

A top-down view of a white desk. In the top left, a newspaper is partially visible with the word 'BUSINESS' and 'asset sale' printed on it. To its right is a silver calculator. Further right is a white computer keyboard. Below the calculator is a white ballpoint pen. In the bottom right corner is a white cup of black coffee on a matching saucer.

Factory Method Pattern

Vishal Parekh
Xiaohan Zhang

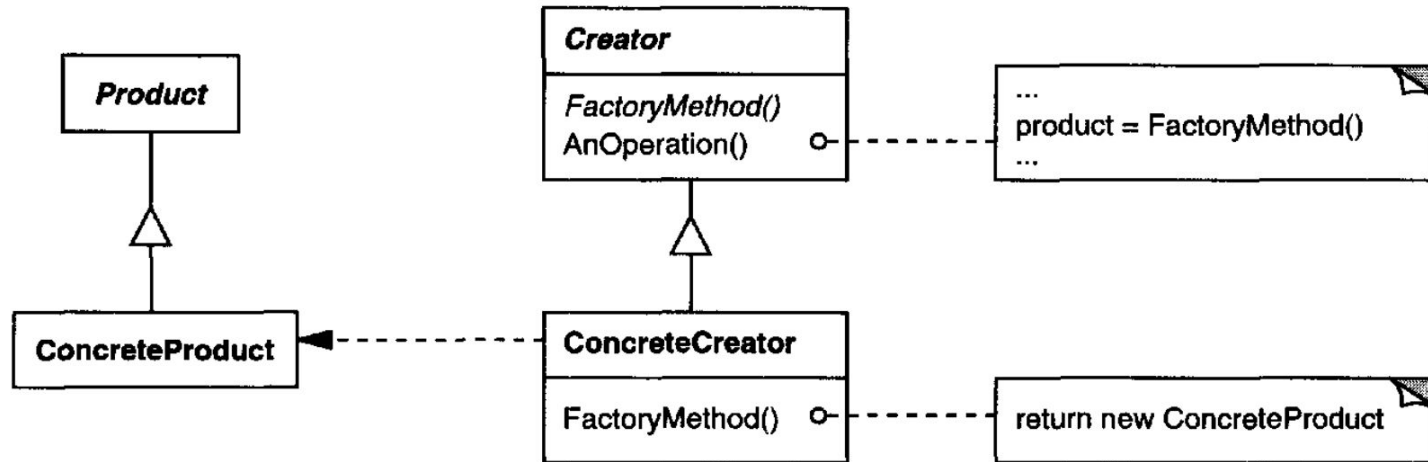
Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Let a class defer instantiation to subclasses at run time.

Virtual Constructor

- Create a copy of an object or a new object without knowing its concrete type.
- Don't know the real type of an object, but need operation on it, same as Factory Method Pattern.

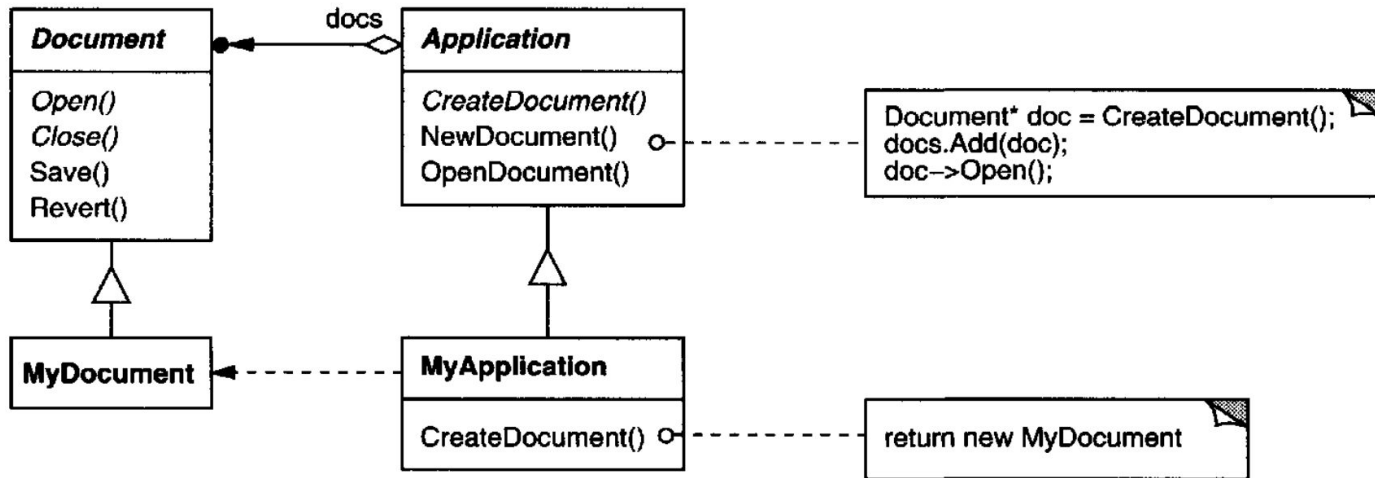
Basic Structure



Motivation

- Frameworks.
 - In frameworks, needs to create objects which may be subclassed by applications, but doesn't know the type of object
 - Factory method encapsulates knowledge of which subclass to instantiate - moves this knowledge out of the framework

