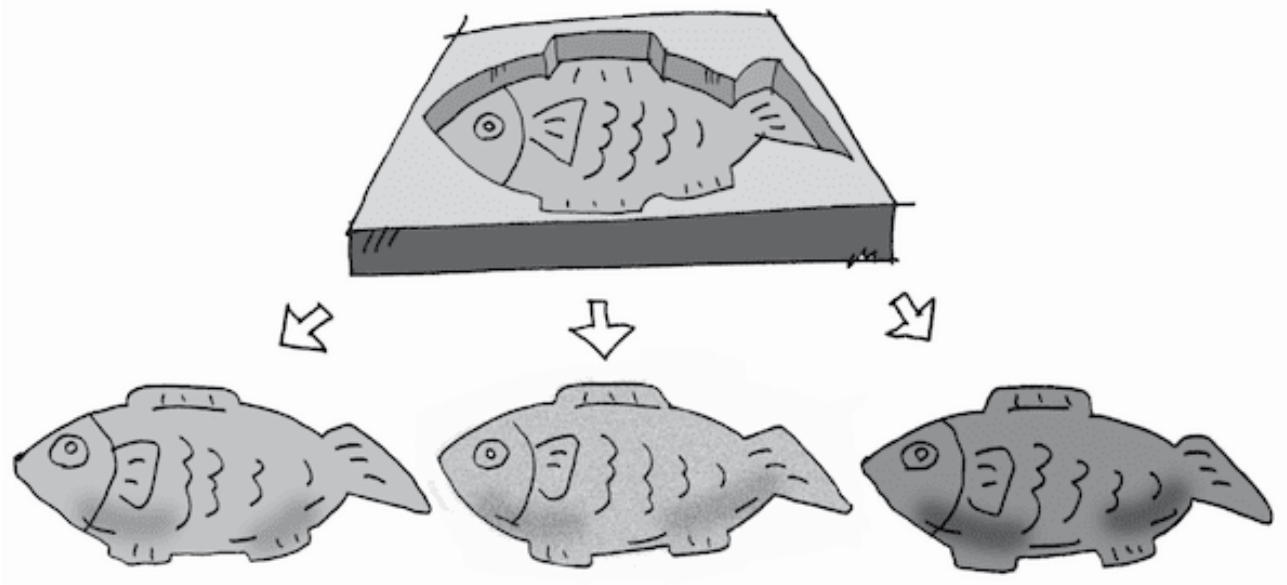
Factory Method Pattern

Subclasses Make Instances



Factory Method Pattern

We now understand the power of the Template method; then, how about applying this idea to the factory and its products?

- How about abstracting a factory (with a skeleton algorithm built in) and its subclasses instantiating the product with different configurations?
- Even better, how about using the same interface method (for example, use() method) for each product?

The Problem

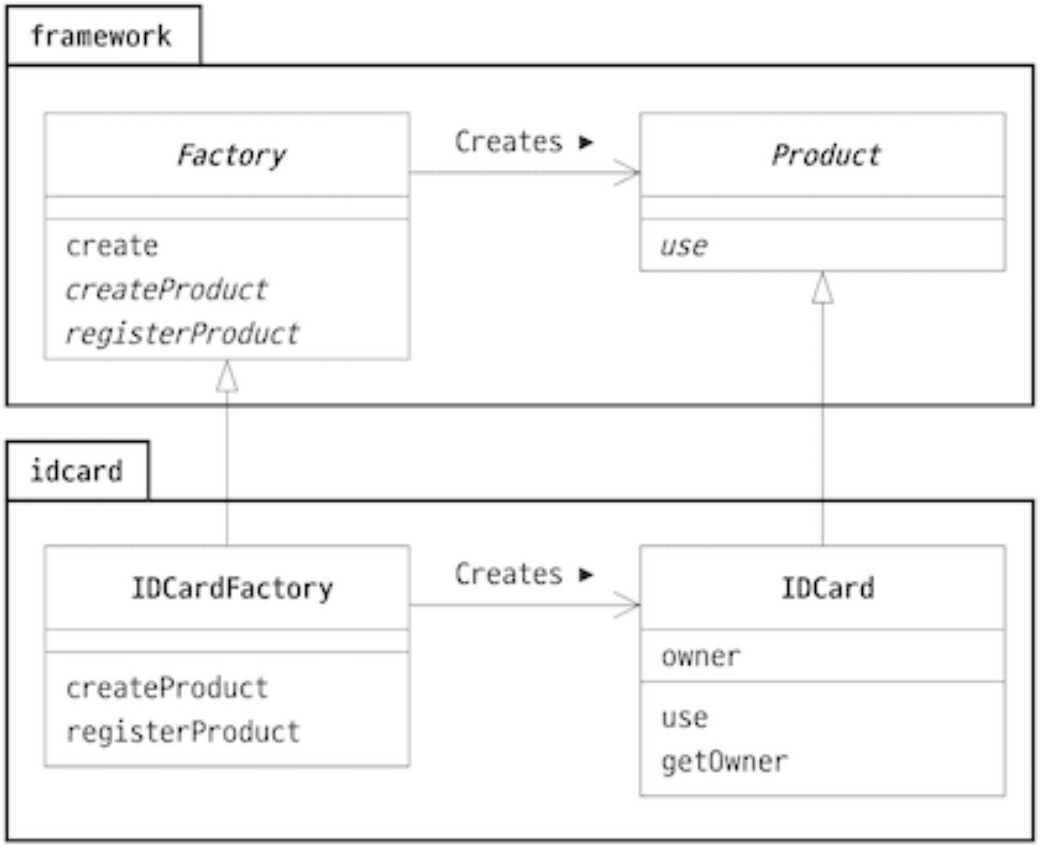
- Objects need **different creation logic** for different configurations
- Avoid using the `new` on concrete classes
- **Encapsulate creation** → easy to extend
- Define a **factory interface** (like Template Method)
- Subclasses decide **which concrete product** to create (unlike Template Method)

