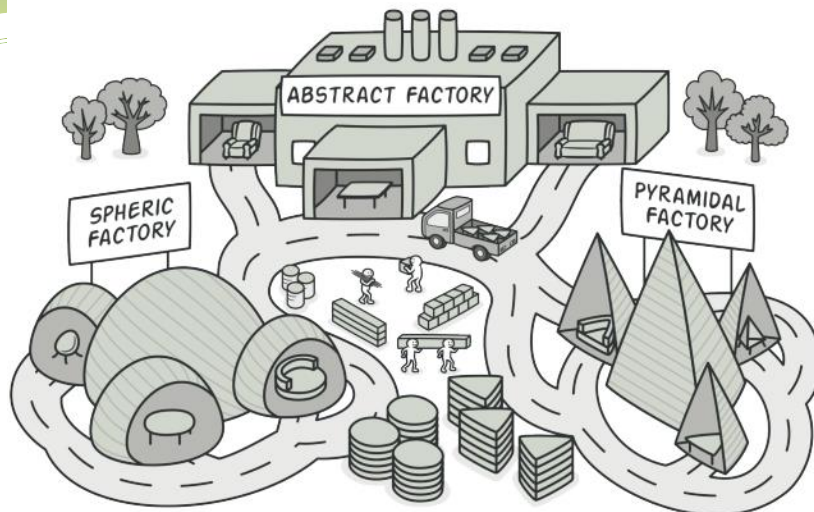


Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif



ABSTRACT FACTORY



Abstract Factory

- **Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

3












Problem

Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

- A family of related products, say: Chair + Sofa + CoffeeTable .
- Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: IKEA , VictorianStyle , ArtDeco .

4

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

Product families and their variants.

5

Problem

You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.



An IKEA sofa doesn't match a Victorian-style chairs.

6

Problem

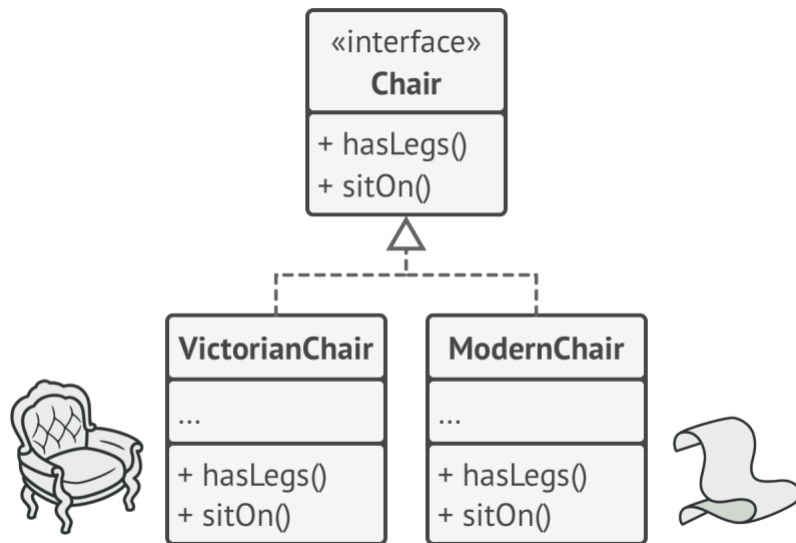
Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.

7

Solution

The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the `Chair` interface; all coffee table variants can implement the `CoffeeTable` interface, and so on.

8



All variants of the same object must be moved to a single class hierarchy.

9

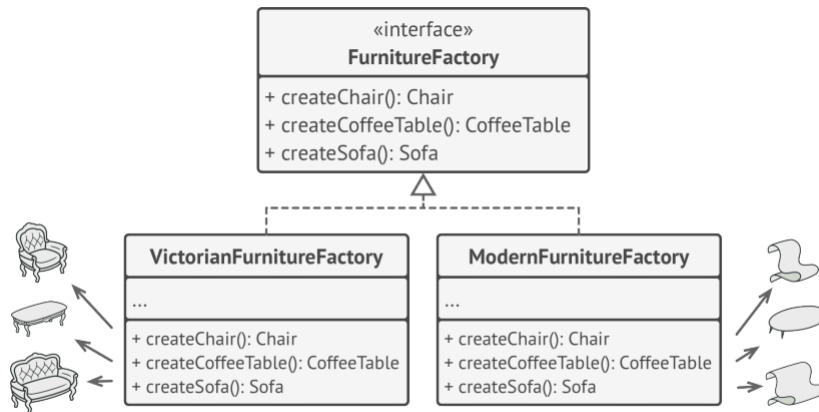
Solution

The next move is to declare the **AbstractFactory interface** with a list of creation methods for all products that are part of the product family (for example, `createChair`, `createSofa` and `createCoffeeTable`). These methods must return **abstract product** types represented by the interfaces we extracted previously: **Chair**, **Sofa**, **CoffeeTable** and so on.