Motivation Example

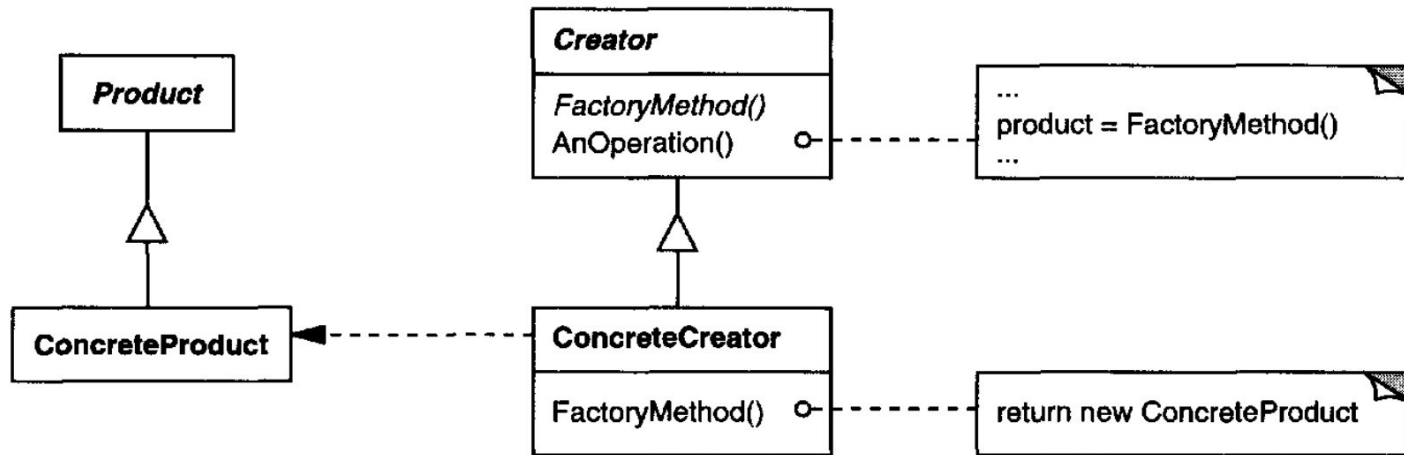


- Why do we use Factory Method?
 - Creates various types of objects without necessarily knowing what kind of object it creates or how to create.
 - Flexible and reusable classes which can be extended easily.

■ Applicability

- Use Factory Method if?
 - a class doesn't know the types of objects it must create
 - a class wants its subclasses to specify the objects it creates
 - the classes decide how to delegate responsibility to one of many other helper classes

Basic Structure



Participants

- **Product**(Document)
 - defines the interface of objects the factory method creates
- **ConcreteProduct**(MyDocument)
 - Concrete Class that implements the Product interface
- **Creator**(Application)
 - declares the factory method, which returns an object of type Product.
 - May also returns a default ConcreteProduct object
- **ConcreteCreator**(MyApplication)
 - overrides the factory method to return an instance of ConcreteProduct

