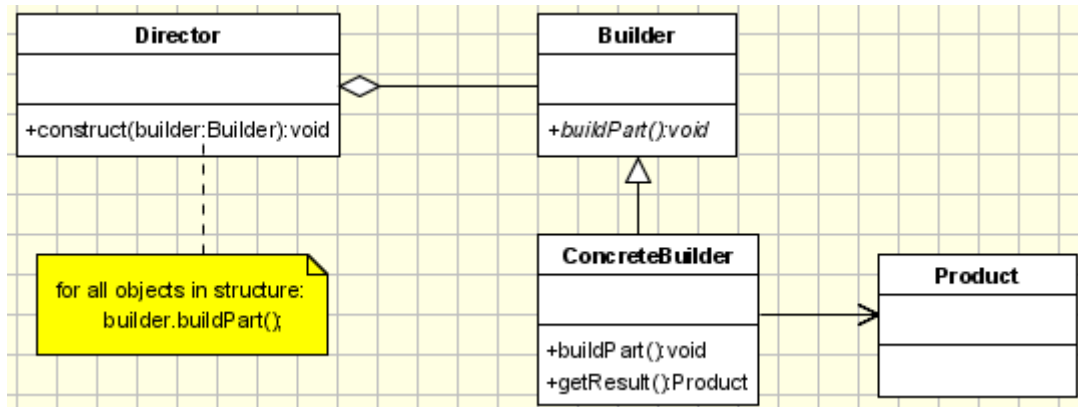


Builder

Diagrama de clases e interfaces



Intención

Este patrón se utiliza para construir paso a paso objetos complejos y en el paso final devolver el objeto creado. Un beneficio adicional es que también permite obtener diferentes representaciones de un mismo objeto y utilizando un único proceso de construcción. No obstante, para que esto sea posible, el proceso de construcción de un objeto debe ser genérico.

Motivación

El propósito de este patrón es simplificar la creación de objetos complejos, que pueden tener diferente representación en cada instanciación. Para ello define una clase, la clase **Builder**, cuyo objetivo es construir instancias del objeto complejo. A primera vista se podría confundir su utilidad con la de alguno de la familia **Factory**. Ha de quedar claro que **Builder** pone el énfasis en la construcción por fases del objeto complejo, mientras que los **Factory** no tienen en cuenta esta noticia, simplemente construyen el objeto en una única fase.

Para entender qué motiva la existencia del patrón veamos el siguiente caso de estudio:

Un PIM (Personal Information Manager) es un tipo de software que funciona como un organizador personal. El propósito de esta herramienta es facilitar el registro, el seguimiento y la gestión de cierta "información personal". Por ejemplo, una aplicación de este tipo podría contemplar las siguientes características:

- Notas personales
- Lista de contactos
- Lista de tareas
- Citas
- Fechas significativas: aniversarios, conmemoraciones, etc.
- Recordatorios

Supongamos que tenemos que construir una pequeña aplicación que permita a sus usuarios gestionar las diferentes citas a eventos sociales a las que tiene que acudir (citas en un contexto amplio: reuniones, encuentros, ferias, etc). Para ello necesitaremos una clase que represente la información asociada a un evento, que permita registrarlo y hacer un seguimiento del mismo. A esta clase podemos llamarla Cita y podría estar compuesta por la siguiente información:

- Fecha de inicio y de fin
- Una descripción de la cita
- La localización de la cita
- Asistentes a la cita

Lógicamente, esta información la introduce el propio usuario cuando crea la cita, por lo que tendremos que definir un constructor que permita inicializar los objetos de Cita.

Pero, ¿qué información se necesita exactamente para crear una cita? Es una buena pregunta, dado que según el tipo de cita se necesitarán unos datos u otros. Algunas citas podrían requerir una lista de asistentes, otras podrían necesitar una fecha de inicio y otra de fin (por ejemplo, la feria anual de tecnología móvil), mientras que otras podrían tener sólo una fecha de inicio (visita al museo Guggenheim). Una vez

consideradas estas opciones, queda claro que la tarea de construir una objeto Cita no es algo trivial.

Hay dos posibilidades para la creación de objetos, ninguna de ellas muy atractiva.

- Crear constructores para cada tipo de cita que desea crear
- Escribir un constructor enorme que contemple todos los casos

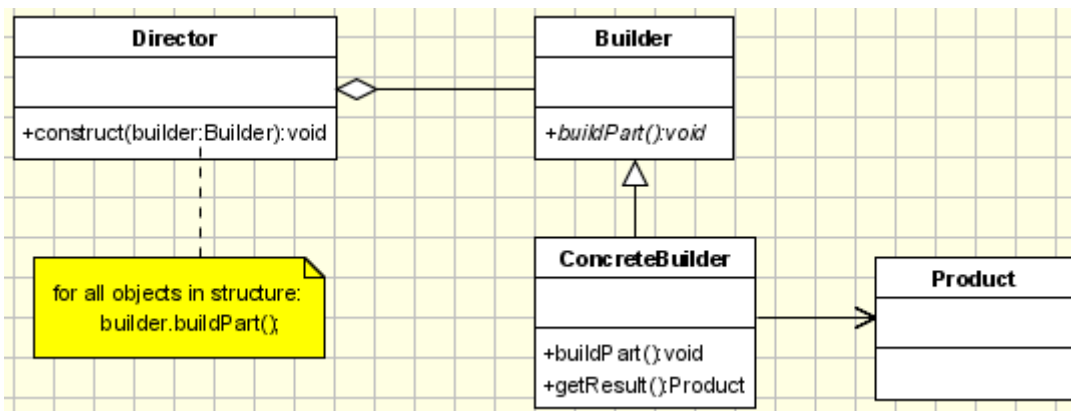
Cada método tiene sus desventajas. La opción de utilizar varios constructores hace que resulte complejo para las clases cliente invocar a los constructores, pues tienen que recordar cuántos hay, cuál es su cometido y la sintaxis de cada uno. La opción de utilizar un único constructor hace que el código de éste se vuelva más complejo, difícil de entender y propenso a errores. Y lo que es peor, ambos enfoques potencialmente pueden causar problemas si más adelante se necesita crear alguna subclase de Cita.

Lo que propone el patrón Builder es resolver esta situación de una manera elegante y eficiente. La propuesta es delegar la responsabilidad de la creación de citas en una clase especial que denominaríamos CitaBuilder, con lo que el código de la propia clase Cita quedaría simplificado en gran medida. CitaBuilder contendrá los métodos que crean cada parte de un Cita. Con esto, se consigue que las clases cliente puedan pasar al Builder sólo los argumentos de una nueva cita que sean relevantes para un tipo de cita en particular. Además, CitaBuilder puede verificar que la información que recibe para crear la cita es válida, reforzando así el cumplimiento de las reglas de negocio. Si en un futuro se necesita subclasificar Cita, podemos tanto crear un nuevo CitaBuilder como subclasificar el existente.

En una sección posterior veremos el código fuente de este caso.