The challenge: how to **encapsulate object creation** while allowing **subclasses to determine** which specific objects to create?

The *Factory Method* as the Solution

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- *Factory Method* lets a class defer instantiation to subclasses.
- The creation process is `standardized` while allowing `customization`.

The Design



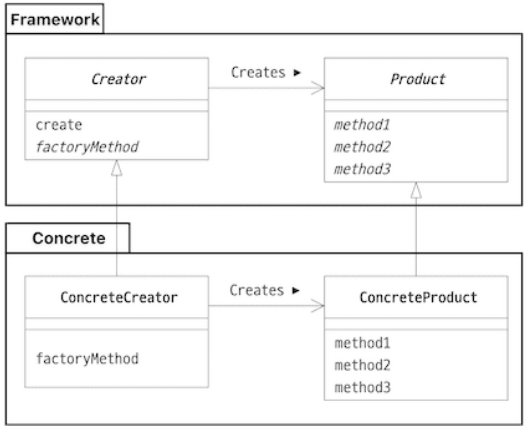
The Factory has created an interface method to instantiate objects, and Product has used an interface method to use the object.

Step 1: Understand the Players

In this design, we have players:

- *Factory* (abstract factory)
    - **ConcreteFactory** (implements factory method)
- *Product* (abstract product interface)
    - **ConcreteProduct** (specific product implementation)

Step 2: Separation of *abstraction* and concretion



- The *Factory* defines the creation process using the Template Method DP.
- **ConcreteFactory** implements the detailed `methods` to create specific products.
    - It returns **ConcreteProduct**.

Code

- Main Method
- Abstract Factory & Product
- IDCard Implementation (Concrete Factory & Product)

Main Method

We call the create_product() method to create the Product.

```
def main():
  print("=== Factory Method Pattern Example ===\n")

  id_factory = IDCardFactory()
  super_factory = SuperCardFactory()

  print("1. Creating ID cards using the factory:")

  card1 = id_factory.create_product("James Bond")
  card2 = super_factory.create_product("Kane")

  print("\n2. Using the created products:")

  card1.use()
  card2.use()
```

Step 1: Create a concrete factory

```
id_factory = IDCardFactory()
super_factory = SuperCardFactory()
```

- **IDCardFactory/SuperCardFactory** are **ConcreteFactory** that implements the Factory interface methods.

Step 2: Use the factory to create products

```
card1 = factory.createProduct1("James Bond")
card2 = factory.createProduct2("Kane")
```

- The `createProduct1() and createProduct2()` methods are the `template method` in the abstract *Factory*.

Abstract Framework

Product Interface

The Product interface has a use() method.

```
class Product(object):
  def __init__(self, name):
    self.name = name

  def use(self):
    pass
```

Factory Interface

It has two factory methods (template methods) to build two different Product objects.

The part1() and part2() will be implemented by subclasses.

```
from product import Product

class Factory(object):
  def createProduct1(self, owner):
    self.part1() # from subclass
    print('create_product1')
    return self._create_product1(owner) # from subclass

  def createProduct2(self, owner):
    self.part2() # from subclass
    print('create_product2')
    return self._create_product2(owner) # from subclass
```

Concrete Implementation

IDCard Product

This concrete product implements the use() method.

```
from product import Product

class IDCard(Product):
  def use(self):
    print(f"Using {self.name}.")
```

IDCard Factory

In this subclass, we provide the detailed implementations by making methods.

```
from factory import Factory
from id_card import IDCard

class IDCardFactory(Factory):
  def part1(self):
    print('IDCardFactory part1')

  def part2(self):
    print('IDCardFactory part2')

  def _create_product1(self, owner):
    print('create_product1')
    return IDCard('created by product 1 process')

  def _create_product2(self, owner):
    print('create_product2')
    return IDCard('created by product 2 process')
```

Output Example

```
1. Creating ID cards using the factory:
IDCardFactory part1
create_product1
create_product1
IDCardFactory part2
create_product2
create_product2

2. Using the created products:
Using created by product 1 process.
Using created by product 2 process.
```

Discussion

DIP

The Factory Method follows DIP (Dependency Inversion Principle) because it depend on abstractions, not concretions.