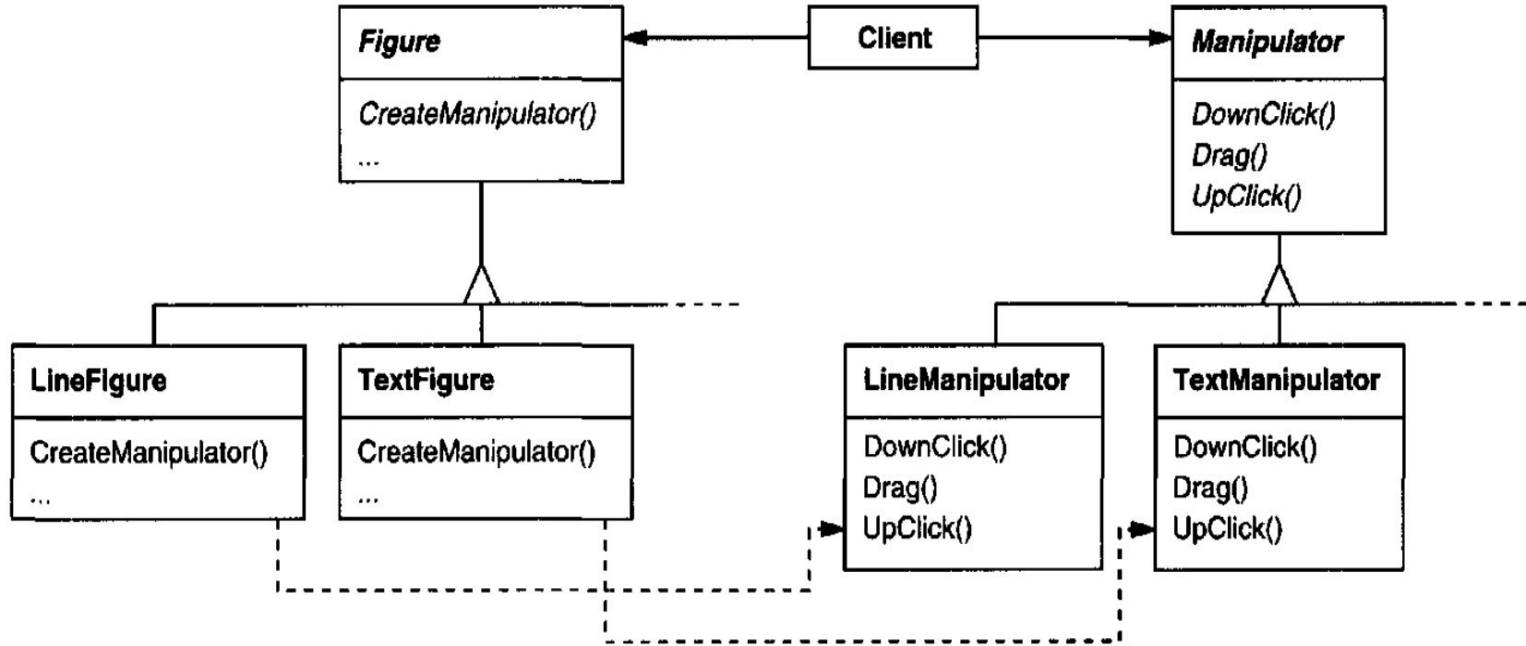
Collaborators

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct
- The client relies on the Creator to obtain a reference to the Product, without the knowledge of which ConcreteProduct will be created.

Consequences

- Factory Method Pattern has the following consequences :
 - Decoupling client from ConcreteProduct classes, only deals with the Product interface
 - Provides hooks for subclasses: Factory Method gives subclasses a hook for providing an extended version of an object.
 - Connects parallel class hierarchies

Consequences Example



Implementation- Need of Factory Method

- The factory method is one of the most used and one of the more robust design patterns.
- If you have many object of the same base type and you manipulate them mostly as abstract objects, then you need a factory. eg: Database Connection
- Factory Method makes a design more customizable and only a little more complicated.
- Other design patterns require new classes, whereas Factory Method only requires a new operation.

Implementation- Major Variations

- Abstract Creator class with no default implementation
 - The creator class is an abstract class and doesn't provide an implementation for the factory method it declares.
 - It requires subclasses to define an implementation, because there is no reasonable default.
 - It's also possible to have an abstract class that defines a default implementation for the factory method, but this is less common.

Implementation- Major Variations

- Concrete creator class with default implementation
 - The creator is a concrete class and provides a default implementation for the factory method.
 - The concrete creator uses the factory method primarily for flexibility.
 - It follows a rule that says “Create objects in a separate operation so that subclasses can override the way they’re created”.

Implementation- Other Variations

- Parameterized factory methods
 - It allows the factory method to create multiple type of products.
 - The factory method takes a parameter that identifies the kind of object to create.
 - Overriding a parameterized factory method lets you easily and selectively extend or change the product that a Creator produces.

Implementation- Parameterized

A parameterized factory method has the following general form, where MyProduct and YourProduct are subclasses of Product:

```
class Creator
{
    public:
    virtual Product* Create(ProductId id);
};
Product* Creator::Create (ProductId id)
{
    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // repeat for remaining products...
    return 0;
}
```

For example, a subclass MyCreator could swap MyProduct and YourProduct and support a new TheirProduct subclass:

```
Product* MyCreator::Create (ProductId id)
{
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;

    // N.B.: switched YOURS and MINE
    if (id == THEIRS) return new TheirProduct;
    return Creator::Create(id);
    // called if all others fail
}
```

Implementation- Other Variations

- Using Templates to avoid subclassing
 - A potential problem with factory method is that they might force you to subclass just to create the appropriate Product objects.
 - A solution in C++ is to provide a template subclass of Creator that's parameterized by the Product class.

