*Un documento en pdf que contenga 2 patrones de diseño explicados con 2 ejemplos de la vida real y con código fuente.

PATRON BUILDER

*Separa la especificación de un Objecto complejo en su construcción

*El mismo proceso de construcción puede crear diferentes representaciones.

*Se basa en directores y Constructores

*Los Constructores son llamados por el director para crear un producto de acuerdo a unas especificaciones

*El Constructor provee partes que se acumulan hasta que el director finalizó.

*Diferentes Constructores pueden proveer diferentes partes.

*Pueden existir diferentes constructores y diferentes directores.

Partes que Interviene

*__IConstructor__
- Interfaz que define que se puede construir

*__Director__
- Una secuencia de operaciones que se sigue para crear un producto.

__*Constructor__
- Una Clase que es invocada por el Director para crear las partes del producto.

__*Producto__
- El objecto bajo construcción.

Ejemplo 1

En una Pizzería se ofrecen varios tipos de pizza y existen tipos de pizza ya establecidos.

Objecto de una sola clase Producto (pizza)

Builder le asigna valores a los determinados productos, para establecer los atributos de la clase producto de una clase abstracta.

*Métodos que permiten asignar valores a los atributos de la clase producto.

*Las dos clases HawainaPIzza y PicantePizza heredan de Pizza Builder la clase abstracta, cada una de estas clases van a tener un atributo que es el producto (clase)pizza

La clase cocina es el director que se va encargar de gestionar la creación del producto, se le asigna un método de pizzaBuilder que es el producto.

src/clases/cocina.java

22 lines (17 sloc) | 438 Bytes — Raw | Blame

```java
package clases;

// CLASE DIRECTOR.

public class Cocina {
    private PizzaBuilder builder;

    public void setPizzaBuilder(PizzaBuilder pizzaBuilder) {
        builder = pizzaBuilder;
    }

    public Pizza getPizza() {
        return builder.getPizza();
    }

    public void construirPizza() {
       builder.crearNuevaPizza();
       builder.buildMasa();
       builder.buildSalsa();
       builder.buildRelleno();
    }
}
```

src/clases/Hawainapizza.java

20 lines (16 sloc) | 378 Bytes

Raw Blame

```java
package clases;

// CONCRETE BUILDER.

public class HawaiPizzaBuilder extends PizzaBuilder {
    @Override
    public void buildMasa() {
        pizza.setMasa("suave");
    }

    @Override
    public void buildSalsa() {
        pizza.setSalsa("dulce");
    }

    @Override
    public void buildRelleno() {
        pizza.setRelleno("chorizo+alcachofas");
    }
}
```

src/clases/Picantepizza.java

20 lines (16 sloc) | 385 Bytes

Raw Blame

```java
package clases;

// CONCRETE BUILDER.

public class PicantePizzaBuilder extends PizzaBuilder {
    @Override
    public void buildMasa() {
        pizza.setMasa("cocida");
    }

    @Override
    public void buildSalsa() {
        pizza.setSalsa("picante");
    }

    @Override
    public void buildRelleno() {
        pizza.setRelleno("pimienta+salchichón");
    }
}
```

src/clases/PizzaBuider.java

18 lines (14 sloc) | 364 Bytes — Raw | Blame

```java
package clases;

// ABSTRACT BUILDER.
public abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void crearNuevaPizza() {
        pizza = new Pizza();
    }

    public abstract void buildMasa();
    public abstract void buildSalsa();
    public abstract void buildRelleno();
}
```

Método

src/Principal/Principal.java

19 lines (13 sloc) | 452 Bytes — Raw | Blame

```java
package principal;

import clases.*;

// CLASE PRINCIPAL.

public class Principal {
    public static void main(String[] args) {
        Cocina cocina = new Cocina(); // Director

        PizzaBuilder hawaiPizzaBuilder = new HawaiPizzaBuilder();
        PizzaBuilder picantePizzaBuilder = new PicantePizzaBuilder();

        cocina.setPizzaBuilder(hawaiPizzaBuilder);

        cocina.construirPizza();
        Pizza pizza = cocina.getPizza();
    }
}
```

En la realidad y basados en estos parámetros las empresas constructores y diseñadoras de Vehículos adoptan este tipo de Patron Builder, ya que tenemos un Producto (Vehículo) al cual se le pueden asignar diferentes atributos, utilizando los diferentes objectos, cambiando el producto a la necesidad y el gusto que se requiera.