The creator depends on the abstract Product interface, not concrete implementations.

The creator class depends only on product interface (create_product in this example) not on concrete product classes (IDCard or SuperCard).

- The creator declares the factory method returning the abstract product.
- Subclasses decide which concrete product to instantiate.
- This reduces coupling between the creator and concrete product classes.

The whole point of Factory Method is to avoid the creator being tied to specific concrete classes.

Q: It's confusing. If it's similar to the Template method, do we need to add detailed algorithm in the create_product so subclasses use?

- **No, you don’t need a detailed algorithm** in `create_product()` for plain **Factory Method**.
- Add a detailed sequence **only if** you want a **Template Method–style** creation pipeline (fixed steps + overridable hooks).

```python
class Factory(ABC):
    def create_product(self, owner):
        """Template method: fixed creation pipeline"""
        ... # algorithm
        product = self._make(owner, parts)  # factory method
        self._register(product)         # fixed step
        return product

    @abstractmethod
    def _make(self, owner, parts):
        pass

class BasicFactory(Factory):
    def _make(self, owner, parts):
        print(f"Assembling Basic Product for {owner} with {parts}")
        return BasicProduct()
```

Parameterized factory method

Q: What if I need to create the Product with different configurations?

A factory method that takes parameters to determine which type of product to create, allowing one factory method to create multiple product variants.

We call this **Parameterized factory method**:

```
class SuperCardFactory(Factory):
  def part1(self):
    print('SuperCardFactory part1')

  def part2(self):
    print('SuperCardFactory part2')

  def create_product(self, owner): # Parameterized factory method
    self.part1()
    self.part2()
    print('create_product from SuperCardFactory')
    return SuperCard(owner)
```

