Factory Pattern

The **Factory Method Pattern** defines an interface for creating an object but lets the subclasses decide which class instantiate. Factory method lets a class defer instantiation to subclasses.

Toni Sellarès
Universitat de Girona

Design Principles

Encapsulate what varies

Factory patterns encapsulate “object creation”.

Software Entities should be Open for Extension, but Closed for Modification (OCP)

Factory pattern allows writing of methods that can instantiate different objects and can be extended to instantiate newly-developed objects, all without modifying the method’s code!

Depend On Abstractions. Do not depend on Concrete Classes (DIP)

However, every time you do a *new* you need to deal with a *concrete* class, not an abstraction.

Factory Method Motivation (1)

At times, an application object may only know that it needs to access a class from within the class hierarchy, but does not know exactly which class from among the set of subclasses of the parent class is to be selected.

The choice of an appropriate class may depend on factors such as:

- The state of the running application.
- Application configuration settings.
- Expansion of requirements or enhancements.

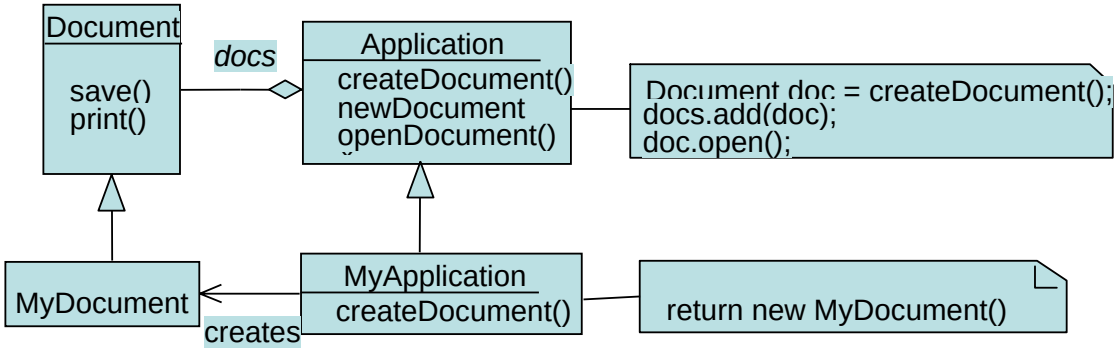
Factory Method Motivation (2)

Factory Method pattern recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*.

A factory method canbe defined as a method in a class that:

- Selects an appropriate class from a class hierarchy based on the application context and other influencing factors.
- Instantiates the selected class and returns it as an instance of the parent class type.

Factory Method Motivation (3)



Application class is responsible for creation and management of Documents

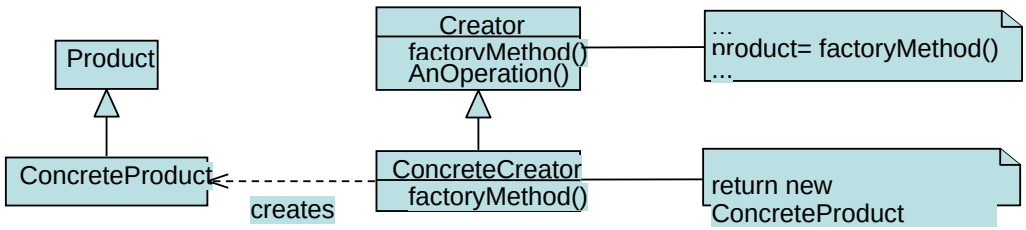
Problem:

- Application class knows WHEN a new document should be created.
- Application class doesn't know WHAT KIND of document to create.

Solution:

- Application defines a virtual function, createDocument().
- MyApplication makes sure that createDocument() will create a product Document) of the correct type.

Factory Method Structure



Product (Document): defines the interface of objects the factory method creates.

ConcreteProduct (MyDocument): implements the Product interface.

Creator (Application): declares the factory method, which returns an object of type Product. May also define a default implementation of the factory method that returns a default ConcreteProduct object.

ConcreteCreator (MyApplication): overrides the factory method to return an instance of a ConcreteProduct.