Now, how about the product variants? For each variant of a product family, we create a **separate factory class** based on the **AbstractFactory** interface. A factory is a class that returns products of a particular kind. For example, the **IKEAFactory** can only create **IKEAChair**, **IKEASofa** and **IKEACoffeeTable** objects.

10

Solution

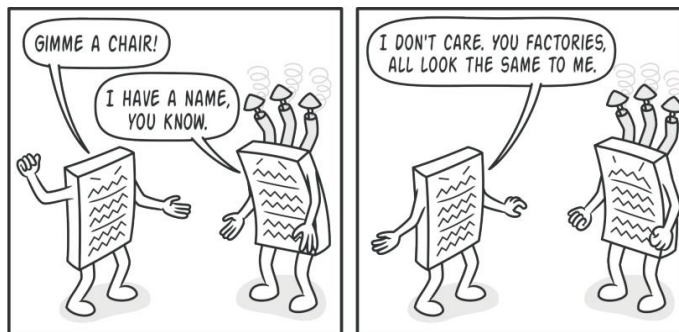


Each concrete factory corresponds to a specific product variant.

11

Solution

The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.



The client shouldn't care about the concrete class of the factory it works with.

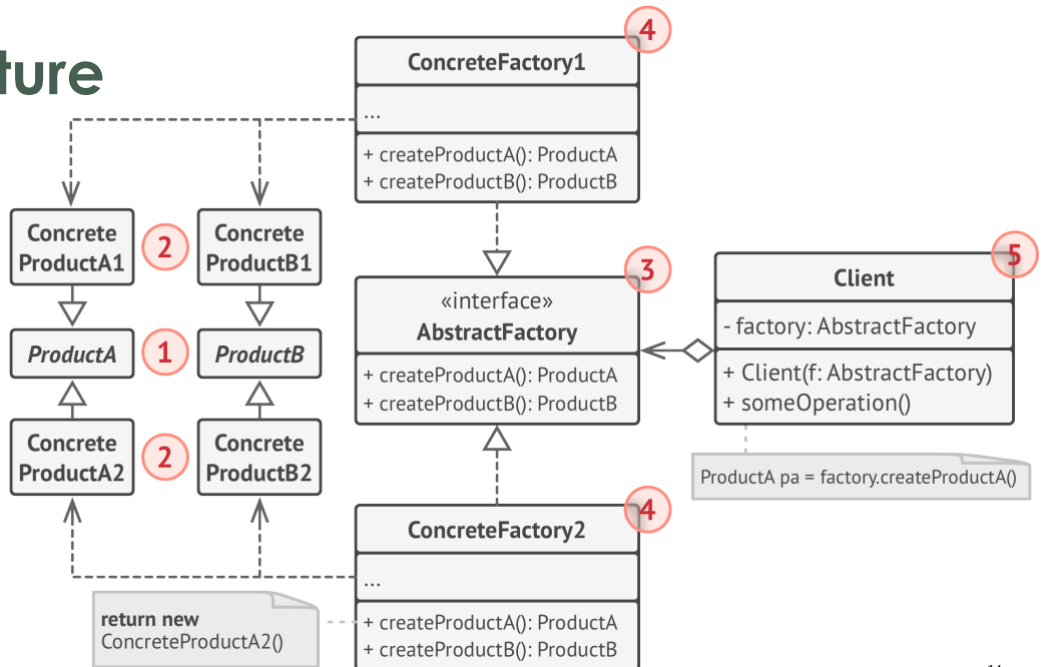
12

- **Solution**


One more thing left to clarify: if the client is only exposed to the abstract interfaces, **what creates the actual factory objects?** Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.

13


Structure



14

- 
- **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
 - **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).
 - The **Abstract Factory interface** declares a set of methods for creating each of the abstract products.
 - **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

