## Factory Method Pattern

1. **Intent**

Factory method pattern defines an interface to create objects. It is also known as Virtual Constructor

1. **Motivation**

In many problems, we are often in situations where we create objects based on some conditions (input parameters, or return values of other methods). Furthermore, the conditions can be subject to changes in the future, so it is not a wise idea to create concrete objects in client code

The Factory Method offer a solution: encapsulate the knowledge to create objects out of the framework

1. **Applicability**

Factory Method pattern is used when:

- A class do not know which class to create an object before runtime

- A class want its subclass to specify the objects

- You want to move responsibilities to helper classes

1. **Participants**

An example: A factory to create pasta with some type of shapes (spaghetti, bucatini, …)

Product (Pasta)

- Defines the interface of objects the factory method creates.

ConcreteProduct (Spaghetti..)

- Implements the Product interface.

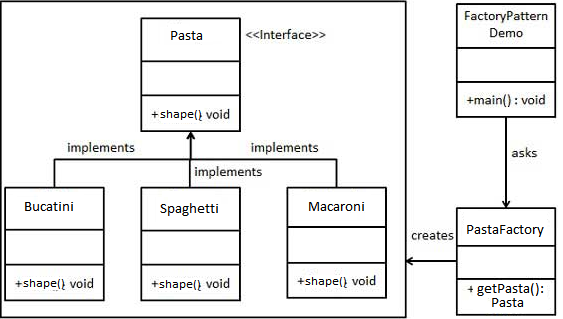
Creator (FactoryPatternDemo)

- Declares the factory method, which returns an object of type Product.

ConcreteCreator (PastaFactory)

- Overrides the factory method to return an instance

1. **Structure**



1. **Collaborations**

Creator depends on its subclasses to define the factory method to return an instance of the appropriate ConcreteProduct.

In this case PastaFactory depends on the pasta shape name to create correct type of pasta

1. **Consequences**

- Factory methods remove the need to bind application-specific classes into your

code. The code only deals with the Product interface, so it can work with

any user-defined ConcreteProduct classes. (We do not need to change the code when we decide to add other shapes of pasta, we just add subclasses and modify the Factory)

- A potential drawback of factory methods is that clients may have to subclass the Creator class only to create a specific ConcreteProduct object. Subclassing is acceptable when the client need to subclass the Creator class anyway, but otherwise the client now have another problem to consider.

- Some additionals consequences:

- Provides hooks for subclasses:

It is always more flexible to create objects with a factory method inside a class than to create an concrete object directly.

1. **Implementation**

Suppose we have a PastaStore class:

public class PastaStore {

public PastaStore (PastaFactory factory) {

this.factory = factory;

}

public Pasta makePasta(String type) {

Pasta pasta;

Pasta = factory.getPasta(type);

Pasta.dry();

Pasta.bag();

Pasta.label();

Pasta.box();

return pasta;

}

}

We now make pizza depends on type

public class PastaFactory {

public Pizza getPasta(String type) {

Pasta pasta= null;

if (type.equals(“spaghetti”)) {

pasta= new Spaghetti();

} else if (type.equals(“macaroni”)) {

pasta= new Macaroni();

} else if (type.equals(“bucatini”)) {

pasta= new Bucatini();

}

return pasta;

}

}

1. **Related Pattern:**

Factory Method is often used together with Abstract Factory