Implementation- Using Templates

```
class Creator
{
    public:
        virtual Product* CreateProduct() = 0 ;
};
template <class TheProduct>
class StandardCreator: public Creator
{
    public:
        virtual Product* CreateProduct();
};
template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct ()
{
    return new TheProduct;
}
```

With this template, the client supplies just the product class—no subclassing of Creator is required.

```
class MyProduct : public Product
{
    Public:
        MyProduct();
};
StandardCreator<MyProduct> myCreator;
```

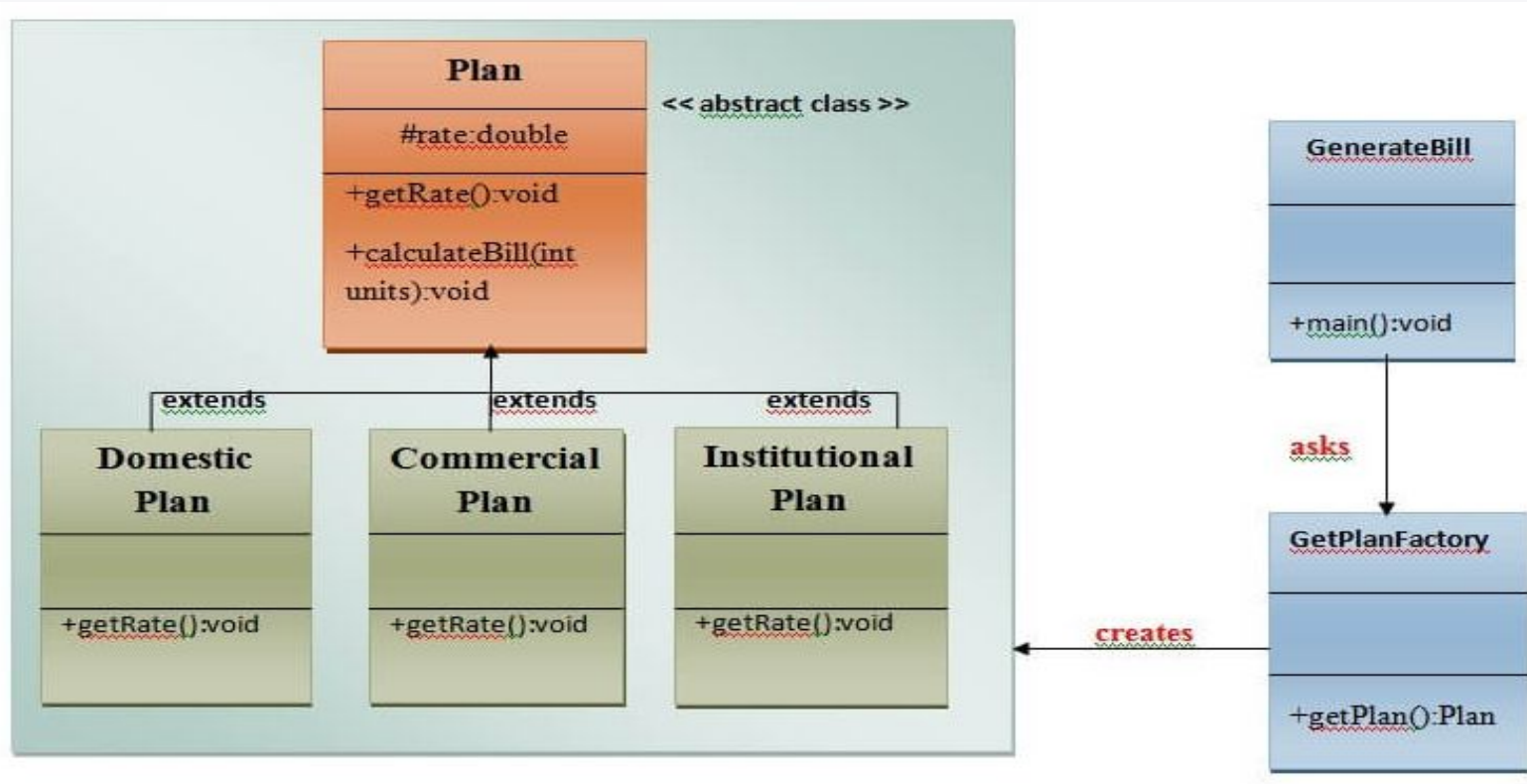
Example - Problem

- Imagine that you are creating a electricity bill management application. The first version of your app can handle only domestic plans of electricity bill, so the bulk of your code lives in a Domestic Plan class.
- After a while, your app becomes so popular that you get tons of requests to include Commercial Electricity Plan and Institutional Plan as well.
- It looks that most of your code is coupled to the Domestic Plan class. Adding Plans would require making changes to the entire codebase. Moreover, if you decide to add another type of plans to the app, you will probably need to make all of those change again.
- You will end up with nasty code riddled with conditionals, which select behaviors depending on classes of plan objects.

