INTRODUCCIÓN A LOS PATRONES DE DISEÑO

UN ENFOQUE PRÁCTICO

Oscar Javier Blancarte Iturralde

PRIMERA EDICIÓN

Introducción
a los
Patrones de Diseño

❝ Un código elegante no es aquel que tiene menos líneas, si no el que saca mayor provecho de ellas ❞
— Oscar Blancarte (2016)

Datos del autor:

Ciudad de México
e-mail: oscarblancarte3@gmail.com

Autor y Editor
© Oscar Javier Blancarte Iturralde

Queda expresamente prohibida la reproducción o transmisión, total o parcial, de este libro por cualquier forma o medio; ya sea impreso, electrónico o mecánico, incluso la grabación o almacenamiento informáticos sin la previa autorización escrita del autor.

Composición y redacción:
Oscar Javier Blancarte Iturralde

Edición:
Oscar Javier Blancarte Iturralde

Portada
Arq. Jaime Alberto Blancarte Iturralde

Primera edición
Se publicó por primera vez en Octubre del 2016
V1.5

Acerca del autor

Oscar Blancarte es originario de Sinaloa, México donde estudió la carrera de Ingeniería en Sistemas Computacionales y rápidamente se mudó a la Ciudad de México donde actualmente radica.

Oscar Blancarte es un Arquitecto de software con más de 11 años de experiencia en el desarrollo y arquitectura de software. Certificado como Java Programmer (Sun microsystems), Análisis y Diseño Orientado a Objetos (IBM) y Oracle IT Architect (Oracle).

A lo largo de su carrera, ha trabajado para diversas empresas del sector de TI, entre las que destacan su participación en diseños de arquitectura de software y consultoría para clientes de los sectores de Retail, Telco y Health Care.

Oscar Blancarte es además autor de su propio blog http://www.oscarblancarteblog.com desde el cual está activamente publicando temas interesantes sobre Arquitectura de software y temas relacionados con la Ingeniería de Software en general. Desde su blog ayuda a la comunidad a resolver dudas y es por este medio que se puede tener una interacción más directa con el autor.

Agradecimientos

Este libro tiene una especial dedicación a mi esposa, Liliana, por brindarme todo su apoyo y amor incondicional. Por estar siempre conmigo y comprenderme durante la redacción de este libro.

A mis padres, por todos los sacrificios que han hecho para hacerme la persona que hoy soy.

Prefacio

Este libro fue creado con la intención de enseñar a sus lectores a utilizar los patrones de diseño de una forma clara y simple, desde un enfoque práctico y con escenarios del mundo real.

Tengo que aceptar que este no es un libro convencional de patrones de diseño debido, principalmente, a que no sigue la misma estructura de las primordiales obras relacionadas con este tema. En su lugar, me quise enfocar en ofrecer una perspectiva del mundo real, en donde el lector pueda aprender a utilizar los patrones de diseño en entornos reales y que puedan ser aplicados a proyectos reales.
Cuando empecé a estudiar sobre patrones de diseño, me di cuenta que siempre se explicaban en escenarios irracionales que poco o ninguna vez podrías utilizar, como por ejemplo para aprender a crear figuras geométricas, hacer una pizza o crear una serie de clases de animales que ladren o maúllen; esos eran los ejemplos que siempre encontraba, que si bien explicaban el concepto, se complicaba entender cómo llevarlos a escenarios reales.

En este libro trato de ir un poco más allá de los ejemplos típicos para crear cosas realmente increíbles. Por ejemplo:

- Crear tu propia consola de línea de comandos.
- Crear tu propio lenguaje para realizar consultas SQL sobre un archivo de Excel.
- Crear aplicaciones que puedan cambiar entre más de una base de datos, por ejemplo, Oracle y MySQL según las necesidades del usuario.
- Administrar la configuración global de tu aplicación.
- Crear un Pool de ejecuciones para controlar el número de hilos ejecutándose simultáneamente, protegiendo nuestra aplicación para no agotar los recursos.
- Utilizar proxis para controlar la seguridad de tu aplicación.

- Utilizar estrategias para cambiar la forma en que los usuarios son autenticados en la aplicación; como podría ser por Base de datos, Webservice, etcétera.
- Crear tu propia máquina de estados para administrar el ciclo de vida de tu servidor.

Éstos son sólo algunos de los 25 ejemplos que abordaremos en este libro, los cuales están acompañados, en su totalidad, con el código fuente para que seas capaz de descargarlos, ejecutarlos y analizarlos desde tu propia computadora.

Finalmente, abordaremos los ejemplos con UML y conceptos fundamentales de la programación orientada a objetos como lo son la Herencia, Polimorfismo, Encapsulamiento, Abstracción, Cohesión y Acoplamiento; de los cuales hablaremos en las secciones A y B de este mismo libro.

Cómo utilizar este libro

Este libro está compuesto por tres grandes secciones, en las cuales se habla de los tres tipos de patrones de diseño; Creacionales, Estructurales y de Comportamiento. Cada patrón de diseño está estructurado de la siguiente manera:

- **Introducción:** Se explica de forma teórica en qué consiste el patrón de diseño, su estructura, componentes y cuándo deberíamos utilizarlo.
- **Escenario:** Se expone un escenario de la vida real en el cual podríamos utilizar el patrón de diseño para después solucionarlo.
- **Solución:** Se intenta resolver el escenario planteado utilizando patrones de diseño.
- **Implementación:** Manos a la obra, en esta sección utilizaremos código para programar la solución propuesta.
- **Ejecución:** Ejecutaremos el código desarrollado para comprobar los resultados tras implementar el patrón de diseño.
- **Siguientes pasos:** Breve resumen de lo aprendido y se plantean una serie de ejercicios para resolver con la finalidad de afianzar los conocimientos adquiridos.

El código lo puedes descargar en:

Repositorio:

Java: https://github.com/oscarjb1/IntroduccionPatronesDiseno.git
C#: https://github.com/oscarjb1/design-patterns-cs

Archivo descargable:

Java: https://github.com/oscarjb1/IntroduccionPatronesDiseno/archive/master.zip
C#: https://github.com/oscarjb1/design-patterns-cs.git

NOTA: Este libro tiene un gran diferencial al resto y es que puedes buscarme a través de mi blog http://www.oscarblancarteblog.com/libros/introduccion-los-patrones-diseno/para consultarme cualquier duda que tengas y con mucho gusto te responderé, de igual forma, puedes enviarme tus comentarios y sugerencias para mejorar este libro.

Requisitos previos

Este libro está pensado para programadores principiantes que emprenden su carrera como programadores de software o para programadores avanzados que no han tenido la oportunidad de tener un entrenamiento apropiado de los patrones de diseño. Es por esta razón que cualquiera con los conocimientos básicos de la programación orientada a objetos, podrá entender el contenido de este libro.

Cada patrón de diseño va a acompañado de un diagrama de clases y de secuencia en UML, por lo que conocimientos básicos de UML pueden ser requeridos para comprender mejor el patrón.

No te preocupes si no comprendes algunos de los conceptos básicos de POO o UML ya que en las secciones A y B abordaremos brevemente estos temas.

INTRODUCCIÓN

Los patrones de diseño tienen su origen en la Arquitectura, cuando en 1979 el Arquitecto Christopher Alexander publicó el libro *Timeless Way of Building*, en el cual hablaba de una serie de patrones para la construcción de edificios, comparando la arquitectura moderna con la antigua y cómo la gente había perdido la conexión con lo que se considera calidad.

Él utilizaba las siguientes palabras: "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."

Más tarde, Christopher Alexander y sus colegas publicaron el volumen *A Pattern Language* en donde intentaron formalizar y plasmar de una forma práctica las generaciones de conocimientos arquitectónicos. En la obra se refieren a los patrones arquitectónicos de la siguiente manera:

Los patrones no son principios abstractos que requieran su redescubrimiento para obtener una aplicación satisfactoria, ni son específicos a una situación particular o cultural; son algo intermedio. Un patrón define una posible solución correcta para un problema de diseño dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones. (1977).

Entre las cosas que se describen en el libro se encuentra la forma de diseñar ciudades y en qué lugar deben ir las perillas de las puertas.

Hasta ese momento los patrones conocidos tenían un enfoque arquitectónico y hablan de cómo construir estructuras, pero fue hasta 1987 cuando Ward Cunningham y Kent Beck, motivados por el pobre entrenamiento que recibían los nuevos programadores en programación orientación a objetos, se dieron cuenta de la gran semejanza que existían entre una buena arquitectura propuesta por Christopher Alexander y la buena arquitectura de la programación orientada a objetos. De tal manera que utilizaron gran parte del trabajo de Christopher para diseñar cinco patrones de interacción hombre-máquina y lo publicaron en el artículo OOPSLA-87 bajo el título *Using Pattern Languages for OO Programs*.

Sin embargo, fue hasta principios de la década de 1990 cuando los patrones de diseño tuvieron su gran debut en el mundo de la informática a partir de la publicación del libro *Design Patterns*, escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes que ya se utilizaban sin ser reconocidos como patrones de diseño.

Por lo tanto, podemos decir que un patrón de diseño es la solución a un problema de diseño, el cual debe haber comprobado su efectividad resolviendo problemas similares en el pasado, también tiene que ser reutilizable, por lo que se deben poder usar para resolver problemas parecidos en contextos diferentes.

En este libro utilizaremos esta definición para los patrones de diseño de software.

Índice

Acerca del autor ____ *4*

Agradecimientos ____ *6*

Prefacio ____ *7*

Cómo utilizar este libro ____ *9*

Requisitos previos ____ *10*

INTRODUCCIÓN ____ *11*

Índice ____ *13*

Importancia de los patrones de diseño ____ *18*

Tipos de patrones de diseño ____ *20*

Patrones Creacionales ____ *21*

Patrón Factory Method ____ **23**

El escenario: ____ 26
La solución: ____ 28
La implementación: ____ 29
La Ejecución: ____ 42
Siguientes pasos: ____ 46

Patrón Abstract Factory ____ **47**

El escenario: ____ 52
La solución: ____ 53
La implementación: ____ 55
La ejecución: ____ 65
Siguientes pasos: ____ 67

Patrón Singleton ____ **68**

El escenario: ____ 72
La solución: ____ 73
La implementación: ____ 74
La Ejecución ____ 79
Siguientes pasos: ____ 80

Patrón Builder ____ **81**

El escenario: ____ 84
La solución: ____ 86
La implementación: ____ 87

La ejecución: 95
Siguientes pasos: 96

Patrón Prototype 97
El escenario: 100
La solución: 101
La ejecución: 109
Siguientes pasos: 110

Patrón Object Pool 111
El escenario: 116
La solución: 117
La implementación: 120
La ejecución: 133
Siguientes pasos: 137

Patrones Estructurales 138

Patrón Adapter 140
El escenario: 143
La solución: 145
La implementación: 147
La ejecución: 158
Siguientes pasos: 159

Patrón Bridge 160
El escenario: 163
La solución: 165
La implementación: 167
La ejecución: 174
Siguientes pasos: 175

Patrón Composite 176
El Escenario: 179
La solución: 181
La implementación: 182
La Ejecución: 191
Siguientes pasos: 198

Patrón Decorator 199
El escenario: 202
La solución: 203
La implementación: 206
La ejecución: 215
Siguientes pasos: 217

Patrón Facade 218
El escenario: 221

La solución: 224
La implementación: 225
La ejecución: 239
Siguientes pasos: 241

Patrón Flyweight 242
El escenario: 246
La solución: 248
La implementación: 250
La ejecución: 257
Siguientes pasos: 261

Patrón Proxy 262
El escenario: 265
La solución: 267
La implementación: 268
La ejecución: 274
Siguientes pasos: 276

Patrones de Comportamiento 277

Patrón Iterator 280
El escenario: 283
La solución: 285
La implementación: 286
La ejecución: 292
Siguientes pasos: 293

Patrón Command 294
El escenario: 297
La solución: 300
La implementación: 303
Siguientes pasos: 331

Patrón Observer 332
El escenario: 335
La solución: 337
La implementación: 339
La ejecución: 347
Siguientes pasos: 348

Patrón Templete Method 349
El escenario: 352
La solución: 354
La implementación: 356
La ejecución: 370
Siguientes pasos: 373

Patrón Strategy 374
El escenario: 377
La solución: 379
La implementación: 381
La ejecución: 390
Siguientes pasos: 394

Patrón Chain of Reponsability 395
El escenario: 399
La solución: 401
La implementación: 403
La ejecución: 420
Siguientes pasos: 423

Patrón Interpreter 424
El escenario: 427
La solución: 430
La implementación: 433
La ejecución: 462
Siguientes pasos: 465

Patrón Mediator 466
El escenario: 469
La solución: 472
La implementación: 474
La ejecución: 490
Siguientes pasos: 492

Patrón Memento 493
El escenario: 496
La solución: 497
La implementación: 498
La ejecución: 506
Siguientes pasos: 507

Patrón Null Object 508
El escenario: 511
La solución: 512
La implementación: 513
La ejecución: 519
Siguientes pasos: 520

Patrón State 521
El escenario: 525
La solución: 528
La implementación: 530
La ejecución: 542
Siguientes pasos: 545

Patrón Visitor _____ **546**
El escenario: _____ 550
La solución: _____ 552
La implementación: _____ 554
La ejecución: _____ 570
Siguientes pasos: _____ 571

A – Programación Orientada a Objetos _____ ***572***

B – Introducción a UML _____ ***581***

CONCLUSIONES _____ ***589***

BIBLIOGRAFÍA _____ ***590***

Importancia de los patrones de diseño

Primero que nada, es importante mencionar que la utilización de patrones de diseño demuestra la madurez de un programador de software ya que utiliza soluciones probadas para problemas concretos que ya han sido probados en el pasado. Toma en cuenta que el dominio de los patrones de diseño es una práctica que se tiene que perfeccionar y practicar, es necesario conocer las ventajas y desventajas que ofrece cada uno de ellos, pero sobre todo requiere de experiencia para identificar dónde se deben de utilizar.
Lo más importante de utilizar los patrones de diseño es que evita tener que reinventar la rueda, ya que son escenarios identificados y su solución está documentada y probada por lo que no es necesario comprobar su efectividad. Además de esto, los patrones de diseño se basan en las mejores prácticas de programación.

Ahora, analizaremos qué se pretende lograr y qué no con los patrones de diseño. Los patrones de diseño pretenden:

- Proporcionar un catálogo de soluciones probadas de diseño para problemas comunes conocidos.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Crear un lenguaje estándar entre los desarrolladores.
- Facilitar el aprendizaje a nuevas generaciones de programadores.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Imponer la solución definitiva a un problema de diseño.
- Eliminar la creatividad inherente al proceso de diseño.

Tomemos en cuenta las ventajas que ofrecen los patrones de diseño, pero es importante recordar que no siempre son aplicables, por lo que forzar un patrón de diseño para resolver un problema incorrecto puede ser un gran error. Sin embargo, también existen los casos en los que podemos realizar pequeñas

variantes de los patrones para solucionar de una mejor manera ciertos escenarios.

Tipos de patrones de diseño

Durante la lectura de este libro estudiaremos una gran cantidad de patrones de diseño, los cuales como veremos más adelante, nos ayudarán a resolver problemas muy concretos y, a pesar que cada uno de éstos intenta resolver un problema distinto, podemos clasificarlos en tres grandes grupos:

Patrones Creacionales: Son patrones de diseño relacionados con la creación o construcción de objetos. Estos patrones intentan controlar la forma en que los objetos son creados implementando mecanismos que eviten la creación directa de objetos.

Patrones Estructurales: Son patrones que tiene que ver con la forma en que las clases se relacionan con otras clases. Estos patrones ayudan a dar un mayor orden a nuestras clases ayudando a crear componentes más flexibles y extensibles.

Patrones de Comportamiento: Son patrones que están relacionados con procedimientos y con la asignación de responsabilidad a los objetos. Los patrones de comportamiento engloban también patrones de comunicación entre ellos.

No te preocupes si en este punto no has logrado comprender estas definiciones, más adelante retomaremos con más detalle estos temas y los explicaremos de forma más detallada.

Patrones Creacionales

Como mencionamos anteriormente, los patrones creacionales nos sirven para controlar la forma en que creamos los objetos, de entrada nos puede parecer un poco extraño, ya que estamos acostumbrados a crear libremente nuestros objetos, sin embargo, existen situaciones donde por conveniencia es necesario establecer un mecanismo que nos permita crear instancias de una forma controlada. Esta necesidad puede nacer debido a que queremos que sólo exista una instancia de una clase o no se sabe exactamente qué objeto debemos instanciar sino hasta en tiempo de ejecución o porque queremos que nuestras clases sean creadas de una forma más simple mediante una clase de utilidad.

Pueden existir cientos de motivos por los cuales deseamos que nuestras clases sean creadas de forma controlada, sin embargo, lo importante es tener la visión de identificarlas y utilizar patrones creacionales que se adapten a nuestro problema.

Los patrones que abordaremos en este capítulo son:

Patrón Factory Method: Patrón que se centra en la creación de una clase fábrica la cual tiene métodos que nos permitan crear objetos de un subtipo determinado.

Patrón Abstract Factory: Patrón muy similar al Factory Method, sin embargo, este patrón nos permite crear objetos de una determinada familia de clases.

Patrón Singleton: Patrón utilizado para controlar la creación de una clase determinada, de esta forma sólo se puede crear una única instancia en toda la aplicación.

Patrón Builder: Patrón que permitir la creación de objetos complejos desde un objeto *Builder*. El objeto *Builder* se compone de una variedad de partes que contribuyen individualmente a la creación del objeto.

Patrón Prototype: Este patrón se centra en la creación de objetos a partir de la clonación de otros objetos existentes. Es mucho más rápido clonar un objeto que crear uno nuevo.

Patrón Object Pool: Patrón que se utiliza para mantener un conjunto de objetos creados listos para ser utilizados, evitando crearlos bajo demanda cada vez que se requieran. Los objetos desocupados son devueltos al *pool* en vez de destruirse (muy utilizado para *Pool* de Conexiones).

Patrón Factory Method

Factory Method permite la creación de objetos de un subtipo determinado a través de una clase *Factory*. Esto es especialmente útil cuando no sabemos, en tiempo de diseño, el subtipo que vamos a utilizar o cuando queremos delegar la lógica de creación de los objetos a una clase *Factory*. Utilizando este patrón podemos crear instancias dinámicamente mediante la configuración, estableciendo cual será la implementación a utilizar en un archivo de texto, XML, *properties* o mediante cualquier otra estrategia.

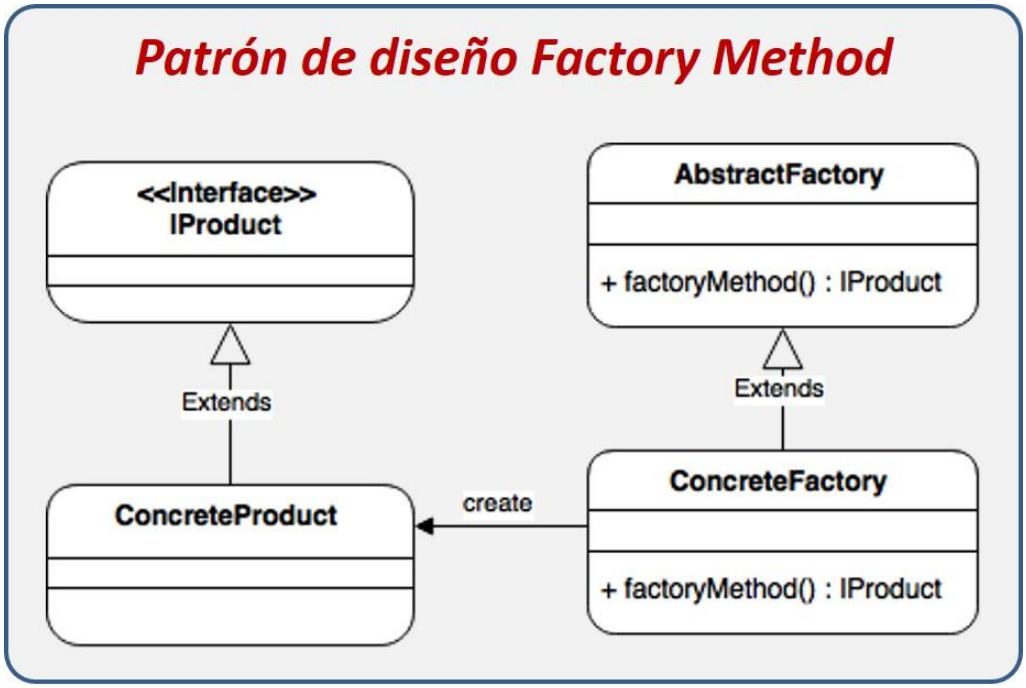


Ilustración 1: Estructura del patrón de diseño Factory Method

Los componentes que conforman el patrón son los siguientes:

- **IProduct:** Representa de forma abstracta el objeto que queremos crear, mediante esta interface se definen la estructura que tendrá el objeto creado.

- **ConcreteProduct:** Representa una implementación concreta de la interface *IProduct*, la cual es creada a través del *ConcreteFactory*.
- **AbstractFactory:** Este componente puede ser opcional, sin embargo, se recomienda la creación de un *AbstractFactory* que define el comportamiento por default de los *ConcreteFactory*.
- **Concrete Factory**: Representa una fábrica concreta la cual es utilizada para la creación de los *ConcreteProduct*, esta clase hereda el comportamiento básico del *AbstractFactory*.

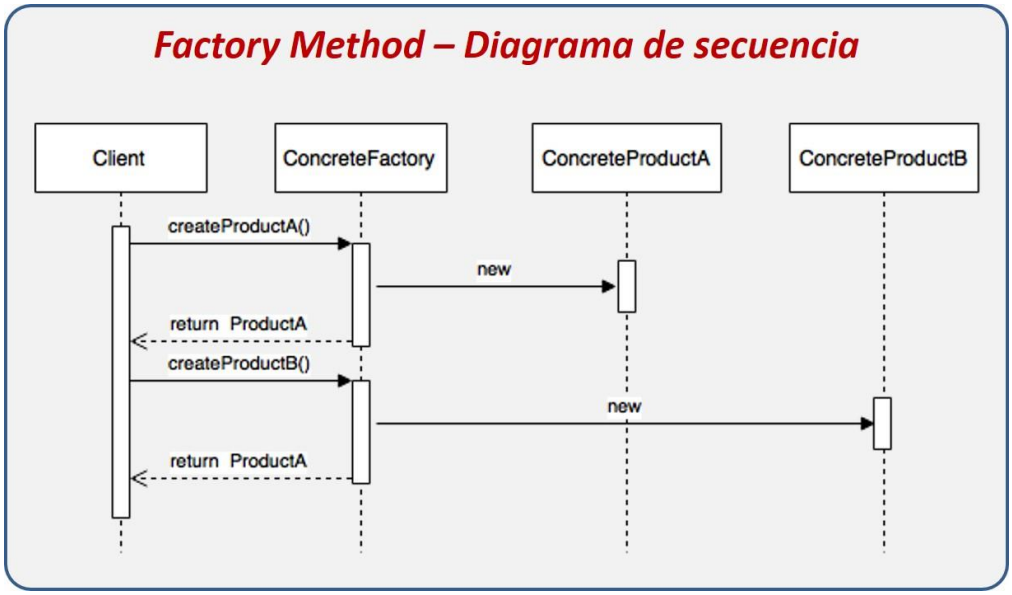


Ilustración 2: Diagrama de secuencia del patrón de diseño Factory.

El diagrama se interpreta de la siguiente manera:

1. El cliente le solicita al *ConcreteFactory* la creación del *ProductA*.
2. El *ConcreteFactory* localiza la implementación concreta de *ProductA* y crea una nueva instancia.
3. El *ConcreteFactory* regresa el *ConcreteProductA* creado.
4. El cliente le solicita al *ConcreteFactory* la creación del *ProductB*.
5. El *ConcreteFactory* localiza la implementación concreta del *ProductB* y crea una nueva instancia.
6. El *ConcreteFactory* regresa el *ConcreteProductB* creado.

Cuándo utilizarlo:

- Cuando la creación directa de un objeto por medio del operador new puede ser perjudicial.
- Cuando no se conoce en tiempo de diseño, la subclase que se utilizará.
- Cuando es necesario controlar la creación de objetos por medio de una interface común.
- Cuando construimos un objeto basado en una serie de condiciones else if o switch.

El escenario:

Es probable que en este punto no tengamos muy claro cómo utilizar este patrón, por lo que lo explicaremos con un ejemplo: Imaginemos un escenario donde deseamos tener la opción de conectarnos a dos bases de datos distintas, como Oracle y MySQL, esto podría darse por la necesidad de darle a los usuarios de nuestra aplicación la posibilidad de tener una base de datos robusta como Oracle, o una opción más económica como MySQL. Sea cual sea el motivo por el cual el usuario decide utilizar una base de datos, nosotros tenemos que tener los mecanismos para soportarla.

Para esto desarrollaremos una clase de acceso a datos (DAO) de productos que nos permite guardar productos y consultarlos, el principal objetivo es que el cliente pueda utilizar el mismo DAO sin la necesidad de cambiar de clase dependiendo la base de datos a utilizar.

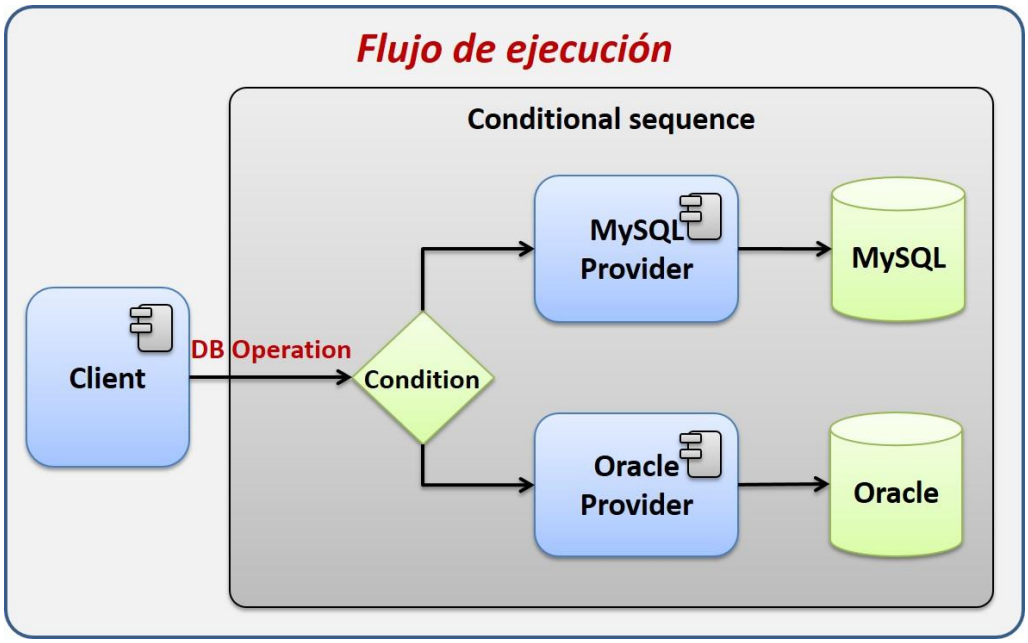


Ilustración 3: Ejecución sin el patrón Factory Method.

En la imagen podemos ver una arquitectura bastante típica en donde la lógica de la ejecución se decide mediante condiciones. Este tipo de diseños puede llegar a

ser bastante anticuado debido a que tendremos que codificar estas condiciones en cualquier parte en donde se requiere utilizar la capa de persistencia, tendríamos que saber qué proveedor de base de datos estamos utilizando y después utilizar el componente para realizar las transacciones para ese tipo de base de datos. En la siguiente sección veremos cómo mejorar esto utilizando el patrón *Factory Method*.

La solución:

Ya analizamos la problemática de resolver este tipo de requerimientos mediante el condicionamiento, por lo que esta vez utilizaremos el patrón de diseño *Factory Method* para que se encargue de la creación del componente adecuado para transaccionar con la base de datos.

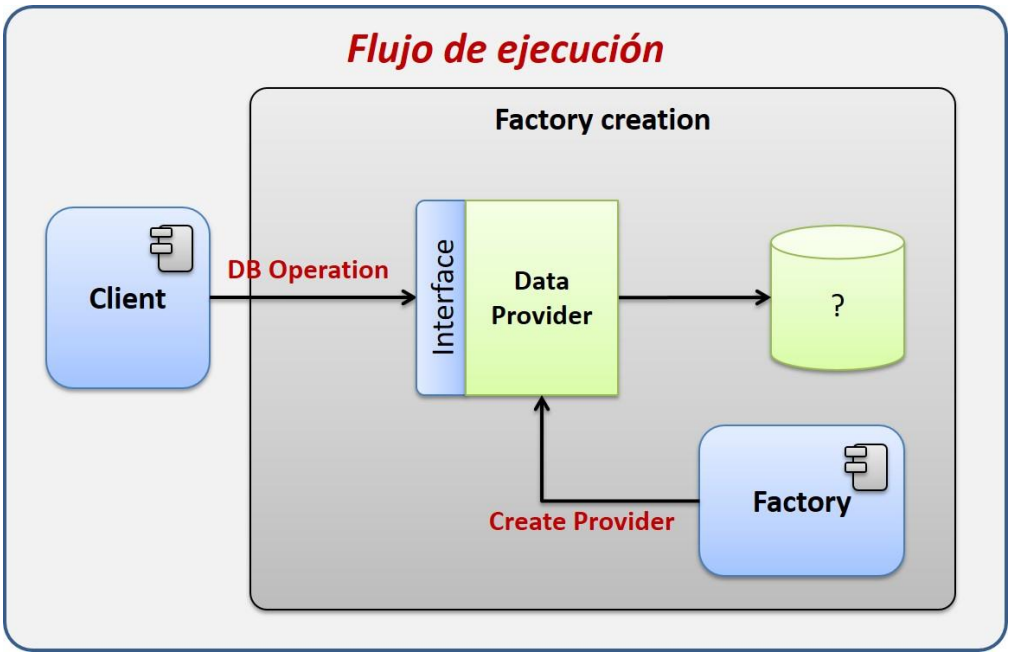


Ilustración 4: Flujo de ejecución utilizando Factory Method.

En la imagen podemos apreciar que en lugar de tener una condición para determinar el componente a utilizar, le dejamos esta responsabilidad al *Factory*, el cual creará la instancia concreta para transaccionar con la base de datos, ¿qué instancia nos regresará? no lo sabemos y eso es lo interesante, todas las instancias que regresa el *Factory* cumplirán el mismo contrato, por lo que podremos utilizar la instancia que sea sin importar a qué base de datos se conecte, también podemos apreciar que la base de datos es desconocida y será hasta en tiempo de ejecución que sabremos sobre qué base de datos estaríamos trabajando.

La implementación:

Hasta este punto ya analizamos el patrón y sabemos el problema que hay que resolver, por lo que ya sólo nos falta llevar la solución a la implementación, para lo cual iniciaremos con la construcción.

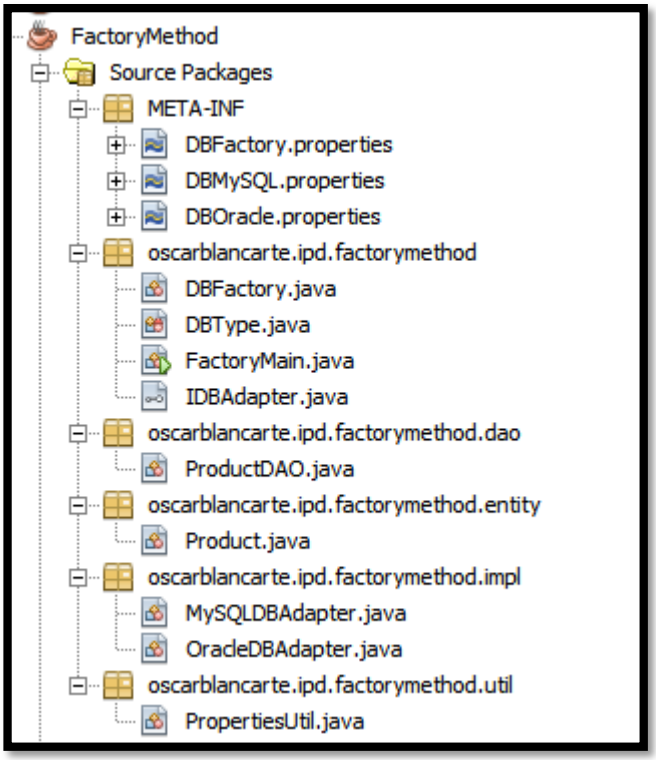


Ilustración 5: Estructura del proyecto FactoryMethod.

Los paquetes que componen el proyecto son los siguientes:

- **oscarblancarte.ipd.factorymethod:** Paquete principal del proyecto en donde se encuentra la clase ejecutable y las clases base para crear el *Factory*.
- **oscarblancarte.ipd.factorymethod.dao:** En este paquete se encuentra nuestra clase de acceso a datos *ProductDAO* la cual utilizará el *Factory* para obtener las conexiones a la base de datos.

- **oscarblancarte.ipd.factorymethod.entity:** Contiene las clases de entidad para persistir los productos.
- **oscarblancarte.ipd.factorymethod.impl:** Contiene las clases concretas que creará nuestro *Factory*.

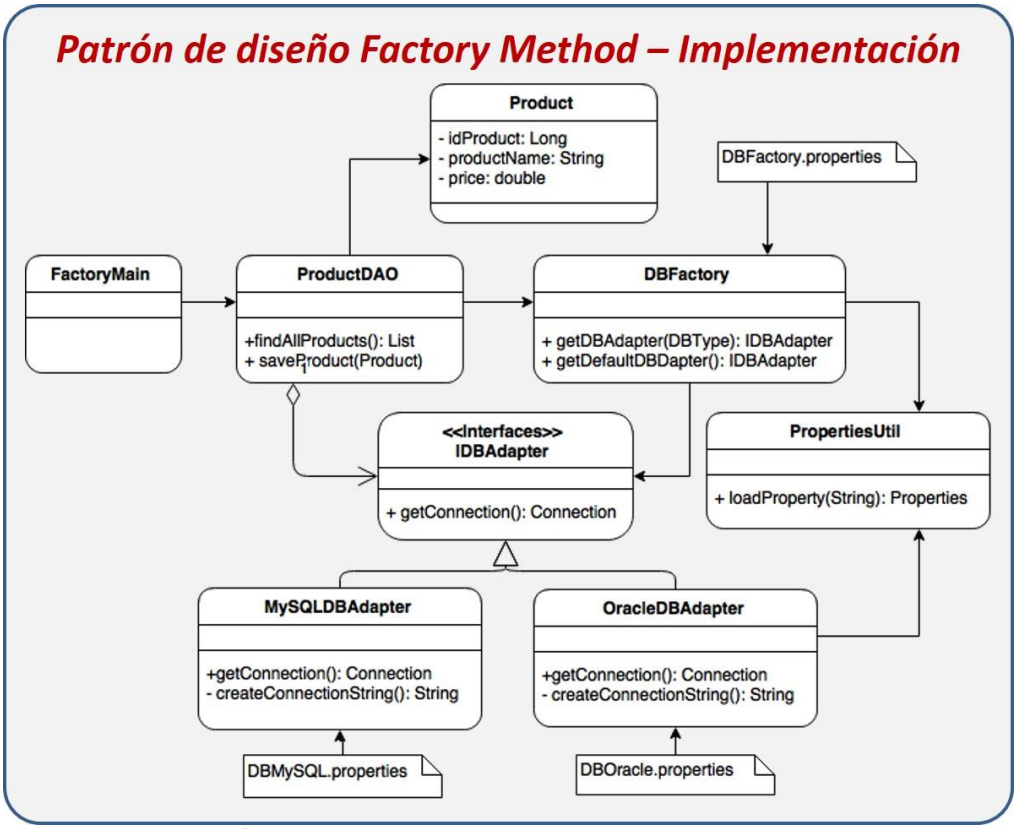


Ilustración 6: Diagrama de clases del proyecto FactoryMethod.

En la imagen anterior podemos apreciar las clases que componen el proyecto y cómo es que éstas se relacionan entre sí.

Scripts

Antes de comenzar, crearemos las tablas necesarias en Oracle y MySQL, los *scripts* son los siguientes:

```
1.  -- Script para Oracle
2.  CREATE TABLE productos (
3.    idProductos NUMERIC(10,0) NOT NULL,
4.    productName VARCHAR(100) NOT NULL,
5.    productPrice DECIMAL(10,2) NOT NULL
6.  );
```

```
1.  -- Script para MySQL
2.  CREATE TABLE `productos` (
3.    `idProductos` INT NOT NULL,
4.    `productName` VARCHAR(100) NOT NULL,
5.    `productPrice` DECIMAL NOT NULL DEFAULT 0.0,
6.    PRIMARY KEY (`idProductos`),
7.    UNIQUE INDEX `productName_UNIQUE` (`productName` ASC))
8.  ENGINE = InnoDB
9.  COMMENT = 'Tabla de productos';
```

DBFactory.properties

Archivo de propiedades para determinar la clase a fabricar, el símbolo # es utilizado como comentario, por lo que estas líneas serán ignoradas en tiempo de ejecución.

```
1.  #defaultDBClass oscarblancarte.tsas.factorymethod.impl.OracleDBAdapter
2.  defaultDBClass oscarblancarte.tsas.factorymethod.impl.MySQLDBAdapter
```

DBOracle.properties

Archivo de propiedades donde definimos los datos de conexión para conectarnos a la base de datos de Oracle. Este archivo será utilizado más adelante para crear la conexión con Oracle.

```
1.  host localhost
2.  port 1521
3.  service xe
4.  user sys as sysdba
5.  password 1234
```

DBMySQL.properties

Archivo de propiedades donde definimos los datos de conexión para conectarnos a la base de datos de MySQL. Este archivo será utilizado más adelante para crear la conexión con MySQL.

```
1. host localhost
2. port 3306
3. dbname pos
4. user root
5. password 1234
```

Interface IDBAdapter:

Esta interface define la estructura de los productos que podrá crear el *Factory*, en este caso habrá dos clases concretas, una para MySQL y otra para Oracle las cuales veremos más adelante.

La interface define el método getConnection el cual crea las conexiones a la base de datos, las clases concretas deberán implementar la lógica para realizar esta conexión.

```
1. package oscarblancarte.ipd.factorymethod;
2.
3. import java.sql.Connection;
4.
5. public interface IDBAdapter {
6.     public Connection getConnection();
7. }
```

Clase OracleDBAdapter:

En *OracleDBAdapter* lo primero que podemos apreciar es que implementan a la interface *IDBAdapter* y el método getConnection para regresar una conexión abierta a la base de datos Oracle. En la línea 21 creamos un bloque *static* para asegurarnos de que el *Driver* JDBC de Oracle sea registrado antes de que el método getConnection sea ejecutado.

Para la creación de la cadena de conexión nos apoyaremos del método createConnectionString, el cual regresa una cadena de conexión apropiada para conectarnos a Oracle. En la línea 46 podemos apreciar que se utiliza una clase de utilería llamada *PropertiesUtil*, la cual mostraremos un poco más adelante, por lo pronto sólo necesitaremos saber que esta clase lee el archivo de propiedades *DBOracle.properties* para obtener los parámetros de conexión.

La variable DB_PROPERTIES es utilizada para determinar la ubicación del archivo de propiedades y establecer conexión con Oracle, en este caso la ubicación es META-INF/DBOracle.properties.

```
1.  package oscarblancarte.ipd.factorymethod.impl;
2.
3.  import java.sql.Connection;
4.  import java.sql.DriverManager;
5.  import java.sql.Statement;
6.  import java.util.Properties;
7.  import oracle.jdbc.OracleDriver;
8.  import oscarblancarte.ipd.factorymethod.IDBAdapter;
9.  import oscarblancarte.ipd.factorymethod.util.PropertiesUtil;
10.
11. public class OracleDBAdapter implements IDBAdapter {
12.
13.     private static final String DB_PROPERTIES = "META-INF/DBOracle.properties";
14.
15.     private static final String DB_SERVICE_PROP = "service";
16.     private static final String DB_HOST_PROP = "host";
17.     private static final String DB_PASSWORD_PROP = "password";
18.     private static final String DB_PORT_PROP = "port";
19.     private static final String DB_USER_PROP = "user";
20.
21.     static {
22.         //Bloque para registrar el Driver de Oracle
23.         try {
24.             new OracleDriver();
25.         } catch (Exception e) {
26.             e.printStackTrace();
27.         }
28.     }
29.
30.     @Override
31.     public Connection getConnection() {
32.         try {
33.             String connectionString = createConnectionString();
34.             Connection connection = DriverManager
35.                 .getConnection(connectionString);
```

```
36.             System.out.println("Connection class ==> "
37.                 +connection.getClass().getName());
38.             return connection;
39.         } catch (Exception e) {
40.             e.printStackTrace();
41.             return null;
42.         }
43.     }
44.
45.     private String createConnectionString() {
46.         Properties prop = PropertiesUtil.loadProperty(DB_PROPERTIES);
47.         String host = prop.getProperty(DB_HOST_PROP);
48.         String port = prop.getProperty(DB_PORT_PROP);
49.         String service = prop.getProperty(DB_SERVICE_PROP);
50.         String user = prop.getProperty(DB_USER_PROP);
51.         String password = prop.getProperty(DB_PASSWORD_PROP);
52.
53.         String connectionString = "jdbc:oracle:thin:"
54.                 +user+"/"+password+"@//"+host+":"+port+"/"+service;
55.         System.out.println("ConnectionString ==> " + connectionString);
56.         return connectionString;
57.     }
58. }
```

Clase MySQLDBAdapter:

Esta clase es muy parecida a la anterior, sin embargo, ésta crea una conexión con MySQL leyendo el archivo de propiedades *DBMySQL.properties*.

```
1.  package oscarblancarte.ipd.factorymethod.impl;
2.
3.  import java.sql.Connection;
4.  import java.sql.DriverManager;
5.  import java.sql.Statement;
6.  import java.util.Properties;
7.  import oscarblancarte.ipd.factorymethod.IDBAdapter;
8.  import oscarblancarte.ipd.factorymethod.util.PropertiesUtil;
9.
10.     public class MySQLDBAdapter implements IDBAdapter {
11.
12.         private static final String DB_PROPERTIES = "META-INF/DBMySQL.properties";
13.
14.         //Propiedades de los archivos properties
15.         private static final String DB_NAME_PROP = "dbname";
16.         private static final String DB_HOST_PROP = "host";
17.         private static final String DB_PASSWORD_PROP = "password";
18.         private static final String DB_PORT_PROP = "port";
19.         private static final String DB_USER_PROP = "user";
```

```
20.
21.         static {
22.             //Bloque para registrar el Driver de MySQL
23.             try {
24.                 new com.mysql.jdbc.Driver();
25.             } catch (Exception e) {
26.                 e.printStackTrace();
27.             }
28.         }
29.
30.         @Override
31.         public Connection getConnection() {
32.             try {
33.                 String connectionString = createConnectionString();
34.                 Connection connection = DriverManager
35.                     .getConnection(connectionString);
36.                 System.out.println("Connection class ==> "
37.                     + connection.getClass().getName());
38.                 return connection;
39.             } catch (Exception e) {
40.                 e.printStackTrace();
41.                 return null;
42.             }
43.
44.         }
45.
46.         private String createConnectionString() {
47.             Properties prop = PropertiesUtil.loadProperty(DB_PROPERTIES);
48.             String host = prop.getProperty(DB_HOST_PROP);
49.             String port = prop.getProperty(DB_PORT_PROP);
50.             String db = prop.getProperty(DB_NAME_PROP);
51.             String user = prop.getProperty(DB_USER_PROP);
52.             String password = prop.getProperty(DB_PASSWORD_PROP);
53.
54.             String connectionString = "jdbc:mysql://" + host
55.                 + ":" + port + "/" + db + "?user=" + user + "&password=" + password;
56.             System.out.println("ConnectionString ==> " + connectionString);
57.             return connectionString;
58.         }
59.     }
```

Clase DBFactory

Esta clase representa el *ConcreteFactory* y es utilizada para fabricar los adaptadores de conexión o los subtipos de *IDBAdapter* como *OracleDBAdapter* y *MySQLDBAdapter*.

La clase define los siguientes métodos:

- getDBAdapter: Método que nos permite solicitarle explícitamente qué tipo de conexión deseamos por medio del parámetro *dbType*, que está definido como una Enumeration, la cual nos permite elegir entre Oracle y MySQL. Este método nos puede servir cuando sabemos de antemano qué conexión deseamos elegir o tenemos la configuración del tipo de base de datos en una configuración externa.

- getDefaultDBAdapter: Es una operación un poco más audaz ya que nos permite obtener una instancia de IDBAdapter previamente configurada, esto es posible mediante un archivo de configuración que es cargado en la línea 26 y 27. Una vez cargado el nombre de la clase, ésta es instanciada en la línea 29 y devuelta por el *Factory*. Este método es particularmente útil cuando podemos pre-configurar qué tipo de conexión estaremos utilizando durante toda la ejecución del programa. Esta configuración es cargada del archivo *DBFactory.properties.*

```
1.  package oscarblancarte.ipd.factorymethod;
2.
3.  import java.util.Properties;
4.  import oscarblancarte.ipd.factorymethod.impl.MySQLDBAdapter;
5.  import oscarblancarte.ipd.factorymethod.impl.OracleDBAdapter;
6.  import oscarblancarte.ipd.factorymethod.util.PropertiesUtil;
7.
8.  public class DBFactory {
9.
10.     private static final String DB_FACTORY_PROPERTY_URL = "META-INF/DBFactory.properties";
11.     private static final String DEFAULT_DB_CLASS_PROP = "defaultDBClass";
12.
13.     public static IDBAdapter getDBadapter(DBType dbType) {
14.         switch (dbType) {
15.             case MySQL:
16.                 return new MySQLDBAdapter();
17.             case Oracle:
18.                 return new OracleDBAdapter();
19.             default:
20.                 throw new IllegalArgumentException("No soportado");
21.         }
22.     }
23.
24.     public static IDBAdapter getDefaultDBAdapter() {
25.         try {
26.             Properties prop = PropertiesUtil.loadProperty(DB_FACTORY_PROPERTY_URL);
27.             String defaultDBClass = prop.getProperty(DEFAULT_DB_CLASS_PROP);
28.             System.out.println("DefaultDBClass ==> " + defaultDBClass);
```

```
29.             return (IDBAdapter) Class.forName(defaultDBClass).newInstance();
30.         } catch (Exception e) {
31.             e.printStackTrace();
32.             return null;
33.         }
34.     }
35. }
```

Enumeration DBType

Enumeración utilizada para definir los tipos de base de datos soportadas.

```
1.  package oscarblancarte.ipd.factorymethod;
2.
3.  public enum DBType {
4.      MySQL, Oracle,
5.  }
```

Clase Product:

Clase utilizada para representar los registros de la base de datos como Objetos, esta clase representa un producto, el cual tiene las propiedades:

- **idProduct:** Identificador único del producto.
- **productName:** Nombre del producto.
- **Price:** Precio del producto.

```
1.  package oscarblancarte.ipd.factorymethod.entity;
2.
3.  public class Product {
4.      private Long idProduct;
5.      private String productName;
6.      private double price;
7.
8.      public Product(Long idProduct, String productName, double price) {
9.          this.idProduct = idProduct;
10.         this.productName = productName;
11.         this.price = price;
12.     }
13.     */Get and Set*/
14. }
```

Clase ProductDAO:

El primer punto relevante que podemos apreciar en la clase *ProductDAO* es que el constructor manda llamar a nuestro factor *DBFactory* con el fin de obtener el *IDBAdapter***,** por defecto, esta configuración le da la ventaja al *ProductDAO* de trabajar siempre de la misma manera sin importar qué base de datos esté utilizando.

La clase define los siguientes métodos:

- **findAllProduct***:* Utilizado para consultar todos los productos de la base de datos
- **saveProduct***:* Utilizado para guardar un nuevo producto.

Si ya hemos trabajado con base de datos, nada de lo que contenga esta clase nos parecerá algo nuevo. Sin embargo, en la línea 21 y 44 podemos observar cómo es que nuestra conexión es creada a partir del *IDBAdapter* construido por nuestro *Factory*, veamos que sin importar cuál sea la base de datos utilizada funcionará sin tener que recompilar o realizar algún cambio en la aplicación.

```
1.  package oscarblancarte.ipd.factorymethod.dao;
2.
3.  import java.sql.Connection;
4.  import java.sql.PreparedStatement;
5.  import java.sql.ResultSet;
6.  import java.util.ArrayList;
7.  import java.util.List;
8.  import oscarblancarte.ipd.factorymethod.IDBAdapter;
9.  import oscarblancarte.ipd.factorymethod.DBFactory;
10. import oscarblancarte.ipd.factorymethod.entity.Product;
11.
12. public class ProductDAO {
13.
14.     private IDBAdapter dbAdapter;
15.
16.     public ProductDAO(){
17.         dbAdapter = DBFactory.getDefaultDBAdapter();
18.     }
19.
20.     public List<Product> findAllProducts(){
21.         Connection connection = dbAdapter.getConnection();
22.         List<Product> productList = new ArrayList<>();
```

```
23.         try {
24.             PreparedStatement statement = connection
25.                     .prepareStatement("SELECT idProductos,productName"
26.                             + ",productPrice FROM Productos");
27.             ResultSet results = statement.executeQuery();
28.             while(results.next()){
29.                 productList.add(new Product(results.getLong(1),
30.                         results.getString(2), results.getDouble(3)));
31.             }
32.             return productList;
33.         } catch (Exception e) {
34.             e.printStackTrace();
35.             return null;
36.         }finally{
37.             try {
38.                 connection.close();
39.             } catch (Exception e) {}
40.         }
41.     }
42.
43.     public boolean saveProduct(Product product){
44.         Connection connection = dbAdapter.getConnection();
45.         try {
46.             PreparedStatement statement = connection
47.                     .prepareStatement("INSERT INTO Productos(idProductos,"
48.                             + "productName, productPrice) VALUES (?,?,?)");
49.             statement.setLong(1, product.getIdProduct());
50.             statement.setString(2, product.getProductName());
51.             statement.setDouble(3, product.getPrice());
52.             statement.executeUpdate();
53.             return true;
54.         } catch (Exception e) {
55.             e.printStackTrace();
56.             return false;
57.         }finally{
58.             try {
59.                 connection.close();
60.             } catch (Exception e) {}
61.         }
62.     }
63.
64. }
```

Clase PropertiesUtil

Esta clase es utilizada como clase de utilidad, define únicamente el método *loadProperty*, el cual lee un archivo de propiedades determinado y lo carga en un objeto de tipo Properties. La URL del archivo a cargar es pasada como parámetro.

```
1.  package oscarblancarte.ipd.factorymethod.util;
2.
3.  import java.io.InputStream;
4.  import java.security.AuthProvider;
5.  import java.util.Properties;
6.
7.  /**
8.   * @author oblancarte
9.   */
10.     public class PropertiesUtil {
11.         public static Properties loadProperty(String propertiesURL){
12.             try {
13.                 Properties properties = new Properties();
14.                 InputStream inputStream = PropertiesUtil.class
15.                     .getClassLoader().getResourceAsStream(propertiesURL);
16.                 properties.load(inputStream);
17.                 return properties;
18.             } catch (Exception e) {
19.                 e.printStackTrace();
20.                 return null;
21.             }
22.         }
23.     }
```

Clase FactoryMain:

Finalmente, la clase *FactoryMain* nos permitirá ejecutar la aplicación y ver las ventajas que trae este patrón.

Analicemos la aplicación:

En la línea 12 y 13 creamos dos productos nuevos.

En la línea 16 creamos nuestro *ProductDAO* para transaccionar con la base de datos; si recordamos, el constructor de esta clase utiliza el *Factory* para determinar la base de datos a utilizar por lo que ya no hay que preocuparnos por esto.

En las líneas 19 y 20 guardamos los productos.

Finalmente, de la línea 23 en adelante consultamos los productos y los imprimimos en pantalla.

La pregunta sería: ¿nos hemos preocupado por determinar la base de datos?, la verdad es que este tema ni siquiera nos interesó, simplemente creamos el DAO, guardamos y consultamos sin interesarnos a qué base de datos estamos conectados.

```
1.  package oscarblancarte.ipd.factorymethod;
2.
3.  import java.sql.SQLException;
4.  import java.util.List;
5.  import oscarblancarte.ipd.factorymethod.dao.ProductDAO;
6.  import oscarblancarte.ipd.factorymethod.entity.Product;
7.
8.  public class FactoryMain {
9.
10.         public static void main(String[] args) throws SQLException {
11.             //Creamos los nuevos productos a registrar
12.             Product productA = new Product(1L, "Producto A", 100);
13.             Product productB = new Product(2L, "Producto B", 100);
14.
15.             //Creamos una instancia del DAO
16.             ProductDAO productDAO = new ProductDAO();
17.
18.             //Persistimos los productos
19.             productDAO.saveProduct(productA);
20.             productDAO.saveProduct(productB);
21.
22.             //Consultamos nuevamente los productos
23.             List<Product> products = productDAO.findAllProducts();
24.             System.out.println("Product size ==> " + products.size());
25.             for(Product product : products){
26.                 System.out.println(product);
27.             }
28.         }
29.     }
```

La Ejecución:

Para comprobar la utilidad del patrón, ejecutaremos el programa conectándonos a la base de datos de Oracle. Editemos el archivo *DBFactory.properties* de la siguiente manera:

```
1. defaultDBClass oscarblancarte.ipd.factorymethod.impl.OracleDBAdapter
2. #defaultDBClass oscarblancarte.ipd.factorymethod.impl.MySQLDBAdapter
```

Seguido, ejecutaremos la clase *FactoryMethodMain* para tener el siguiente resultado:

```
run:
DefaultDBClass ==> oscarblancarte.tsas.factorymethod.impl.OracleDBAdapter
ConnectionString ==> jdbc:oracle:thin:sys as sysdba/1234@//localhost:1521/xe
Product size ==> 0
ConnectionString ==> jdbc:oracle:thin:sys as sysdba/1234@//localhost:1521/xe
ConnectionString ==> jdbc:oracle:thin:sys as sysdba/1234@//localhost:1521/xe
ConnectionString ==> jdbc:oracle:thin:sys as sysdba/1234@//localhost:1521/xe
Product size ==> 2
Product{idProduct=1, productName=Producto A, price=100.0}
Product{idProduct=2, productName=Producto B, price=100.0}
BUILD SUCCESSFUL (total time: 2 seconds)
```

Ilustración 7: Resultados con la configuración de Oracle.

Analizando la salida del programa podemos apreciar lo siguiente:

La línea 1 nos indica que la clase creada por el *Factory* fue *OracleDBAdapter*, la cual es configurada desde el archivo de propiedades *DBFactory.properties*.

En la segunda línea vemos el *String* de conexión creado por la clase *OracleDBAdapter* mediante la configuración del archivo *DBOracle.properties*.

En la línea 3 imprimimos el número de registros que hay en la base de datos por el momento.

Después, insertamos dos productos en la base de datos e imprimimos los datos de dichos productos.

Resultado de la Consulta

SQL | Todas las Filas Recuperadas: 2 en 0.046 segundos

	IDPRODUCTOS	PRODUCTNAME	PRODUCTPRICE
1	1	Producto A	100
2	2	Producto B	100

Ilustración 8: Consulta a la base de datos Oracle.

En la imagen podemos apreciar cómo es que los registros se crearon correctamente en la base de datos Oracle.

