуктов одной из вариаций.

В рамках стратегии у нас была бы абстрактная фабрика, создающая абстрактных пехотинцев, лучников и кавалеристов, а также реализации этой фабрики. Фабрика, создающая римских легионеров и, к примеру, фабрика, создающая воинов карфагена.

Абстракция — важнейший принцип данного паттерна. Клиенты фабрики работают с ней и с продуктами только через абстрактные интерфейсы. Поэтому можно не задумываться о том, каких воинов мы сейчас создаем, а передать эту обязанность какой-нибудь конкретной реализации абстрактной фабрики.

Продолжаем автоматизировать кофейню

На прошлой лекции мы изучили паттерн фабричный метод, с помощью которого нам удалось расширить кофейный бизнес и открыть несколько новых точек по продаже кофе.

Сегодня мы продолжим работу по модернизации нашего дела. С помощью паттерна абстрактная фабрика мы заложим фундамент для нового десктопного приложения для заказа кофе онлайн.

Когда пишем приложение для десктопа, мы всегда должны думать о кроссплатформенности. Наше приложение должно работать и на macOS, и на Windows (спойлер: Linux останется тебе в качестве домашнего задания).

Как будет выглядеть наше приложение? Довольно просто: это будет форма, которая состоит из текстового поля, поля выбора и кнопки. Если у тебя есть опыт использования разных операционных систем, ты точно заметил, что на винде кнопки отрисовываются не так, как на маке. Как, впрочем, и все остальное...

Итак, начнем. В роли семейств продуктов, как ты наверное уже понял, у нас будут выступать элементы графического интерфейса:

- кнопки;
- текстовые поля;
- поля для выбора.

Дисклеймер. Внутри каждого интерфейса мы могли бы определить методы, вроде `onClick`, `onValueChanged` или `onInputChanged`. Т.е. методы, которые позволят нам обрабатывать различные события (нажатие кнопки, ввод текста, выбор значения в поле выбора). Все это сознательно опущено, чтобы не перегружать пример и сделать его более наглядным для

Давай определим абстрактные интерфейсы для наших продуктов:

```
public interface Button {}
public interface Select {}
public interface TextField {}
```

Для каждой операционной системы мы должны создавать элементы интерфейса в стиле данной операционной системы. Мы пишем для Windows и MacOS.

Создадим реализации под Windows:

```
public class WindowsButton implements Button {
}

public class WindowsSelect implements Select {
}

public class WindowsTextField implements TextField {
}
```

Теперь то же самое для MacOS:

```
public class MacButton implements Button {
}

public class MacSelect implements Select {
}

public class MacTextField implements TextField {
}
```

Отлично. Теперь мы можем приступить к нашей абстрактной фабрике, которая будет создавать все существующие абстрактные типы продуктов:

```
public interface GUIFactory {

    Button createButton();
    TextField createTextField();
    Select createSelect();

}
```

Превосходно. Как видишь, пока что ничего сложного. Дальше все также просто. По аналогии с продуктами, создаем различные реализации нашей фабрики для каждой OS. Начнем с Windows:

```
public class WindowsGUIFactory implements GUIFactory {
    public WindowsGUIFactory() {
        System.out.println("Creating gui factory for Windows OS");
    }

    public Button createButton() {
```

```
    }

    public TextField createTextField() {
        System.out.println("Creating TextField for Windows OS");
        return new WindowsTextField();
    }

    public Select createSelect() {
        System.out.println("Creating Select for Windows OS");
        return new WindowsSelect();
    }
}
```

Вывод в консоль внутри методов и конструкторе добавлен для дальнейшей демонстрации работы.

Теперь для macOS:

```
public class MacGUIFactory implements GUIFactory {
    public MacGUIFactory() {
        System.out.println("Creating gui factory for macOS");
    }

    @Override
    public Button createButton() {
        System.out.println("Creating Button for macOS");
        return new MacButton();
    }

    @Override
    public TextField createTextField() {
        System.out.println("Creating TextField for macOS");
        return new MacTextField();
    }

    @Override
    public Select createSelect() {
        System.out.println("Creating Select for macOS");
        return new MacSelect();
    }
}
```

Заметь: каждый метод, согласно сигнатуре, возвращает абстрактный тип. Но внутри метода мы создаем конкретную реализацию продукта. Это единственное место, где мы контролируем создание конкретных экземпляров.

