V. 1.1

Diseño de Sistemas

Unidad 3: Diseño con Objetos - parte 3

Pablo Sabatino Martín Agüero

2019

UTN.BA
DPTO. INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DISEÑO DE SISTEMAS

Unidad 3:

Patrones de Diseño: Patrones Creacionales. Patrones estructurales. Patrones de comportamiento.

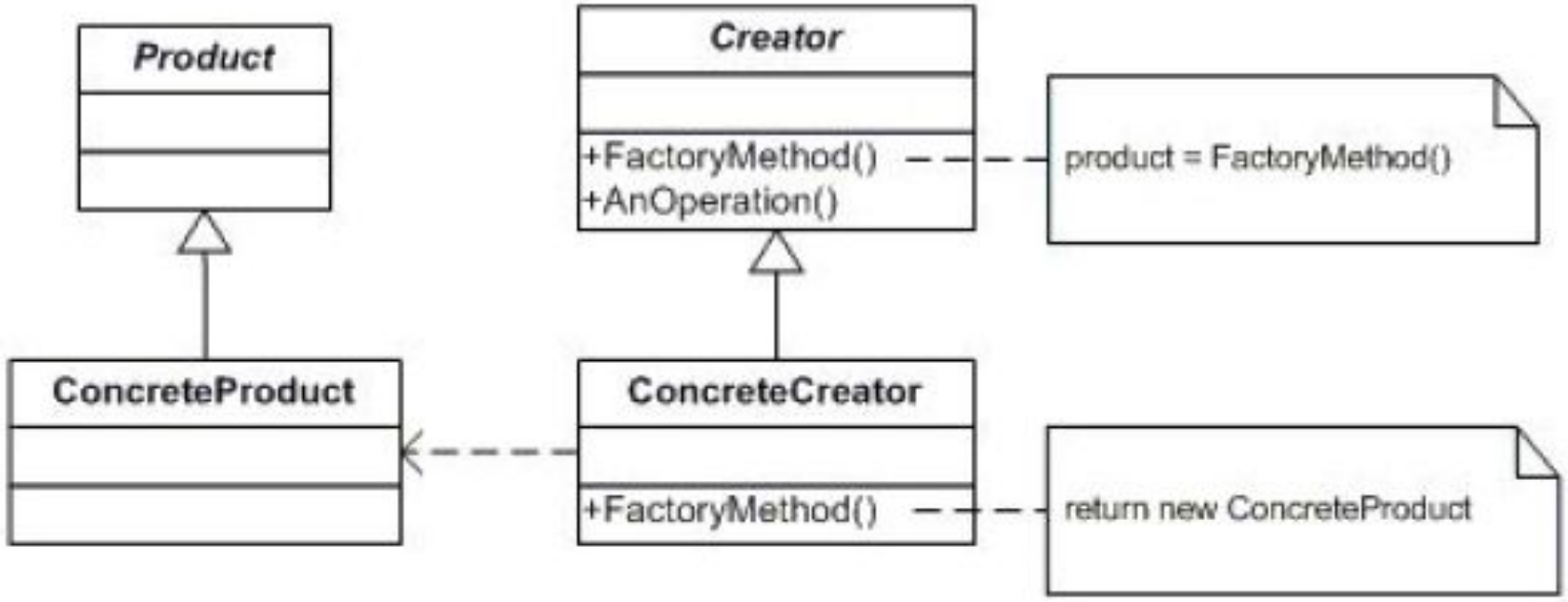
Patrones de Diseño

Factory Method

Patrones de Diseño

Factory Method

Propósito: Definir una interfaz para la creación de un objeto, pero delegando a las subclases la responsabilidad de crear las instancias apropiadas.



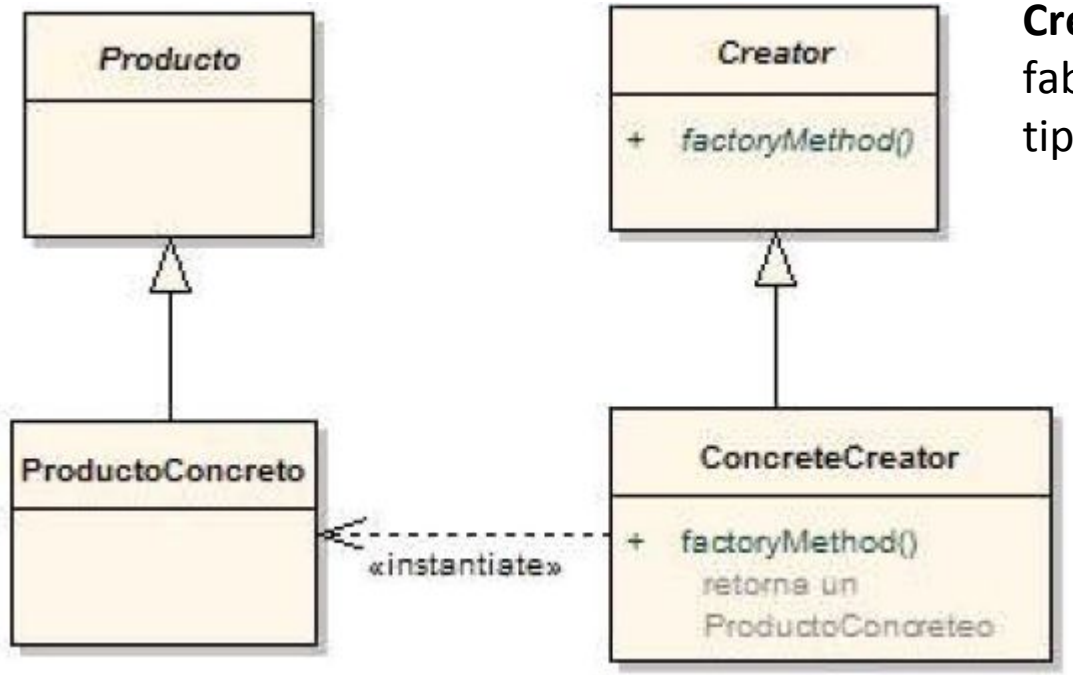
Creacional

Factory Method

- Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear. Requiere que deba delegar la responsabilidad a una subclase.
- Podemos utilizar este patrón cuando definamos una clase a partir de la que se crearán objetos pero sin saber de qué tipo son, siendo otras subclases las encargadas de decidirlo.
- Se puede utilizar para solucionar el problema que se presenta cuando tenemos que crear un objeto pero a priori no sabemos de que tipo de objeto tiene que ser. Esto podría ser, porque depende de alguna opción que seleccione el usuario en la aplicación.

Patrones de Diseño

Factory Method



Creator declara el método de fabricación, que devuelve un objeto de tipo **Producto.**

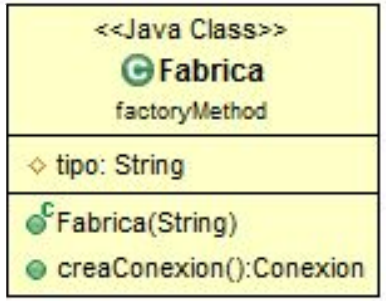
ConcreteCreator redefine el método de fabricación para devolver un objeto **ProductoConcreto**.

