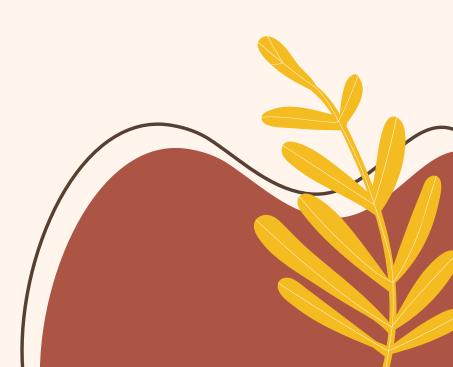
The factory pattern



START

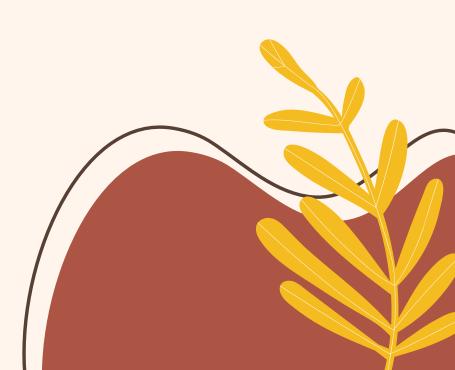


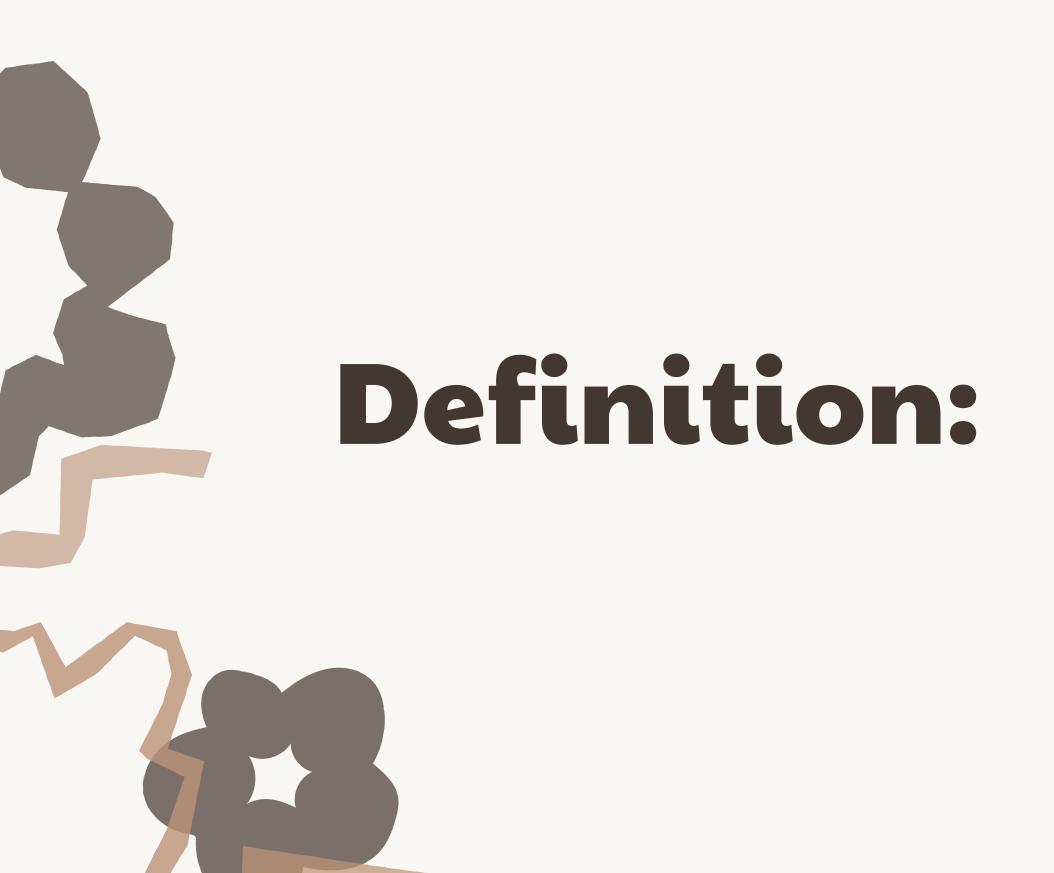


The factory pattern



START



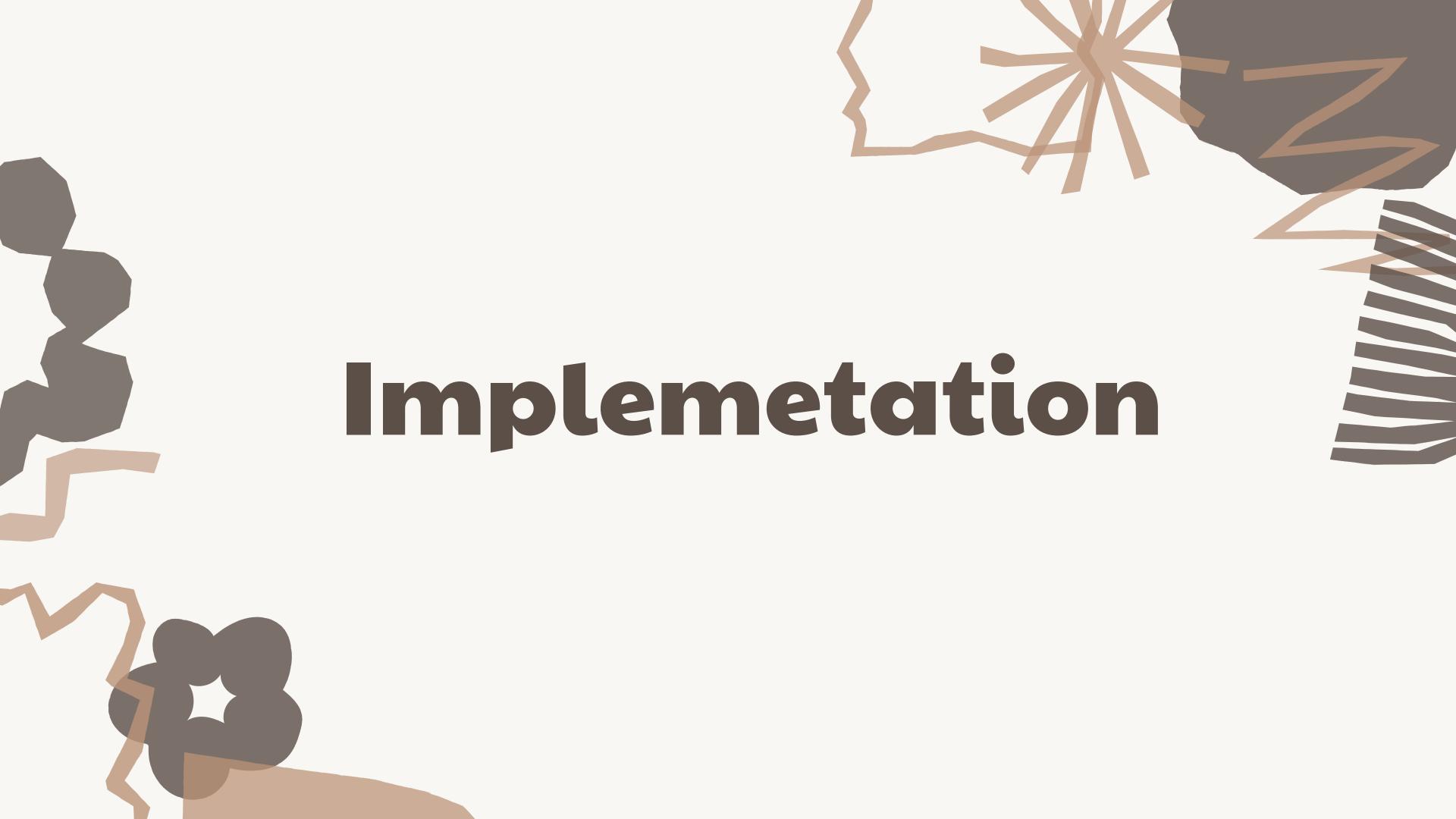


The factory pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The goal of the factory pattern is to abstract the process of object creation and provide a simple way for clients 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.



```
#include <iostream>
3 // Shape Interface
4 class Shape {
5 public:
   virtual void draw() = 0;
   };
9 // Concrete Shapes
10 class Circle : public Shape {
11 public:
12 void draw() {
13 std::cout << "Drawing a Circle" << std::endl;</pre>
14
15 };
17 class Rectangle : public Shape {
18 public:
19 void draw() {
22 };
24 class Triangle : public Shape {
25 public:
26 void draw() {
28
29 };
31 // ShapeFactory Interface
32 class ShapeFactory {
33 public:
  virtual Shape* createShape() = 0;
35 };
37 // Concrete Shape Factories
38 class CircleFactory : public ShapeFactory {
39 public:
      Shape* createShape() {
         return new Circle();
   };
```

```
class RectangleFactory : public ShapeFactory {
    public:
47
        Shape* createShape() {
            return new Rectangle();
    };
50
51
    class TriangleFactory : public ShapeFactory {