Resultados con MySQL

La siguiente prueba la realizaremos con MySQL por lo que modificaremos el archivo *DBFactory.properties* de la siguiente manera:

```
1. #defaultDBClass oscarblancarte.ipd.factorymethod.impl.OracleDBAdapter
2. defaultDBClass oscarblancarte.ipd.factorymethod.impl.MySQLDBAdapter
```

Ejecutamos la clase *FactoryMethodMain* para obtener el siguiente resultado:

```
run:
DefaultDBClass ==> oscarblancarte.tsas.factorymethod.impl.MySQLDBAdapter
ConnectionString ==> jdbc:mysql://localhost:3306/pos?user=root&password=1234
Product size ==> 0
ConnectionString ==> jdbc:mysql://localhost:3306/pos?user=root&password=1234
ConnectionString ==> jdbc:mysql://localhost:3306/pos?user=root&password=1234
ConnectionString ==> jdbc:mysql://localhost:3306/pos?user=root&password=1234
Product size ==> 2
Product{idProduct=1, productName=Producto A, price=100.0}
Product{idProduct=2, productName=Producto B, price=100.0}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 9: Resultados con la configuración de MySQL

Observemos que el resultado es casi idéntico al de Oracle, sin embargo, tiene las siguientes diferencias:

1. Cambia la clase creada por el *Factory*, esta vez crea una instancia de la clase *MySQLDBAdapter* la cual se configuró desde el archivo *DBFactory.properties*.
2. La cadena de conexión cambió para adaptarse a la forma de conexión de MySQL.

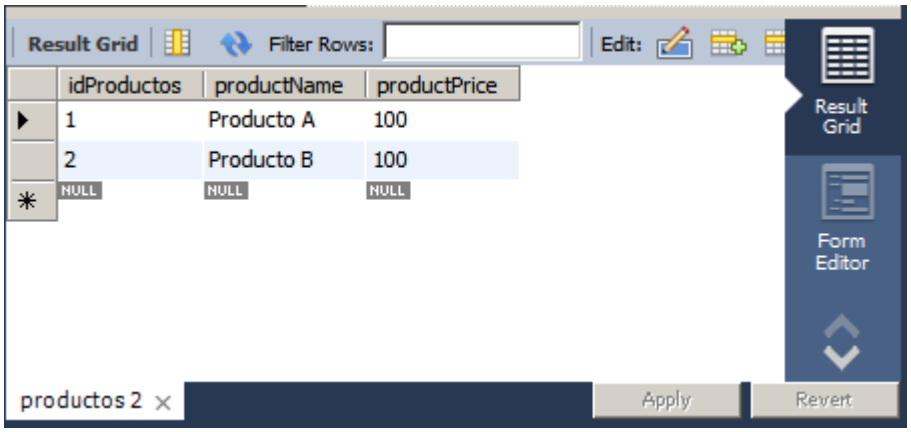


Ilustración 10: Resultados de la ejecución en la base de datos.

En la imagen podemos apreciar cómo es que los registros se crearon correctamente en la base de datos MySQL.

Como podemos apreciar no se requirió realizar ningún cambio en el programa para que la aplicación se adaptara a la nueva base de datos, simplemente fue necesario crear una subclase de *IDBAdapter* para poder conectarnos a la base de datos que queramos.

Siguientes pasos:

Hemos implementado una fábrica de conexiones para Oracle y para MySQL, sin embargo, esto no termina aquí, ya que podríamos crear nuevas clases que extiendan de *IDBAdapter* para implementar conexiones con otras bases de datos como SQL Server, Derby o Postgres. En este ejemplo hemos construido conexiones de base de datos, no obstante, podemos utilizar el patrón para construir otro tipo de objetos.

Para reforzar los conocimientos adquiridos podríamos realizar los siguientes ejercicios:

1. Crear una nueva subclase de *IDBAdapter* para conectarnos a Microsoft SQL Server o Postgres.
2. Modificar el programa para que pregunte al usuario qué base de datos quiere utilizar en vez de tomar la implementación por default configurada en el archivo *DBFactory.properties*.

Patrón Abstract Factory

El patrón de diseño Abstract Factory busca agrupar un conjunto de clases que tiene un funcionamiento en común, llamadas familias, las cuales son creadas mediante un *Factory*. Este patrón es especialmente útil cuando requerimos tener ciertas familias de clases para resolver un problema, sin embargo, puede que se requieran crear implementaciones paralelas de estas clases para resolver el mismo problema pero con una implementación distinta.

Se le conoce como familia a un conjunto de clases que buscan resolver la misma problemática, por ejemplo, los driver de JDBC, todas estas clases implementan las interfaces estándares de Java; sin embargo, cada *driver* tiene implementaciones distintas para conectarse a una base de datos específica.

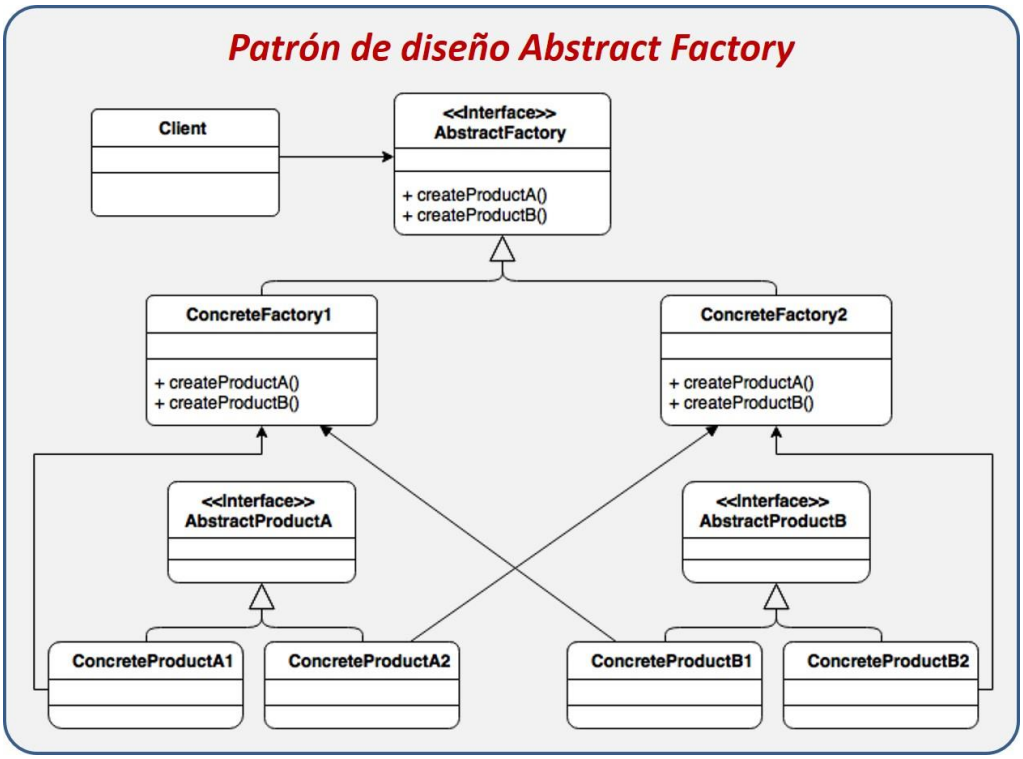


Ilustración 11: Estructura del patrón de diseño Abstract Factory.

El patrón de diseño *Abstract Factory* se puede llegar a confundir con el patrón *Factory Method*, la diferencia entre ambos es que *Abstract Factory* se enfoca en la creación de familias completas de objetos y el *Factory Method* se enfoca en la creación individual de objetos.

La estructura de *Abstract Factory* puede resultar muy enredosa ya que tiene muchos componentes y éstos aparentemente se mezclan entre sí. Para comprender mejor cómo funciona este patrón explicaremos cada componente.

- **Client:** Representa la persona o evento que dispara la ejecución del patrón.
- **AbstractProduct (A, B):** Interfaces que definen la estructura de los objetos de cada familias.
- **ConcreteProduct (A, B):** Clases que heredan de *AbstractProduct* con el fin de implementar familias de objetos concretos.
- **ConcreteFactory:** Representan las fábricas concretas que servirán para crear las instancias de todas las clases de la familia. En esta clase debe existir un método para crear cada una de las clases de la familia.
- **AbstractFactory:** Define la estructura de las fábricas y deben proporcionar un método para cada clase de la familia.

Las familias de productos están representadas en la siguiente imagen:

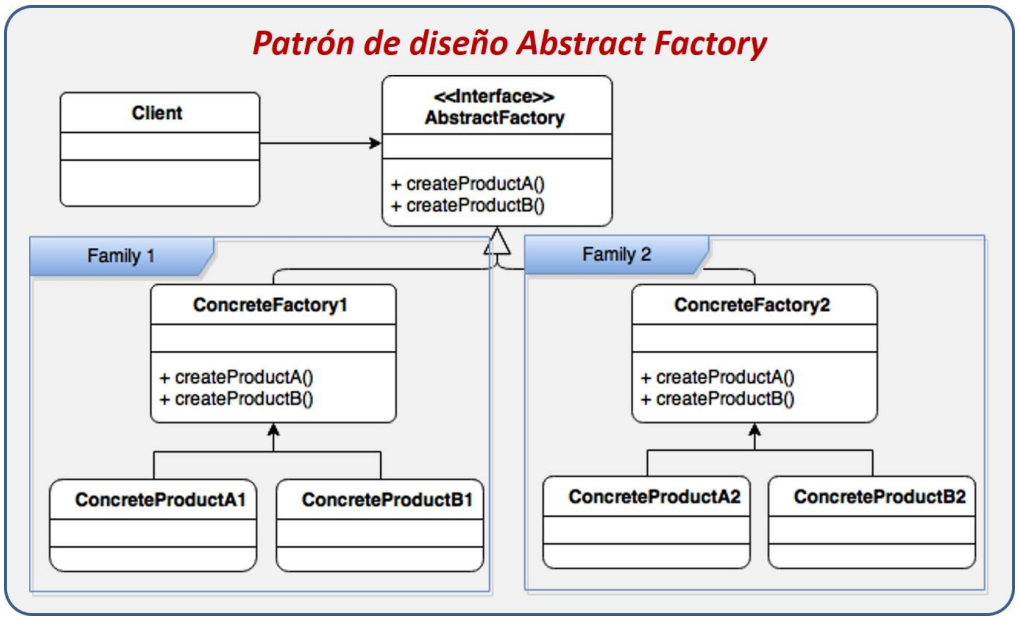


Ilustración 12: Patrón Abstract Factory organizado por familias.

En la imagen anterior podemos apreciar más claramente cuáles son las familias de clases. En la imagen se eliminaron las interfaces de *AbstractProductA* y *AbstractProductB* para que se pudiera apreciar mejor. Podemos ver que cada familia de productos tiene un *ConcreteFactory*, este *factory* nos permitirá crear las instancias de las familias sin conocer realmente las clases concretas.

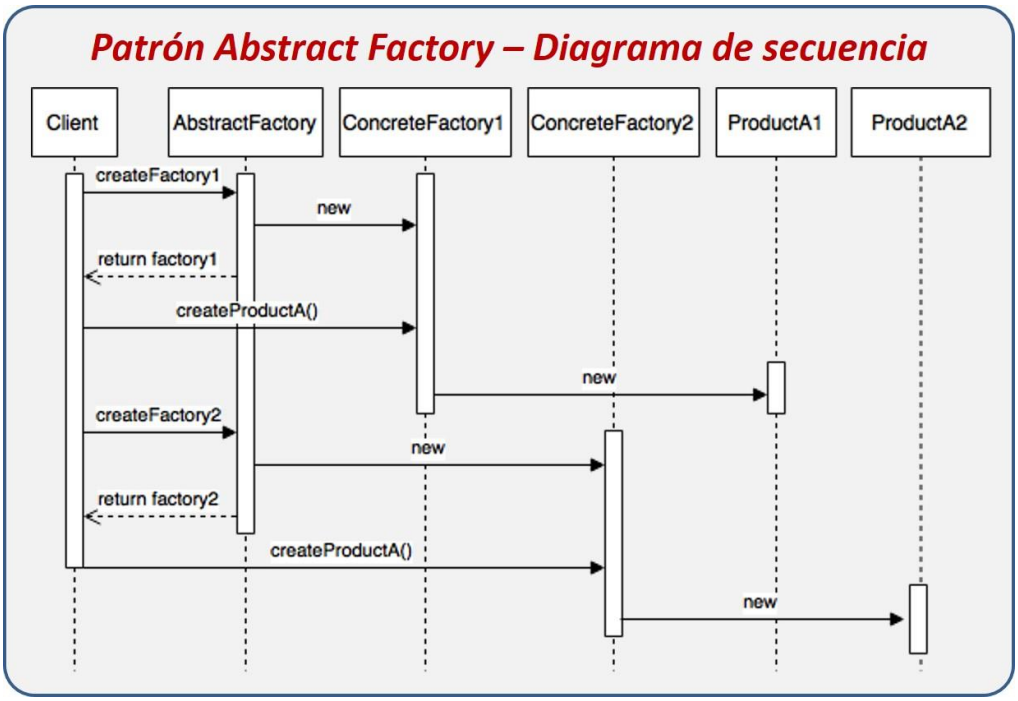


Ilustración 13: Diagrama de secuencia del patrón Abstract Factory.

El diagrama se interpreta de la siguiente manera:

1. El cliente solicita la creación del *ConcreteFactory1* al *AbstractFactory*.
2. El *AbstractFactory* crea una instancia del *ConcreteFactory1* y la regresa.
3. El cliente le solicita al *ConcreteFactory1* la creación de un *ProductA*.
4. El *ConcreteFactory1* crea una instancia del *ProductA1*, el cual es parte de la familia1 y lo regresa.
5. El cliente esta vez solicita la creación del *ConcreteFactory2* al *AbstractFactory*.
6. El *AbstractFactory* crea una instancia del *ConcreteFactory2*.
7. El cliente le solicita al *ConcreteFactory2* la creación de un *ProductA*.
8. El *ConcreteFactory2* crea una instancia del *ProductA2* el cual es parte de la familia2 y lo regresa.

Como vemos en el diagrama, el cliente en ningún momento solicita una clase concreta del *ProductA*, en su lugar, le deja la responsabilidad de crear la instancia correcta al *ConcreteFactory*.

Cuándo utilizarlo:

- Cuando la creación directa de un objeto por medio del operador *new* puede ser perjudicial.
- Cuando no se conoce en tiempo de diseño, la familia de clases que se utilizará en ejecución.
- Cuando es necesario controlar la creación de objetos por medio de una interface común.
- Cuando es necesario trabajar con distintas plataformas, manteniendo la misma estructura y compatibilidad con todas ellas.

El escenario:

Existen ocasiones donde nuestra aplicación necesita soporte para conectarse al *BackEnd* mediante dos estrategias. Esto es utilizado cuando queremos darle la oportunidad a nuestros clientes que eligen qué tipo de conexión les parece mejor; por ejemplo, podemos crear una aplicación que se comunique al sistema *BackEnd* mediante servicios **SOA** o **RestFul**, no obstante, será el cliente quien tome esta decisión, por lo que nuestro sistema debe estar preparado para soportar esta Arquitectura.

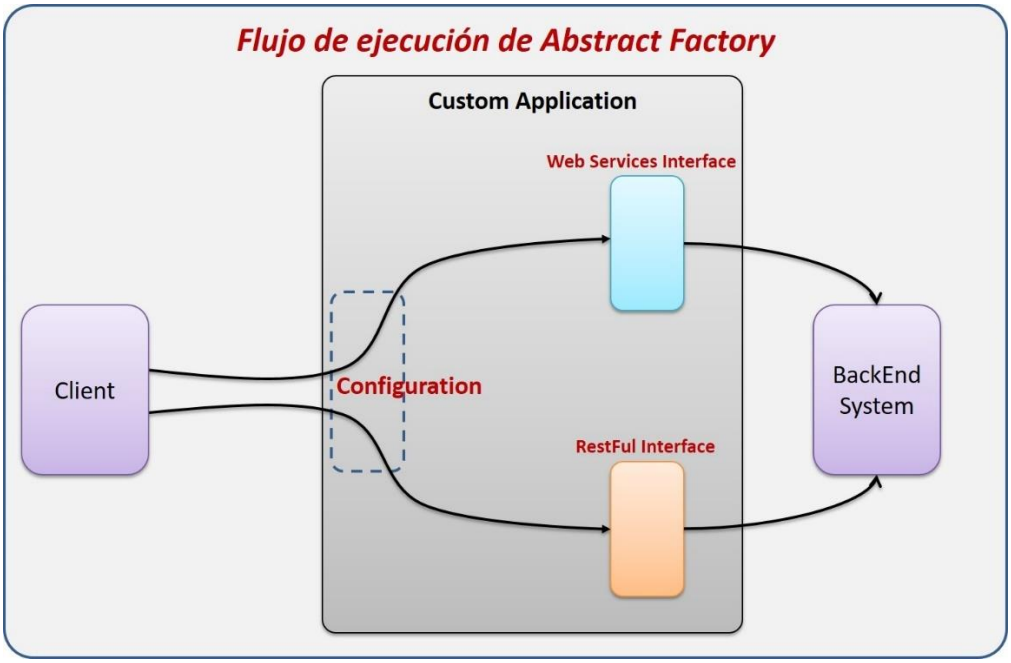


Ilustración 14: Invocación al sistema *BackEnd* por Webservices y RestFul.

En la imagen, el cliente intenta comunicarse al servidor de *BackEnd*, para esto, existen dos escenarios: realizar la invocación por WebService o por servicios RestFul. La aplicación estará preparada mediante configuración para determinar en tiempo de ejecución qué servicio utilizará, lo interesante de la configuración es que se puede mover en cualquier momento y la aplicación seguirá funcionando como si nada hubiera pasado.

La solución:

Para dar solución a este problema, haremos uso del patrón de diseño *Abstract Factory*. Como ya vimos, este patrón nos permite crear familias de clases que cumplan la misma funcionalidad pero con implementación diferente. Por lo cual, crearemos una interface para cada servicio que nos proporcione el sistema *BackEnd*, estas interfaces representarán de forma abstracta los servicios que tendremos disponibles (*AbstractProduct*). Por cada *AbstractProduct* tendremos que crear una implementación para WebService y RestFul con el objetivo de tener la funcionalidad implementada para cada familia.

El siguiente paso será crear una Fábrica que nos permita crear cada uno de los *ConcreteProduct* (deberá existir una fábrica por cada familia).

Finalmente, tendremos un archivo de configuración que nos indicará qué familia de clases vamos a utilizar.

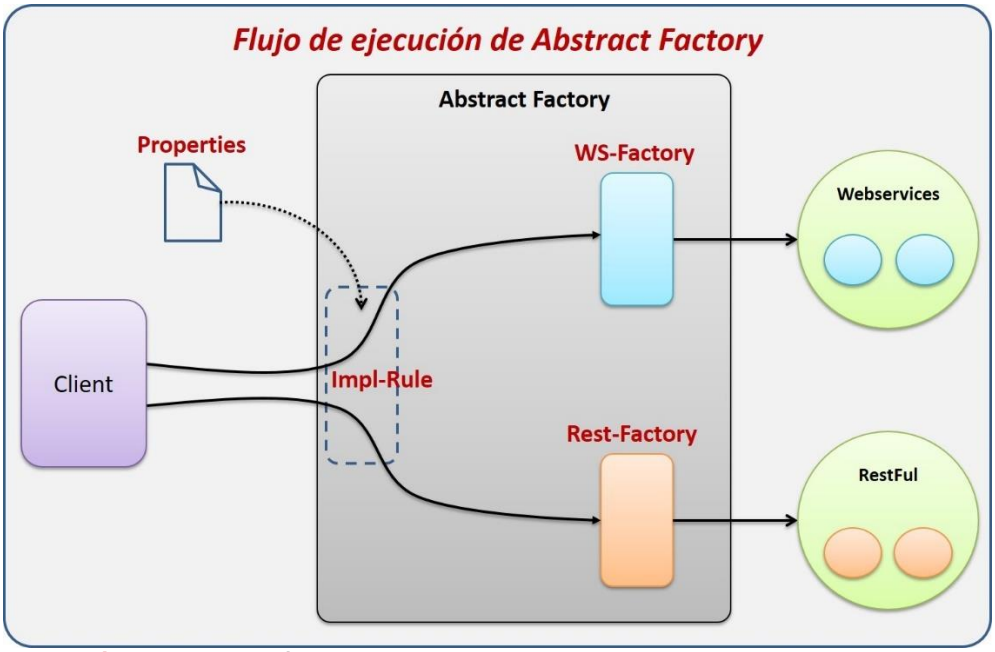


Ilustración 15: Invocación de los servicios de *BackEnd* mediante un AbstractFactory.

En la imagen apreciamos que el cliente desea comunicarse con el sistema *BackEnd*, sin embargo, el cliente no sabe ni le interesa qué familia de productos

se utilizará, ya que el *Abstract Factory* con ayuda del archivo de configuración determinarán la familia de productos a utilizar.
Si el archivo indica que utiliza la familia de comunicación por WebService, entonces se ejecutará el primer escenario de la imagen; de otro modo, se ejecutará el escenario por RestFul.

Como éste es sólo un ejemplo de cómo utilizar el patrón, no incluiremos la lógica de conexión a WebService y Restul; en su lugar simularemos que nos conectamos al servicio *BackEnd*.

La implementación:

Con una idea muy clara de lo que hay que hacer, iniciaremos definiendo las clases del programa.

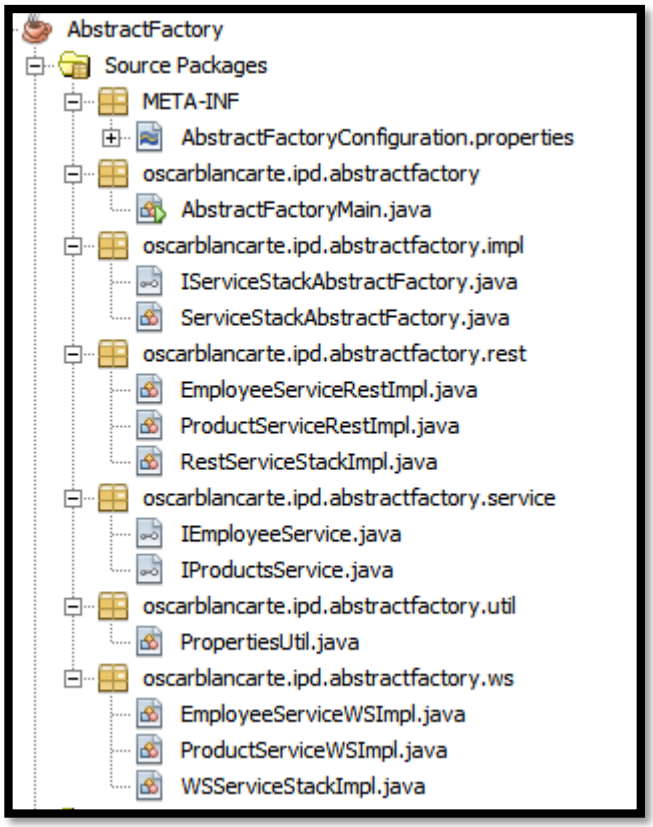


Ilustración 16: Estructura del proyecto Abstract Factory.

El proyecto está compuesto por los siguientes paquetes:

- **oscarblancarte.ipd.abstractfactory**: Paquete principal del proyecto en donde se encuentra la clase ejecutable.
- **oscarblancarte.ipd.abstractfactory.impl:** Paquete con las clases bases para crear los *Abstract Factory*.
- **oscarblancarte.ipd.abstractfactory.rest:** Paquete con la familia de clases para comunicación por *Rest.*

- **oscarblancarte.ipd.abstractfactory.service**: Paquete con las interfaces para crear los servicios. Los web services y los servicios *Rest* tendrán que implementar estas interfaces.
- **oscarblancarte.ipd.abstractfactory.util:** Paquete con clases de utilidad.
- **oscarblancarte.ipd.abstractfactory.ws:** Paquete con la familia de clases para comunicación por web services.
- **META-INF:** Paquete con el archivo de configuración para determinar el tipo de servicios a utilizar.

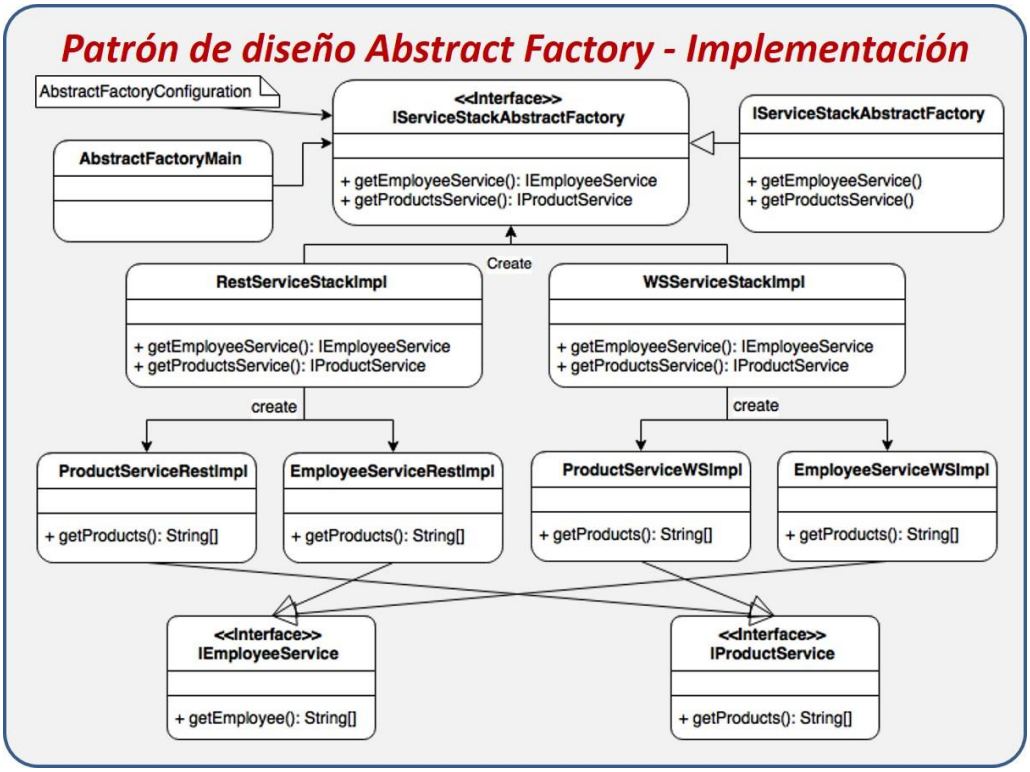


Ilustración 17: Diagrama de clases del proyecto FactoryMethod.

En el diagrama podemos observar las clases que componen el proyecto y cómo es que éstas se relacionan entre sí, en esta imagen podemos ver con más claridad cómo están formadas las familias.

Clase ServiceStackAbstractFactory:

Esta es la implementación del *AbstractFactory* que nos permite obtener la fábrica según lo indicado en el archivo de configuración. El archivo *AbstractFactoryConfiguration.properties* es cargado para leer la fábrica por *default*.

Dentro de la clase creamos el método estático *createServiceFactory* el cual, se encarga de leer el archivo de configuración para determinar la fábrica a crear. La clase que se especifique en el archivo de propiedades deberá implementar la interface *IServiceStackAbstraFactory*.

```
1.  package oscarblancarte.ipd.abstractfactory.impl;
2.
3.  import java.util.Properties;
4.  import oscarblancarte.ipd.abstractfactory.util.PropertiesUtil;
5.
6.  public class ServiceStackAbstractFactory  {
7.
8.      public ServiceStackAbstractFactory() {}
9.
10.         public static IServiceStackAbstractFactory createServiceFactory() {
11.             Properties props = PropertiesUtil.loadProperty(
12.                     "META-INF/AbstractFactoryConfiguration.properties");
13.             String factoryClass =
14.                     props.getProperty("serviceProductImplClass");
15.             try {
16.                 return (IServiceStackAbstractFactory)
17.                         Class.forName(factoryClass).newInstance();
18.             } catch (Exception e) {
19.                 e.printStackTrace();
20.                 return null;
21.             }
22.         }
23.     }
```

Resource AbstractFactoryConfiguration.properties:

Este archivo tiene un formato plano, en él se define las dos clases que utilizaremos como fábricas, una para WebService (***WSServiceStackImpl***) y otra para RestFul (***RestServiceStackImpl***), sin embargo, sólo una podrá estar activa al mismo tiempo.

```
1. serviceProductImplClass oscarblancarte.ipd.abstractfactory.ws.WSServiceStackImpl
2. #serviceProductImplClass oscarblancarte.ipd.abstractfactory.rest.RestServiceStackImpl
```

Interface IServiceStackAbstractFactory:

Interface que define la estructura del *ConcreteAbstractFactory*, en ella se debe definir un método por cada clase que conforma la familia.
En este ejemplo tenemos dos servicios, uno que consulta los empleados y otro que consulta los productos, por ello esta interface tiene dos métodos, que son:

- **getEmployeeService:** Servicio para consultar los empleados.
- **getProductService:** Servicio para consultar los productos.

Como veremos más adelante, los dos métodos regresan interfaces y no definen clases concretas ya que esta interface deberá ser implementada por cada familia de clases y éstas regresarán las clases concretas.

```
1. package oscarblancarte.ipd.abstractfactory.impl;
2.
3. import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
4. import oscarblancarte.ipd.abstractfactory.service.IProductsService;
5.
6. public interface IServiceStackAbstractFactory {
7.     public IEmployeeService getEmployeeService();
8.     public IProductsService getProductsService();
9. }
```

Clase RestServiceStackImpl:

Esta es la primera fábrica concreta que tenemos para crear todos los objetos de la familia de servicios RestFul. Esta clase implementa *IServiceStackAbstractFactory* por lo que debe de proveer los métodos *getEmployeeService* y *getProductService*.

```
1. package oscarblancarte.ipd.abstractfactory.rest;
2.
3. import oscarblancarte.ipd.abstractfactory.impl.IServiceStackAbstractFactory;
4. import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
```

```
5. import oscarblancarte.ipd.abstractfactory.service.IProductsService;
6.
7. public class RestServiceStackImpl implements IServiceStackAbstractFactory{
8.
9.     @Override
10.         public IEmployeeService getEmployeeService() {
11.             return new EmployeeServiceRestImpl();
12.         }
13.
14.         @Override
15.         public IProductsService getProductsService() {
16.             return new ProductServiceRestImpl();
17.         }
18.     }
```

En la clase sólo debemos crear objetos de la familia RestFul, sin embargo, el consumidor no notará la diferencia ya que las implementaciones van encapsuladas tras las interfaces *IProductService* e *IEmployeeService*.

Clase WSServiceStackImpl:

Esta clase representa la fábrica de objetos de la familia de servicios por WebService, esta fábrica deberá crear únicamente objetos que sirvan para conectarnos al sistema de ***BackEnd*** mediante WebService.

```
1. package oscarblancarte.ipd.abstractfactory.ws;
2.
3. import oscarblancarte.ipd.abstractfactory.impl.IServiceStackAbstractFactory;
4. import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
5. import oscarblancarte.ipd.abstractfactory.service.IProductsService;
6.
7. public class WSServiceStackImpl implements IServiceStackAbstractFactory{
8.
9.     @Override
10.         public IEmployeeService getEmployeeService() {
11.             return new EmployeeServiceWSImpl();
12.         }
13.
14.         @Override
15.         public IProductsService getProductsService() {
16.             return new ProductServiceWSImpl();
17.         }
18.     }
```

La clase implementa *IServiceStackAbstractFacotry*, lo que le obliga a implementar los métodos *getEmployeeService* y *getProductsService*, los cuales regresan instancias de la familia de servicios por WebService.

Interface IEmployeeService:

Como lo hablamos hace un momento, las fábricas deberán crear objetos para consumir los servicios del sistema *BackEnd*, esta interface define el primer servicio y es utilizado para consultar el listado de empleados.

```
1. package oscarblancarte.ipd.abstractfactory.service;
2.
3. public interface IEmployeeService {
4.     public String[] getEmployee();
5. }
```

Esta interface deberá ser implementada dos veces, una para consumir el servicio por *WebService* y otro por *RestFul.*

Interface IProductService:

Esta interface también representa un servicio del sistema *BackEnd***,** en este caso el servicio nos servirá para consumir los productos del sistema *BackEnd*.

```
1. package oscarblancarte.ipd.abstractfactory.service;
2.
3. public interface IProductsService {
4.     public String[] getProducts();
5. }
```

Esta interface también tendrá que ser implementada dos veces, una para consumir el servicio por *Rest* y el otro por *WebService*.

Clase ProductServiceRestImpl:

Esta clase es la primera implementación del servicio de consulta de productos, la clase hereda de *IProductService* e implementa el método *getProducts.*

```
1. package oscarblancarte.ipd.abstractfactory.rest;
2.
3. import oscarblancarte.ipd.abstractfactory.service.IProductsService;
4.
5. public class ProductServiceRestImpl implements IProductsService{
6.     private static final String[] PRODUCTS = new String[]{"Teclado","Mouse", "Monitor"};
7.
8.     @Override
9.     public String[] getProducts() {
10.            System.out.println("RestFul");
11.            return PRODUCTS;
12.        }
13.    }
```

En esta clase estamos simulando que invocamos a un *RestFul* y que nos regresa una lista de productos.

Clase EmployeeServiceRestImpl:

Esta clase nos servirá para consultar los empleados desde el sistema *BackEnd* mediante un servicio *RestFul*. La clase hereda de *IEmployeeService* e implementa el método *getEmployee* el cual simulará hacer la conexión al sistema *BackEnd* para consultar los empleados.

```
1. package oscarblancarte.ipd.abstractfactory.rest;
2.
3. import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
4.
5. ublic class EmployeeServiceRestImpl implements IEmployeeService{
6.     private static final String[] EMPLOYEES = new String[]{"Juan","Pedro", "Manuel"};
7.
8.     @Override
9.     public String[] getEmployee() {
10.            System.out.println("RestFul");
11.            return EMPLOYEES;
12.        }
13.    }
```

Clase ProductServiceWSImpl:

Clase perteneciente a la familia de WebService, simula invocar a un WebService para consultar la lista de productos, la case implementa *IProductService* por lo que debe implementar el método getProducts.

```
1. package oscarblancarte.ipd.abstractfactory.ws;
2.
3. import oscarblancarte.ipd.abstractfactory.service.IProductsService;
4.
5. public class ProductServiceWSImpl implements IProductsService {
6.
7.     private static final String[] PRODUCTS = new String[]{"Refresco", "Jugo", "Fruta"};
8.
9.     public String[] getProducts() {
10.            System.out.println("WebServices");
11.            return PRODUCTS;
12.        }
13.    }
```

Clase EmployeeServiceWSImpl:

Esta clase pertenece a la familia de WebService, la cual simula la consulta de empleados, la clase implementa la interface *IEmployeeService* y define el método *getEmployee*.

```
1. package oscarblancarte.ipd.abstractfactory.ws;
2.
3. import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
4.
5. public class EmployeeServiceWSImpl implements IEmployeeService {
6.
7.     private static final String[] EMPLOYEES = new String[]{"Maria", "Rebeca", "Liliana"};
8.
9.     @Override
10.        public String[] getEmployee() {
11.            System.out.println("WebServices");
12.            return EMPLOYEES;
13.        }
14.
15.    }
```

Clase AbstractFactoryMain:

Clase ejecutable que utilizaremos para probar nuestro patrón de diseño.

```
1.  package oscarblancarte.ipd.abstractfactory;
2.
3.  import java.util.Arrays;
4.  import oscarblancarte.ipd.abstractfactory.impl.IServiceStackAbstractFactory;
5.  import oscarblancarte.ipd.abstractfactory.impl.ServiceStackAbstractFactory;
6.  import oscarblancarte.ipd.abstractfactory.service.IEmployeeService;
7.  import oscarblancarte.ipd.abstractfactory.service.IProductsService;
8.
9.  public class AbstractFactoryMain {
10.
11.         public static void main(String[] args) {
12.             IServiceStackAbstractFactory factory =
13.                     ServiceStackAbstractFactory.createServiceFactory();
14.             IEmployeeService employeeService = factory.getEmployeeService();
15.             IProductsService productService = factory.getProductsService();
16.
17.             System.out.println("EmployeeService class > "
18.                     + employeeService.getClass().getCanonicalName());
19.             System.out.println("ProductService class  > "
20.                     + productService.getClass().getCanonicalName());
21.
22.             System.out.println("getEmployee > "
23.                     + Arrays.toString(employeeService.getEmployee()));
24.             System.out.println("getProduct  > "
25.                     + Arrays.toString(productService.getProducts()));
26.         }
27.     }
```

Lo primero que hacemos es crear una instancia de nuestro *AbstractFactory* en la línea 12, internamente éste define qué clase será la fábrica por *default* mediante el archivo de propiedades explicado anteriormente, en las líneas 14 y 15 se obtienen los servicios, en este punto el cliente no es capaz de diferenciar qué familia de clases está utilizando debido a que son encapsulados tras las interfaces.

Para fines de prueba, imprimimos el nombre de las clases que regresó el *factory* en las líneas 17 y 19, éstas deberán estar relacionadas con la familia de clases que fue creada por el *AbstractFactory*.

Finalmente, imprimimos el resultado obtenido de la ejecución de los servicios. El resultado variará según la implementación para comprobar que las clases ejecutadas son distintas.

La ejecución:

Con el fin de demostrar cómo es posible cambiar de una familia de servicios a otra, realizaremos dos ejecuciones:

La primera será con la familia de WebService, por lo cual, nos asegurémonos de tener el archivo *AbstractFactoryConfiguration.properties* de la siguiente manera:

```
1. serviceProductImplClass oscarblancarte.ipd.abstractfactory.ws.WSServiceStackImpl
2. #serviceProductImplClass oscarblancarte.ipd.abstractfactory.rest.RestServiceStackImpl
```

Seguido, ejecutaremos la clase ***AbstractFactoryMain*** para ver el siguiente resultado:

```
run:
EmployeeService class > oscarblancarte.ipd.abstractfactory.ws.EmployeeServiceWSImpl
ProductService class  > oscarblancarte.ipd.abstractfactory.ws.ProductServiceWSImpl
WebServices
getEmployee > [Maria, Rebeca, Liliana]
WebServices
getProduct  > [Refresco, Jugo, Fruta]
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 18: Resultado de la ejecución con la familia de Webservices

Los primero que vemos en la salida son los nombre de las clases que el *ConcreteFactory* ha creado, en este caso son las dos clases de la familia de WebService *EmployeeServiceWSImpl* y *ProductServiceWSImpl*.

Las siguientes líneas indican que fueron ejecuciones por WebService e imprimen la lista de empleados y productos retornados.

La segunda ejecución la realizaremos con la familia de RestFul, por lo cual nos asegurémonos de tener el archivo de propiedades de la siguiente manera

```
1. #serviceProductImplClass oscarblancarte.ipd.abstractfactory.ws.WSServiceStackImpl
2. serviceProductImplClass oscarblancarte.ipd.abstractfactory.rest.RestServiceStackImpl
```

Seguido, ejecutaremos la clase *AbstractFactoryMain* para ver el siguiente resultado:

```
run:
EmployeeService class > oscarblancarte.ipd.abstractfactory.rest.EmployeeServiceRestImpl
ProductService class  > oscarblancarte.ipd.abstractfactory.rest.ProductServiceRestImpl
RestFul
getEmployee > [Juan, Pedro, Manuel]
RestFul
getProduct  > [Teclado, Mouse, Monitor]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 19: Resultado de ejecución con la familia Restful.

El resultado es muy similar al anterior; sin embargo, la familia de clases retornadas es distinta, en este caso las clases retornadas por el *ConcreteFactory* son *EmployeeServiceRestImpl* y *ProductServiceRestImpl*. También vemos la leyenda *"RestFul"* lo que indica que la ejecución se realizó por esta familia.

Siguientes pasos:

Como ya hemos analizado, el patrón *Abstract Factory* es ideal para agrupar familias de clases que realizan un comportamiento similar, pero con una implementación diferente.

Para fortalecer los conocimientos realicemos los siguientes ejercicios:

1. Modifiquemos los servicios de Restful y WebService para que se conecten a servicios reales.

2. Agreguemos una tercera familia que se comunique por http al servidor, éste podría mandar la información en el formato que más nos agrede.

3. Agregar una cuarta familia para que se comunique por *Socket* al servidor.

4. Agreguemos un servicio extra al *BackEnd*, por ejemplo, la consulta de usuarios del sistema, para después, implementarlo en el *AbstractFactory* para ser consultado por WebService y Restful.

Patrón Singleton

El patrón de diseño ***Singleton*** (soltero) recibe su nombre debido a que sólo se puede tener una única instancia para toda la aplicación de una determinada clase, esto se logra restringiendo la libre creación de instancias de esta clase mediante el operador new e imponiendo un constructor **privado** y un **método estático** para poder obtener la instancia.
La intención de este patrón es garantizar que solamente pueda existir una única instancia de una determinada clase y que exista una referencia global en toda la aplicación.

Singleton es el patrón más pequeño en cuanto a número de clases requeridas para ser implementado, debido a que sólo requiere de la clase ***Singleton*** en sí; sin embargo, esta clase debe tener ciertas características para que pueda ser considerada un ***Singleton***, una de ellas es que debe tener un solo constructor privado, el cual impida la creación directa de instancias. Debe tener una referencia *static* de sí misma con el fin de almacenar de forma global la única instancia que se creará.

De inicio esta referencia sería *null*, por lo que se deberá crear un método *public* y *static* que permita obtener una instancia de la clase, este método se asegurará de crear la instancia sólo en caso de que no exista una referencia creada anteriormente, en caso de existir deberá regresarla.

Existen escenarios, sobre todo en aplicaciones que trabajan con Multi-Hilos, en los que una creación convencional puede ser un problema, ya que si dos hilos intentan crear una instancia al mismo tiempo ambos crearán una instancia y se rompería el propósito de este patrón, por lo que sería necesario hacer un método privado encargado únicamente de crear la instancia; el cual esté protegido de la concurrencia como podría ser agregar el modificador de acceso synchronized evitando de esta manera que los dos hilos puedan crear la instancia al mismo tiempo.

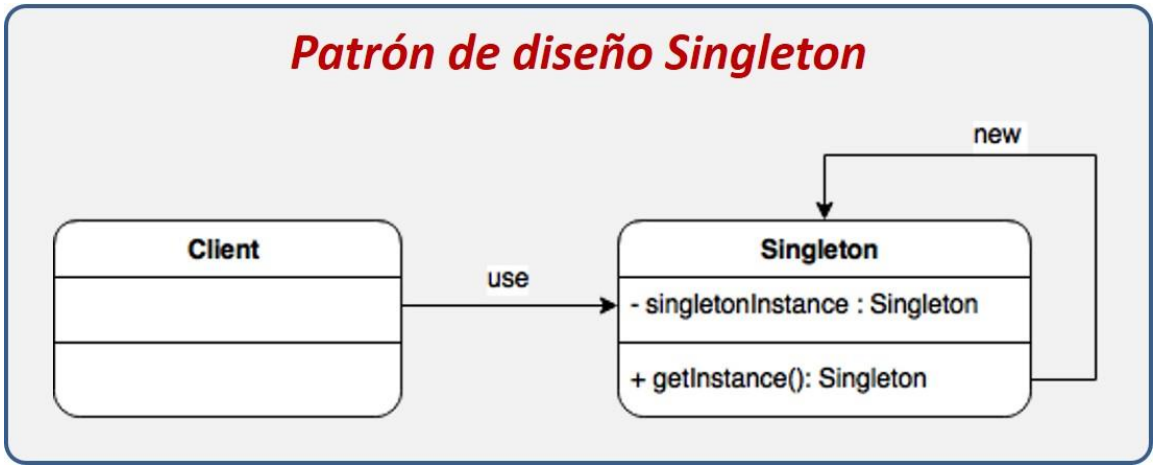


Ilustración 20: Estructura del patrón de diseño Singleton.

Los componentes que conforman el patrón son los siguientes:

- **Client:** Componente que desea obtener una instancia de la clase *Singleton*.
- **Singleton:** Clase que implementa el patrón *Singleton*, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación.

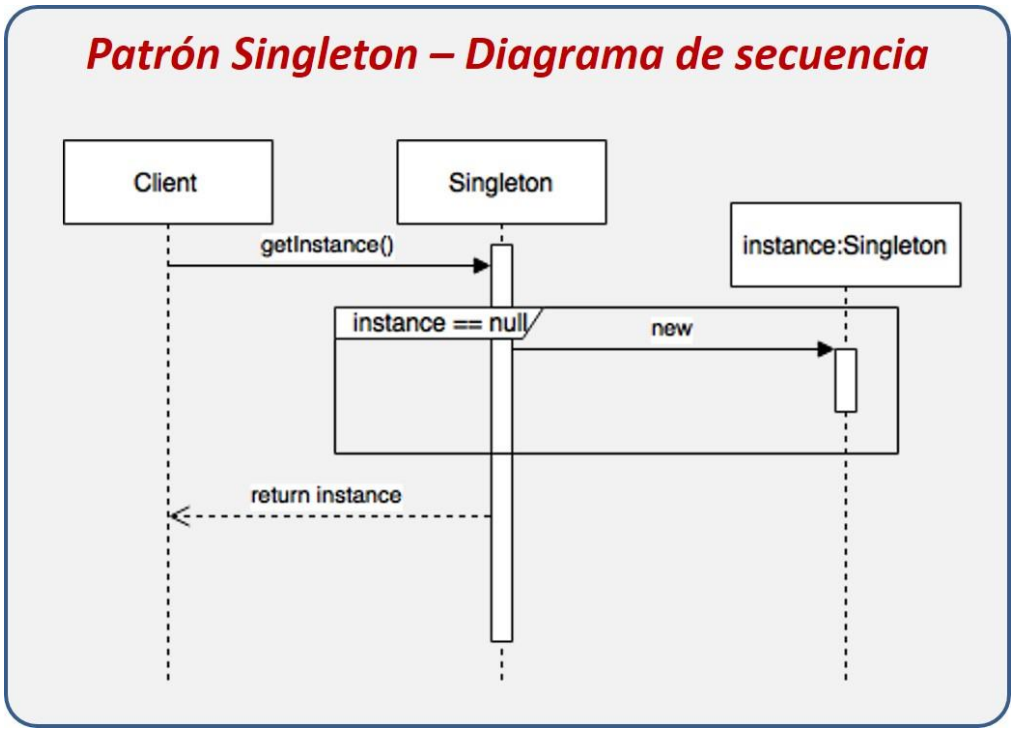


Ilustración 21: Diagrama de secuencia del patrón Singleton.

El diagrama se interpreta de la siguiente manera:

1. El cliente solicita la instancia al *Singleton* mediante el método estático getInstance.
2. El *Singleton* validará si la instancia ya fue creada anteriormente, de no haber sido creada entonces se crea una nueva.
3. Se regresa la instancia creada en el paso anterior o se regresa la instancia existente.

Como podemos ver en el diagrama, sólo una instancia será creada, cuando el cliente solicite la creación, éste regresará la instancia que fue creada anteriormente y en caso de no existir creará la única instancia que existirá en la aplicación.

Cuándo utilizarlo:

- Cuando es requerido tener una única instancia en toda la aplicación y se requiere un acceso global a la misma.

- Cuando la creación de objetos por medio del operador new es perjudicial.

El escenario:

Para comprender mejor cómo es que funciona este patrón, realizaremos un ejemplo simple. Se ha solicitado crear una aplicación que, como parte de su proceso de inicio, cargue la configuración guardada en un archivo de propiedades, esta configuración deberá estar disponible para toda la aplicación incluyendo los módulos que ésta podría tener. Dicha configuración tiene un requerimiento especial y es que deberá existir un solo objeto de configuración en toda la aplicación, en la cual, todos los módulos puedan interactuar, de tal manera que si un módulo afecta alguna propiedad de la configuración el resto del módulo lo pueda ver reflejado.

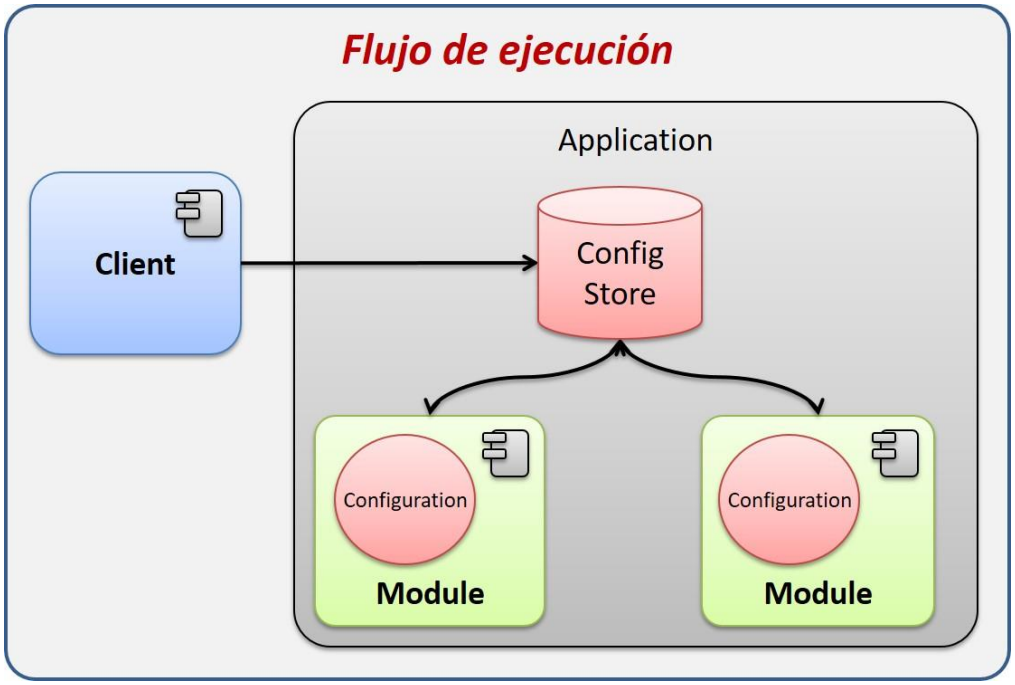


Ilustración 22: Flujo de ejecución sin el patrón Singleton.

En la imagen podemos apreciar un escenario donde cada módulo de la aplicación lee directamente la configuración, obteniendo con esto, que cada uno tenga un objeto de configuración distinto, si uno de los módulos afecta la configuración, el resto de los módulos no será capaz de verlo; obteniendo con esto un grave problema, de igual forma si uno decide guardar la configuración sobrescribirá la configuración que otro módulo ha guardado.

La solución:

Para resolver el problema de la configuración, utilizaremos el patrón de diseño ***Singleton,*** el cual nos permitirá controlar la forma en la que creamos un objeto de configuración, el patrón de diseño limitará la creación de este objeto a una única instancia para toda la aplicación; de esta manera, cuando un módulo quiera obtener el objeto de configuración, éste recibirá la misma instancia que el resto de módulos.
Todos los módulos podrán ver la misma información en el objeto y de la misma manera, cuando alguien realice un cambio sobre ésta, serán visibles para todos los módulos.

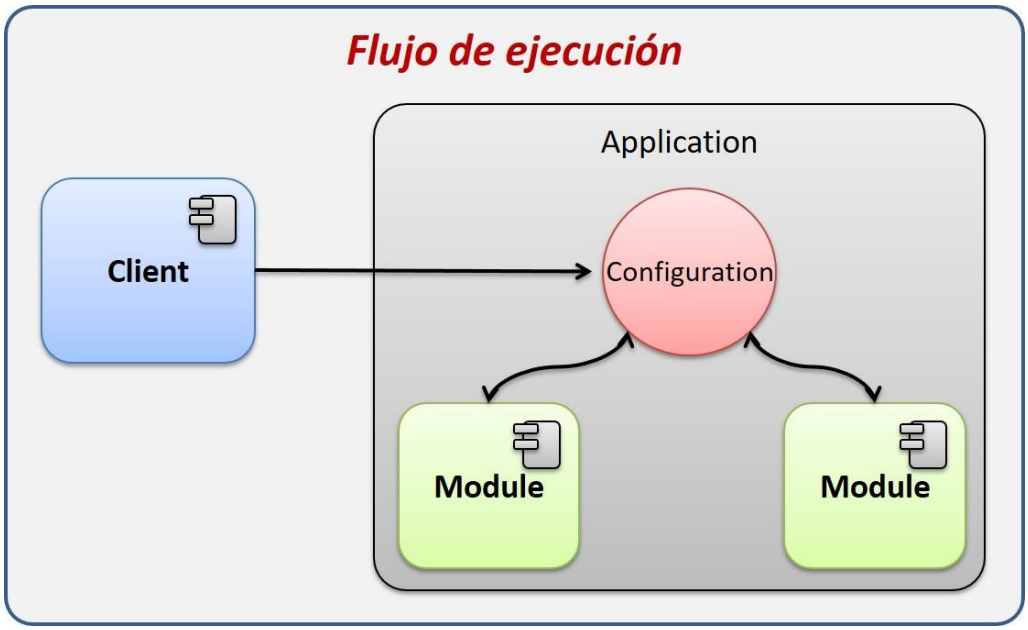


Ilustración 23: Flujo de ejecución utilizando el patrón Singleton.

En la imagen podemos ver cómo se implementaría la aplicación utilizando el patrón de diseño *Singleton*, vemos que el objeto de configuración sale de los módulos y se posiciona a nivel de aplicación como un objeto único, también vemos que los módulos se comunican con el mismo objeto de configuración por lo que si hay algún cambio en éste (incluso de forma externa como lo hace el cliente) todos los módulos verán la misma configuración.

La implementación:

Una vez analizado el escenario iniciaremos con su construcción.

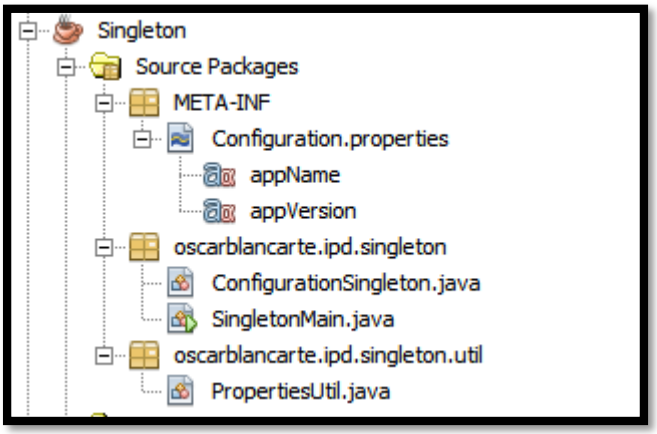


Ilustración 24: Estructura del proyecto Singleton.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.singleton:** Paquete principal de la aplicación donde se encuentra la case *Singleton* y la clase principal del proyecto.
- **oscarblancarte.ipd.singleton.util**: Paquete con clases de utilidad.

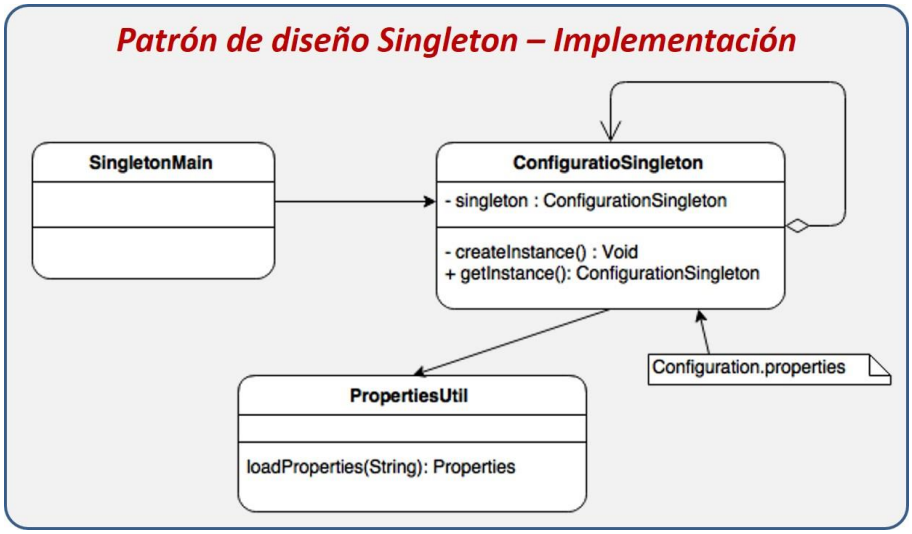


Ilustración 25: Diagrama de clases del proyecto Singleton.

En la imagen podemos apreciar las clases que componen el proyecto *Singleton* y cómo es que se relacionan entre sí.

Clase ConfigurationSingleton:

Esta clase representa al *Singleton* y podemos ver que tiene una referencia a sí misma, mediante una variable estática, en ella, almacenaremos la referencia de la única instancia que se creará durante la vida de la aplicación.

La clase cuenta con las siguientes variables:

- **CONFIGURATION_PROP:** URL en donde se encuentra el archivo de propiedades de la aplicación.
- **APP_NAME_PROP:** Nombre de la propiedad para obtener este valor del nombre del archivo de propiedades.
- **APP_VERSION_APP:** Nombre de la propiedad para obtener este valor de la versión del archivo de propiedades.
- **APPNAME:** Variable donde se almacenará el nombre de la aplicación.

Los métodos definidos son:

- **createInstance:** Permite la creación de la única instancia de la aplicación. Este método creará la instancia sólo en el caso de que la variable Singleton sea nula.
- **getInstance:** Método estático que permite obtener la instancia *Singleton*, validando previamente si existe, en otro caso la crea y la regresa.

Finalmente, en el constructor de la clase cargamos la configuración almacenada en el archivo de propiedades con ayuda de la clase de utilidad *PropertiesUtil*. El constructor se ejecutará una sola vez durante toda la vida de la aplicación.

```
1. package oscarblancarte.ipd.singleton;
2.
3. import java.util.Properties;
4. import oscarblancarte.ipd.singleton.util.PropertiesUtil;
5.
```

```
6.  public class ConfigurationSingleton {
7.
8.     private static ConfigurationSingleton singleton;
9.
10.        private static final String CONFIGURATION_PROP = "META-INF/Configuration.properties";
11.
12.        private static final String APP_NAME_PROP = "appName";
13.        private static final String APP_VERSION_PROP = "appVersion";
14.
15.        private String appName;
16.        private String appVersion;
17.
18.        private ConfigurationSingleton() {
19.           Properties prop = PropertiesUtil.loadProperty(CONFIGURATION_PROP);
20.           this.appName = prop.getProperty(APP_NAME_PROP);
21.           this.appVersion = prop.getProperty(APP_VERSION_PROP);
22.        }
23.
24.        private static synchronized void createInstance(){
25.           if(singleton ==null){
26.              singleton = new ConfigurationSingleton();
27.           }
28.        }
29.
30.        public static ConfigurationSingleton getInstance() {
31.           if(singleton == null) {
32.              createInstance();
33.           }
34.           return singleton;
35.        }
36.
37.        */ GET AND SETS */
38.     }
```

Configuration.properties:

Archivo de propiedades utilizado para cargar la configuración inicial de la aplicación.

```
1.  appName App name dummy
2.  appVersion 10.2.2.x
```

Clase PropertiesUtil:

Esta clase ya la hemos analizado anteriormente, sobra decir que sirve únicamente para cargar un archivo de propiedades.

Clase SingletonMain:

Para concluir, creamos la clase *SingletonMain***,** la cual utilizaremos para probar el funcionamiento de nuestro patrón. Para el ejemplo, no crearemos objetos que representen los módulos, en su lugar, el cliente intentará crear dos instancias del *Singleton* y veremos qué es lo que sucede.

```
1.  package oscarblancarte.ipd.singleton;
2.
3.  public class SingletonMain {
4.
5.     public static void main(String[] args) {
6.        ConfigurationSingleton singletonA = ConfigurationSingleton.getInstance();
7.        ConfigurationSingleton singletonB = ConfigurationSingleton.getInstance();
8.
9.        System.out.println(singletonA);
10.            System.out.println(singletonB);
11.            System.out.println("Misma referencia ==> " + (singletonA == singletonB));
12.
13.            singletonA.setAppName("Singleton Pattern");
14.            singletonB.setAppVersion("1.0x");
15.
16.            System.out.println("SingletonA ==> " + singletonA);
17.            System.out.println("SingletonB ==> " + singletonB);
18.
19.            singletonA = null;
20.            singletonB = null;
21.
22.            singletonA = ConfigurationSingleton.getInstance();
23.            System.out.println("SingletonA ==> " + singletonA);
24.         }
25.      }
```

En la línea 6, la instancia *Singleton* no habrá sido creada, por lo que el método *getInstance* solicitará a *createInstance* la creación de la instancia, para finalmente regresarla. Cuando lleguemos a la línea 7, la instancia *Singleton* ya habrá sido creada, pero en lugar de crear una nueva instancia se retornará la que fue previamente creada.

En la línea 11 realizamos una validación para determinar si la variable *singletonA* y *singletonB* apuntan a la misma referencia, esta condición deberá regresar siempre *true*, de lo contrario significa que algo estamos haciendo mal y que cada invocación a *getInstance* está regresando una instancia diferente.

En la línea 13, modificamos el atributo *appName* de la referencia *singletoA*, en la línea 14, modificamos el atributo *appVersion* de la referencia *singletonB*, en las líneas 16 y 17, imprimimos los atributos de *singletonA* y *singletonB* para comprobar que, a pesar de modificar los valores en diferentes variables; los valores son iguales, debido a que ambos apuntan a la misma instancia.

Finalmente, pasamos las referencias de *singletonA* y *singletonB* a null en las líneas 19 y 20, para posteriormente volver a solicitar la instancia e imprimir sus valores (líneas 22 y 23). Veremos que a pesar de haber perdido la referencia a la instancia, los valores permanecen tal cual los habíamos dejado, esto debido a que *Singleton* se asegura que se cree una sola instancia y ésta permanezca activa durante toda la vida de la aplicación.

La Ejecución

Para comprobar todo lo que explicamos anteriormente, ejecutaremos la clase *SingletonMain* con el fin de ver los resultados que arroja.

```
run:
ConfigurationSingleton{appName=App name dummy, appVersion=10.2.2.x}
ConfigurationSingleton{appName=App name dummy, appVersion=10.2.2.x}
Misma referencia ==> true
SingletonA ==> ConfigurationSingleton{appName=Singleton Pattern, appVersion=1.0x}
SingletonB ==> ConfigurationSingleton{appName=Singleton Pattern, appVersion=1.0x}
AppName A ==> ConfigurationSingleton{appName=Singleton Pattern, appVersion=1.0x}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 26: Resultado de la ejecución del proyecto Singleton.

Lo primero que veremos será el *toString* de las variables *singletonA* y *singletonB* para darnos cuenta que ambas son idénticas en valores, sin embargo, esto no demuestra que las dos variables referencien al mismo objeto, es por eso que, en la siguiente línea, realizamos una validación para asegurarnos, el resultado se muestra como *"Misma referencia ==> true".* Con esto ya estamos seguros que en realidad las dos variables referencian al mismo objeto.

El siguiente paso es modificar las propiedades de cada variable, en la variable *singletonA* cambiamos el nombre y en *singletonB* cambiamos la versión, volvemos a imprimir el contenido de las variables mediante el método *toString* y el resultado es que ambas siguen teniendo los mismos valores.

Finalmente, perdemos la referencia al objeto asignando a *null* las dos variables y reasignamos un valor mediante el método *getInstance,* he imprimimos sus propiedades, veamos que siguen estando tal cual las dejamos y esto es porque el *Singleton* permite mantener una única instancia global durante toda la vida de la aplicación.

Siguientes pasos:

Hemos visto cómo el patrón *Singleton* nos puede ayudar para controlar la creación de las instancias, limitando a una sola instancia por clase.