Example - Solution

- The Factory Method pattern suggests replacing direct object creation (using a new operator) with a call to a special "factory" method.
- The constructor call should be moved inside that method. Objects returned by factory methods are often referred to as "products."
- Subclasses may return different concrete products as long as these products have a common base class or interface.
- Clients of a factory method do not care about the particular type of a product they receive. They work with all products using a common product interface.

Minimal Code Implementation - UML Diagram



Minimal Code Implementation

Step 1: Create a Plan abstract class.

```
import java.io.*;

abstract class Plan{

    protected double rate;

    abstract void getRate();

    public void calculateBill(int units){
        System.out.println(units*rate);
    }

}

} //end of Plan class.
```

Step 2: Create the concrete classes that extends Plan abstract class.

```
class DomesticPlan extends Plan{
    // @override
    public void getRate(){
        rate=3.50;
    }
} //end of DomesticPlan class.
```

```
class CommercialPlan extends Plan{
    // @override
    public void getRate(){
        rate=7.50;
    }
} //end of CommercialPlan class.
```

```
class InstitutionalPlan extends Plan{
    // @override
    public void getRate(){
        rate=5.50;
    }
} //end of InstitutionalPlan class.
```

Minimal Code Implementation

Step 3: Create GetPlanFactory to create object of concrete classes

```
class GetPlanFactory{
```

```
//use getPlan method to get object of type Plan
```

```
public Plan getPlan(String planType){
```

```
    if(planType == null){
```

```
        return null;
```

```
    }
```

```
    if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
```

```
        return new DomesticPlan();
```

```
    }
```

```
    else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
```

```
        return new CommercialPlan();
```

```
    }
```

```
    else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
```

```
        return new InstitutionalPlan();
```

```
    }
```

```
    return null;
```

```
}
```

```
}//end of GetPlanFactory class.
```

Step 4: Generate Bill by using the GetPlanFactory to get the object

```
import java.io.*;
```

```
class GenerateBill{
```

```
    public static void main(String args[])throws IOException{
```

```
        GetPlanFactory planFactory = new GetPlanFactory();
```

```
        System.out.print("Enter the name of plan for which the bill will be generated: ");
```

```
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

```
        String planName=br.readLine();
```

```
        System.out.print("Enter the number of units for bill will be calculated: ");
```

```
        int units=Integer.parseInt(br.readLine());
```

```
        Plan p = planFactory.getPlan(planName);
```

```
        //call getRate() method and calculateBill()method of DomesticPaln.
```

```
        System.out.print("Bill amount for "+planName+" of "+units+" units is: ");
```

```
        p.getRate();
```

```
        p.calculateBill(units);
```

```
    }
```

```
}//end of GenerateBill class.
```

■ Pros and Cons

- Follows the Open/Closed Principle.
- Avoids tight coupling between concrete products and code that uses them.
- Simplifies code due to moving all creational code to one place.
- Simplifies adding new products to the program.
- Requires extra subclasses.
- The factory has to be used for a family of objects.

Known Uses

- JAVA API
 - Used in Several Places in the Java.
 - URLConnection has a method getContent that returns the content as an appropriate object (html, gif etc.)
- .NET Framework Class Library
 - Systems.Collections.IEnumerable,
 - System.Net.WebRequest
 - System.Security.Cryptography

Related Patterns

- Abstract Factory
- Factory methods
- Prototypes

References

- □Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (n.d.).
Design patterns.
- <https://www.oodesign.com/factory-method-pattern.html>
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns
- <https://www.javatpoint.com/factory-method-design-pattern>
- <https://refactoring.guru/design-patterns/factory-method>

A top-down view of a minimalist white desk. In the upper left, a small green plant with round leaves sits in a white pot. To its right is a white computer keyboard. Further right is a white computer mouse. In the lower right, a white coffee cup sits on a matching saucer. In the lower left, a folded newspaper is visible, with the word "BUSINESS" and the number "16" printed on it. A white rectangular box with a piece of tape on its left side is centered on the desk, containing the text "Thank You".

Thank You