Implementación

El siguiente diagrama muestra los participantes en el patrón:



Descripción de los participantes:

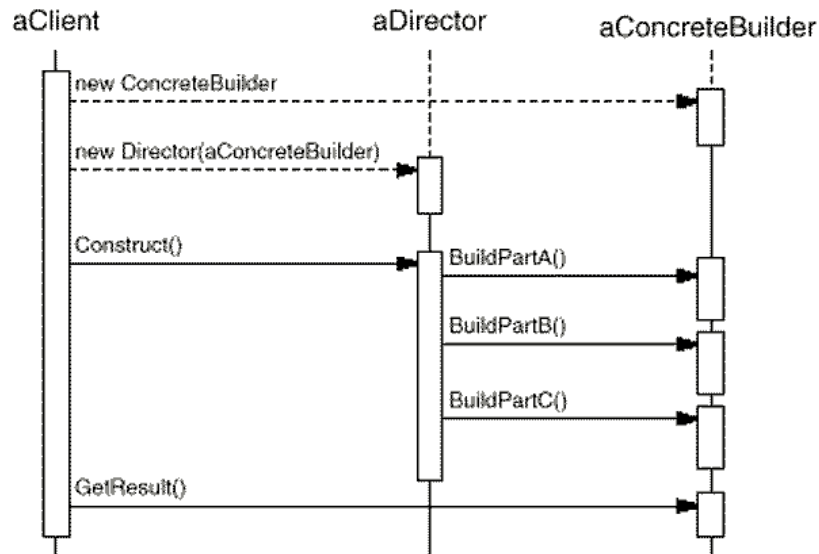
Director: Gestiona la construcción del objeto complejo utilizando para ello la interfaz de Builder. Debe conocer el proceso genérico de construcción para poder dirigir la elaboración del producto, es decir, a qué métodos `buildPart()` de Builder llamar y en qué orden.

Builder: Es la clase responsable de construir los sucesivos componentes (o partes) del objeto complejo (Product) a medida que se le solicita desde Director. Builder especifica una interfaz pública abstracta y tiene una subclase por cada forma de representación de Product.

ConcreteBuilder: Clases que implementan a Builder. Construyen diferentes representaciones de Product, por lo que cada ConcreteBuilder proporciona una implementación de `buildPart()` diferente, según convenga para la creación de una representación de Product en particular. Su método `getResult()` permite recuperar el objeto complejo.

Product: Representa el objeto complejo que ha sido construido.

Funcionamiento del patrón



Una clase cliente quiere obtener una determinada representación de un producto, por lo que inicia un objeto Builder y un objeto Director (conteniendo al Builder creado). El objeto Builder puede ser cualquiera de la jerarquía de Builder, lo importante es que sea capaz de llevar a cabo la construcción del producto según la representación deseada. A continuación, la clase cliente invoca el método `construct()` de Director, cuyo valor de retorno será el Product deseado.

El método `construct()` de Director invoca a los métodos `buildPart()` del Builder que tiene asociado. La ejecución secuencial de estos métodos va conformando el objeto complejo, por lo que una vez ejecutado el último, Director (o Client, esto es optativo) invoca a `getResult()` para obtener el objeto construido.

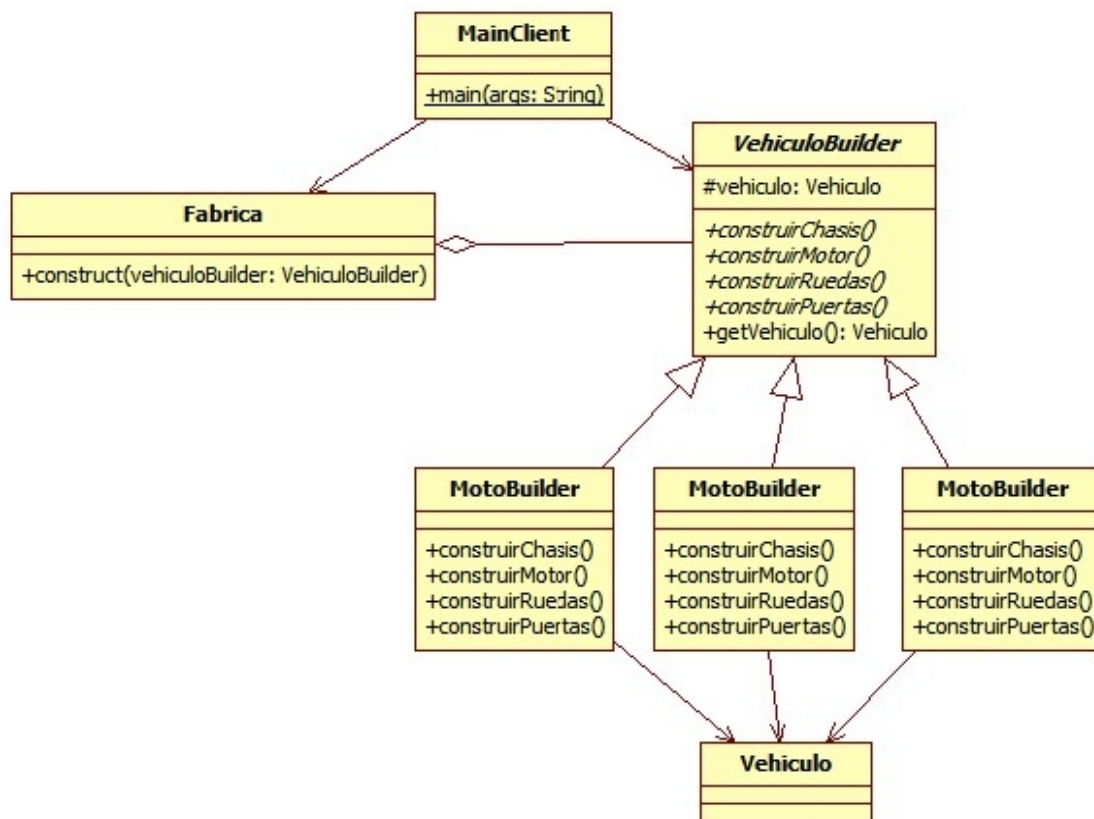
La clase Builder crea en su interior un objeto Product vacío y mediante la ejecución de sus métodos `buildPart()` (ejecutados por el Director) va construyendo el Product, revisando la lógica necesaria para que el objeto se cree adecuadamente o no se cree.

Aplicabilidad y Ejemplos

Veamos algunos ejemplos en los que sería apropiado el uso de este patrón.

Ejemplo 1 - Fabricante de vehículos

Tomemos el caso de un fabricante de vehículos. A partir de un conjunto de piezas puede construir un coche, un camión o una moto. En este caso, el Builder es la clase VehiculoBuilder. VehiculoBuilder especifica la interfaz para la construcción de cualquiera de los tres tipos de vehículos. Las clases ConcreteBuilder son los Builders asociados a cada uno de los objetos que se fabricarán. El Product es, por supuesto, el vehículo que se está construyendo. La clase Director es la clase Fabrica.



Vamos el código.

Vehiculo.java

Este es el objeto complejo que alguno de los builders concretos debe construir, devolviendo una de las representaciones: camión, coche o moto. Notad que uno de los

atributos contiene un HashMap para contener las diferentes piezas que conforman cada tipo de vehiculo. Los builders concretos son quienes añadirán estas partes.

```
package creacionales.builder.vehiculos;

import java.util.HashMap;

// Product
public class Vehiculo {
    private String tipo;

    public Vehiculo(String tipo) {
        this.tipo = tipo;
    }

    private HashMap<String, String> partes =
        new HashMap<String, String>();

    public void addParte(String clave, String parte) {
        partes.put(clave, parte);
    }

    public String getParte(String clave) {
        return partes.get(clave);
    }

    @Override
    public String toString() {
        return "\n-----" +
            "Tipo de vehiculo: " + tipo + "\n" +
            " Chasis : " + getParte("chasis") + "\n" +
            " Motor : " + getParte("motor") + "\n" +
            " Num. ruedas: " + getParte("ruedas") + "\n" +
            " Num. puertas : " + getParte("puertas");
    }
}
```