En el ejemplo vimos cómo el patrón fue utilizado para administrar la configuración, pero lo podemos utilizar para otro tipo de propósito como: leer archivos de sólo lectura, asegurarnos de cargar sólo una vez alguna información y tenerla disponible de forma global o para algún otro problema en el que necesitemos que únicamente se cree una instancia de una clase.
Para reforzar los conocimientos adquiridos, podríamos realizar las siguientes actividades:

- Desarrollemos una aplicación que cree varios hilos, cada hilo deberá obtener la referencia al *Singleton* y modificarla, cuando todos los hilos terminen, comprobemos cómo quedaron los atributos de la configuración.
- Modifiquemos la aplicación para que cargue la información desde una base de datos, a la vez, cuando una propiedad sea actualizada, esta deberá reflejarse en la base de datos para que sea cargada cuandoreiniciemos la aplicación.
- Agreguemos mas propiedades a la configuración, las cuales deberán ser accesibles desde cualquier parte de la aplicación.

Patrón Builder

Este es un patrón bastante simple, pero muy útil, que permite crear objetos complejos a través de uno más simple. Es muy común encontrarnos con situaciones en las cuales tenemos que crear objetos compuestos de forma manual y repetidas veces, lo que nos lleva a tener que establecer cada propiedad del objeto, y si este, además tiene objetos compuestos dentro, tendremos que crearlos primero para después ser asignados al objeto que estamos construyendo. Esto desde luego que se hace una tarea tediosa y cansada, sobre todo cuando tenemos que crear objetos de manera frecuente.

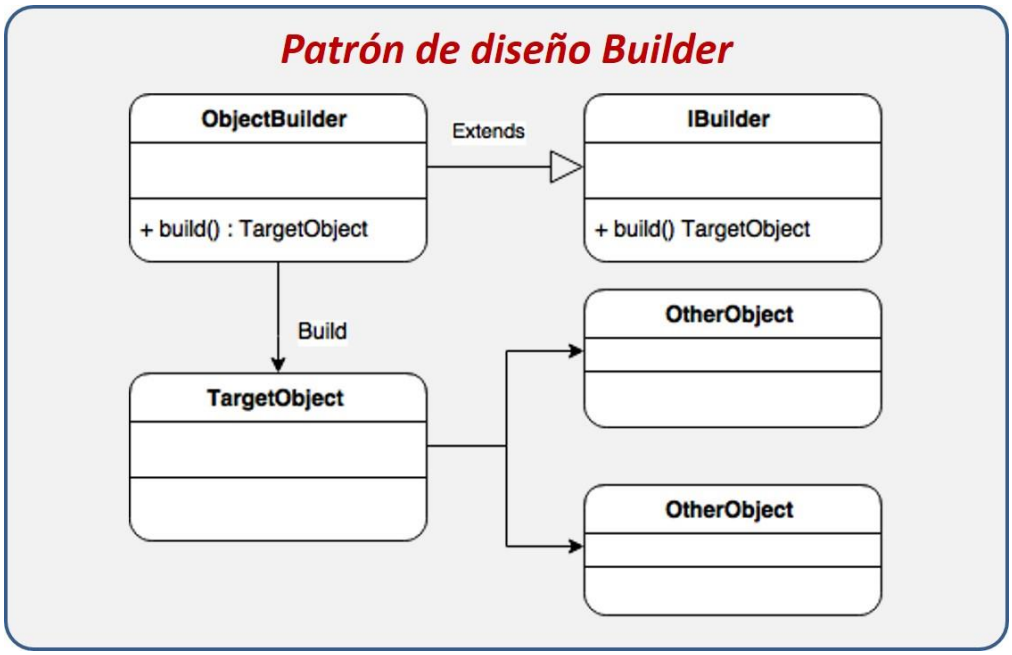


Ilustración 27: Estructura del patrón de diseño Builder.

Los componentes que conforman el patrón son los siguientes:

- **IBuilder:** Este componente no es obligatorio en todos los casos, sin embargo, es buena práctica especificar una interface común que tendrán todos los *Builder* que definiremos en nuestra aplicación, puede ser una interface que defina únicamente el método *build*.

- **ObjectBuilder:** Esta es la clase que utilizaremos para crear los *TarjetObjet.* Esta clase debe de heredar de *IBuilder* e implementar el método *build*, el cual será utilizado para crear al *TarjetObject*. Como regla general todos los métodos de esta clase retornan una instancia de sí mismo, con la finalidad de agilizar la creación. Esta clase por lo general es creada como una clase interna del *TargetObject*.
- **TarjetObjet:** Representa el objeto que deseamos crear con ayuda del *ObjectBuilder*, ésta puede ser una clase simple o puede ser una clase muy compleja que tenga dentro más objetos.
- **OtherObjets:** Representa los posibles objetos que deberán ser creados cuando el *TarjetObject* sea construido por el *ObjectBuilder*.

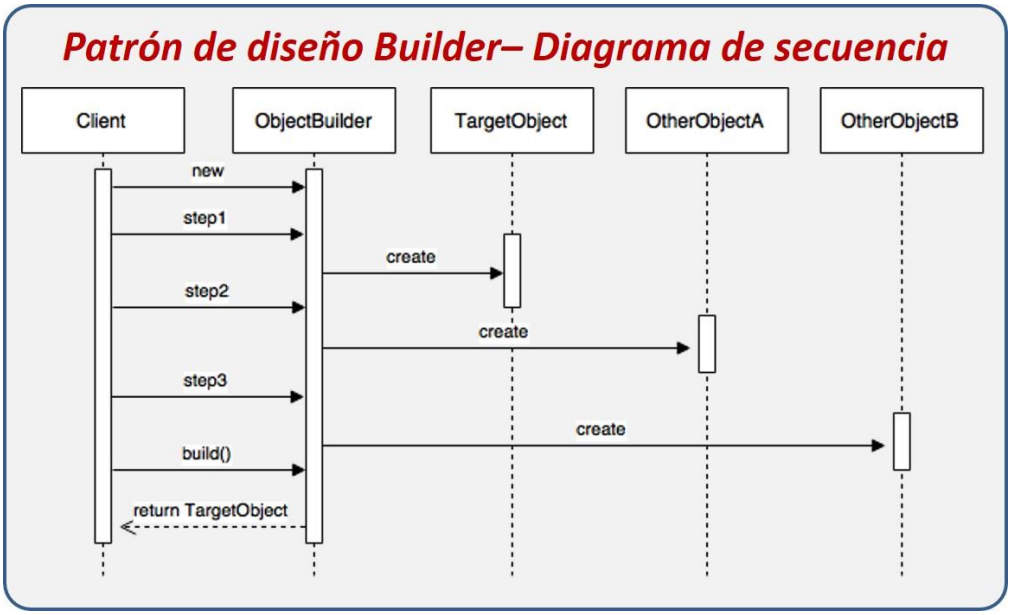


Ilustración 28: Diagrama de secuencia del patrón de diseño Builder.

El diagrama se interpreta de la siguiente manera:

- El cliente crea una instancia del *ObjectBuilder*.
- El cliente ejecuta el paso 1 de la creación en el *ObjectBuilder*.
    - Internamente el *ObjectBuilder* crea al *TargetObject*.
- El cliente ejecuta el paso 2 de la creación en el *ObjectBuilder*.

    - Internamente el *ObjectBuileder* crea un *OtherObjectA*.
- El cliente ejecuta el paso 3 de la creación en el *ObjectBuilder*.
    - Internamente el *ObjectBuilder* crear el *OtherObjectB*.
- El cliente solicita al *ObjectBuilder* la creación del *TargetObject*, éste toma todos los objetos creados anteriormente, los asocia al *TargetObject* y lo regresa.

Utilizarlo cuando:

- Cuando necesitamos un mecanismo simple para crear objetos complejos.

- Cuando necesitamos crear repetidamente objetos complejos.

- Cuando necesitamos ocultar a los usuarios la complejidad de la creación de un objeto determinado.

El escenario:

Para comprender mejor cómo funciona el patrón, realizaremos un ejemplo clásico, donde es necesario crear la estructura completa de un empleado, el cual puede tener una dirección, uno o varios teléfonos y varios contactos; además de los datos básicos como su nombre, edad y género.
La importancia de crear un *Builder* es para tener un mecanismo que nos permita crear un Empleado de forma rápida y limpia, sin tener la necesidad de crear todos los objetos que lo componen, pero también, nos libra de conocer exactamente la lógica de creación, el orden en que éstos deberán ser creados y cómo es que cada objeto se debe asociar a otro.

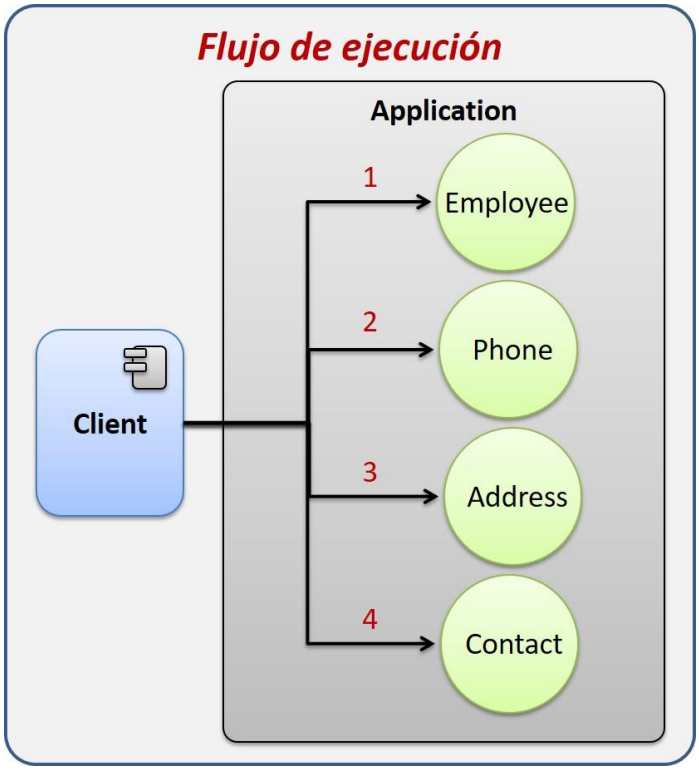


Ilustración 29: Flujo de ejecución sin utilizar el patrón de diseño Builder

En la imagen podemos ver cómo es que se crean normalmente los objetos sin utilizar el patrón de diseño *Builder*. El cliente se encarga de crear individualmente cada objeto en un orden específico, ya que algunos objetos dependen de otros

para su creación, vemos también que los objetos son creados en orden, pero no vemos cómo es que se relacionan con lo demás y cómo es que éstos deberán ser asignados a otros objetos. Toda esta complejidad y el detalle fino de cómo se debe construir un objeto, es lo que nos ayuda a resolver el patrón.

La solución:

Para poder dar una solución al problema planteado anteriormente, implementaremos el patrón de diseño *Builder* que nos ayudará a separar la lógica de construcción de un objeto complejo de la del programa en sí.

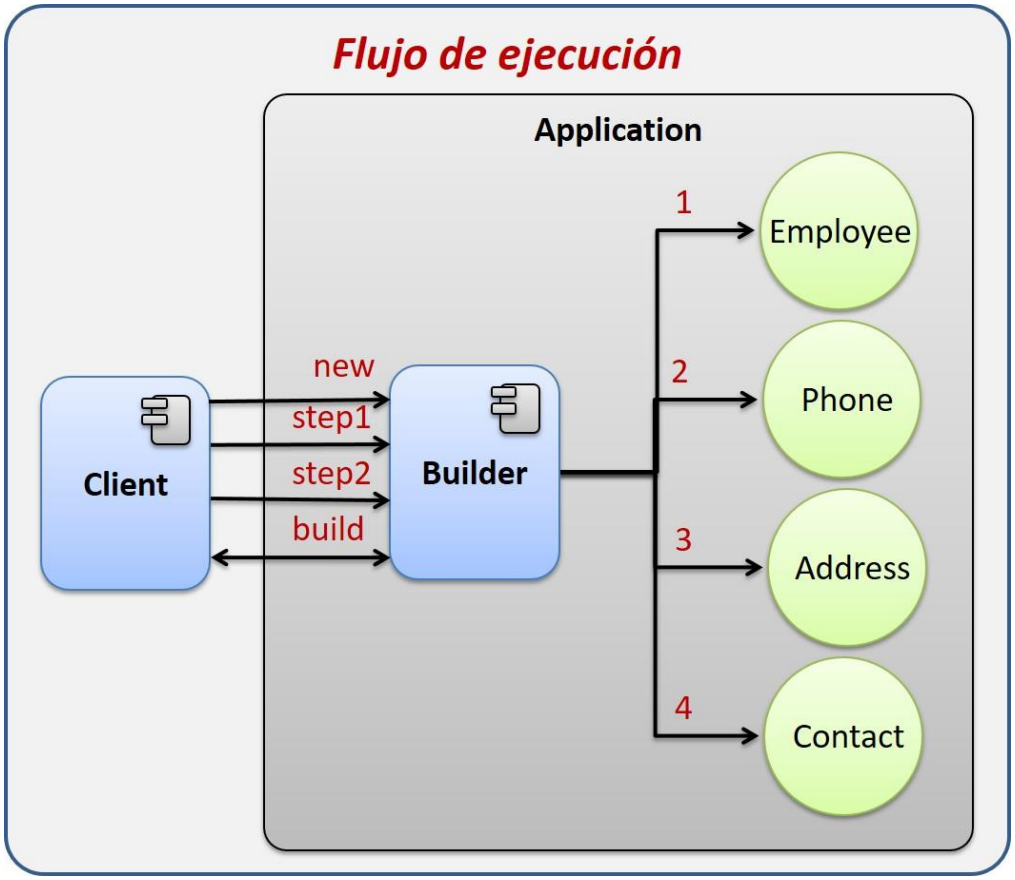


Ilustración 30: Flujo de ejecución utilizando el patrón de diseño Builder.

Analizando la imagen anterior, podemos ver que se crea un objeto *Builder*, el cual, se coloca justo delante de los objetos a crear, de esta manera, el cliente se comunica únicamente con el *Builder* para la creación del Empleado; podemos ver que el cliente al no tener una relación directa con el empleado, no tiene porqué conocer los detalle finos sobre la estructura de los objetos y menos el orden en que deberán ser creados y relacionados entre sí.

La implementación:

Con el escenario planteado y con el conocimiento de lo que tenemos que hacer, iniciaremos con la implementación de un programa que demuestre cómo utilizar el patrón.

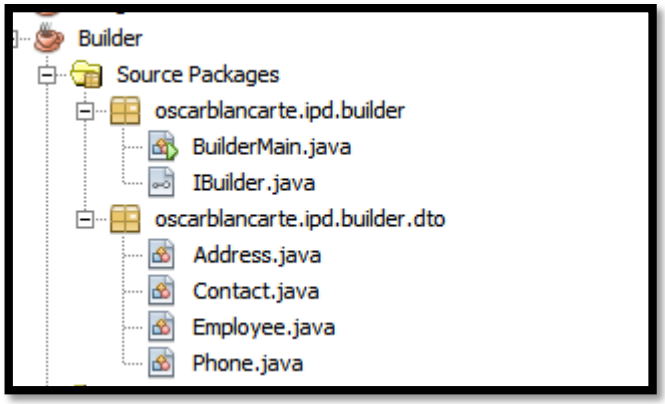


Ilustración 31: Estructura del proyecto Builder.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.builder:** Paquete principal de la aplicación en donde se encuentra la clase principal y la interface para crear los *Builders.*
- **oscarblancarte.ipd.builder.dto:** Paquete con los objetos que serán creados por el *Builder*.

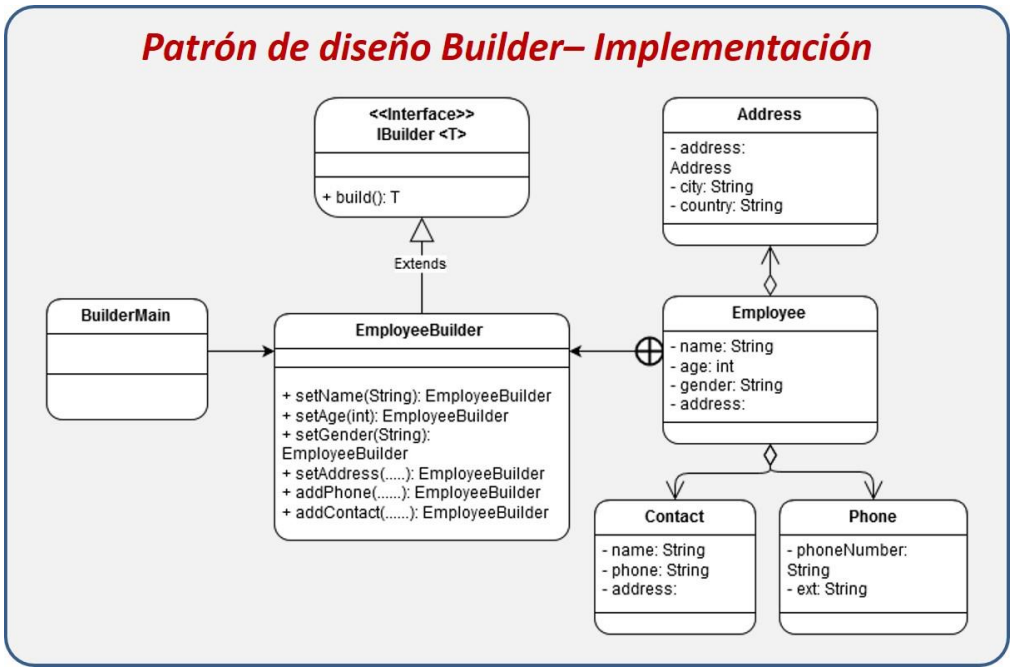


Ilustración 32: Diagrama de clases del proyecto Builder.

Interface IBuilder:

La interface *IBuilder* la utilizaremos pare tener un interface en común entre todos los *Builder* que creemos en la aplicación, esta interface es opcional y no es necesario crearla, sin embargo, la creamos pensando en que podríamos tener más clases *Builder* para crear otros objetos.

Podemos apreciar que la interface tiene un tipo genérico *T*, el cual lo reemplazaremos por el tipo de objeto que queramos construir, el genérico también lo utilizamos en el método *build,* el cual regresa un objeto de tipo T, este método será el que finalmente construya al Empleado.

```
1. package oscarblancarte.ipd.builder;
2.
3. public interface IBuilder<T> {
4.     public T build();
5. }
```

Clase Employee:

La clase *Employee* es la clase más interesante de este programa, ya que además de representar el objeto al que nos estamos enfocando y que queremos construir, también tiene una *inner class* llamada *EmployeeBuilder* que implementa el patrón de diseño *Builder*, esta clase tiene la particularidad de ser una clase *static* lo que hace que sea totalmente independiente de una instancia de la clase *Employee*.

La clase *EmployeeBuilder* tiene un conjunto de métodos que nos permite establecer las propiedades del empleado y sus relaciones con otros objetos. Estos métodos deben retornar una instancia de sí mismo, como ya lo habíamos comentado; esta característica será esencial y más adelante veremos por qué. Finalmente, podemos observar que la clase implementa la interface *IBuilder* y define como tipo genérico la clase *Employee*, lo que la obliga a definir el método *build* con el mismo tipo de retorno.

```
1.  package oscarblancarte.ipd.builder.dto;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import oscarblancarte.ipd.builder.IBuilder;
6.
7.  public class Employee {
8.
9.      private String name;
10.         private int age;
11.         private String gender;
12.         private Address adress;
13.         private List<Phone> phones;
14.         private List<Contact> contacs;
15.
16.         private Employee(String name, int age, String gender, Address adress,
17.                 List<Phone> phones, List<Contact> contacs) {
18.             this.name = name;
19.             this.age = age;
20.             this.gender = gender;
21.             this.adress = adress;
22.             this.phones = phones;
23.             this.contacs = contacs;
24.         }
25.
26.         public Employee(String name, int age, String gender) {
```

```
27.         this.name = name;
28.         this.age = age;
29.         this.gender = gender;
30.     }
31.
32.     public Employee() {
33.     }
34.
35.     /** GET and SET */
36.
37.     public static class EmployeeBuilder implements IBuilder<Employee>{
38.
39.         private String name;
40.         private int age;
41.         private String gender;
42.         private Address adress;
43.         private final List<Phone> phones = new ArrayList<>();
44.         private final List<Contact> contacs = new ArrayList<>();
45.
46.         public EmployeeBuilder() {
47.         }
48.
49.         public EmployeeBuilder setName(String name) {
50.             this.name = name;
51.             return this;
52.         }
53.
54.         public EmployeeBuilder setAge(int age) {
55.             this.age = age;
56.             return this;
57.         }
58.
59.         public EmployeeBuilder setGender(String gender) {
60.             this.gender = gender;
61.             return this;
62.         }
63.
64.         public EmployeeBuilder setAdress(String address, String city,
65.                 String country, String cp) {
66.             adress = new Address(address, city, country, cp);
67.             return this;
68.         }
69.
70.         public EmployeeBuilder addPhones(String phoneNumber, String ext,
71.                 String phoneType) {
72.             phones.add(new Phone(phoneNumber, ext, phoneType));
73.             return this;
74.         }
75.
76.         public EmployeeBuilder addContacs(String name, String phoneNumber,
77.                 String ext, String phoneType,String address, String city,
78.                 String country, String cp) {
79.             contacs.add(new Contact(name, new Phone(phoneNumber, ext, phoneType)
```

```
80.                     ,new Address(address, city, country, cp)));
81.             return this;
82.         }
83.
84.         public EmployeeBuilder addContacs(String name, String phoneNumber,
85.                 String ext, String phoneType) {
86.             contacs.add(new Contact(name, new Phone(phoneNumber, ext, phoneType)));
87.             return this;
88.         }
89.
90.         @Override
91.         public Employee build() {
92.             return new Employee(name, age, gender, adress, phones, contacs);
93.         }
94.     }
95. }
```

Clase Phone:

La clase *Phone* no tiene nada en particular, sólo define las propiedades *phoneNumber*, *ext* y *phoneType*. Esta clase representará un teléfono del empleado o de un contacto.

```
1. package oscarblancarte.ipd.builder.dto;
2.
3. public class Phone {
4.     private String phoneNumber;
5.     private String ext;
6.     private String phoneType;
7.
8.     public Phone() {
9.     }
10.
11.         public Phone(String phoneNumber, String ext, String phoneType) {
12.             this.phoneNumber = phoneNumber;
13.             this.ext = ext;
14.             this.phoneType = phoneType;
15.         }
16.         /** GET and SET */
17.     }
```

Clase Address:

La clase *Address* representará el domicilio de un empleado o de un contacto, la clase no tiene nada en particular únicamente define las propiedades *address*, *city*, *country*, *cp*.

```
1.  package oscarblancarte.ipd.builder.dto;
2.
3.  public class Address {
4.      private String address;
5.      private String city;
6.      private String country;
7.      private String cp;
8.
9.      public Address() {
10.         }
11.
12.         public Address(String address, String city, String country, String cp) {
13.             this.address = address;
14.             this.city = city;
15.             this.country = country;
16.             this.cp = cp;
17.         }
18.
19.         /** GET and SET */
20.     }
```

Clase Contact:

La clase *Contact* representa un contacto del empleado, un empleado puede tener varios contactos asociados. Esta clase define las propiedades *name*, *phone* y *address*.

```
1.  package oscarblancarte.ipd.builder.dto;
2.
3.  public class Contact {
4.      private String name;
5.      private Phone phone;
6.      private Address address;
7.
8.      public Contact() {
9.      }
10.
11.     public Contact(String name, Phone phone,Address address) {
```

```
12.         this.name = name;
13.         this.phone = phone;
14.     }
15.
16.     public Contact(String name, Phone phone) {
17.         this.name = name;
18.         this.phone = phone;
19.     }
20.     /** GET and SET */
21. }
```

Clase BuilderMain

Por último, la clase *BuilderMain* es una clase ejecutable que intenta demostrar el uso del *Builder* al crear una instancia de la clase *Employee*.

```
1.  package oscarblancarte.ipd.builder;
2.
3.  import oscarblancarte.ipd.builder.dto.Employee;
4.
5.  public class BuilderMain {
6.
7.      public static void main(String[] args) {
8.          Employee emp = new Employee.EmployeeBuilder()
9.                  .setName("Oscar Javier Blancarte Iturralde")
10.                 .setGender("Male")
11.                 .setAge(29)
12.                 .setAdress("Aragón 189 int 404 col. Álamos, delegación Benito "
13.                         + "Juarez", "México D.F.", "México", "03400")
14.                 .addContacs("Rene Blancarte", "1122334455", "123", "Casa",
15.                         "Chapultepect No. 123 Col. Militar", "México"
16.                         , "México", "10023")
17.                 .addContacs("Jaime Blancarte", "3344556677", null, "Celular")
18.                 .addPhones("4567890234", null, "Celular")
19.                 .addPhones("7788990099", null, "Casa")
20.                 .build();
21.         System.out.println(emp);
22.     }
23. }
```

En la línea 8 podemos observar cómo es que se inicia con la creación del empleado y la instrucción termina hasta la línea 23, observemos que toda la creación es sólo una instrucción, esto es posible ya que los métodos para establecer las propiedades regresan al mismo *EmployeBuilder*, que nos permite seguir estableciendo más propiedades sin finalizar la instrucción; en último lugar, llamamos al método *build*, encargado de crear la instancia del Empleado como vemos en la línea 20.

Prestemos atención en que en ningún momento tuvimos que crear un objeto *Employee*, *Address*, *Contact*, *Phone* ya que *EmployeeBuilder* proporcionan métodos más agradables.

La ejecución:

Para demostrar la efectividad del patrón, ejecutaremos la clase *BuilderMain*:

```
run:
Employee{name=Oscar Javier Blancarte Iturralde, age=29, gender=Male,
adress=Address{address=Aragon 189 int 404 col. Alamos, delegacion Benito Juarez, city=Mexico D.F., country=M
exico, cp=03400},
phones=[Phone{phoneNumber=4567890234, ext=null, phoneType=Celular}, Phone{phoneNumber=7788990099, ext=null,
phoneType=Casa}],
contacs=[Contact{name=Rene Blancarte, phone=Phone{phoneNumber=1122334455, ext=123, phoneType=Casa}}, Contact
{name=Jaime Blancarte, phone=Phone{phoneNumber=3344556677, ext=null, phoneType=Celular}}]}
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 33: Resultado de la ejecución de la clase *BuilderMain*.

El resultado es bastante simple, se imprime el *toString* de la clase *Employee* que está configurada para imprimir los valores de todas sus propiedades. Podemos ver que en un solo paso hemos creado un objeto compuesto de forma sencilla y sin conocer realmente la estructura del empleado.

Siguientes pasos:

Hemos demostrado las cualidades del patrón de diseño *Builder* a la hora de crear instancias de objetos sin la necesidad de conocer muy bien cómo es que está implementada la clase por dentro, el usuario final no requiere saber dónde tiene que poner los contactos o dónde poner el domicilio ni ninguna otra propiedad, ya que el objeto *Builder* es el que se encarga de hacer esto por él de una forma muy elegante, esto nos ayuda mucho sobre todo si el proceso de creación del objeto es muy repetitivo en nuestro programa.

Para fortalecer los conocimientos adquiridos realicemos los siguientes ejercicios:

- Modifiquemos la clase *Employee* para agregarle una nueva relación con la clase *Department* (departamento) la cual será una nueva clase que tendremos que crear, la clase departamento deberá tener el nombre del departamento al que pertenece y finalmente agreguemos la construcción de este objeto al *EmployeeBuilder*.

- Creemos una nuevo *Builder* que nos permita crear objetos de tipo *Contact* (contacto), para lo cual tendremos que heredar de *IBuilder*.

Patrón Prototype

El patrón Prototype basa su funcionalidad en la clonación de objetos, estos nuevos objetos son creados mediante un *pool* de prototipos elaborados previamente y almacenados. Este patrón es especialmente útil cuando necesitamos crear objetos basados en otros ya existentes o cuando se necesita la creación de estructuras de objetos muy grandes, este patrón nos ayuda también a ocultar la estrategia utilizada para clonar un objeto.

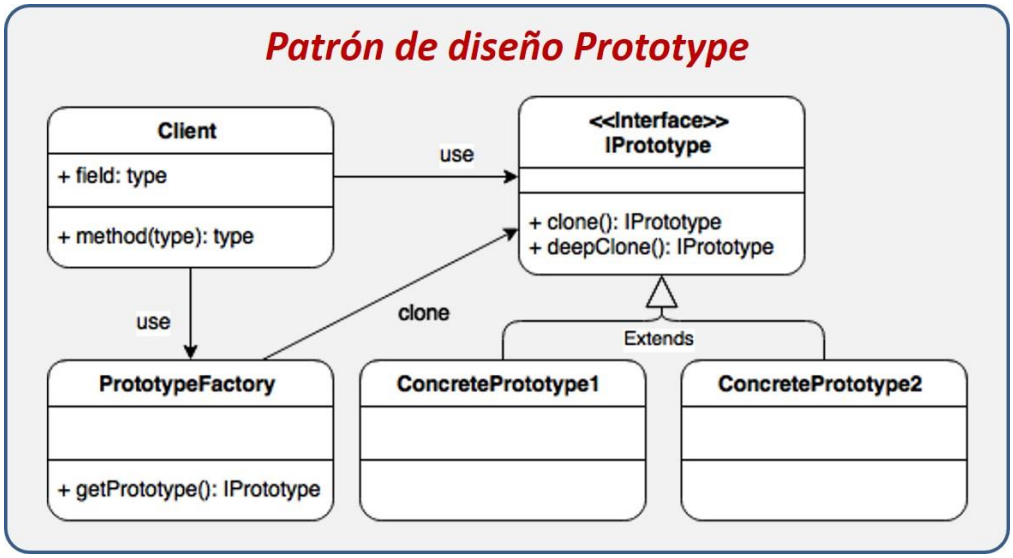


Ilustración 34: Estructura del patrón de diseño Prototype.

Los componentes que conforman el patrón son los siguientes:

- **Client:** Componente que interactúa con los prototipos.
- **IPrototype**: Este componente es por lo general una interface y define los atributos mínimos de un prototipo, esta interface debe contar por lo menos con alguno de los dos tipos de clonación. Clonación superficial (*clone*) o clonación en profundidad (*deepClone*) los cuales explicaremos más adelante.
- **ConcretePrototype**: Implementaciones concretas de *IPrototype*, los cuales son objetos que podrán ser clonados.

- **PrototypeFactory**: Componente que utilizaremos para mantener el *cache* de los prototipos existentes, así como para crear clonaciones de los mismos.

La clonación es el proceso por el cual un objeto es copiado para crear otro idéntico, y se puede dar de dos formas; la primera es una clonación del objeto de una forma superficial, es decir, se crea una copia del objeto principal o el objeto raíz de la estructura, pero todos los objetos internos no se clonan, en vez de ello, son compartidos por la instancia original y la instancia clonada.

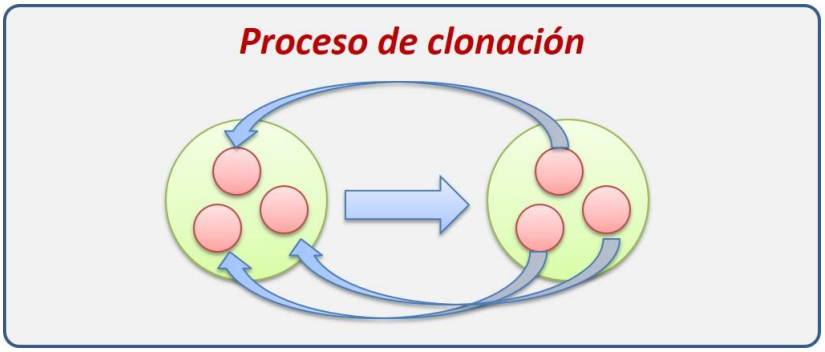


Ilustración 35: Imagen de una clonación superficial.

En la imagen podemos apreciar que el objeto principal fue clonado, sin embargo, todos los objetos internos siguen siendo los mismos de la instancia original.

Por otro lado, la clonación en profundidad realiza una copia idéntica del prototipo, incluyendo todos los objetos que éste contenga, de esta forma tendríamos una copia perfecta de todo el prototipo.

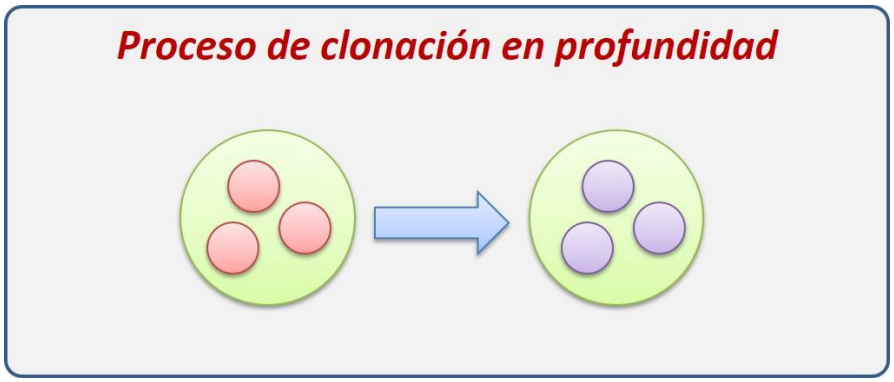


Ilustración 36: Imagen de una clonación en profundidad.

Esta clonación crea una copia total de todo el prototipo, incluyendo todos los objetos que contiene. Este tipo de clonación nos asegura que no podremos alterar de ninguna forma el prototipo original.

Cuándo utilizarlo:

- Cuando se necesitan crear nuevos objetos basados en objetos existentes.
- Cuando la creación directa de un objeto por medio del operador new puede ser perjudicial.
- Cuando se tiene gran cantidad de objetos con atributos repetidos, siempre es mucho más rápido clonar que crear nuevos objetos y *setear* cada valor.

El escenario:

Para comprender mejor cómo funciona este patrón de diseño, hablaremos de un escenario muy común en las empresas. Sin duda, todos sabemos que las empresas que venden productos o servicios tiene un catálogo de los productos que ofrecen, estos productos tiene un precio asociado que por lo general es fijo, sin embargo, existen los escenarios en los que contamos con clientes mayoristas o clientes VIP a los cuales siempre les manejamos un precio especial; por lo cual, tenemos la necesidad de manejar un listado de precios distinto para estos clientes, pero ¿qué pasaría si nuestro listado de productos es muy grande y capturar un listado nuevo hace perder mucho tiempo?.

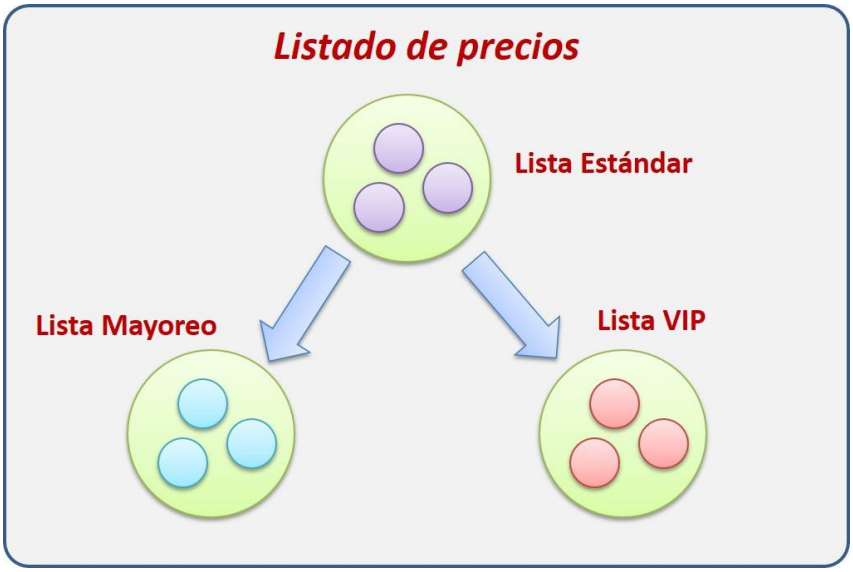


Ilustración 37: Creación de listado de precios a partir de uno existente.

Lo que necesitamos es crear un listado de precios a partir de uno ya existente, el cual tenga los precios estándares y sobre éste crear otros con un descuento adicional.

La solución:

Para poder dar una solución práctica a este escenario, podemos utilizar el patrón de diseño *Prototype*, con el objetivo de utilizar el listado de precio estándar para crear dos nuevas listas de precios; crearemos la lista de precios para clientes de mayoreo, dándoles un 10% de descuento sobre el listado original, seguido, crearemos un nuevo listado de precios para clientes VIP el cual tendrá un 10% de descuento adicional sobre el listado de clientes de mayoreo; para lo cual, el listado de mayoreo también lo utilizaremos como prototipo. En la siguiente imagen podemos apreciar mejor lo explicado:

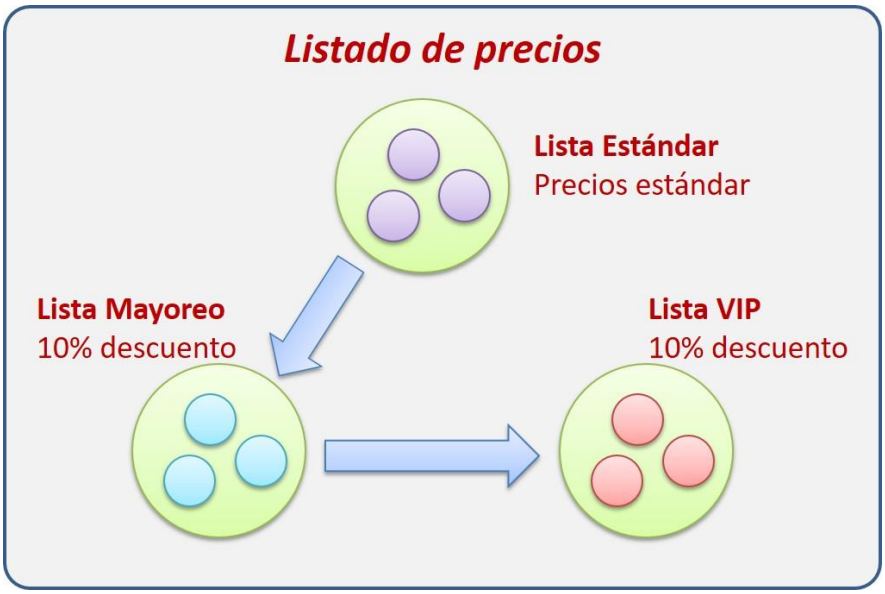


Ilustración 38: Creación de listas de precios a partir de otra.

Así, podemos basarnos en la estructura de una lista ya existente para crear una totalmente nueva.

La implementación:

Una vez explicado el escenario y la solución, empezaremos a construir un programa que nos ayude a demostrar la utilidad del patrón de diseño.

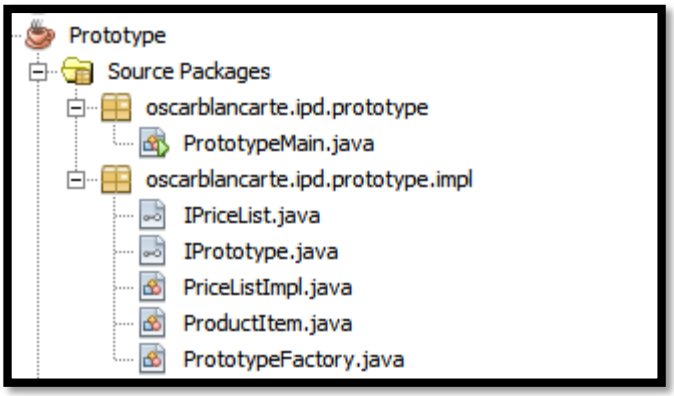


Ilustración 39: Estructura del proyecto Prototype.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.prototype:** Paquete principal del proyecto con la clase ejecutable del proyecto.
- **oscarblancarte.ipd.prototype.impl:** Paquete con las clases de implementación del patrón *Prototype*.

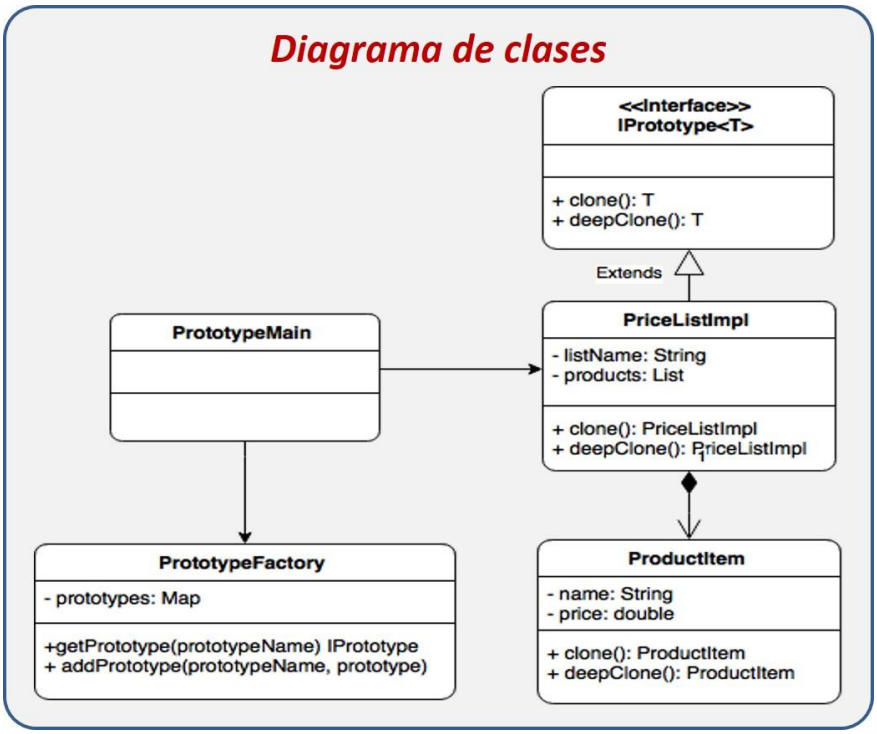


Ilustración 40: Diagrama de clases del proyecto Prototype.

Clase IPrototype:

El primer componente que definiremos será la interface *IPrototype*, la cual nos servirá como base para crear todos los prototipos.

Se definen los siguientes métodos:

- **clone:** Permite realizar una clonacion superficial del prototipo.
- **deepClone:** Permite realizar una clonación profunda del prototipo.

```
1.  package oscarblancarte.ipd.prototype.impl;
2.
3.  public interface IPrototype<T extends IPrototype> extends Cloneable{
4.      public T clone();
5.      public T deepClone();
6.  }
```

Implementamos la interface *Cloneable* para aprovechar la funcionalidad de clonación estándar de Java y definimos el genérico T para asegurarnos de que todos los prototipos implementen la interface *IPrototype*.

Clase PrototypeFactory:

Esta clase sigue el patrón de diseño *Factory* que, como vimos más atrás, la utilizaremos para delegarle la responsabilidad de crear los prototipos que tendríamos que hacer nosotros.

```
1.  package oscarblancarte.ipd.prototype.impl;
2.
3.  import java.util.HashMap;
4.
5.  public class PrototypeFactory {
6.      private static HashMap<String,IPrototype> prototypes = new HashMap<>();
7.
8.      public static IPrototype getPrototype(String prototypeName){
9.          return prototypes.get(prototypeName).deepClone();
10.         }
11.
12.         public static void addPrototype(String prototypeName,IPrototype prototype){
13.             prototypes.put(prototypeName, prototype);
14.         }
15.     }
```

Se definen los siguientes métodos:

- **addPrototype**: Permite registrar nuevos prototipos para poder clonarlos más adelante.
- **getPrototype**: Realiza una clonación en profundidad del prototipo solicitado, el prototipo será buscado por medio de su nombre.

Todos los prototipos registrados en el *Factory* se almacenan en el Map *prototipes*.

Clase PriceListImpl:

Esta clase será la implementación de una lista de precios, lo primero que podemos observar es que implementa *IPrototype* y con ello los métodos *clone* y *deepClone* que nos servirán para hacer la clonación de la lista de precio.

```
1.  package oscarblancarte.ipd.prototype.impl;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class PriceListImpl implements IPriceList<PriceListImpl>{
7.      private String listName;
8.      private List<ProductItem> products = new ArrayList<ProductItem>();
9.
10.         public PriceListImpl(String listName) {
11.             this.listName = listName;
12.         }
13.
14.         /** GET and SET */
15.
16.         @Override
17.         public PriceListImpl clone() {
18.             PriceListImpl clone = new PriceListImpl(listName);
19.             clone.setProducts(products);
20.             return clone;
21.         }
22.
23.         @Override
24.         public PriceListImpl deepClone() {
25.             List<ProductItem> cloneProducts = new ArrayList<>();
26.             for(ProductItem item : this.products){
27.                 ProductItem cloneItem = item.clone();
28.                 cloneProducts.add(cloneItem);
29.             }
30.             PriceListImpl clone = new PriceListImpl(listName);
31.             clone.setProducts(cloneProducts);
32.             return clone;
33.         }
34.
35.         @Override
36.         public String toString() {
37.             String items = "";
38.             for(ProductItem item : this.products){
39.                 items += "\t" + item.toString() + "\n";
40.             }
41.             return "PriceListImpl{listName=" + listName + ", products=\n" + items + '}';
42.         }
43.     }
```

Veamos que el método *clone* únicamente clona la lista de precios pero todos los productos no son clonados y son compartidos con la lista de precios original. El método *deepClone* por otra parte, realiza una clonación en profundidad, incluyendo a los productos asociados.
Finalmente, el método *toString* es utilizado para imprimir el listado de los productos.

Clase ProductItem:

Esta clase la utilizaremos para representar a los productos de la lista de precios, la lista de precios puede tener uno o muchos *ProductItem* asociados. La clase hereda de *IPrototype* con el fin de poder realizar clonaciones de los *ítems.* Cuando realicemos una clonación en profundidad de una lista de precios el método clone de los *ProductItem* será llamado.

```
1.  package oscarblancarte.ipd.prototype.impl;
2.
3.  public class ProductItem implements IPrototype<ProductItem> {
4.      private String name;
5.      private double price;
6.
7.      public ProductItem(){
8.      }
9.
10.         public ProductItem(String name, double price) {
11.             this();
12.             this.name = name;
13.             this.price = price;
14.         }
15.
16.         /** GET and SET */
17.
18.         @Override
19.         public ProductItem clone() {
20.             return new ProductItem(this.name,this.price);
21.         }
22.
23.         @Override
24.         public ProductItem deepClone() {
25.             return clone();
26.         }
27.
28.         @Override
```

```
29.         public String toString() {
30.             return "ProductItem{" + "name=" + name + ", price=" + price + '}';
31.         }
32.     }
```

Clase PrototypeMain:

Ya por último, tenemos la clase *PrototypeMain* desde la cual podremos finalmente demostrar lo que hemos estado explicado.

```
1.  package oscarblancarte.ipd.prototype;
2.
3.  import oscarblancarte.ipd.prototype.impl.PriceListImpl;
4.  import oscarblancarte.ipd.prototype.impl.ProductItem;
5.  import oscarblancarte.ipd.prototype.impl.PrototypeFactory;
6.
7.  public class PrototypeMain {
8.
9.      public static void main(String[] args) {
10.
11.             //Creamos la lista de precios inicial, ésta tiene los productos con
12.             //el precio de lista.
13.             PriceListImpl standarPriceList =
14.                     new PriceListImpl("Standar Price List");
15.             for(int c = 1; c<=5; c++){
16.                 ProductItem item = new ProductItem("Product " + c, c*2);
17.                 standarPriceList.addProductItem(item);
18.             }
19.             PrototypeFactory.addPrototype(
20.                     standarPriceList.getListName(), standarPriceList);
21.
22.             //Segunda lista para clientes de mayoreo a partir de la lista
23.             //estándar con un 10% de descuento sobre la lista de precio estándar.
24.             PriceListImpl wholesalePriceList = (PriceListImpl)
25.                     PrototypeFactory.getPrototype("Standar Price List");
26.             wholesalePriceList.setListName("Wholesale Price List");
27.             for(ProductItem item : wholesalePriceList.getProducts()){
28.                 item.setPrice(item.getPrice()*0.90);
29.             }
30.             PrototypeFactory.addPrototype(
31.                     wholesalePriceList.getListName(), wholesalePriceList);
32.
33.             //Tercera lista de precios para clientes VIP a partir de la lista
34.             //de mayoreo con 10% de descuento sobre la lista de precios de mayoreo.
35.             PriceListImpl vipPriceList = (PriceListImpl)
36.                     PrototypeFactory.getPrototype("Wholesale Price List");
37.             vipPriceList.setListName("VIP Price List");
38.             for(ProductItem item : vipPriceList.getProducts()){
39.                 item.setPrice(item.getPrice()*0.90);
```

```
40.         }
41.
42.         //Imprimimos las listas de precios.
43.         System.out.println(standarPriceList);
44.         System.out.println(wholesalePriceList);
45.         System.out.println(vipPriceList);
46.     }
47. }
```

De la línea 13 a la 18 creamos de forma dinámica la lista de precios estándar, esta lista contendrá cinco productos con un precio distinto. En la línea 19, registramos este listado de precios en nuestro *PrototypeFactory* con el fin de crear nuevos listados de precios a partir de él.

En la línea 24 podemos apreciar las ventajas de este patrón, ya que creamos un nuevo listado de precios a partir del primero, este nuevo listado de precios tiene el nombre de ***Wholesale Price List***. En las líneas 27 a la 29, hacemos un recálculo del precio para darle un 10% de descuento. En la línea 30 registramos este listado de precios para que pueda ser utilizado como prototipo de otros listados de precios.

Nuevamente en la línea 35, clonamos la lista de precios **Wholesale Price List** con el fin de crear una tercera lista para clientes VIP, esta lista tendrá un 10% de descuento sobre la lista de precio de mayoreo (*Wholesale Price List*), este cálculo lo hacemos en la línea 38 a 40.

Por último, en la línea 43 a la 45 imprimimos los tres listados de precios para ver los resultados.

La ejecución:

Para demostrar la funcionalidad del programa ejecutaremos la clase *PrototypeMain* para obtener el siguiente resultado:

```
run:
PriceListImpl{listName=Standar Price List, products=
        ProductItem{name=Product 1, price=2.0}
        ProductItem{name=Product 2, price=4.0}
        ProductItem{name=Product 3, price=6.0}
        ProductItem{name=Product 4, price=8.0}
        ProductItem{name=Product 5, price=10.0}
}
PriceListImpl{listName=Wholesale Price List, products=
        ProductItem{name=Product 1, price=1.8}
        ProductItem{name=Product 2, price=3.6}
        ProductItem{name=Product 3, price=5.4}
        ProductItem{name=Product 4, price=7.2}
        ProductItem{name=Product 5, price=9.0}
}
PriceListImpl{listName=VIP Price List, products=
        ProductItem{name=Product 1, price=1.62}
        ProductItem{name=Product 2, price=3.24}
        ProductItem{name=Product 3, price=4.86}
        ProductItem{name=Product 4, price=6.48}
        ProductItem{name=Product 5, price=8.1}
}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 41: Resultado de la ejecución de la clase *PrototypeMain*.

Como vemos en la imagen, el primer listado tiene los precios originales, la segunda fue creada a partir de la primera y se le aplicó un descuento del 10%, la tercera fue creada a partir de la segunda lista de precios y se le aplicó un 10% de descuento sobre una lista que ya tenía un 10% de descuento sobre el precio original.

Siguientes pasos:

Este patrón es especialmente útil cuando requerimos crear objetos basados en otros ya existentes, aunque también puede ser muy ventajoso cuando se requiere hacer copia de estructuras de objetos.

Para fortalecer los conocimientos adquiridos realicemos los siguientes ejercicios:

- Creemos una nueva lista de precios que ofrezca un descuento del 30% sobre el precio de lista.

- Creemos una nueva clase llamada *Product* la cual tenga los datos del producto (nombre y precios), ésta será agregada al *ProductItem* y deberá ser contemplada en el proceso de clonación en profundidad.

- Modifiquemos el proyecto original para que en lugar de crear clonaciones en profundidad cree clonaciones superficiales, ejecutemos el programa y analicemos el resultado, hay que determinar por qué todas las listas tienen los mismos precios.

Patrón Object Pool

Este es un patrón muy utilizado cuando se requiere trabajar con una gran cantidad de objetos, los cuales son computacionalmente caros de crear. Este patrón tiene una gran ventaja en escenarios donde nuestro programa requiere dichos objetos por un corto periodo de tiempo y que luego de eso, son desechados. La ventaja que brinda este patrón es que nos permite reutilizar los objetos con el fin de evitar la tarea de crearlos cada vez que nuestra aplicación los requiere, manteniendo así un almacén de objetos creados previamente para ser utilizados.

Cuando la aplicación requiere uno de estos objetos, en vez de crearlo, le solicita al *ObjectPool* que le preste uno; la aplicación utilizará el objeto por un tiempo determinado y cuando lo desocupe lo regresará al *ObjectPool* para que otro proceso pueda reutilizarlo.

El *ObjectPool* por *default* inicia con una cantidad de objetos determinados, cuando un proceso solicite un objeto al *ObjectPool* éste regresará uno de los objetos disponibles, si en el momento de la solicitud el *ObjetPool* no tiene ninguno disponible, creará uno nuevo y lo prestará a nuestra aplicación.

El *Object Pool* también tiene un límite máximo de objetos, por lo que si se alcanza este número, ya no creará más, en vez de esto, pondrá en espera al proceso hasta que un objeto sea liberado o se llegue a un tiempo de espera máximo.

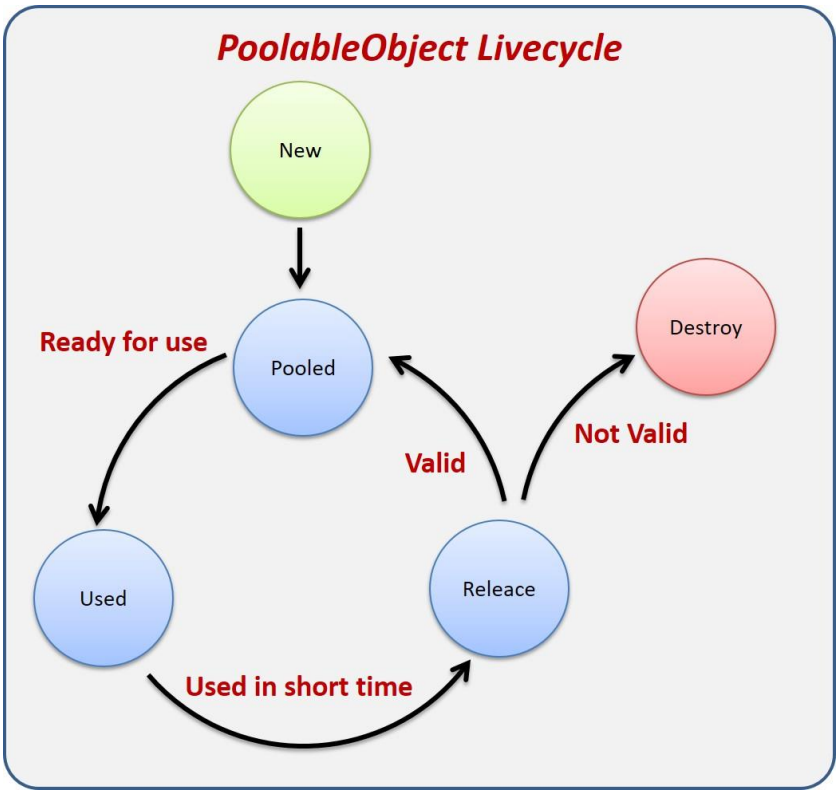


Ilustración 42: Ciclo de vida de un objeto dentro del ObjectPool.

El ciclo de vida de un objeto es el siguiente:

- **New:** Un nuevo objeto es creado debido a que el *ObjectPool* no tiene más objetos disponibles o porque está iniciando y necesita crear las instancias mínimas.
- **Pooled:** El objeto nuevo o liberado es puesto nuevamente en el *ObjectPool* con el fin de que esté disponible para ser utilizado por otro proceso.
- **Used:** El objeto es solicitado por un proceso y no estará disponible hasta que sea liberado.
- **Releace:** El objeto es liberado por el proceso que lo solicitó y puesto nuevamente en el *ObjectPool* o es destruido en caso de invalidarse.
- **Destroy:** El objeto es destruido por una solicitud o porque ya no sirve más.

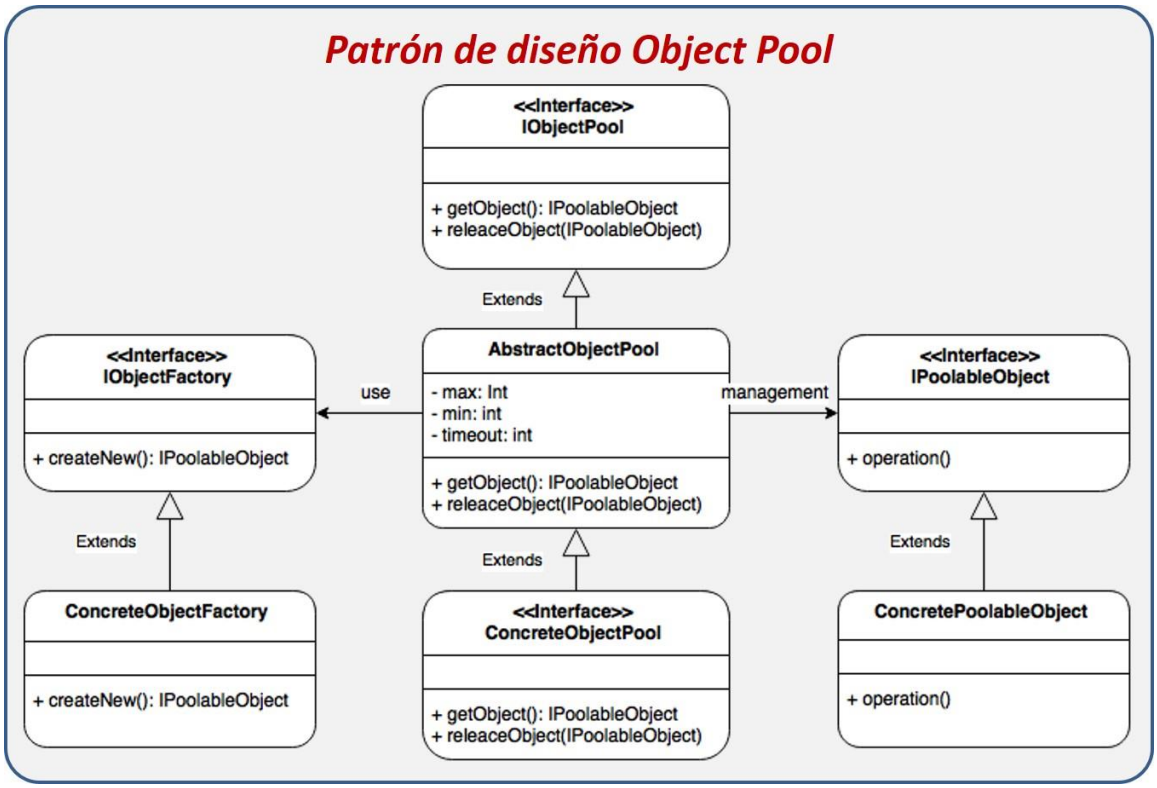


Ilustración 43: Estructura del patrón de diseño Object Pool.

Los componentes que conforman el patrón son los siguientes:

- **IObjectPool:** Interface que define la estructura mínima de un *Object Pool*, en ella definiremos los métodos *get* y *releace*, los cuales serán utilizados para obtener y devolver un objeto respectivamente.
- **AbstractObjectPool:** Clase abstracta que definirá el comportamiento por *default* de un *Object Pool*, esta clase puede ser utilizada como base para crear *ConcreteObjectPool* de una forma muy rápida.
- **ConcreteObjectPool:** Corresponde a una implementación completa del *Object Pool*.
- **IPoolableObject:** Interface que deberá implementar todos los objetos que deseamos administrar por el *ObjectPool*. En esta interface podemos definir la estructura mínima que deben de tener los objetos.

