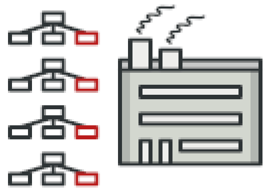
/ Design Patterns / Abstract Factory / C++



Abstract Factory in C++

Abstract Factory is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.

Abstract Factory defines an interface for creating all distinct products but leaves the actual product creation to concrete factory classes. Each factory type corresponds to a certain product variety.

The client code calls the creation methods of a factory object instead of creating products directly with a constructor call (`new` operator). Since a factory corresponds to a single product variant, all its products will be compatible.

Client code works with factories and products only through their abstract interfaces. This lets the client code work with any product variants, created by the factory object. You just create a new concrete factory class and pass it to the client code.

If you can't figure out the difference between various factory patterns and concepts, then read our **Factory Comparison**.

Learn more about Abstract Factory →

Usage of the pattern in C++

Complexity: ★★☆

Popularity: ★★★

Usage examples: The Abstract Factory pattern is pretty common in C++ code. Many frameworks and libraries use it to provide a way to extend and customize their standard components.

Identification: The pattern is easy to recognize by methods, which return a factory object. Then, the factory is used for creating specific sub-components.

Navigation

Intro

Conceptual Example
main
Output

Conceptual Example

This example illustrates the structure of the **Abstract Factory** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
/**
 * Each distinct product of a product family should have a base interface. All
 * variants of the product must implement this interface.
 */
class AbstractProductA {
 public:
  virtual ~AbstractProductA(){};
  virtual std::string UsefulFunctionA() const = 0;
};

/**
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductA1 : public AbstractProductA {
 public:
  std::string UsefulFunctionA() const override {
    return "The result of the product A1.";
  }
};

class ConcreteProductA2 : public AbstractProductA {
  std::string UsefulFunctionA() const override {
    return "The result of the product A2.";
  }
};

/**
 * Here's the 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.
 */
class AbstractProductB {
  /**
   * Product B is able to do its own thing...
   */
 public:
  virtual ~AbstractProductB(){};
  virtual std::string UsefulFunctionB() const = 0;
  /**
   * ...but it also can collaborate with the ProductA.
   *
   * The Abstract Factory makes sure that all products it creates are of the
   * same variant and thus, compatible.
   */
  virtual std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator) const = 
};

/**
```

```
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductB1 : public AbstractProductB {
 public:
  std::string UsefulFunctionB() const override {
    return "The result of the product B1.";
  }
  /**
   * The variant, Product B1, is only able to work correctly with the variant,
   * Product A1. Nevertheless, it accepts any instance of AbstractProductA as an
   * argument.
   */
  std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator) const override {
    const std::string result = collaborator.UsefulFunctionA();
    return "The result of the B1 collaborating with ( " + result + " )";
  }
};

class ConcreteProductB2 : public AbstractProductB {
 public:
  std::string UsefulFunctionB() const override {
    return "The result of the product B2.";
  }
  /**
   * The variant, Product B2, is only able to work correctly with the variant,
   * Product A2. Nevertheless, it accepts any instance of AbstractProductA as an
   * argument.
   */
  std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator) const override {
    const std::string result = collaborator.UsefulFunctionA();
    return "The result of the B2 collaborating with ( " + result + " )";
  }
};

/**
 * The Abstract Factory interface declares a set of methods that return
 * different abstract products. These products are called a family and are
 * related by a high-level theme or concept. Products 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 products of
 * another.
 */
class AbstractFactory {
 public:
  virtual AbstractProductA *CreateProductA() const = 0;
  virtual AbstractProductB *CreateProductB() const = 0;
};

/**
```

```
 * Concrete Factories produce a family of products that belong to a single
 * variant. The factory guarantees that resulting products are compatible. Note
 * that signatures of the Concrete Factory's methods return an abstract product,
 * while inside the method a concrete product is instantiated.
 */
class ConcreteFactory1 : public AbstractFactory {
 public:
  AbstractProductA *CreateProductA() const override {
    return new ConcreteProductA1();
  }
  AbstractProductB *CreateProductB() const override {
    return new ConcreteProductB1();
  }
};

/**
 * Each Concrete Factory has a corresponding product variant.
 */
class ConcreteFactory2 : public AbstractFactory {
 public:
  AbstractProductA *CreateProductA() const override {
    return new ConcreteProductA2();
  }
  AbstractProductB *CreateProductB() const override {
    return new ConcreteProductB2();
  }
};

/**
 * The client code works with factories and products only through abstract
 * types: AbstractFactory and AbstractProduct. This lets you pass any factory or
 * product subclass to the client code without breaking it.
 */

void ClientCode(const AbstractFactory &factory) {
  const AbstractProductA *product_a = factory.CreateProductA();
  const AbstractProductB *product_b = factory.CreateProductB();
  std::cout << product_b->UsefulFunctionB() << "\n";
  std::cout << product_b->AnotherUsefulFunctionB(*product_a) << "\n";
  delete product_a;
  delete product_b;
}

int main() {
  std::cout << "Client: Testing client code with the first factory type:\n";
  ConcreteFactory1 *f1 = new ConcreteFactory1();
  ClientCode(*f1);
  delete f1;
  std::cout << std::endl;
```

```
    std::cout << "Client: Testing the same client code with the second factory type:\n";
    ConcreteFactory2 *f2 = new ConcreteFactory2();
    ClientCode(*f2);
    delete f2;
    return 0;
}
```

Output.txt: Execution result

```
Client: Testing client code with the first factory type:
The result of the product B1.
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:
The result of the product B2.
The result of the B2 collaborating with the (The result of the product A2.)
```

READ NEXT

Design Patterns: Builder in C++ →

RETURN

← Design Patterns: Factory Method in C++

Abstract Factory in Other Languages

Home
Refactoring
Design Patterns
Premium Content
Forum
Contact us