52
    public:
54
        Shape* createShape() {
            return new Triangle();
56
    };
57
    // Abstract GUI Factory Interface
    class GUIFactory {
61
    public:
        virtual Button* createButton() = 0;
62
63
       virtual Menu* createMenu() = 0;
64
        virtual TextBox* createTextBox() = 0;
    };
66
     // Concrete GUI Factories
    class WindowsGUIFactory : public GUIFactory {
    public:
70
        Button* createButton() {
71
            return new WindowsButton();
72
73
        Menu* createMenu() {
74
            return new WindowsMenu();
76
77
78
        TextBox* createTextBox() {
            return new WindowsTextBox();
80
81
    };
```

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

```
126   class MacButton : public Button {
127 public:
          void paint() {
             std::cout << "Painting a Mac Button" << std::endl;</pre>
131 };
      class LinuxButton : public Button {
         void paint() {
             std::cout << "Painting a Linux Button" << std::endl;</pre>
138 };
140 class Menu {
       virtual void paint() = 0;
143 };
145 class WindowsMenu : public Menu {
         void paint() {
148
             std::cout << "Painting a Windows Menu" << std::endl;</pre>
150 };
152 class MacMenu : public Menu {
         void paint() {
             std::cout<< "Painting a Mac Menu" << std::endl;</pre>
157 };
159 class LinuxMenu : public Menu {
160 public:
         void paint() {
             std::cout << "Painting a Linux Menu" << std::endl;</pre>
164 };
```

```
class TextBox {
      public:
167
          virtual void paint() = 0;
168
      };
169
170
      class WindowsTextBox : public TextBox {
171
      public:
172
173
          void paint() {
               std::cout << "Painting a Windows TextBox" << std::endl;</pre>
174
175
      };
176
177
      class MacTextBox : public TextBox {
178
      public:
179
          void paint() {
180
               std::cout << "Painting a Mac TextBox" << std::endl;</pre>
181
182
      };
183
184
      class LinuxTextBox : public TextBox {
185
      public:
186
          void paint() {
187
               std::cout << "Painting a Linux TextBox" << std::endl;</pre>
188
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 <u>linux</u>
       guiFactory = new LinuxGUIFactory();
    #endif
    Button* button = guiFactory->createButton();
   Menu* menu = guiFactory->createMenu();
    TextBox* textBox = guiFactory->createTextBox();
    button->paint();
    menu->paint();
    textBox->paint();
    return 0;
```



Disavantages

Add runtimes overhead

3 May violate the Open-Closed Principle

2 Complexity

Difficult to refactor