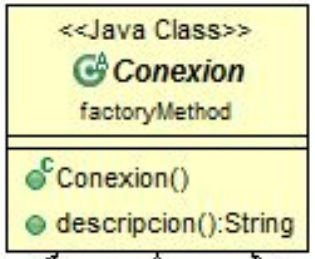
Creacional

Patrones de Diseño

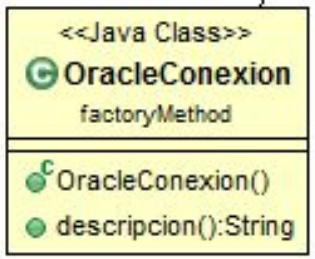
Factory Method

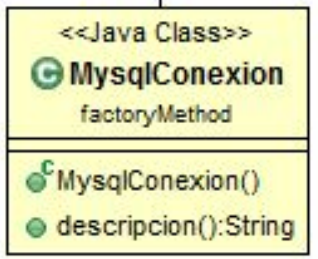
```
public class Fabrica {
    protected String tipo;

    public Fabrica(String t){
        tipo = t;
    }
    //retorna un objeto tipo de conexion
    public Conexion creaConexion(){
        if (tipo.equalsIgnoreCase("Oracle")){
            return new OracleConexion();
        } else if (tipo.equalsIgnoreCase("Mysql")){
            return new MysqlConexion();
        } else {
            return new MongoConexion();
        }
    }
}
```

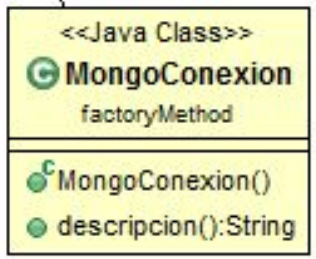





```
public abstract class Conexion {
    public Conexion(){}

    public String descripcion(){
        return "Conexion Generica";
    }
}
```








```
public class OracleConexion extends Conexion {

    public OracleConexion(){};

    public String descripcion(){
        return "...connected with Oracle";
    }
}
```

```
public class MysqlConexion extends Conexion {
    public MysqlConexion(){};

    public String descripcion(){
        return "...connected with MYSQL";
    }
}
```

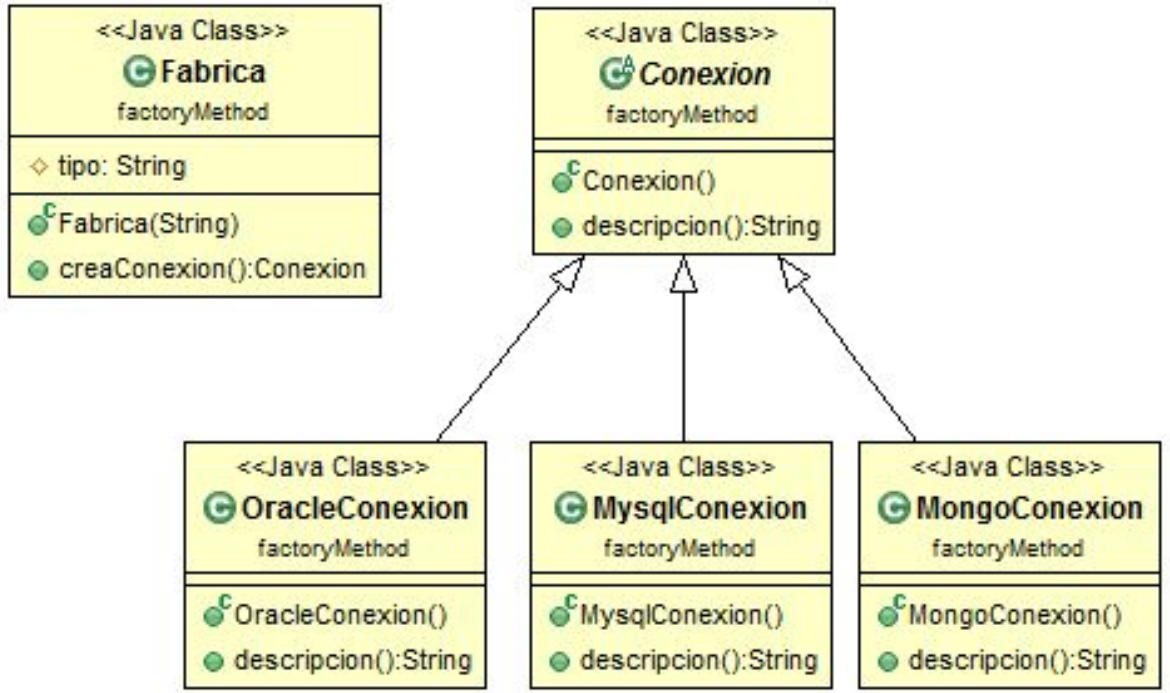
```
public class MongoConexion extends Conexion {
    public MongoConexion(){};

    public String descripcion(){
        return "Connected with MongoDB";
    }
}
```

Creacional

Patrones de Diseño

Factory Method

```
public class factoryMethodTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Fabrica miFabrica;
        Conexion miConexion;

        miFabrica = new Fabrica("Oracle");
        miConexion = miFabrica.creaConexion();

        String salida = "Esta conectado con " + miConexion.descripcion();
        System.out.println(salida);
    }
}
```



