Builder Pattern

Introducción y nombre

Builder. Creacional.

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.

Intención

Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

También conocido como

Patrón constructor o virtual builder.

Motivación

Los objetos que dependen de un algoritmo tendrán que cambiar cuando el algoritmo cambia. Por lo tanto, los algoritmos que estén expuestos a dicho cambio deberían ser separados, permitiendo de esta manera reutilizar algoritmos para crear diferentes representaciones. En otras palabras, permite a un cliente construir un objeto complejo especificando sólo su tipo y contenido, ocultándole todos los detalles de la construcción del objeto.

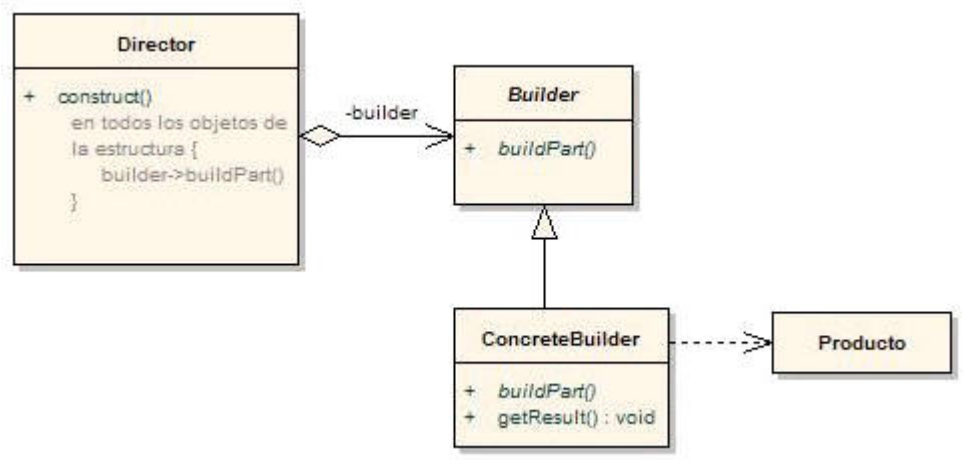
Solución

Se debe utilizar este patrón cuando sea necesario:

Independizar el algoritmo de creación de un objeto complejo de las partes que constituyen el objeto y cómo se ensamblan entre ellas.

Que el proceso de construcción permita distintas representaciones para el objeto construido, de manera dinámica.

Diagrama UML



Participantes

Producto: representa el objeto complejo a construir.

Builder: especifica una interface abstracta para la creación de las partes del Producto. Declara las operaciones necesarias para crear las partes de un objeto concreto.

ConcreteBuilder: implementa Builder y ensambla las partes que constituyen el objeto complejo.

Director: construye un objeto usando la interfaz Builder. Sólo debería ser necesario especificar su tipo y así poder reutilizar el mismo proceso para distintos tipos.

Colaboraciones

El Cliente crea el objeto Director y lo configura con el objeto Builder deseado.

El Director notifica al constructor cuándo una parte del Producto se debe construir.

El Builder maneja los requerimientos desde el Director y agrega partes al producto.

El Cliente recupera el Producto desde el constructor.

Consecuencias

Permite variar la representación interna de un producto.

El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.

Si se cambia la representación interna basta con crear otro Builder que respete la interfaz.

Separa el código de construcción del de representación.

Las clases que definen la representación interna del producto no aparecen en la interfaz del Builder.

Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto.

Distintos Directores pueden usar un mismo ConcreteBuilder.

Da mayor control en el proceso de construcción.

Permite que el Director controle la construcción de un producto paso a paso.

Sólo cuando el producto está acabado lo recupera el director del builder.

Implementación

Generalmente un Builder abstracto define las operaciones para construir cada componente que el Director podría solicitar.

El ConcreteBuilder implementa estas operaciones y le otorga la inteligencia necesaria para su creación.

Para utilizarlo el Director recibe un ContreBuilder.

Código de muestra

Realizaremos un ejemplo de un auto, el cual consta de diferentes partes para poder construirse.