VehiculoBuilder.java

Builder abstracto; define la interfaz de operaciones que permiten construir vehículos y recuperar el vehículo construido. Estas operaciones serán invocadas por el Director. Notad que declara un vehículo en modo protegido, por lo que sus subclasses tendrán acceso directo.

```
package creacionales.builder.vehiculos;

// Builder (abstracto)
public abstract class VehiculoBuilder {

    protected Vehiculo vehiculo;

    public abstract void construirChasis();
    public abstract void construirMotor();
}
```

```

        public abstract void construirRuedas();
        public abstract void construirPuertas();

        public Vehiculo getVehiculo() {
            return vehiculo;
        }
    }
}

```

MotoBuilder.java

Builder concreto. Proporciona su propia interfaz para la construcción de vehículos tipo moto.

```

package creacionales.builder.vehiculos;

// ConcreteBuilder1
public class MotoBuilder extends VehiculoBuilder {

    @Override
    public void construirChasis() {
        vehiculo = new Vehiculo("Moto");
        vehiculo.addParte("chasis", "chasis moto");
    }

    @Override
    public void construirMotor() {
        vehiculo.addParte("motor", "500 cc");
    }

    @Override
    public void construirRuedas() {
        vehiculo.addParte("ruedas", "2");
    }

    @Override
    public void construirPuertas() {
        vehiculo.addParte("puertas", "0");
    }
}

```

CocheBuilder2.java

Builder concreto. Proporciona su propia interfaz para la construcción de vehículos tipo coche.

```

package creacionales.builder.vehiculos;

// ConcreteBuilder2
public class CocheBuilder extends VehiculoBuilder {

    @Override
    public void construirChasis() {
        vehiculo = new Vehiculo("Coche");
    }
}

```



```

        vehiculo.addParte("chasis", "chasis coche");
    }

    @Override
    public void construirMotor() {
        vehiculo.addParte("motor", "2500 cc");
    }

    @Override
    public void construirRuedas() {
        vehiculo.addParte("ruedas", "4");
    }

    @Override
    public void construirPuertas() {
        vehiculo.addParte("puertas", "4");
    }
}

```

CamionBuilder.java

Builder concreto. Proporciona su propia interfaz para la construcción de vehículos tipo camión.

```

package creacionales.builder.vehiculos;

// ConcreteBuilder3
public class CamionBuilder extends VehiculoBuilder {

    @Override
    public void construirChasis() {
        vehiculo = new Vehiculo("Camion");
        vehiculo.addParte("chasis", "chasis camion");
    }

    @Override
    public void construirMotor() {
        vehiculo.addParte("motor", "10000 cc");
    }

    @Override
    public void construirRuedas() {
        vehiculo.addParte("ruedas", "12");
    }

    @Override
    public void construirPuertas() {
        vehiculo.addParte("puertas", "3");
    }
}

```

Fabrica.java

Director del patrón. Recibe un Builder y lo emplea para ejecutar paso a paso el algoritmo general. A pesar de que recibe una referencia de Builder, esta referencia está apuntando a un objeto cuyo tipo es un Builder concreto (CamionBuilder, CocheBuilder o MotoBuilder), por tanto, lo que se va a construir siempre es algo concreto.

```
package creacionales.builder.vehiculos;

// Director
public class Fabrica {

    // El Director necesita saber los pasos generales
    // para la construcción de un vehiculo
    public void construct(VehiculoBuilder vehiculoBuilder) {
        vehiculoBuilder.construirChasis();
        vehiculoBuilder.construirMotor();
        vehiculoBuilder.construirRuedas();
        vehiculoBuilder.construirPuertas();
    }
}
```

MainClient.java

Se crea la fábrica (Director), se crean los builders concretos que se consideren y a continuación se invoca a construct() de Director pasándole uno de los Builder instanciados como argumento. Esto devuelve un producto complejo totalmente acabado

```
package creacionales.builder.vehiculos;

// Client
public class MainClient {
    public static void main(String args[]) {
        Fabrica fabrica = new Fabrica(); // Crear la fabrica

        // Crear tres builder concretos
        VehiculoBuilder vb1 = new CamionBuilder();
        VehiculoBuilder vb2 = new CocheBuilder();
        VehiculoBuilder vb3 = new MotoBuilder();

        // Construirlos y mostrarlos
        fabrica.construct(vb1);
        Vehiculo v1 = vb1.getVehiculo();
        System.out.println(v1);

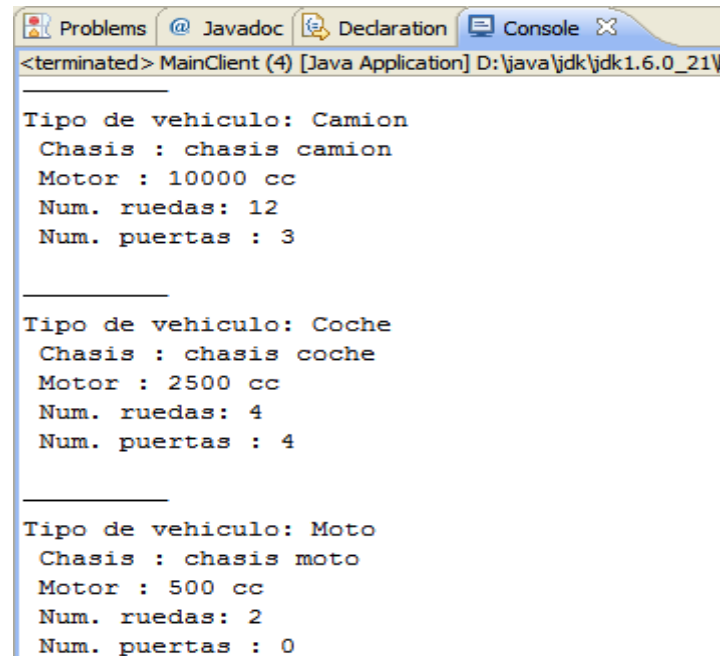
        fabrica.construct(vb2);
        Vehiculo v2 = vb2.getVehiculo();
        System.out.println(v2);
    }
}
```

```

        fabrica.construct(vb3);
        Vehiculo v3 = vb3.getVehiculo();
        System.out.println(v3);
    }
}

```

Salida:



```

<terminated> MainClient (4) [Java Application] D:\java\jdk\jdk1.6.0_21\

Tipo de vehiculo: Camion
Chasis : chasis camion
Motor : 10000 cc
Num. ruedas: 12
Num. puertas : 3

Tipo de vehiculo: Coche
Chasis : chasis coche
Motor : 2500 cc
Num. ruedas: 4
Num. puertas : 4

Tipo de vehiculo: Moto
Chasis : chasis moto
Motor : 500 cc
Num. ruedas: 2
Num. puertas : 0

```

Ejemplo 2 – Gestión de citas del sistema PIM

Leer la sección *Motivación* de este documento para entender el funcionamiento de un sistema PIM y los objetivos de la pequeña aplicaciones que se requiere.

Lo que aporta este ejemplo respecto al anterior de los vehículos, es que demuestra la facilidad con la que este patrón permite ampliar o sustituir la funcionalidad de un builder. En concreto veremos que si queremos crear una cita de tipo “encuentro”, como sería el caso de una feria de tres días de duración, debe ser necesario proporcionar tanto la fecha de inicio como la de finalización. Para ello se define *EncuentroBuilder*, que aprovecha toda la funcionalidad de *CitaBuilder* pero comprueba que exista obligatoriamente una fecha de finalización.

Veamos primero un resumen de cada clase según su papel en el patrón.

CitaBuilder, EncuentroBuilder: Builders concretos. *CitaBuilder* es la superclase de *EncuentroBuilder*, aunque *CitaBuilder* es aquí una clase concreta.

Planificador: Director.

Cita: Product.

