



The factory pattern



START





Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040



The factory pattern



START





Definition:

The factory_pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The goal of the factory pattern is to abstract the process of object creation and provide a simple way for clients to obtain new objects without needing to know the details of how they are created.

.

Types of Factory Patterns

Factory Method: The Factory Method pattern is a creational design pattern that provides an interface for creating objects, but delegates the responsibility of creating objects to subclasses. This pattern allows you to create objects without specifying the exact class of object that will be created, and it provides a way for subclasses to determine which class of object to create.

Abstract Factory: The Abstract Factory pattern is also a creational design pattern that provides an interface for creating families of related objects, without specifying their concrete classes. This pattern provides a way to create objects that are related to each other, and it ensures that the objects created by a factory are compatible with each other.

Implementation



```

1  #include <iostream>
2
3  // Shape Interface
4  class Shape {
5  public:
6      virtual void draw() = 0;
7  };
8
9  // Concrete Shapes
10 class Circle : public Shape {
11 public:
12     void draw() {
13         std::cout << "Drawing a Circle" << std::endl;
14     }
15 };
16
17 class Rectangle : public Shape {
18 public:
19     void draw() {
20         std::cout << "Drawing a Rectangle" << std::endl;
21     }
22 };
23
24 class Triangle : public Shape {
25 public:
26     void draw() {
27         std::cout << "Drawing a Triangle" << std::endl;
28     }
29 };
30
31 // ShapeFactory Interface
32 class ShapeFactory {
33 public:
34     virtual Shape* createShape() = 0;
35 };
36
37 // Concrete Shape Factories
38 class CircleFactory : public ShapeFactory {
39 public:
40     Shape* createShape() {
41         return new Circle();
42     }
43 };
44

```

```

45 class RectangleFactory : public ShapeFactory {
46 public:
47     Shape* createShape() {
48         return new Rectangle();
49     }
50 };
51
52 class TriangleFactory : public ShapeFactory {
53 public:
54     Shape* createShape() {
55         return new Triangle();
56     }
57 };
58
59 // Abstract GUI Factory Interface
60 class GUIFactory {
61 public:
62     virtual Button* createButton() = 0;
63     virtual Menu* createMenu() = 0;
64     virtual TextBox* createTextBox() = 0;
65 };
66
67 // Concrete GUI Factories
68 class WindowsGUIFactory : public GUIFactory {
69 public:
70     Button* createButton() {
71         return new WindowsButton();
72     }
73
74     Menu* createMenu() {
75         return new WindowsMenu();
76     }
77
78     TextBox* createTextBox() {
79         return new WindowsTextBox();
80     }
81 };
82

```

```

83 class MacGUIFactory : public GUIFactory {
84 public:
85     Button* createButton() {
86         return new MacButton();
87     }
88
89     Menu* createMenu() {
90         return new MacMenu();
91     }
92
93     TextBox* createTextBox() {
94         return new MacTextBox();
95     }
96 };
97
98 class LinuxGUIFactory : public GUIFactory {
99 public:
100     Button* createButton() {
101         return new LinuxButton();
102     }
103
104     Menu* createMenu() {
105         return new LinuxMenu();
106     }
107
108     TextBox* createTextBox() {
109         return new LinuxTextBox();
110     }
111 };
112
113 // GUI Components
114 class Button {
115 public:
116     virtual void paint() = 0;
117 };
118
119 class WindowsButton : public Button {
120 public:
121     void paint() {
122         std::cout << "Painting a Windows Button" << std::endl;
123     }
124 };
125

```

```

126 class MacButton : public Button {
127 public:
128     void paint() {
129         std::cout << "Painting a Mac Button" << std::endl;
130     }
131 };
132
133 class LinuxButton : public Button {
134 public:
135     void paint() {
136         std::cout << "Painting a Linux Button" << std::endl;
137     }
138 };
139
140 class Menu {
141 public:
142     virtual void paint() = 0;
143 };
144
145 class WindowsMenu : public Menu {
146 public:
147     void paint() {
148         std::cout << "Painting a Windows Menu" << std::endl;
149     }
150 };
151
152 class MacMenu : public Menu {
153 public:
154     void paint() {
155         std::cout << "Painting a Mac Menu" << std::endl;
156     }
157 };
158
159 class LinuxMenu : public Menu {
160 public:
161     void paint() {
162         std::cout << "Painting a Linux Menu" << std::endl;
163     }
164 };
165

```



```

166 class TextBox {
167 public:
168     virtual void paint() = 0;
169 };
170
171 class WindowsTextBox : public TextBox {
172 public:
173     void paint() {
174         std::cout << "Painting a Windows TextBox" << std::endl;
175     }
176 };
177
178 class MacTextBox : public TextBox {
179 public:
180     void paint() {
181         std::cout << "Painting a Mac TextBox" << std::endl;
182     }
183 };
184
185 class LinuxTextBox : public TextBox {
186 public:
187     void paint() {
188         std::cout << "Painting a Linux TextBox" << std::endl;
189     }
190 };
191

```

```

// Client Code
int main() {
    // Using Factory Method
    ShapeFactory* shapeFactory = new CircleFactory();
    Shape* shape = shapeFactory->createShape();
    shape->draw();

    // Using Abstract Factory
    GUIFactory* guiFactory = nullptr;
#ifdef _WIN32
    guiFactory = new WindowsGUIFactory();
#elif __APPLE__
    guiFactory = new MacGUIFactory();
#elif __linux__
    guiFactory = new LinuxGUIFactory();
#endif

    Button* button = guiFactory->createButton();
    Menu* menu = guiFactory->createMenu();
    TextBox* textBox = guiFactory->createTextBox();

    button->paint();
    menu->paint();
    textBox->paint();

    return 0;
}

```



Advantages

1 Encapsulates object creation



3 Reusability

2 Decouples clients from concrete classes

4 Simplifies testing





Disadvantages

1 Add runtimes overhead

3 May violate the Open-Closed Principle



2 Complexity

4 Difficult to refactor





THANK YOU!