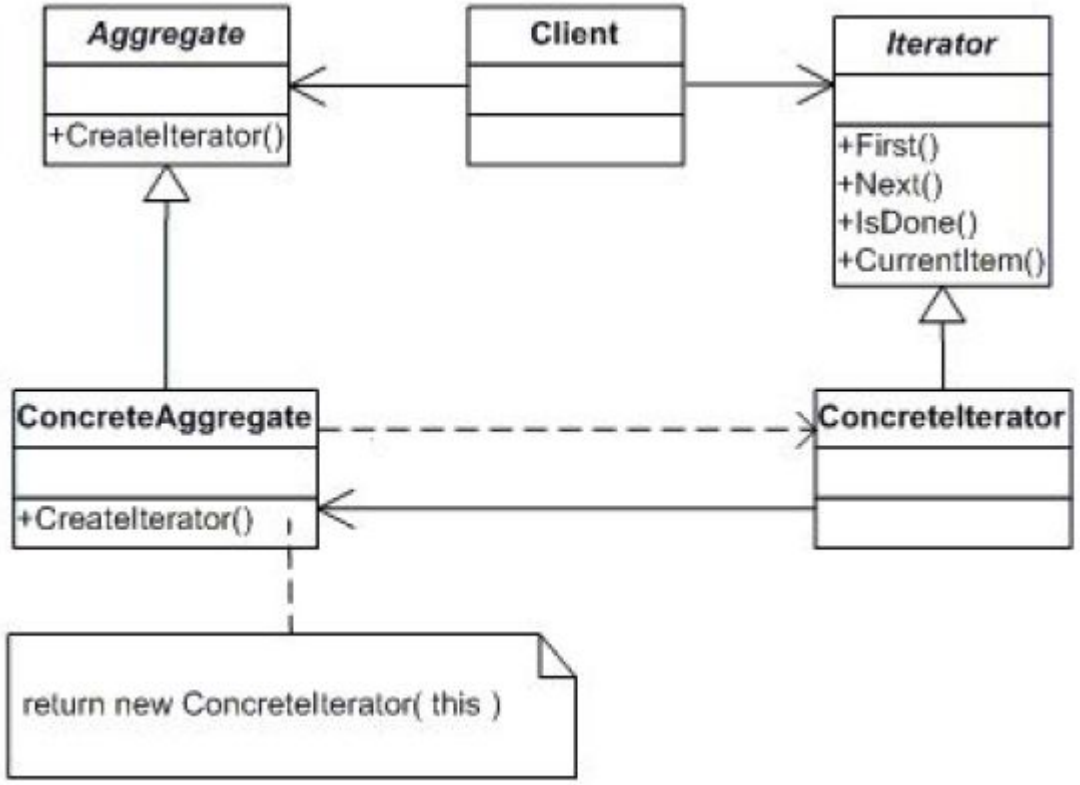
Creacional

Patrones de Diseño

Iterator

Iterator

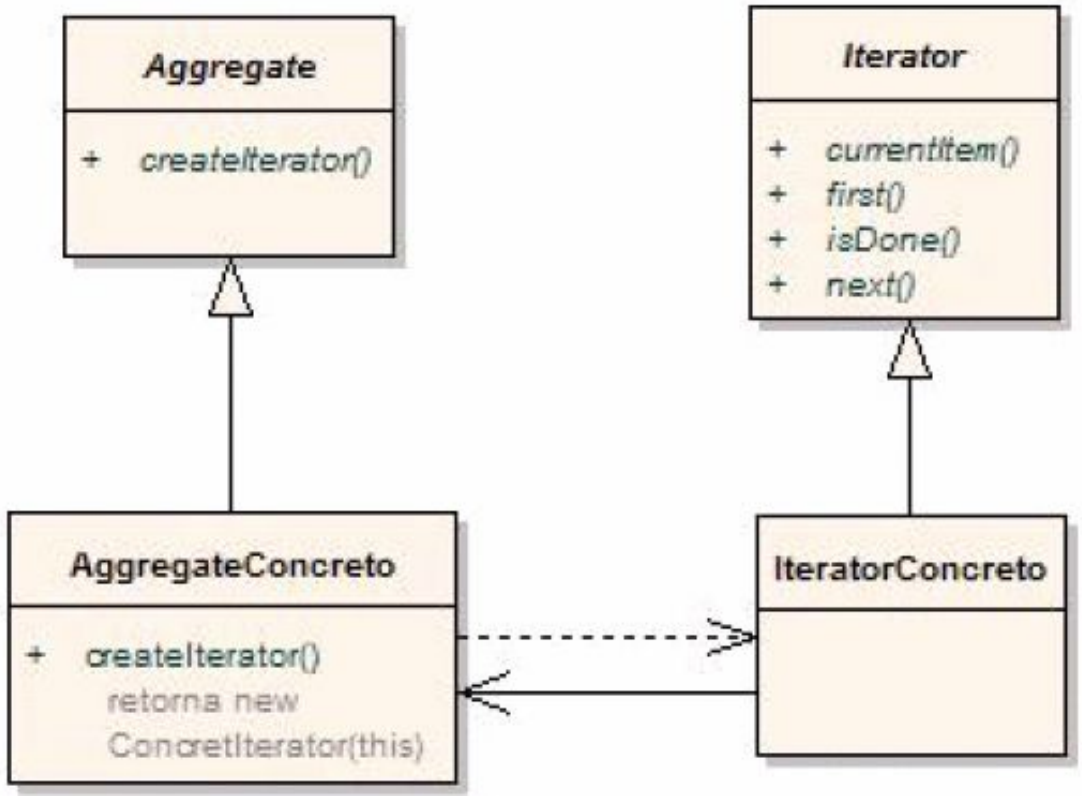
Propósito: Proveer una forma de acceder a los elementos de una colección de objetos en orden secuencial, sin exponer la implementación real de la colección.



Comportamiento

Patrones de Diseño

Iterator



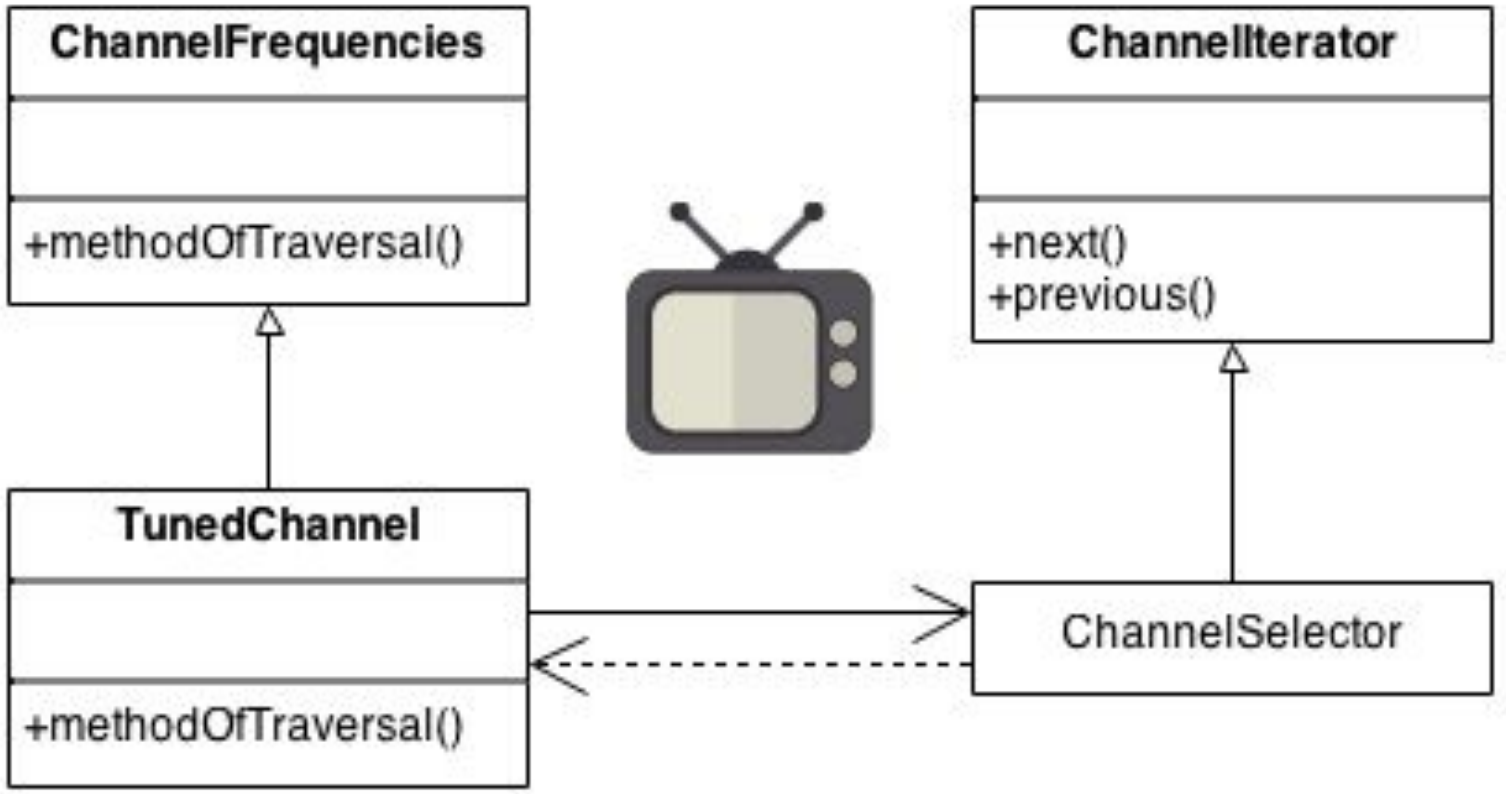
El iterator se encarga de saber cómo iterar la colección, siendo esta implementación totalmente transparente para el cliente.

Comportamiento

Iterator

- Define un interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase accede a una colección solamente a través de un interface independiente de la clase que implementa el interface.
- Una clase necesita un modo uniforme de acceder al contenido de diferentes colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.

Un ejemplo: En los antiguos televisores, se utilizaba un dial para cambiar de canal. En los televisores modernos, se utiliza un botón siguiente y anterior para iterar secuencialmente entre la colección de canales.



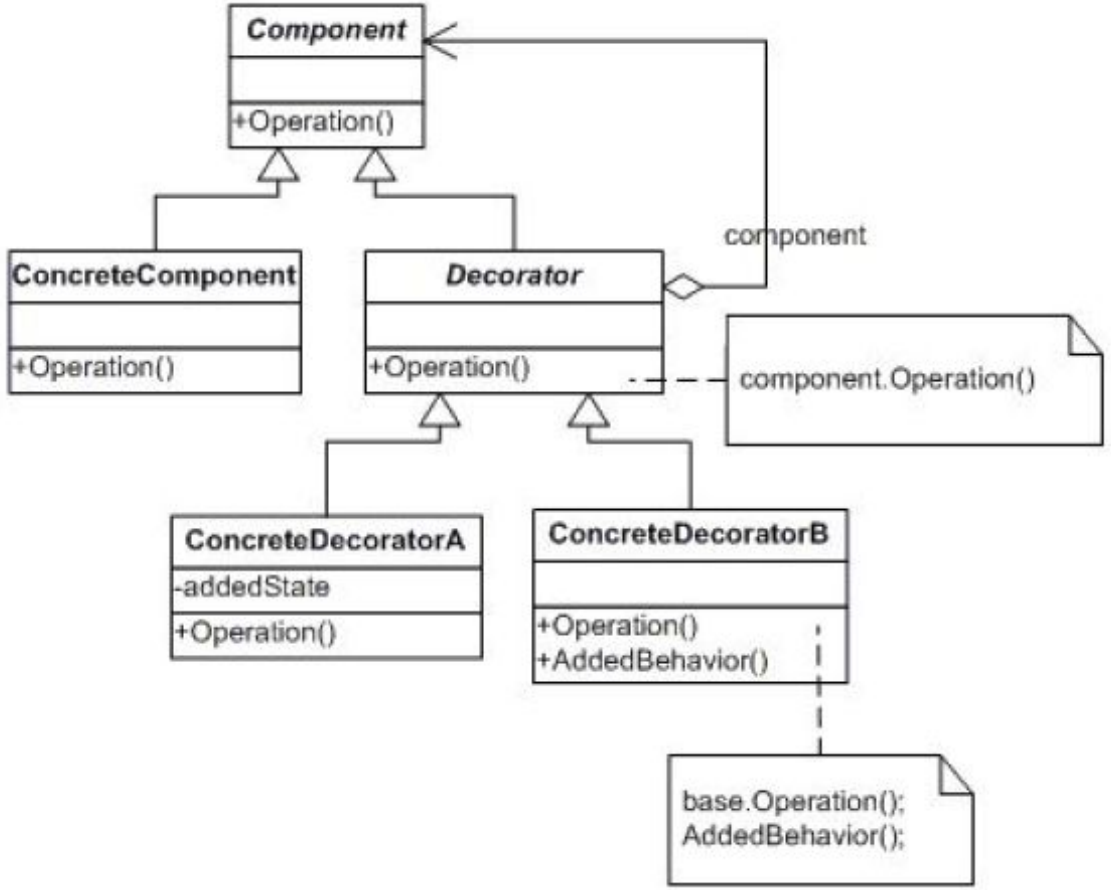
Patrones de Diseño

Decorator

Patrones de Diseño

Decorator

Propósito: Agregar dinámicamente responsabilidades (funcionalidad) extra a un objeto. Es una forma flexible que sirve de alternativa a subclassing para extender funcionalidad.



