Learn about Design Patterns in our eBook

Abstract Factory

Intent

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

Problem

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

1. A family of related products, say `Chair` + `Sofa` + `CoffeeTable`.
2. Several variants of this family. For example, products `Chair` + `Sofa` + `CoffeeTable` are available in these variants: `Modern`, `Victorian`, `ArtDeco`.

Product families and their variants.

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.

A Modern style sofa doesn't match Victorian style chairs.

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.

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.

All variants of the same object must have similar interfaces.

The next move is to declare the *Abstract Factory*—an 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.

Each concrete factory corresponds to a specific product variant.

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 `ModernFurnitureFactory` can only create `ModernChair`, `ModernSofa` and `ModernCoffeeTable` objects.

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 your business logic receives, 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.

Say the client wants a factory to produce a chair. The client doesn't have to be aware of the factory's class, nor does it matter what kind of chair it gets. Whether it's a Modern model or a Victorian-style chair, the client must treat all chairs in the same manner, using the abstract chair interface. With this approach, the only thing that the client knows about the chair is that it implements the `Chair` method. In some way, this interface variant of the chair is expected. It's always match the type of sofa or coffee table produced by the same factory object.

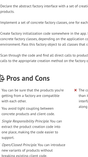
There's one more thing left to clarify: if the client is only exposed to the abstract interfaces, what creates the actual factory objects? Usually the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration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

Structure

1. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/table/sofa) can be implemented in a different variant. Different implementations of a product should be able to work with the same client code.
2. **Abstract Product** declares distinct parts of a product family. Concrete products are all variants of the same product family. This family of products may have several variants, but the products of one variant are compatible with the products of another variant.
3. **Concrete Factory** produces objects of a specific variant. As it returns abstract products, the concrete factory doesn't care which variant of product is created. For example, the `ModernFurnitureFactory` can produce `ModernChair`, `ModernSofa` and `ModernCoffeeTable`.
4. **Abstract Factory** lists creation methods for all products of the family. Every method returns an abstract product. The client code can then work with any variant of a product family, but it won't be able to create the actual concrete product. For example, a client can ask a factory to create a chair, but it will receive a `ModernChair` when it uses the `ModernFurnitureFactory`.
5. **Client** works with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that your business logic receives, as well as the product variant that the client code receives, without breaking the actual client code.

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.



The cross-platform UI classes diagram.

The same UI elements in a cross-platform application are expected to behave similarly but look slightly different in different operating systems. For example, buttons in Windows stay in their own windows with the style of the current operating system. You wouldn't want your program to render macOS controls when it's executed in Windows.

The Abstract Factory interface declares a set of creation methods that the client code can use to create different kinds of UI elements. Each factory corresponds to and builds elements specific to operating systems and creates that UI elements that match that particular OS.

It works like this: when an application launches, it checks the type of the current operating system. The app uses this information to create a factory object from a class that matches the operating system. Later, when the app needs UI elements, it asks the factory to create UI elements. This prevents the wrong elements from being created.

With this approach, the client code doesn't depend on concrete classes of factories and UI elements as long as it works with these objects via their abstract interfaces. This also lets the client code support further factories or UI elements that you might add later in the future.

As a result, you don't need to modify the client code each time you add a new variation of UI elements or a new variation of them. The client code runs a new factory class that produces these elements and slightly modifies the app's initialization code so it selects that class when appropriate.

```
// The abstract factory interface declares a set of methods that
// return different abstract products. These products are called
// family and are related to a high-level theme or concept. Products
// belonging of one family are usually able to collaborate among
// themselves. A family of products may have several variants,
// but the products of one variant are incompatible with the
// products of another variant.
interface GUIFactory {
    method createButton(): Button
    method createCheckbox(): Checkbox

// Concrete factories produce a family of products that belong
// to a single variant. The factory guarantees that the
// resulting products are compatible. Signature of the concrete
// factory's methods return an abstract product, while inside
// the method a concrete product is instantiated.
class WinFactory implements GUIFactory {
    method createButton(): Button Do
        return new WinButton()
    method createCheckbox(): Checkbox Do
        return new WinCheckbox()

// Each concrete family has a corresponding product variant.
class MacFactory implements GUIFactory {
    method createButton(): Button Do
        return new MacButton()
    method createCheckbox(): Checkbox Do
        return new MacCheckbox()

// Each distinct product of a product family should have a base
// interface. All variants of the product must implement this
// interface.
interface Button {
    method paint()

// Concrete products are created by corresponding concrete
// factories.
class WinButton implements Button Do
    method paint() Do
        // Render a button in Windows style.

class MacButton implements Button Do
    method paint() Do
        // Render a button in macOS style.

// Here's the base interface of another product. All products
// can interact with each other but proper interaction is
// possible only between products of the same concrete variant.
interface Checkbox Do
    method paint()

class WinCheckbox implements Checkbox Do
    method paint() Do
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox Do
    method paint() Do
        // Render a checkbox in macOS style.

// The client code works with factories and products only
// through abstract type: Factory and Button. This lets you pass
// into you app any factory or product variation to the client
// code without breaking it.
class Application {
    private field factory: GUIFactory
    private field button: Button
    // Usually, the application gets a factory. Do
    // this.factory = factory
    method f(f: Do
        this.factory = f
    method applicationLogic()
    method paint() Do
        button.paint()

// The application picks the factory type depending on the
// current environment settings or configuration files.
// If initialization of the initialization stages
// is complex, it may be placed in a
// separate initialization class.
class ApplicationConfigurator {
    method main() Do
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new RuntimeException("Error: Unknown operating system.")

        Application app = new Application(factory)
```

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.**
- The Abstract Factory provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

- **Consider implementing the Abstract Factory when you have a class with a set of Factory Methods that blur its primary responsibility.**
- In a well-designed program each class is responsible only for one thing. When a class deals with several product families, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown Abstract Factory implementation.

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.
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.

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 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.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype** or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.
- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.
- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Code Examples

Extra Content

- Read our **Factory Comparison** to learn more about the differences between various factory patterns and concepts.

Support our free website and own the eBook!

- 6.5 design patterns and 6 principles explained in depth
- 400 well-structured, easy to read, page-free pages
- 223 clear and helpful illustrations and diagrams
- An archive with code examples in 11 languages
- 90 devices supported: PDF, EPUB/MOBI/AZW3 formats

Learn more

RETURN
Factory Method
READ NEXT
Factory Comparison

© 2014-2024 Refactoring.Guru. All rights reserved.
Illustrations by Dmitry Zhart
Terms & Conditions Privacy Policy Content Usage Policy About us

Ukrainian office:
PSB Office Solutions
Ukraine, 03150
Nauchnaya 10/22
Email: support@refactoring.guru

Spanish office:
Calle Jorge Juan 19
Delegación Tacuba, 11590
Mexico City, Mexico
Email: support@refactoring.guru