- **ConcretePoolableObject:** Estos son los objetos reales que el *ObjectPool* administrará, deberán heredar de *IPoolableObject*.
- **IObjectFactory:** Interface que definirá la estructura de la fábrica de objetos *ConcretePoolableObject*.
- **ConcreteObjectFactory:** Fábrica concreta que implementa *IObjectFactory* para la creación de objetos *ConcretePoolableObject*.

Un detalle importante a resaltar es que el componente *ConcreteObjectFactory* sigue el patrón de diseño ***Factory Method*** y es utilizados para delegar la responsabilidad de crear objetos nuevos cuando éstos se agotan.
Es importante notar que lo hacemos así porque el *ObjectPool* es el responsable de proveer los objetos, pero no sabe cómo es que deben de ser creados.

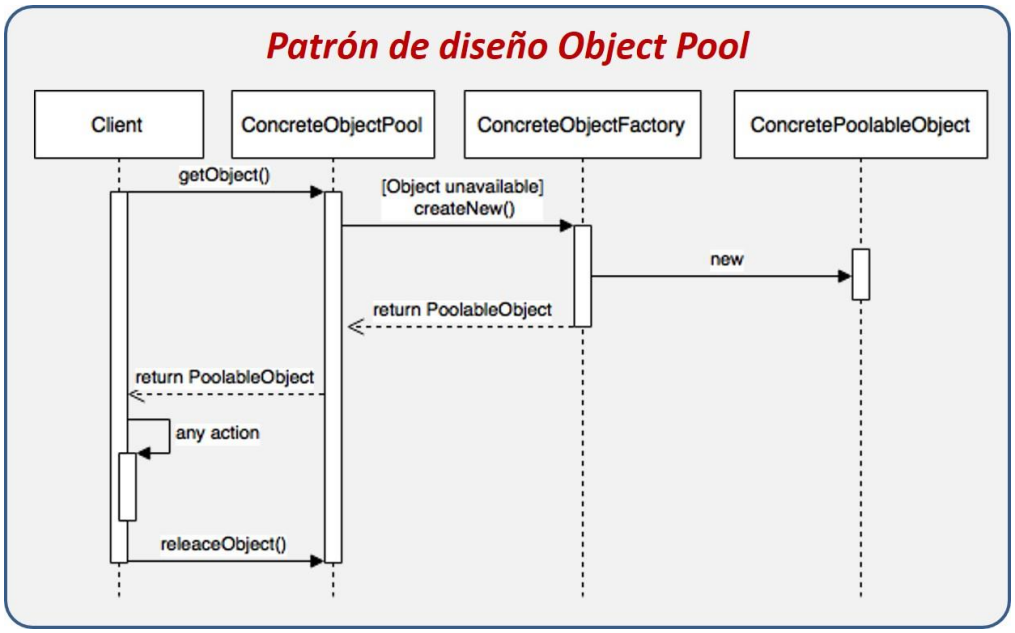


Ilustración 44: Diagrama de secuencia del patrón de diseño Object Pool.

El diagrama se interpreta de la siguiente manera:

1. El cliente solicita un objeto al *ConcreteObjectPool*.

2. El *ConcreteObjectPool* valida si existen objetos disponibles, de no ser así, solicitará la creación de un nuevo objeto al *ConcreteObjectFactory*.
3. El *ConcreteObjectFactory* creará un nuevo objeto de tipo *ConcretePoolableObject*.
4. El *ConcreteObjectPool* regresará el objeto al cliente.
5. El cliente realiza cualquier acción con el objeto.
6. El cliente regresa el objeto al *ConcreteObjectPool*.

Al diagrama anterior sólo le faltaría agregar las reglas de instancias máximas, mínimas y el *timeout*, sin embargo, estas características las analizaremos más adelante.

Cuándo utilizarlo:

- Cuando la creación de los objetos es computacionalmente cara.

- Cuando los objetos son utilizados por un corto periodo y luego son desechados.

- Mejorar el performance mediante un *pool* de objetos previamente creados.

El escenario:

En este ejemplo, imaginaremos que tenemos que desarrollar una aplicación multi-hilos, esta aplicación tendrá que hacer muchas tareas de forma simultánea, sin embargo, por cuestión de rendimiento, tenemos que administrar el número máximo de procesos que se pueden ejecutar al mismo tiempo, dejando en espera el resto de procesos; los procesos esperarán hasta que otro finalice para poder ser ejecutados, si la espera pasa de un tiempo determinado, el programa negará el servicio y los procesos no serán realizados.

Por cuestión de practicidad, los procesos simplemente se ejecutarán por 10 segundos y terminarán, sin embargo, si lo deseamos podemos implementar alguna tarea como escribir en una base de datos, consumir un WebService o procesar algun archivo de texto.

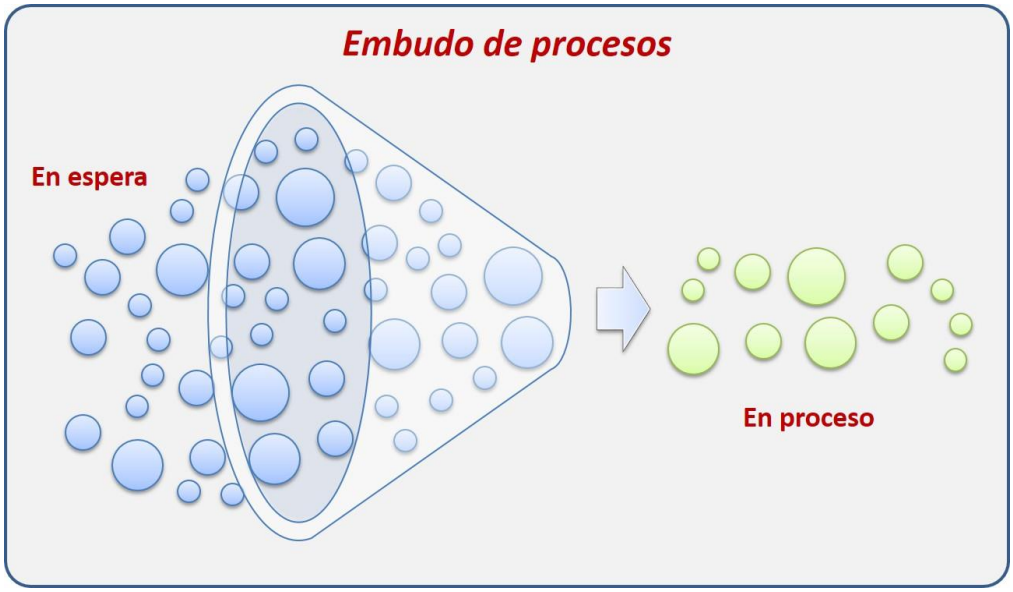


Ilustración 45: Forma en la que se atenderán los procesos.

En la imagen podemos apreciar cómo un conjunto de procesos intentan ser atendidos, pero dicho número será limitado, lo que provocará que los procesos que lleguen después, tengan que esperar a que los primeros que llegaron terminen.

La solución:

Como vimos hace un momento, el principal problema a resolver es desarrollar un componente que nos permita limitar el número de procesos que se pueden ejecutar al mismo tiempo y que los procesos puedan esperar sólo un tiempo determinado antes de desistir, por lo que crear un *ObjectPool* de procesos podría ser una excelente estrategia.

Una buena idea será crear objetos que nos permitan controlar la ejecución de cada proceso y estos objetos pueden ser administrados por un *ObjectPool*, de esta forma cada vez que una aplicación requiera lanzar un proceso tendrá que solicitar al *ObjectPool* un objeto de ejecución, el *ObjectPool* le prestará uno, sin embargo, si no hay objetos disponibles éste lo pondrá a la espera hasta que un componente libere un objeto.

El *ObjectPool* será configurable para mantener a los solicitantes en espera un tiempo determinado, si no logra satisfacer la necesidad del objeto en dicho tiempo, éste terminará la solicitud con una excepción personalizada.

La siguiente imagen trata de explicar el funcionamiento expuesto:

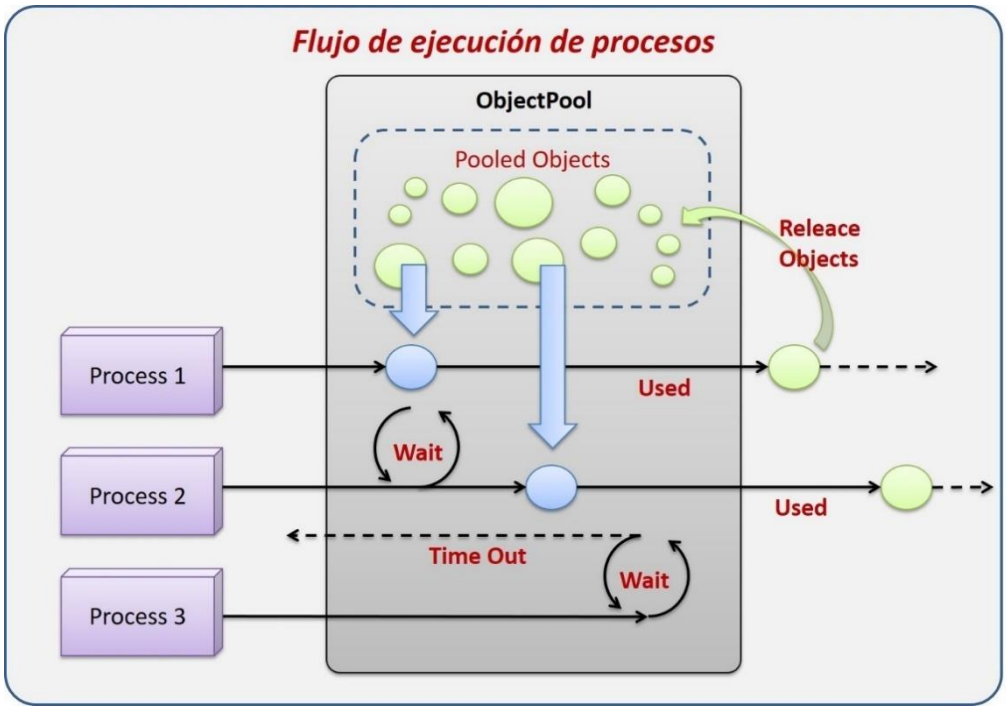


Ilustración 46: Flujo de la ejecución de procesos

Como vemos, existen tres posibles escenarios que se pudieran presentar cuando solicitamos un objeto al *ObjectPool*:
El primero que vemos (Process 1) solicita al *ObjectPool* un objeto de ejecución, el *pool* le presta un objeto de inmediato debido a que en ese momento tiene objetos disponibles, el proceso utiliza el objeto un tiempo determinado y lo regresa al *ObjectPool* para que el siguiente proceso pueda utilizarlo.

El segundo (Process 2) al igual que el proceso 1, intenta obtener un objeto de ejecución, sin embargo, éste no tiene tanta suerte debido a que todos los objetos están ocupados, entonces el *ObjectPool* lo tiene en espera para que otro proceso libere un objeto, después de un tiempo, el *pool* tiene objetos disponibles y le presta uno, de aquí en adelante el proceso continúa igual que en proceso 1.

El tercero (Process 3) también intenta obtener un objeto de ejecución, sin embargo, el *pool* no tiene disponibles y tras un tiempo de espera considerable, el

ObjectPool rechaza la solicitud lanzando una excepción al proceso, debido a que no fue posible prestarle un objeto.

Tanto el número máximo, mínimo y el tiempo de espera deberán ser configurables, con la finalidad de poder mover estos valores en cualquier momento.

La implementación:

Una vez que tenemos claro el problema y cómo resolverlo, iniciaremos con la implementación.

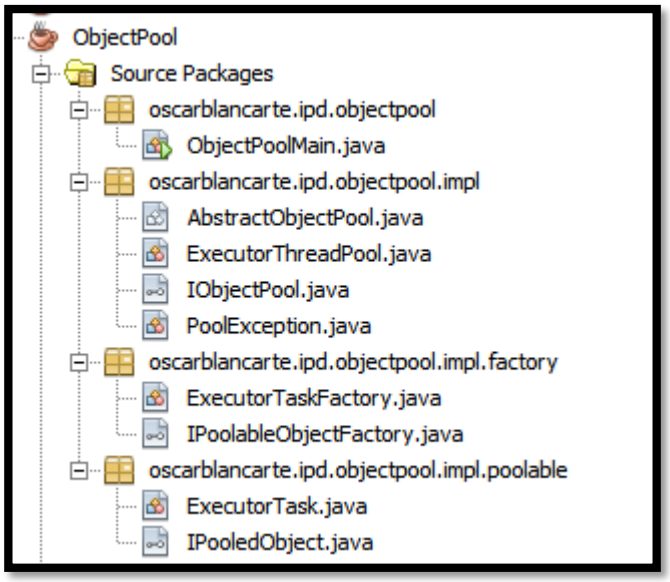


Ilustración 47: Estructura del proyecto ObjectPool.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.objectpool:** Paquete principal del proyecto con la clase ejecutable.
- **oscarblancarte.ipd.objectpool.impl:** Paquete con las clases bases para la creación del *Object Pool*.
- **oscarblancarte.ipd.objectpool.impl.factory:** Paquete con las clases para la fábrica de objetos.
- **oscarblancarte.ipd.objectpool.impl.poolable**: Paquete con las clases de los objetos administrados por el *Object Pool***.**

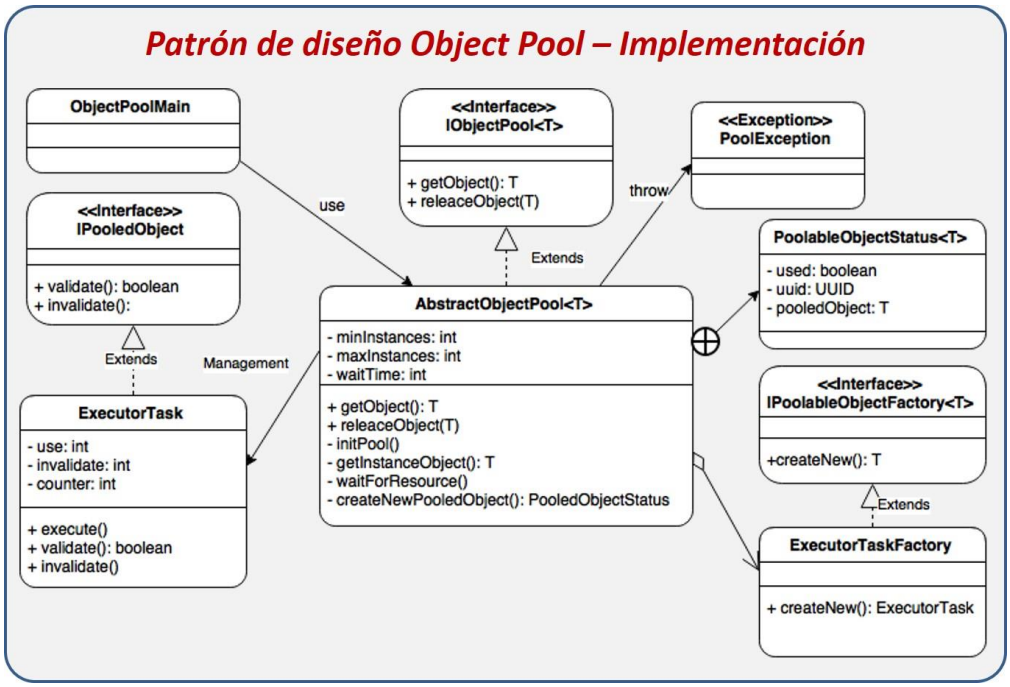


Ilustración 48: Diagrama de clases del proyecto ObjectPool.

Interface IObjectPool:

Esta interface será utilizada para establecer las operaciones mínimas que deberán tener los *ObjectPool*. Una interface no es obligatoria, sin embargo, lo hemos desarrollado así pensando que se puede reutilizar para crear *ObjectPool* más personalizados en el futuro.

La interface define un tipo genérico *T*, el cual deberá heredar de *IPooledObject*, este tipo genérico lo sobrescribiremos cuando creeremos nuestro *ConcreteObjectPool*.

```
1. package oscarblancarte.ipd.objectpool.impl.poolable;
2.
3. public interface IObjectPool<T extends IPooledObject> {
4.     public T getObject() throws PoolException;
5.     public void releaceObject(T pooledObject);
6. }
```

Los métodos definidos son los siguientes:

- **getObject:** Método que utilizaremos para solicitar un objeto del *pool*. Este método lanza la excepción *PoolException* tras agotarse el tiempo de espera para obtener un objeto disponible.
- **releaceObject:** Método utilizado para regresar un objeto al *pool* una vez que ya no sea requerido.

Clase PoolException:

Excepción personalizada para lanzar cualquier error dentro del *ObjectPool*, esta clase hereda de *Exception*.

```
1. package oscarblancarte.ipd.objectpool.impl;
2.
3. public class PoolException extends Exception {
4.
5.     public PoolException(String message) {
6.         super(message);
7.     }
8. }
```

Clase AbstractObjectPool:

Esta es la clase más importante de todas, ya que a pesar de ser abstracta, contiene el funcionamiento básico de un *ObjectPool*, esta clase implementa la interface *ConcreteObjectPool* y servirá como base para crear *ConcreteObjectPool*.

Las propiedades de la clase son las siguientes:

- **maxInstances:** Esta variable determina el número máximo de instancias que debe tener el *Object Pool*, este valor es recibido en el constructor.
- **waitTime:** Variable que almacena el tiempo que un proceso puede esperar por un objeto, este valor es recibido en el constructor.
- **poolableObjectFactory:** Representa al *Factory* utilizado para crear los nuevos objetos, este objeto es recibido en el constructor.

- **fullStack:** Pila con todos los objetos administrados por el *Object Pool*, aquí se encuentra tanto los que están disponibles como los que están ocupados.
- **useStack:** Pila que tiene únicamente los objetos que son utilizados.
- **freeStack:** Pila que tiene únicamente los objetos que están disponibles.

Los métodos son los siguientes:

- **initPool:** Método utilizado para inicializar el *Object Pool* con los objetos mínimos, delimitado por la variable *minInstance*.
- **getInternalObject:** Método utilizado para recuperar una instancia disponible, si no hay intenta crear una nueva, de no poder obtener una instancia entonces regresa *null*.
- **getObject:** Método utilizado por los procesos para obtener un objeto del *pool*, este método bloquea la ejecución hasta que es posible obtener un objeto o hasta que se da un *TimeOut* definido por el parámetro *waitTime*.
- **waitForResource:** Método auxiliar que se encarga de bloquear los procesos hasta que se encuentra un objeto disponible o se da un *TimeOut*, este método es utilizado por el método *getObject*.
- **releaceObject:** Método utilizado para regresar un objeto al *pool*, este método es invocado por los consumidores del *pool* una vez que ya no necesitan más al objeto y lo desean regresar.
- **createNewPoolableObject:** Método utilizado para crear un objeto nuevo mediante el uso del *IObjectFactory*, el cual es enviado como parámetro en la creación del *pool* y es almacenado en la variable *poolableObjectFactory*.
- **toString:** Imprime la información de los objetos únicamente con fin ilustrativo.

```
1.  package oscarblancarte.ipd.objectpool.impl;
2.
3.  import java.util.Calendar;
4.  import java.util.Stack;
5.  import java.util.UUID;
6.  import java.util.concurrent.LinkedBlockingDeque;
7.  import java.util.concurrent.TimeUnit;
```

```
8.
9.  public abstract class AbstractObjectPool<T extends IPooledObject>
10.     implements IObjectPool<T> {
11.
12.     private final int minInstances;
13.     private final int maxInstances;
14.     private final int waitTime;
15.
16.     private final IPoolableObjectFactory<T> poolableObjectFactory;
17.
18.     private final Stack<PooledObjectStatus<T>> fullStack = new Stack<>();
19.     private final Stack<PooledObjectStatus<T>> useStack = new Stack<>();
20.     private final Stack<PooledObjectStatus<T>> freeStack = new Stack<>();
21.
22.     public AbstractObjectPool(int minInstances, int maxInstances, int waitTime
23.             , IPoolableObjectFactory<T> poolableObjectFactory) {
24.         System.out.println("========== STARTING ============");
25.         this.minInstances = minInstances;
26.         this.maxInstances = maxInstances;
27.         this.waitTime = waitTime;
28.         this.poolableObjectFactory = poolableObjectFactory;
29.         initPool();
30.         System.out.println("========== FINISH ============");
31.         System.out.println();
32.     }
33.
34.     private void initPool() {
35.         for (int c = fullStack.size(); c < minInstances; c++) {
36.             PooledObjectStatus<T> createNewPooledObject = createNewPooledObject();
37.             freeStack.push(createNewPooledObject);
38.         }
39.     }
40.
41.     private static class PooledObjectStatus<T> {
42.
43.         boolean used;
44.         UUID uuid;
45.         T pooledObject;
46.
47.         public PooledObjectStatus(T pooledObject) {
48.             this.used = false;
49.             this.uuid = UUID.randomUUID();
50.             this.pooledObject = pooledObject;
51.         }
52.     }
53.
54.     private T getInternalObject() throws PoolException {
55.         synchronized (freeStack) {
56.             if (!freeStack.isEmpty()) {
57.                 PooledObjectStatus<T> first = this.freeStack.pop();
58.                 first.used = true;
59.                 System.out.println("Provisioning1 Object > "
60.                         + first.uuid.toString());
```

```
61.             useStack.push(first);
62.             return first.pooledObject;
63.         }
64.         synchronized (fullStack) {
65.             if (fullStack.size() < maxInstances) {
66.                 PooledObjectStatus<T> returnObject = createNewPooledObject();
67.                 returnObject.used = true;
68.                 System.out.println("Provisioning2 Object > "
69.                         + returnObject.uuid.toString());
70.                 useStack.push(returnObject);
71.                 return returnObject.pooledObject;
72.             } else {
73.                 return null;
74.             }
75.         }
76.     }
77. }
78.
79. @Override
80. public T getObject() throws PoolException {
81.     T internalObject = getInternalObject();
82.     if (internalObject != null) {
83.         return internalObject;
84.     }
85.     return waitForResource();
86. }
87.
88. private T waitForResource() throws PoolException {
89.     Calendar future = Calendar.getInstance();
90.     future.add(Calendar.MILLISECOND, waitTime);
91.     do {
92.         PooledObjectStatus<T> returnObject = null;
93.         synchronized (freeStack) {
94.             if (!freeStack.isEmpty() && !freeStack.peek().used) {
95.                 returnObject = freeStack.pop();
96.                 returnObject.used = true;
97.                 useStack.push(returnObject);
98.                 System.out.println("Provisioning3 Object > "
99.                         + returnObject.uuid.toString());
100.                 return returnObject.pooledObject;
101.             }
102.         }
103.
104.         if (returnObject == null || returnObject.used) {
105.             if (waitTime != 0 && System.currentTimeMillis()
106.                     >= future.getTimeInMillis()) {
107.                 throw new PoolException("Tiempo de espera agotado");
108.             }
109.             try {
110.                 Thread.sleep(1000);
111.             } catch (Exception e) {
112.             }
113.         }
```

```
114.
115.            } while (true);
116.        }
117.
118.        private PooledObjectStatus<T> createNewPooledObject() {
119.            T newObject = poolableObjectFactory.createNew();
120.            PooledObjectStatus<T> pooled = new PooledObjectStatus<>(newObject);
121.            fullStack.push(pooled);
122.            System.out.println("New PoolableObject{UUID=" + pooled.uuid.toString()
123.                    + ", poolSize=" + fullStack.size() + "}");
124.            return pooled;
125.        }
126.
127.        @Override
128.        public void releaceObject(T pooledObject) {
129.            for (PooledObjectStatus<T> item : this.fullStack) {
130.                if (item.pooledObject == pooledObject) {
131.                    if (pooledObject.validate()) {
132.                        freeStack.push(item);
133.                        useStack.remove(item);
134.                        item.used = false;
135.                        System.out.println("Object returned > "
136.                                + item.uuid.toString());
137.                        return;
138.                    } else {
139.                        System.out.println("Object Invalidate ==> "
140.                                + item.uuid.toString());
141.                        pooledObject.invalidate();
142.                        fullStack.remove(item);
143.                        useStack.remove(item);
144.                        synchronized(freeStack){
145.                            initPool();
146.                        }
147.                        return;
148.                    }
149.                }
150.            }
151.        }
152.
153.        @Override
154.        public String toString() {
155.            return "AbstractObjectPool ==> currentSize > '"+fullStack.size()+"', "
156.                    + "free > '"+freeStack.size()+"', used > '"+useStack.size()+"'";
157.        }
158.    }
```

El método *getObject* será el más importante, debido a que nos permitirá recuperar un objeto del *pool*, este método tiene tres escenarios; el primero es cuando el *pool* tiene objetos disponibles en la Pila *freeStack*, de ser así entonces procede a

regresar el primer objeto a la Pila y pasarlo a la Pila *useStack*, el segundo escenario es cuando no tiene objetos disponibles pero el número de objetos no alcanza su tope máximo (*maxInstances*), en este caso el *pool* procede a crear una nueva instancia y regresarla al consumidor (estos dos escenarios los tratamos en el método *getInternalObject*), el tercer escenario es cuando el *pool* no tiene objetos disponibles pero además llegó al tope máximo de instancias, en este caso, el *pool* procede a poner al consumidor en espera hasta que tenga un objeto disponible, este escenario lo tratamos mediante el método *waitForResource*, encargado de obtener un objeto de *freeStack*; de no conseguirlo, valida que el tiempo de espera no se haya superado. En otro caso, el hilo se duerme un segundo para posteriormente repetir los pasos anteriores hasta que se termina el tiempo de espera o se obtiene una instancia disponible.

El segundo método importante es *releaceObject*,el cual será utilizado para liberar los objetos una vez que son desocupados por los consumidores. Este método recibe como parámetro el objeto a regresar, sin embargo, éste antes de regresarlo al *pool*, invoca al método *validate*, con el fin de determinar si el objeto sigue sirviendo al *pool*, de seguir sirviendo entonces el objeto es regresado al *pool*, en otro caso, el objeto es invalidado mediante el método *invalide* con el fin de realizar alguna tarea importante antes de desechar el objeto, como podrá ser cerrar conexiones o liberar algún recurso que estuviera abierto.

Clase ExecutorThreadPool:

Esta clase representa un *Object Pool* terminado, la clase hereda de *AbstractObjectPool* lo que indica que ya tiene toda la funcionalidad por *default* para crear un *pool*. El motivo por el cual creamos esta clase es para tener un *pool* exclusivo de *ExecutorTask*.

```
1. package oscarblancarte.ipd.objectpool.impl;
2.
3. public class ExecutorThreadPool extends AbstractObjectPool<ExecutorTask>{
4.
5.     public ExecutorThreadPool(int minInstances, int maxInstances, int waitTime,
6.             IPoolableObjectFactory<ExecutorTask> poolableObjectFactory) {
7.         super(minInstances, maxInstances, waitTime, poolableObjectFactory);
8.     }
```

```
9. }
```

Veamos que definimos el tipo genérico como *ExecutorTask* lo que hará que los métodos *getObject* y *releaceObject* regresen y reciban este tipo de objeto.
El constructor únicamente recibe los parámetros mínimos para llamar al constructor de su súper clase *AbstractObjectPool.*

Interface IPoolableObject:

Ésta será la interface que todos los objetos del *pool* deberán implementar, es utilizada para identificar los objetos que podrán ser administrados por el *ObjectPool*.

```
1. package oscarblancarte.ipd.objectpool.impl;
2.
3. public interface IPooledObject {
4.     public boolean validate();
5.     public void invalidate();
6. }
```

En la clase podemos apreciar los siguientes métodos:

- **validate:** Utilizada para validar si el objeto aún es apto para ser reutilizado. Existen objetos que con el tiempo ya no son reutilizables como es el caso de las conexiones a bases de datos que por problemas de conexión o permisos, son cerradas, en estos casos el objeto es inservible y debe ser desechado para crear nuevos.
- **invalidate:** Este método es ejecutado antes de ser eliminado del *pool* tras haber sido validado, este método se encarga de realizar cualquier tarea antes de desechar el objeto, como por ejemplo cerrar conexiones, liberar recursos, etcétera.

Clase ExecutorTask:

Esta clase representa el *ConcretePoolableObject* de nuestro modelo de componentes y serán las instancias de esta clase las que finalmente administre el *Object Pool*.

```
1.  package oscarblancarte.ipd.objectpool.impl.poolable;
2.
3.  public class ExecutorTask implements IPooledObject {
4.      private int uses;
5.      private static int invalidate;
6.      private static int counter;
7.
8.      public void execute() {
9.          int c = ++counter;
10.             uses++;
11.             System.out.println("execute ==> " + c);
12.             try {
13.                 Thread.sleep(5000);
14.             } catch (Exception e) {
15.             }
16.             System.out.println("execute end ==> " + c);
17.         }
18.
19.         @Override
20.         public boolean validate() {
21.             return uses < 2;
22.         }
23.
24.         @Override
25.         public void invalidate() {
26.             invalidate++;
27.             System.out.println("Invalidate Counter ==> " + invalidate);
28.         }
29.     }
```

Se definen las siguientes propiedades:

- **uses:** Contador que lleva el registro de cuántas veces se ha utilizado cada instancia de la clase. Para demostrar la característica de la invalidad de los objetos, sólo permitiremos que cada objeto se pueda utilizar dos veces, después de ésto el objeto será invalidado y eliminado del *Object Pool*.
- **invalidate:** Contador que lleva el registro de cuántos objetos han sido invalidados durante toda la vida del *Object Pool*.
- **Counter:** Lleva el registro del número de procesos que se han ejecutado.

Las propiedades que acabamos de definir son únicamente para demostrar en tiempo de ejecución cómo es que se comporta el *Object Pool*, por lo que no afecta el comportamiento.

Los métodos de esta clase son:

- **execute:** Método que simula la ejecución de un proceso, el cual incrementa el contador (*counter*) de ejecuciones y el número de veces que esta instancia se ha utilizado (*uses*).
- **validate**: Método que se utiliza para verificar si la instancia aún le sirve al *pool*, en este caso como forma de demostración, validamos que la instancia no se haya utilizado más de dos veces, de ser así entonces la instancia ya no servirá y regresará *false*.
- **invalidate**: Una vez que el objeto ha sido validado (*validate*) y éste no le sirve más al *pool*, este método se ejecuta para hacer una acción antes de proceder a eliminar el objeto del *pool*. Aquí sólo imprimimos el número de objetos que han sido invalidados.

Interface IPoolableObjectFactory:

Anteriormente mencionamos que el *ObjectPool* no es el responsable de crear los objetos, en su lugar delega esta responsabilidad a un *factory*, es por eso que esta interface definirá la estructura mínima que éste deberá tener.

```
1. package oscarblancarte.ipd.objectpool.impl.factory;
2.
3. public interface IPoolableObjectFactory<T extends IPooledObject> {
4.     public T createNew();
5. }
```

Lo primero que notamos es que tiene un tipo genérico *T*, el cual deberá ser cualquier subclase de *IPooledObject*, este tipo será sobrescrito con el fin de determinar exactamente qué tipo de objeto creará.

El método *createNew* es llamado por el *ObjectPool* para crear un nuevo objeto cuando sea requerido.

Clase ExecutorTaskFactory:

Esta clase hereda de *IPoolablObjectFactory* y sobrescribe el tipo genérico por *ExecutorTask*, lo cual nos dice que el método *createNew* creará objetos de este tipo.

```
1. package oscarblancarte.ipd.objectpool.impl.factory;
2.
3. public class ExecutorTaskFactory  implements IPoolableObjectFactory<ExecutorTask>{
4.     public ExecutorTask createNew(){
5.         return new ExecutorTask();
6.     }
7. }
```

Para este ejemplo, la creación se hace mediante un simple *new*, sin embargo, existen casos donde necesitaremos de una conexión, en la cual deberán estar los datos de conexión y el *driver* para crearla.

Clase ObjectPoolMain:

Clase principal de la aplicación, la cual utilizaremos para probar el patrón. Lo primero que observamos dentro del método *main* es la creación del *Object Pool* (línea 13), en este ejemplo creamos una instancia de *ExecutorThreadPool* y le indicamos por medio de los parámetros que deberá iniciar con 2 objetos y aumentar hasta un máximo de 6, le indicamos que el tiempo máximo de espera es de 100 segundos y le mandamos una instancia de *ExecutorTaskFactory* como argumento final para que pueda crear nuevos objetos.

Lo que sigue es un ciclo donde enviaremos 10 hilos independientes, estos procesos solicitarán un objeto de ejecución (línea 21) con la finalidad de hacer una tarea específica, el *pool* regresa un objeto (si tiene disponible), de no ser así los deja en espera. Una vez que el proceso recupera un objeto del *pool*, se ejecuta la tarea (línea 22) y al terminar lo regresará al *pool* (línea 23).

Observemos que la solicitud del objeto se encuentra dentro de un bloque *try*, debido a que si el tiempo de espera termina, el *Object Pool* lanzará una excepción y utilizamos el bloque *catch* para demostrarlo.

```
1.  package oscarblancarte.ipd.objectpool;
2.
3.  import java.util.concurrent.ScheduledThreadPoolExecutor;
4.  import java.util.concurrent.ThreadPoolExecutor;
5.  import oscarblancarte.ipd.objectpool.impl.ExecutorTask;
6.  import oscarblancarte.ipd.objectpool.impl.ExecutorTaskFactory;
7.  import oscarblancarte.ipd.objectpool.impl.ExecutorThreadPool;
8.  import oscarblancarte.ipd.objectpool.impl.PoolException;
9.
10. public class ObjectPoolMain {
11.
12.     public static void main(String[] args) {
13.         final ExecutorThreadPool pool =
14.             new ExecutorThreadPool(2, 6, 1000*100, new ExecutorTaskFactory());
15.
16.         for (int c = 0; c < 10; c++) {
17.             new Thread(new Runnable() {
18.                 @Override
19.                 public void run() {
20.                     try {
21.                         ExecutorTask task = pool.getObject();
22.                         task.execute();
23.                         pool.releaceObject(task);
24.                     } catch (PoolException e) {
25.                         System.out.println("Error ==> " + e.getMessage());
26.                         e.printStackTrace();
27.                     }
28.                 }
29.             }).start();
30.         }
31.
32.         try {
33.             System.in.read();
34.             System.out.println(pool);
35.         } catch (Exception e) {
36.             System.out.println("Out disponible");
37.         }
38.
39.     }
40. }
```

Recordemos que el número de instancias máximas, mínimas así como el tiempo de espera; es configurable, así que podemos mover los parámetros para probar diferentes configuraciones. Esto lo podemos hacer desde el constructor del *Object Pool* (línea 14).

La ejecución:

Una vez que hemos terminado de implementar la solución, ejecutaremos la clase *ObjectPoolMain* para obtener el siguiente resultado:

```
1.  =========== STARTING ============
2.  New PoolableObject{UUID=fb923314-f37e-4866-af56-4ab9e27d1b31, poolSize=1}
3.  New PoolableObject{UUID=5e941c08-9aba-42a1-b84a-1a3b32f1443a, poolSize=2}
4.  =========== FINISH ============
5.
6.  Provisioning Object > 5e941c08-9aba-42a1-b84a-1a3b32f1443a
7.  Provisioning Object > fb923314-f37e-4866-af56-4ab9e27d1b31
8.  execute ==> 2
9.  New PoolableObject{UUID=0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f, poolSize=3}
10. Provisioning Object > 0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f
11. execute ==> 3
12. execute ==> 1
13. New PoolableObject{UUID=dbc839c0-7003-4498-8b24-8f99b8c78179, poolSize=4}
14. Provisioning Object > dbc839c0-7003-4498-8b24-8f99b8c78179
15. execute ==> 4
16. New PoolableObject{UUID=6f4ed7f1-943f-481c-b024-ce0332829408, poolSize=5}
17. Provisioning Object > 6f4ed7f1-943f-481c-b024-ce0332829408
18. execute ==> 5
19. New PoolableObject{UUID=a4cb8066-1af7-4a41-8ba7-185f25a4776e, poolSize=6}
20. Provisioning Object > a4cb8066-1af7-4a41-8ba7-185f25a4776e
21. execute ==> 6
22. execute end ==> 3
23. execute end ==> 5
24. execute end ==> 4
25. execute end ==> 2
26. execute end ==> 1
27. Object returned > dbc839c0-7003-4498-8b24-8f99b8c78179
28. Object returned > fb923314-f37e-4866-af56-4ab9e27d1b31
29. execute end ==> 6
30. Object returned > a4cb8066-1af7-4a41-8ba7-185f25a4776e
31. Object returned > 0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f
32. Object returned > 5e941c08-9aba-42a1-b84a-1a3b32f1443a
33. Object returned > 6f4ed7f1-943f-481c-b024-ce0332829408
34. Provisioning Object > 6f4ed7f1-943f-481c-b024-ce0332829408
35. execute ==> 7
36. Provisioning Object > 5e941c08-9aba-42a1-b84a-1a3b32f1443a
37. execute ==> 8
38. Provisioning Object > 0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f
39. execute ==> 9
40. Provisioning Object > a4cb8066-1af7-4a41-8ba7-185f25a4776e
41. execute ==> 10
42. execute end ==> 8
43. Object Invalidate ==> 5e941c08-9aba-42a1-b84a-1a3b32f1443a
44. Invalidate Counter ==> 1
45. execute end ==> 7
46. Object Invalidate ==> 6f4ed7f1-943f-481c-b024-ce0332829408
```

```
47. Invalidate Counter ==> 2
48. execute end ==> 9
49. Object Invalidate ==> 0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f
50. Invalidate Counter ==> 3
51. execute end ==> 10
52. Object Invalidate ==> a4cb8066-1af7-4a41-8ba7-185f25a4776e
53. Invalidate Counter ==> 4
```

Lo primero que hace el *pool* es iniciar las instancias mínimas, en este caso 2 (líneas 1 a 4).
La siguiente parte es más compleja ya que al tratarse de hilos el resultado puede ser distinto en cada ejecución que realicemos, ya que los hilos corren dependiendo de nuestro sistema operativo y la disponibilidad de recursos que éste tenga al momento de la ejecución, por lo que puede que el hilo que iniciamos primero termine después que el último. Es por eso que más que decir qué pasa en cada línea, analizaremos cómo es que debe funcionar.

La siguiente línea, indica que un objeto ha sido prestado a un proceso, la serie de caracteres que viene en el mensaje representan el identificador del objeto, con el fin de saber cuándo se presta y cuando se regresa.

```
1. Provisioning Object > 5e941c08-9aba-42a1-b84a-1a3b32f1443a
```

La siguiente línea indica que un proceso se empieza a ejecutar, el número que representa es el número de proceso; el cual se incrementa con cada ejecución con el fin de identificar los procesos que se han ejecutado.

```
1. execute ==> 1
```

Líneas como la siguiente, indican que un nuevo objeto se ha creado en el *pool*, esto puede ser debido a que todavía no se alcanzaba el máximo de registros o porque un objeto devuelto ha sido invalidado. Cuando el objeto se crea se indica su UUID y el tamaño actual del *pool*.

```
1. New PoolableObject{UUID=0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f, poolSize=3}
```

Los objetos devueltos son regresados al *pool* y el siguiente mensaje es impreso. En el mensaje vemos el UUID del objeto devuelto.

```
1. Object returned > dbc839c0-7003-4498-8b24-8f99b8c78179
```

Cuando un objeto devuelto no pasa satisfactoriamente la validación (método *validate*) este es invalidado y desechado por el *pool*. El siguiente mensaje indica que el objeto fue invalidado e indica su UUID.

```
1. Object Invalidate ==> 6f4ed7f1-943f-481c-b024-ce0332829408
```

Seguido de que un objeto es invalidado, se muestra el siguiente mensaje informativo, el cual indica el número total de objetos que han sido invalidados.

```
1. Invalidate Counter ==> 2
```

El flujo que observaremos es el siguiente:

- Líneas de inicialización del *pool* con las instancias por *default*, las cuales están representadas por las líneas *Provisioning Object >,* veremos tantas como el valor de *minInstances*, seguido de estas líneas veremos otras como estas *execute ==>,* lo que indica que se está corriendo un proceso con el objeto recuperado.
- Cuando no queden más objetos empezaremos a ver mensajes como el siguiente: *New PoolableObject{UUID=0d7f8ec5-cfe8-422d-9996-3b46e7c1d51f, poolSize=3}* lo que indica que se creó un objeto nuevo porque aún no se ha llegado al límite o porque otras instancias fueron invalidadas.

- Seguido, empezaremos a ver *execute end ==>* lo que indica que los procesos que iniciaron están terminando.
- Los procesos que terminen regresarán los objetos al *pool* por lo que veremos los siguientes mensajes: *Object returned >*.
- Después de que termine un cierto número de procesos empezaremos a ver estos mensajes: *Object Invalidate ==> y Invalidate Counter ==>*, esto nos indicará que un objeto fue utilizado dos veces y será desechado.

Bajo esta premisa, es importante analizar los mensajes para determinar qué está pasando, algunas de las cosas que no deberíamos ver y que indica que nuestro *pool* está mal son las siguientes:

- Ver más de una vez el siguiente mensaje *provisioning Object >* antes de ver que el objeto ha sido devuelto al *pool* con el mensaje *Object returned >*.
- El número de procesos que están corriendo simultáneamente es mayor que el *maxInstances*.
- El número de objetos en el *fullStack* es mayor que el *maxInstances*
- La suma de los objetos en el *useStack* y *freeStack* es mayor que *fullStack*.

Si seguimos estas instrucciones seremos capaces de entender cómo es que los resultados se arrojan en pantalla.

- Es importante que juguemos con los valores de *maxInstance*, *minInstances* y *waitTime* para que entendamos mejor cómo funciona, como el número de combinación de resultados es infinito nos llevaremos de tarea mover estos valores y analizar lo que está pasando.

Siguientes pasos:

Hemos demostrado la utilidad de un *Object Pool* en cosas tan simples como administrar procesos, pero los *pools* tiene muchas más aplicaciones interesantes.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios.

- Modifiquemos la aplicación para que los objetos sean invalidados tras tener más de 2 minutos de haber sido creados.

- Hemos hablado acerca de que nuestro *Object Pool* tiene un mínimo y un máximo de instancias, las cuales indican el número de objetos que deberá tener el *pool* al inicio y el número máximo de instancias que podrá crear, sin embargo, un tema de performance podría ser mantener el número máximo de instancias creadas todo el tiempo, por lo cual modificaremos el programa para eliminar aquellas instancias que tiene un tiempo determinado sin utilizarse.

- Utilicemos el *AbstracObjectPool* que hemos creado para generar un nuevo *Object Pool* de conexiones a base de datos, para lo cual tendremos que crear una subclase *IPooledObject* para las conexiones y un *IPoolableObjectFactory* para crearlas. Recordemos que tendremos que abrir la conexiones al momento de crear los objetos y de cerrar la conexión cuando invalidemos el objeto.

Patrones Estructurales

Los patrones de diseño estructurales describen cómo los objetos y clases se pueden combinar para formar estructuras más grandes y complejas ayudando a estructurar los objetos, las relaciones y la herencia entre las clases; de tal forma que facilitan la adaptación entre interfaces, agregar y gestionar objetos en tiempo de ejecución.

Este tipo de patrones nos ayudan a crear estructuras de clases muy versátiles ya que se basan en la composición en lugar de crear estructuras estáticas. Serán especialmente útiles si no se tiene muy claro cómo debemos de crear estructuras de objetos complejas, las cuales están compuestas de otras clases y que además deben trabajar de forma homogénea en tiempo de ejecución. Algunos de los patrones que aquí se describen son tan básicos que muchas veces los utilizamos sin darnos cuenta.

Los patrones que abordaremos en este capítulo son:

Patrón Adapter: Resuelve escenarios donde existen interfaces incompatibles creando adaptadores que nos ayuden a comunicarnos con las interfaces de una forma genérica.

Patrón Bridge: Patrón utilizado para desacoplar una abstracción de su implementación permitiendo la variación entre ellas.

Patrón Composite: Patrón de diseño que nos permite la creación de estructuras de objetos complejas mediante la agregación.

Patrón Decorator: Patrón de diseño utilizado para agregar funcionalidad a un objeto mediante la asociación de otros objetos. Este patrón nos evita la creación de clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

Patrón Facade: Mediante la implementación de este patrón, se pueden crear fachadas que permitan ocultar la complejidad de interactuar con un conjunto de

subsistemas, mediante la implementación de interfaces de alto nivel las cuales se encarguen de realizar la comunicación con todos los subsistemas.

Patrón Flyweight: Patrón enfocado en la creación de objetos ligeros. Los objetos ligeros se logran mediante la abstracción de partes reutilizables que son compartidos por otros objetos en vez de crear nuevos cada vez que son requeridos.

Patrón Proxy: Patrón de diseño utilizado para la mediación entre un objeto y otro. Se conoce como mediación a las acciones que se pueden hacer antes y después de realizar la acción solicitada.

Patrón Adapter

El patrón de diseño *Adapter* es utilizado cuando tenemos interfaces de software incompatibles, las cuales a pesar de su incompatibilidad tiene una funcionalidad similar. Este patrón es implementado cuando se desea homogeneizar la forma de trabajar con estas interfaces incompatibles, para lo cual se crea una clase intermedia que funciona como un adaptador. Esta clase adaptador proporcionará los métodos para interactuar con la interface incompatible.

Este patrón es muy similar a los adaptadores de corriente o de video que convierten una entrada en otra como por ejemplo de Micro-HDMI a HDMI o Jack M de 3.5mm (Audio) a USB. En el caso del software el concepto es igual, creamos clases que convierten una entrada o salida a otra para poder interactuar con una clase.

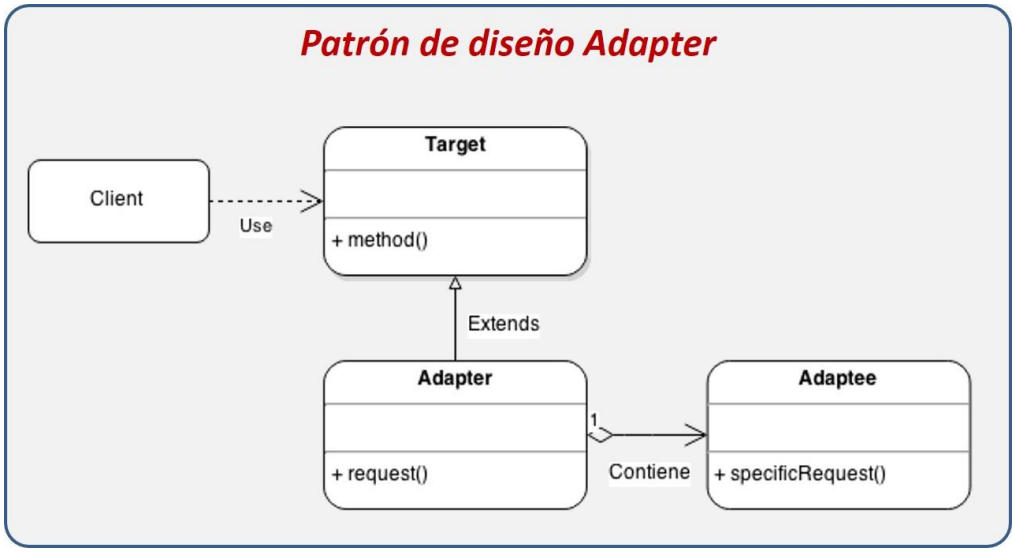


Ilustración 49: Estructura del patrón Adapter.

Los componentes que conforman el patrón son los siguientes:

- **Client:** Actor que interactua con el *Adapter*.

- **Target:** Interface que nos permitirá homogenizar la forma de trabajar con las interfaces incompatibles, esta interface es utilizada para crear los *Adapter*.
- **Adapter:** Representa la implementación del *Target*, el cual tiene la responsabilidad de mediar entre el *Client* y el *Adaptee*. Oculta la forma de comunicarse con el *Adaptee*.
- **Adaptee**: Representa la clase con interface incompatible.

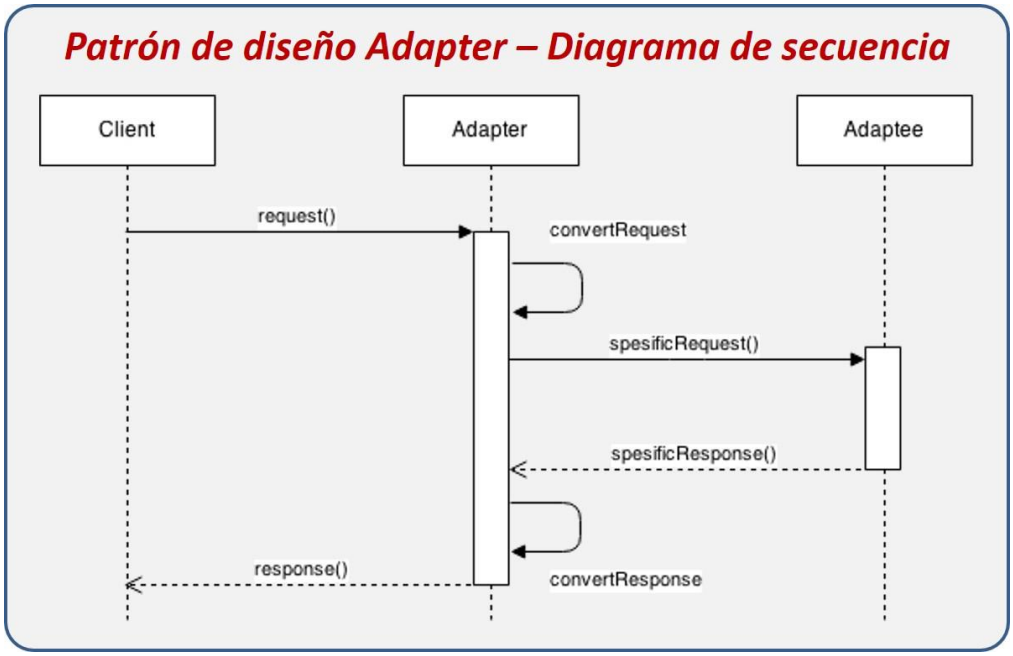


Ilustración 50: Diagrama de secuencia del patrón Adapter.

El diagrama anterior se interpreta de la siguiente manera:

1. El *Client* invoca al *Adapter* con parámetros genéricos.
2. El *Adapter* convierte los parámetros genéricos en parámetros específicos del *Adaptee*.
3. El *Adapter* invoca al *Adaptee*.
4. El *Adaptee* responde.
5. El *Adapter* convierte la respuesta del *Adaptee* a una respuesta genérica para el *Client*.

6. El *Adapter* responde al *Client* con una respuesta genérica.

Cuándo utilizarlo:

- Cuando buscamos homogeneizar la forma de trabajar con interfaces incompatibles mediante una interface en común.
- Cuando las interfaces del sistema viejo no se ajustan al nuevo sistema.

El escenario:

Para este ejemplo imaginaremos que tenemos un sistema encargado de realizar préstamos personales por medio de bancos. Nuestro sistema lo que hará es utilizar las API's proporcionadas por los bancos para comunicarnos con ellos, esto con el fin de validar si el banco le puede prestar la cantidad solicitada, el sistema se tendrá que ir a los dos bancos para validar esta información y decirle al cliente si es posible prestarle el efectivo.

Sin embargo, no todo es tan simple ya que cada banco brinda sus propias API´s para comunicarse con ellos. El primer banco **XBank** proporciona una interface síncrona y el segundo banco **YBank** proporciona una interface asíncrona. Cada API en esencia requiere la misma información para hacer la solicitud, sin embargo, el nombre de las clases, variables y tipos de datos son distintos entre cada API´s.

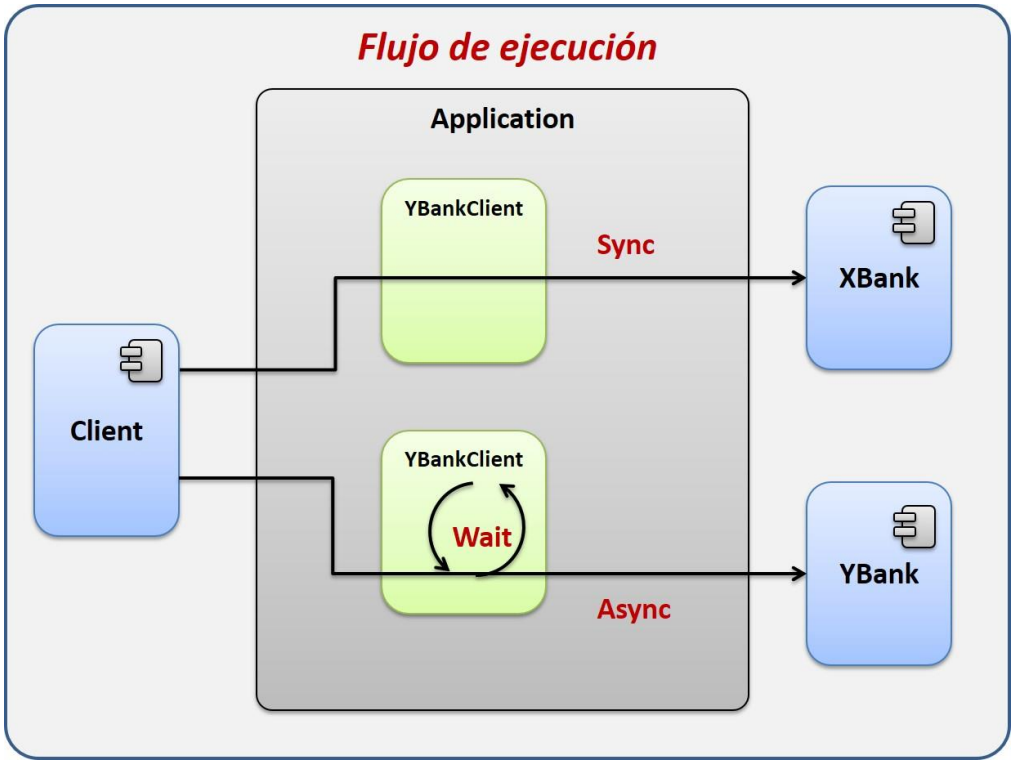


Ilustración 51: Flujo de ejecución de las dos API's.

Como vemos en la imagen, un banco recibe peticiones síncronas y otra asíncronas. Los tipos y nombre de los parámetros son distintos por lo que es imposible ejecutarlos de la misma manera, esto nos obliga a realizar dos procesos independientes de invocación, donde cada proceso se adecúa a las características del API.

Nuestro trabajo consiste en implementar un mecanismo que nos permita ejecutar las dos API´s de una forma genérica, de tal forma que el cliente siempre envié y reciba lo mismo sin importar que API utilice. Esta solución también debe ser compatible con nuevos bancos que se puedan agregar en el futuro.

La solución:

Para solucionar este tipo de problemas es especialmente útil el patrón Adapter, ya que es un escenario donde tenemos interfaces incompatibles, sin embargo, el funcionamiento de las dos API's es muy parecido.

El primer paso para dar solución a este problema será definir una interface que nos permita interactuar con ambas interfaces de una forma genérica (*Target*), y después crear las implementaciones que se encarguen de interactuar con las API´s bancarias (*Adapter*); estas implementaciones se encargarán de convertir la entrada del *Target* en una entrada válida para el API y enviarlo, una vez que tengamos la respuesta, el *Adapter* tendrá que convertir la respuesta a una respuesta genérica que pueda entender el *Client*.

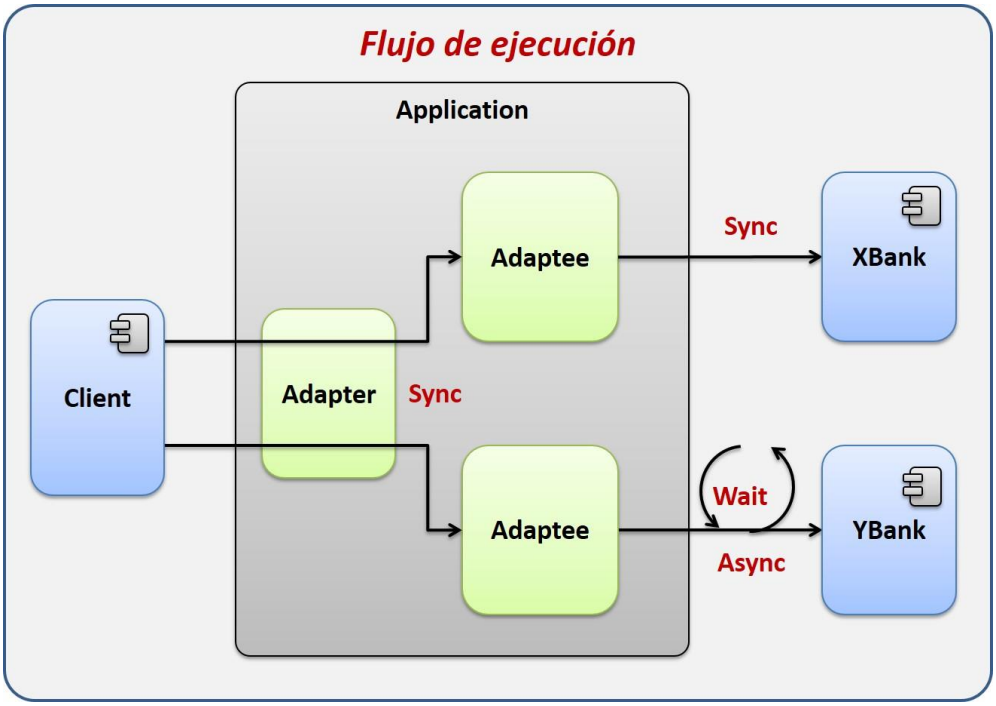


Ilustración 52: Invocando las API's bancarias por medio del patrón Adapter.

En esta imagen podemos apreciar que el *Client* no hace invocaciones directas a las API's bancarias, en su lugar crea un adaptador para cada API con el fin de establecer interfaces compatibles para el *Client*, el cliente consume las API's por

medio de estos adaptadores y se asegura que el mensaje enviado y recibido siempre será el mismo sin importar la implementación que haya detrás del *Adaptee*.

Un aspecto importate que podemos ver en la imagen es que una API es síncrona y la otra es asíncrona, esto presenta un gran problema ya que hace que los *Adaptee* tengan que homologar la forma de invocación, de tal manera que los *Adaptee* tendrán que encargarse de que las respuestas hacia el cliente siempre sean síncronas sin importar el API bancario utilizado.

La implementación:

Una vez comprendido para qué sirve el patrón de diseño y el problema a solucionar, iniciaremos con la implementación.

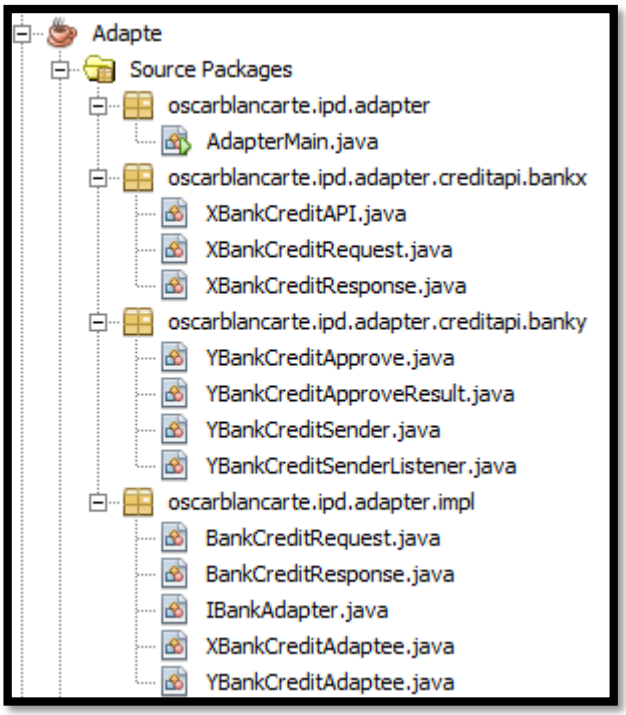


Ilustración 53: Estructura del proyecto Adapter.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.adapter:** Contiene únicamente la clase principal del proyecto.
- **oscarblancarte.ipd.adapter.creditapi.bankx:** Contiene las clases correspondientes al API del banco XBank
- **oscarblancarte.ipd.adapter.creditapi.banky:** Contiene las clases correspondientes al API del banco YBank
- **oscarblancarte.ipd.adapter.impl:** Contiene las clases de implementación del patrón Adapter.