[code]

```
public class Motor {
   private Integer numero;
   private String potencia;

   public Motor() {
   }

   public Integer getNumero() {
      return numero;
   }
```

```
    public void setNumero(Integer numero) {
        this.numero = numero;
    }

    public String getPotencia() {
        return potencia;
    }

    public void setPotencia(String potencia) {
        this.potencia = potencia;
    }
}

public class Auto {

    private int cantidadDePuertas;
    private String modelo;
    private String marca;
    private Motor motor;

    public int getCantidadDePuertas() {
        return cantidadDePuertas;
    }

    public void setCantidadDePuertas(int cantidadDePuertas) {
        this.cantidadDePuertas = cantidadDePuertas;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public Motor getMotor() {
        return motor;
    }

    public void setMotor(Motor motor) {
        this.motor = motor;
    }
}
```

Utilizaremos la clase AutoBuilder para que sirve para base de construcción de los distintos tipos de Autos:

```
public abstract class AutoBuilder {
    protected Auto auto = new Auto();

    public Auto getAuto() {
        return auto;
    }

    public void crearAuto() {
        auto = new Auto();
    }

    public abstract void buildMotor();
```

```
    public abstract void buildModelo();

    public abstract void buildMarca();

    public abstract void buildPuertas();
}
```

Realizaremos dos builders concretos que son: FordBuilder y FiatBuilder. Cada Builder tiene el conocimiento necesario para saber como se construye su auto.

```
public class FiatBuilder extends AutoBuilder {

    @Override
    public void buildMarca() {
        auto.setMarca("Fiat");
    }

    @Override
    public void buildModelo() {
        auto.setModelo("Palio");
    }

    @Override
    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(232323);
        motor.setPotencia("23 HP");
        auto.setMotor(motor);
    }

    @Override
    public void buildPuertas() {
        auto.setCantidadDePuertas(2);
    }
}
```

A modo de simplificar el aprendizaje, estos builders construyen objetos relativamente sencillos. Se debe tener en cuenta que la complejidad para la construcción de los objetos suele ser mayor.

```
public class FordBuilder extends AutoBuilder {

    @Override
    public void buildMarca() {
        auto.setMarca("Ford");
    }

    @Override
    public void buildModelo() {
        auto.setModelo("Focus");
    }

    @Override
    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(21212);
        motor.setPotencia("20 HP");
        auto.setMotor(motor);
    }

    @Override
    public void buildPuertas() {
        auto.setCantidadDePuertas(4);
    }
}
```

Por último, realizaremos la clase Concesionario. Lo primero que debe hacerse con esta clase es enviarle el tipo de auto que se busca construir (Ford, Fiat, etc). Luego, al llamar al método constructAuto(), la construcción se

realizará de manera automática.

```
public class Concesionaria {

    private AutoBuilder autoBuilder;

    public void construirAuto() {
        autoBuilder.buildMarca();
        autoBuilder.buildModelo();
        autoBuilder.buildMotor();
        autoBuilder.buildPuertas();
    }

    public void setAutoBuilder(AutoBuilder ab) {
        autoBuilder = ab;
    }

    public Auto getAuto() {
        return autoBuilder.getAuto();
    }
}
```

La invocación desde un cliente sería:

```
public class Main {
    public static void main(String[] args) {
        Concesionaria concesionaria = new Concesionaria();
        concesionaria.setAutoBuilder(new FordBuilder());
        concesionaria.construirAuto();
        Auto auto = concesionaria.getAuto();
        System.out.println(auto.getMarca());
    }
}
```

[/code]

Cuándo utilizarlo

Esta patrón debe utilizarse cuando el algoritmo para crear un objeto suele ser complejo e implica la interacción de otras partes independientes y una coreografía entre ellas para formar el ensamblaje. Por ejemplo: la construcción de un objeto Computadora, se compondrá de otros muchos objetos, como puede ser un objeto PlacaDeSonido, Procesador, PlacaDeVideo, Gabinete, Monitor, etc.

Patrones relacionados

Con el patrón Abstract Factory también se pueden construir objetos complejos, pero el objetivo del patrón Builder es construir paso a paso, en cambio, el énfasis del Abstract Factory es tratar familias de objetos.

El objeto construido con el patrón Builder suele ser un Composite.

El patrón Factory Method se puede utilizar el Builder para decidir qué clase concreta instanciar para construir el tipo de objeto deseado.

El patrón Visitor permite la creación de un objeto complejo, en vez de paso a paso, dando todo de golpe como objeto visitante.