Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Factory Method Pattern

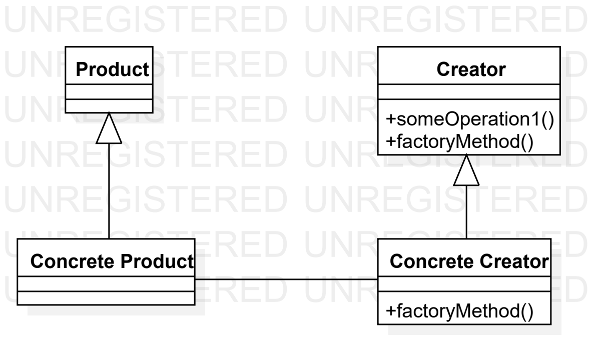
Factory Method: Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Also known as **Virtual Constructor**

Factory Method: Applicability

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects 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: Structure



Factory Objects

- In order to understand Factory Method patterns, we must first understand factory objects.
- A **factory object** operates like a factory in the real world, and creates objects.
- Factory objects **makes software easier to maintain and change**, because object creation happens in the factories. The methods that use these Factories can then focus on other behavior.

Factory Objects: Example

- Imagine a situation where you have a software that implements an *online store that sells knives*.
- Perhaps when the store first opens, you only produce `steak knives` and `chef'knives`. You would have a superclass with those two subclasses.
- In your system, first, a knife object is created.
- Conditionals then determine which subclass of Knife is actually instantiated. This act of instantiating a class to create an object of a specific type is known as **concrete instantiation**.
- In Java, concrete instantiation is indicated with the operator "new".

Factory Objects: Example (contd.)

- However, imagine that the store is successful and adds more knife types to sell. New subclasses will need to be added, such as `BreadKnife`, `ParingKnife`, or `FilletKnife`. The list of conditionals would need to grow and grow as new knife types are added.

- However, the methods of the knife after it is created, such as `sharpening`, `polishing`, and `packaging`, would likely stay the same, no matter the type of knife. This creates a complicated situation.

- Instead of making Knives in the store, it may be better to create them in a Factory object.

Factory Objects: Example (contd.)

- A factory object is an object whose role is to create "product" objects of particular types.
- **In this example**, the methods of `sharpening`, `polishing`, and `packaging` would remain in the `orderKnife` method. However, the responsibility of creating the product will be delegated to another object: a "`Knife Factory`".
- The code for deciding which knife to create, and the code for deciding which subclass of `Knife` to instantiate, are moved into the class for this factory.

Factory Objects: Example (contd.)

```
public class KnifeFactory extends KnifeStore{
{
    public Knife createKnife(String knifeType)
    {
        Knife knife = null;
        // create Knife object
        If (knifeType.equalsIgnoreCase("steak"))
        {
            knife = new SteakKnife();
        }
        else if (knifeType.equalsIgnoreCase("chefs")) {
            knife = new ChefsKnife();
        }
        return knife;
    }
}
```

Factory Objects: Example (contd.)

- The code to create a `Knife` object has been moved into a method of a new class called `KnifeFactory`. This allows the `KnifeStore` class to be a client for the `Knife Factory`.
- The `KnifeFactory` object to use is passed into the constructor for the `KnifeStore` class.
- Instead of performing concrete instantiation itself, the `orderKnife` method delegates the task to the factory object.

Factory Objects (contd.)

- Concrete instantiation is the primary purpose of Factories.
- In general, a factory object is an instance of a factory class, which has a method to create product objects.

Factory Objects (contd.)

```
public abstract class KnifeStore
{
    public Knife orderKnife(String knifeType)
    {
        Knife knife;
        knife = createKnife(knifeType);

        knife.sharpening();
        knife.polishing();
        knife.packaging();
        return knife;
    }
    abstract Knife createKnife(String knifeType);
}
```

Benefits of Factory Objects

There are numerous benefits to using factory objects.

- One of these benefits is that it is **much simpler to add new types of an object to the object factory** without modifying the client code.
- Factories allow client code to operate on generalizations. This is known as coding to an interface, not an implementation.
- The use of the word "factory" serves a good metaphor here. Using Factory objects is similar to using a factory in real-life to create knives—the stores do not usually make the names themselves, but get them from a factory.
- Essentially, using factory objects means that you have **cut out redundant code and made the software easier to modify**, particularly if there are multiple clients that want to instantiate the same set of classes.

Factory Method

- The Factory Method pattern does not use a Factory object to create the objects, instead, the Factory Method uses a separate "method" in the same class to create objects.
- The power of Factory Methods come in particular from how they create specialized product objects.
- Generally, in order to create a "specialized" product object, a Factory Object approach would subclass the factory class.

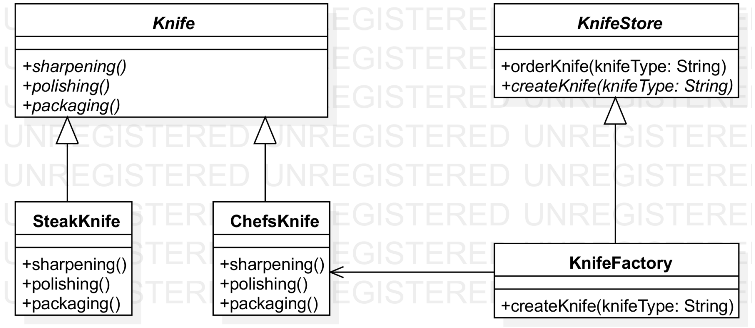
  For example,

  a subclass of the `KnifeStore` Factory called KnifeFactory would make `ChefsKnife` and `SteakKnife` product objects.

Factory Method (contd.)

- A Factory Method approach, however, would use a KnifeFactory subclass of KnifeStore. The KnifeFactory has a method—the “Factory Method”—that is responsible for creating `ChefsKnife` and `SteakKnife` product objects instead.
- This design pattern’s intent is to define an interface for creating objects, but let the subclasses decide which class to instantiate.
- So, instead of working with a factory object, we specialize or subclass the class that uses the Factory Method.
- Each subclass must define its own Factory Method. This is known as **letting the subclasses decide** how objects are made.

Factory Method: Example (contd.)



Factory Method: Example (contd.)

- In the above diagram, the "`Knife`" and "`KnifeStore`" classes are italicized to indicate that they are abstract classes that cannot be instantiated.
- In summary, this diagram indicates to us that `KnifeFactory`, and any other `KnifeStore` subclass defined must have its own `createKnife()` method.
- This structure is the core of the Factory Method design pattern.

Factory Method: Summary

- A simple factory like this **returns an instance of any one of several possible classes that have a common parent class**.
- The common parent class can be **an abstract class, or an interface**.
- The calling program typically has a way of telling the factory what it wants, and the factory makes the decision which subclass should be returned to the calling program. It then creates an instance of that subclass, and then returns it to the calling program.