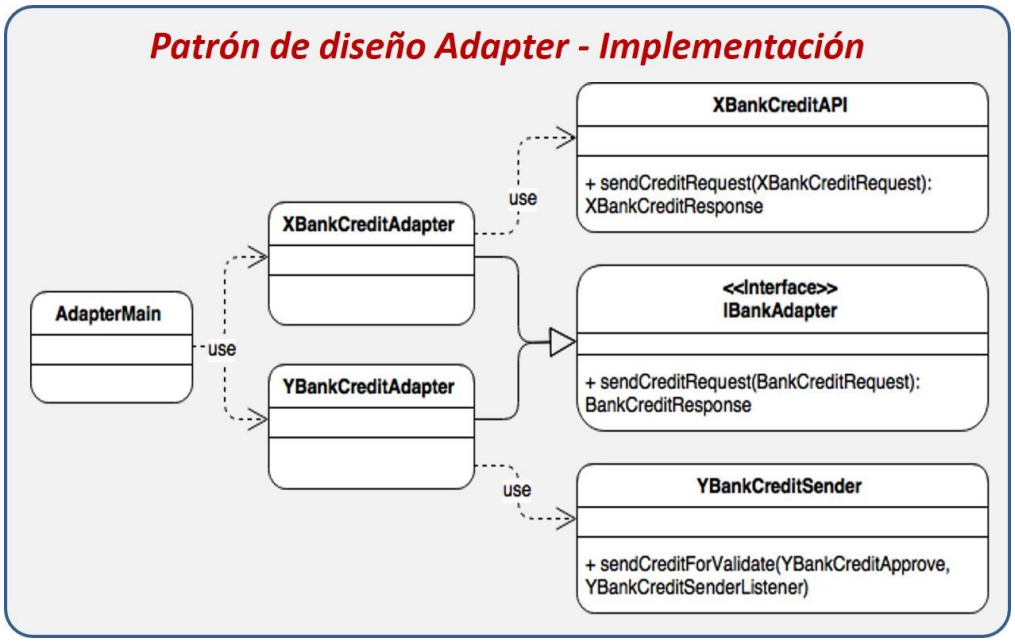


Ilustración 54: Diagrama de clases del proyecto Adapter.

En la imagen anterior podemos apreciar las principales clases del proyecto y cómo es que éstas se relacionan entre sí. Se omiten las clases utilizadas como *Request* y *Response* para hacer más claro el diagrama.

Interface IBankAdapter:

Interface que utilizaremos para definir la estructura de los adaptadores, esta interface representa al *Target* de nuestro modelo de clases.

```
1.  package oscarblancarte.ipd.adapter.impl;
2.
3.  public interface IBankAdapter {
4.      public BankCreditResponse sendCreditRequest(BankCreditRequest request);
5.  }
```

En la interface se define el método *sendCreditRequest*, el cual servirá para solicitar al banco la autorización del crédito. Este método será implementado por

los adaptadores para comunicarse con un banco concreto. Observemos que es síncrono.

Clase BankCreditRequest:

Clase que representa el request genérico utilizado por los adaptadores, éste tendrá que ser convertido al *request* del banco concreto.

```
1.  package oscarblancarte.ipd.adapter.impl;
2.
3.  public class BankCreditRequest {
4.      private String customer;
5.      private double amount;
6.
7.     /** GET and SET **/
8.  }
```

Observemos muy detenidamente los atributos de la clase y los tipos de datos, los atributos son *customer* que representa el nombre del cliente y *amount* representa el monto solicitado.

Clase BankCreditResponse:

Clase que representa la respuesta genérica utilizada por los adaptadores. Cuando el banco responda la solicitud de crédito lo hará con su propia clase de respuesta definida en el API bancario, por lo cual esta respuesta deberá ser convertida a esta clase con la finalidad de que siempre sea igual sin importar el banco al cual se mandó la solicitud.

```
1.  package oscarblancarte.ipd.adapter.impl;
2.
3.  public class BankCreditResponse {
4.      public boolean approved;
5.
6.      /** GET and SET **/
7.  }
```

Observemos que la respuesta es simplemente un *boolean* que indica si el crédito fue aprobado.

Clase XbankCreditAdaptee:

Representa el adaptador entre el *Client* y **XBank**, esta clase se encarga de homologar la forma en que se comunica el *Client* con **XBank**.

```
1.  package oscarblancarte.ipd.adapter.impl;
2.
3.  import oscarblancarte.ipd.adapter.creditapi.bankx.XBankCreditAPI;
4.  import oscarblancarte.ipd.adapter.creditapi.bankx.XBankCreditRequest;
5.  import oscarblancarte.ipd.adapter.creditapi.bankx.XBankCreditResponse;
6.
7.  public class XBankCreditAdaptee implements IBankAdapter{
8.
9.      @Override
10.     public BankCreditResponse sendCreditRequest(BankCreditRequest request) {
11.
12.         XBankCreditRequest xrequest = new XBankCreditRequest();
13.         xrequest.setCustomerName(request.getCustomer());
14.         xrequest.setRequestAmount(request.getAmount());
15.
16.         XBankCreditAPI api = new XBankCreditAPI();
17.         XBankCreditResponse xresponse = api.sendCreditRequest(xrequest);
18.
19.         BankCreditResponse response = new BankCreditResponse();
20.         response.setApproved(xresponse.isAproval());
21.         return response;
22.     }
23. }
```

Esta clase convierte el *request* genérico en un *request* específico para **XBank** y cuando éste responde convierte la respuesta específica de **XBank** en una respuesta genérica para el *Client*.

- De la línea 12 a la 14 se realiza la conversión del *request*, se convierte del formato homogéneo del adaptador al específico del banco.
- En la línea 17 se llama al API para autorizar el crédito.
- De la línea 19 a la 21 se convierte la respuesta específica de **XBank** a una genérica para el *Client* y se regresa.

Clase YBankCreditApproveResult:

Esta clase sirve como adaptador entre el *Client* y el API de **YBank**, la clase hereda de *IBankAdapter* con el fin de mantener una interface homologada entre el *Client* y **YBank**. Además, hereda de *YBankCreditSenderListener* para ser notificado cuando **YBank** conteste de forma asíncrona.

```
1.  package oscarblancarte.ipd.adapter.impl;
2.
3.  import oscarblancarte.ipd.adapter.creditapi.banky.YBankCreditApprove;
4.  import oscarblancarte.ipd.adapter.creditapi.banky.YBankCreditApproveResult;
5.  import oscarblancarte.ipd.adapter.creditapi.banky.YBankCreditSender;
6.  import oscarblancarte.ipd.adapter.creditapi.banky.YBankCreditSenderListener;
7.
8.  public class YBankCreditAdaptee implements IBankAdapter, YBankCreditSenderListener {
9.
10.         private YBankCreditApproveResult yresponse;
11.
12.         @Override
13.         public BankCreditResponse sendCreditRequest(BankCreditRequest request) {
14.             YBankCreditApprove yrequest = new YBankCreditApprove();
15.             yrequest.setCredit((float) request.getAmount());
16.             yrequest.setName(request.getCustomer());
17.
18.             YBankCreditSender sender = new YBankCreditSender();
19.             sender.sendCreditForValidate(yrequest, this);
20.
21.             do {
22.                 try {
23.                     Thread.sleep(10000);
24.                     System.out.println("YBank petición en espera....");
25.                 } catch (Exception e) {
26.                     e.printStackTrace();
27.                 }
28.             } while (yresponse == null);
29.
30.             BankCreditResponse response = new BankCreditResponse();
31.             response.setApproved(yresponse.getApproved() == "Y" ? true : false);
32.             return response;
33.         }
34.
35.         @Override
36.         public void notifyCreditResult(YBankCreditApproveResult result) {
37.             this.yresponse = result;
38.         }
39.     }
```

Podemos apreciar que el método *sendCreditRequest* es síncrono a pesar de que el API bancario es asíncrono, esto se debe a que esta clase espera hasta tener la respuesta de **YBank** para regresarla de forma síncrona. Cuando el *request* es enviado a **YBank**, éste entra en un ciclo hasta que la respuesta de **YBank** llega, la respuesta es notificada por el método *notifyCreditResult*, el cual es llamado por el API cuando la respuesta está lista.

- De las líneas 14 a 16 se prepara el *request* específico para **YBank**.
- En las líneas 18 y 19 se envía la solicitud a **YBank**.
- De la línea 21 a la 28 se pone en espera al hilo hasta que se tiene una respuesta de **YBank**.
- La línea 37 se ejecuta cuando **YBank** responde y con esto se rompe el ciclo anterior.
- De las líneas 30 a 32 se convierte la respuesta específica de **YBank** a una respuesta genérica para el *Client*.

XBank API - XBankCreditAPI:

Esta es la clase que el banco **XBank** nos proporciona para enviar las solicitudes de crédito.

```
1.  package oscarblancarte.ipd.adapter.creditapi.bankx;
2.
3.  public class XBankCreditAPI {
4.
5.      public XBankCreditResponse sendCreditRequest(XBankCreditRequest request){
6.          XBankCreditResponse response =new XBankCreditResponse();
7.          if(request.getRequestAmount()<= 5000){
8.              response.setAproval(true);
9.          }else{
10.             response.setAproval(false);
11.         }
12.         return response;
13.     }
14. }
```

Veamos el método llamado *sendCreditRequest*, que es síncrono, recibe como entrada un tipo llamado *XBankCreditRequest*, regresa un tipo llamado *XBankCreditResponse*, la lógica para autorizar un crédito es que el monto no sea mayor de $5000.

XBank API - XBankCreditRequest:

Esta clase representa el *request* necesario para enviar una solicitud a **XBank**. Veamos con detalle el nombre y tipo de datos de los atributos de la clase.

```
1.  package oscarblancarte.ipd.adapter.creditapi.bankx;
2.
3.  public class XBankCreditRequest {
4.      private String customerName;
5.      private double requestAmount;
6.
7.   /** GET and SET **/
8.  }
```

Los atributos que tenemos son: *customerName* (nombre del cliente) y *requestAmount* (monto solicitado).

XBank API - XBankCreditResponse:

Clase que representa la respuesta de **XBank**, la cual mediante un *boolean* indica si el crédito está aprobado.

```
1.  package oscarblancarte.ipd.adapter.creditapi.bankx;
2.
3.  public class XBankCreditResponse {
4.      public boolean aproval;
5.
6.      /** GET and SET **/
7.  }
```

YBank API - YBankCreditSender:

Esta clase representa el API que nos proporciona **YBank** para enviar las solicitudes de crédito.
Analicemos esta API: tiene un método llamado *sendCreditForValidate*, este método no regresa nada (*void*), sin embargo, recibe como parámetro una clase *listener* a la cual notificará cuando tenga una respuesta, lo que lo hace un servicio con una respuesta asíncrona, el método recibe como parámetro una clase llamada *YBankCreditApprove* y el *listener* *YBankCreditSenderListener*.

```
1.  package oscarblancarte.ipd.adapter.creditapi.banky;
2.
3.  public class YBankCreditSender {
4.
5.      public void sendCreditForValidate(final YBankCreditApprove request,
6.          final YBankCreditSenderListener listener) {
7.
8.          new Thread(new Runnable() {
9.              @Override
10.             public void run() {
11.                 System.out.println("YBank recibió  la solicitud en un
12.                     momento y tendrá la respuesta, sea paciente por favor");
13.                 YBankCreditApproveResult response = new YBankCreditApproveResult();
14.                 if (request.getCredit() <= 1500) {
15.                     response.setApproved("Y");
16.                 } else {
17.                     response.setApproved("N");
18.                 }
19.                 try {
20.                     Thread.sleep(1000 * 30);
21.                 } catch (Exception e) {
22.                     e.printStackTrace();
23.                 }
24.
25.                 listener.notifyCreditResult(response);
26.             }
27.         }).start();
28.     }
29. }
```

Cuando mandamos la solicitud, el API la ejecuta en un hilo independiente y no regresa nada de momento, cuando el banco tiene la respuesta notifica al *Client* mediante la interface *YBankCreditSenderListener* (línea 25), el API dará el crédito

si el monto solicitado no excede los $1500, este API regresa **'Y'** en caso de que el crédito fue aprobado y **'N'** cuando el crédito fue rechazado. Observemos que a diferencia del **XBank**, el cual regresa un booleano, **YBank** regresa con un *String* si se le autoriza el crédito.

YBank API - YBankCreditSenderListener:

Interface que deberá ser implementada para ser notificado cuando la respuesta asíncrona sea enviada por **YBank**.

```
1.  package oscarblancarte.ipd.adapter.creditapi.banky;
2.
3.  public interface YBankCreditSenderListener {
4.      public void notifyCreditResult(YBankCreditApproveResult result);
5.  }
```

El método *notifyCreditResult* es invocado cuando el resultado está disponible, la respuesta se envía bajo la clase *YBankCreditApproveResult*.

YBank API - YBankCreditApprove

Clase que representa el *request* de **YBank** para la solicitud de crédito, la clase recibe el *name* (nombre del cliente) y *credit* (monto del crédito).

```
1.  package oscarblancarte.ipd.adapter.creditapi.banky;
2.
3.  public class YBankCreditApprove {
4.      public String name;
5.      public float credit;
6.
7.      /** GET and SET **/
8.  }
```

YBank API – YbankCreditApproveResult

La clase representa la respuesta de la solicitud de crédito de ***YBank***.

```
1.  package oscarblancarte.ipd.adapter.creditapi.banky;
2.
3.  public class YBankCreditApproveResult {
4.
5.     public String approved;
6.
7.     /**  GET and SET **/
8.  }
```

Clase AdapterMain:

Finalmente, en esta clase podemos probar el resultado de todo lo explicado anteriormente, ésta la utilizaremos para probar la implementación.

```
1.  package oscarblancarte.ipd.adapter;
2.
3.  import oscarblancarte.ipd.adapter.impl.BankCreditRequest;
4.  import oscarblancarte.ipd.adapter.impl.BankCreditResponse;
5.  import oscarblancarte.ipd.adapter.impl.IBankAdapter;
6.  import oscarblancarte.ipd.adapter.impl.XBankCreditAdaptee;
7.  import oscarblancarte.ipd.adapter.impl.YBankCreditAdaptee;
8.
9.  public class AdapterMain {
10.
11.    public static void main(String[] args) {
12.       //Request genérico para las dos API's
13.       BankCreditRequest request = new BankCreditRequest();
14.       request.setCustomer("Oscar Blancarte");
15.       request.setAmount(1000);
16.
17.       IBankAdapter xBank = new XBankCreditAdaptee();
18.       BankCreditResponse xresponse = xBank.sendCreditRequest(request);
19.       System.out.println("xBank approved > " + xresponse.isApproved() + "\n");
20.
21.       IBankAdapter yBank = new YBankCreditAdaptee();
22.       BankCreditResponse yresponse = yBank.sendCreditRequest(request);
23.       System.out.println("yBank approved > " + yresponse.isApproved() + "\n");
24.
25.       if (xresponse.isApproved()) {
26.          System.out.println("xBank aprobó su crédito, felicidades!!");
27.       } else if (yresponse.isApproved()) {
28.          System.out.println("yBank aprobó su crédito, felicidades!!");
29.       } else {
30.          System.out.println("lo sentimos su crédito no ha sido aprobado");
31.       }
32.    }
33. }
```

- Observemos que de las líneas 11 a la 15 creamos un *request* para solicitar un crédito para nuestro cliente.
- En la línea 17 a la 19 invocamos al *Adapter* de **XBank**, el cual recibe nuestro *request* genérico y nos regresa una respuesta también genérica.
- De las líneas 21 a la 23 enviamos el mismo *request* genérico al *Adapter* de **YBank** el cual nos regresa una respuesta genérica.
- De las líneas 25 a la 31 validamos las respuestas de solicitud de crédito del cliente y le damos los resultados.

En ningún momento vimos que una llamada es síncrona y la otra asíncrona ya que esta lógica la encapsula el *Adapter*, además que el *request* que le enviamos a los dos adaptadores es el mismo, la respuesta que tenemos de los dos adaptadores también es la misma.
De esta manera el cliente ha consumido las dos API's de una forma completamente homogénea y se ha solucionado el problema de incompatibilidad de interfaces.

La ejecución:

Para probar la aplicación, ejecutaremos la clase *AdapterMain* y tendremos el siguiente resultado:

```
run:
xBank approved > true

yBank recibió su la solicitud en un momento tendrá la respuesta, sea paciente porfavor
yBank peticion en espera....
yBank peticion en espera....
yBank peticion en espera....
yBank approved > true

xBank aprobó su crédito, felicidades!!
BUILD SUCCESSFUL (total time: 30 seconds)
```

Ilustración 55: Resultado de la ejecución de la clase AdapterMain.

Lo primero que veremos será la autorización inmediata de **XBank** que tiene una implementación síncrona; seguido, veremos que la solicitud a **YBank** ha sido enviada y nos pedirá que esperemos, el hilo se quedará en espera y nos dirá que la petición está en espera cada 10 segundos, finalmente veremos la respuesta de **YBank**. Con las dos respuestas le diremos el resultado al cliente.

Con este simple ejemplo hemos visto el poder de este patrón para trabajar con interfaces incompatibles, hemos visto que las clases se llaman distinto, los tipos de datos son diferentes y la respuestas son asíncronas y síncronas, sin embargo, nada de esto importó ya que por medio de *Adapter* pudimos crear una estructura homogénea, que para el cliente fue totalmente transparente.

Siguientes pasos:

Como clave recordemos que este patrón trata de solucionar escenarios donde tenemos interfaces incompatibles pero que comparten una funcionalidad muy parecida.

Para fortalecer los conocimientos obtenidos realicemos los siguientes ejercicios:

- Agréguenos un tercer banco llamado **ZBank** que tenga una respuesta síncrona, el API deberá regresar verdadero cuando se le autorice el crédito, pero deberá lanzar una excepción cuando el crédito no sea autorizado.

- Implementemos el patrón Factory Method para crear el *Adaptador concrete* por medio del nombre del banco.

- Creemos una aplicación nueva que tenga dos clases para consultar los datos de los empleados de una empresa. Una clase deberá consultar los datos directamente sobre la base de datos y la otra deberá leer los datos mediante un servicio SOA o REST, sin embargo, las dos clases deberán recibir el nombre del empleado a buscar y las firmas de los métodos deberán ser distintas. Al final, implementaremos el patrón *Adapter* para hacer compatibles las dos clases y nuestra aplicación pueda cambiar de implementación sin afectar al programa.

Patrón Bridge

Bridge es un patrón de diseño utilizado para desacoplar una abstracción de su implementación de manera que las dos puedan ser modificadas por separado sin necesidad de modificar la otra; dicho de otra manera, se desacopla una abstracción de su implementación para que puedan variar independientemente.

Bridge es muy utilizado cuando dos piezas de software están relacionadas directamente, sin embargo, existe una gran probabilidad de que una de las dos partes cambie y esto puede llevar a la necesidad de modificar la pieza del otro lado. Para solucionar este problema *Bridge* propone crear una estructura de clases basada en la agregación, en la cual, una clase puente es utilizada para desacoplar la clase que queremos utilizar y el cliente, de tal forma que este último no conozca la clase destino, permitiendo que cualquier parte pueda cambiar sin afectar a la otra.

Este patrón puede ser confundido con *Adapter* ya que incluso *Bridge* se implementa utilizando el patrón *Adapter*, sin embargo, existe una diferencia entre los dos, la cual es clave para distinguirlo, y es que *Bridge* se centra en desacoplar la abstracción de la implementación por lo que tiene sentido utilizarse cuando existen interfaces compatibles. En cambio *Adapter* es utilizada para resolver escenarios donde las interfaces no son compatibles.

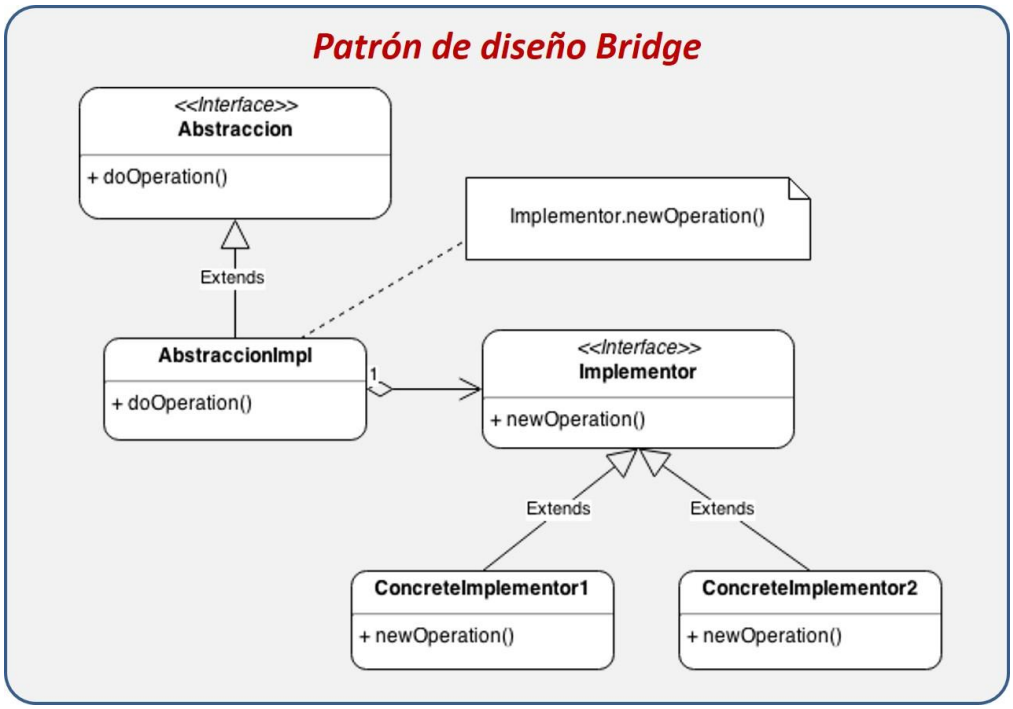


Ilustración 56: Estructura del patrón Bridge.

Los componentes se explican a continuación:

- **Abstraction:** Interface que define la estructura de la clase adaptadora o *bridge*.
- **AbstractionImpl:** Clase utilizada como puente para desacoplar a *Abstraction* de *Implementor*, la clase hereda de *Abstraction*.
- **Implementor:** Define una estructura de clase común para todas los *ConcreteImplementor*. Esta interface no es estrictamente requerida para implementar el patrón.
- **ConcreteImplementor:** Conjunto de clases que heredan de *Implementor* y que son propensas a cambiar, es por esta razón que se opta por implementar el patrón de diseño *Bridge*.

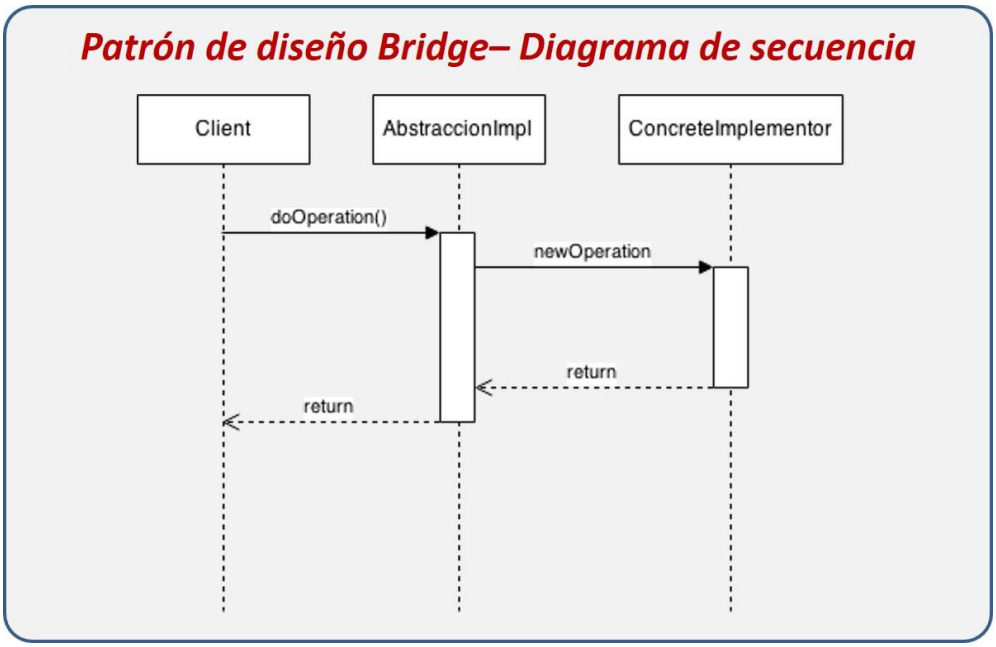


Ilustración 57: Forma en la que trabaja el patrón Bridge.

Los pasos son los siguientes:

1. El cliente ejecuta una operación de *AbstraccionImpl*.
2. *AbstraccionImpl* replica la petición a *ConcreteImplementor*, en este paso el *AbstraccionImpl* pudiera realizar una conversión de los parámetros para ejecutar al *ConcreteImplementor*.
3. *ConcreteImplementor* regresa los resultados al *AbstraccionImpl*.
4. El *AbstraccionImpl* convierte los resultados del *ConcreteImplementor* para ser devueltos al cliente.

Cuándo utilizarlo:

- Cuando existen clases fuertemente relacionadas entre sí, pero que alguna de las dos es propensa a cambiar con el tiempo.

El escenario:

Para comprender mejor cómo funciona este patrón realizaremos un escenario simple pero funcional. Nos han pedido que realicemos un programa que se comunique con otra aplicación. Los requerimientos hacen mucho énfasis en la confidencialidad de la información que viaja, por lo que nos han pedido que encriptemos la información antes de enviarla, sin embargo, los usuarios de la aplicación destino no han definido exactamente cuál será el método de encriptamiento con el que se le deben de enviar los mensajes, es por eso que nos han pedido que creemos un componente que sea versátil en este sentido y que además, si cambia el método de encriptamiento no afecte al componente ya desarrollado.

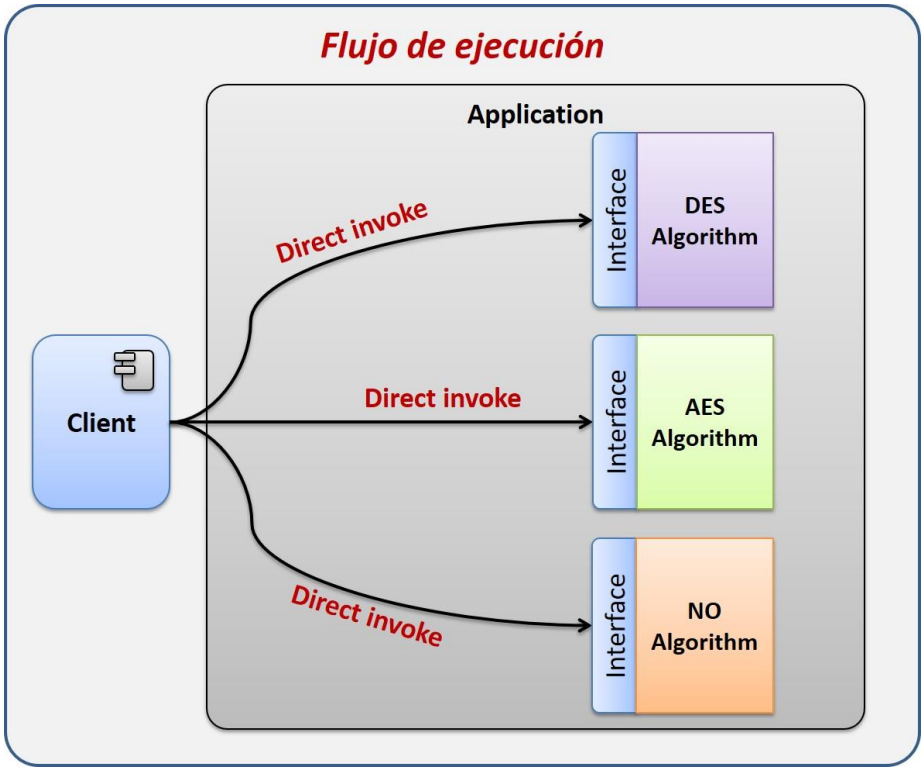


Ilustración 58: Escenario de invocación directa.

En la imagen apreciamos una aplicación que tiene tres métodos de encriptamiento, cuando los requerimientos cambien y el método de

encriptamiento utilizado también cambie, el cliente está obligado a hacer un cambio para apuntar al nuevo método de encriptamiento, esto puede suceder cuantas veces el método de encriptamiento cambie.

La solución:

Para darle una solución eficiente a este problema podemos optar por utilizar el patrón Bridge, el cual nos puede ayudar a desacoplar al cliente de la implementación de encriptamiento utilizada, de esta forma el cliente únicamente se comunicará con el puente y evitará tener una relación con el método de encriptamiento utilizado.

Para implementar correctamente el patrón Bridge deberemos crear una interface que el cliente conozca, la cual denominaremos Abstracción, ésta servirá como un puente entre el cliente y la implementación de encriptación. El puente o *bridge* permitirá desacoplar las dos partes permitiendo cambiar el algoritmo de encriptamiento sin afectar al cliente.

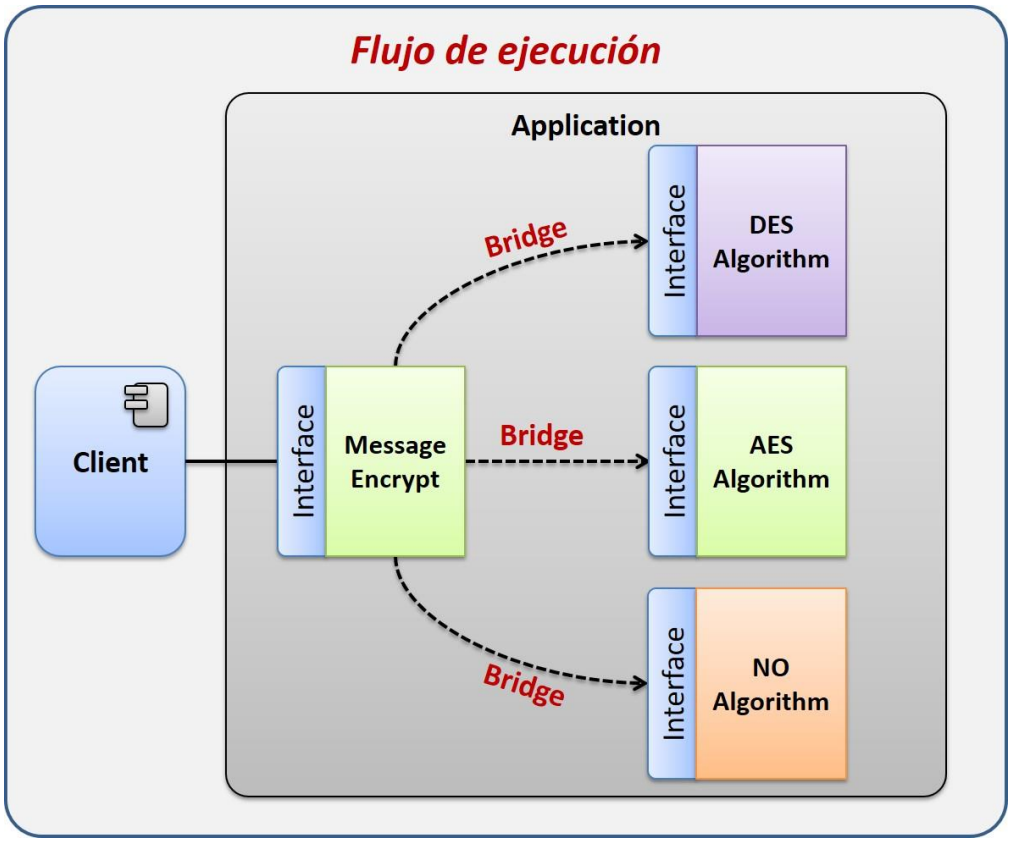


Ilustración 59: Encriptamiento por medio del patrón Bridge.

En la imagen podemos apreciar que esta vez el cliente no se comunica directamente con las implementaciones de encriptamiento, en su lugar, delega esta responsabilidad a la Abstracción. Vemos tres métodos de encriptamiento:

- **DES Algorithm:** Implementación que encripta el mensaje siguiente, el algoritmo Data Encryption Standard.
- **AES Algorithm:** Implementación que encripta el mensaje siguiente, el algoritmo Advanced Encryption Standard.
- **NO Algorithm:** Implementación que no encripta el mensaje.

Puede sonar extraño que tengamos una implementación que no encripte el mensaje, ¿No sería mejor en tal caso mejor, no invocar nada y omitir este paso?, en ocasiones la respuesta es **NO**, ya que esto afectaría al cliente teniendo que modificarlo para que ya no realice la encriptación, y luego, cuando se decidan a sí encriptar el mensaje, se tendría que hacer una segunda modificación para llamar al *Bridge*, es por esta razón por la que mejor se crea una implementación *dummy* y de esta forma el cambio realizando en la implementación no afecta al cliente.

La implementación:

Con el escenario y la solución claros iniciaremos con la implementación de la solución.

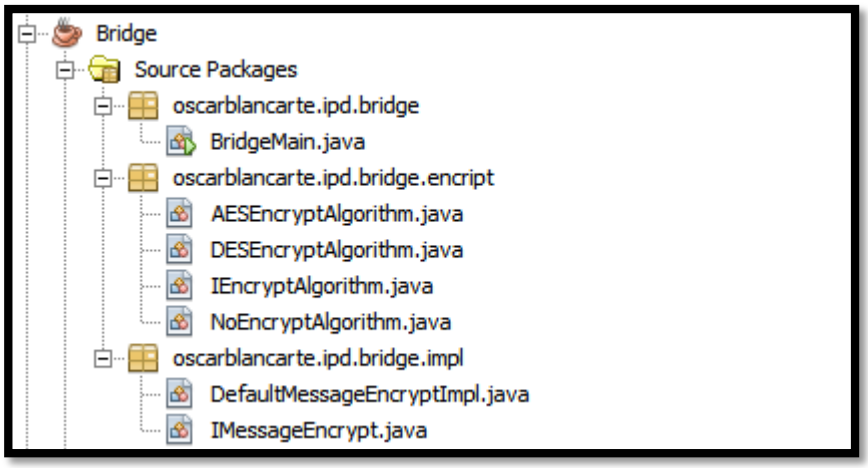


Ilustración 60: Estructura del proyecto Bridge.

En la imagen apreciamos los distintos paquetes del proyecto los cuales se explican a continuación:

- **oscarblancarte.ipd.bridge:** Paquete por *default* de la aplicación el cual únicamente contiene la clase ejecutable del proyecto.
- **oscarblancarte.ipd.bridge.encript:** Define las implementaciones de los algoritmos de encriptación así como su interface en común.
- **Oscarblancarte.ipd.bridge.impl:** Define la Abstracción y su implementación por *default*.

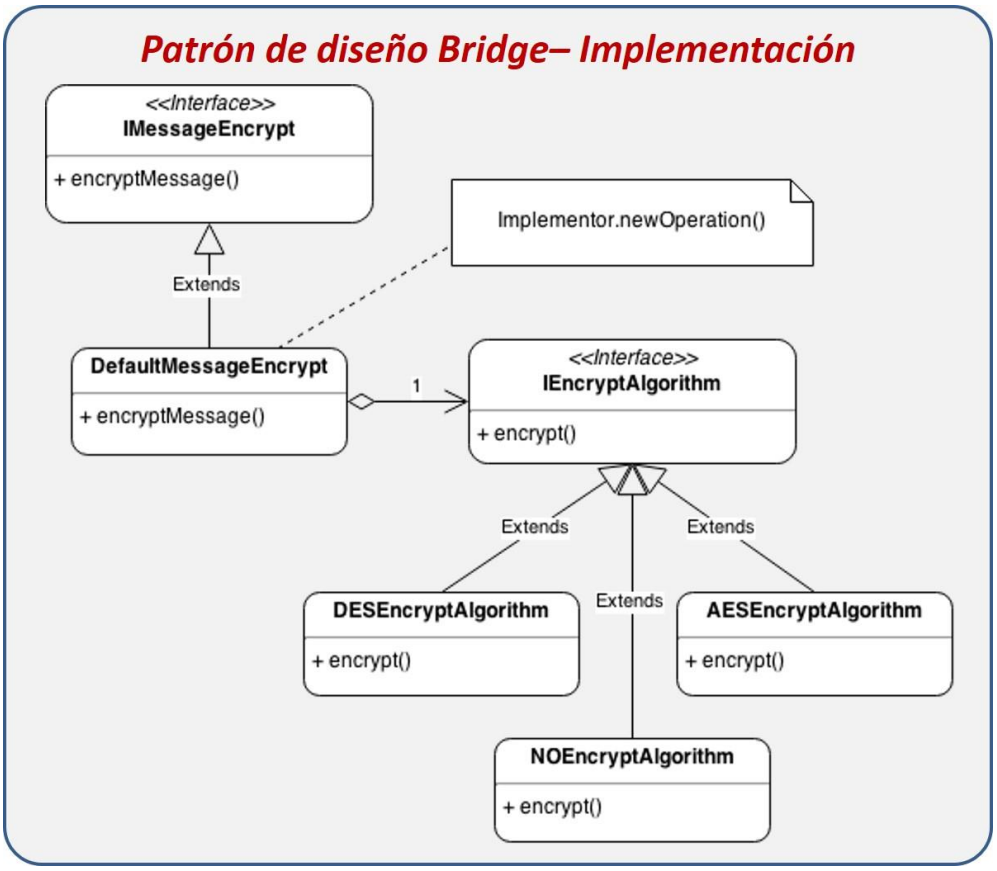


Ilustración 61: Diagrama de clases del proyecto.

Interface IEncryptAlgorithm:

Interface común para todas las implementaciones de algoritmos de encriptamiento.

```
1. package oscarblancarte.ipd.bridge.encript;
2.
3. public interface IEncryptAlgorithm {
4.     public String encrypt(String message, String password) throws Exception;
5. }
```

La interface únicamente define el método *encrypt* el cual se utilizará para encriptar los mensajes.

Clase NoEncryptAlgorithm:

Clase que hereda de *IEncryptAlgorithm*, la clase únicamente regresa el mensaje recibido sin realizar ningún tipo de encriptamiento.

```
1.  package oscarblancarte.ipd.bridge.encript;
2.
3.  public class NoEncryptAlgorithm implements IEncryptAlgorithm{
4.      @Override
5.      public String encrypt(String message, String password) throws Exception {
6.          return message;
7.      }
8.  }
```

Clase DESEncryptAlgorithm:

Clase que hereda de *IEncryptAlgorithm* y encripta el mensaje con el algoritmo Data Encryption Standard.

```
1.  package oscarblancarte.ipd.bridge.encript;
2.
3.  import javax.crypto.Cipher;
4.  import javax.crypto.SecretKey;
5.  import javax.crypto.SecretKeyFactory;
6.  import javax.crypto.spec.DESKeySpec;
7.  import sun.misc.BASE64Encoder;
8.
9.  public class DESEncryptAlgorithm implements IEncryptAlgorithm {
10.
11.         @Override
12.         public String encrypt(String message, String password) throws Exception {
13.             DESKeySpec dks = new DESKeySpec(message.getBytes());
14.             SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
15.             SecretKey desKey = skf.generateSecret(dks);
16.             Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
17.             desCipher.init(Cipher.ENCRYPT_MODE, desKey);
18.
19.             BASE64Encoder base64encoder = new BASE64Encoder();
20.             return base64encoder.encode(desCipher.doFinal(message.getBytes()));
21.         }
22.     }
```

Clase AESEncryptAlgorithm:

Clase que hereda de *IEncryptAlgorithm* y encripta el mensaje mediante el algoritmo Advance Encryption Standard.

```
1.  package oscarblancarte.ipd.bridge.encript;
2.
3.  import java.security.Key;
4.  import javax.crypto.Cipher;
5.  import javax.crypto.spec.SecretKeySpec;
6.  import sun.misc.BASE64Encoder;
7.
8.  public class AESEncryptAlgorithm implements IEncryptAlgorithm {
9.
10.         @Override
11.         public String encrypt(String message, String password) throws Exception {
12.             Key key = new SecretKeySpec(password.getBytes(), "AES");
13.             Cipher c = Cipher.getInstance("AES");
14.
15.             c.init(Cipher.ENCRYPT_MODE, key);
16.             byte[] encVal = c.doFinal(message.getBytes());
17.             String encryptedValue = new BASE64Encoder().encode(encVal);
18.             return encryptedValue;
19.         }
20.     }
```

Interface IMessageEncrypt:

Interface que define la estructura que deberán tener las clases puente, mediante esta interface el cliente se comunicará con los algoritmos de encriptamiento concretos.

```
1.  package oscarblancarte.ipd.bridge.impl;
2.
3.  public interface IMessageEncrypt {
4.      public String encryptMessage(String message, String password)throws Exception;
5.  }
```

En la clase se define el método *encryptMessage* la cual delegará la responsabilidad de encriptamiento a una de las implementaciones.

Clase DefaultMessageEncryptImpl:

Clase que representa al *bridge* o puente, la clase implementa *IMessageEncrypt* y delega la responsabilidad al método de encriptamiento asignado durante la creación del objeto.

```
1.  package oscarblancarte.ipd.bridge.impl;
2.
3.  import oscarblancarte.ipd.bridge.encript.IEncryptAlgorithm;
4.
5.  public class DefaultMessageEncryptImpl implements IMessageEncrypt{
6.      private IEncryptAlgorithm encryptAlgorith;
7.
8.      public DefaultMessageEncryptImpl(IEncryptAlgorithm encryptAlgorith){
9.          this.encryptAlgorith = encryptAlgorith;
10.         }
11.
12.         @Override
13.         public String encryptMessage(String message, String password)throws Exception {
14.             return encryptAlgorith.encrypt(message, password);
15.         }
16.     }
```

En el constructor recibe la implementación, a la cual delegará la responsabilidad de encriptar el mensaje y en el método *encryptMessage* es donde el puente entre el cliente y la implementación se realizan. En este paso puede existir una transformación de los parámetros de entrada y salida de la clase de encriptamiento para que coincidan con los esperados por el cliente. En este punto también es habitual utilizar el patrón *Adapter*, sobre todo si las clases de encriptamiento tuvieran interfaces incompatibles.

Clase BridgeMain:

Finalmente, la clase *BridgeMain* la utilizaremos para probar el programa y validar el escenario.

```
1.  package oscarblancarte.ipd.bridge;
2.
3.  import oscarblancarte.ipd.bridge.encript.AESEncryptAlgorithm;
4.  import oscarblancarte.ipd.bridge.encript.DESEncryptAlgorithm;
```

```
5.  import oscarblancarte.ipd.bridge.encript.NoEncryptAlgorithm;
6.  import oscarblancarte.ipd.bridge.impl.DefaultMessageEncryptImpl;
7.
8.  public class BridgeMain {
9.
10.         public static void main(String[] args) {
11.             IMessageEncrypt aesImpl = new DefaultMessageEncryptImpl(
12.                     new AESEncryptAlgorithm());
13.             IMessageEncrypt desImpl = new DefaultMessageEncryptImpl(
14.                     new DESEncryptAlgorithm());
15.             IMessageEncrypt noImpl = new DefaultMessageEncryptImpl(
16.                     new NoEncryptAlgorithm());
17.
18.             try {
19.                 final String message = "<Persona><Nombre>Oscar Blancarte</Nombre></Persona>";
20.                 String messageAES = aesImpl.encryptMessage(message, "HG58YZ3CR9123456");
21.                 System.out.println("messageAES > " + messageAES + "\n");
22.
23.                 String messageDES = desImpl.encryptMessage(message, "XMzDdG4D03CKm2Ix");
24.                 System.out.println("messageDES > " + messageDES + "\n");
25.
26.                 String messageNO = noImpl.encryptMessage(message, null);
27.                 System.out.println("messageNO > " + messageNO + "\n");
28.             } catch (Exception e) {
29.                 e.printStackTrace();
30.             }
31.         }
32.     }
```

En la línea 11 a la 16 podemos apreciar como creamos varias Abstracciones a las cuales se les establece la implementación que utilizarán en el constructor. Cuando estas abstracciones sean utilizadas por el cliente, no se dará cuenta qué implementación se está utilizando y lo que pase dentro de la Abstracción será totalmente transparente para él.

En la línea 19 se establece el mensaje a encriptar.

En las líneas 20 y 21 se realiza una encriptación con AES y se imprime el resultado de la encriptación.

En las líneas 23 y 24 se realiza la encriptación con DES y se imprime en pantalla.

En la línea 26 se realiza una encriptación *dummy*, únicamente regresa el mensaje de entrada intacto, sin embargo, al hacer esto estamos respetando la forma de trabajar del cliente.

Cualquiera que sea la implementación utilizada, el cliente no se da cuenta ya que desacoplamos la implementación de la abstracción, lo cual es el objetivo de este ejemplo, podemos apreciar que desde el cliente estamos instanciando las Abstracciones, sin embargo, esto lo podríamos realizar en otra parte, de tal manera que el cliente sólo reciba la abstracción por *default* y podrá tener el método de encriptamiento definido desde la configuración.

La ejecución:

Con el fin de probar la aplicación, ejecutaremos la clase ***BridgeMain*** y tendremos el siguiente resultado:

```
run:
messageAES > /ys2+wMWsTC++PBgb2DgL31foKvPUkfSx/hpYy8oaCim+9Avd3fJ0y6NqUL/TRUj33Vrvw5hynDc
6UpsWR72wA==

messageDES > rVc5QkTsmujeP2BskfIsPs3P9VqYbAn5oFHOE+ZqFZGTPb+nwejarI1cUIDefSO7B/gR+ODlibE=

messageNO > <Persona><Nombre>Oscar Blancarte</Nombre></Persona

BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 62: Resultado de la ejecución del proyecto Bridge.

Para simplificar el ejemplo se omite el envío del mensaje, debido a que lo que nos interesa es ver cómo funciona el patrón *bridge* y no cómo enviar un mensaje.

Este ejemplo es muy pequeño pero sirve para demostrar el uso del patrón.

Siguientes pasos:

Hemos visto cómo el patrón Bridge es especialmente útil cuando tenemos escenarios donde es importante desacoplar la implementación, sin embargo, hablamos hace un momento de que *bridge* se puede implementar utilizando *Adapter*.
Si nos damos cuenta, en este ejemplo todas las implementaciones heredan de una clase en común la cual es *IEncryptAlgorithm* lo que nos facilita mucho el trabajo, pero ¿qué pasaría si las implementaciones tuvieran interfaces incompatibles?, esto nos llevaría a crear un *Adapter* entre la implementación y la abstracción con el fin de lograr que las interfaces fueran compatibles para la Abstracción.

Para fortalecer los conocimientos adquiridos realicemos los siguientes ejercicios:

1. Realicemos un nuevo algoritmo de encriptación que implementen *IEncriptAlgorithm*, lo importante sería crear una nueva implementación y utilizarla desde el cliente.

2. Creemos un nuevo algoritmo o utilicemos el implementado anteriormente, pero esta clase no debe de implementar *IEncryptAlgoriht*, la idea es que utilicemos el patrón *Adapter* para convertir esta interface incompatible en una compatible con el resto de los algoritmos.

3. Implementemos el patrón de diseño *Factory* con la finalidad de construir el *Bridge* mediante configuración, definamos la clase mediante un archivo de configuración y que el Factory lea el archivo y defina la clase, para finalmente crear el *Bridge* con dicha clase.

Patrón Composite

El patrón de diseño *Composite* nos sirve para construir estructuras complejas partiendo de otras mucho más simples; dicho de otra manera, podemos crear estructuras compuestas que están conformadas por otras estructuras más pequeñas.

Este patrón es muy útil cuando requerimos manipular un conjunto de elementos muy parecidos entre si de una forma uniforme, por lo cual estas clases deben pertenecer a la misma familia y tener un ancestro en común, el cual define el comportamiento mínimo que deben de tener para poder ser tratadas uniformemente y poder construir estructuras jerárquicas compuestas por objetos más simples. Los objetos simples y compuestos son conocidos como *Leaf* y *Composite* respectivamente y el ancestro en común es llamado *Componente*. Estas estructuras son construidas en forma de árbol en donde la recursividad juega un papel muy importante para sacar el máximo provecho a toda la estructura.

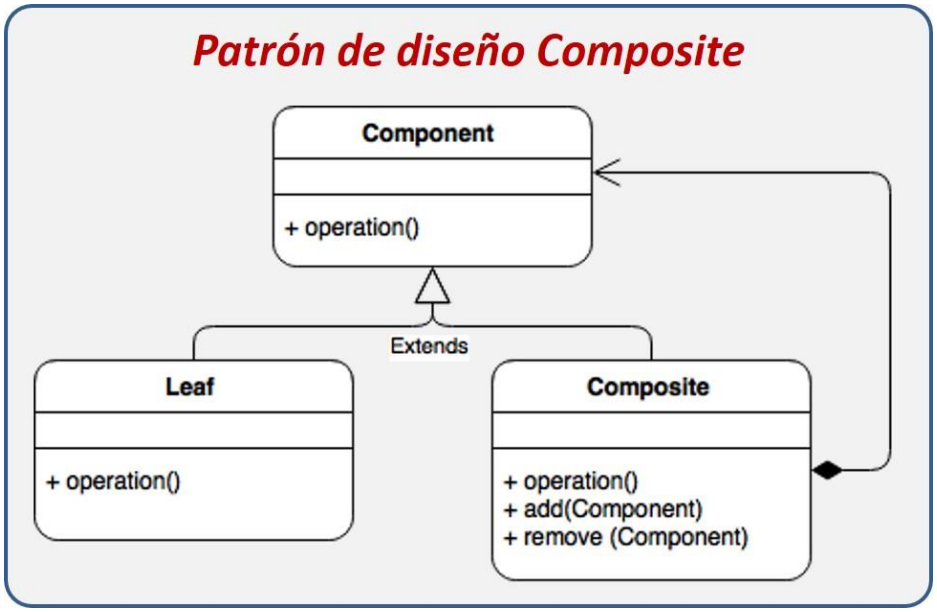


Ilustración 63: Estructura del patrón de diseño Composite.

Los componentes del patrón se explican a continuación:

Component: Generalmente es una interface o clase abstracta que tiene las operaciones mínimas que serán utilizadas, este componente deberá ser extendido por los otros dos componentes *Leaf* y *Composite*.

Leaf: El *leaf* u hoja, representa la parte más simple o pequeña de toda la estructura y ésta hereda de *Component*. *Leaf* recibe su nombre de la teoría de árboles, donde se le nombra así a todo nodo que no tiene descendencia, en este caso son clases simple que no están compuestas de otras.

Composite: Este componente es el que le da vida al patrón de diseño ya que este objeto está conformado por un conjunto de *Component* y *Leaf*. En teoría de árboles este componente representaría una rama.

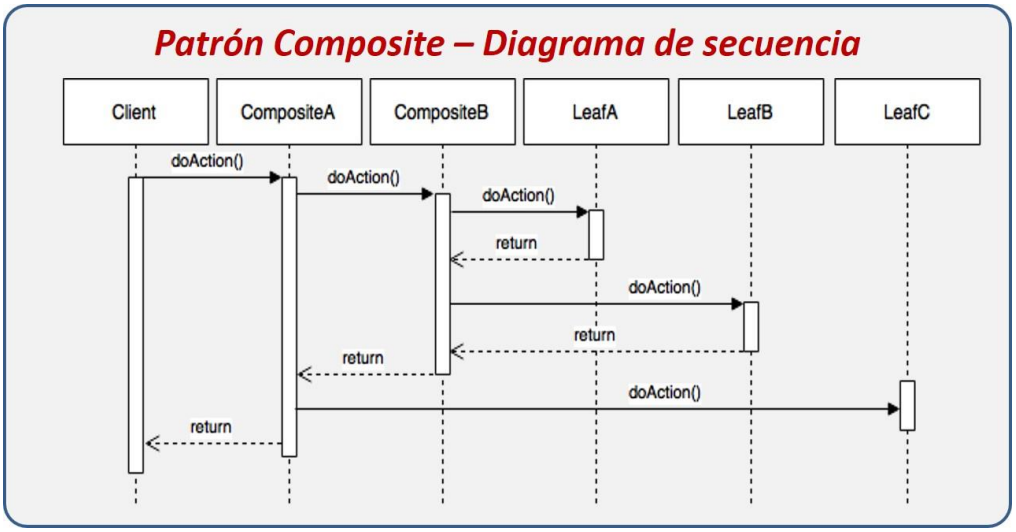


Ilustración 64: Diagrama de secuencia del patrón Composite.

Antes que nada, observemos que **CompositeA** es un objeto compuesto y que está formado por **CompositeB** y **LeafC**, a su vez **CompositeB** está formado por **LeafA** y **LeafB**. Con este entendido pasemos a analizar el diagrama:

1. El cliente realizar una acción sobre el CompositeA.
2. CompositeA a su vez realiza una acción sobre CompositaB.

3. CompositeB realiza una acción sobre LeafA y LeafB y el resultado es devuelto a CompositeA.
4. CompositeA propaga la acción sobre LeafC, el cual le regresa un resultado.
5. CompositeA obtiene un resultado final tras la evaluación de toda la estructura y el cliente obtiene un resultado.

Observemos que el patrón Composite forma una estructura de árbol, donde la raíz es el objeto compuesto que agrupa todos los demás objetos. Sin embargo, el compuesto tiene las operaciones para evaluar toda la estructura, evitando así que el cliente tenga que conocer los detalles finos de la estructura. El recorrido de toda la estructura se realiza mediante recursividad.

Cuándo utilizarlo:

- Cuando detectamos estructuras complejas que podrían dividirse en elementos más pequeños, los cuales están muy relacionados entre sí, tanto en funcionamiento como en estructura.

El Escenario:

Imaginemos un sistema de punto de venta, en el cual se le pueden vender al cliente una serie de productos, estos productos pueden ser simples (*Leaf*) o paquetes (*Composite*). El sistema permitirá crear órdenes de ventas que estarán compuestas por uno o muchos productos.

El precio de los productos deberá ser calculado mediante la suma del precio de todos los productos internos. La siguiente imagen ilustra la estructura de un paquete y cómo es que el precio del mismo es calculado basado en el precio de los productos que lo componen.

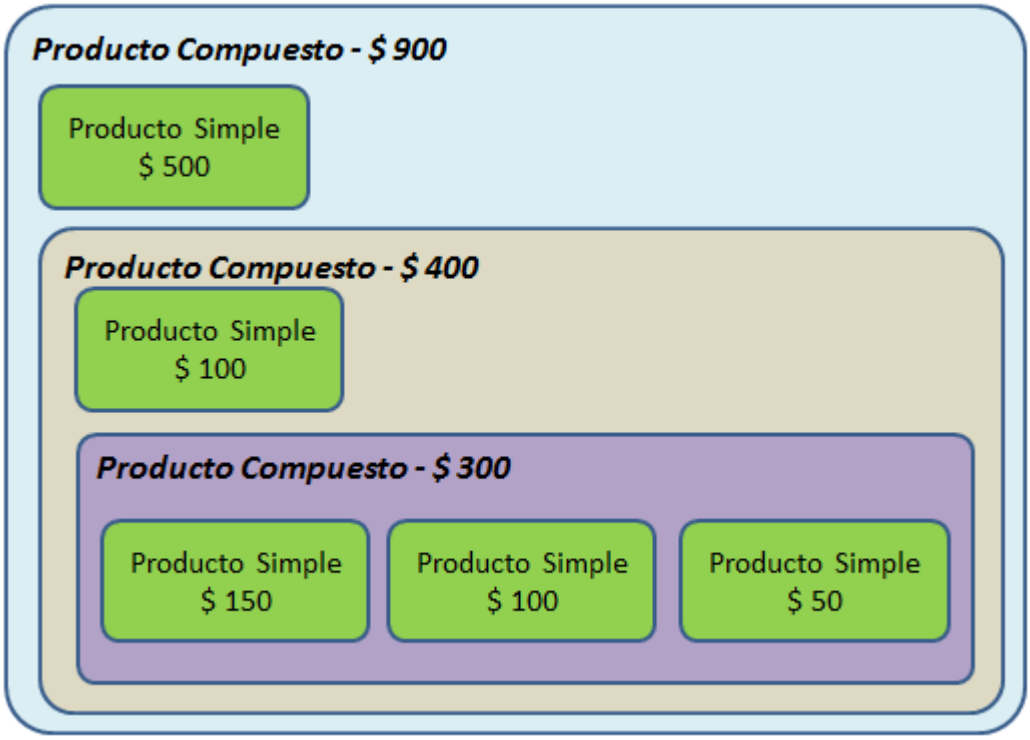


Ilustración 65: Estructura de un producto compuesto.

En la imagen podemos apreciar la estructura que conforma un producto compuesto, los productos compuestos están conformados por productos simples y por productos compuestos. Los productos compuestos internos pueden a la vez

estar creados por más productos simples y compuestos creando una estructura en forma de árbol.

Algunos puntos importantes a tomar en cuenta:

- Un paquete es creado mediante una serie de productos simples y compuestos.
- No debe existir una diferencia entre la forma que tratamos un producto simple de uno compuesto.
- El precio de un paquete es la suma de todos los productos simples que contenga.
- El sistema deberá mostrar el total de la orden y los productos que contiene.

La solución:

Para solucionar el problema planteado, tendremos que diseñar una estructura de productos que nos permita crear productos simples y compuestos; con los productos simples no tendríamos problemas, pero los productos compuestos son más complicados. Para crear una estructura flexible utilizaremos el patrón de diseño Composite, el cual nos permitirá crear estructuras de productos compuestas mediante productos simples.

En la siguiente imagen veremos cómo deberíamos implementar la estructura de productos, de tal manera que sea fácilmente analizable y determinar su precio mediante la suma del precio de todos los productos que componen el compuesto.

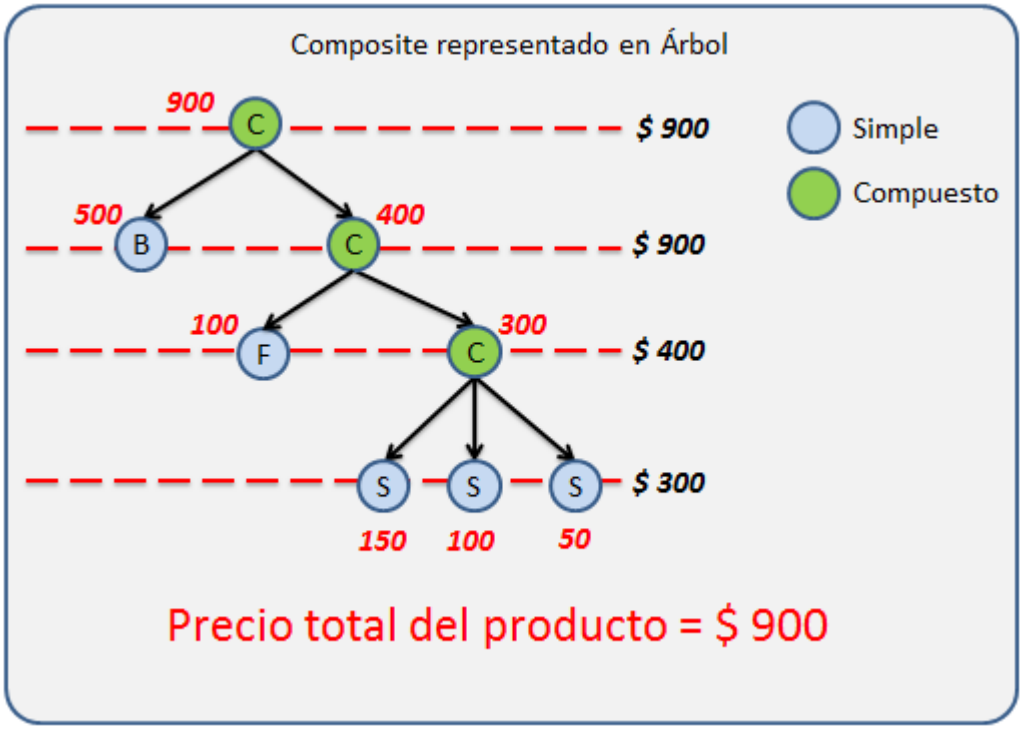


Ilustración 66: Cálculo del precio en un producto compuesto.

Como podemos ver en la imagen, el precio de un producto compuesto es igual a la suma del precio de todos sus productos simples; éstos, sumados en forma recursiva, nos darán el precio total del producto. Otra de las ventajas es que nos permite tener una estructura organizada de cómo está creado un producto.

La implementación:

Una vez que hemos explicado el escenario, iniciaremos con la implementación de la solución.

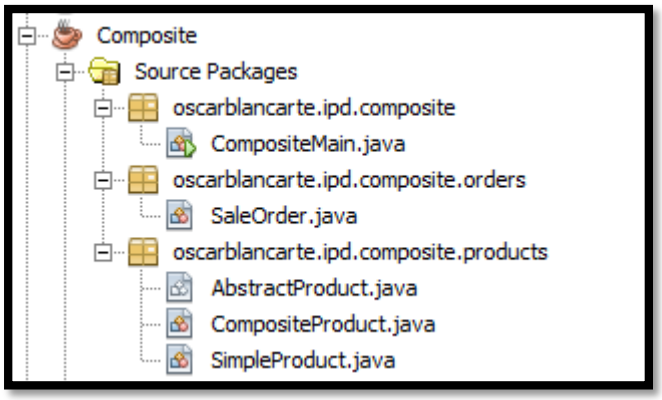


Ilustración 67: Estructura del proyecto Composite.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.composite:** Paquete principal con la clase ejecutable del proyecto.
- **oscarblancarte.ipd.composite.orders:** Paquete con las clases para la creación de las órdenes.
- **oscarblancarte.ipd.composite.products:** Paquete con la estructura de los productos implementados con el patrón Composite.

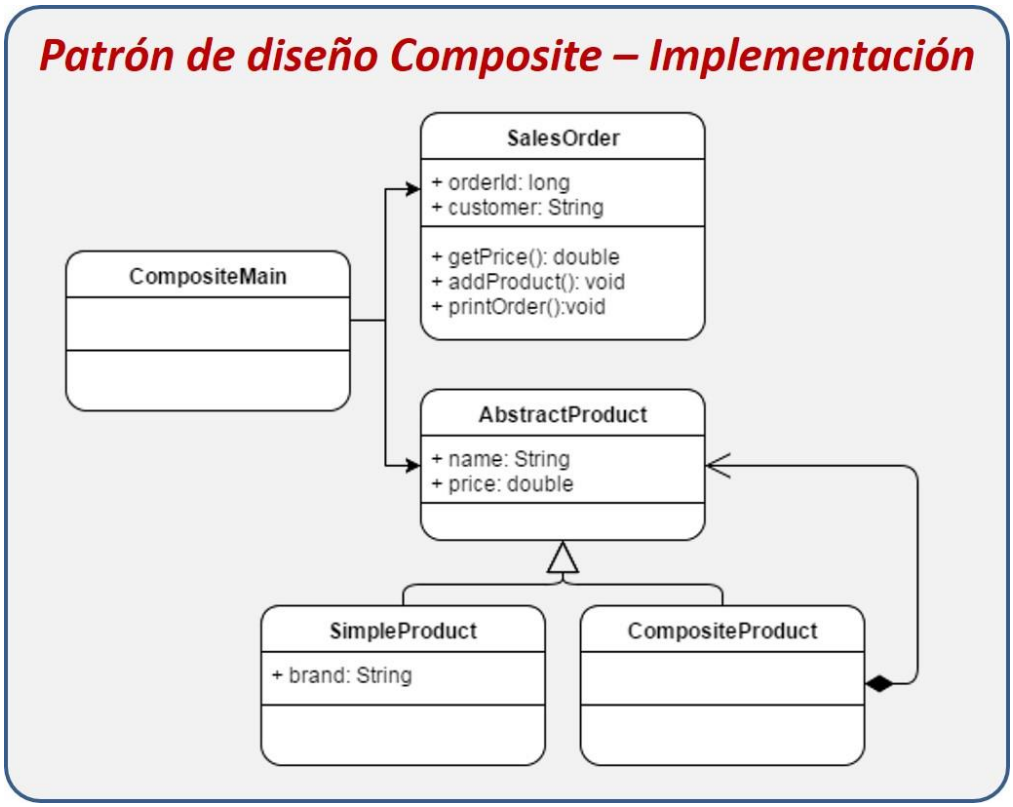


Ilustración 68: Estructura del proyecto Composite.

En la imagen se muestran las clases que componen el proyecto Composite y cómo es que están relacionadas entre sí.

Clase AbstractProduct:

Clase base utilizada para crear los productos simples y compuestos.