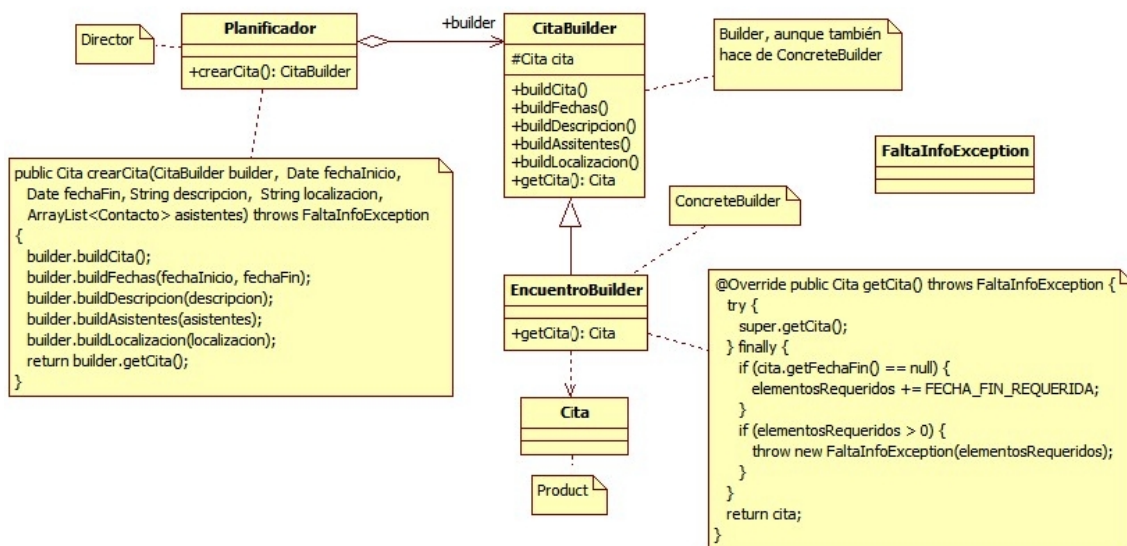
Contacto: JavaBean necesitado por Cita para representar a la persona o personas que asistirá/n a la cita.

FaltaInfoException: Excepción que se produce cuando alguno de los builders comprueba que falta información para crear correctamente la cita.

MainClientCita, MainClientEncuentro: Clase Client. La primera muestra el uso del patrón para crear una cita. La segunda intenta crear una cita de tipo “encuentro”, pero proporciona un valor nulo para la fecha de finalización, por lo que salta la excepción `FaltaInfoException` y el objeto no se crea. Seguidamente, corrige el problema y el objeto se crea correctamente.

Utiles: Clase de soporte con métodos estáticos para las clases cliente.

El siguiente diagrama de clases muestra la personalización del patrón para el caso particular de la aplicación PIM (se han omitido las clases cliente).



Veamos el código:

Cita.java

```
package creacionales.builder.citas.beans;
```

```

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

public class Cita {
    private Date fechaInicio;
    private Date fechaFin;
    private String descripcion;
    private ArrayList<Contacto> asistentes = new
ArrayList<Contacto>();
    private String localizacion;
    public static final String EOL_STRING = System
        .getProperty("line.separator");

    public Date getFechaInicio() { return fechaInicio; }
    public Date getFechaFin() { return fechaFin; }
    public String getDescripcion() { return descripcion; }
    public ArrayList<Contacto> getAsistentes() { return asistentes;
    }
    public String getLocalizacion() { return localizacion; }

    public String getFechaInicioBonita() {
        DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm");
        return sdf.format(fechaInicio);
    }

    public String getFechaFinBonita() {
        if (fechaFin != null) {
            DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy
HH:mm");
            return sdf.format(fechaFin);
        } else {
            return "N/A";
        }
    }

    public void setDescripcion(String nuevaDescripcion) {
        descripcion = nuevaDescripcion;
    }

    public void setLocalizacion(String nuevaLocalizacion) {
        localizacion = nuevaLocalizacion;
    }

    public void setFechaInicio(Date nuevaFechaInicio) {
        fechaInicio = nuevaFechaInicio;
    }

    public void setFechaFin(Date nuevaFechaFin) {
        fechaFin = nuevaFechaFin;
    }

    public void setAsistentes(ArrayList<Contacto> nuevoAsistentes) {
        if (nuevoAsistentes != null) {
            asistentes = nuevoAsistentes;
        }
    }
}

```

```

    public void addAsistentes(Contacto asistente) {
        if (!asistentes.contains(asistente)) {
            asistentes.add(asistente);
        }
    }

    public void removeAsistente(Contacto asistente) {
        asistentes.remove(asistente);
    }

    @Override
    public String toString() {
        return " Descripcion: " + descripcion + EOL_STRING
            + " Fecha inicio: " + getFechaInicioBonita()
            + EOL_STRING + " Fecha fin: " + getFechaFinBonita()
            + EOL_STRING + " Localizacion: " + localizacion
            + EOL_STRING + " Asistentes: " + asistentes;
    }
}

```

Contacto.java

```

package creacionales.builder.citas.beans;

public class Contacto {

    private static final long serialVersionUID = 1L;
    public static final String ESPACIO = " ";

    private String nombre;
    private String apellidos;
    private String categoria;
    private String organizacion;

    public Contacto(String nuevoNombre, String nuevoApellidos,
        String nuevaCategoria, String nuevaOrganizacion)
    {
        nombre = nuevoNombre;
        apellidos = nuevoApellidos;
        categoria = nuevaCategoria;
        organizacion = nuevaOrganizacion;
    }

    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos; }
    public String getCategoria() { return categoria; }
    public String getOrganizacion() { return organizacion; }

    public void setNombre(String nuevoNombre) {
        nombre = nuevoNombre;
    }

    public void setApellidos(String nuevoApellidos) {
        apellidos = nuevoApellidos;
    }
}

```

```

    public void setCategoria(String nuevaCategoria) {
        categoria = nuevaCategoria;
    }

    public void setOrganizacion(String nuevaOrganicacion) {
        organizacion = nuevaOrganicacion;
    }

    @Override
    public String toString() {
        return nombre + ESPACIO + apellidos +
            ESPACIO + " - Categoria: " + categoria +
            ESPACIO + " - Organizacion: " + organizacion + "\n";
    }
}

```

CitaBuilder.java

```

package creacionales.builder.citas;

import java.util.Date;
import java.util.ArrayList;

import creacionales.builder.citas.beans.Cita;
import creacionales.builder.citas.beans.Contacto;

public class CitaBuilder {

    public static final int FECHA_INI_REQUERIDA = 1;
    public static final int FECHA_FIN_REQUERIDA = 2;
    public static final int DESCRIPCION_REQUERIDA = 4;
    public static final int ASISTENTES_REQUERIDOS = 8;
    public static final int LOCALIZACIÓN_REQUERIDA = 16;

    protected Cita cita;
    protected int elementosRequeridos;

    public void buildCita() {
        cita = new Cita();
    }

    public void buildFechas(Date fechaInicio, Date fechaFin) {
        Date fechaActual = new Date();
        if ((fechaInicio != null) &&
            (fechaInicio.after(fechaActual))) {
            cita.setFechaInicio(fechaInicio);
        }
        if ((fechaFin != null) && (fechaFin.after(fechaInicio))) {
            cita.setFechaFin(fechaFin);
        }
    }

    public void buildDescripcion(String nuevaDescripcion) {
        cita.setDescripcion(nuevaDescripcion);
    }

    public void buildAsistentes(ArrayList<Contacto> asistentes) {