PATRON OBSEVER

***Alto grado de independencia entre el objecto observador y los observadores interesados.

*El objecto observador no requiere ningún tipo de información acerca de los observadores, puesto que la interacción se realiza de manera independiente atreves de la interfaz de los observadores.

*El observador no siempre se entera de los cambios de los observadores y mas aún cuando el numero de observadores es muy alto.

Ejemplo 2

***Tipo Objecto noticiero (observador)

*Los suscriptores en este caso (3) son los observadores

*La clase noticiero crea una instancia del objecto observado(noticiero), creando las tres instancias de los suscriptores.

*El noticiero en primera instancia cambia de estado y da la primera noticia e inmediatamente notifica a sus observadores (suscriptores) de la nueva noticia, este método se vuelve repetitivo para los cambios de estados y las notificaciones a los observadores.

**La claseNoticieroTest, notifica de los cambios a los suscriptores

PatronObserver/noticiero/observable.java

PatronObserver/noticiero/Noticiero.java

29 lines (23 sloc) | 597 Bytes

```java
import java.util.ArrayList;
import java.util.List;

public class Noticiero implements IObservable{
  private List<IObserver> observers = new ArrayList<>();
  private String nuevoEstado;

  public void agregarObserver(IObserver IOb){
    observers.add(IOb);
  }

  public void removerObserver(IObserver IOb){
    observers.remove(IOb);
  }

  public void notifica(){
    for (IObserver iObserver : observers) {
      iObserver.actualizar();
    }
  }

  public String getEstado() {
    return nuevoEstado;
  }

  public void setEstado(String nuevoEstado) {
    this.nuevoEstado = nuevoEstado;
  }
}
```

PatronObserver/noticiero/NoticieroTest.java

31 lines (26 sloc) | 1 KB

```java
public class NoticieroTest {
  public NoticieroTest() {
    Noticiero noticiero = new Noticiero();
    Suscriptor suscriptor1 = new Suscriptor(noticiero, "Monica");
    Suscriptor suscriptor2 = new Suscriptor(noticiero, "Karla");
    Suscriptor suscriptor3 = new Suscriptor(noticiero, "Raúl");
    noticiero.agregarObserver(suscriptor1);
    noticiero.agregarObserver(suscriptor2);
    noticiero.agregarObserver(suscriptor3);

    //********* flujo de prueba *********
    //Actualiza nuevo noticia (estado)
    noticiero.setEstado("Científicos descrubren agua en la luna");
    noticiero.notifica();

    //Actualiza nuevo noticia (estado)
    noticiero.setEstado("Precio del petróleo a la baja");
    noticiero.notifica();

    //Remueve al suscriptor: Karla
    noticiero.removerObserver(suscriptor2);

    //Actualiza nuevo noticia (estado)
    noticiero.setEstado("¡Prepárate para las proximas vacaciones!");
    noticiero.notifica();
  }

  public static void main(String[] args) {
    new NoticieroTest();
  }
}
```

PatronObserver/noticiero/Suscriptor.java

14 lines (12 sloc) | 425 Bytes

Raw Blame

```java
public class Suscriptor implements IObserver{
  private Noticiero noticiero;
  private String nombreSuscriptor;

  public Suscriptor(Noticiero noticiero, String nombreSuscriptor) {
    this.noticiero = noticiero;
    this.nombreSuscriptor = nombreSuscriptor;
  }

  public void actualizar() {
    System.out.println( "Hey " + nombreSuscriptor +
                        ", nueva noticia: " + noticiero.getEstado());
  }
}
```

