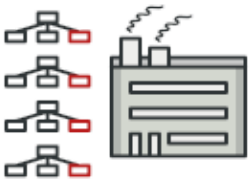

Java 抽象工厂模式讲解和代码示例

抽象工厂是一种创建型设计模式，它能创建一系列相关的对象，而无需指定其具体类。

抽象工厂定义了用于创建不同产品的接口，但将实际的创建工作留给了具体工厂类。每个工厂类型都对应一个特定的产品变体。

在创建产品时，客户端代码调用的是工厂对象的构建方法，而不是直接调用构造函数（`new`操作符）。由于一个工厂对应一种产品变体，因此它创建的所有产品都可相互兼容。

客户端代码仅通过其抽象接口与工厂和产品进行交互。该接口允许同一客户端代码与不同产品进行交互。你只需创建一个具体工厂类并将其传递给客户端代码即可。

> 如果你不清楚**工厂**、**工厂方法**和**抽象工厂**模式之间的区别，请参阅**工厂模式比较**。

进一步了解抽象工厂模式 →

在 Java 中使用模式

复杂度： ★★☆

流行度： ★★★

使用示例：抽象工厂模式在 Java 代码中很常见。许多框架和程序库会将它作为扩展和自定义其标准组件的一种方式。

以下是来自核心 Java 程序库的一些示例：

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

识别方法：我们可以通过方法来识别该模式——其会返回一个工厂对象。接下来，工厂将被用于创建特定的子组件。

导航

跨平台 GUI 组件系列及其创建方式

在本例中，按钮和复选框将被作为产品。它们有两个变体：macOS 版和 Windows 版。

抽象工厂定义了用于创建按钮和复选框的接口。而两个具体工厂都会返回同一变体的两个产品。

客户端代码使用抽象接口与工厂和产品进行交互。同样的代码能与依赖于不同工厂对象类型的多种产品变体进行交互。

buttons: 第一个产品层次结构

buttons/Button.java

```
package refactoring_guru.abstract_factory.example.buttons;

/**
 * Abstract Factory assumes that you have several families of products,
 * structured into separate class hierarchies (Button/Checkbox). All products of
 * the same family have the common interface.
 *
 * This is the common interface for buttons family.
 */
public interface Button {
    void paint();
}
```

buttons/MacOSButton.java

```
package refactoring_guru.abstract_factory.example.buttons;

/**
 * All products families have the same varieties (MacOS/Windows).
 *
 * This is a MacOS variant of a button.
 */
public class MacOSButton implements Button {

    @Override
    public void paint() {
        System.out.println("You have created MacOSButton.");
    }
}
```

buttons/WindowsButton.java

```
package refactoring_guru.abstract_factory.example.buttons;

/**
 * All products families have the same varieties (MacOS/Windows).
 *
 * This is another variant of a button.
 */
public class WindowsButton implements Button {

    @Override
    public void paint() {
        System.out.println("You have created WindowsButton.");
    }
}
```

checkboxes: 第二个产品层次结构

checkboxes/Checkbox.java

```
package refactoring_guru.abstract_factory.example.checkboxes;

/**
 * Checkboxes is the second product family. It has the same variants as buttons.
 */
public interface Checkbox {
    void paint();
}
```

checkboxes/MacOSCheckbox.java

```
package refactoring_guru.abstract_factory.example.checkboxes;

/**
 * All products families have the same varieties (MacOS/Windows).
 *
 * This is a variant of a checkbox.
 */
public class MacOSCheckbox implements Checkbox {

    @Override
    public void paint() {
        System.out.println("You have created MacOSCheckbox.");
    }
}
```

checkboxes/WindowsCheckbox.java

```
package refactoring_guru.abstract_factory.example.checkboxes;

/**
 * All products families have the same varieties (MacOS/Windows).
 *
 * This is another variant of a checkbox.
 */
public class WindowsCheckbox implements Checkbox {

    @Override
    public void paint() {
        System.out.println("You have created WindowsCheckbox.");
    }
}
```

factories

factories/GUIFactory.java: 抽象工厂

```
package refactoring_guru.abstract_factory.example.factories;

import refactoring_guru.abstract_factory.example.buttons.Button;
import refactoring_guru.abstract_factory.example.checkboxes.Checkbox;

/**
 * Abstract factory knows about all (abstract) product types.
 */
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

factories/MacOSFactory.java: 具体工厂（macOS）

```
package refactoring_guru.abstract_factory.example.factories;

import refactoring_guru.abstract_factory.example.buttons.Button;
import refactoring_guru.abstract_factory.example.buttons.MacOSButton;
import refactoring_guru.abstract_factory.example.checkboxes.Checkbox;
import refactoring_guru.abstract_factory.example.checkboxes.MacOSCheckbox;

/**
 * Each concrete factory extends basic factory and responsible for creating
```

```
 * products of a single variety.
 */
public class MacOSFactory implements GUIFactory {

    @Override
    public Button createButton() {
        return new MacOSButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new MacOSCheckbox();
    }
}
```

factories/WindowsFactory.java: 具体工厂（Windows）

```
package refactoring_guru.abstract_factory.example.factories;

import refactoring_guru.abstract_factory.example.buttons.Button;
import refactoring_guru.abstract_factory.example.buttons.WindowsButton;
import refactoring_guru.abstract_factory.example.checkboxes.Checkbox;
import refactoring_guru.abstract_factory.example.checkboxes.WindowsCheckbox;

/**
 * Each concrete factory extends basic factory and responsible for creating
 * products of a single variety.
 */
public class WindowsFactory implements GUIFactory {

    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}
```

app

app/Application.java: 客户端代码

```
package refactoring_guru.abstract_factory.example.app;

import refactoring_guru.abstract_factory.example.buttons.Button;
import refactoring_guru.abstract_factory.example.checkboxes.Checkbox;
import refactoring_guru.abstract_factory.example.factories.GUIFactory;

/**
 * Factory users don't care which concrete factory they use since they work with
 * factories and products through abstract interfaces.
 */
public class Application {
    private Button button;
    private Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }

    public void paint() {
        button.paint();
        checkbox.paint();
    }
}
```

Demo.java: 程序配置

```
package refactoring_guru.abstract_factory.example;

import refactoring_guru.abstract_factory.example.app.Application;
import refactoring_guru.abstract_factory.example.factories.GUIFactory;
import refactoring_guru.abstract_factory.example.factories.MacOSFactory;
import refactoring_guru.abstract_factory.example.factories.WindowsFactory;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {

    /**
     * Application picks the factory type and creates it in run time (usually at
     * initialization stage), depending on the configuration or environment
     * variables.
     */
    private static Application configureApplication() {
        Application app;
        GUIFactory factory;
        String osName = System.getProperty("os.name").toLowerCase();
        if (osName.contains("mac")) {
            factory = new MacOSFactory();
```

```
            app = new Application(factory);
        } else {
            factory = new WindowsFactory();
            app = new Application(factory);
        }
        return app;
    }

    public static void main(String[] args) {
        Application app = configureApplication();
        app.paint();
    }
}
```

OutputDemo.txt: 执行结果

```
You create WindowsButton.
You created WindowsCheckbox.
```

继续阅读

Java 生成器模式讲解和代码示例 →

返回

← Java 设计模式

抽象工厂在其他编程语言中的实现

Khmelnitske shosse 19 / 27, Kamianets-Podilskyi, 乌克兰, 32305

Email: support@refactoring.guru

图片作者： Dmitry Zhart