```

```

        if ((asistentes != null) && (!asistentes.isEmpty())) {
            cita.setAsistentes(asistentes);
        }
    }

    public void buildLocalizacion(String nuevaLocalizacion) {
        if (nuevaLocalizacion != null) {
            cita.setLocalizacion(nuevaLocalizacion);
        }
    }

    public Cita getCita() throws FaltaInfoException {
        elementosRequeridos = 0;

        if (cita.getFechaInicio() == null) {
            elementosRequeridos += FECHA_INI_REQUERIDA;
        }

        if (cita.getLocalizacion() == null) {
            elementosRequeridos += LOCALIZACION_REQUERIDA;
        }

        if (cita.getAsistentes().isEmpty()) {
            elementosRequeridos += ASISTENTES_REQUERIDOS;
        }

        if (elementosRequeridos > 0) {
            throw new FaltaInfoException(elementosRequeridos);
        }

        return cita;
    }

    public int getElementosRequeridos() {
        return elementosRequeridos;
    }
}

```

EncuentroBuilder.java

```

package creacionales.builder.citas;

import creacionales.builder.citas.beans.Cita;

public class EncuentroBuilder extends CitaBuilder {

    @Override
    public Cita getCita() throws FaltaInfoException {
        try {
            super.getCita();
        } finally {
            if (cita.getFechaFin() == null) {
                elementosRequeridos += FECHA_FIN_REQUERIDA;
            }

            if (elementosRequeridos > 0) {

```



```

        throw new
FaltaInfoException(elementosRequeridos);
    }
    }
    return cita;
}
}

```

Planificador.java

```

package creacionales.builder.citas;

import java.util.Date;
import java.util.ArrayList;

import creacionales.builder.citas.beans.Cita;
import creacionales.builder.citas.beans.Contacto;

public class Planificador {

    public Cita crearCita(CitaBuilder builder,      Date fechaInicio,
Date fechaFin,
        String descripcion,      String localizacion,
ArrayList<Contacto> asistentes)
        throws FaltaInfoException {

        builder.buildCita();
        builder.buildFechas(fechaInicio, fechaFin);
        builder.buildDescripcion(descripcion);
        builder.buildAsistentes(asistentes);
        builder.buildLocalizacion(localizacion);

        return builder.getCita();
    }
}

```

FaltaInfoException.java

```

package creacionales.builder.citas;

public class FaltaInfoException extends Exception {

    private static final long serialVersionUID = 1L;
    private static final String MENSAJE =
        "La cita no se puede crear porque falta informacion";

    public static final int FECHA_INI_REQUERIDA = 1;
    public static final int FECHA_FIN_REQUERIDA = 2;
    public static final int DESCRIPCIÓN_REQUERIDA = 4;
    public static final int ASISTENTES_REQUERIDOS = 8;
    public static final int LOCALIZACIÓN_REQUERIDA = 16;

    private int faltaInfo;

    public FaltaInfoException(int elementosRequeridos) {
        super(MENSAJE);
    }
}

```

```

        faltaInfo = elementosRequeridos;
    }

    public int getFaltaInfo() {
        return faltaInfo;
    }
}

```

MainClientCita.java

```

package creacionales.builder.citas.main;

import java.util.Date;

import creacionales.builder.citas.CitaBuilder;
import creacionales.builder.citas.FaltaInfoException;
import creacionales.builder.citas.Planificador;
import creacionales.builder.citas.beans.Cita;

// importacion estatica de los métodos de la clase Utiles
import static creacionales.builder.citas.main.Utiles.*;

public class MainClientCita {

    public static void main(String[] arguments) {
        Cita cita = null;

        print("Ejemplo del patron Builder\n");
        print("Aplicacion para crear objetos Citas del sistema
PIM.\n");

        print("Creando un planificador [clase Director]");
        Planificador planificador = new Planificador();

        System.out.println("Creando un CitaBuilder [clase Builder]\n");
        CitaBuilder citaBuilder = new CitaBuilder();

        try {
            print("Creando una nueva Cita mediante un
CitaBuilder");

            Date fechaCitaIni = crearFecha(2066, 9, 22, 12, 30);
            Date fechaCitaFin = null;
            String descripcion = "Presentacion Java 7";
            String localizacion = "Hotel New-Tech, Barcelona";

            // CONSTRUCCION DE LA CITA

            cita = planificador.crearCita(citaBuilder,
fechaCitaIni, fechaCitaFin,
                                descripcion, localizacion,
crearAsistentes(4));

            mostrarResultados(cita);
        } catch (FaltaInfoException exc) {
            printExcepciones(exc);
        }
    }
}

```

```

    }
}
}

```

MainClientEncuentro.java

```

package creacionales.builder.citas.main;

import java.util.Date;

import creacionales.builder.citas.CitaBuilder;
import creacionales.builder.citas.EncuentroBuilder;
import creacionales.builder.citas.FaltaInfoException;
import creacionales.builder.citas.Planificador;
import creacionales.builder.citas.beans.Cita;

//importacion estatica de los métodos de la clase Utiles
import static creacionales.builder.citas.main.Utiles.*;

public class MainClientEncuentro {

    public static void main(String[] arguments) {
        Cita cita = null;

        print("Ejemplo del patron Builder\n");
        print("Aplicacion para crear objetos Citas del sistema
PIM.\n");

        print("Creando un planificador [clase Director]");
        Planificador planificador = new Planificador();

        print("Creando un EncuentroBuilder");
        CitaBuilder encuentroBuilder = new EncuentroBuilder();

        try {
            print("Creando una nueva cita mediante un
EncuentroBuilder");
            print("(notad que se lanzara la excepcion porque un
encuentroBuilder");
            print("(requiere una fecha de finalizacion");

            Date fechaCitaIni = crearFecha(2066, 9, 22, 12, 30);
            Date fechaCitaFin = null; // POR ESTO SE LANZA LA
EXCEPCION

            String descripcion = "Presentacion Oracle 28g";
            String localizacion = "Hotel Castellana, Madrid";

            cita = planificador.crearCita(encuentroBuilder,
fechaCitaIni, fechaCitaFin,
                                descripcion, localizacion,
crearAsistentes(4));

            mostrarResultados(cita);
        } catch (FaltaInfoException exc) {
            printExcepciones(exc);
        }
    }
}

```

```

        print("Creando una nueva cita mediante un
EncuentroBuilder");
        print("(esta vez se proporciona una fecha de
finalizacion)");
        try {
            Date fechaCitaIni = crearFecha(2077, 4, 22, 12, 30);
            Date fechaCitaFin = crearFecha(2077, 4, 23, 18, 00);
            String descripcion = "Feria dispositivos moviles";
            String localizacion = "Salon Central, Girona";

            cita = planificador.crearCita(encuentroBuilder,
fechaCitaIni, fechaCitaFin,
                                descripcion, localizacion,
crearAsistentes(2));

            mostrarResultados(cita);
        } catch (FaltaInfoException exc) {
            printExcepciones(exc);
        }
    }
}

```

Utiles.java

```

package creacionales.builder.citas.main;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.Random;

import creacionales.builder.citas.FaltaInfoException;
import creacionales.builder.citas.beans.Cita;
import creacionales.builder.citas.beans.Contacto;

/**
 * Clase con metodos estaticos para dar soporte
 * a las dos clases cliente
 */
class Utiles {
    static void mostrarResultados(Cita cita) {
        print("Cita creada correctamente.");
        print("Informacion sobre la cita:");
        print(cita);
        print("\n");
    }

    static Date crearFecha(int anyo, int mes, int dia,
int hora, int minuto)
    {
        Calendar fechaCreator = Calendar.getInstance();
        fechaCreator.set(anyo, mes, dia, hora, minuto);
        return fechaCreator.getTime();
    }

    static ArrayList<Contacto> crearAsistentes(int totalAsistentes)