15

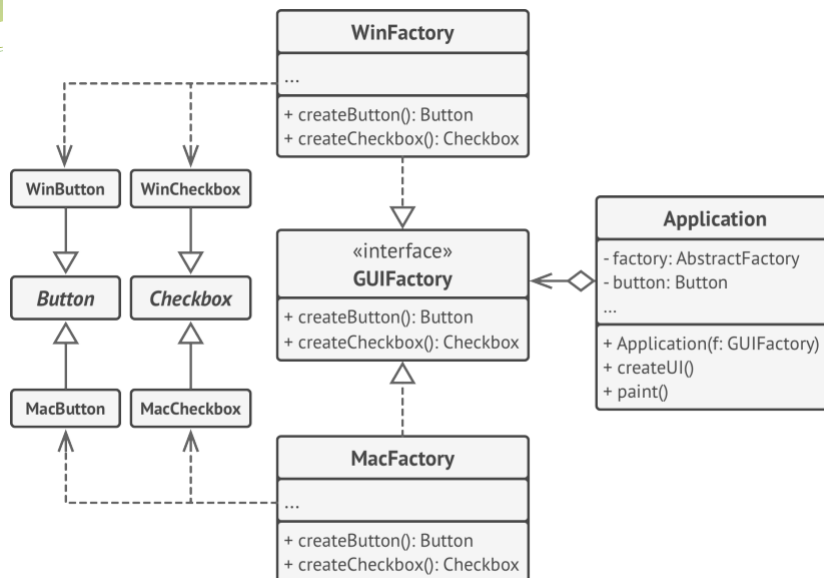
- 
- Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding abstract products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory. The Client can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

16

Pseudocode


- This example illustrates how the Abstract Factory pattern can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes, while keeping all created elements consistent with a selected operating system.

17




The cross-platform UI classes example.

18



```
1 // The abstract factory interface declares a set of methods that
2 // return different abstract products. These products are called
3 // a family and are related by a high-level theme or concept.
4 // Products of one family are usually able to collaborate
   among
5 // themselves. A family of products may have several variants,
6 // but the products of one variant are incompatible with the
7 // products of another variant.
8 interface GUIFactory is
9 method createButton():Button
10 method createCheckbox():Checkbox
11
12
```

19



```
13 // Concrete factories produce a family of products that belong
14 // to a single variant. The factory guarantees that the
15 // resulting products are compatible. Signatures of the concrete
16 // factory's methods return an abstract product, while inside
17 // the method a concrete product is instantiated.
18 class WinFactory implements GUIFactory is
19 method createButton():Button is
20 return new WinButton()
21 method createCheckbox():Checkbox is
22 return new WinCheckbox()
23
24 // Each concrete factory has a corresponding product variant.
25 class MacFactory implements GUIFactory is
26 method createButton():Button is
27 return new MacButton()
28 method createCheckbox():Checkbox is
29 return new MacCheckbox()
30
31
32
```


20

```
33 // Each distinct product of a product family should have a base
34 // interface. All variants of the product must implement this
35 // interface.
36 interface Button is
37 method paint()
38
39 // Concrete products are created by corresponding concrete
40 // factories.
41 class WinButton implements Button is
42 method paint() is
43 // Render a button in Windows style.
44
45 class MacButton implements Button is
46 method paint() is
47 // Render a button in macOS style.
48
```

21


```
49 // Here's the base interface of another product. All products
50 // can interact with each other, but proper interaction is
51 // possible only between products of the same concrete variant.
52 interface Checkbox is
53 method paint()
54
55 class WinCheckbox implements Checkbox is
56 method paint() is
57 // Render a checkbox in Windows style.
58
59 class MacCheckbox implements Checkbox is
60 method paint() is
61 // Render a checkbox in macOS style.
62
63
64
```

22



```
65 // The client code works with factories and products only
66 // through abstract types: GUIFactory, Button and Checkbox. This
67 // lets you pass any factory or product subclass to the client
68 // code without breaking it.
69 class Application is
70 private field button: Button
71 constructor Application(factory: GUIFactory) is
72 this.factory = factory
73 method createUI() is
74 this.button = factory.createButton()
75 method paint() is
76 button.paint()
77
78
```

23



```
79 // The application picks the factory type depending on the
80 // current configuration or environment settings and creates it
81 // at runtime (usually at the initialization stage).
82 class ApplicationConfigurator is
83 method main() is
84 config = readApplicationConfigFile()
85
86 if (config.OS == "Windows") then
87 factory = new WinFactory()
88 else if (config.OS == "Mac") then
89 factory = new MacFactory()
90 else
91 throw new Exception("Error! Unknown operating system.")
92
93 Application app = new Application(factory)
```

24



Java Code Example

25



Applicability

- Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility

26



How to Implement

1. Map out a matrix of distinct product types versus variants of these products.
2. Declare abstract product interfaces for all product types. Then make all concrete product classes implement these interfaces.
3. Declare the abstract factory interface with a set of creation methods for all abstract products.
4. Implement a set of concrete factory classes, one for each product variant.

27



How to Implement

5. Create factory initialization code somewhere in the app. It should instantiate one of the concrete factory classes, depending on the application configuration or the current environment. Pass this factory object to all classes that construct products.
6. Scan through the code and find all direct calls to product constructors. Replace them with calls to the appropriate creation method on the factory object.

28

Pros and Cons

- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.
- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

29

Review

30