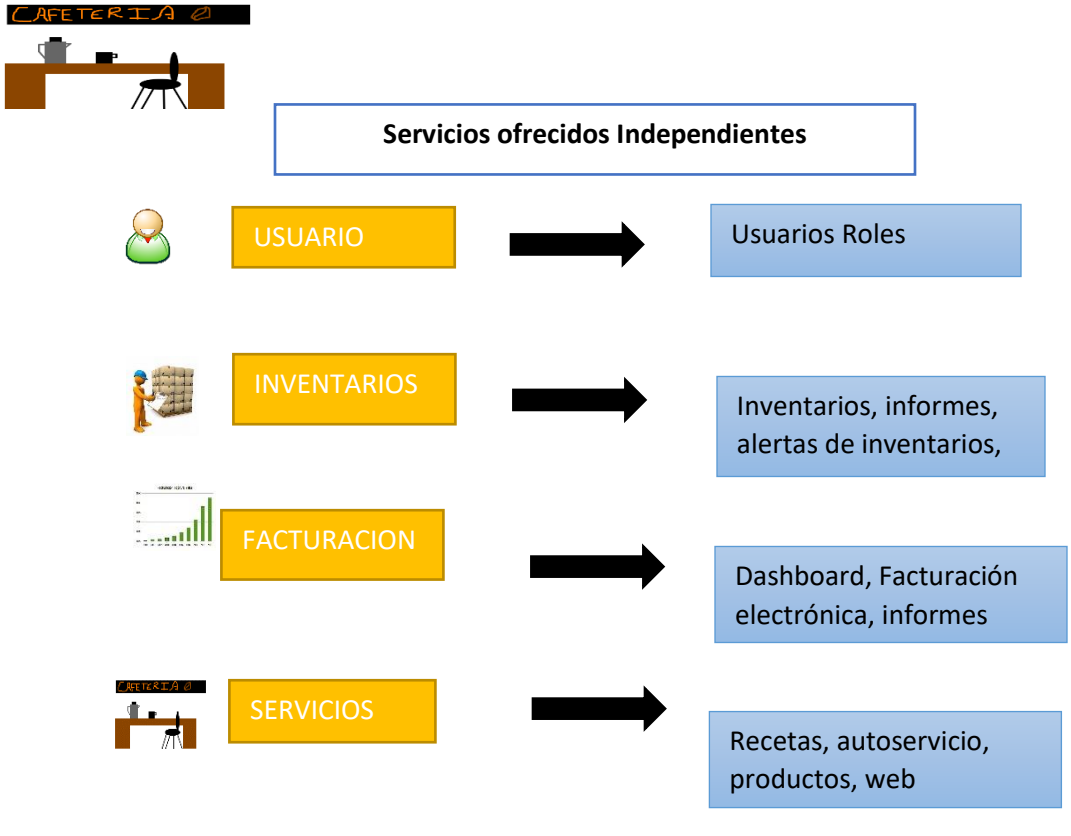
En la realidad y basados en estos parámetros las empresas y compañías que tiene a su cargo la notificación de sus productos a los clientes son las que más utilizan el PatronObserver, tales como lo indica el ejemplo, noticieros, suscripciones a revistas e incluso plataformas streaming utilizan esta clase de Patrón de diseño.

Un documento en pdf con el análisis de cómo se estructuraría toda la arquitectura por medio de aplicación monolítica y microservicios para un sistema de información de una cafetería.

Microservicio

Son un enfoque Arquitectónico para el desarrollo del software compuesto por pequeños servicios independientes que se comunican atreves de la API.
Tiene como objetivo aislar los distintos componentes de una aplicación con el fin de que cada uno sea una aplicación por si misma.

Para el desarrollo de un sistema de Microservicios para una cafetería se puede ofrecer varios servicios como:
* Dashboard con Información en tiempo real de ventas, inventario, etc.
* Descargue automático de inventario por receta.
* Facturación electrónica.
* Manejo de múltiples negocios con una sola cuenta.
* Informes por día, mes, año, productos, meseros, y más.
* Crea los roles y usuarios que necesites.
*Alertas por falta de inventario.
* Funciona en cualquier tipo de dispositivo (celular, tablet y computadora).
* Proceso del pedido completo desde la mesa a la caja y a la cocina, ahorrando tiempo y mejorando el servicio
* No requiere instalación.



Monolítico.

Tienen como características el uso de la base de un código único para sus servicios y funcionalidades, el sistema no se puede trabajar en varios ambientes al mismo tiempo de carga de los requerimientos, al actualizarse puesto que requiere que se aplica todo en conjunto.

Para el desarrollo de un sistema de un servicio Monolítico para una cafetería se puede ofrecer servicios como:
*Facturación impresión
*Flujos de caja
*Base de datos de inventario
*Roles y usuarios
*Requiere instalación
*Menú recetas.