Estructural

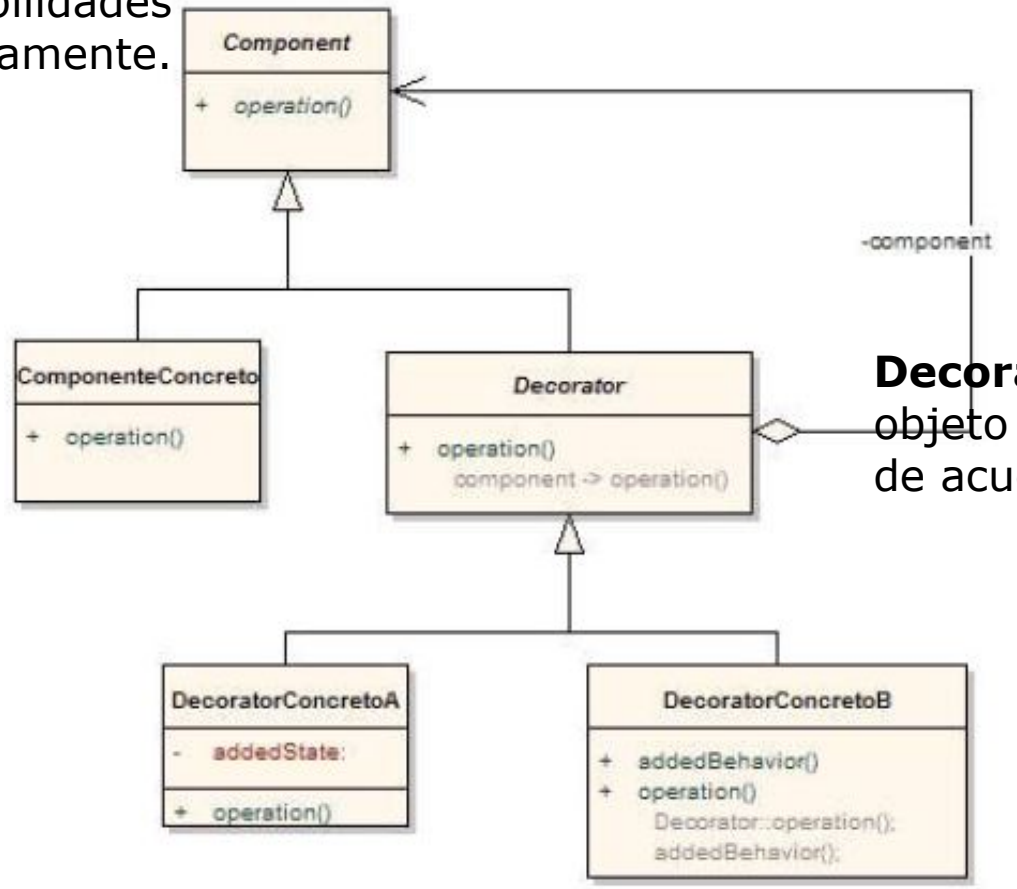
Decorator

- El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que *la herencia* para extender las funcionalidades.
- En algunas situaciones se desea adicionar responsabilidades a un objeto pero no a toda la clase.

Patrones de Diseño

Decorator

Component define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.



ComponenteConcreto define el objeto al que se le puede adicionar una responsabilidad.

Decorator mantiene una referencia al objeto **Component** y define una interface de acuerdo con la interface de Component.

DecoratorConcreto adiciona la responsabilidad al Component.

Estructural

Patrones de Diseño

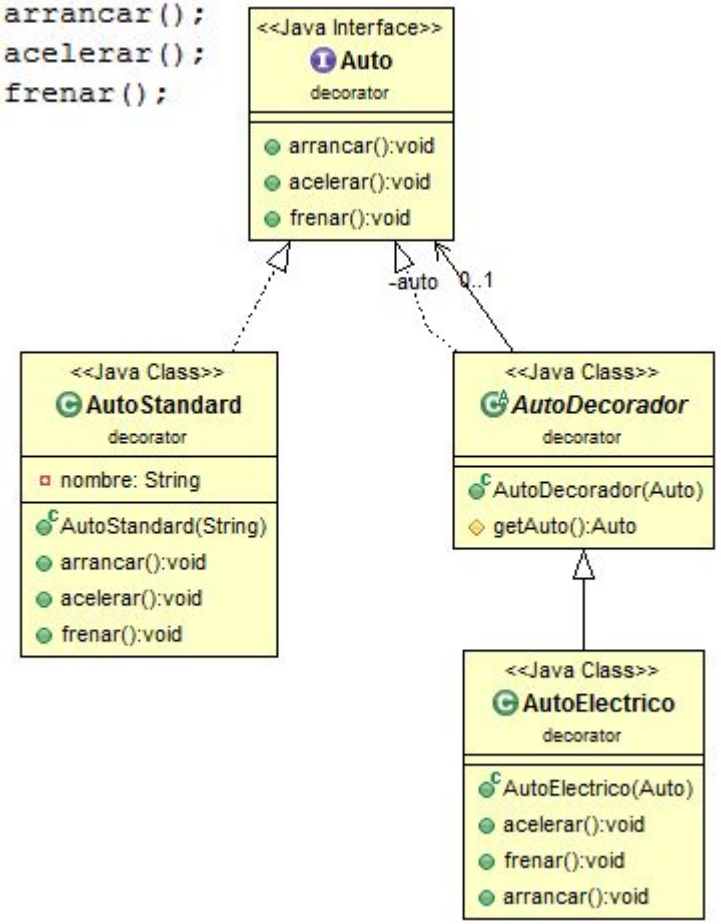
Decorator

ComponenteConcreto define el objeto al que se le puede adicionar una responsabilidad.

```
public class AutoStandard implements Auto{

    private String nombre;

    public AutoStandard(String autoCualquiera){
        nombre = autoCualquiera;
    };

    public void arrancar(){
        System.out.println("Arrancando el " + nombre);
    };

    public void acelerar(){
        System.out.println("Acelerando el " + nombre);
    };

    public void frenar(){
        System.out.println("frenando 3 2 1..." + nombre + " detenido");
    };


}
```

```
public interface Auto {
    public void arrancar();
    public void acelerar();
    public void frenar();
}
```

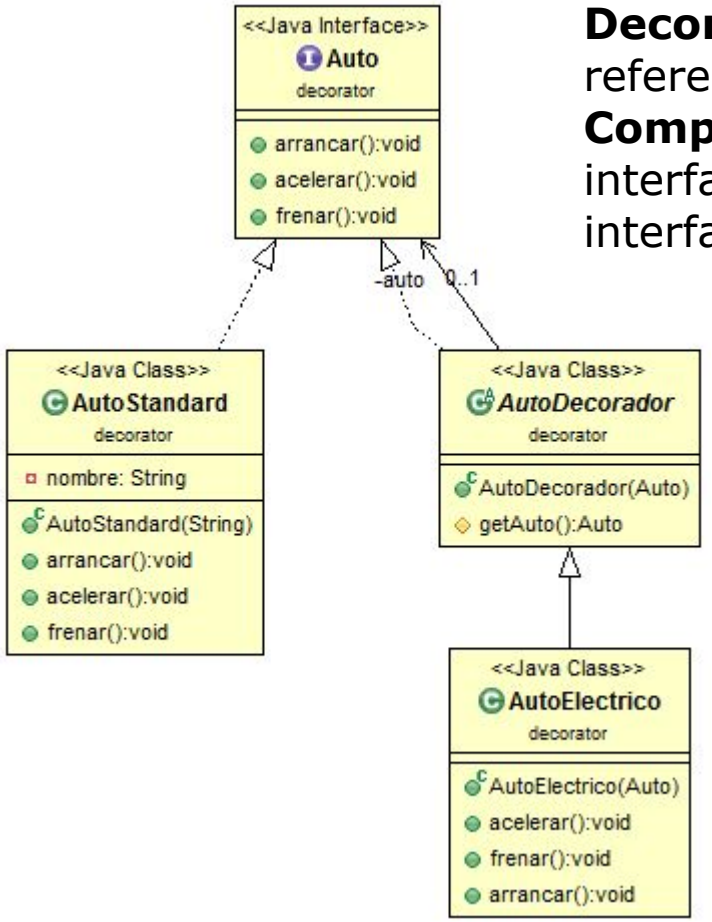
Component define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.



