

NOMBRE	
AREA	Software
REGIMEN	Semestral

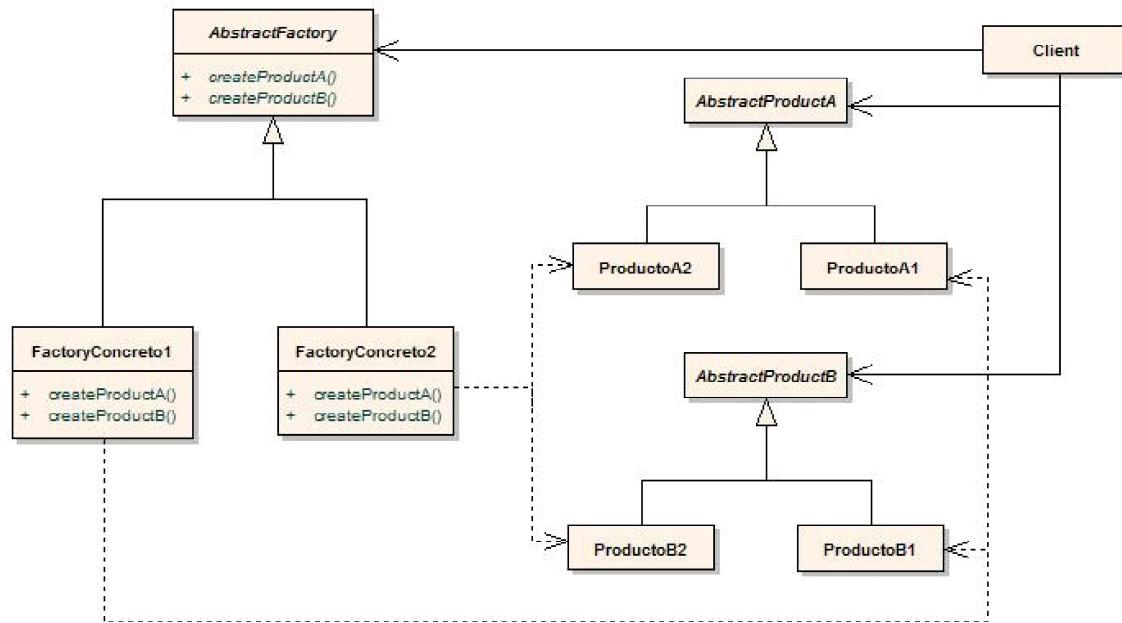
Abstract Factory

Este patrón crea diferentes familias de objetos. Su objetivo principal es soportar múltiples estándares que vienen definidos por las diferentes jerarquías de herencia de objetos. Es similar al Factory Method, sólo que esta orientado a combinar productos.

Se debe utilizar este patrón cuando:

- Un sistema se debe configurar con una de entre varias familias de productos.
- Una familia de productos que están relacionados y fueron hechos para utilizarse juntos.

Diagrama de Clases



AbstractFactory: declara una interfaz para la creación de objetos de productos abstractos.

ConcreteFactory: implementa las operaciones para la creación de objetos de productos concretos.

AbstractProduct: declara una interfaz para los objetos de un tipo de productos.

ConcreteProduct: define un objeto de producto que la correspondiente factoría concreta se encargaría de crear, a la vez que implementa la interfaz de producto abstracto.

NOMBRE	
AREA	Software
REGIMEN	Semestral

Client: utiliza solamente las interfaces declaradas en la factoría y en los productos abstractos.

Una única instancia de cada FactoryConcreto es creada en tiempo de ejecución. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

Ahora que explique que rol ocupa cada uno en el diagrama, les pido un poco de atención en lo siguiente: veamos que relación tienen los FactoryConcretos con respectos a los productos. Esto es, FactoryConcreto1 crea una relación entre un producto de la familia A y un producto de la familia B. Y, por otro lado, tenemos que el FactoryConcreto2 crea una relación entre otros dos productos de ambas familias.