```

```

{
    ArrayList<Contacto> grupo = new ArrayList<Contacto>();

    for (int i = 0; i < totalAsistentes; i++) {
        grupo.add(new Contacto(getNombre(), getApellidos(),
                                "Empleado", "EmpresaX, S.A.));
    }
    return grupo;
}

static String getNombre() {
    String nombre = "";
    int num = new Random().nextInt(5);
    switch (num) {
        case 0: nombre = "Luisa";
            break;
        case 1: nombre = "Jaime";
            break;
        case 2: nombre = "Eduardo";
            break;
        case 3: nombre = "Jose Manuel";
            break;
        case 4: nombre = "Martina";
            break;
        case 5: nombre = "Raul";
            break;
    }
    return nombre;
}

static String getApellidos() {
    String apellidos = "";
    int num = new Random().nextInt(5);
    switch (num) {
        case 0: apellidos = "Garcia Perez";
            break;
        case 1: apellidos = "Sanchez Lopez";
            break;
        case 2: apellidos = "Casals Pi";
            break;
        case 3: apellidos = "Anglada Ferras";
            break;
        case 4: apellidos = "Martinez Fernandez";
            break;
        case 5: apellidos = "Rodriguez Hernan";
            break;
    }
    return apellidos;
}

static void printExcepciones(FaltaInfoException exc) {
    int codigoEstado = exc.getFaltaInfo();
    final String sufijo = " para completar esta cita.";

    print("Imposible crear cita: se necesita informacion
adicional");
    if ((codigoEstado & FaltaInfoException.FECHA_INI_REQUERIDA)
> 0) {
        print(" Se requiere una fecha de inicio"+sufijo);
    }
}

```

```

    }
    if ((codigoEstado & FaltaInfoException.FECHA_FIN_REQUERIDA)
> 0) {
        print(" Se requiere una fecha de
finalizacion"+sufijo);
    }
    if ((codigoEstado &
FaltaInfoException.DESCRIPCION_REQUERIDA) > 0) {
        print(" Se requiere una descripcion"+sufijo);
    }
    if ((codigoEstado &
FaltaInfoException.ASISTENTES_REQUERIDOS) > 0) {
        print(" Se requiere al menos un asistente"+sufijo);
    }
    if ((codigoEstado &
FaltaInfoException.LOCALIZACION_REQUERIDA) > 0) {
        print(" Se requiere una localizacion"+sufijo);
    }
    print("\n");
}

static void print(Object obj) {
    System.out.println(obj);
}
}

```

Salida mediante MainClientCita:

```

<terminated> MainClientCita (4) [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (06/10/2011 08:27:52)
Ejemplo del patron Builder

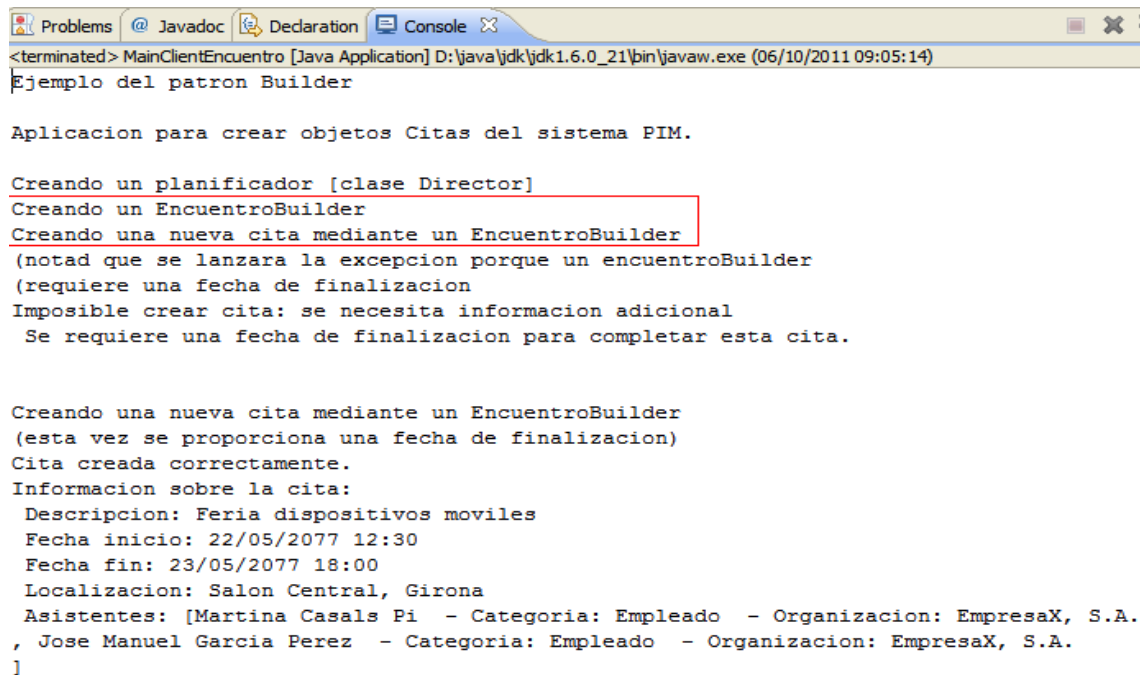
Aplicacion para crear objetos Citas del sistema PIM.

Creando un planificador [clase Director]
Creando un CitaBuilder [clase Builder]

Creando una nueva Cita mediante un CitaBuilder
Cita creada correctamente.
Informacion sobre la cita:
Descripcion: Presentacion Java 7
Fecha inicio: 22/10/2066 12:30
Fecha fin: N/A
Localizacion: Hotel New-Tech, Barcelona
Asistentes: [Jose Manuel Martinez Fernandez - Categoria: Empleado - Organizacion: EmpresaX, S.A.
, Martina Martinez Fernandez - Categoria: Empleado - Organizacion: EmpresaX, S.A.
, Luisa Anglada Ferras - Categoria: Empleado - Organizacion: EmpresaX, S.A.
, Luisa Casals Pi - Categoria: Empleado - Organizacion: EmpresaX, S.A.
]

```

Salida mediante MainClientEncuentro:



```
<terminated> MainClientEncuentro [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (06/10/2011 09:05:14)
Ejemplo del patron Builder

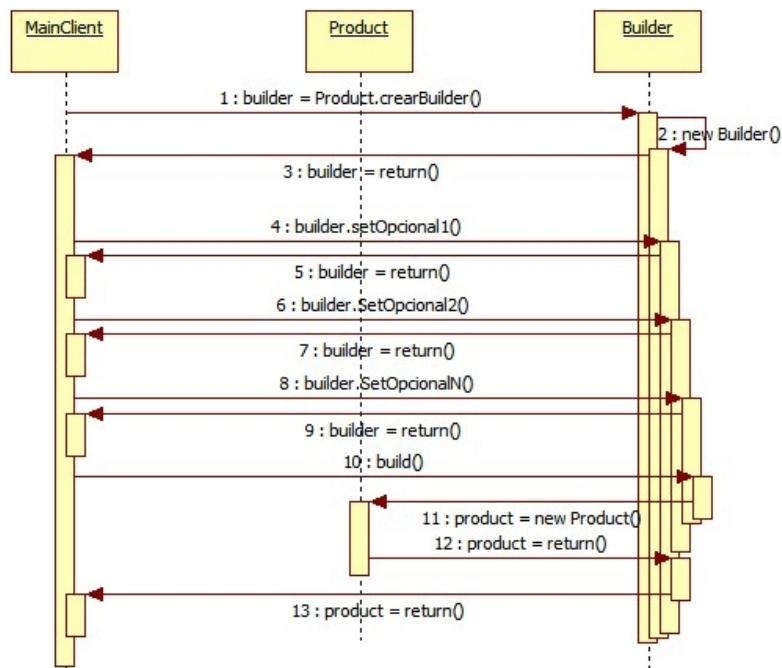
Aplicacion para crear objetos Citas del sistema PIM.

Creando un planificador [clase Director]
Creando un EncuentroBuilder
Creando una nueva cita mediante un EncuentroBuilder
(notad que se lanzara la excepcion porque un encuentroBuilder
requiere una fecha de finalizacion
Imposible crear cita: se necesita informacion adicional
Se requiere una fecha de finalizacion para completar esta cita.

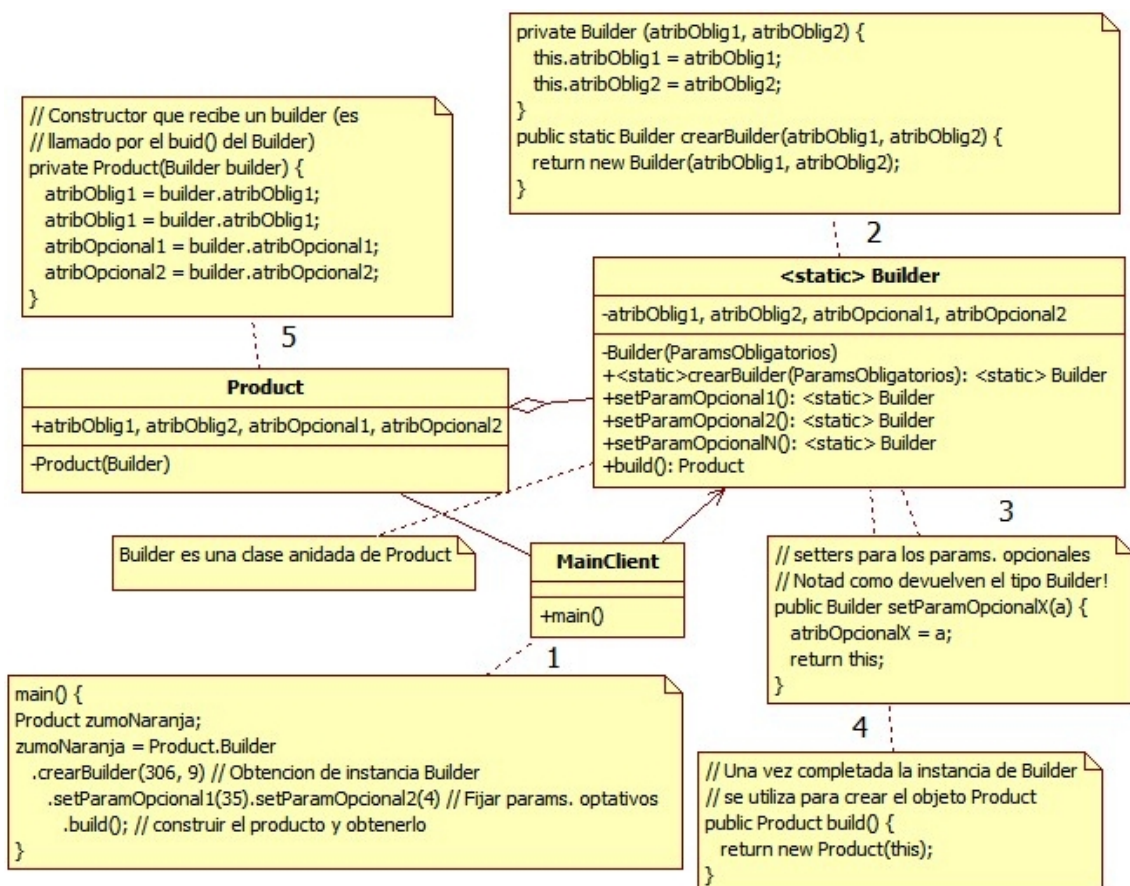
Creando una nueva cita mediante un EncuentroBuilder
(esta vez se proporciona una fecha de finalizacion)
Cita creada correctamente.
Informacion sobre la cita:
Descripcion: Feria dispositivos moviles
Fecha inicio: 22/05/2077 12:30
Fecha fin: 23/05/2077 18:00
Localizacion: Salon Central, Girona
Asistentes: [Martina Casals Pi - Categoria: Empleado - Organizacion: EmpresaX, S.A.
, Jose Manuel Garcia Perez - Categoria: Empleado - Organizacion: EmpresaX, S.A.
]
```

Ejemplo 3 – Variación del patrón Builder de GoF en el que el Buidar es una clase estática contenida dentro del objeto complejo.

Esta variante difiere bastante de las vistas, ya que aquí lo que tenemos es una clase (Builder) anidada dentro de otra clase (Product, el objeto complejo). La clase anidada, llamada generalmente Builder, es una clase estática que se utiliza para construir objetos de la clase exterior. Veamos el diagrama de secuencias:



La figura siguiente es el diagrama de clases dinámico de esta variante de Builder:



A tener en cuenta:

- La clase Product tiene unos atributos obligatorios y otros opcionales.
- La clase Builder tiene los mismos atributos que la clase Product.

El funcionamiento del diagrama de clases dinámico anterior es el siguiente:

1. Desde una clase cliente, en lugar de crear el objeto de la clase exterior directamente (Product), llamamos a un método estático de factoría (otra opción es llamar a un constructor de Builder) con todos los parámetros obligatorios. En la figura este método es crearBuilder().
2. El método crearBuilder() llama a su constructor privado y guarda los parámetros obligatorios en sus atributos correspondientes. Retorna al cliente el nuevo objeto Builder creado. En estos momentos ya tenemos un objeto Builder en la clase cliente.
3. De nuevo en la clase cliente, llamamos a los setters del objeto Builder para establecer todos los parámetros opcionales que nos interesen. Se retorna el objeto Builder a la clase cliente.
4. De nuevo en la clase cliente, ya está todo a punto para construir el Product, por lo que invocamos al método build() del objeto Builder. Este método llama al constructor privado de Product, pasándole una referencia a sí mismo, ya que el Product se creará a partir de los atributos del Builder.
5. De esta manera el Product se inicializa y ya no puede cambiar, esto es, se trata de un objeto inmutable. El Product es retornado a la clase cliente.

Veamos el código:

Product.java

```
package creacionales.builder.nutricion;

public class Product {

    private final int cantidad; // (mL) requerido
    private final int unidades; // (por envase) requerido
    private final int calorías; // (kcal) requerido
    private final int grasas; // (g) opcional
    private final int sodio; // (mg) opcional
    private final int potasio; // (mg) opcional
```

```

private final int carbohidratos; // (g) opcional

/*
 * Clase Builder: Se trata de una clase anidada. NOTA:
 * Cuando una clase interna NO es estática sólo se pueden
 * crear instancias de ella mediante una instancia de la
 * clase que la envuelve. Por este motivo Builder es
 * estática: porque se crea la instancia de Builder antes
 * que la de Product!
 */
public static class Builder {

    // parametros obligatorios
    private final int cantidad;
    private final int unidades;
    private final int calorías;

    // parametros opcionales-inicializados a su valor
    predeterminando
    private int grasas = 0;
    private int sodio = 0;
    private int potasio = 0;
    private int carbohidratos = 0;

    // Constructor privado. Esto es opcional, podría haber sido
    publico
    // y entonces no haría falta el metodo de factoria estatico
    crearBuilder
    // (Cuestion de gustos)
    private Builder (int cantidad, int unidades, int calorías)
    {
        this.cantidad = cantidad;
        this.unidades = unidades;
        this.calorías = calorías;
    }

    // Metodo de factoria estatico que crear la instancia del
    // Builder invocando al constructor privado. Inicializa
    // los atributos obligatorios
    public static Builder crearBuilder(int cantidad, int
    unidades, int calorías) {
        return new Builder(cantidad, unidades, calorías);
    }

    // setters para establecer los parametros opcionales.
    // Notad como devuelven el tipo Builder!
    public Builder grasas(int val) {
        grasas = val;
        return this;
    }

    public Builder sodio(int val) {
        sodio = val;
        return this;
    }

    public Builder potasio(int val) {
        potasio = val;
        return this;
    }
}