Estructural

Patrones de Diseño

Decorator



Decorator mantiene una referencia al objeto **Component** y define una interface de acuerdo con la interface de **Component**.

```
public abstract class AutoDecorador implements Auto{
    private Auto auto;

    public AutoDecorador(Auto auto){
        this.auto = auto;
    }

    protected Auto getAuto(){
        return auto;
    }
}
```

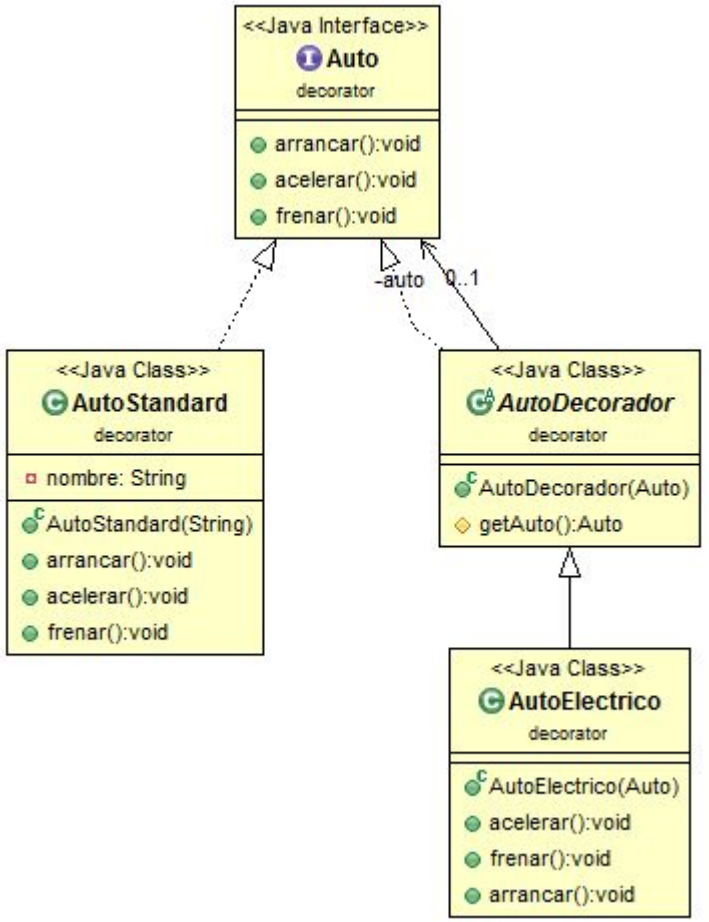
```
public class AutoElectrico extends AutoDecorador{
    public AutoElectrico(Auto cualquier_auto){
        super(cualquier_auto);
    }
    public void acelerar(){
        System.out.println("AutoElectrico:  Acelerando Electricamente");
        getAuto().acelerar();
    }
    public void frenar(){
        System.out.println("AutoElectrico:  Frenando Electricamente");
        getAuto().frenar();
    }
    public void arrancar(){
        System.out.println("AutoElectrico: Arrancando Electricamente");
        getAuto().arrancar();
    }
}
```

DecoratorConcreto adiciona la responsabilidad al Component.

Estructural

Patrones de Diseño

Decorator




```
public class decoratorTEST {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Auto auto = new AutoStandard("peugeot");
        auto.arrancar();
        auto.acelerar();
        auto.frenar();
        // Auto electrico //
        Auto auto_decorado = new AutoElectrico( new AutoStandard("peugeot 206"));
        auto_decorado.acelerar();
        auto_decorado.arrancar();
        auto_decorado.frenar();
        }
}
```

Estructural

Patrones de Diseño

Ejercicio grupal

El objetivo es diseñar un cliente de email (similar al Mozilla Thunderbird o MS Outlook). Este cliente manejará una única cuenta de mail. Debe permitir enviar, recibir, copiar y mover emails entre carpetas. Una cuenta tiene una carpeta de entrada predeterminada (Inbox) y una carpeta en la cual se almacenan los mensajes enviados (Sent Mails). Algunas de las características con las que se deberá contar son:

- El usuario puede exportar los mensajes y carpetas en diferentes formatos. Por ejemplo, exportar las carpetas a directorios y subdirectorios y los emails a archivos de texto dentro de los directorios, o exportar a un único archivo XML.
- Para descargar los mensajes, el cliente debe conectarse con el servidor de correo. Existen diferentes servidores de email o Mail Transfer Agents (MTA), por ejemplo, Sendmail y Postfix. Existen clases que se conectan directamente con estos servidores, la clase SendmailConnection y la clase PostfixConnection, pero ambas definen un método diferente para recibir email, por ejemplo: en SendmailConnection el método es getMail() y en PostfixConnection el método es lookupEmail(). Es deseable que el cliente de mail pueda usar cualquiera de estas clases de manera indistinta.

Proponer un diseño que pueda incluir uno o más Patrones de Diseño. Comunicar la solución con un diagrama de clases.