Теперь пришло время написать класс формы. Это Java-класс, поля которого являются элементами интерфейса:

```
public class OrderCoffeeForm {
    private final TextField customerNameTextField;
    private final Select coffeTypeSelect;
    private final Button orderButton;

    public OrderCoffeeForm(GUIFactory factory) {
        System.out.println("Creating order coffee form");
```

```
        coffeTypeSelect = factory.createSelect();
        orderButton = factory.createButton();
    }
}
```

В конструктор формы передается абстрактная фабрика, которая создает элементы интерфейса. Мы будем передавать в конструктор нужную реализацию фабрики, чтобы у нас создавались элементы интерфейса под ту или иную ОС.

```
public class Application {
    private OrderCoffeeForm orderCoffeeForm;

    public void drawOrderCoffeeForm() {
        // Определим имя операционной системы, получив значение системной проперти через System.getPr
        String osName = System.getProperty("os.name").toLowerCase();
        GUIFactory guiFactory;

        if (osName.startsWith("win")) { // Для windows
            guiFactory = new WindowsGUIFactory();
        } else if (osName.startsWith("mac")) { // Для mac
            guiFactory = new MacGUIFactory();
        } else {
            System.out.println("Unknown OS, can't draw form :( ");
            return;
        }
        orderCoffeeForm = new OrderCoffeeForm(guiFactory);
    }

    public static void main(String[] args) {
        Application application = new Application();
        application.drawOrderCoffeeForm();
    }
}
```

Если мы запустим приложение на винде, получим следующий вывод:

```
Creating gui factory for Windows OS
Creating order coffee form
Creating TextField for Windows OS
Creating Select for Windows OS
Creating Button for Windows OS
```

На маке вывод будет следующим:

```
Creating gui factory for macOS
Creating order coffee form
Creating TextField for macOS
Creating Select for macOS
Creating Button for macOS
```

```
Unknown OS, can't draw form :(
```

Ну а мы с тобой делаем следующий вывод. Мы написали каркас для приложения с графическим пользовательским интерфейсом, в котором создаются ровно те элементы интерфейса, которые уместны в данной ОС.

Повторим тезисно, что мы создали:

- Семейство продуктов: поле для ввода, поле выбора и кнопку.
- Различные реализации семейства данных продуктов, для Windows и macOS.
- Абстрактную фабрику, внутри которой определили интерфейс для создания наших продуктов.
- Две реализации нашей фабрики, каждая из которых отвечает за создание определенного семейства продуктов.
- Форму, Java-класс, полями которого являются абстрактные элементы интерфейса, которые инициализируются в конструкторе нужными значениями с помощью абстрактной фабрики.
- Класс приложения. Внутри него мы создаем форму, которой передаем в конструктор нужную реализацию нашей фабрики.

Итого: мы реализовали паттерн абстрактная фабрика.

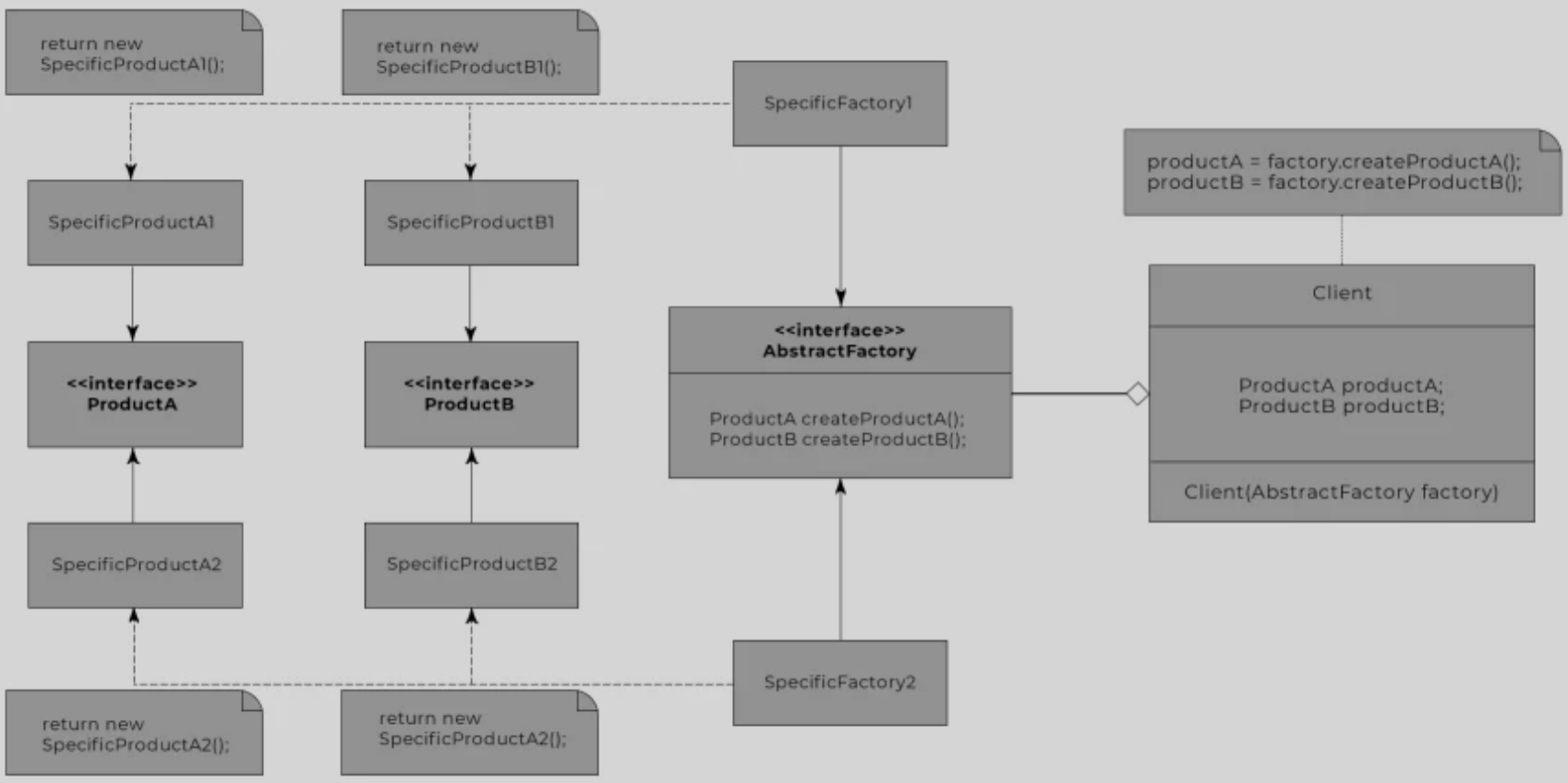
Абстрактная фабрика: инструкция по применению