```
1.  package oscarblancarte.ipd.composite;
2.
3.  public abstract class AbstractProduct {
4.
5.      protected String name;
6.      protected double price;
7.
8.      public AbstractProduct(String name, double price) {
9.          super();
10.             this.name = name;
```

```
11.             this.price = price;
12.         }
13.
14.         /** GET and SET */
15.     }
```

Esta clase la utilizaremos para definir la estructura mínima de un producto, logrando con ello que todos los productos definidos en nuestra aplicación puedan ser tratados de forma homogénea.
Se definen las propiedades para el nombre y el precio del producto (name y price).

Clase SimpleProduct:

Clase que representa al producto simple, este producto no podrá estar compuesto por otros productos, por lo que representa la forma más básica de un producto.

```
1.  package oscarblancarte.ipd.composite;
2.
3.  public class SimpleProduct extends AbstractProduct {
4.
5.      protected String brand;
6.
7.      public SimpleProduct(String name, double price, String brand) {
8.          super(name, price);
9.          this.brand = brand;
10.         }
11.         /** GET and SET */
12.     }
```

La clase hereda de *AbstractProduct* con la finalidad de mantener una estructura compatible con todos los productos que podrían ser parte de un producto compuesto.

Se agrega únicamente una propiedad para la marca del producto (brand).

Clase CompositeProduct:

Esta clase representa a los productos compuestos, la utilizaremos para la creación de los paquetes. Observemos que la clase hereda de *AbstractProduct* para mantener una estructura compatible con los productos simples.

```
1.  package oscarblancarte.ipd.composite;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class CompositeProduct extends AbstractProduct {
7.
8.      private List<AbstractProduct> products = new ArrayList<AbstractProduct>();
9.
10.         public CompositeProduct(String name) {
11.             super(name, 0);
12.         }
13.
14.         @Override
15.         public double getPrice() {
16.             double price = 0d;
17.             for (AbstractProduct child : products) {
18.                 price += child.getPrice();
19.             }
20.             return price;
21.         }
22.
23.         @Override
24.         public void setPrice(double price) {
25.             throw new UnsupportedOperationException();
26.         }
27.
28.         public void addProduct(AbstractProduct product) {
29.             this.products.add(product);
30.         }
31.
32.         public boolean removeProduct(AbstractProduct product) {
33.             return this.products.remove(product);
34.         }
35.     }
```

Dentro de la clase tenemos una lista de *AbstractProduct*, esta lista nos permitirá agregar los productos que componen el producto compuesto. También, observemos que el método *setPrice* ha sido sobrescrito para lanzar una excepción en caso de que se quiera modificar directamente; por otro lado, el método *getPrice*

también es sobrescrito para obtener el precio mediante la suma de todos los productos que contiene.
Adicionalmente se agregan los métodos *addProduct* y *removeProduct* con la finalidad de quitar o remover productos del compuesto.

Clase SaleOrder:

Clase que intenta representar una orden de venta a muy alto nivel, con la finalidad de simular cómo es que la aplicación de punto de venta utilizará los productos simples y compuestos para vendérselos a nuestros clientes.

```
1.  package oscarblancarte.ipd.composite;
2.
3.  import java.text.DecimalFormat;
4.  import java.text.NumberFormat;
5.  import java.util.ArrayList;
6.  import java.util.List;
7.
8.  public class SaleOrder {
9.
10.         private long orderId;
11.         private String customer;
12.         private List<AbstractProduct> products = new ArrayList<AbstractProduct>();
13.
14.         public SaleOrder(long orderId, String customer) {
15.             super();
16.             this.orderId = orderId;
17.             this.customer = customer;
18.         }
19.
20.         public double getPrice() {
21.             double price = 0d;
22.             for (AbstractProduct child : products) {
23.                 price += child.getPrice();
24.             }
25.             return price;
26.         }
27.
28.         public void addProduct(AbstractProduct product) {
29.             products.add(product);
30.         }
31.
32.         public void printOrder() {
33.
34.             NumberFormat formater = new DecimalFormat("###,##0.00");
35.             System.out.println("\\\n==========================================="
36.                     + "\nOrden: " + orderId + "\nCliente: " + customer
```

```
37.                 + "\nProductos:\n");
38.             for (AbstractProduct prod : products) {
39.                 System.out.println(prod.getName() + "\t\t\t$ "
40.                     + formater.format(prod.getPrice()));
41.             }
42.             System.out.println("Total: " + formater.format(getPrice())
43.                 + "\n==========================================");
44.         }
45.
46.         /** GET and SET */
47.     }
```

En la clase tenemos los siguientes métodos:

- **getPrice:** Permite obtener el precio total de la orden sumando el precio de todos los productos contenidos.
- **addProduct:** Método utilizado para agregarle productos a la orden.
- **printOrder:** Método utilizado para simular el *ticket* de venta, en el cual imprimimos los productos contenidos, el precio de cada producto y el monto total de la orden.

Clase CompositeMain:

Finalmente, tenemos la clase *CompositeMain*, la utilizaremos para emular la creación de órdenes de venta.

```
1.  package oscarblancarte.ipd.composite;
2.
3.  import java.util.Random;
4.
5.  public class CompositeMain {
6.
7.      private final Random random = new Random();
8.
9.      private SimpleProduct ram4gb;
10.         private SimpleProduct ram8gb;
11.         private SimpleProduct disk500gb;
12.         private SimpleProduct disk1tb;
13.         private SimpleProduct cpuAMD;
14.         private SimpleProduct cpuIntel;
15.         private SimpleProduct smallCabinete;
16.         private SimpleProduct bigCabinete;
17.         private SimpleProduct monitor20inch;
```

```
18.     private SimpleProduct monitor30inch;
19.     private SimpleProduct simpleMouse;
20.     private SimpleProduct gammerMouse;
21.
22.     private CompositeProduct gammerPC;
23.     private CompositeProduct homePC;
24.     private CompositeProduct pc2x1;
25.
26.     private void buildProducts() {
27.         //Construcción de productos simples
28.         ram4gb = new SimpleProduct("Memoria RAM 4GB", 750, "KingStone");
29.         ram8gb = new SimpleProduct("Memoria RAM 8GB", 1000, "KingStone");
30.         disk500gb = new SimpleProduct("Disco Duro 500GB", 1500, "ACME");
31.         disk1tb = new SimpleProduct("Disco Duro 1TB", 2000, "ACME");
32.         cpuAMD = new SimpleProduct("AMD phenon", 4000, "AMD");
33.         cpuIntel = new SimpleProduct("Intel i7", 4500, "Intel");
34.         smallCabinete = new SimpleProduct("Gabinete Pequeï¿½o", 2000, "ExCom");
35.         bigCabinete = new SimpleProduct("Gabinete Grande", 2200, "ExCom");
36.         monitor20inch = new SimpleProduct("Monitor 20'", 1500, "HP");
37.         monitor30inch = new SimpleProduct("Monitor 30'", 2000, "HP");
38.         simpleMouse = new SimpleProduct("Raton Simple", 150, "Genius");
39.         gammerMouse = new SimpleProduct("Raton Gammer", 750, "Alien");
40.
41.         //Computadora para Gammer que incluye 8gb de ram,disco de 1tb, procesador Intel i7
42.         //Gabinete grande, monitor de 30' y un mouse gammer.
43.         gammerPC = new CompositeProduct("Gammer PC");
44.         gammerPC.addProduct(ram8gb);
45.         gammerPC.addProduct(disk1tb);
46.         gammerPC.addProduct(cpuIntel);
47.         gammerPC.addProduct(bigCabinete);
48.         gammerPC.addProduct(monitor30inch);
49.         gammerPC.addProduct(gammerMouse);
50.
51.        //Computadora para Casa que incluye 4gb de ram,disco de 500gb, procesador AMD Phenon
52.         //Gabinete chico, monitor de 20' y un mouse simple.
53.         homePC = new CompositeProduct("Casa PC");
54.         homePC.addProduct(ram4gb);//
55.         homePC.addProduct(disk500gb);
56.         homePC.addProduct(cpuAMD);
57.         homePC.addProduct(smallCabinete);
58.         homePC.addProduct(monitor20inch);
59.         homePC.addProduct(simpleMouse);
60.
61.         //Paque compuesto de dos paquetes: paquete Gammer PC y Home PC
62.         pc2x1 = new CompositeProduct("Paquete PC Gammer + Casa");
63.         pc2x1.addProduct(gammerPC);
64.         pc2x1.addProduct(homePC);
65.     }
66.
67.     private void orderSimpleProducts() {
68.
69.         SaleOrder simpleProductOrder = new SaleOrder(random.nextInt(), "Rene Blancarte");
70.         simpleProductOrder.addProduct(ram4gb);
```

```
71.             simpleProductOrder.addProduct(disk1tb);
72.             simpleProductOrder.addProduct(simpleMouse);
73.             simpleProductOrder.printOrder();
74.         }
75.
76.         private void orderGammerPC() {
77.             SaleOrder gammerOrder = new SaleOrder(1, "Juan Perez");
78.             gammerOrder.addProduct(gammerPC);
79.             gammerOrder.printOrder();
80.         }
81.
82.         private void orderHomePC() {
83.             SaleOrder homeOrder = new SaleOrder(2, "Marcos Guerra");
84.             homeOrder.addProduct(homePC);
85.             homeOrder.printOrder();
86.         }
87.
88.         private void orderCombo() {
89.             SaleOrder comboOrder = new SaleOrder(3, "Paquete 2x1 en PC");
90.             comboOrder.addProduct(pc2x1);
91.             comboOrder.printOrder();
92.         }
93.
94.         private void orderBig() {
95.             SaleOrder customOrder = new SaleOrder(4, "Oscar Blancarte");
96.             customOrder.addProduct(homePC);
97.             customOrder.addProduct(ram8gb);
98.             customOrder.addProduct(ram4gb);
99.             customOrder.addProduct(monitor30inch);
100.            customOrder.addProduct(gammerMouse);
101.            customOrder.printOrder();
102.        }
103.
104.        public static void main(String[] args) {
105.            CompositeMain main = new CompositeMain();
106.            main.buildProducts();
107.
108.            main.orderSimpleProducts();
109.            main.orderHomePC();
110.            main.orderGammerPC();
111.            main.orderCombo();
112.            main.orderBig();
113.        }
114.    }
```

La clase *CompositeMain* será el ejecutable de nuestro ejemplo; desde aquí armaremos un conjunto de productos simples y compuestos, los cuales los utilizaremos para crear órdenes de venta e imprimir su detalle en pantalla.

Lo primero que analizaremos será el método *buildProducts* ya que desde éste crearemos todos los productos simples y compuestos que utilizaremos en el programa. Los productos simples los podemos apreciar de la línea 27 a la 39 y los productos compuestos los podemos ver de la línea 41 a la 64.

Para separar el programa en varios ejemplos he decidido crear cinco métodos, los cuales forman una órden distinta con diferentes configuraciones de productos:

- *orderSimpleProduct:* Método que crea una orden con sólo productos simples.
- *orderHomePC:* Método que crea una orden con un producto compuesto, el cual corresponde a la venta de un paquete de computadora para casa con un precio económico.
- *orderGamePC:* Método que crea una orden con un producto compuesto, el cual corresponde a la venta de un paquete de computadora para gammer con un precio más elevado.
- *orderCombo*: Método que crea una orden con un producto compuesto, sin embargo, el producto que se vende en esta orden es creado a partir de otros dos productos compuestos, el producto que se vende es un paquete que contempla una pc para casa y otra para gammer.
- *orderBig*: Método que crea una orden compuesta de la combinación de productos compuestos y simples, en este caso se está vendiendo una pc para casa, dos memorias RAM, monitor y mouse.

La Ejecución:

Para hacer más fácil de entender la ejecución del ejemplo, ejecutaremos un solo método a la vez de los cinco que acabamos de explicar, con el fin de poder analizar detenidamente cada escenario.

Primera prueba:

Para la primera ejecución comentaremos los métodos como se muestra a continuación:

```
1.  public static void main(String[] args) {
2.          CompositeMain main = new CompositeMain();
3.          main.buildProducts();
4.
5.          main.orderSimpleProducts();
6.  //        main.orderHomePC();
7.  //        main.orderGammerPC();
8.  //        main.orderCombo();
9.  //        main.orderBig();
10.         }
```

Ejecutamos el programa y obtendremos la siguiente salida:

```
run:
\
==============================================
Orden: -1908610927
Cliente: Rene Blancarte
Productos:

Memoria RAM 4GB                  $ 750.00
Disco Duro 1TB                   $ 2,000.00
Raton Simple                     $ 150.00
Total: 2,900.00
==============================================
BUILD SUCCESSFUL (total time: 2 seconds)
```

Ilustración 69: Resultado de la ejecución.

Lo primero que podemos apreciar es el número de orden generado aleatoriamente y el nombre de nuestro cliente seguido de una lista de productos con un precio y finalmente el total de la orden.

Este ejemplo es muy claro ya que si revisamos el precio de los productos que están listados en la orden, son los mismos que definimos en nuestro programa. Y finalmente el total de la orden, que no es más que la suma de los productos.
Este ejemplo es muy simple y no tiene nada extraño que nos pueda llamar la atención.

Segunda prueba:

Para la segunda ejecución comentaremos los métodos como se muestra a continuación:

```
1.  public static void main(String[] args) {
2.          CompositeMain main = new CompositeMain();
3.          main.buildProducts();
4.
5.  //        main.orderSimpleProducts();
6.          main.orderHomePC();
7.  //        main.orderGammerPC();
8.  //        main.orderCombo();
9.  //        main.orderBig();
10.         }
```

Ejecutamos el programa y obtendremos la siguiente salida:

```
run:
\
==============================================
Orden: 2
Cliente: Marcos Guerra
Productos:

Casa PC                    $ 9,900.00
Total: 9,900.00
==============================================
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 70: Resultado de la ejecución.

Nuevamente vemos el número de orden y el nombre de nuestro cliente despúes, tenemos los productos de la orden, en este caso es sólo uno, lo interesante resalta a continuación, el producto Casa PC en realidad es un producto compuesto definido de la siguiente manera:

```
1. homePC = new CompositeProduct("Casa PC");
2. homePC.addProduct(ram4gb);//
3. homePC.addProduct(disk500gb);
4. homePC.addProduct(cpuAMD);
5. homePC.addProduct(smallCabinete);
6. homePC.addProduct(monitor20inch);
7. homePC.addProduct(simpleMouse);
```

Por tal motivo Casa PC es un producto que está formado a partir de los productos simples ram4gb ($750), disk500gb ($1500), cpuAMD ($4000), smallCabinete ($2000), monitor20inch ($1500), simpleMouse ($150), y si sumados esto nos dará la suma de $9,900. Por tal motivo el total de la orden también será el mismo.

Tercera prueba:

Para la tercera ejecución comentaremos los métodos como se muestra a continuación:

```
1. public static void main(String[] args) {
2.         CompositeMain main = new CompositeMain();
3.         main.buildProducts();
4.
5. //         main.orderSimpleProducts();
6. //         main.orderHomePC();
7.         main.orderGammerPC();
8. //         main.orderCombo();
9. //         main.orderBig();
10.     }
```

Ejecutamos el programa y obtendremos la siguiente salida:

```
run:
\
==============================================
Orden: 1
Cliente: Juan Perez
Productos:

Gammer PC                          $ 12,450.00
Total: 12,450.00
==============================================
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 71: Resultado de la ejecución

Esta ejecución es bastante parecida a la anterior, sin embargo, agregamos otro producto compuesto correspondiente a una pc para gammer, el precio del producto *gammer* PC es de $12,450 y está compuesto de la siguiente manera:

```
1. gammerPC = new CompositeProduct("Gammer PC");
2. gammerPC.addProduct(ram8gb);
3. gammerPC.addProduct(disk1tb);
4. gammerPC.addProduct(cpuIntel);
5. gammerPC.addProduct(bigCabinete);
6. gammerPC.addProduct(monitor30inch);
7. gammerPC.addProduct(gammerMouse);
```

El desglose de los precios es el siguiente: ram8gb ($1000), disk1tb ($2000), cpuIntel ($4500), bigCabinete ($2200), monitor30inch ($2000), gammerMouse ($750) y si sumados esto nos dará la suma de $12,450. Por tal motivo el total de la orden también será el mismo.

Cuarta prueba:

Para la cuarta ejecución comentaremos los métodos como se muestra a continuación:

```
1. public static void main(String[] args) {
2.         CompositeMain main = new CompositeMain();
3.         main.buildProducts();
```

```
4.
5. //      main.orderSimpleProducts();
6. //      main.orderHomePC();
7. //      main.orderGammerPC();
8.        main.orderCombo();
9. //      main.orderBig();
10.     }
```

Ejecutamos el programa y obtendremos la siguiente salida:

```
run:
\
==============================================
Orden: 3
Cliente: Paquete 2x1 en PC
Productos:

Paquete PC Gammer + Casa                        $ 22,350.00
Total: 22,350.00
==============================================
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 72: Resultado de la ejecución.

Nuevamente un resultado semejante; número de orden y nombre del cliente, sin embargo, en este caso estamos vendiendo un producto más complejo ya que es un producto creado a partir de otros dos productos compuestos, además, estos productos compuestos están creados a partir de varios productos simples, por lo tanto, para desglosar el Paquete PC *Gammer* + Casa es necesario desglosar el precio de todos los productos que lo componen.

Home PC = ram4gb ($750) + disk500gb ($1500) + cpuAMD (4000) + smallCabinete (2000) + monitor20inch (1500) + simpleMouse ($150) = **9,900.00.**

Gammer PC = ram8gb ($1000) + disk1tb ($2000) + cpuIntel (4500) + bigCabinete (2200) + monitor30inch ($2000) + gammerMouse ($750) = **12,450.00.**

Paquete PC Gammer + Casa = Home PC (9,900) + Gammer PC (12,450) = **22,350.**

Como podemos apreciar, calcular el precio del producto fue un poco más complejo, ya que primero se tenía que resolver el precio de los productos complejos antes de poder determinar el precio total del producto.

Quinta prueba:

Para finalizar, realizaremos un ejemplo más complejo en el cual se venda una combinación de productos simples con productos compuestos.
Para la quinta ejecución comentaremos los métodos como se muestra a continuación:

```
1.    public static void main(String[] args) {
2.        CompositeMain main = new CompositeMain();
3.        main.buildProducts();
4.
5. //       main.orderSimpleProducts();
6. //       main.orderHomePC();
7. //       main.orderGammerPC();
8. //       main.orderCombo();
9.        main.orderBig();
10.     }
```

Ejecutamos el programa y obtendremos la siguiente salida:

```
run:
\
=============================================
Orden: 4
Cliente: Oscar Blancarte
Productos:

Casa PC                    $ 9,900.00
Memoria RAM 8GB                    $ 1,000.00
Memoria RAM 4GB                    $ 750.00
Monitor 30'                        $ 2,000.00
Raton Gammer                       $ 750.00
Total: 14,400.00
=============================================
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 73: Resultado de la ejecución.

En este punto el resultado ya no nos sorprende ya que realizamos la venta del paquete Casa PC y sabemos que cuesta 9,900 y la venta de los productos simples tales como las memorias, monitor y ratón.

El precio total de la orden es conformado por la suma del precio de los productos simples más el precio de los productos compuestos, lo que nos da como resultado 14,400.00.

Siguientes pasos:

Hemos logrado crear estructuras complejas mediante objetos más pequeños, sin embargo, desarrollaremos los siguientes ejercicios con la finalidad de fortalecer el conocimiento adquirido.

- Agreguemos un nuevo atributo a la clase *AbstractProduct* para definir la tasa de impuesto que tendrá el producto, el impuesto deberá ser calculado y mostrado a nivel de paquete.

- Agreguemos otro atributo a la clase *AbstractProduct* para definir la cantidad, esta cantidad deberá ser calculada a la hora de mostrar el precio del paquete.

- De la misma manera que agregamos productos para crear un paquete, podríamos agregar un producto de descuento con precio negativo, hagamos este ejemplo y analicemos los resultados.

Patrón Decorator

El patrón decorator está diseñado para solucionar problemas donde la jerarquía con subclasificación no puede ser aplicada, o se requiere de un gran impacto en todas las clases de la jerarquía con el fin de poder lograr el comportamiento esperado. *Decorator* permite al usuario añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven a la anterior dándole funcionamiento extra.

Este patrón está diseñado con la finalidad de que múltiples decoradores puedan ser apilados uno encima del otro agregando funcionalidad por cada decorador que es añadido a la estructura.

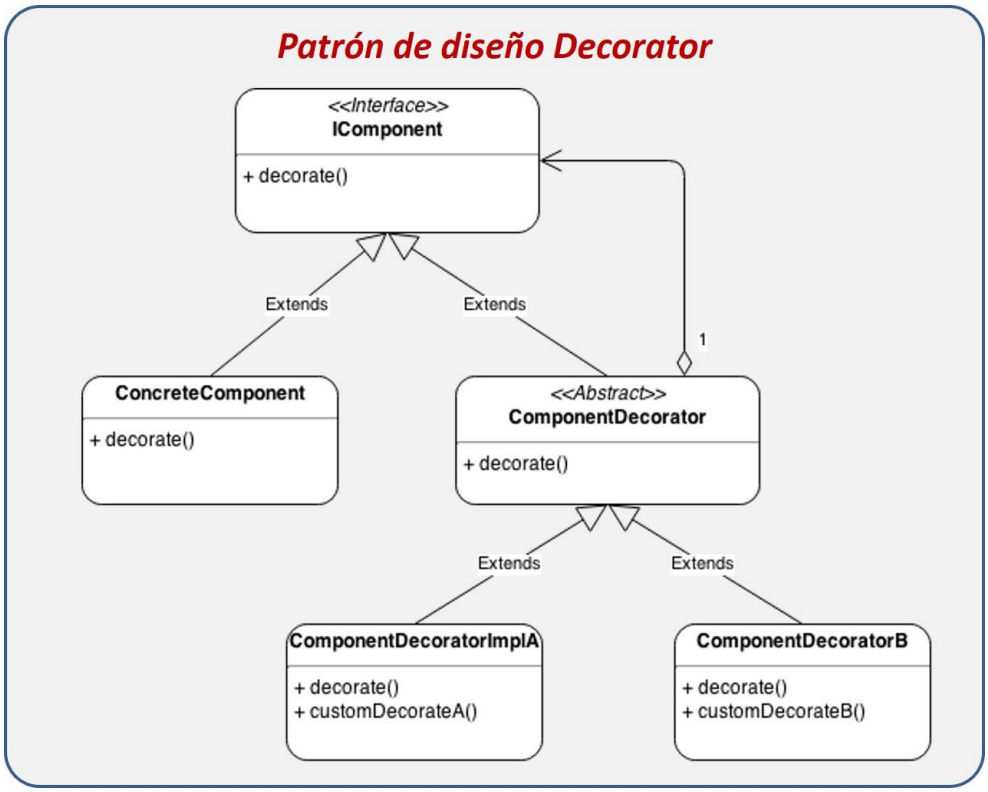


Ilustración 74: Estructura del patrón Decorator.

En la imagen podemos apreciar los distintos componentes que conforman el patrón de diseño Decorator los cuales se explican a continuación:

- **IComponent:** Interface que define la estructura minina del componente o componentes que pueden ser decorados.
- **ConcreteComponent**: Implementación de *IComponent* y define un objeto concreto que puede ser decorado.
- **ComponentDecorator:** Por lo general es una clase abstracta que define la estructura mínima de un Decorador, el cual mínimamente deben de heredar de *IComponent* y contener alguna subclase de *IComponent* al cual decorarán.
- **ComponentDecoratorImpl:** Representan todos los decoradores concretos que heredan de *ComponentDecorator*.

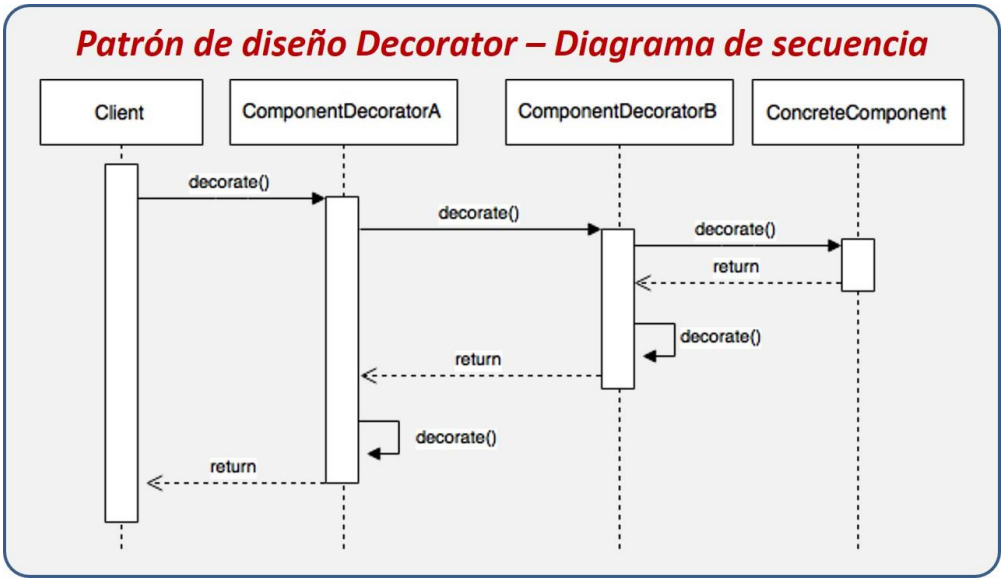


Ilustración 75: Diagrama de secuencia del patrón Decorator.

El flujo de ejecución es el siguiente:

1. El **Cliente** realiza una operación sobre el ***DecoratorA***.
2. El **DecoratorA** realiza la misma operación sobre **DecoradorB**.
3. El **decoradorB** realiza una acción sobre *ConcreteComponente*.
4. El **DecoradorB** ejecuta una operación de decoración.

5. El **DecoradorA** ejecuta una operación de decoración.
6. El **Cliente** recibe como resultado un objeto decorado por todos los **Decoradores**, los cuales encapsularon el *Component* en varias capas.

Cuándo utilizarlo:

- Cuando necesitamos agregar dinámicamente una nueva funcionalidad sobre un objeto y esta funcionalidad se agrega sobre capas de funcionalidad previas.
- Cuando implementar una nueva funcionalidad mediante la herencia es muy complicado o se requieren de fuertes cambios para implementarlos.

El escenario:

Para este escenario tenemos un requerimiento algo especial, nos piden que elaboremos un mecanismo que nos permita modificar un mensaje para ser enviado, el requerimiento es que el mensaje puede ser modificado de muchas maneras y que además, puede haber cambios considerables en la forma en que transformamos el mensaje. De entrada nos piden que dado un mensaje (POJO) creemos un nuevo mensaje de texto, el cual sea convertido en XML, luego lo envolvamos en un mensaje SOAP y finalmente lo encriptamos, como salida deberemos tener únicamente el mensaje encriptado (por lo pronto, porque pueden cambiar los requerimientos). El problema aquí es que tendremos que diseñar una solución versátil que nos permite agregar funcionalidad de transformación al mensaje sin afectar lo ya construido.

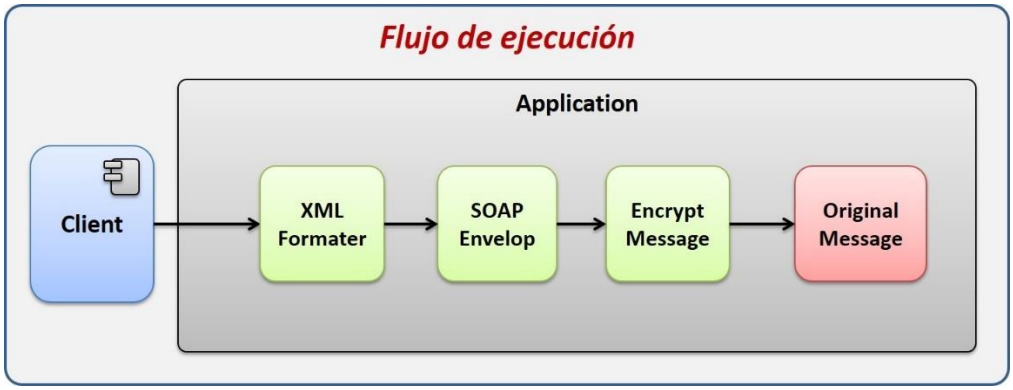


Ilustración 76: Proceso de transformación del mensaje.

En la imagen podemos apreciar cómo es que el mensaje debe de convertirse, cada paso de la conversión es controlado por un decorador que se agrega sobre el mensaje original (POJO). El reto será proponer una solución que nos permite agregar nueva funcionalidad sin afectar lo ya construido y si un nuevo proceso se debe de hacer en la conversión del mensaje, bastará con agregar un nuevo decorador.

La solución:

Debido a la naturaleza cambiante y que no queremos afectar los componentes ya construidos; cuando un cambio suceda, una solución por subclasficación (Subclases) queda descartada ya que al definir nuevas herencias habrá que implementar éstas dentro de la estructura y esto hace que se integren a lo ya construido.

Lo que se propone es implementar el patrón de diseño Decorator, que nos permitirá encapsular el mensaje dentro de otras clases decorativas que ayuden a transformar el mensaje.

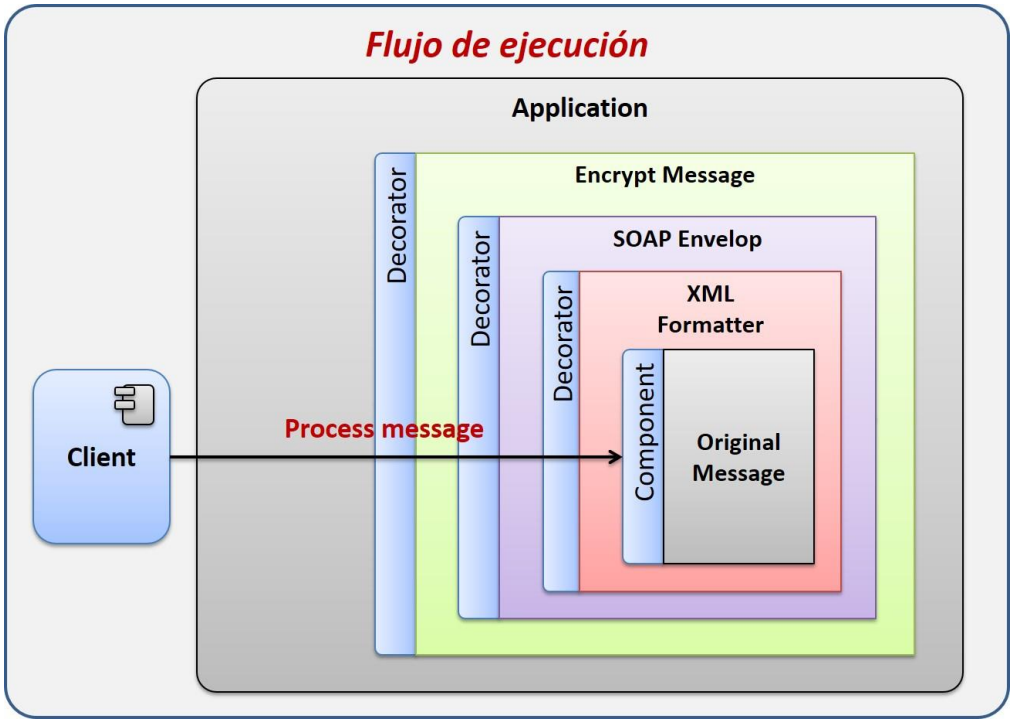


Ilustración 77: Estructura formada por la agregación.

En la imagen vemos que el cliente solicita el procesamiento del mensaje, sin embargo, éste no lo hace directamente con el mensaje original, en lugar de esto se lo solicita al decorador que encapsula al mensaje, el decorador por lo tanto hace lo mismo, le solicita al siguiente decorador que procese el mensaje; el

segundo decorador hace lo mismo pero esta vez con el mensaje original, cuando éste regresa su valor, XML Formater transforma el mensaje en XML, SOAP Envelop toma el mensaje en XML y lo encapsula dentro de un mensaje SOAP, Encrypt Message toma el mensaje SOAP y lo encripta. Cuando todo el proceso de decoración termine tendremos un mensaje totalmente transformado.
Un punto importante en este diseño es que si nos piden hacer algo adicional como por ejemplo poner usuario y *password* al mensaje, sólo nos quedará crear dinámicamente, un nuevo decorador y encapsular el mensaje.

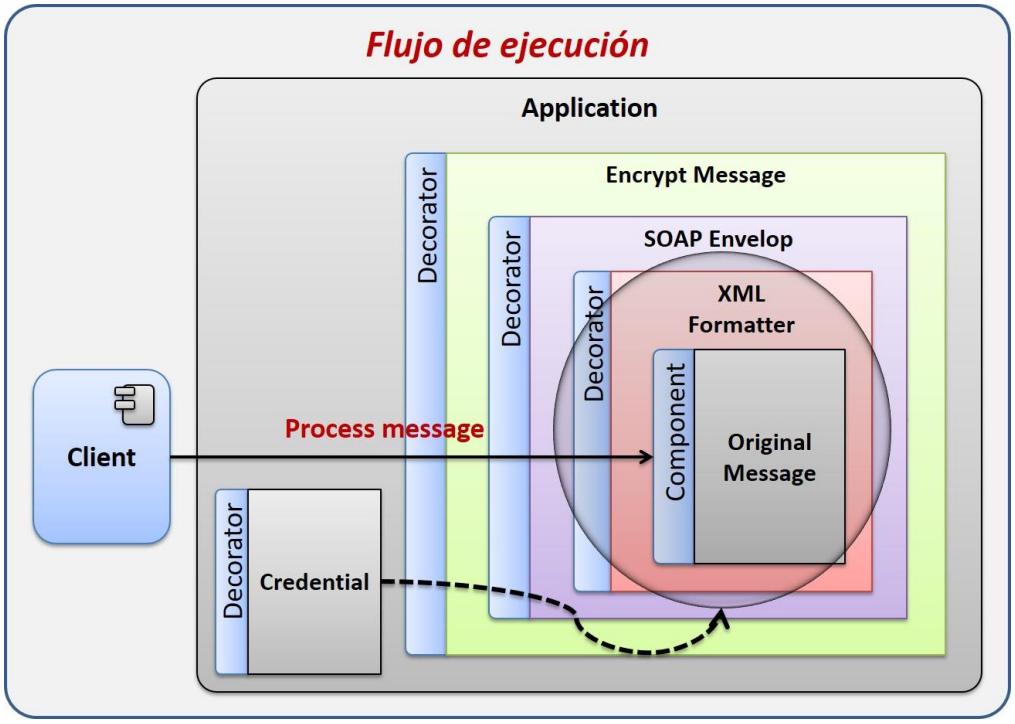


Ilustración 78: Inserción dinámica de un decorador.

En la imagen apreciamos cómo es posible agregar un nuevo decorador en cualquier parte de la estructura cambiando parcial o totalmente el mensaje de salida (los decoradores pueden agregarse en cualquier parte y en cualquier orden para tener resultados distintos). Por ejemplo, imaginemos que primero envolvemos el mensaje en SOAP y luego lo encriptamos, el resultado será un mensaje totalmente encriptado, sin embargo, si cambiamos el orden: primero encriptamos y luego envolvemos el mensaje en SOAP, tendremos un resultado

muy diferente ya que saldrá un mensaje SOAP totalmente entendible pero con el *Payload* encriptado, o simplemente podríamos quitar el paso de SOAP envelop y sólo encriptar el XML del mensaje original.

La implementación:

Ya que hemos visto cómo funciona un decorador iniciaremos con la implementación.

Ilustración 79: Estructura del proyecto Decorator.

En la imagen podemos ver los distintos paquetes que conforman el proyecto *Decorator* los cuales se explican a continuación:

- **Oscarblancarte.ipd.decorator:** Paquete por *default* del proyecto, contiene únicamente la clase ejecutable del proyecto: *DecoratorMain*.
- **Oscarblancarte.ipd.decorator.impl.decorators:** Paquete que contiene todos los decoradores del proyecto, incluyendo la interface que deben de implementar *MessageDecorator*.
- **Oscarblancarte.ipd.decorator.impl.message:** Paquete con todas las implementaciones de mensajes y la interface que deben de seguir *IMessage*.

La estructura de las imágenes se puede apreciar mejor en el siguiente cuadro:

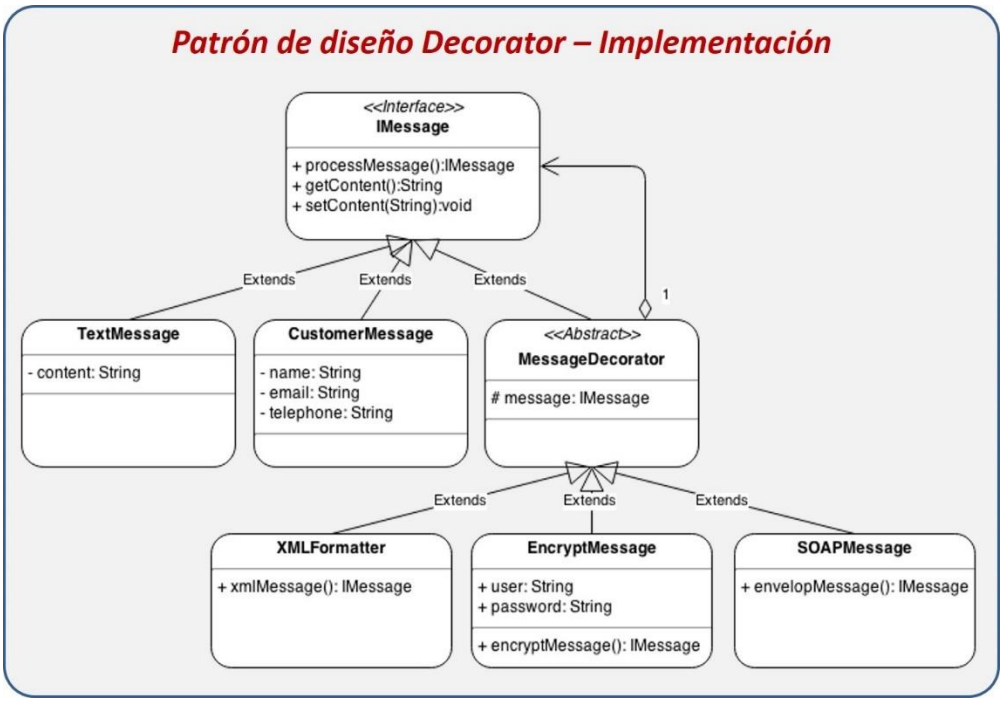


Ilustración 80: Diagrama de clases del proyecto Decorator.

Interface IMessage:

La interface *IMessage* corresponde al IComponente de la estructura del patrón, todos los compontes y decoradores que creemos deberán implementar esta interface.

```
1.  package oscarblancarte.ipd.decorator.impl.message;
2.
3.  public interface IMessage {
4.
5.      public IMessage processMessage();
6.      public String getContent();
7.      public void setContent(String content);
8.  }
```

Se definen los siguientes métodos:

- **processMessage:** Método utilizado para transformar el mensaje.
- **getContent:** Método que regresa el contenido del mensaje en *String*.
- **setContent:** Método que nos permite establecer el contenido del mensaje.

Clase CustomerMessage:

Implementación de *IMessage* la cual permite crear mensajes con los datos de un cliente.

```
1.  package oscarblancarte.ipd.decorator.impl.message;
2.
3.  public class CustomerMessage implements IMessage {
4.
5.      private String name;
6.      private String email;
7.      private String telephone;
8.
9.      public CustomerMessage(String name, String email, String telephone) {
10.         this.name = name;
11.         this.email = email;
12.         this.telephone = telephone;
13.     }
14.
15.     @Override
16.     public IMessage processMessage() {
17.         return this;
18.     }
19.
20.     @Override
21.     public String getContent() {
22.         return toString();
23.     }
24.
25.     @Override
26.     public String toString() {
27.         return "QueryCustomerMessage{" + "name=" + name + ", email="
28.             + email + ", telephone=" + telephone + '}';
29.     }
30.
31.     @Override
32.     public void setContent(String content) {
33.         throw new UnsupportedOperationException("Not supported yet.");
34.     }
35.     /** GEN and SET */
36. }
```

En la clase se definen tres propiedades, *name* (nombre del cliente), *email* y *telephone* (teléfono) los cuales se utilizarán para la creación del mensaje de texto.

Clase TextMessage:

Clase utilizada para crear los mensajes de texto. El mensaje es almacenado en la propiedad *content*.

```
1.  package oscarblancarte.ipd.decorator.impl.message;
2.
3.  public class TextMessage implements IMessage {
4.
5.      private String content;
6.
7.      public TextMessage() {
8.      }
9.
10.         @Override
11.         public IMessage processMessage() {
12.             return this;
13.         }
14.
15.         @Override
16.         public String getContent() {
17.             return content;
18.         }
19.
20.         @Override
21.         public void setContent(String content) {
22.             this.content = content;
23.         }
24.         /** GET and SET */
25.     }
```

Clase MessageDecorator:

Clase utilizada como base para crear a los decoradores, ésta recibe en el constructor un *IMessage*. Cada decorador que se cree deberá de heredar de esta clase e implementar los métodos *setContent* y *getContent* según se requiera.

```
1.  package oscarblancarte.ipd.decorator.impl.decorators;
2.
```

```
3. import oscarblancarte.ipd.decorator.impl.message.IMessage;
4.
5. public abstract class MessageDecorator implements IMessage {
6.     protected IMessage message;
7.
8.     public MessageDecorator(IMessage message) {
9.         this.message = message;
10.        }
11.
12.        @Override
13.        public void setContent(String content) {
14.            message.setContent(content);
15.        }
16.
17.        @Override
18.        public String getContent() {
19.            return message.getContent();
20.        }
21.    }
```

Clase EncryptMessage:

Clase decoradora que encripta un mensaje. La clase recibe en su constructor el usuario y *passw*ord requeridos para encriptar el mensaje.

```
1. package oscarblancarte.ipd.decorator.impl.decorators;
2.
3. import java.security.Key;
4. import javax.crypto.Cipher;
5. import javax.crypto.spec.SecretKeySpec;
6. import oscarblancarte.ipd.decorator.impl.message.IMessage;
7. import sun.misc.BASE64Encoder;
8.
9. public class EncryptMessage extends MessageDecorator {
10.
11.        private String user;
12.        private String password;
13.
14.        public EncryptMessage(String user, String password, IMessage message) {
15.            super(message);
16.            this.user = user;
17.            this.password = password;
18.        }
19.
20.        @Override
21.        public IMessage processMessage() {
22.            this.message = message.processMessage();
23.            encryptMessage();
```

```
24.             return message;
25.         }
26.
27.         private IMessage encryptMessage() {
28.             try {
29.                 Key key = new SecretKeySpec(password.getBytes(), "AES");
30.                 Cipher c = Cipher.getInstance("AES");
31.
32.                 c.init(Cipher.ENCRYPT_MODE, key);
33.                 byte[] encVal = c.doFinal(message.getContent().getBytes());
34.                 String encryptedValue = new BASE64Encoder().encode(encVal);
35.                 message.setContent(encryptedValue);
36.                 return message;
37.             } catch (Exception e) {
38.                 e.printStackTrace();
39.                 throw new RuntimeException();
40.             }
41.
42.         }
43.         /** GET and SET */
44.     }
```

Esta clase sobrescribe el método *procesMessage* para delegar el procesamiento al mensaje que encapsula y después encriptar el mensaje.

El método *encryptMessa* únicamente encripta el contenido del mensaje y regresa el mensaje ya transformado.

Clase SOAPEnvelopMessage:

Clase decoradora que encapsula el contenido del mensaje en formato SOAP.

```
1.  package oscarblancarte.ipd.decorator.impl.decorators;
2.
3.  import oscarblancarte.ipd.decorator.impl.message.IMessage;
4.
5.  public class SOAPEnvelopMessage extends MessageDecorator {
6.
7.      public SOAPEnvelopMessage(IMessage message) {
8.          super(message);
9.      }
10.
11.         @Override
12.         public IMessage processMessage() {
13.             message.processMessage();
```

```
14.             message =  envelopMessage();
15.             return message;
16.         }
17.
18.         private IMessage envelopMessage() {
19.             String soap = "<soapenv:Envelope xmlns:soapenv="
20.                     + "\"http://schemas.xmlsoap.org/soap/envelope/\" "
21.                     + "xmlns:ser=\"http://service.dishweb.cl.com/\">\n"
22.                     + "   <soapenv:Header/>\n"
23.                     + "   <soapenv:Body>\n"
24.                     + message.getContent()
25.                     + "\n"
26.                     + "   </soapenv:Body>\n"
27.                     + "</soapenv:Envelope>";
28.             message.setContent(soap);
29.             return message;
30.         }
31.
32.     }
```

El método *processMessage* delega el procesamiento al mensaje que encapsula para después tomar el mensaje devuelto y encapsularlo en un mensaje SOAP.

El método *envelopMessage* es el encargado de encapsular el contenido del mensaje en un documento SOAP.

Clase XMLFormatterDecorate:

Clase decoradora que convierte un mensaje a un mensaje en formato XML.

```
1.  package oscarblancarte.ipd.decorator.impl.decorators;
2.
3.  import java.io.ByteArrayOutputStream;
4.  import javax.xml.bind.*;
5.  import javax.xml.bind.Marshaller;
6.  import javax.xml.namespace.QName;
7.  import oscarblancarte.ipd.decorator.impl.message.*;
8.
9.  public class XMLFormatterDecorate extends MessageDecorator {
10.
11.         public XMLFormatterDecorate(IMessage message) {
12.             super(message);
13.         }
14.
15.         @Override
```

```
16.         public IMessage processMessage() {
17.             message = message.processMessage();
18.             message = xmlMessage();
19.             return message;
20.         }
21.
22.         private IMessage xmlMessage() {
23.             try {
24.                 JAXBContext jc = JAXBContext.newInstance(message.getClass());
25.                 JAXBElement<IMessage> je2 = new JAXBElement<IMessage>(
26.                     new QName(message.getClass().getName()),
27.                     (Class<IMessage>) message.getClass(), message);
28.                 Marshaller marshaller = jc.createMarshaller();
29.                 marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
30.                 ByteArrayOutputStream output = new ByteArrayOutputStream();
31.                 marshaller.marshal(je2, output);
32.                 return new TextMessage(new String(output.toByteArray()));
33.             } catch (Exception e) {
34.                 e.printStackTrace();
35.                 throw new RuntimeException("Error al convertir el XML");
36.             }
37.         }
38.     }
```

El método *processMessage* delega la responsabilidad de procesamiento del mensaje al decorador que encapsula para después convertir el mensaje en XML.

El método *xmlMessage* es el encargado de convertir un mensaje en XML.

Clase DecoratorMain:

Clase principal del proyecto que utilizaremos para comprobar la efectividad del patrón Decorator.

```
1.  package oscarblancarte.ipd.decorator;
2.
3.  import oscarblancarte.ipd.decorator.impl.decorators.SOAPEnvelopMessage;
4.  import oscarblancarte.ipd.decorator.impl.message.IMessage;
5.  import oscarblancarte.ipd.decorator.impl.decorators.EncryptMessage;
6.  import oscarblancarte.ipd.decorator.impl.message.CustomerMessage;
7.  import oscarblancarte.ipd.decorator.impl.decorators.XMLFormatterDecorate;
8.
9.  public class DecoratorMain {
10.
11.         public static void main(String[] args) {
```

```
12.
13.         CustomerMessage customerMessage = new CustomerMessage(
14.                 "Oscar Blancarte", "oscarblancarte3@gmail.com", "554433445566");
15.         System.out.println("Original Message ==> " + customerMessage);
16.
17.         IMessage message1 = new EncryptMessage("usuario", "HG58YZ3CR9123456",
18.                 new SOAPEnvelopMessage(
19.                     new XMLFormatterDecorate(customerMessage))).processMessage();
20.         System.out.println("message1 ====================================>\n"
21.                 + message1.getContent() + "\n\n");
22.
23.         IMessage message2 = new SOAPEnvelopMessage(
24.                 new EncryptMessage("usuario", "HG58YZ3CR9123456",
25.                     new XMLFormatterDecorate(customerMessage))).processMessage();
26.         System.out.println("message2 ====================================>\n"
27.                 + message2.getContent());
28.     }
29.
30. }
```

De las líneas 13 a la 15 podemos apreciar cómo es creado el mensaje original, el cual corresponde a los datos de un cliente.

En la línea 17 a hasta la 21, vemos cómo el mensaje original es encapsulado tras tres decoradores, los cuales se encargarán de convertir el mensaje en XML, después de encapsularlo en un mensaje SOAP y finalmente encriptar el mensaje. El procedimiento es el siguiente: *XMLFormatterDecorate* transformará el mensaje en XML, luego el mensaje en XML lo tomará *SOAPEnvelopMessage* y lo convertirá en un mensaje SOAP, finalmente *EncryptMessage* tomará el mensaje SOAP y lo encriptará. Como salida de esta ejecución esperamos un mensaje encriptado en su totalidad.

De las líneas 23 a la 27 haremos un procedimiento similar al anterior pero en esta ocasión cambiamos el orden en el que están los decoradores. El orden es *XMLFormatterDecorator*, *EncryptMessage*, *SOAPEnvelopMessage*, esto dará como resultado un mensaje SOAP no encriptado pero con su *payload* encriptado.

La ejecución:

Para probar la aplicación ejecutamos la clase *DecoratorMain* y se tendrá la siguiente salida:

```
run:
Original Message ==> QueryCustomerMessage{name=Oscar Blancarte,
email=oscarblancarte3@gmail.com, telephone=554433445566}
message1 ======================================>
yL6L0i2ti49snrUs07FR8dLlF8yK4quF/YucCRHvgM0ScPoxvlb/vZZ5NVToqCdVCtcR1W7oCQkK
DyXp/oEjrHmYPjkZ+0JJ1wAeGK8Xw6z89I/P+1fClOgoJC7EoSQot3upYxg8f1W97fjB+syo1HLb
3nKltfrhb0Rmjr10Aqheh5wKwWkjGyYydpLQ4z+802Gh8sn7j1UY5teBzvcHkIOR+DrU8hHWMBYs
2Pv08V8L+FSTUIO4W/xRDR4/axXyv/0CHHiejRW9f2PsBuL3SGzF8OvaBl7jInPZVbSTo13F9Qs3
1GH+/JQnMsV73+NTfPNhxdxuUjGF4YkMVTyXiVbXjpAGt62Zrq5Vix+XXKljXnt1y9G+4NmsDw3j
OWnphyGN1KkUquc4mHi5PbKIECkyTslG5RdQD0c/UC/1D7UolP6zN9Wm2d+uA80+PT23iIhozndJ
d2vadcZISsblp9BH3WwkGMQyTE21hnmheBK+fGx+V7XJfs2ZXX23m4v5GMK0aE3dadqVIkSVuO3n
YFO8xvGJKyd6IdpaCD7UtXVrCjnNgbApzBZ7+yDZIdHg3lv76M2sf3h0coIvmO970vqqwn39t/zq
/3f+GiiPoqkwexBQtiF2ZF6yRWbXT8yctzdV0v9oyrj41BUs/MAKgTh2U51oFw7p35dyQmKtusUB
9m2qIacg39eBfjKrxKrPGrMcijn/vHqzawbuNUNCGKBrroaHbTICeSwnEHmkr2mFTR+CbPmp8tpg
zuBIGHOiaduQHYusXjakpAQBnEEw8n1boAtGm4P8lCatDqBwDZPJQARYd+j3QfQC0EXumQRe


message2 ======================================>
<soapenv:Envelope xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://service.dishweb.cl.com/">
   <soapenv:Header/>
   <soapenv:Body>
AVWuXOhNueRAbzSzoFST0nkxFui31oGsYPT6mD9BMgc1kjOwqAxDYbEfuiLjAkr/eLFw0BZRuaoS
K9SXpbleZxqzHIo5/7x6s2sG7jVDQhiga66Gh20yAnksJxB5pK9phU0fgmz5qfLaYM7gSBhzordm
HdJ5vcBp5flsZ0t9Db2oCfrpSrqhEGzpfYxk89sjMy5Z99GvS65EVKwJUhmM//WuDNbeyFL58uDn
vJoHgI+nmmvo8Pwij5Eiz72q72gM5l5c1r8nICdYAhVC92m8DhBYjsuNFeoT1VIQk09Oz9W1qCzr
ReUgL5ukHOSGnPWdE8oqFlX2o4jzJiAkmxXtqO3PgwLbYOVmm10jrOi3yiFGlFD5p4GlXwVcRt2+
WdbIDgCK4NxSnr0IKXccYKkodbWvpNSaEqFis6f9u1I7p2Fr23LddqPbldZwxyetCd7Q6aaw+mI4
H7frylELbhIfuUpG2asQ9zMUVdOtrqz0bb1NZ8mT1Z8RTONxxEe5jyNz67XY+oJzNGYn/qJkh8bv
5NdRsAIRE1fDziPze6TGg/5WUk8STUGn119WeN55kPuT
   </soapenv:Body>
</soapenv:Envelope>
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 81: Resultado de la ejecución de DecoratorMain.

En la primera parte tenemos el mensaje original que corresponde a los datos de un cliente, nombre, correo electrónico y teléfono.

En la segunda parte tenemos el mensaje anterior pero ya fue procesado por tres decoradores que convirtieron el mensaje. El proceso tiene como resultado un mensaje totalmente encriptado.
La tercera parte es también el mensaje encriptado, por se ha cambiado el orden de los decoradores, por lo que se realiza primero el encriptamiento y después la envoltura en SOAP, es por eso que tenemos un mensaje parcialmente encriptado.

Siguientes pasos:

Como vimos, este patrón nos permite agregar nueva funcionalidad sin afectar las estructuras existentes, esto debido a que solamente es necesario incorporar un nuevo decorador en cualquier parte de la estructura o simplemente modificar el orden de los decoradores para tener una salida totalmente distinta.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios:

- Agreguemos un nuevo decorador que incruste en el *header* del mensaje SOAP, las credenciales del usuario (usuario y *password*).

- Agreguemos un nuevo decorador que inserte al pie de página los datos de la empresa que está mandado el correo; por ejemplo nombre de la empresa, dirección, teléfonos y correo electrónico (ficticios).