Esto ya debería darnos una pista sobre el funcionamiento del AbstractFactory: se crea una clase por cada relación que necesitemos crear. Esto quedará más claro en el ejemplo a continuación.

Ejemplo

Hagamos de cuenta que tenemos dos familias de objetos:

- 1) La clase TV, que tiene dos hijas: Plasma y LCD.
- 2) La clase Color, que tiene dos hijas: Amarillo y Azul (los mejores colores, sin duda!).

Más alla de todos los atributos/métodos que puedan tener la clase Color y TV, lo importante aquí es destacar que Color define un método abstracto:

```
public abstract void colorea(TV tv);
```

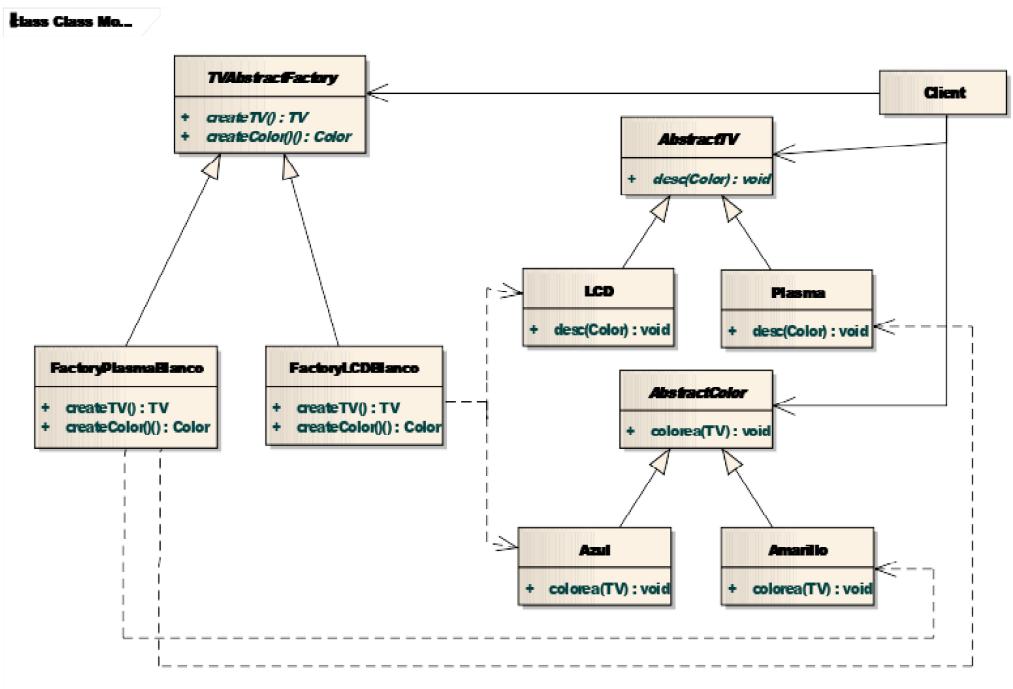
Este método es la relación que une las dos familias de productos. Dado que es un método abstracto, Azul debe redefinirlo:

```
public void colorea(TV tv) {
    System.out.println("Pintando de azul el "+ tv.getDescripcion());
```

NOMBRE	
AREA	Software
REGIMEN	Semestral
}	

Lo mismo ocurre con Amarillo:

```
public void colorea(TV tv) {
    System.out.println("Pintando de amarillo el "+ tv.getDescripcion());
}
```



Bien, veamos las clases correspondientes antes de continuar con nuestros ejemplos.