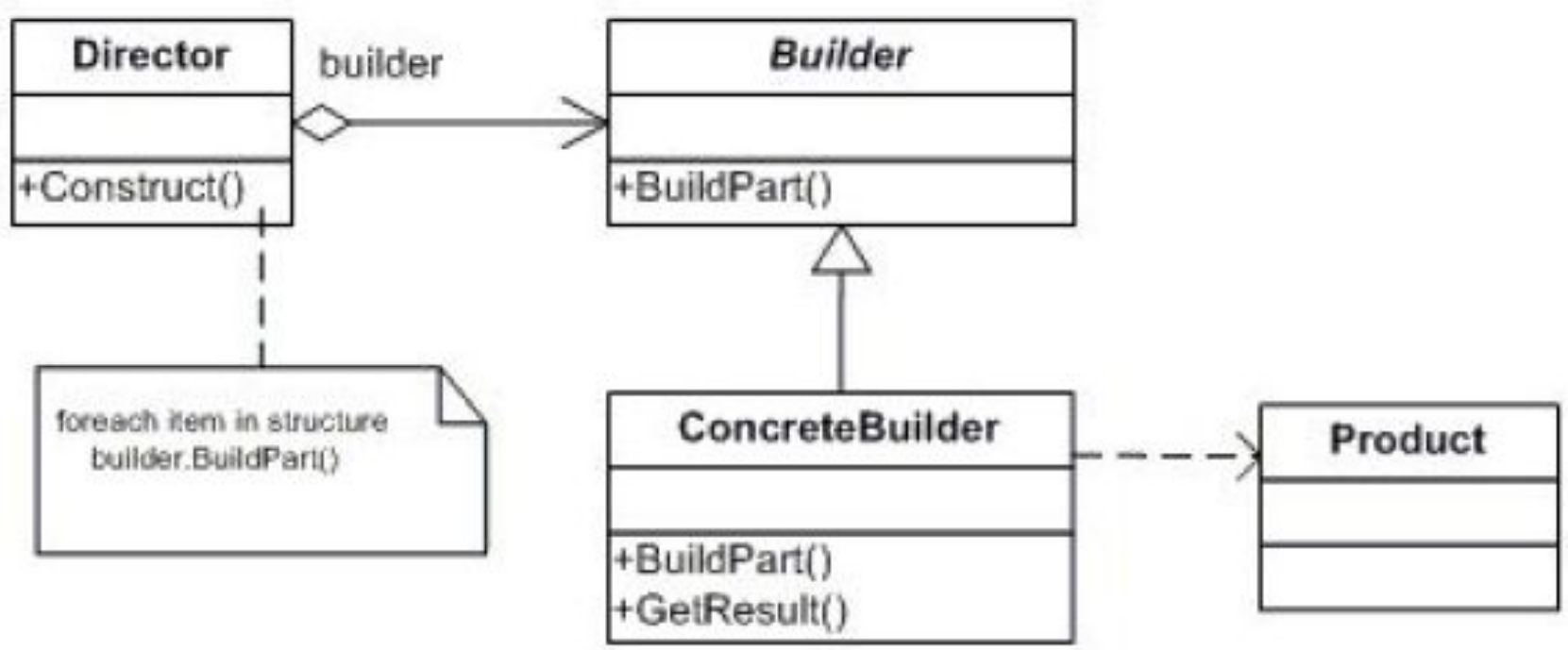
Patrones de Diseño

Builder

Patrones de Diseño

Builder

Propósito: Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.



Creacional

Proceso de ensamblaje
Diferentes configuraciones

Builder

- Su objetivo es instanciar objetos complejos que en general están compuestos por varios elementos y que admiten diversas configuraciones.

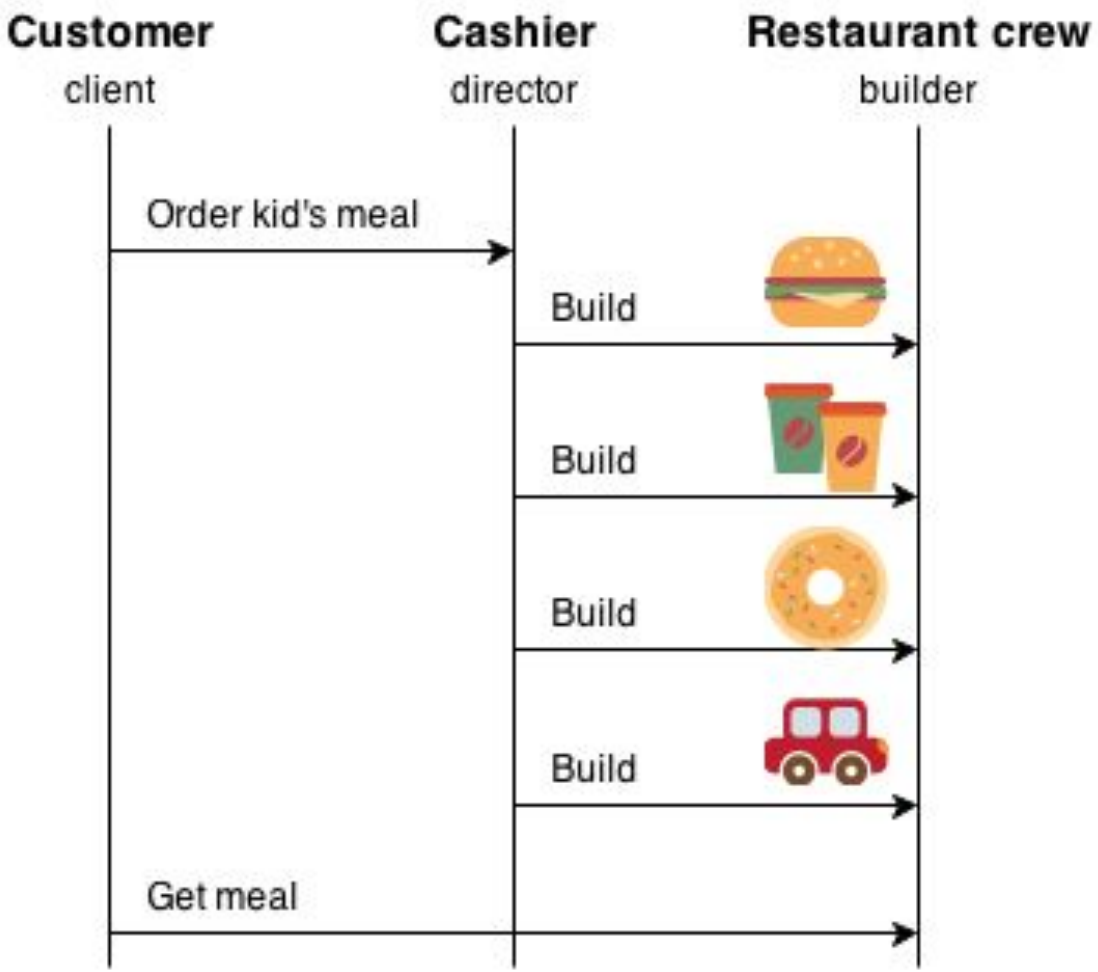
- Cuando hablamos de construcción nos referimos al proceso, mientras que cuando hablamos de representación nos estaremos refiriendo a los datos que componen el objeto.

Builder

- Permite la creación de una variedad de objetos complejos desde un objeto fuente, el cual se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo.
- Un único proceso de construcción debe ser capaz de construir distintos objetos complejos, abstrayéndonos de los detalles particulares de cada uno de los tipos.

Patrones de Diseño

Builder



Creacional

Builder

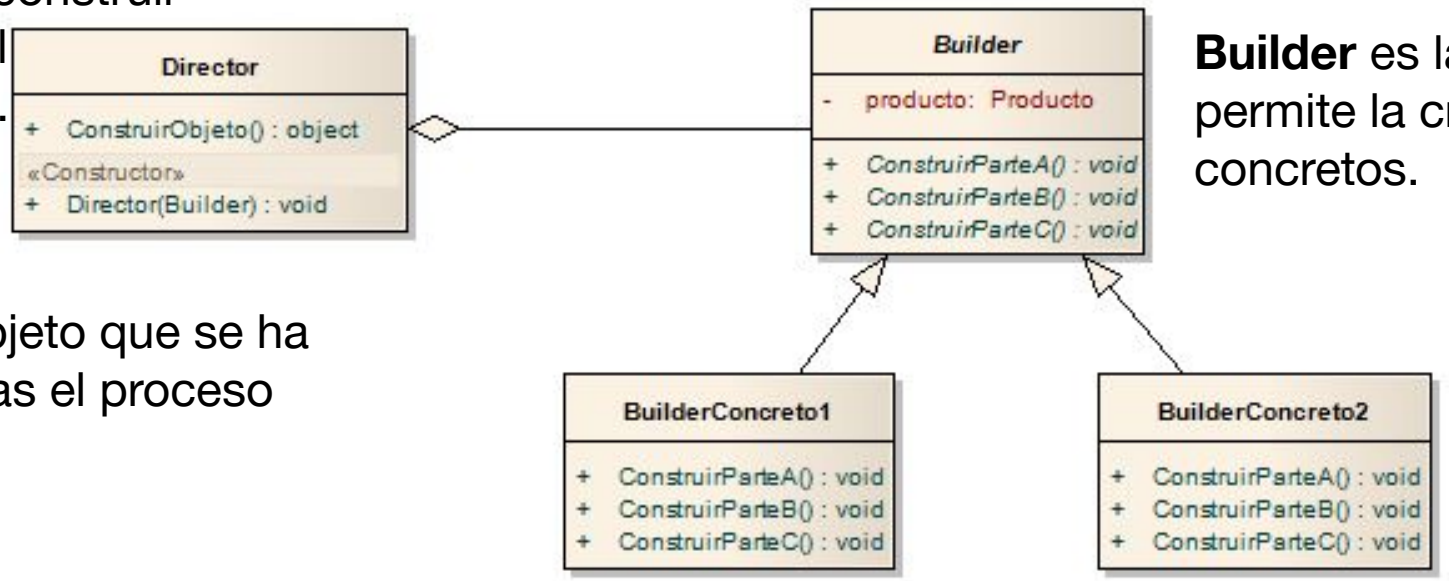
Se usa cuando:

- Nuestro sistema trata con objetos complejos (compuestos por muchos atributos) pero el número de configuraciones es limitada.
- El algoritmo de creación del objeto complejo puede independizarse de las partes que lo componen y del ensamblado de las mismas.
- La solución será crear un constructor que permita construir todos los tipos de objetos, ayudándose de constructores concretos encargados de la creación de cada tipo en particular. Un objeto director será el encargado de coordinar y ofrecer los resultados.