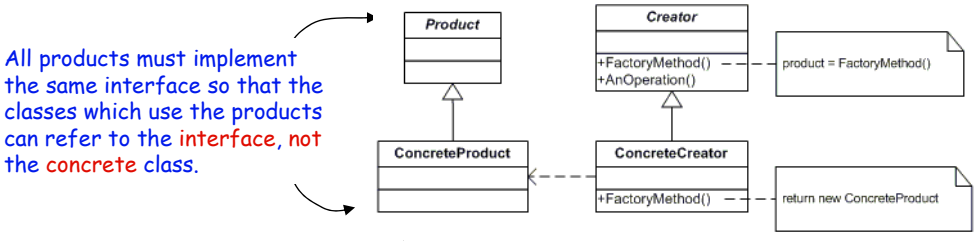
Factory Method Pattern Defined

The **Factory Method Pattern** defines an interface for creating an object but lets the subclasses decide which class instantiate. Factory method lets a class defer instantiation to subclasses.

Why decides?

The Creator is the class that contains the implementations for all the methods to manipulate the products, except for the factory method!

The abstract factoryMethod ( )is what all Creator subclasses must implement.



All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.

The ConcreteCreator implements the factoryMethod ( ), which is the method that actually produces the products.

The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has knowledge of how to create these products.

Factory Method Consequences

- Advantage
  - Concrete (Dynamic) types are isolated from the client code.
- Provides hooks for subclasses
  - A standard technique for subclasses to affect their parents' behavior.
- Potential disadvantage
  - clients might have to subclass the Creator class just to create a particular (i.e., 1) ConcreteProduct object.

Factory Method Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the object it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method Implementation

- Two major varieties
  - Creator declares ABSTRACT factory method, ConcreteCreator implements it.
  - Creator defines a default implementation for factory method.
- Parameterized factory methods
  - lets the factory method to create *multiple* kinds of objects
  - factory methods takes a parameter: a kind of object to create all products have to share a Product interface.

Factory Method: Structural Example

```
/**
 * Declares the factory method, which returns an object of type Product. Creator
 * may also define a default implementation of the factory method that return a
 * default ConcreteProduct object. May call the factory method to create
 * a Product object.
 */

public abstract class Creator {
    protected abstract Product factoryMethod();
}


/**
 * Overrides the factory method to return an instance of a ConcreteProduct.
 */

public class ConcreteCreator extends Creator {
    protected Product factoryMethod()
       { return new ConcreteProduct();
       }
}
```

```
/**
 * Defines the interface of objects the factory method creates.
 */

public interface Product { }

/**
 *  Implements the Product interface.
 */

public class ConcreteProduct implements Product { }

/**
 * Test driver for the pattern.
 */

public class Test {
    public static void main( String arg[] ){
        Creator c = new ConcreteCreator();
        Product p = c.factoryMethod()
        System.out.println ("Created", p);}
}
```

Example: Reading Image Files (1)

Consider a program to read image files and make thumbnails out of them.

The program supports different image formats, represented by a reader class for each format:

```
public interface ImageReader {
    public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader {
    public GifReader(InputStream in) {
        // check that it's a gif, throw exception if it's not, then
        // if it is decode it.
    }
    public DecodedImage getDecodedImage() {
        return decodedImage;
    }
}

public class JpegReader implements ImageReader {
       //...
}
```

Example: Reading Image Files (2)

Each time the program reads an image it needs to create a reader of the appropriate type based on some information in the file. This logic can be encapsulated in a factory method:

```
public class ImageReaderFactory {
    public static ImageReader getImageReader( InputStream is ) {
        int imageType = figureOutImageType(is);
        switch( imageType ) {
        case ImageReaderFactory.GIF:
            return new GifReader(is);
        case ImageReaderFactory.JPEG:
            return new JpegReader(is);
        // etc.
        }
    }
}

public class ImageProcessor {
    public static void main( String[] args ) {
       String filename = args[0];
       ImageReaderFactory factory = new ImageReaderFactory();
       ImageReader out = factory.getImageReader(filename);
       printOut(out.getDecodedImage());
    }
}
```