NOMBRE		
AREA	Software	REGIMEN Semestral

```

public abstract class Color {
    private String descripcion;

    public abstract void colorea(TV tv);

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

}

public class Amarillo extends Color {
    private boolean esPrimario;

    @Override
    public void colorea(TV tv) {
        System.out.println("Pintando de amarillo el " + tv.getDescripcion());
    }

    public boolean isEsPrimario() {
        return esPrimario;
    }

    public void setEsPrimario(boolean esPrimario) {
        this.esPrimario = esPrimario;
    }

}

public class Azul extends Color {
    private boolean esPrimario;

    @Override
    public void colorea(TV tv) {
        System.out.println("Pintando de azul el " + tv.getDescripcion());
    }

    public boolean isEsPrimario() {
        return esPrimario;
    }

    public void setEsPrimario(boolean esPrimario) {
        this.esPrimario = esPrimario;
    }

}

```

NOMBRE		
AREA	Software	REGIMEN Semestral

```

public abstract class TV implements Cloneable {
    private String marca;
    private int pulgadas;
    private String color;
    private String descripcion;
    private double precio; // todos con sus get y set

    public TV(){
    }

    public TV(String marca, int pulgadas, String color, double precio) {
        setMarca(marca);
        setPulgadas(pulgadas);
        setPrecio(precio);
        setColor(color);
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class LCD extends TV {
    private double costoFabricacion; // con sus get y set

    public LCD(String marca, int pulgadas, String color, double precio,
               double costoFabricacion) {
        super(marca, pulgadas, color, precio);
        setCostoFabricacion(costoFabricacion);
    }

    public LCD(){
        setDescripcion("LCD");
    }

    public double getCostoFabricacion() {
        return costoFabricacion;
    }

    public void setCostoFabricacion(double costoFabricacion) {
}

```

NOMBRE		
AREA	Software	REGIMEN Semestral

```

public class Plasma extends TV {
    private double anguloVision;
    private double tiempoRespuesta; // todos con sus get y set

    public Plasma(String marca, int pulgadas, String color, double precio,
                  double anguloVision, double tiempoRespuesta) {

        super(marca, pulgadas, color, precio);
        setAnguloVision(anguloVision);
        setTiempoRespuesta(tiempoRespuesta);
    }

    public Plasma(){
        setDescripcion("Plasma... proximamente será un LED");
    }

    public double getAnguloVision() {
        return anguloVision;
    }
}

```

Escenario: nuestra empresa se dedica a darle un formato estético específico a los televisores LCD y Plasma. Se ha decidido que todos los LCD que saldrán al mercado serán azules y los plasma serán amarillos. Ahora bien, una solución simple sería en la clase Azul colocar el LCD y en la clase Amarillo colocar el Plasma y todo funcionaría de maravillas. ¿Cuál sería el problema? Que esta todo hardcodeado. Esto quiere decir que el hecho de que los LCD sean azules y los plasmas amarillos es una decisión del negocio y, como tal, puede variar (y de hecho el negocio varía constantemente).

Por ejemplo, que pasa si mañana se agrega otro color o me cambian el color del LCD o mucho peor, ¿que pasa si se crea otro producto LED y también se lo quiere pintar de Azul?

Para evitar un dolor de cabeza conviene separar estas familias y utilizar el Abstract Factory:

```

public abstract class TvAbstractFactory {

    public abstract TV createTV();
    public abstract Color createColor();
}

```

Y los FactoryConcretos, que relacionan las familias:

NOMBRE		
AREA	Software	REGIMEN Semestral

```

public class FactoryLcdAzul extends TvAbstractFactory {

    public Color createColor() {
        return new Azul();
    }

    public TV createTV() {
        return new LCD();
    }
}

public class FactoryPlasmaAmarillo extends TvAbstractFactory {

    public Color createColor() {
        return new Amarillo();
    }

    public TV createTV() {
        return new Plasma();
    }
}

```

Consecuencias

Se oculta a los clientes las clases de implementación: los clientes manipulan los objetos a través de las interfaces o clases abstractas.

Facilita el intercambio de familias de productos: al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta.

Mejora la consistencia entre productos: el uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.

Como inconveniente podemos decir que no siempre es fácil soportar nuevos tipos de productos si se tiene que extender la interfaz de la Factoría abstracta.