Patrones de Diseño

Builder

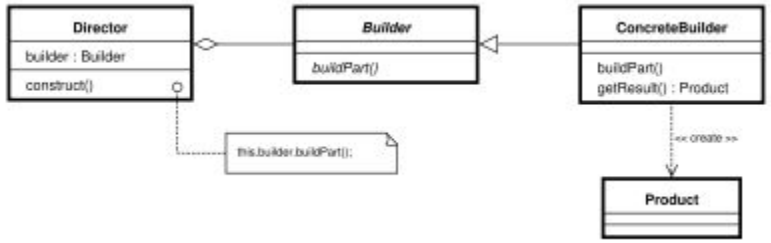
Director encarga de construir un objeto utilizando el Constructor (***Builder***).



Builder es la Interfaz abstracta que permite la creación de objetos concretos.

Producto: objeto que se ha construido tras el proceso definido.

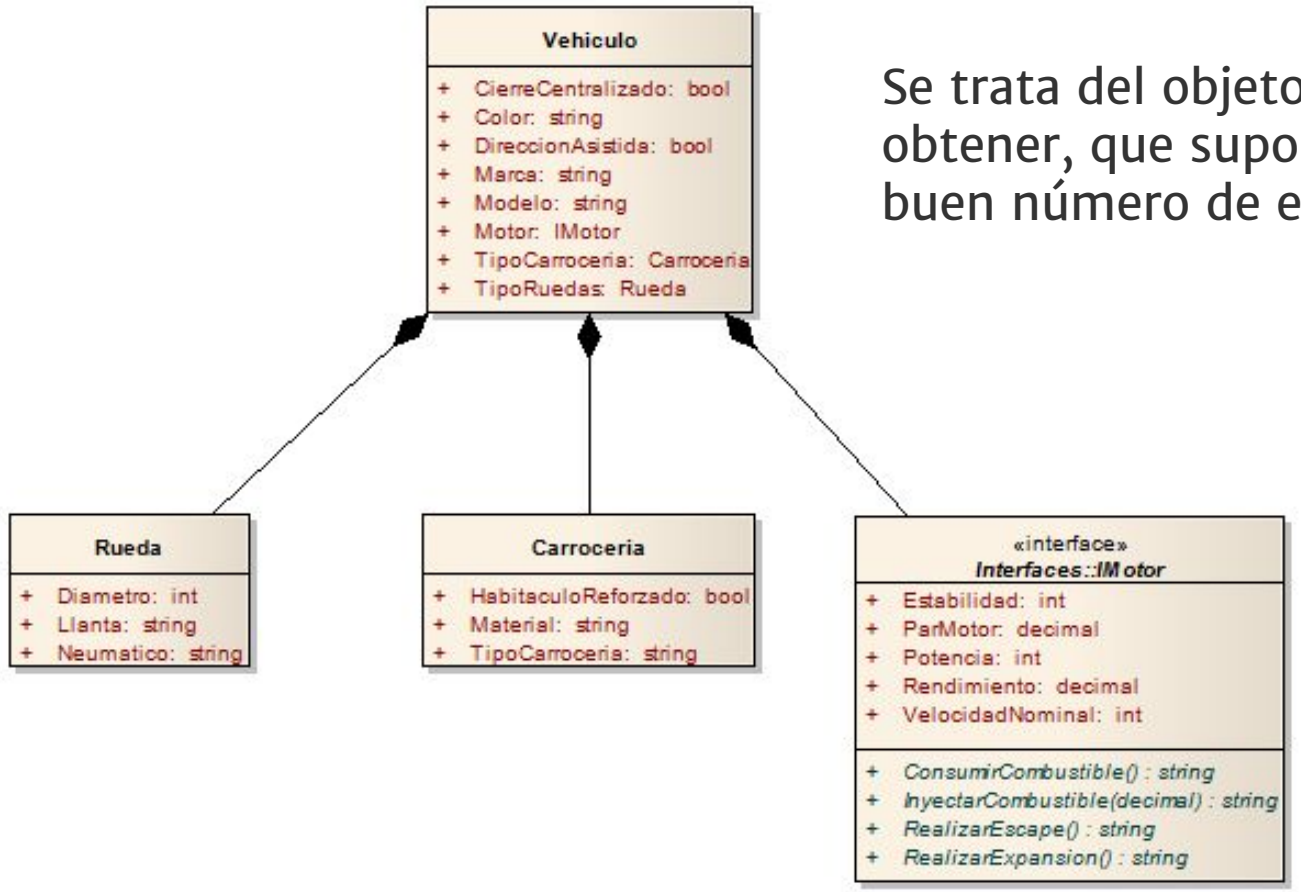
BuilderConcreto es la implementación concreta del ***Builder*** definida para cada uno de los tipos. Permite crear el objeto concreto recopilando y creando cada una de las partes que lo compone. Ensambla las partes que constituyen el objeto complejo.



Creacional

Patrones de Diseño

Builder



Se trata del objeto complejo (Vehículo) que queremos obtener, que suponemos que estará compuesto por un buen número de elementos.

Un único proceso de construcción debe ser capaz de construir distintos objetos complejos, abstrayéndonos de los detalles particulares de cada uno de los tipos.

Creacional

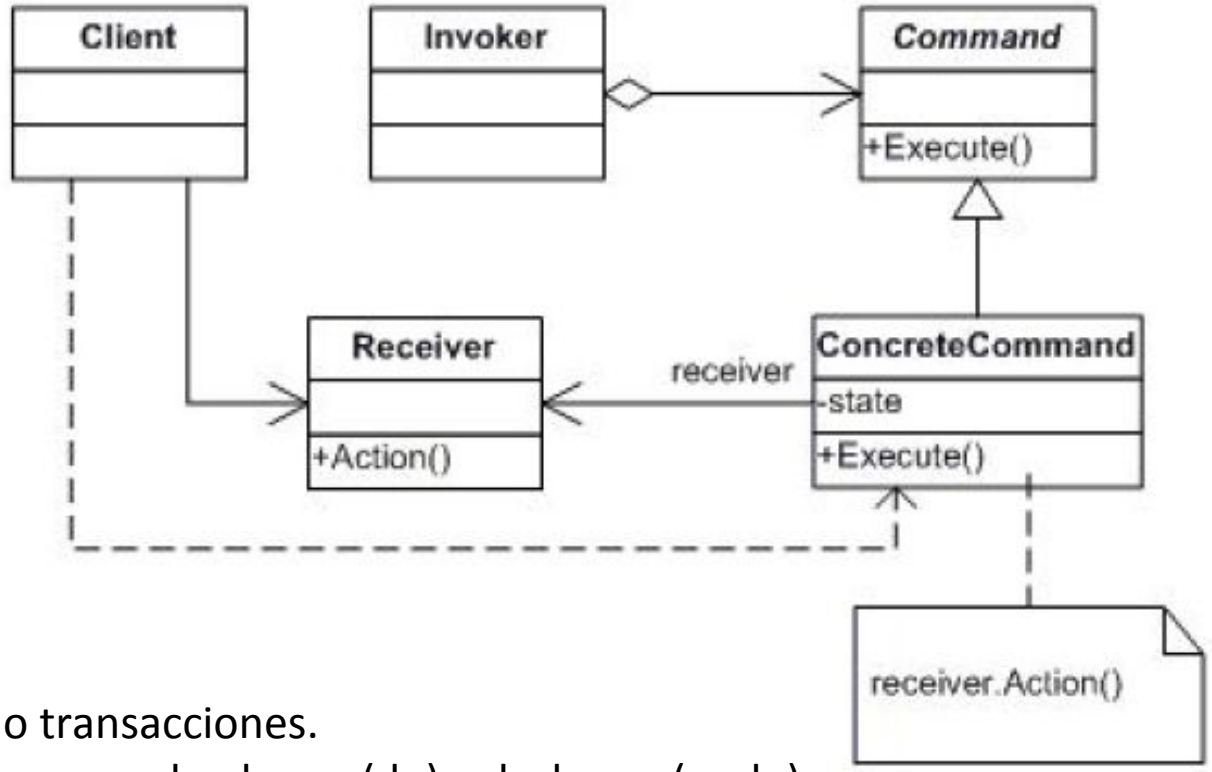
Patrones de Diseño

Command

Patrones de Diseño

Command

Propósito: Encapsular una solicitud en un objeto, permitiendo que ésta sea parametrizable por las clases clientes, encolada o ser deshecha (undo).