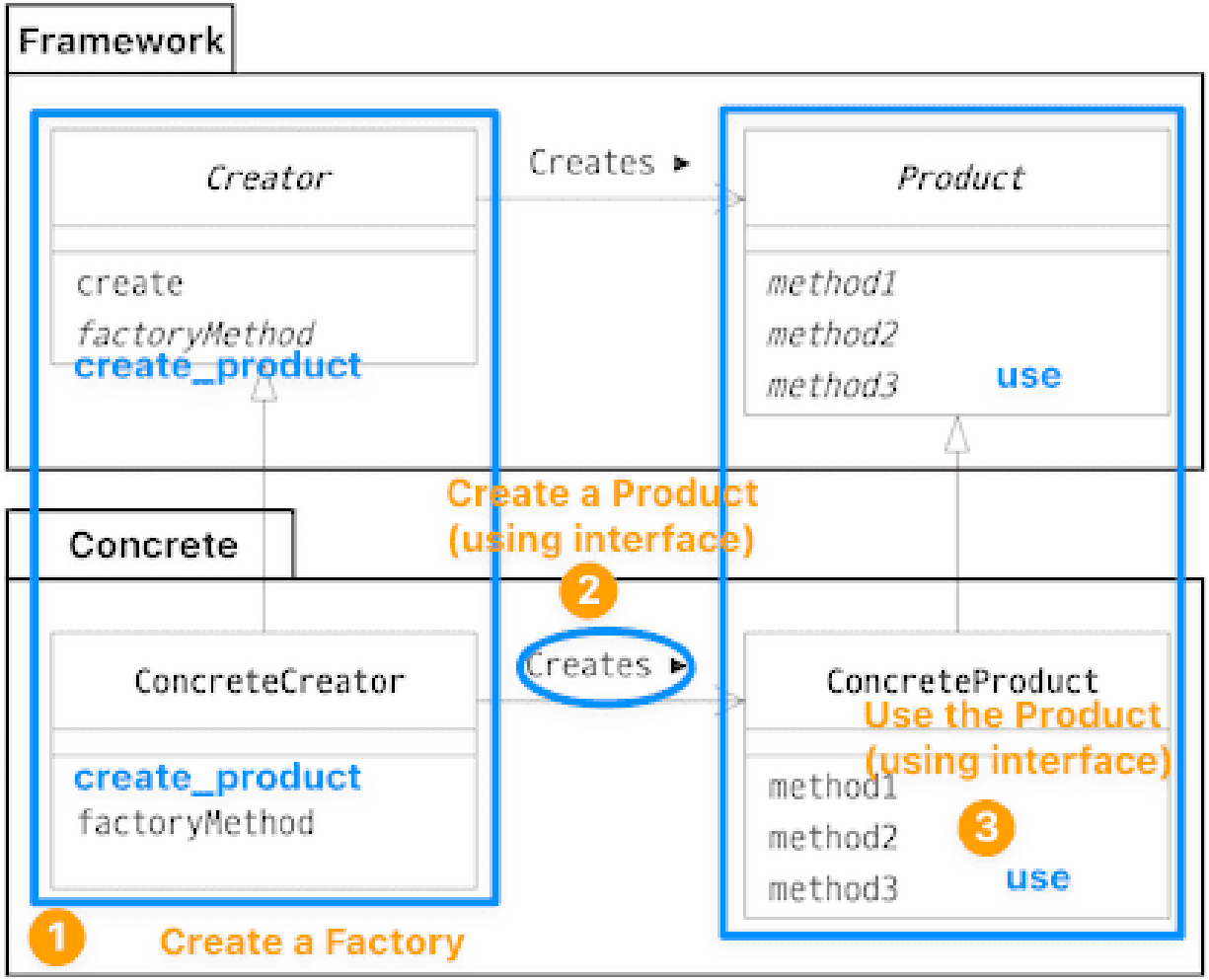
Factory Method Benefits

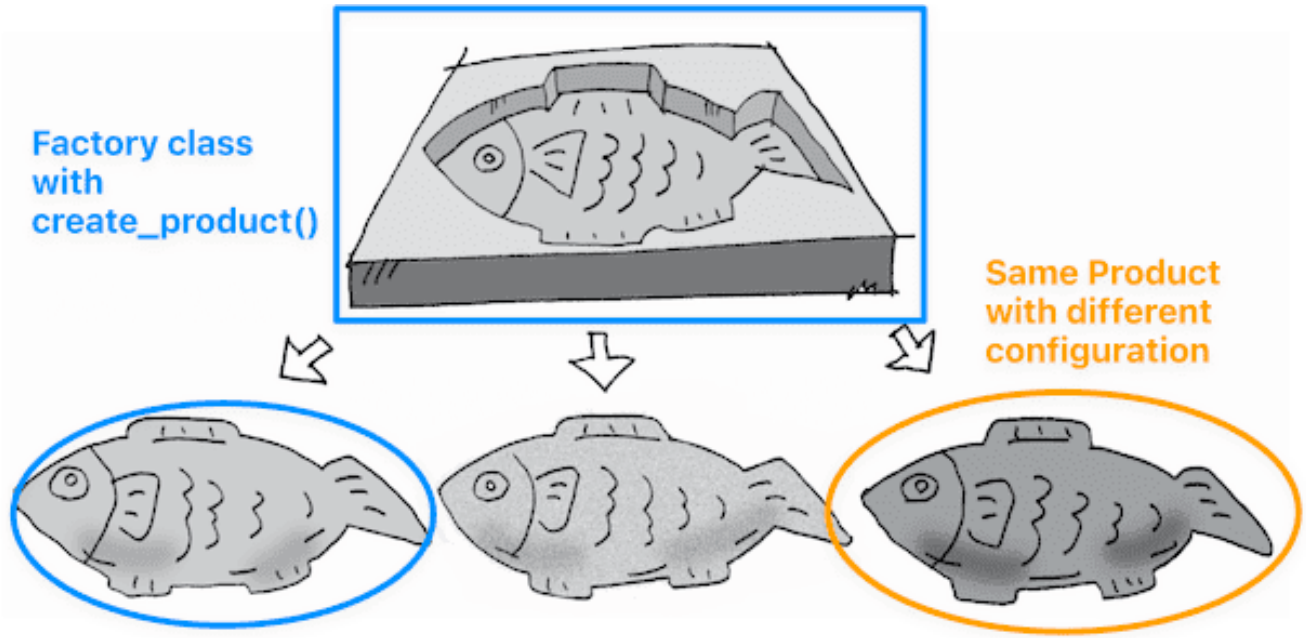
- **Encapsulation**: Object creation logic is encapsulated
- **Flexibility**: Easy to add new product types
- **Consistency**: All products go through the same creation process
- **Extensibility**: New factories can be added without modifying existing code

Related Patterns

- **Template Method** (Outline algorithm): Factory Method often relies on Template Method to outline the overall creation process, while letting subclasses decide the exact steps for object creation.

UML


Framework
Creator
create
factoryMethod
create_product
Creates
Product
method1
method2
method3
use
Create a Product
(using interface)
2
Concrete
ConcreteCreator
create_product
factoryMethod
Creates
ConcreteProduct
Use the Product
(using interface)
method1
method2
method3
3
use
1
Create a Factory


Factory class
with
create_product()
Same Product
with different
configuration