Design Patterns

Abstract Factory Pattern
Who should create?

ebru@hacettepe.edu.tr
ebruakcapinarsezer@gmail.com
http://yunus.hacettepe.edu.tr/~ebru/
@ebru176
Ekim 2017

*revised from, www.uwosh.edu/faculty_staff/huen/262/f09/slides/10_Strategy_Pattern.ppt
Creator of Creators

• What is object family?

May be you kill the dreams but prevention of mismatchings is important (model1lefdoor-model2rightdoor)
What is?

• This pattern is one level of abstraction higher than factory pattern.

• This means that the abstract factory returns the factory of classes. Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.

Intend?

“provide an interface for creating families of related or dependent objects without specifying their concrete class”
When it is useful?

• a system should be independent of how its products are created, composed, and represented

• a system should be configured with one of multiple families of products

• a family of related product objects is designed to be used together, and you need to enforce this constraint

• you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations
Components

AbstractFactory (GUIFactory)
• declares an interface for operations that create abstract product objects.

ConcreteFactory (WinFactory, OSXFactory)
• implements the operations to create concrete product objects.

AbstractProduct (Button)
• declares an interface for a type of product object.

ConcreteProduct (WinButton, OSXButton)
• defines a product object to be created by the corresponding concrete factory.
• implements the AbstractProduct interface.

Client
• uses only interfaces declared by AbstractFactory and AbstractProduct classes
Easy Sample (No Config)
public abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
    public abstract Button createButton();
}

class WinFactory extends GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}

class OSXFactory extends GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}

---

Factories
public abstract class Button {
    private String caption;
    public abstract void paint();
    public String getCaption(){
        return caption;
    }
    public void setCaption(String caption){
        this.caption = caption;
    }
}
class WinButton extends Button {
    public void paint() {
        System.out.println("I'm a WinButton: " + getCaption());
    }
}
class OSXButton extends Button {
    public void paint() {
        System.out.println("I'm a OSXButton : " + getCaption());
    }
}
public class Application {
    public static void main(String[] args) {
        GUIFactory aFactory = GUIFactory.getFactory(); // get Concrete Factory
        Button aButton = aFactory.createButton(); // get Concrete Product
        aButton.setCaption("Play"); // use Concrete Product
        aButton.paint();
    }
}

//output is
//I'm a WinButton: Play
//or
//I'm a OSXButton: Play
Sample 2 – with config, instant product creation

- Suppose we need to get the specification of various parts of a computer based on which work the computer will be used for.
- The different parts of computer are, say Monitor, RAM and Processor. The different types of computers are PC, Workstation and Server.
abstract base class Computer

```java
abstract class Computer {
    private static Computer comp;

    public abstract Parts getRAM();
    public abstract Parts getProcessor();
    public abstract Parts getMonitor();

    public static Computer getComputer(String computerType) {
        if (computerType.equals("PC")) {
            comp = new PC();
        } else if (computerType.equals("Workstation")) {
            comp = new Workstation();
        } else if (computerType.equals("Server")) {
            comp = new Server();
        }
        return comp;
    }
}
```
class Parts {

    public String specification;
    public Parts() {}
    public Parts(String specification) {
        this.specification = specification;
    }
    public String getSpec() {
        return specification;
    }
}

class RAM extends Parts {
    public RAM(String specification) {
        this.specification = specification;
    }
}

class Processor extends Parts {
    public Processor(String specification) {
        this.specification = specification;
    }
}

class Monitor extends Parts {
    public Monitor(String specification) {
        this.specification = specification;
    }
}
sub-classes of Computer

class PC extends Computer {
    public Parts getRAM() {
        return new RAM("8 GB");
    }
    public Parts getProcessor() {
        return new Processor("M");
    }
    public Parts getMonitor() {
        return new Monitor("15 inches");
    }
}

class Workstation extends Computer{
    public Parts getRAM() {
        return new RAM("32 GB");
    }
    public Parts getProcessor() {
        return new Processor("Z Series");
    }
    public Parts getMonitor() {
        return new Monitor("19 inches");
    }
}

class Server extends Computer{
    public Parts getRAM() {
        return new RAM("128 GB");
    }
    public Parts getProcessor() {
        return new Processor("Zeon");
    }
    public Parts getMonitor() {
        return new Monitor("17 inches");
    }
}
public class Client {

    public static void main(String args[]) {

        Computer c = Computer.getComputer("Workstation");
        System.out.println(c.getMonitor().getSpec());
        System.out.println(c.getRAM().getSpec());
        System.out.println(c.getProcessor().getSpec());

    }

}
public class LowResFactory extends ResFactory() {
    public DisplayDriver getDisplayDriver() {
        return (new LRDD);
    }
    public PrinterDriver getPrinterDriver() {
        return (new LRPD);
    }
}

public class HighResFactory {
    public DisplayDriver getDisplayDriver() {
        return (new HRDD);
    }
    public PrinterDriver getPrinterDriver() {
        return (new HRPD);
    }
}

public abstract class ResFactory() {
    public static Resfactory getFactory() {
        if (System.power is Low)
            return LowResFactory()
        else  return HighResFactory()
    }
    public abstract DisplayDriver getDisplayDriver();
    public abstract PrinterDriver getPrinterDriver();
}

public class Client {
    public static void main(String args[]) {
        PrinterDriver p = (ResFactory.getFactory()).getPrinterDriver();
        DisplayDriver d = (ResFactory.getFactory()).getDisplayDriver();
        p.print();
        d.display();
    }
}

public abstract class DisplayDriver {
    public abstract display();
}

public abstract class PrinterDriver {
    public abstract print();
}

public class LRDD extends DisplayDriver {
    public display() {
    }
}

public class HRDD extends DisplayDriver {
    public display() {
    }
}

public class LRDP extends PrinterDriver {
    public print() {
    }
}

public class HRDP extends PrinterDriver {
    public print() {
    }
}
• *It isolates concrete classes.*

Product class names are isolated in the implementation of the concrete factory. They do not appear in the client code.
• It makes exchanging product families easy.

The class of a concrete factory appears only once in the application (when it is instantiated) Easy to change the concrete factory an application uses. The whole product family changes at once
• *It promotes consistency among products.*

When products are designed to work together, it’s important that an application use objects only from one family at a time. Abstract Factory makes this easy to enforce
Builder vs Abstract Factory

- Builder constructs a complex object step by step depending on the data presented to it

- Abstract Factory returns a family of related classes
Abstract Factory vs Factory Method

• The Factory Method can make a set of objects, with the objects created only as a set.

The Abstract Factory can make a number of related objects, with each object created individually.

So, maybe in a restaurant analogy the abstract factory would be "a la carte" and the factory method "fixed price".