```

```

    }

    public Builder carbohidratos(int val) {
        carbohidratos = val;
        return this;
    }

    // Finalmente, el método Builder.build() es quien crea
    // la instancia de Product
    public Product build() {
        return new Product(this);
    }
}

/*
 * _____
 *
 * Final de clase Builder
 *
 * _____
 */

// Constructor privado que recibe un builder (es llamado
// por el método build() del Builder)
private Product(Builder builder) {
    cantidad = builder.cantidad;
    unidades = builder.unidades;
    calorias = builder.calorias;
    grasas = builder.grasas;
    sodio = builder.sodio;
    potasio = builder.potasio;
    carbohidratos = builder.carbohidratos;
}

@Override
public String toString() {
    return "[\nvalores obligatorios: cantidad=" + cantidad + ",
unidades=" + unidades
        + ", calorias=" + calorias + "\n" + "valores
optativos: grasas=" + grasas + ", sodio="
        + sodio + ", potasio=" + potasio + ",
carbohidratos="
        + carbohidratos + "\n]\n";
}
}

```

MainClient.java

```

package creacionales.builder.nutricion;

import creacionales.builder.nutricion.Product.Builder;

public class MainClient {

    public static void main(String[] args) {

        /*
         * Primer ejemplo -largo-

```

```

        *
        */

        // Obtenemos una instancia de Builder. Para conseguirla
        // tenemos que pasar los parametros obligatorios para
        // la creación del producto
        Builder builder = Product.Builder.crearBuilder(240, 8,
100);

        // Ahora establecemos los parametros optativos
        builder.sodio(35).grasas(27).carbohidratos(39);

        // Finalmente creamos el producto
        Product cola = builder.build();

        System.out.println("cola" + cola);

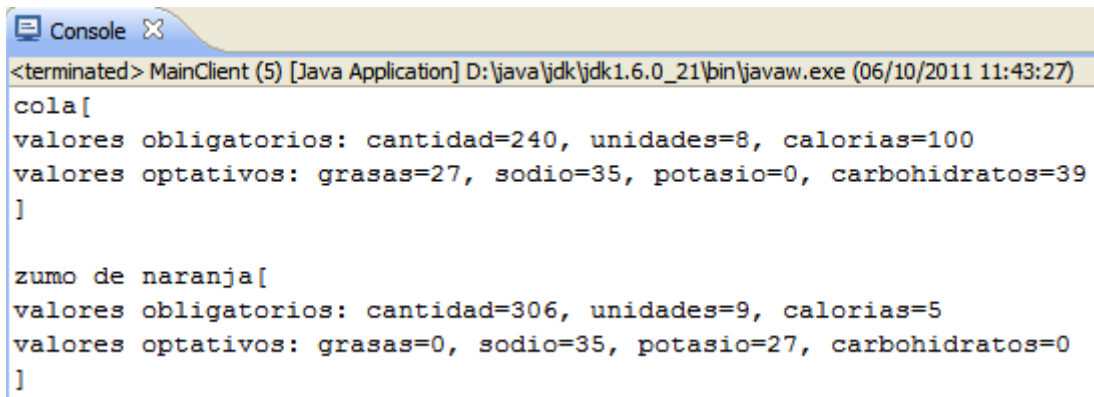
        /*
        * Segundo ejemplo -corto-
        */
        Product zumoNaranja = Product.Builder.crearBuilder(306, 9,
5)
            .sodio(35).potasio(27).build();

        System.out.println("zumo de naranja" + zumoNaranja);

    }
}

```

Salida:



```

<terminated> MainClient (5) [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (06/10/2011 11:43:27)
cola[
valores obligatorios: cantidad=240, unidades=8, calorías=100
valores optativos: grasas=27, sodio=35, potasio=0, carbohidratos=39
]

zumo de naranja[
valores obligatorios: cantidad=306, unidades=9, calorías=5
valores optativos: grasas=0, sodio=35, potasio=27, carbohidratos=0
]

```

Consideraciones de esta variante del patrón:

- Para invocar al método crearBuilder() de la clase Builder, al ser ésta interna, la llamada requiere del prefijo de la clase externa: Product.crearBuilder().
- Product es inmutable: sus atributos toman valor sólo una vez, en su constructor privado, a partir de los datos procedentes del objeto Builder.

- Los métodos setter de Builder devuelven un objeto Builder, por lo que las invocaciones de estos métodos se pueden encadenar. Esto es lo que simula el uso de parámetros opcionales con nombre, como lo hacen otros lenguajes como ADA y Python.
- Por otro lado, debemos ser cuidadosos con los valores que pueden tomar los parámetros que recibe el objeto Builder y que servirán para construir nuestro objeto final. Por ejemplo, supongamos que el componente Sodio sólo puede tomar valor entre 0 y 200 y se hace el siguiente intento de construcción de nuestro objeto:

```
Product batidoFresa=Product.crearBuilder(240, 8, 100).
    sodio(-35).potasio(27).build();
```

Entonces podemos controlar tal hecho en el método build():

```
public Product build() {
    if (this.sodio<0) {
        throw new IllegalStateException("El valor para el " +
            "sodio debe estar entre [0 y 200]: "+sodio);
    }
    return new Product(this);
}
```

Por lo que se lanzaría una excepción e impediríamos que se creara el Product de manera incorrecta.

- Aunque para mayor seguridad podríamos poner un control antes del método build(), esto es, en el propio método setter de la clase Builder:

```
public Builder sodio(int val) {
    if (val>=0 && val<=200) {
        sodio = val;
        return this;
    }
    throw new IllegalArgumentException("El valor para " +
        "el sodio debe estar entre [0 y 200]: "+val);
}
```

Como podemos deducir, esta variante del patrón Builder también es muy potente y flexible. Así, un único Builder puede ser capaz de construir múltiples objetos de la clase que lo envuelve, ya que los parámetros de su constructor se pueden ajustar para crear una variedad de ellos. También puede rellenar ciertos campos de manera automática, tal como números de serie incrementados para cada nuevo objeto, etc.

Problemas específicos e implementación

Builder vs. Abstract Factory

Abstract Factory es similar a Builder en que también puede crear objetos complejos. La diferencia principal es que Builder se centra en construir objetos complejos mediante una serie de fases, mientras que la motivación de Abstract Factory es la creación de familias de objetos, sean estos complejos o no. Builder devuelve el producto en su fase final, en cambio Abstract Factory lo retorna inmediatamente, en cuanto le es posible.

No se necesita una jerarquía de herencia ni interfaz común para los productos

En la práctica, los productos creados por los builders concretos tienen estructuras significativamente diferentes, por lo que no hay razón para derivar diferentes productos de una clase base. Esto también distingue al patrón Builder de la familia de patrones Factory, ya que éstos crean objetos derivados de un tipo base.

Patrones relacionados

Como Builder, Abstract Factory también puede construir objetos complejos. La gran diferencia es que Builder se centra en construir el objeto por etapas, mientras que Abstract Factory hace énfasis en construir productos de familias de objetos (sean estos productos simples o complejos). Builder retorna el producto una vez que ha alcanzado su última fase de construcción. En cambio, Abstract Factory lo devuelve en cuanto puede, intentando que esto sea lo antes posible.