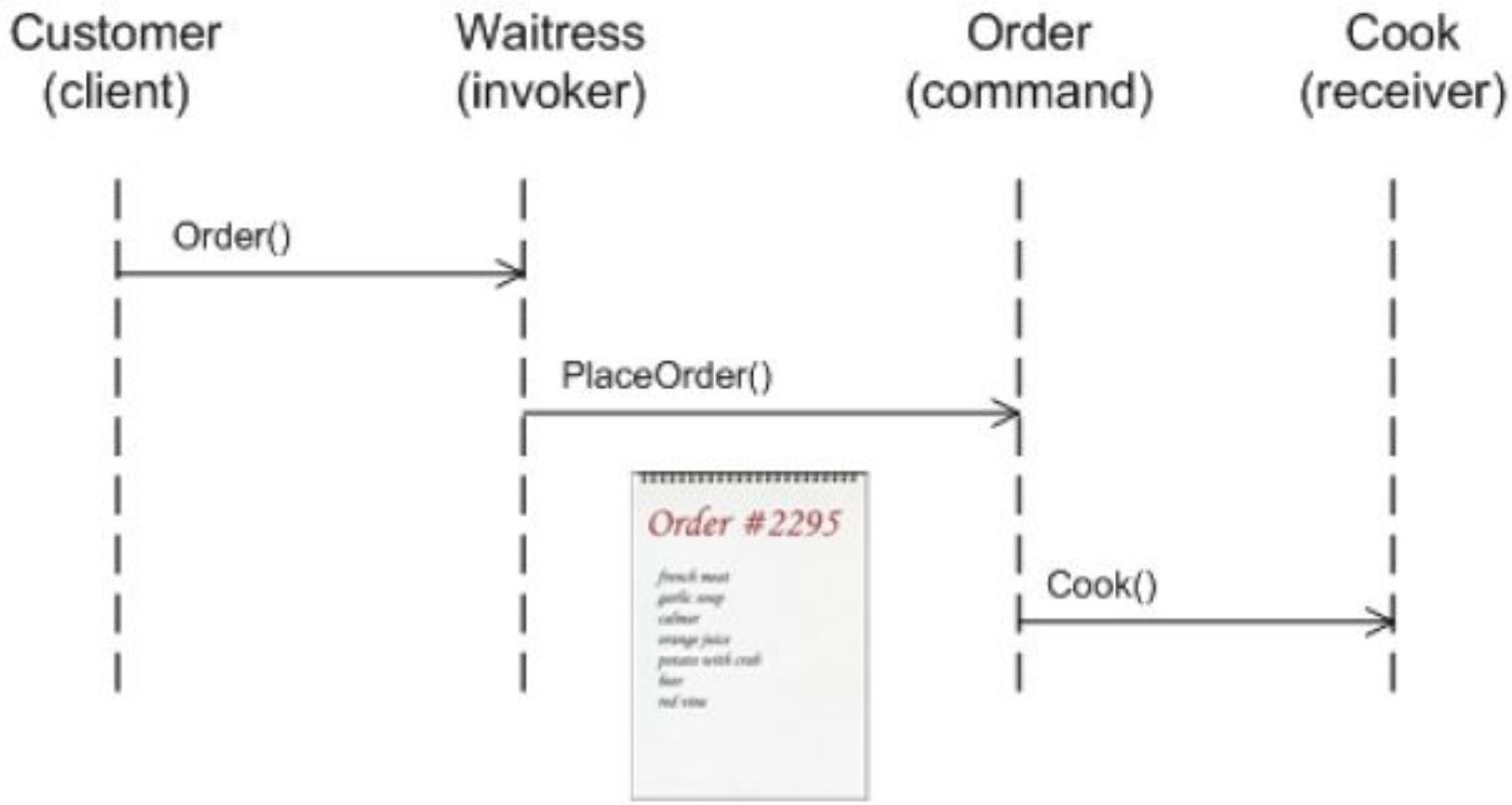
Se usa para:

- Encolar (queue) acciones o transacciones.
- Atomizar operaciones que se pueden hacer (do) y deshacer (undo).

Comportamiento

Patrones de Diseño

Command



Ventajas:

- Desacoplamiento de las clase cliente y la clase que realiza la acción.
- Se pueden agregar nuevos Commands sin modificar la estructura existente.

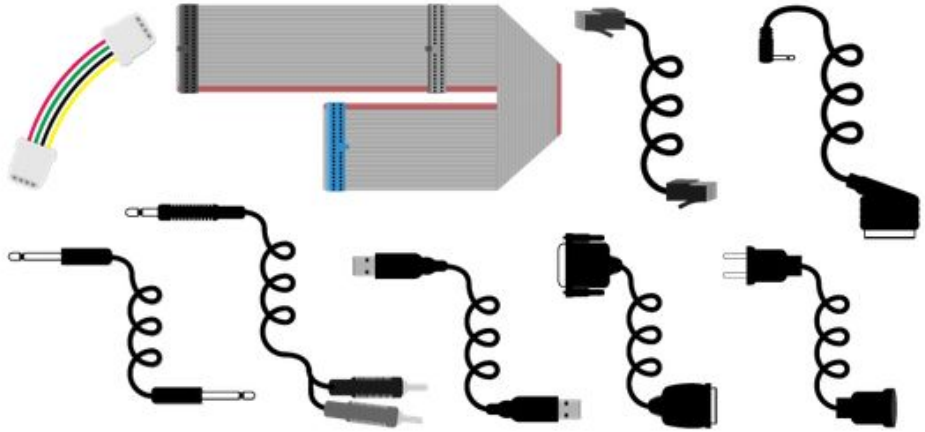
Comportamiento

Patrones de Diseño

Ejercicio

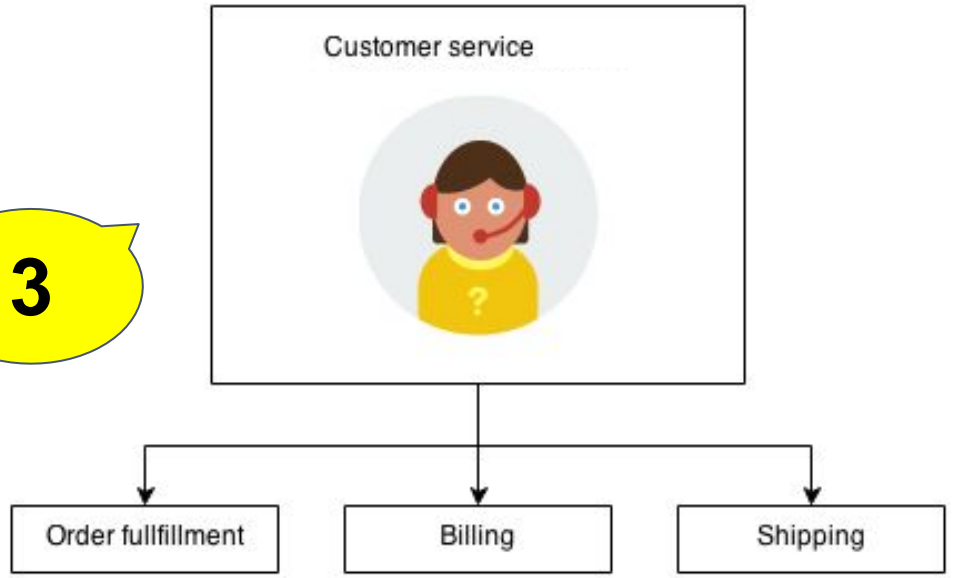
¿Con cuáles patrones de diseño se pueden asociar estas figuras?

1



2

3



Consejo: Consultar apunte de referencia.

Referencias

Gamma, E. et al., Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley. 1994.

Metsker, S., The Design Patterns Java Workbook.
Addison-Wesley. 2002.

Shvets, A., Design Patterns Explained Simply.
Source Making. 2015.

Do Factory http://www.dofactory.com/

SourceMaking https://sourcemaking.com