- Agreguemos un nuevo decorador que en lugar de convertir el mensaje en SOAP lo pase a formato ***REST***.

Patrón Facade

El patrón Facade (fachada) tiene la característica de ocultar la complejidad de interactuar con un conjunto de subsistemas proporcionando una interface de alto nivel, la cual se encarga de realizar la comunicación con todos los subsistemas necesarios. La fachada es una buena estrategia cuando requerimos interactuar con varios subsistemas para realizar una operación concreta ya que se necesita tener el conocimiento técnico y funcional para saber qué operaciones de cada subsistema tenemos que ejecutar y en qué orden, lo que puede resultar muy complicado cuando los sistemas empiezan a crecer demasiado.

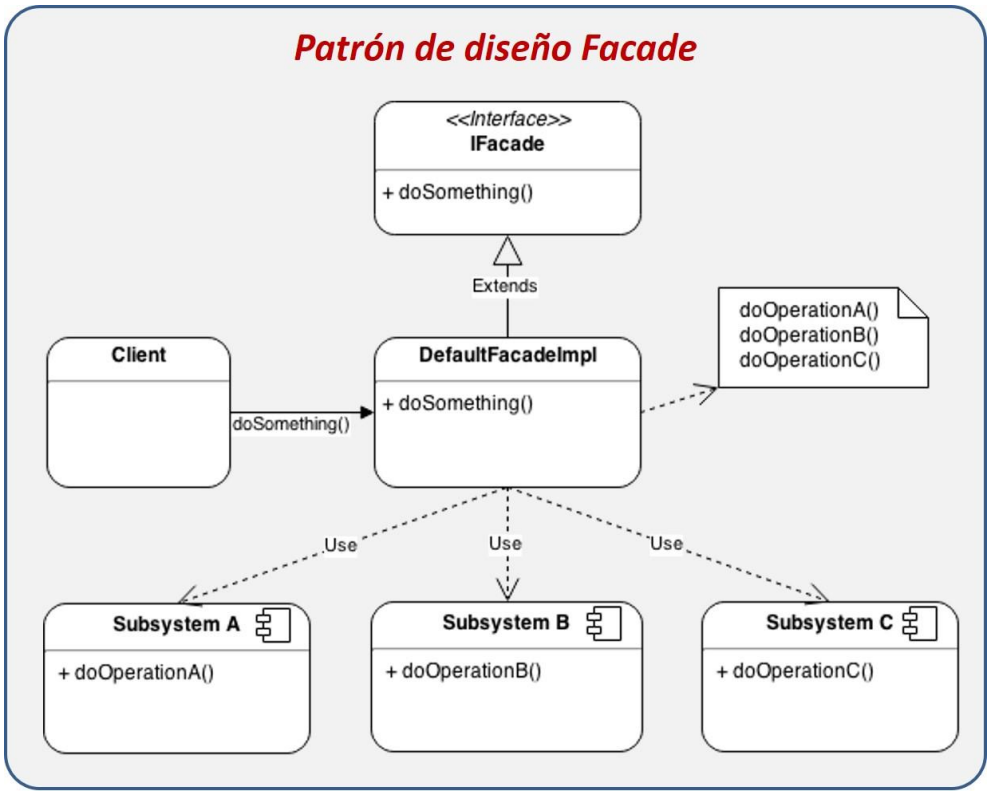


Ilustración 82: Estrucura del patrón Facade.

En la imagen podemos apreciar los componentes que integran el patrón Facade, los cuales se explican a continuación:

- **IFacade:** Proporciona una interface de alto nivel que oculta la complejidad de interactuar con varios sistemas para realizar una operación.
- **Client:** Sistema o evento que interactúa con la fachada.
- **DefaultFacadeImpl:** Representa la implementación de *IFacade* y se encarga de comunicarse con todos los subsistemas.
- **Subsystems:** Representa módulos o subsistemas que exponen interfaces para comunicarse con ellos.

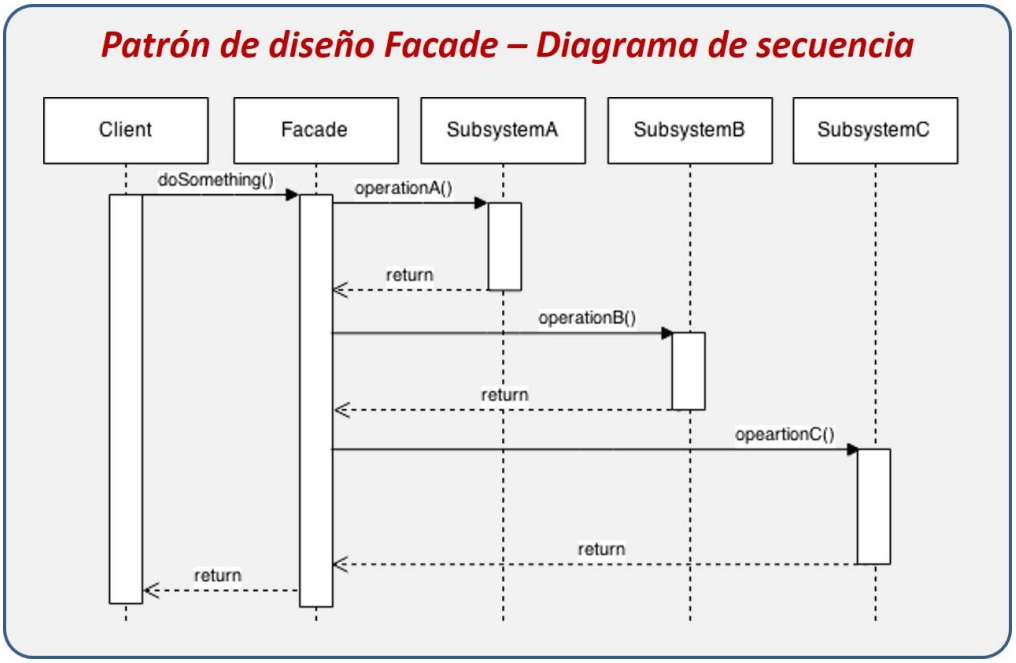


Ilustración 83: Diagrama de secuencia del patrón Facade.

El flujo de ejecución se interpreta de la siguiente manera:

1. El cliente invoca una operación de la fachada.
2. La fachada se comunica con el SubsystemA para realizar una operación.
3. La fachada se comunica con el SubsystemB para realizar una operación.
4. La fachada se comunica con el SubsystemC para realizar una operación.
5. La fachada responde al cliente con el resultado de la operación.

Como vemos en el diagrama, la fachada nos ahorra el esfuerzo de comunicarnos directamente con los subsistemas, ahorrándonos tener que conocer cómo funciona cada subsistema. Esto nos proporciona la venta de ejecutar una sola operación la cual nos arroje el resultado de la interacción con varios subsistemas.

Cuándo utilizarlo:

- Cuando interactuar con un conjunto de subsistemas es complicado, debido a que es necesario conocer los objetos necesarios para tener una interacción recíproca con cada sistema.
- Cuando queremos construir interfaces de alto nivel para nuestros usuarios.

El escenario:

Para este ejemplo imaginemos que nuestro cliente tiene un portal web desde el cual puede brindar ciertos servicios a sus clientes, sin embargo, nuestro cliente nos ha pedido que agreguemos una nueva funcionalidad que le permita hacer pagos en línea. El pago en línea no sólo implica hacer el cobro, si no que tenemos que afectar a varios subsistemas para que el pago sea aplicado correctamente y el cliente sea notificado que el pago se realizó exitosamente.

Nuestro cliente ya cuenta con algunos subsistemas con los cuales será necesario interactuar, los subsistemas que tiene el cliente son:

- **BankSystem:** Sistema de bancos que le permite realizar los cargos a los clientes por medio de los datos de la tarjeta de crédito o débito.
- **BillingSystem:** Sistema de facturación, este sistema tiene el detalle del saldo del cliente.
- **CRMSystem:** Sistema en donde gestiona la información del cliente.
- **EmailSystem:** Componente de software que utiliza para mandar correos electrónicos a sus clientes.

Cuando el cliente introduzca los datos de su tarjeta y confirme el pago, el sistema deberá hacer lo siguiente:

1. Ir al sistema de Bancos y aplicar el pago.
    a. El sistema valida que los bines (Prefijo del número de tarjeta que identifica) sean válidos.
2. Ir al sistema de facturación y aplicar el pago realizado.
    a. El sistema validó que el monto de pago no sea mayor al saldo del cliente.
3. Ir al sistema CRM y afectar el *status* del cliente cuando se realice un pago.
4. El *status* del cliente cambia a Activo únicamente cuando el saldo, después del pago, es menor o igual a 50$.
5. Ir al modulo de envio de correos electrónicos y notificar al cliente del pago realizado.

Bins: Corresponde a los primeros dígitos de una tarjeta que identifica de qué proveedor es (AMEX, VISA, MASTERCARD), así como el banco al que pertenece y si es de crédito o débito.

Otras reglas que se deben de tener en cuenta son que el cliente debe de existir para aplicar el pago y que el *status* de éste no sea 'Baja', ya que indica que este cliente ya canceló el servicio.

Un esquema que podríamos ver es el siguiente:

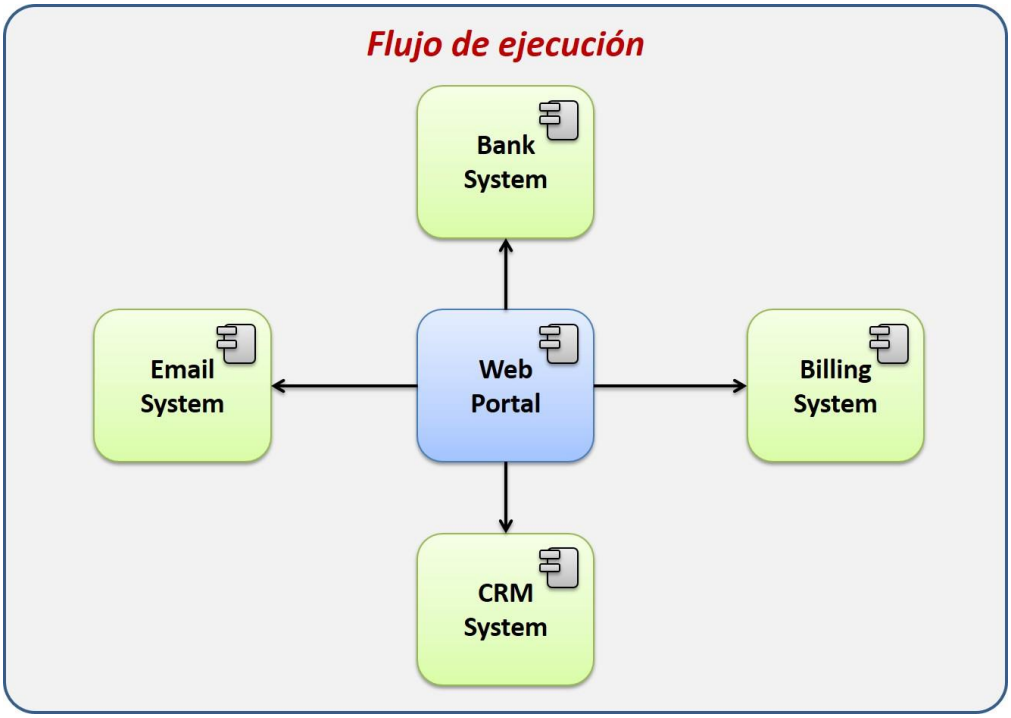


Ilustración 84: Escenario de comunicación de punto a punto.

El escenario que estamos viendo en la imagen es más común de lo que puedan imaginar, sin embargo, en la mayoría de los casos es un pésimo diseño, ya que el portal web debe de conocer la implementación de todos los subsistemas, debe conocer las operaciones a ejecutar y saber el orden en el que tienen que hacerlo. También tiene el riesgo de que si la interface de algunos de los subsistemas

cambia, afectará directamente al portal web, teniendo que modificar la aplicación, recompilar y subir el cambio.

Es por esta razón que nos ha solicitado diseñar una solución que le permita realizar el pago desde el portal web de una manera más segura y fácil de implementar.

La solución:

Ya vimos los requerimientos y ya sabemos que una solución de punto a punto no es recomendable en la mayoría de los casos, por eso la mejor estrategia será implementar el patrón Facade. Este patrón nos permitirá exponer una interface de alto nivel que sólo le solicite al portal web la información minina para realizar todos los pasos que vimos en el escenario.

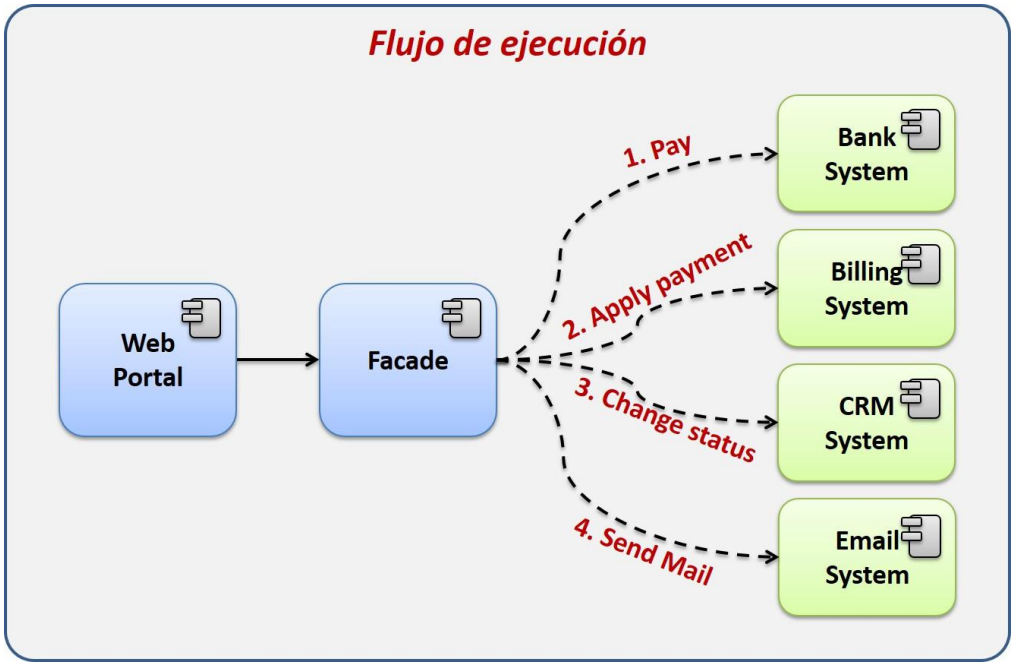


Ilustración 85: Implementación del pago con el patrón Facade.

Si vemos la imagen podemos apreciar una gran diferencia. En lugar de que nuestra aplicación se encuentre en medio de todos los subsistemas interactuando con todos, mejor colocamos la aplicación en un extremo y se agrega una fachada que expondrá la funcionalidad para realizar todos los pasos del pago, de esta forma no es necesario saber qué pasos implica realizar un pago y cualquier cambio en las interfaces de los subsistemas afectará únicamente a la fachada y no al portal web.

La implementación:

Iniciaremos con la implementación analizando la estructura del proyecto:

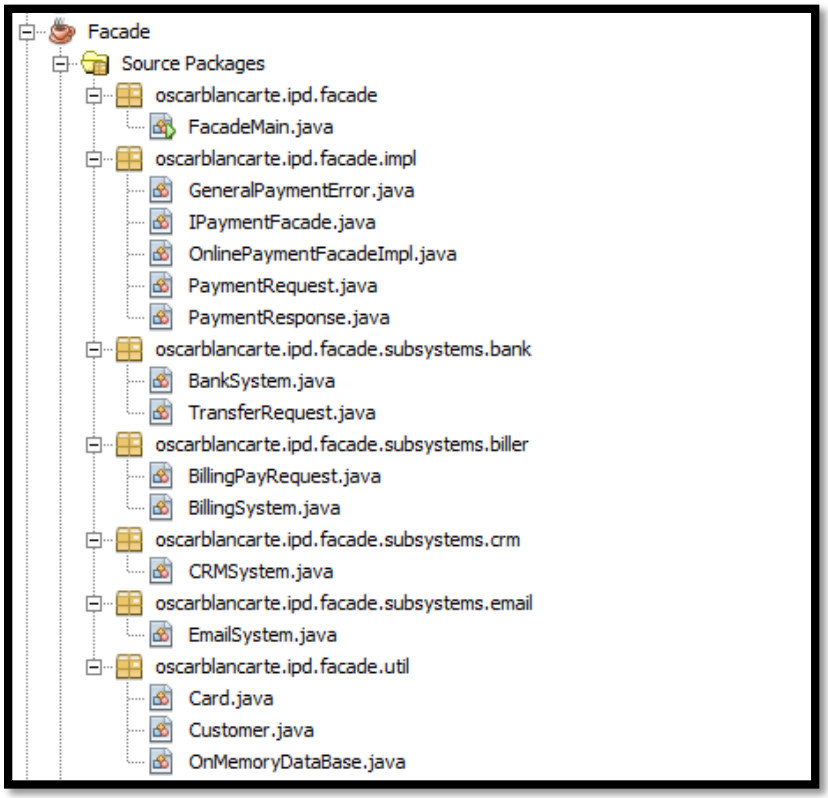


Ilustración 86: Estructura del proyecto Facade.

Los paquetes que componente el proyecto son:

- **oscarblancarte.ipd.facade:** Paquete *default* del proyecto, el cual tiene la clase ejecutable del proyecto *FacadeMain*.
- **oscarblancarte.ipd.facade.impl:** Contiene las clases e interfaces necesarias para crear la fachada de pago.
- **oscarblancarte.ipd.facade.subsystems.bank:** Contiene las clases del subsistema de bancos.
- **oscarblancarte.ipd.facade.subsystems.biller:** Contiene las clases del subsistema de facturación.

- **oscarblancarte.ipd.facade.subsystems.crm:** Contiene las clases del subsistema de CRM.
- **oscarblancarte.ipd.facade.subsystems.email:** Contiene las clases del subsistema de envío de correos electrónicos.
- **Oscarblancarte.ipd.facade.util:** Contiene clases de utilidad como *OnMemoryDataBase*, la cual pretende simular una base de datos donde guardamos los clientes y los datos de las tarjetas.

NOTA: Normalmente las clases de los componentes de banca, facturación, etcétera; tendrían que ser proyectos independientes pero por practicidad se pone en este mismo proyecto.

Las relaciones de las clases se pueden apreciar mejor en la siguiente imagen:

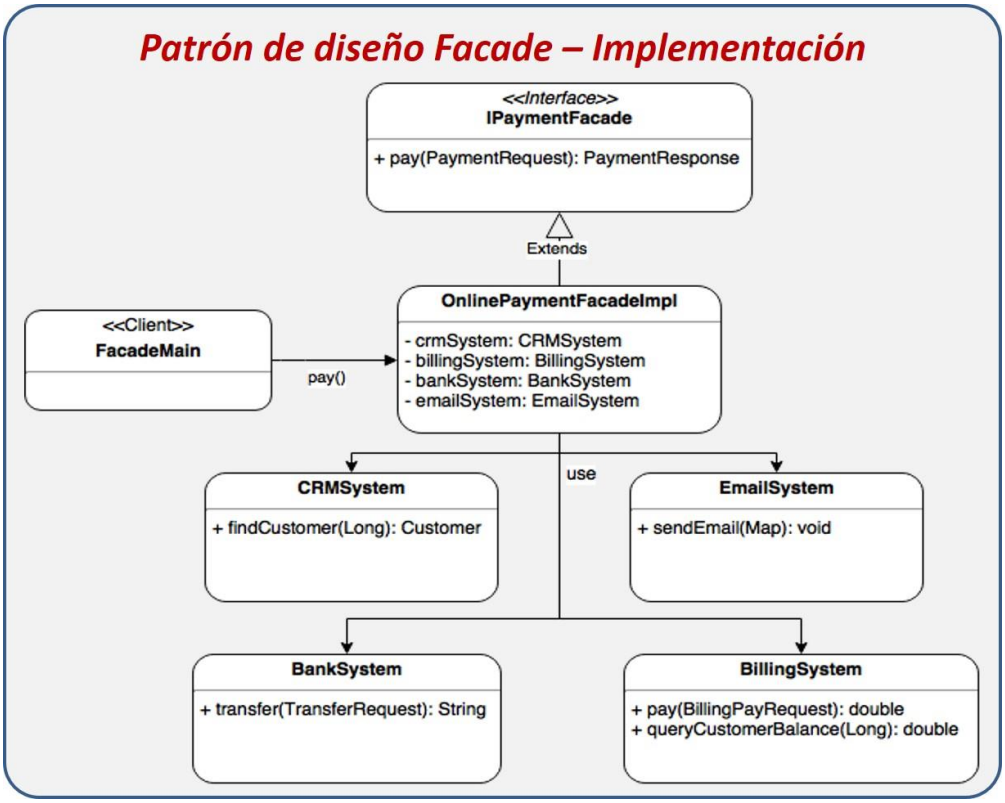


Ilustración 87: Diagrama de clases del proyecto Facade.

El diagrama muestra cómo las clases están relacionadas con el resto, sólo se agregaron las de mayor importancia para el ejemplo, las cuales corresponden a las clases de la fachada y los subsistemas.

Interface IPaymentFacade:

La interface *IPaymentFacade* define la estructura que la fachada de pago en línea deberá de tener.

```
1. package oscarblancarte.ipd.facade.impl;
2.
3. public interface IPaymentFacade {
4.     public PaymentResponse pay(PaymentRequest paymentRequest)throws GeneralPaymentError;
5. }
```

Se define un único método (pay) que recibe los datos mínimos para hacer un pago, como es el ID del cliente y los datos de la tarjeta.

Clase GeneralPaymentError:

Clase de *exception custom* que será lanzada cuando se presenta algún error en el servicio de pago.

```
1.  package oscarblancarte.ipd.facade.impl;
2.
3.  public class GeneralPaymentError extends Exception {
4.
5.      public GeneralPaymentError(String message) {
6.          super(message);
7.      }
8.  }
```

Clase OnlinePaymentFacadeImpl:

La clase *OnlinePaymentFacadeImpl* representa la fachada concreta. En ésta se implementa el método *pay*, la cual es una operación de alto nivel, en ella se realizan todas las acciones para aplicar el pago en todos los subsistemas.

En la clase podemos ver las variables crmSystem, billingSystem, bankSystem y emailSenderSystem los cuales representa las interfaces expuestas por los subsistemas para comunicarse con ellos.

```
1.  package oscarblancarte.ipd.facade.impl;
2.
3.  import java.util.HashMap;
4.  import oscarblancarte.ipd.facade.subsystems.bank.*;
5.  import oscarblancarte.ipd.facade.subsystems.biller.*;
6.  import oscarblancarte.ipd.facade.subsystems.crm.CRMSystem;
7.  import oscarblancarte.ipd.facade.subsystems.email.EmailSystem;
8.  import oscarblancarte.ipd.facade.util.*;
9.
10.     public class OnlinePaymentFacadeImpl implements IPaymentFacade {
11.
12.         private static final CRMSystem crmSystem = new CRMSystem();
13.         private static final BillingSystem billingSyste = new BillingSystem();
14.         private static final BankSystem bankSyste = new BankSystem();
15.         private static final EmailSystem emailSenderSystem = new EmailSystem();
16.
17.         @Override
18.         public PaymentResponse pay(PaymentRequest request)
19.                 throws GeneralPaymentError {
20.             Customer customer = crmSystem.findCustomer(request.getCustomerId());
21.             //Validate Set
22.             if(customer==null){
23.                 throw new GeneralPaymentError("El cliente con id '"
24.                         +request.getCustomerId()+"' no existe.");
25.             }else if("Baja".equals(customer.getStatus())){
26.                 throw new GeneralPaymentError("El cliente con id '"
27.                         +request.getCustomerId()+"' está dado de baja.");
28.             }else if(request.getAmmount() >
29.                     billingSyste.queryCustomerBalance(customer.getId())){
30.                 throw new GeneralPaymentError("Se está intentado realizar un pago "
31.                         + "\n\tmayor al saldo del cliente");
32.             }
33.
34.             //Pago bancario. Se realiza el cargo a la tarjeta.
35.             TransferRequest transfer = new TransferRequest(
36.                     request.getAmmount(),request.getCardNumber(),
37.                     request.getCardName(), request.getCardExpDate(),
38.                     request.getCardNumber());
39.             String payReference = bankSyste.transfer(transfer);
40.
41.             //Afectación del saldo en el sistema de facturación.
42.             BillingPayRequest billingRequest = new BillingPayRequest(
43.                     request.getCustomerId(), request.getAmmount());
44.             double newBalance = billingSyste.pay(billingRequest);
45.
```

```
46.         //El cliente se reactiva si el nuevo saldo es menor de 51$
47.         String newStatus = customer.getStatus();
48.         if(newBalance<=50){
49.             OnMemoryDataBase.changeCustomerStatus(request.getCustomerId(), "Activo");
50.             newStatus = "Activo";
51.         }
52.
53.         //Envío de la confirmación de pago por Email.
54.         HashMap<String,String> params = new HashMap<>();
55.         params.put("$name", customer.getName());
56.         params.put("$ammount", request.getAmmount()+"");
57.         params.put("$newBalance", newBalance+"");
58.         String number = request.getCardNumber();
59.         String subfix = number.substring(number.length()-4, number.length());
60.         params.put("$cardNumber", subfix);
61.         params.put("$reference", payReference);
62.         params.put("$newStatus", newStatus);
63.         emailSenderSystem.sendEmail(params);
64.
65.         return new PaymentResponse(payReference, newBalance, newStatus);
66.
67.     }
68. }
```

Los pasos para aplicar el pago en todos los subsistemas son los siguientes:

- En la línea 21 se consultan los datos del cliente por medio de la interface de CRM.
- En la línea 22 validamos que el cliente exista, de no ser asi lanzamos una excepción con el error.
- En la línea 25 validamos que el cliente no esté dado de baja, de ser asi se lanza un error al aplicar el pago.
- En la línea 28 se valida que el monto del pago no sea mayor al saldo del cliente.
- En la línea 35 a la 39 se realiza el pago al sistema de bancos, para lo cual se mandan los datos de la tarjeta.
- De las líneas 41 a la 44 se aplica el pago en el sistema de facturación, para lo cual se manda el ID del cliente y el monto a aplicar.
- De las líneas 46 a la 51 se cambia el *status* del cliente a Activo siempre y cuando su saldo no sea mayor de 50$.

- De las líneas 53 a la 63 se manda el correo electrónico al cliente informándole que se realizó su pago exitosamente y le informamos los datos referentes al pago; como número de confirmación, el nuevo saldo y su nuevo *status* tras haber realizado el pago.

Finalmente, en la línea 65 se responde con el número de confirmación del pago, el nuevo saldo y el estatus.

Como vemos, esta operación se encarga de hacer todo por nosotros sin conocer absolutamenta nada sobre la implementación de los subsistemas.

Clase PaymentRequest:

Esta clase corresponde al *request* de la operación *pay* de la fachada, la cual tiene los datos mínimos para aplicar un pago, los datos requeridos son el ID del cliente y los datos de la tarjeta.

```
1.  package oscarblancarte.ipd.facade.impl;
2.
3.  public class PaymentRequest {
4.
5.      private Long customerId;
6.      private double ammount;
7.      private String cardNumber;
8.      private String cardName;
9.      private String cardExpDate;
10.         private String cardSecureNum;
11.
12.         public PaymentRequest() {
13.         }
14.
15.         /** GET and SET */
16.
17.     }
```

Clase PaymentResponse:

Corresponde a la respuesta de la operación *pay*. La respuesta incluye número de confirmación de pago, nuevo *status* y saldo.

```
1.  package oscarblancarte.ipd.facade.impl;
2.
3.  public class PaymentResponse {
4.
5.      private String paymentConfirmNumber;
6.      private double newBalance;
7.      private String customerStatus;
8.
9.      public PaymentResponse(String paymentConfirmNumber,
10.             double newBalance, String customerStatus) {
11.
12.             this.paymentConfirmNumber = paymentConfirmNumber;
13.             this.newBalance = newBalance;
14.             this.customerStatus = customerStatus;
15.         }
16.
17.         /** GET and SET */
18.     }
```

Clase Card:

Clase utilizada para representar la configuración de una tarjeta. En la clase se incluye el prefijo, la compañía (AMEX, Mastercard, VISA) y el tipo de tarjeta (crédito, debito).

```
1.  package oscarblancarte.ipd.facade.util;
2.
3.  public class Card {
4.      private String prefix;
5.      private String company;
6.      private String cardType;//{Credit|Debit}
7.
8.      public Card(String prefix, String company, String cardType) {
9.          this.prefix = prefix;
10.             this.company = company;
11.             this.cardType = cardType;
12.         }
13.
14.         /** GET and SET */
15.     }
```

Clase OnMemoryDataBase:

Esta clase intenta simular una base de datos donde se guardan los datos de los clientes y las tarjetas. Básicamente se tienen dos colecciones (*CUSTOMER_MAP* y *CARD_BINS*) en donde se guardan los clientes y bines del sistema.

```
1.  package oscarblancarte.ipd.facade.util;
2.
3.  import java.util.HashMap;
4.
5.  public class OnMemoryDataBase {
6.
7.      private static final HashMap<Long, Customer> CUSTOMER_MAP = new HashMap<>();
8.      private static final HashMap<String, Card> CARD_BINS = new HashMap<>();
9.
10.         static {
11.             CUSTOMER_MAP.put(1L, new Customer(1L, "Oscar Blancarte", 500, "Suspendido"));
12.             CUSTOMER_MAP.put(2L, new Customer(2L, "Juan Pérez", 300, "Suspendido"));
13.             CUSTOMER_MAP.put(3L, new Customer(3L, "Adrián López", 100, "Activo"));
14.             CUSTOMER_MAP.put(4L, new Customer(4L, "Melisa Mares", 100, "Baja"));
15.
16.             CARD_BINS.put("123", new Card("123", "VISA", "Credit"));
17.             CARD_BINS.put("234", new Card("234", "MASTERCARD", "Debit"));
18.             CARD_BINS.put("345", new Card("345", "AMEX", "Credit"));
19.         }
20.
21.         public static Customer findCustomerById(Long id) {
22.             return CUSTOMER_MAP.get(id);
23.         }
24.
25.         public static void changeCustomerStatus(Long id, String newStatus) {
26.             Customer customer = findCustomerById(id);
27.             customer.setStatus(newStatus);
28.             System.out.println("Cambio de status del cliente '" + customer.getName()
29.                     + "'" + newStatus);
30.         }
31.
32.         public static boolean validateCardBins(String creditCardPrefix) {
33.             if (CARD_BINS.containsKey(creditCardPrefix)) {
34.                 String company = CARD_BINS.get(creditCardPrefix).getCompany();
35.                 System.out.println("Tarjeta válida > '" + creditCardPrefix + "', "
36.                         + company + "\n");
37.                 return true;
38.             } else {
39.                 System.out.println("Tarjeta inválida >\n");
40.                 return false;
41.             }
42.         }
43.
```

```
44.         public static String getCardCompany(String creditCardPrefix) {
45.             if (CARD_BINS.containsKey(creditCardPrefix)) {
46.                 return CARD_BINS.get(creditCardPrefix).getCompany();
47.             }
48.             throw new RuntimeException("Tarjeta no existe");
49.         }
50.     }
```

De la línea 10 a la 18 inicializamos a los clientes y tarjetas por *default*.

Tenemos los siguientes métodos:

- **findCustomerById:** Busca un cliente por medio del ID.
- **changeCustomerStatus:** Cambia el *status* de un cliente.
- **validateCardBins:** Verifica si el prefijo de la tarjeta es válido según la configuración de bines definidas en la colección CARD_BINS.
- **getCardCompany:** Regresa la compañía a la que pertenece la tarjeta según la configuración de Bines.

Clase TransferRequest:

Esta clase representa el *request* para el sistema bancario, es la que recibe los datos de la tarjeta y el monto del cargo.

```
1.  package oscarblancarte.ipd.facade.subsystems.bank;
2.
3.  public class TransferRequest {
4.
5.      private double ammount;
6.      private String cardNumber;
7.      private String cardName;
8.      private String cardExpDate;
9.      private String cardSecureNum;
10.
11.         public TransferRequest(double ammount, String cardNumber,
12.             String cardName, String cardExpDate, String cardSecureNum) {
13.
14.             this.ammount = ammount;
15.             this.cardNumber = cardNumber;
16.             this.cardName = cardName;
17.             this.cardExpDate = cardExpDate;
18.             this.cardSecureNum = cardSecureNum;
```

```
19.         }
20.
21.         /** GET and SET */
22.     }
```

Clase BankSystem:

Clase que representa el sistema bancario, éste ofrece un método llamado *transfer* que recibe los datos para realizar el cargo a la tarjeta.

```
1.  package oscarblancarte.ipd.facade.subsystems.bank;
2.
3.  import java.util.UUID;
4.  import oscarblancarte.ipd.facade.impl.GeneralPaymentError;
5.  import oscarblancarte.ipd.facade.util.OnMemoryDataBase;
6.
7.  public class BankSystem {
8.
9.      public String transfer(TransferRequest request) throws GeneralPaymentError {
10.             String cardNumerPrefix = request.getCardNumber().substring(0, 3);
11.             if (!OnMemoryDataBase.validateCardBins(cardNumerPrefix)) {
12.                 throw new GeneralPaymentError("Tarjeta inválida.");
13.             }
14.             String cardCompany = OnMemoryDataBase.getCardCompany(cardNumerPrefix);
15.             if ("AMEX".equals(cardCompany) && request.getCardNumber().length() != 15) {
16.                 throw new GeneralPaymentError("El número de la tarjeta es inválido");
17.             } else if (("VISA".equals(cardCompany) || "MASTERCARD".equals(cardCompany))
18.                     && request.getCardNumber().length() != 16) {
19.                 throw new GeneralPaymentError("El número de la tarjeta es inválido");
20.             }
21.             String number = request.getCardNumber();
22.             String cardNumerSubfix = number.substring(number.length()-4, number.length());
23.             System.out.println("Se ha realizado un cargo al cliente '"
24.                     + request.getCardName() + "' \n"
25.                     + "\tpor el monto de '" + request.getAmmount() + "' a la tarjeta "
26.                     + "terminación '"+cardNumerSubfix+"'.\n");
27.
28.             return UUID.randomUUID().toString();
29.         }
30.     }
```

Los pasos para realizar el cargo son los siguientes:

- En la línea 11 se valida la tarjeta contra los bines para asegurarnos que sea válida.
- En la línea 19 se valida si la tarjeta es AMEX, entonces el número de tarjeta debe de ser de 15 dígitos.
- En la línea 17 se valida si la tarjeta es VISA o Mastercard, entonces debe de ser de 16 digitos.
- De las líneas 23 a la 26 se imprime el detalle del cargo, monto cargado, el nombre del cliente y la terminación de la tarjeta. Finalmente, se regresa la confirmación del pago.

Clase BillingSystem:

Representa al sistema de facturación.

```
1.  package oscarblancarte.ipd.facade.subsystems.biller;
2.
3.  import oscarblancarte.ipd.facade.util.Customer;
4.  import oscarblancarte.ipd.facade.util.OnMemoryDataBase;
5.
6.  public class BillingSystem {
7.
8.      public double queryCustomerBalance(Long customerId) {
9.          Customer customer = OnMemoryDataBase.findCustomerById(customerId);
10.         return customer.getBalance();
11.     }
12.
13.     public double pay(BillingPayRequest billingPay) {
14.         Customer customer = OnMemoryDataBase.findCustomerById(billingPay.getCustomerId());
15.         customer.setBalance(customer.getBalance() - billingPay.getAmount());
16.         System.out.println("Pago aplicado al cliente '"+customer.getName()+"', "
17.                 + "el nuevo saldo es '"+customer.getBalance()+"'");
18.         return customer.getBalance();//new Balance.
19.     }
20. }
```

Se definen los siguientes métodos.

- **queryCustomerBalance:** Método que regresa el saldo del cliente.

- **Pay:** Método utilizado para aplicar un pago restando el saldo del cliente.

Clase BillingPayRequest:

Clase que representa el *request* del sistema de facturación, el cual recibe el ID del cliente y el monto a aplicar.

```
1.  package oscarblancarte.ipd.facade.subsystems.biller;
2.
3.  /**
4.
5.
6.   */
7.  public class BillingPayRequest {
8.
9.      private Long customerId;
10.     private double amount;
11.
12.     public BillingPayRequest(Long customerId, double amount) {
13.         this.customerId = customerId;
14.         this.amount = amount;
15.     }
16.     /** GET and SET */
17. }
```

Class CRMSystem:

La clase *CRMSystem* representa al Sistema CRM y es utilizado para consultar la información del cliente.

```
1.  package oscarblancarte.ipd.facade.subsystems.crm;
2.
3.  import oscarblancarte.ipd.facade.util.Customer;
4.  import oscarblancarte.ipd.facade.util.OnMemoryDataBase;
5.
6.  public class CRMSystem {
7.
8.      public Customer findCustomer(Long customerId){
9.          return OnMemoryDataBase.findCustomerById(customerId);
10.     }
11. }
```

El metodo *findCustomer* busca al cliente por medio del ID.

Clase EmailSystem:

Esta clase representa al módulo de envío de correos electrónicos.

```
1.  package oscarblancarte.ipd.facade.subsystems.email;
2.
3.  import java.util.Map;
4.
5.  public class EmailSystem {
6.
7.      public void sendEmail(Map<String, String> values) {
8.          String templete = "\n*************************************\n"
9.                  + "|To: $name\n"
10.                 + "|from: FacadeSystem\n"
11.                 + "|\n"
12.                 + "|Muchas gracias por utilizar el servicio en \n"
13.                 + "|línea para realizar sus pagos.\n"
14.                 + "|\n"
15.                 + "|Hace un momento acabamos de recibir un pago:\n"
16.                 + "|\n"
17.                 + "|Monto del pago: $ammount.\n"
18.                 + "|Nuevo Saldo: $newBalance.\n"
19.                 + "|Tarjeta terminación: $cardNumber\n"
20.                 + "|Referencia de pago: $reference\n"
21.                 + "|Nuevos status: $newStatus\n"
22.                 + "|\n"
23.                 + "|Gracias por su preferencia.\n"
24.                 + "|\n"
25.                 + "|Este correo no deberá ser contestado ya que \n"
26.                 + "|ha sido enviado por un proceso automático"
27.                 + "\n*************************************";
28.
29.         for(String str : values.keySet()){
30.             templete = templete.replace(str, values.get(str));
31.         }
32.
33.         System.out.println(templete);
34.     }
35. }
```

El método *sendEmail* simula enviar un correo electrónico al cliente con los datos del pago, su nuevo saldo y estatus.

Clase FacadeMain:

Esta clase la utilizaremos para probar el programa y validar cómo es que una fachada reduce, en gran medida, el esfuerzo necesario para hacer una operación que requiere de varios subsistemas.

```
1.  package oscarblancarte.ipd.facade;
2.
3.  import oscarblancarte.ipd.facade.impl.IPaymentFacade;
4.  import oscarblancarte.ipd.facade.impl.OnlinePaymentFacadeImpl;
5.  import oscarblancarte.ipd.facade.impl.PaymentRequest;
6.
7.  public class FacadeMain {
8.
9.      public static void main(String[] args) {
10.         PaymentRequest request = new PaymentRequest();
11.         request.setAmmount(500);
12.         request.setCardExpDate("10/2015");
13.         request.setCardName("Oscar Blancarte");
14.         request.setCardNumber("1234567890123456");
15.         request.setCardSecureNum("345");
16.         request.setCustomerId(1L);
17.
18.         try {
19.             IPaymentFacade paymentFacade = new OnlinePaymentFacadeImpl();
20.             paymentFacade.pay(request);
21.         } catch (Exception e) {
22.             System.out.println(e.getMessage());
23.         }
24.     }
25. }
```

De la línea 10 a la 16 se crea la solicitud para la fachada, a la cual se le envía el monto a pagar (500$), el ID del cliente (1) y los datos de la tarjeta.
En las líneas 19 y 20 se envía la solicitud de pago a la fachada y se obtiene la respuesta de la misma.

La ejecución:

Para probar el programa ejecutaremos la clase *FacadeMain* para ver el siguiente resultado:

```
run:
Tarjeta valida > '123', VISA

Se ha realizado un cargo al cliente 'Oscar Blancarte'
        por el monto de '500.0' a la tarjeta terminación '3456'.

Pago aplicado al cliente 'Oscar Blancarte', el nuevo saldo es '0.0'
Cambio de status del cliente 'Oscar Blancarte'Activo

*************************************
|To: Oscar Blancarte
|from: FacadeSystem
|
|Muchas gracias por utilizar el servicio en
|linea para realizar sus pagos.
|
|Hace un momento acabamos de recibir un pago:
|
|Monto del pago: 500.0.
|Nuevo Saldo: 0.0.
|Tarjeta terminación: 3456
|Referencia de pago: b61c64d8-b17a-4929-81d4-b1849b75e4da
|Nuevos status: Activo
|
|Gracias por su preferencia.
|
|Este correo no deberá ser contestado ya que
|ha sido enviado por un proceso automático
*************************************
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 88: Resultado de la ejecución.

La ejecución se realiza de la siguiente manera:

- Se confirma que la tarjeta es VISA y que es válida.
- Se ha realizado un cargo a la tarjeta con terminacón 3456 por un monto de 500$.

- El pago ha sido aplicado al cliente y vemos el nuevo saldo que es de 0.0$.
- El *status* del cliente ha cambiado a Activo.
- Y finalmente, vemos el correo de confirmación del pago en donde aparecen los detalles del mismo.

Siguientes pasos:

Si nos damos cuenta con unos cuantos parámetros hemos logrado hacer todos los pasos para aplicar un pago en varios subsistemas sin el funcionamiento interno de cada uno de éstos. Además, es importante resaltar que si un paso extra es requerido para aplicar el pago, sólo se tendría que mover la fachada sin afectar a la aplicación.

Para fortalecer los conocimientos adquirIdos realizaremos los siguientes ejercicios:

- Integremos un nuevo subsistema llamado *RewardSystem* el cual dé puntos a los clientes por sus compras, cada 100 pesos pagados le dará al cliente 10 puntos electrónicos.

- Continuando con el punto anterior, modifiquemos la aplicación para que el sistema, antes de realizar el cargo al banco, valide si la compra se puede pagar con los puntos. Cada 10 puntos equivalen a 1$. Si los puntos alcanzan a cubrir la compra en su totalidad se deberán usar y no realizar el cargo a la tarjeta.

- Agreguemos un nuevo subsistema, el cual simule enviar un SMS al celular del cliente cuando un pago se ha realizado.

Patrón Flyweight

Flyweight es un patrón que centra su atención en la construcción de objetos ligeros, mediante la abstracción de las partes reutilizables que pueden ser compartidas con otros objetos, esto con el fin de que en lugar de crear objetos cada vez que sea requerido, podamos reutilizar objetos creados por otras instancias logrando con ello reducir en gran medida la capacidad de memoria requerida por la aplicación.
Este patrón es utilizado cuando la optimización de los recursos es algo primordial ya que elimina la redundancia de objetos con propiedades idénticas.

Muy a menudo creamos objetos que son exactamente iguales y en muchos de los casos no nos damos cuenta ya que esto no afecta a los recursos del sistema. Sin embargo, ¿qué pasaría si estos objetos idénticos empiezan a saturar nuestra memoria? Es allí cuando ponemos atención en el asunto y tratamos de dar una solución. *Flyweight* centra su estrategia en dividir los objetos en dos partes:

- **Estado Intrínseco:** Son aquellas partes de un objeto que se repiten y que muchos de los mismos objetos tienen en común. Esta parte por lo general no cambia, o si cambia, afecta a todos los objetos por igual. Esta parte es la que se separa del objeto original con el fin de compartirla con el resto de los demás objetos.

- **Estado Extrínseco:** Corresponde a la parte única de cada objeto, la cual, es distinta a todas las demás por lo que no es candidata a ser reutilizada o compartida.

NOTA: *Flyweight* se apoya en el patrón *Factory Method* para su implementación.

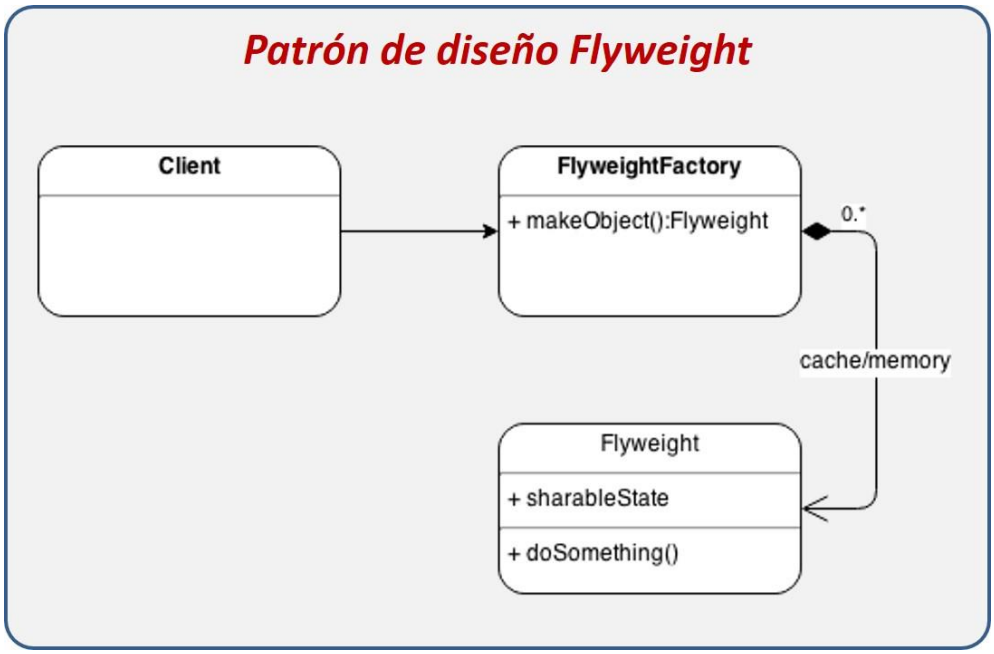


Ilustración 89: Estructura del patrón Flyweight.

Los componentes que conforman el patrón son:

- **Client:** Objeto que dispara la ejecución.
- **FlyweightFactory:** Fábrica que utilizaremos para crear los objetos *Flyweight* u objetos ligeros.
- **Flyweight:** Corresponde a los objetos que deseamos reutilizar con el fin de que nuestros objetos sean más ligeros.

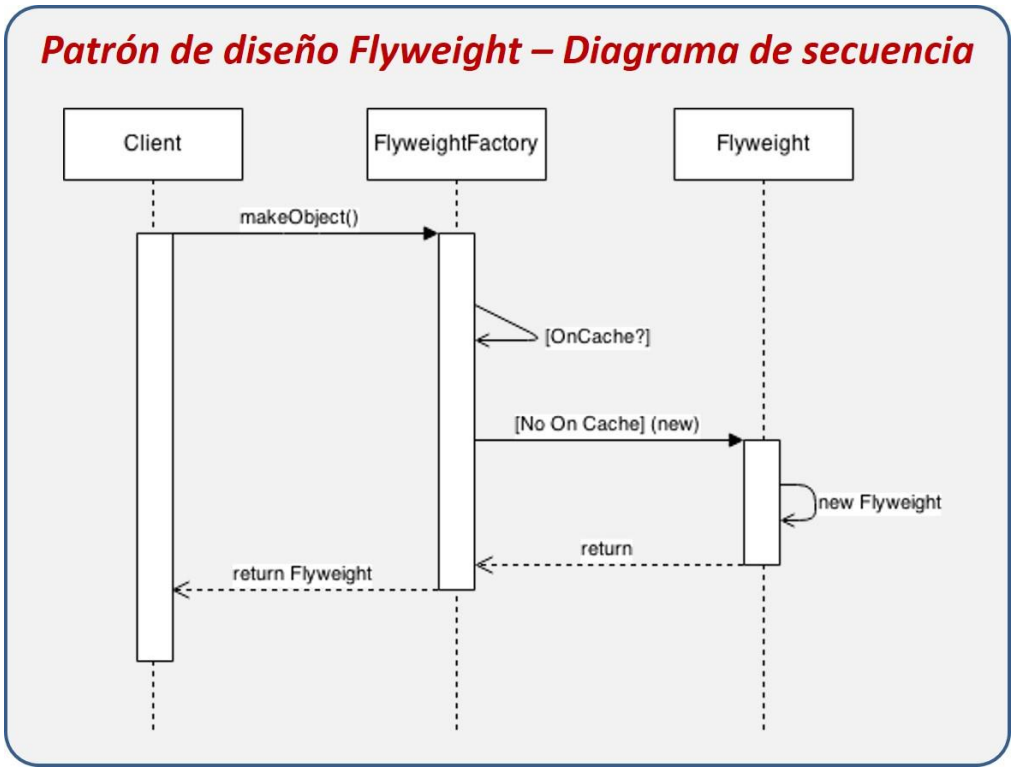


Ilustración 90: Diagrama de secuencia del patrón Flyweight.

El diagrama anterior se interpreta de la siguiente manera:

1. El cliente solicita al *Factory* la creación de un objeto *Flyweight*.
2. El *Factory* antes de crear el objeto, valida si ya existe un objeto idéntico al que se le está solicitando. De ser así, regresa el objeto existente; de no existir, crea el nuevo objeto y lo almacena en caché para ser reutilizado más adelante.
3. El objeto *Flyweight* se crea o es tomado del caché y es devuelto al cliente.

Flyweight es un patrón que utiliza caché para almacenar todas las instancias creadas con el fin de ser reutilizadas cuando un cliente solicita un objeto con características idénticas a alguno ya existente.

Cuándo utilizarlo:

- Cuando identificamos que varios de los objetos que creamos contienen la misma información y vemos que podemos tener objetos compartidos.
- Cuando es necesario optimizar el uso de la memoria que utilizan nuestros objetos.

El escenario:

Para comprender mejor cómo es que trabaja este patrón, desarrollaremos un ejemplo muy simple.

Imaginemos que estamos desarrollando una aplicación de reproducción de música, en la cual los usuarios pueden crear sus propias listas de reproducción. Un usuario puede crear cuántas listas desee y en cada lista puede agregar cualquier número de canciones. De la misma manera un usuario puede agregar la misma canción en más de una lista de reproducción.

Uno de los principales problemas que tenemos aquí, es que la misma canción puede estar en muchas listas de reproducción, no sólo en las listas del mismo usuario sino que una canción puede estar en muchas listas de reproducción de muchos de nuestros usuarios.

Un problema típico son los *hits* del momento, ya que más del 80% de nuestros usuarios tendrán una lista de reproducción con uno o más de éstos y tendremos copias de objetos idénticas en cada lista de reproducción.

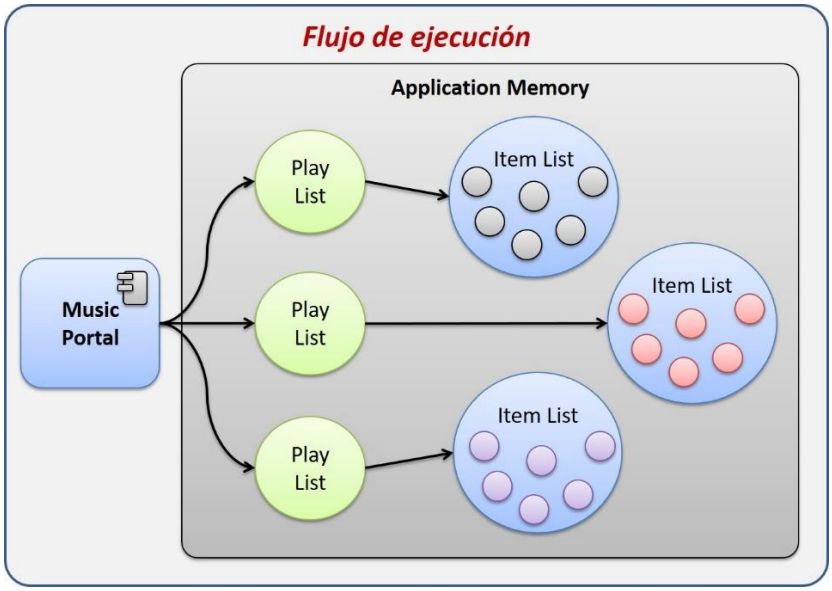


Ilustración 91: Listas de reproducción sin utilizar Flyweight.

Como vemos en la imagen, tenemos tres listas de reproducción; probablemente las canciones contenidas sean exactamente iguales, sin embargo, para el servidor cada ítem representa un objeto independiente que está consumiendo memoria, por lo que es importante implementar alguna solución que nos ayude a reducir la memoria que estos objetos están utilizando.

La solución:

Para darle una solución al problema de las lista de reproducción, podemos utilizar el patrón de diseño *Flyweight*, que nos ayudará a reutilizar los objetos de las listas.

Hace un momento hablamos de dividir los objetos en dos partes (Intrínseco y Extrínseco). Nuestra tarea consistirá en saber qué partes de la lista de reproducción pueden ser reutilizadas y cuáles no.
En nuestro ejemplo tendremos básicamente dos objetos: el primero representará la lista de reproducción que tendrá un ID y un nombre. Estas dos propiedades son diferentes en cada lista y las clasificaremos como la parte extrínseca.
Por otra parte, tenemos el objeto que representa los Items de la lista, éste tiene un ID, nombre de la canción y un arreglo de bits que corresponde al binario de la canción.

Estas propiedades serán idénticas para todos los Items que correspondan a la misma canción, por lo que podemos reutilizar esta parte, de tal forma que cuando un usuario intente agregar una canción que ya existe en caché, ésta en vez de ser creada, se tomará del caché y se reutilizará

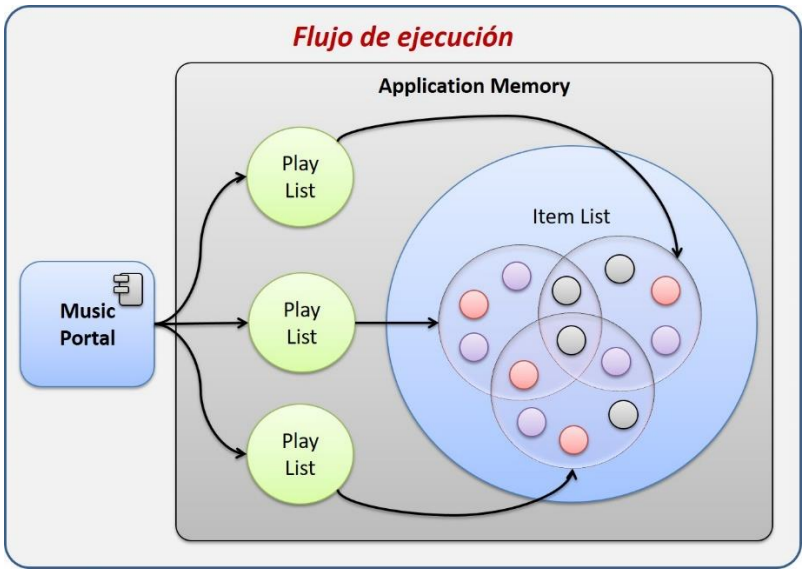


Ilustración 92: Lista de reproducción utilizando el patrón Flyweight.

En la imagen podemos ver que las listas de reproducción son objetos independientes, sin embargo, los Items de las listas hacen referencia a un *pool* de objetos compartidos entre las demás listas de reproducción; de esta manera se ahorra una gran cantidad de recursos del servidor.

La implementación:

Ya hemos visto el problema a resolver y sabemos cómo solucionarlo, lo único que nos queda es implementarlo.

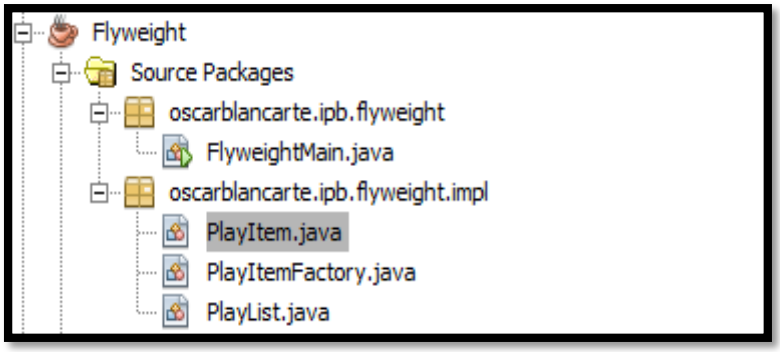


Ilustración 93: Estructura del proyecto Flyweight.

Los paquetes del proyecto *Flyweight* son los siguientes:

- **Oscarblancarte.ipd.flyweight:** Paquete por *default* que tiene la clase principal del proyecto, clase *FlyweightMain*.
- **Oscarblancarte.ipd.flyweight.impl:** Paquete en el que ponemos toda la implementación del patrón, en ella tenemos los objetos de la lista de reproducción y el *Factory* necesario para crear los objetos.

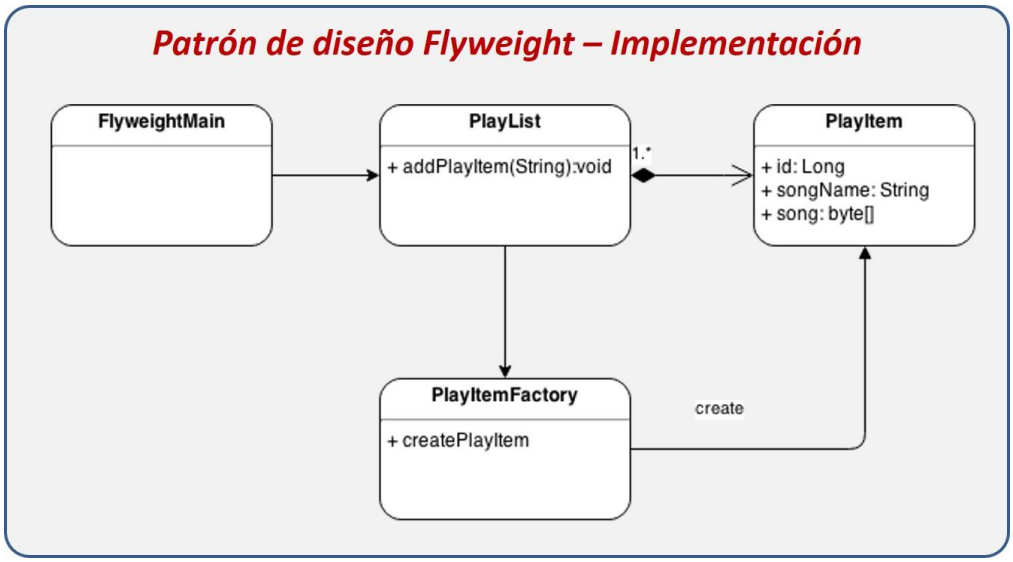


Ilustración 94: Diagrama de clases del proyecto Flyweight.

En la imagen podemos apreciar cómo están relacionadas las clases del proyecto para dar la solución propuesta.

Clase PlayItem:

Clase que representa la parte intrínseca y que podremos reutilizar. Esta clase corresponde a una canción dentro de la lista de reproducción, la cual tiene un ID para identificarla, el nombre de la canción y un arreglo de *bytes* con el fin de simular que tiene el binario de la canción para darle un mayor peso al objeto en memoria y poder comprobar las ventajas del patrón.