Абстрактная фабрика — шаблон проектирования для управления созданием различных семейств продуктов без привязки к конкретным классам продуктов.

Применяя данный шаблон, необходимо:

1. Определить сами семейства продуктов. Предположим, у нас имеются их два:
   a. `SpecificProductA1`, `SpecificProductB1`
   b. `SpecificProductA2`, `SpecificProductB2`
2. Для каждого продукта внутри семейства определить абстрактный класс (интерфейс). В нашем случае это:
   a. `ProductA`
   b. `ProductB`
3. Внутри каждого семейства продуктов, каждый продукт должен реализовывать интерфейс, определенный на шаге 2.
4. Создать абстрактную фабрику, с методами create для каждого продукта, определенного на шаге 2. В нашем случае такими методами будут:
   a. `ProductA createProductA();`
   b. `ProductB createProductB();`
5. Создать реализации абстрактной фабрики так, чтобы каждая реализация управляла созданием продуктов одного семейства. Для этого внутри каждой реализации абстрактной фабрики необходимо реализовать все методы create, так, чтобы внутри них создавались и возвращались конкретные реализации продуктов.

Ниже представлена UML диаграмма, которая иллюстрирует описанную выше инструкцию:



Теперь напишем код по данной инструкции:

```
// Определим общие интерфейсы продуктов
public interface ProductA {}
public interface ProductB {}

// Создадим различные реализации (семейства) наших продуктов
public class SpecificProductA1 implements ProductA {}
public class SpecificProductB1 implements ProductB {}

public class SpecificProductA2 implements ProductA {}
public class SpecificProductB2 implements ProductB {}

// Создадим абстрактную фабрику
public interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}

// Создадим реализацию абстрактной фабрики для создания продуктов семейства 1
public class SpecificFactory1 implements AbstractFactory {

    @Override
    public ProductA createProductA() {
        return new SpecificProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new SpecificProductB1();
    }
}

```

```
34
35      @Override
36      public ProductA createProductA() {
37          return new SpecificProductA2();
38      }
39
40      @Override
41      public ProductB createProductB() {
42          return new SpecificProductB2();
43      }
44  }
```

Домашнее задание

+43

Комментарии (4) популярные новые старые

JavaCoder

Введите текст комментария

Denis Уровень 40, Москва, Russia 15 февраля, 12:21

спасибо!

Ответить 0

Сергей Java Developer в **Адвантум** 17 августа 2021, 20:28

Хорошее объяснение и внятные диаграммы

Ответить +1

Ekaterina Belousova Уровень 0 31 марта 2021, 13:18

Спасибо, друг, всё по полочкам разложил

Ответить +3

Валентин Кудинов Уровень 41, Самара, Россия 2 апреля 2020, 13:41

Лекция понятная Спасибо!
Нужно подправить опечатки в названии методов cretеProductA.

Ответить 0

НАЧАТЬ ОБУЧЕНИЕ

Курсы программирования
Курс Java
Помощь по задачам
Подписки
Задачи-игры

Пользователи
Статьи
Форум
Чат
Истории успеха
Активности

О нас
Контакты
Отзывы
FAQ
Поддержка

RUSH

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский