Factory method pattern

From Wikipedia, the free encyclopedia

In class-based programming, the *factory method pattern* is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

Contents

Overview

The Factory Method [1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The Factory Method design pattern solves problems like: [2]

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Creating an object directly within the class that requires (uses) the object is inflexible because it commits the class to a particular object and makes it impossible to change the instantiation independently from (without having to change) the class.

The Factory Method design pattern describes how to solve such problems:

- Define a separate operation (*factory method*) for creating an object.
- Create an object by calling a *factory method*.

This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate). See also the UML class diagram below.

Definition

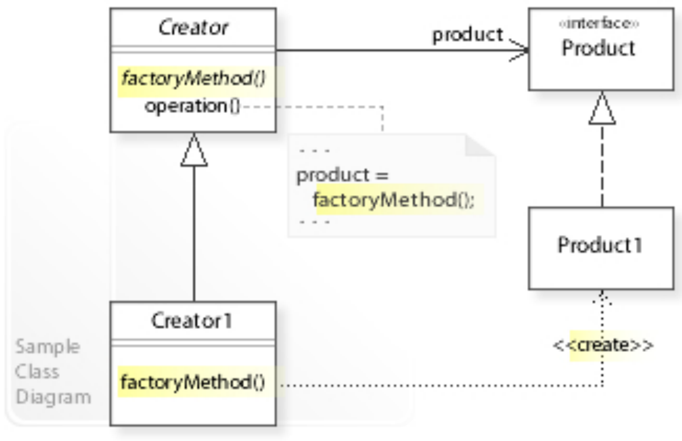
"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses." (Gang Of Four)

Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.[3]

Structure

UML class diagram

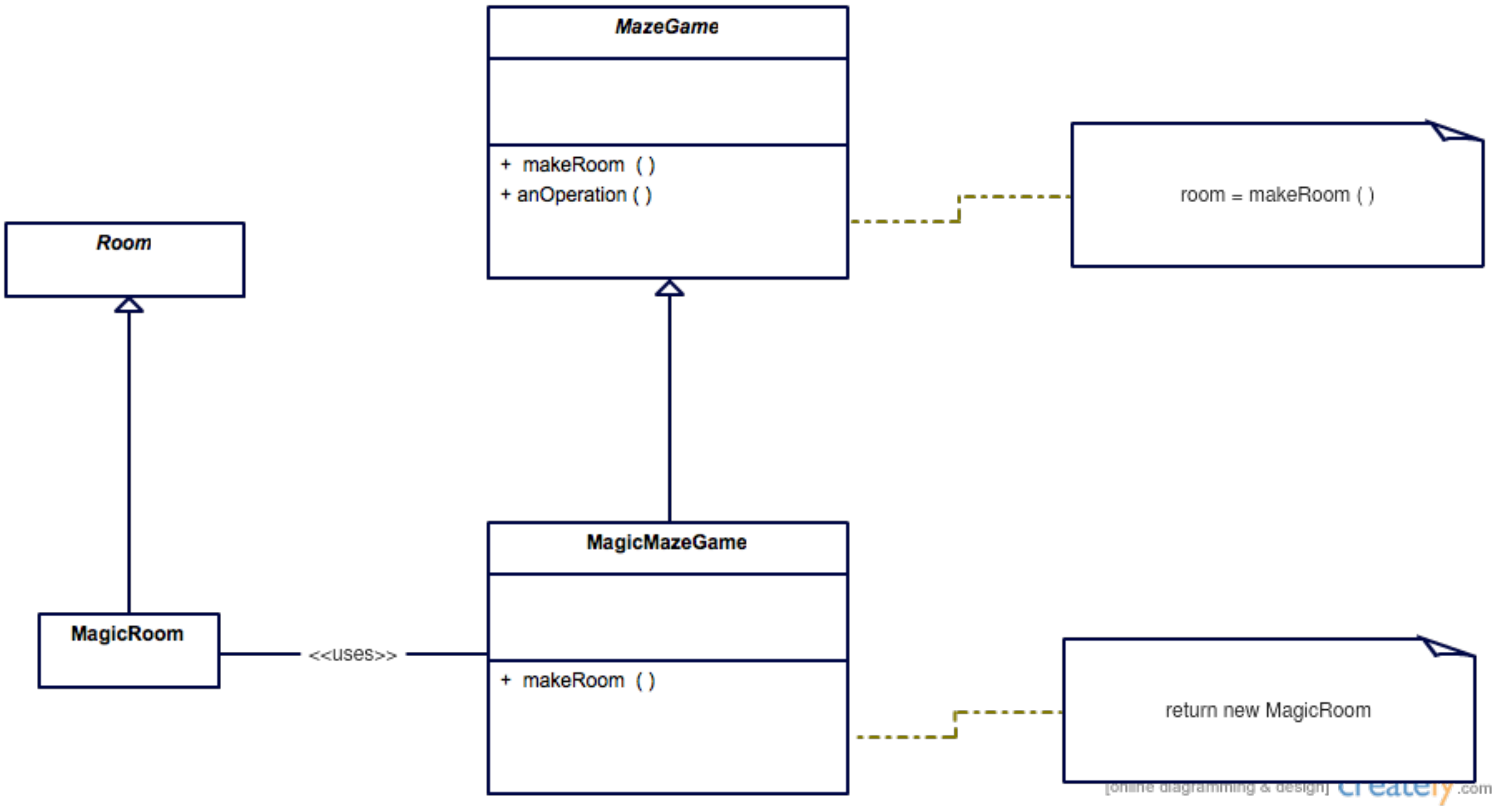


A sample UML class diagram for the Factory Method design pattern. [4]

In the above UML class diagram, the `Creator` class that requires a `Product` object doesn't instantiate the `Product1` class directly. Instead, the `Creator` refers to a separate `factoryMethod()` to create a product object, which makes the `Creator` independent of which concrete class is instantiated. Subclasses of `Creator` can redefine which class to instantiate. In this example, the `Creator1` subclass implements the abstract `factoryMethod()` by instantiating the `Product1` class.

Example

Structure



`Room` is the base class for a final product (`MagicRoom` or `OrdinaryRoom`). `MazeGame` declares the abstract factory method to produce such a base product. `MagicRoom` or `OrdinaryRoom` are subclasses of the base product implementing the final product. `MagicMazeGame` and `OrdinaryMazeGame` are subclasses of `MazeGame` implementing the factory method producing the final products. Thus factory methods decouple callers (`MazeGame`) from the implementation of the concrete classes. This makes the "new" Operator redundant, allows adherence to the Open/closed principle and makes the final product more flexible in the event of change.

Example implementations

Java

A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow players to be transported at random (this Java example is similar to one in the book *Design Patterns*). The MazeGame uses Rooms but it puts the responsibility of creating Rooms to its subclasses which create the concrete classes. The regular game mode could use this template method:

```
public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();

    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }

    abstract protected Room makeRoom();
}
```

In the above snippet, the `MazeGame` constructor is a template method that makes some common logic. It refers to the `makeRoom` factory method that encapsulates the creation of rooms such that other rooms can be used in a subclass. To implement the other game mode that has magic rooms, it suffices to override the `makeRoom` method:

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();
```

PHP

Another example in PHP follows, this time using interface implementations as opposed to subclassing (however the same can be achieved through subclassing). It is important to note that the factory method can also be defined as public and called directly by the client code (in contrast with the Java example above).

```
/* Factory and car interfaces */

interface CarFactory
{
    public function makeCar();
}

interface Car
{
    public function getType();
}

/* Concrete implementations of the factory and car */

class SedanFactory implements CarFactory
{
    public function makeCar()
    {
        return new Sedan();
    }
}

class Sedan implements Car
{
    public function getType()
    {
        return 'Sedan';
    }
}

/* Client */

$factory = new SedanFactory();
$car = $factory->makeCar();
print $car->getType();
```

VB.NET

Factory pattern deals with the instantiation of objects without exposing the instantiation logic. In other words, a Factory is actually a creator of objects which have a common interface.

```
'Empty vocabulary of actual object
Public Interface IPerson
    Function GetName() As String
End Interface

Public Class Villager
    Implements IPerson
    Public Function GetName() As String Implements IPerson.GetName
        Return "Village Person"
    End Function
End Class

Public Class CityPerson
    Implements IPerson
    Public Function GetName() As String Implements IPerson.GetName
        Return "City Person"
    End Function
End Class

Public Enum PersonType
    Rural
    Urban
End Enum

''' <summary>
''' Implementation of Factory - Used to create objects
''' </summary>
Public Class Factory
    Public Function GetPerson(type As PersonType) As IPerson
        Select Case type
```

```
            Case PersonType.Rural
                Return New Villager()
            Case PersonType.Urban
                Return New CityPerson()
            Case Else
                Throw New NotSupportedException()
        End Select
    End Function
End Class
```

C#

Same code for C#

```
//Empty vocabulary of actual object
public interface IPerson
{
    string GetName();
}

public class Villager : IPerson
{
    public string GetName()
    {
        return "Village Person";
    }
}

public class CityPerson : IPerson
{
    public string GetName()
    {
        return "City Person";
    }
}

public enum PersonType
{
    Rural,
    Urban
}

/// <summary>
/// Implementation of Factory - Used to create objects
/// </summary>
public class Factory
{
    public IPerson GetPerson(PersonType type)
    {
        switch (type)
        {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
            default:
                throw new NotSupportedException();
        }
    }
}
```

In the above code you can see the creation of one interface called IPerson and two implementations called Villager and CityPerson. Based on the type passed into the Factory object, we are returning the original concrete object as the interface IPerson.

A factory method is just an addition to Factory class. It creates the object of the class through interfaces but on the other hand, it also lets the subclass decide which class is instantiated.

```
public interface IProduct
{
    string GetName();
    string SetPrice(double price);
}

public class Phone : IProduct
{
    private double _price;

    public string GetName()
    {
        return "Apple TouchPad";
    }

    public string SetPrice(double price)
    {
        this._price = price;
        return "success";
    }
}

/* Almost same as Factory, just an additional exposure to do something with the created method */
public abstract class ProductAbstractFactory
{
    protected abstract IProduct DoSomething();

    public IProduct GetObject() // Implementation of Factory Method.
    {
        return this.DoSomething();
    }
}

public class PhoneConcreteFactory : ProductAbstractFactory
{
    protected override IProduct DoSomething()
    {
        IProduct product = new Phone();
        //Do something with the object after you get the object.
        product.SetPrice(20.30);
        return product;
    }
}
```

You can see we have used DoSomething in concreteFactory. As a result, you can easily call DoSomething() from it to get the IProduct. You might also write your custom logic after getting the object in the concrete Factory Method. The GetObject is made abstract in the Factory interface.

Python

```
from abc import ABCMeta, abstractmethod
```

```python
from enum import Enum


class Person(metaclass=ABCMeta):

    @abstractmethod
    def get_name(self):
        raise NotImplementedError("You should implement this!")


class Villager(Person):
    def get_name(self):
        return "Village Person"


class CityPerson(Person):
    def get_name(self):
        return "City Person"


class PersonType(Enum):
    RURAL = 1
    URBAN = 2


class Factory:
    def get_person(self, person_type):
        if person_type == PersonType.RURAL:
            return Villager()
        elif person_type == PersonType.URBAN:
            return CityPerson()
        else:
            raise NotImplementedError("Unknown person type.")


factory = Factory()
person = factory.get_person(PersonType.URBAN)
print(person.get_name())
```

Uses

- In ADO.NET, IDbCommand.CreateParameter (http://msdn2.microsoft.com/en-us/library/system.data.idbcommand.createparameter.aspx) is an example of the use of factory method to connect parallel class hierarchies.
- In Qt, QMainWindow::createPopupMenu (http://qt-project.org/doc/qt-5.0/qtwidgets/qmainwindow.html#createPopupMenu) is a factory method declared in a framework that can be overridden in application code.
- In Java, several factories are used in the javax.xml.parsers (http://download.oracle.com/javase/1.5.0/docs/api/javax/xml/parsers/package-summary.html) package. e.g. javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory.

See also

- *Design Patterns*, the highly influential book
- Design pattern, overview of design patterns in general
- Abstract factory pattern, a pattern often implemented using factory methods
- Builder pattern, another creational pattern

- Template method pattern, which may call factory methods
- Joshua Bloch's idea of a *static factory method*, which he says has no direct equivalent in *Design Patterns*.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 107ff. ISBN 0-201-63361-2.
2. "The Factory Method design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=c03&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-17.
3. Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike; Loukides, Mike, eds. "Head First Design Patterns" (http://shop.oreilly.com/product/9780596007126.do) (paperback). **1**. O'REILLY: 162. ISBN 978-0-596-00712-6. Retrieved 2012-09-12.
4. "The Factory Method design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=c03&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.

- Martin Fowler; Kent Beck; John Brant; William Opdyke; Don Roberts (June 1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Cox, Brad J.; (1986). *Object-oriented programming: an evolutionary approach*. Addison-Wesley. ISBN 978-0-201-10393-9.
- Cohen, Tal; Gil, Joseph (2007). "Better Construction with Factories" (http://tal.forum2.org/static/cv/Factories.pdf) (PDF). *Journal of Object Technology*. Bertrand Meyer. Retrieved 2007-03-12.

External links

- Factory method in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/FactoryMethod.xml) (a Design Description Language)
- Consider static factory methods (http://drdobbs.com/java/208403883) by Joshua Bloch

Retrieved from "https://en.wikipedia.org/w/index.php?title=Factory_method_pattern&oldid=799256416"