```
1.  package oscarblancarte.ipb.flyweight.impl;
2.
3.  public class PlayItem {
4.
5.      private Long id;
6.      private String songName;
7.      private byte[] song = new byte[1000000];
8.
9.      public PlayItem(Long id, String songName) {
10.             this.id = id;
11.             this.songName = songName;
12.         }
```

```
13.
14.         public PlayItem() {
15.         }
16.
17.         @Override
18.         public String toString() {
19.             return "PlayItem{" + "id=" + id + ", songName=" + songName + '}';
20.         }
21.         /** GET and SET */
22.     }
```

Clase PlayItemFactory:

Esta clase es una fábrica de *PlayItem*, la cual antes de crear un objeto nuevo, valida que no exista ya en caché. Para validar si la canción existe hace una búsqueda por nombre sobre el Map *PLAY_ITEMS*.

```
1.  package oscarblancarte.ipb.flyweight.impl;
2.
3.  import java.util.HashMap;
4.  import java.util.Map;
5.
6.  public class PlayItemFactory {
7.      public static boolean enableFlyweight = true;
8.      private static final Map<String,PlayItem> PLAY_ITEMS = new HashMap<>();
9.      private static Long idSequence = 0L;
10.     public static PlayItem createPlayItem(String songName){
11.         if(enableFlyweight && PLAY_ITEMS.containsKey(songName)){
12.             return PLAY_ITEMS.get(songName);
13.         }else{
14.             PlayItem playItem = new PlayItem(++idSequence, songName);
15.             PLAY_ITEMS.put(songName, playItem);
16.             return playItem;
17.         }
18.     }
19. }
```

La clase define el método *createPlayItem* que es el encargado de buscar el objeto en memoria y crearlo en caso de no existir.

En la línea 11 busca el objeto en el *Map* y en la 12 lo regresa en caso de existir.

De la línea 14 a la 16 crea el nuevo objeto, lo agrega al *Map* y lo regresa.

Clase PlayList:

Esta clase representa la lista de reproducción y es la parte que no podemos reutilizar ya que el nombre de la lista puede cambiar de objeto a objeto y es el usuario quien se encarga de ponerle el nombre que desee. Esta clase corresponde a la parte extrínseca.

```
1.  package oscarblancarte.ipb.flyweight.impl;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class PlayList {
7.
8.      private String playListName;
9.      private List<PlayItem> playItems = new ArrayList<>();
10.
11.         public PlayList(String playListName) {
12.             this.playListName = playListName;
13.         }
14.
15.         public void addPlayItem(String songName) {
16.             playItems.add(PlayItemFactory.createPlayItem(songName));
17.         }
18.
19.         public void printList() {
20.             String out = "\nPlayList > " + playListName;
21.             for (PlayItem playItem : playItems) {
22.                 out += "\n\t" + playItem.toString();
23.             }
24.             System.out.println(out);
25.         }
26.         /** GET and SET */
27.     }
```

Podemos apreciar que tiene una lista de objetos *PlayItem*, los cuales son la parte reutilizable.

Clase FlyweightMain:

Corresponde a la clase principal del proyecto desde la cual podremos validar el ejemplo.

En la clase definimos dos arreglos: *playListNames* son un conjunto de nombres para las listas de reproducción y *songNames* es una lista de nombres de canciones.
Ambos arreglos se llenarán dinámicamente simulando ser las listas de reproducción de los usuarios y el arreglo *songName* será el total de canciones que se podrán utilizar para crear las listas de reproducción.

```
1.  package oscarblancarte.ipb.flyweight;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import java.util.Random;
6.  import oscarblancarte.ipb.flyweight.impl.PlayItemFactory;
7.  import oscarblancarte.ipb.flyweight.impl.PlayList;
8.
9.  public class FlyweightMain {
10.
11.         private static final String[] songNames = new String[1000];
12.         private static final String[] playListNames = new String[174];
13.         private static final List<PlayList> playLists = new ArrayList<>();
14.
15.         public static void main(String[] args) {
16.             System.out.println(""
17.                     + "Proceso de creación de playlist iniciado, este procecso puede\n"
18.                     + "ser muy tardado por la gran cantidad de objetos que se \n"
19.                     + "crearán, por favor espere un momento hasta que se le \n"
20.                     + "notifique que el proceso ha terminado.");
21.
22.             Runtime runtime = Runtime.getRuntime();
23.             System.out.println("MaxMemory > " + (runtime.maxMemory()/1000000));
24.             PlayItemFactory.enableFlyweight = false;
25.             initArrays();
26.             createRandonPlayList();
27.
28.             System.out.println("total playlist > " + playLists.size());
29.
30.             long memoryUsed = runtime.totalMemory() - runtime.freeMemory();
31.             System.out.println ("Memory Used => " + (memoryUsed / 1000000));//
32.         }
33.
34.         private static void createRandonPlayList() {
```

```
35.         Random random = new Random();
36.         int p = 0;
37.         for (int c = 0; c < playListNames.length; c++) {
38.             PlayList playList = new PlayList(playListNames[c]);
39.             for (int i = 0; i < 10; i++) {
40.                 int song = random.nextInt(songNames.length);
41.                 playList.addPlayItem(songNames[song]);
42.             }
43.             playLists.add(playList);
44.             //System.out.println("Completado "+c+" unidades " + (c%playListNames.length));
45.             if(c!=0 && (c+1)%(playListNames.length/10)==0){
46.                 p+=10;
47.                 System.out.println("Completado "+ p +"%");
48.                 System.out.println("Listas creadas " + playLists.size());
49.             }
50.         }
51.     }
52.
53.     private static void initArrays() {
54.         for (int c = 0; c < songNames.length; c++) {
55.             songNames[c] = "Song " + (c + 1);
56.         }
57.
58.         for (int c = 0; c < playListNames.length; c++) {
59.             playListNames[c] = "PlayList " + (c + 1);
60.         }
61.     }
62. }
```

Los métodos definidos son:

- **initArrays:** Método que crea dinámicamente los nombres de las listas de reproducción y los nombres de las canciones.
- **createRandomPlayList**: Método que crea de forma aleatoria una combinación de diez canciones por cada lista de reproducción definida en el arreglo *playListNames*.

En la línea 22 y 23 imprimimos la cantidad de memoria disponible para nuestra aplicación.

La línea 24 es muy importante, ya que en ella habilitamos o deshabilitamos el patrón *Flyweight*. Más adelante en la sección de ejecución veremos su uso.

En las líneas 24 y 25 iniciamos el nombre de las listas y canciones y creamos 1,000 listas de reproducción aleatorias con 10 canciones cada una.

En la línea 31 imprimimos la memoria disponible tras finalizar el programa.

La ejecución:

Para demostrar el poder de utilizar este patrón, ejecutaremos la clase *FlyweightMain* con dos configuraciones diferentes: la primera la ejecutaremos utilizando el patrón *Flyweight* y la otra sin él.

Iniciaremos ejecutando el programa sin utilizar el patrón. Para ello, editaremos la clase *FlyweightMain* y modificaremos la siguiente línea:

```
16.     private static final String[] playListNames = new String[174];
```

Modificamos la línea 16 y establecemos el tamaño del arreglo *playListNames* en 174, lo que significa que crearemos este número de listas de reproducción.

Modificamos la línea 28 para que quede de la siguiente manera:

```
28.     PlayItemFactory.enableFlyweight = false;
```

Esta propiedad en *false* servirá para deshabilitar el patrón *Flyweight* lo que ocasionará que todos los *PlayItem* sean objetos distintos.

Finalmente ejecutamos el programa:

```
run:
Proceso de creación de playlist iniciado,este proceso puede
ser muy tardado por la gran cantidad de objetos que se
crearan, por favor espere un momento hasta que se le
notifique que el proceso ha terminado.
MaxMemory > 1886
Compleado 10%
Listas creadas 17
Compleado 20%
Listas creadas 34
Compleado 30%
Listas creadas 51
Compleado 40%
Listas creadas 68
Compleado 50%
Listas creadas 85
Compleado 60%
Listas creadas 102
Compleado 70%
Listas creadas 119
Compleado 80%
Listas creadas 136
Compleado 90%
Listas creadas 153
Compleado 100%
Listas creadas 170
total playlist > 174
Memory Used => 1740
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 95: Resultado de la primera ejecución.

Como podemos ver, hemos creado 174 listas de reproducción, la memoria máxima de nuestro programa es de 1886mb, y en la última línea nos dice que la memoria utilizada fue de 1740mb. Esta ejecución ha dejado casi al límite nuestra aplicación ya que si aumentamos en uno las listas de reproducción, nuestro programa se quedará sin memoria y finalizará inesperadamente.

Cambiemos nuevamente la línea 16 como se ve a continuación:

```
16.    private static final String[] playListNames = new String[176];
```

Procedemos a ejecutar la aplicación nuevamente y el resultado es el siguiente:

```
run:
Proceso de creación de playlist iniciado,este proceso puede
ser muy tardado por la gran cantidad de objetos que se
crearan, por favor espere un momento hasta que se le
notifique que el proceso ha terminado.
MaxMemory > 1886
Compleado 10%
Listas creadas 17
Compleado 20%
Listas creadas 34
Compleado 30%
Listas creadas 51
Compleado 40%
Listas creadas 68
Compleado 50%
Listas creadas 85
Compleado 60%
Listas creadas 102
Compleado 70%
Listas creadas 119
Compleado 80%
Listas creadas 136
Compleado 90%
Listas creadas 153
Compleado 100%
Listas creadas 170
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at oscarblancarte.ipb.flyweight.impl.PlayItem.<init>(PlayItem.java:11)
        at oscarblancarte.ipb.flyweight.impl.PlayItemFactory.createPlayItem(PlayItemFactory.java:18)
        at oscarblancarte.ipb.flyweight.impl.PlayList.addPlayItem(PlayList.java:36)
        at oscarblancarte.ipb.flyweight.FlyweightMain.createRandonPlayList(FlyweightMain.java:45)
        at oscarblancarte.ipb.flyweight.FlyweightMain.main(FlyweightMain.java:30)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 96: Resultado fallido por falta de memoria.

Como vemos, el programa llegó al límite con tan sólo 176 listas de reproducción debido a que se le terminó la memoria.

Para la siguiente prueba, modificaremos la línea 16 y estableceremos el tamaño del arreglo en 4,100,000 como se observa a continuación:

```
16.    private static final String[] playListNames = new String[4100000];
```

Y modificaremos la línea 28 para habilitar el patrón *Flyweight* como se observa a continuación:

```
28.    PlayItemFactory.enableFlyweight = true;
```

Ejecutamos el programa y la salida es la siguiente:

```
run:
Proceso de creación de playlist iniciado,este proceso puede
ser muy tardado por la gran cantidad de objetos que se
crearan, por favor espere un momento hasta que se le
notifique que el proceso ha terminado.
MaxMemory > 1886
Compleado 10%
Listas creadas 410000
Compleado 20%
Listas creadas 820000
Compleado 30%
Listas creadas 1230000
Compleado 40%
Listas creadas 1640000
Compleado 50%
Listas creadas 2050000
Compleado 60%
Listas creadas 2460000
Compleado 70%
Listas creadas 2870000
Compleado 80%
Listas creadas 3280000
Compleado 90%
Listas creadas 3690000
Compleado 100%
Listas creadas 4100000
total playlist > 4100000
Memory Used => 1766
BUILD SUCCESSFUL (total time: 19 seconds)
```

Ilustración 97: Resultado de la ejecución con el patrón Flyweight.

Hemos creado 4,100,000 listas de reproducción exitosamente. Podemos notar que la diferencia es abismal, estamos hablando de que hemos creado 175 vs 4,100,000. Si aplicamos un poco de matemáticas 4,100,000/175 = 23428.57, esto nos dice que hemos creado 23,438 veces más listas de reproducción que con el método tradicional. Esta cifra es realmente inmensa e imaginemos la diferencia que podría hacer en una aplicación real que tiene más listas de reproducción y muchísimas más canciones.

Siguientes pasos:

Hemos demostrado la eficiencia de este patrón a la hora de optimizar los recursos en una aplicación con relativamente pocas canciones, sin embargo, recordemos que la capacidad final de objetos que este programa puede crear dependerá de la memoria que nuestro sistema operativo le asigne a nuestra máquina virtual y los parámetros de creación de ésta. No obstante, creemos que este ejemplo es lo suficientemente claro para demostrar su utilidad.

- Como siguientes pasos podríamos asociarle a los *PlayItem* un objeto que corresponda al artista, en este caso un artista puede tener muchas canciones, por lo que el objeto artista podría ser totalmente reutilizable.

- Con la finalidad de optimizar aún más la aplicación podríamos crear un sistema de almacenamiento (base de datos, archivo plano, XML, Json, etc.) que permite almacenar los playlist que menos se utilizan, y mantener en memoria únicamente los más utilizados o hasta donde la memoria del programa nos permita.

Patrón Proxy

Este es un patrón de diseño que centra su atención en la mediación entre un objeto y otro. Se dice mediación porque este patrón nos permite realizar ciertas acciones antes y después de realizar la acción deseada por el usuario. El Proxy se caracteriza por que el cliente ignora totalmente que una mediación se está llevando acabo debido a que el cliente recibe un objeto idéntico en estructura al esperado, y no es consciente de la implementación tras la interface ejecutada, de esta manera el cliente interactúa con el Proxy sin saberlo.

Otro de los usos más comunes de los *proxys* es para representar de forma local objetos remotos que se encuentran en otro servidor o aplicación. El *proxy* actúa como una máscara para encapsular la lógica y ejecutar al objeto remoto.

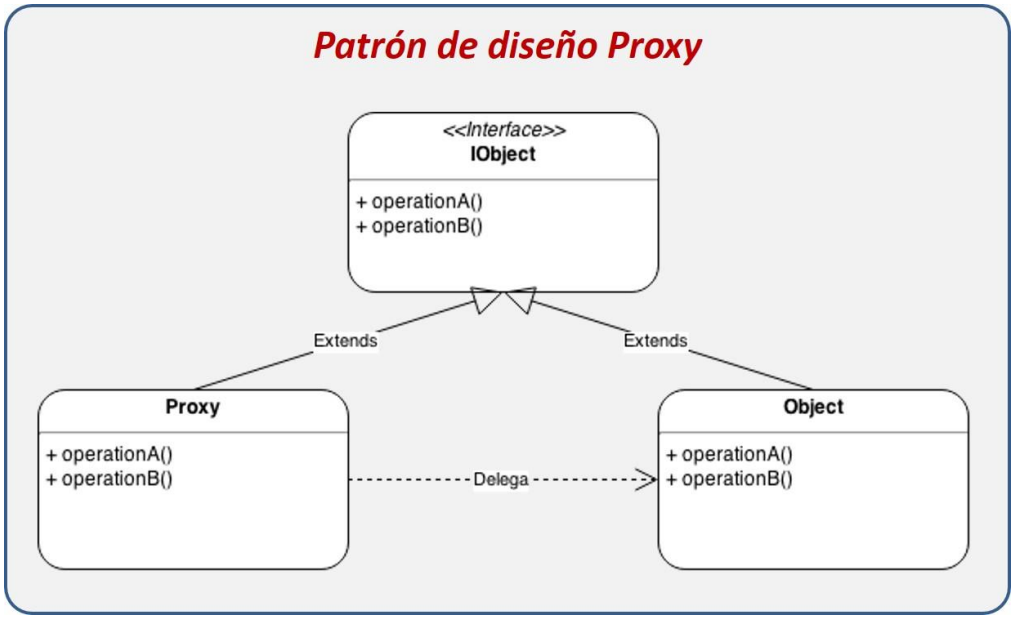


Ilustración 98: Estructura del patrón Proxy.

Los componentes que conforman el patrón son los siguientes:

- **IObject:** Representa la interface común entre el **Objet** y el **Proxy**.
- **Object:** Representa el objeto al que el cliente quiere ejecutar.

- **Proxy:** Clase que implementa **IObject** y delega la responsabilidad al **Object**, sin embargo, puede realizar una acción antes y después de llamar al **Object**.

Un detalle importante del *Proxy* es que puede que incluso el **Object** no sea ejecutado nunca debido a que las acciones posteriores no fueron satisfactorias, pero el cliente nunca sabrá esto.

Para asegurarnos de que el cliente utilice el Proxy en lugar del **Object,** será necesario implementar un mecanismo que nos permite controlar la creación de los objetos, por lo cual, utilizar patrones creacionales sería una excelente estrategia.

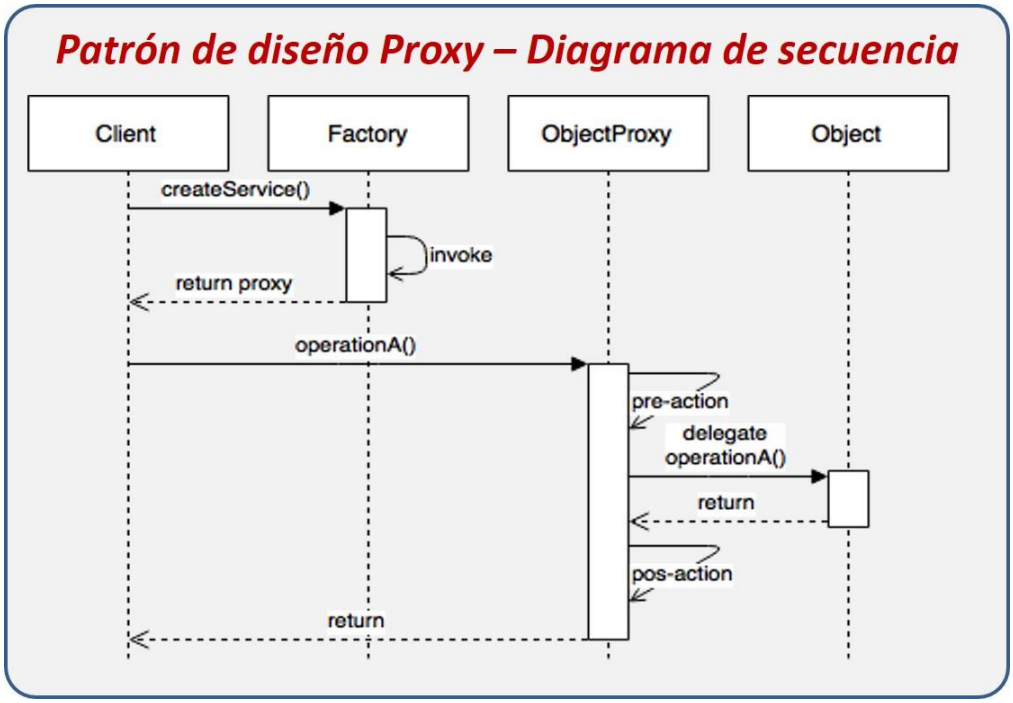


Ilustración 99: Diagrama de secuencia del patrón Proxy.

El diagrama anterior se interpreta de la siguiente manera:

1. El cliente solicita al *Factory* un Objeto.

2. El *Factory* crea un *Proxy* que encapsule al *Object*.
3. El cliente ejecuta el *Proxy* creado por el *Factory*.
4. El *Proxy* realiza una o varias acciones previas a la ejecución del *Object*.
5. El *Proxy* delega la ejecución al *Object*.
6. El *Proxy* realiza una o varias acciones después de la ejecución del *Object*.
7. El *Proxy* regresa un resultado.

El resultado devuelto por el *Proxy* no es precisamente el devuelto por el *Object*, ya que el *Proxy* puede tomar ciertas acciones que modifiquen el resultado o incluso creen uno totalmente nuevo antes de responder al cliente.

Cuándo utilizarlo:

- Cuando lo que buscamos es controlar la forma en que se ejecuta un objeto sin afectar al consumidor, agregando acciones previas y posteriores a la ejecución de la operación solicitada.

- Cuando queremos representar objetos remotos de forma local, creando un proxy que permite establecer la comunicación con el objeto remoto.

- Virtualizar un objeto real, el cual represente a un objeto de gran tamaño o que su creación sea costosa. Este proxy carga al objeto real justo en el momento en que es requerido y no desde un inicio.

El escenario:

Nos han pedido que desarrollemos un portal para ejecutar procesos, estos procesos solamente pueden ser ejecutados por personas autorizadas y deben ser auditadas, de tal forma que si un usuario ejecuta un proceso este deberá quedar registrado para su posterior análisis.

El cliente nos ha pedido que se puedan realizar algunas acciones previas a la ejecución y otras acciones posteriores, pero aún no está seguro, por lo que es importante dejar el sistema preparado para agregar estas nuevas acciones sin realizar un gran impacto en las clases existentes.

Para dar solución a este problema nos han proporcionado los servicios:

- **SecurityService:** Servicio que valida si el usuario tiene privilegios para ejecutar el proceso.
- **AuditService:** Servicio para registrar cuando un usuario ejecuta un proceso.

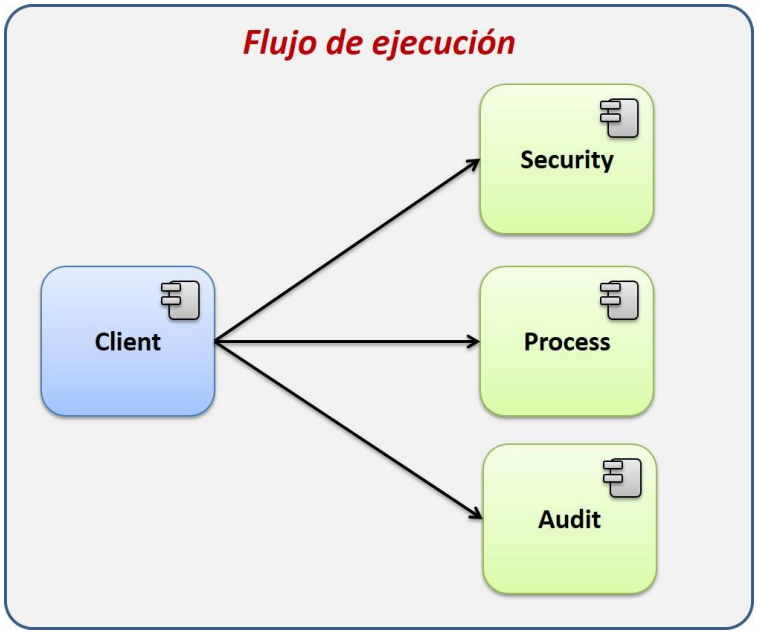


Ilustración 100: Ejecución del proceso de forma tradicional.

En la imagen vemos que el cliente es el encargado de validar si tiene privilegios, luego ejecutar el proceso y finalmente hacer la auditoria. Este escenario plantea dos problemas: el primero es que si agregamos una acción previa o posterior a la ejecución del proceso tendremos que modificar al *Client* para que realice esta acción, la segunda es que el cliente se está encargando de hacer los pasos previos y posteriores, siendo que está no es responsabilidad del cliente, ya que a él lo único que le interesa es ejecutar los procesos.

La solución:

Ahora desarrollaremos una solución basada en *Proxys*, los cuales se encargarán de realizar las acciones previas y posteriores a la ejecución del proceso.

Para esto deberemos crear una nueva clase que encapsule a los procesos, también obligaremos al cliente a obtener la instancia del proceso por medio de un *Factory*. El *Factory* nos aseguraría que el Cliente siempre reciba un *Proxy* en vez de *Object* de ejecución de procesos, de esta forma también nos aseguramos de que si el *Proxy* agrega nuevas funcionalidades pre y post, el cliente no se entera y así no afectar los componentes ya desarrollados.

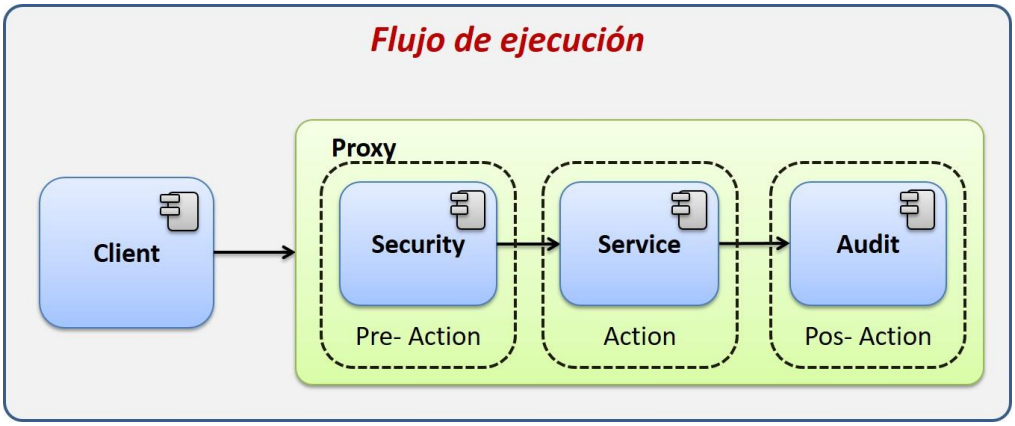


Ilustración 101: Implementación de la ejecución de procesos utilizando Proxy.

En la imagen vemos como el cliente en vez de ejecutar el servicio de ejecución de procesos (***Action***) ejecuta al *Proxy*, el cual tiene como pre-acción la ejecución del servicio de seguridad, con el fin de validar si tiene privilegios de ejecución. Si el servicio de seguridad falla, el *Proxy* terminará sin haber llamado al ***Service***. El *Proxy* también tiene una pos-acción la cual registra en el servicio de auditoria la ejecución del proceso por parte del usuario.

Una cosa importante que podemos hacer con el proxy y que no tenemos contemplado, es que tenga alguna política de fallo en la cual reintente la ejecución en caso de fallo o que mande alguna notificación al departamento de IT con la causa del problema (todo esto sin que el cliente lo sepa).

La implementación:

Una vez que entendemos el escenario y sabemos cómo solucionar el problema iniciaremos con la implementación.
La siguiente imagen muestra los paquetes que conforman el proyecto:

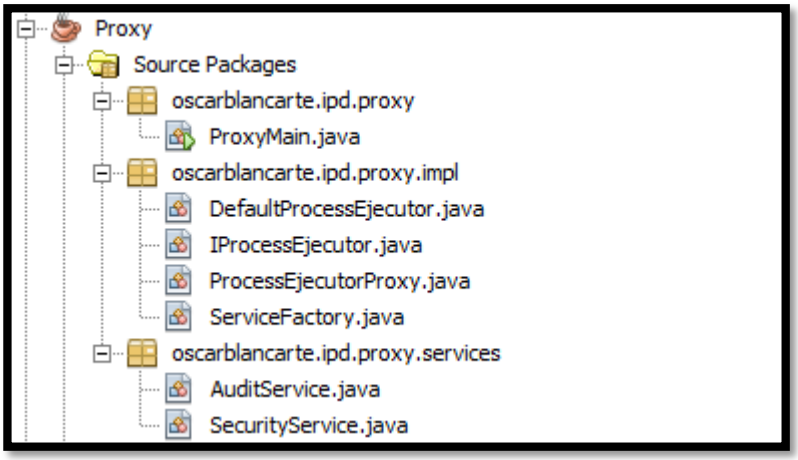


Ilustración 102: Estructura del proyecto Proxy.

Los paquetes son los siguientes:

- **oscarblancarte.ipd.proxy:** Paquete por *default* el cual únicamente tiene la clase principal ***ProxyMain***.
- **oscarblancarte.ipd.proxy.impl:** Paquete en el que están las clases para crear los *proxys*.
- **oscarblancarte.ipd.proxy.services:** Paquete donde se encuentran los servicios externos que nos ayudarán a auditar y autorizar la ejecución de los procesos.

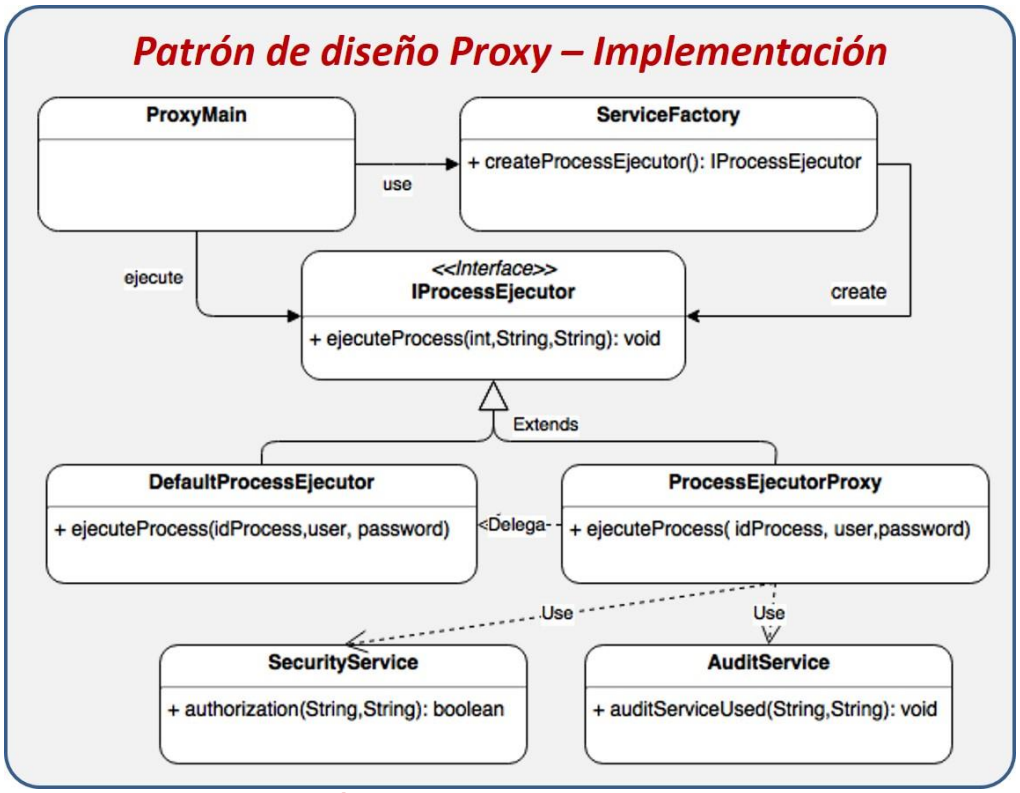


Ilustración 103: Estructura del proyecto Proxy.

En la imagen anterior podemos ver las clases que componen el proyecto Proxy así como la forma en que están estas relacionadas entre sí.

Clase AuditService:

Clase que intenta simular el funcionamiento de un servicio de Auditoria, el cual recibe el usuario y el proceso que ejecutó. El servicio únicamente imprime en pantalla el registro de la ejecución junto con la fecha y hora.

```
1. package oscarblancarte.ipd.proxy.services;
2.
3. import java.text.SimpleDateFormat;
4. import java.util.Date;
5.
6. public class AuditService {
7.     public void auditServiceUsed(String user, String service){
8.         SimpleDateFormat formater= new SimpleDateFormat("dd/MM/YYYY hh:mm:ss");
```

```
9.        System.out.println(user + " utilizo el servicio > "
10.                    + service + ", a las " +formater.format(new Date()));
11.        }
12.    }
```

Clase SecurityService:

Clase que intenta simular un servicio de seguridad y proporciona un método de autorización donde el único usuario válido es oblancarte con el *password* 1234, el servicio regresa *true* en caso de que la autorización se lleve a cabo con éxito y *false* en caso contrario.

```
1.  package oscarblancarte.ipd.proxy.services;
2.
3.  public class SecurityService {
4.      public boolean authorization(String user,String password){
5.          if(user.equals("oblancarte") && password.equals("1234")){
6.              System.out.println("Usuario " + user + " autorizado");
7.              return true;
8.          }else{
9.              System.out.println("Usuario " + user + " NO autorizado");
10.                 return false;
11.             }
12.         }
13.     }
```

Interface IProcessEjecutor:

Interface que define la estructura de los ejecutores de procesos así como de los *Proxys*.

```
1.  package oscarblancarte.ipd.proxy.impl;
2.
3.  public interface IProcessEjecutor {
4.      public void ejecuteProcess(int idProcess, String user, String password)
5.              throws Exception;
6.  }
```

Se define el método *ejecuteProcess*, el cual es el encargado de ejecutar los procesos mediante el *idProces*. El usuario y *password* se utilizan para la autorización.

Class DefaultProcessEjecutor:

Esta es la clase que el usuario requiere para ejecutar los procesos, sin embargo, controlaremos su creación por medio del *Factory* para asegurarnos de regresarle un *proxy* en lugar de una instancia de esta clase. La clase hereda de *IProcessEjecutor*.

```
1.  package oscarblancarte.ipd.proxy.impl;
2.
3.  public class DefaultProcessEjecutor implements IProcessEjecutor {
4.
5.      @Override
6.      public void ejecuteProcess(int idProcess,String user,String password)
7.          throws Exception {
8.
9.          System.out.println("procesos " + idProcess + " en ejecución");
10.             System.out.println("procesos " + idProcess + " terminado");
11.         }
12.     }
```

La clase implementa el método *ejecuteProcess* el cual simula realizar una tarea.

Class ProcessEjecutorProxy:

Clase utilizada como *Proxy*, esta clase hereda de *IProcessEjecutor* al igual que *DefaultProcessEjecutor*, sin embargo, ésta no realiza realmente la tarea, en su lugar, valida los privilegios del usuario y luego delega la ejecución del proceso a *DefaultProcessEjecutor* y finalmente audita la ejecución.

```
1.  package oscarblancarte.ipd.proxy.impl;
2.
3.  import oscarblancarte.ipd.proxy.services.AuditService;
4.  import oscarblancarte.ipd.proxy.services.SecurityService;
5.
6.  public class ProcessEjecutorProxy implements IProcessEjecutor {
```

```
7.
8.      @Override
9.      public void ejecuteProcess(int idProcess,String user,String password)
10.                 throws Exception {
11.             SecurityService securityService = new SecurityService();
12.             if(!securityService.authorization(user, password)){
13.                 throw new Exception("El usuario "'+user
14.                         +"' no tiene privilegios para ejecutar el proceso");
15.             }
16.
17.             DefaultProcessEjecutor ejecutorProcess = new DefaultProcessEjecutor();
18.             ejecutorProcess.ejecuteProcess(idProcess, user, password);
19.
20.             AuditService audit = new AuditService();
21.             audit.auditServiceUsed(user, DefaultProcessEjecutor.class.getName());
22.         }
23.     }
```

(***Pre-Action***) En la línea 11 a la 15 se realiza la validación de los privilegios del usuario, si el usuario no existe o no tiene los privilegios suficientes el proceso terminará sin haber llegado a la acción real.

(***Action***) En las líneas 17 y 18 se realiza la ejecución del proceso por medio de la clase *DefaultProcessEjecutor*.

(***Pos-Action***) De las líneas 20 a la 22 se realiza la auditoria del proceso.

Class ServiceFactory:

Clase utilizada como *Factory*, la cual en vez de regresar la clase *DefaltProcessEjector* regresa una instancia de la clase *ProcessEjecutorProxy* que en realidad es un *proxy.*

```
1.  package oscarblancarte.ipd.proxy.impl;
2.
3.  public class ServiceFactory {
4.
5.      public static IProcessEjecutor createProcessEjecutor(){
6.          return new ProcessEjecutorProxy();
7.      }
8.  }
```

Class ProcessEjecutorProxy:

Clase principal del proyecto que será utilizada para probar nuestra aplicación.

```
1.  package oscarblancarte.ipd.proxy;
2.
3.  import oscarblancarte.ipd.proxy.impl.IProcessEjecutor;
4.  import oscarblancarte.ipd.proxy.impl.ServiceFactory;
5.
6.  public class ProxyMain {
7.
8.      public static void main(String[] args) {
9.          String user = "oblancarte";
10.             String password = "1234";
11.             int process = 1;
12.             IProcessEjecutor processEjecutor = ServiceFactory.createProcessEjecutor();
13.             try {
14.                 processEjecutor.ejecuteProcess(process, user, password);
15.             } catch (Exception e) {
16.                 System.out.println(e.getMessage());
17.             }
18.         }
19.     }
```

Los pasos de la ejecución son los siguientes:

- De las líneas 9 a la 11 se configuran los parámetros para ejecutar el proceso.
- En la línea 12 se obtiene una instancia del *Proxy* por medio del *Factory*.
- En la línea 14 se ejecuta el proceso.

La ejecución:

Iniciaremos ejecutando la aplicación mediante la clase *ProxyMain* para lo cual nos aseguraremos de tener el usuario, *password* y processId de la siguiente manera:

```
1. String user = "oblancarte";
2. String password = "1234";
3. int process = 1;
```

Y tendremos el siguiente resultado:

```
run:
Usuario oblancarte autorizado
procesos 1 en ejecución
procesos 1 terminado
oblancarte utilizo el servicio > oscarblancarte.ipd.proxy.impl.DefaultProcessEjecutor, a las 11/03/2015 06:03:04
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 104: Resultado exitoso de la ejecución.

Lo primero que vemos es que el usuario *oblancarte* fue autorizado para ejecutar el proceso.

Luego vemos cómo el proceso inicia y termina.

Finalmente vemos el mensaje de auditoria donde indica que el usuario *oblancarte* ejecutó el proceso de la clase *DefaultProcessEjecutor* y la fecha.

Para el siguiente ejemplo cambiaremos los parámetros de la siguiente manera:

```
1. String user = "oblancarte";
2. String password = "2345";
3. int process = 1;
```

Con esta configuración habremos cambiado el password del usuario y ejecutamos nuevamente el programa.

```
run:
Usuario oblancarte NO autorizado
El usuario 'oblancarte' no tiene privilegios para ejecutar el proceso
BUILD SUCCESSFUL (total time: 1 second)
```

Ilustración 105: Ejecución fallida del proceso.

En esta nueva ejecución vemos que el usuario *oblancarte* no tiene privilegios debido a que el *password* es incorrecto.

Como vemos, el *Proxy* fue quien realizó la validación del usuario y terminó sin haber si quiera intentado ejecutar el proceso.

Siguientes pasos:

Hemos comprobado la efectividad del patrón de diseño *Proxy* y hemos visto cómo es que este patrón de diseño nos permite colgar funcionalidad extra en la llamada a una operación, agregando pasos antes y después de la ejecución de la operación deseada.

Para reforzar los conocimientos adquiridos en este capítulo, realizaremos los siguientes ejercicios:

- Agreguemos un paso extra a nuestro *proxy*, el cual antes de ejecutar el proceso y después de validar al usuario, cuente el número de veces que se ha ejecutado el proceso; si el proceso se ejecuta más de tres veces en un mismo día, deberá lanzar un error.

- Creemos un nuevo *proxy* que sólo autentifique al usuario pero que no audite la ejecución.

- Modifiquemos el *Factory* para que por medio de un parámetro, le indiquemos qué *proxy* queremos utilizar. El *Factory* te deberá permitir obtener un proxy auditable y otro no auditable, el cual lo hicimos en el ejercicio anterior.

Patrones de Comportamiento

Este tipo de patrones de diseño se caracterizan del resto debido a que se centran en la colaboración, relaciones y delegación de responsabilidades entre otras clases logrando con esto simplificar la forma en que los objetos se comunican e interactúan entre sí; a diferencia de los patrones de diseño creacionales y estructurales donde los creacionales se enfocan en la creación de los objetos y los estructurales en crear estructuras de clases flexibles que ayuden a crear estructuras extensibles y desacopladas. Los patrones de comportamiento se enfocan en la asignación de responsabilidades a los objetos y como es que estos se comunican entre sí.

Este tipo de patrones dan solución a muchos de los problemas con los que nos encontramos diariamente. Es muy común utilizar algunos de los patrones de comportamiento durante el trabajo del día a día y es probable que los identifiquemos rápidamente.

Los patrones que abordaremos en este capítulo son:

Patrón Iterator: Nos permite recorrer estructuras de datos ya sean simples o complejas sin conocer la estructura interna de la misma, mediante la creación de una interface que nos permite obtener el siguiente elemento de la estructura.

Patrón Command: Patrón que nos permite ejecutar operaciones sin saber exactamente qué clase atenderá nuestra petición. Mediante los comandos podemos encapsular la lógica de ejecución, de tal manera que el cliente sólo sepa de la existencia del comando y los parámetros necesarios para su ejecución.

Patrón Observer: Patrón de diseño utilizado para observar los cambios o eventos generados por otros objetos con la finalidad de realizar acciones en consecuencia de algún cambio sobre nuestro objeto observado.

Patrón Templete Method: Patrón de diseño enfocado en la reutilización de código, mediante la creación de templetes, los cuales son clases con

funcionalidad y pasos predefinidos para que las subclases sobrescriban o personalicen algunos de los pasos predefinidos por el templete.

Patrón Strategy: Permite mediante la implementación de estrategias, el intercambio de responsabilidad, las estrategias son clases que definen un comportamiento a la clase que las contienen y mediante este intercambio de estrategias es como se logra alterar el comportamiento de las clases en tiempo de ejecución.

Patrón Chain of Responsability: Permite resolver problemas mediante el encadenamiento de objetos, cada objeto que es encadenado va resolviendo parte del problema, este patrón es utilizado cuando la resolución por medio de la herencia no es posible.

Patrón Interpreter: Patrón diseñado para la interpretación de lenguajes que mediante un conjunto de expresiones que representan dicho lenguaje es capaz de interpretarlo y arrojar un resultado como parte de esta interpretación.

Patrón Mediator: Patrón diseñado para gestionar la forma en que un conjunto de clases se comunican entre sí, sobre todo cuando las clases tienen una comunicación directa y el número de clases es grande. Mediator crea una capa de mediación que permite intervenir entre la comunicación de estas clases.

Patrón Memento: Permite capturar el estado de un objeto en un tiempo determinado, con la finalidad de poder regresar a este estado en cualquier momento.

Patrón Null Object: Patrón que nace de la necesidad de evitar regresar valores nulos, los cuales pudieren ocasionar problemas en tiempo de ejecución, en su lugar el patrón propone le creación de implementaciones básicas, las cuales implementen las operaciones sin cuerpo.

Patrón State: Permite la implementación de aplicaciones que basan su comportamiento dependiendo del estado en el que se encuentre la aplicación.

Una aplicación puede pasar por varios estados y cada estado podrá ir cambiando a medida que se interactúa con la aplicación.

Patrón Visitor: Patrón que permite separar las operaciones de la estructura de datos en cuestión, el visitante permite recorrer una estructura de datos completa mediante un método especial, el cual, permite que el visitante vaya a cada nodo de la estructura, obteniendo con esto un resultado. Visitante se implementa por lo general en estructuras en forma de árbol.

Patrón Iterator

Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma. Es especialmente útil cuando trabajamos con estructuras de datos complejas, ya que nos permite recorrer sus elementos mediante un Iterador, el Iterador es una interface que proporciona los métodos necesarios para recorrer los elementos de la estructura de datos, los métodos más comunes son:

- **hasNext:** Método que regresa un booleano para indicar si existen más elementos en la estructura por recorrer. *True* si existen más y *false* si hemos llegado al final y no hay más elementos por recorrer.
- **next:** Regresa el siguiente elemento de la estructura de datos.

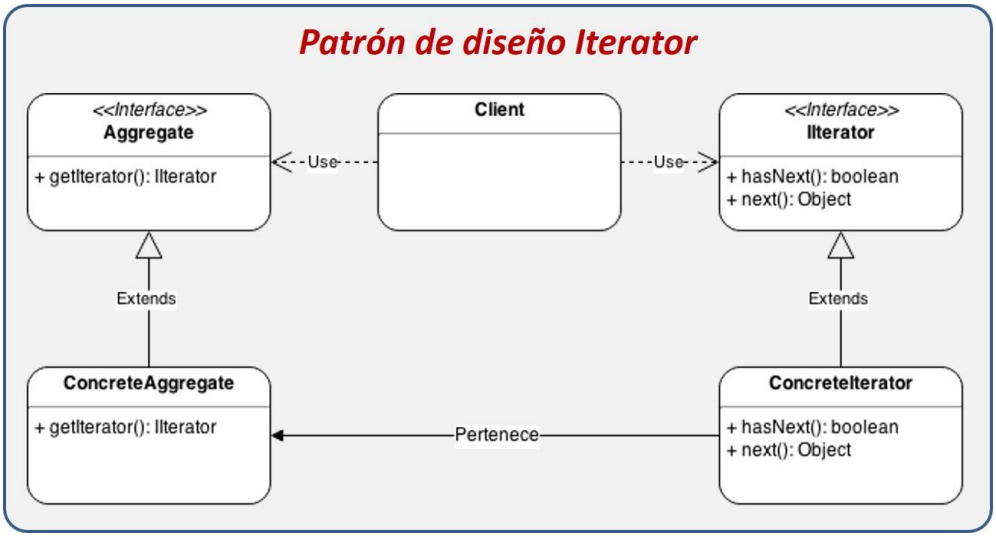


Ilustración 106: Estructura del patrón de diseño Iterator.

Los elementos del patrón Iterator se describen a continuación:

- **Client:** Actor que utiliza al Iterator.
- **Aggregate:** Interface que define la estructura de las clases que pueden ser iteradas.

- **ConcreteAggregate:** Clase que contiene la estructura de datos que deseamos iterar.
- **IIterator:** Interface que define la estructura de los iteradores, la cual define los métodos necesarios para poder realizar la iteración sobre el *ConcreteAggregator*.
- **ConcreteIterator:** Implementación de un iterador concreto, el cual hereda de *IIterator* para implementar de forma concreta cómo iterar un *ConcreteAggregate*.

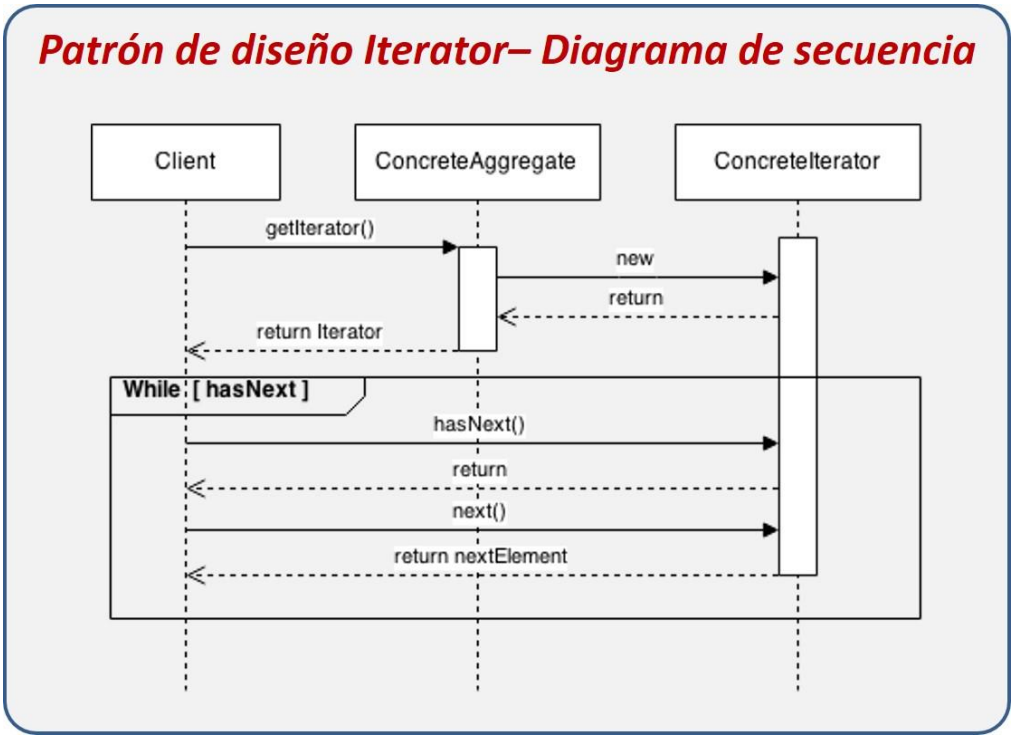


Ilustración 107: Diagrama de secuencia del patrón Iterator.

El diagrama anterior se interpreta de la siguiente manera:

1. El cliente solicita al *ConcreteAggregate* la creación de un iterador.
2. El *ConcreteAggregate* crea un nuevo Iterador.

3. El cliente, para recorrer los elementos, entra en un ciclo hasta que no existen más elementos en el iterador, el método **hasNext** le indicará cuándo se ha llegado al final.
4. El cliente solicita el siguiente elemento al iterador mediante el método *next*.
5. Si existen más elementos nos regresamos al paso tres, esto se repite hasta finalizar el recorrido.

Podemos ver que recorrer la estructura es realmente sencillo y no es necesario saber cómo está implementada, lo único que nos interesa es saber que existen más elementos en la lista y que tenemos una operación para recuperar el siguiente elemento de la estructura.

Cuándo utilizarlo:

- Cuando deseamos proporcionar una forma simple de recorrer los elementos de nuestra estructura.
- Cuando la estructura es compleja y recorrerla puede resultar difícil para quien no está familiarizado con nuestra estructura.

El escenario:

Actualmente estamos desarrollando un programa que recibe una estructura de Empleados, estos empleados tiene a su vez cero o muchos subordinados, el problema que tenemos es que es necesario obtener todos los empleados de la estructura.
Internamente, la estructura de datos es un Árbol en donde la raíz es el CEO y las ramas son los empleados de más bajo rango, el orden con el cual hay que recorrer los empleados no es muy importante pero sí que recorramos todos.

De entrada tenemos que la estructura de datos interna es un Árbol y esto podría hacer que batallemos a la hora de recorrerlo ya que necesitaremos algún procedimiento recursivo que nos asegure el recorrido completo del Árbol. Otro problema podría ser si cambiará la estructura de datos a otro tipo, esto provocaría que nuestro programa deje de funcionar.

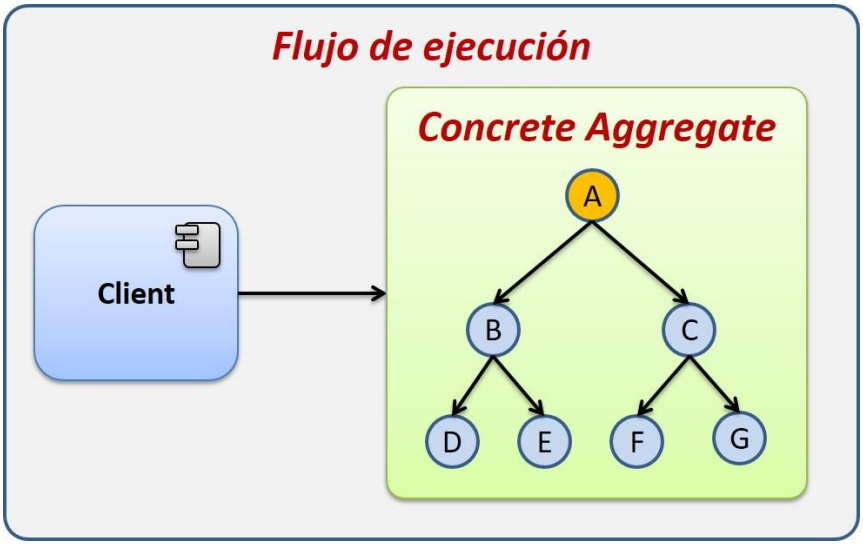


Ilustración 108: Recorrido de la estructura en forma de Árbol.

En la imagen vemos como el cliente desea recorrer manualmente la estructura de datos, logrando con esto que el cliente tenga que conocer la implementación exacta con la que fue diseñada la estructura de datos, si esta estructura cambia, obligaremos al cliente a modificar el programa para recorrer la estructura de forma correcta.

Por esta razón nos han pedido que diseñemos una solución que nos permita recorrer los elementos de la estructura y que además nos protejamos en caso de que la estructura interna cambie mediante el empleo de una interface que encapsule dicha lógica.

La solución:

Debido a la necesidad de implementar un mecanismo que nos permita recorrer toda la estructura sin conocer los detalles de su implementación y que cualquier cambio en la estructura de datos no afecte al cliente, hemos decidido implementar el patrón Iterator con el que resolveremos estos problemas de una forma muy simple.

La idea consiste en hacer que nuestra clase (que contiene la estructura de datos) nos proporcione una operación con la que podremos crear un Iterador, este iterador deberá seguir una interface que defina los métodos mínimos **hasNext** y **next** que nos servirán para validar que existan más elementos en la estructura y el otro para obtener el elemento siguiente respectivamente.

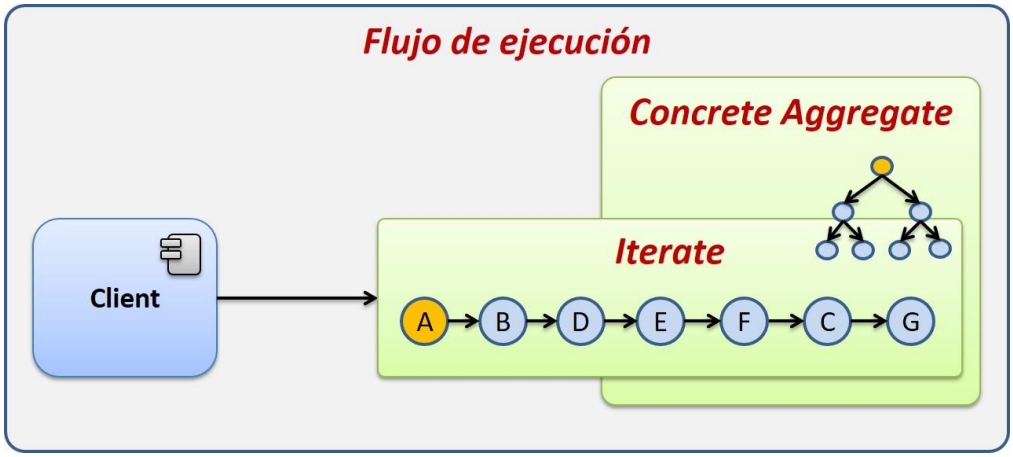


Ilustración 109: Recorrido de la estructura mediante un Iterador.

Vemos como el cliente en vez de recorrer la estructura por sí mismo, mejor le solicita la creación de un Iterator, éste representara el contenido de una forma líneal y fácil de recorrer y de esta forma si la implementación cambia, al cliente no le afectará debido a que seguirá recibiendo los elementos de forma correcta.

Si alguna modificación se realiza sobre la estructura de datos o el orden de los elementos se recorre, sólo se tendrá que modificar el iterador para adecuarse a los cambios pero el cliente seguirá funcionando como siempre.

La implementación:

Una vez que entendimos el patrón y sabemos lo que tenemos que hacer, pasaremos a la implementación.

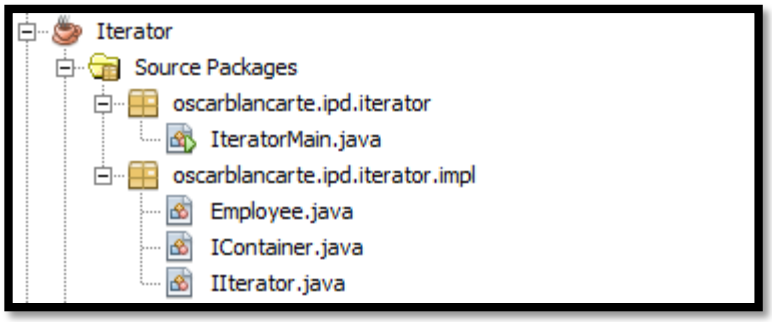


Ilustración 110: Estructura del proyecto Iterator.

En la imagen podemos apreciar la estructura que tiene el proyecto Iterator.

- **Oscarblancarte.ipd.iterator:** Paquete principal que contiene la clase ejecutable del proyecto la cual es *IteratorMain.*
- **Oscarblancarte.ipd.iterator.impl:** Paquete en el cual tenemos la implementación del Iterador.

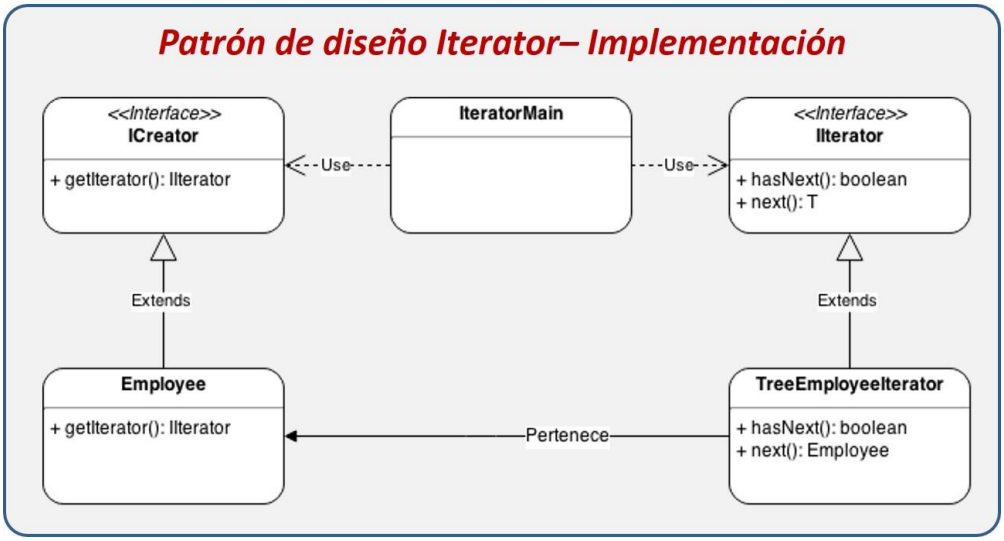


Ilustración 111: Diagrama de clases del proyecto Iterator.

En la imagen podemos apreciar las clases que componen el proyecto Iterator y cómo es que éstas se relacionadas entre sí.

Interface IContainer:

Interface que debe implementar las clases que quieran ser iteradas, la interface tiene un tipo genérico con el fin de que sea más fácil para el cliente utilizar el iterador.

```
1. package oscarblancarte.ipd.iterator.impl;
2.
3. public interface IContainer<T> {
4.     public IIterator<T> createIterator();
5. }
```

Se define únicamente el método *createIterator*, el cual lo utilizaremos para crear un nuevo iterador.

Interface IIterator:

Interface que define los métodos mínimos que un iterador debe tener. Se define un tipo genérico el cual será sobrescrito por la implementación.

```
1. package oscarblancarte.ipd.iterator.impl;
2.
3. public interface IIterator<T> {
4.     public  boolean hasNext();
5.     public T next();
6. }
```

Se definen los siguientes métodos:

- **hasNex:** Indica si hay más elementos por recorrer en el iterador.
- **next:** regresa el siguiente elemento del iterador, el cual será de tipo T.

Class Employee y la clase interna TreeEmployeeIterator:

La clase *Employee* representa la estructura a recorrer, en ella tenemos el nombre del empleado así como el rol al que pertenece, además tiene una lista de empleados los cuales son sus subordinados.

Podemos ver que la clase implementa *IContainer* y sobrescribe el tipo genérico a *Employeee*, con esto nos aseguramos que el iterador que cree será únicamente de empleados.

```
1.package oscarblancarte.ipd.iterator.impl;
2.
3.import java.util.ArrayList;
4.import java.util.Arrays;
5.import java.util.LinkedList;
6.import java.util.List;
7.
8.public class Employee implements IContainer<Employee> {
9.
10.        private String name;
11.        private String role;
12.        private List<Employee> subordinates;
13.
14.        public Employee(String role, String puesto, Employee... subordinates) {
15.            this.name = role;
16.            this.role = puesto;
17.            this.subordinates = Arrays.asList(subordinates);
18.        }
19.
20.        @Override
21.        public IIterator<Employee> createIterator() {
22.            return new TreeEmployeeIterator();
23.        }
24.
25.        private class TreeEmployeeIterator implements IIterator<Employee> {
26.
27.            private LinkedList<Employee> list = new LinkedList<Employee>();
28.            private int index = 0;
29.
30.            public TreeEmployeeIterator() {
31.                addRecursive(Employee.this);
32.            }
33.
34.            public void addRecursive(Employee employee) {
35.                list.add(employee);
36.                if (employee.getSubordinates() != null) {
37.                    for (Employee subordinate : employee.getSubordinates()) {
38.                        addRecursive(subordinate);
39.                    }
40.                }
41.            }
```

```
42.
43.         @Override
44.         public boolean hasNext() {
45.             if (list.isEmpty()) {
46.                 return false;
47.             }
48.
49.             return index < list.size();
50.         }
51.
52.         @Override
53.         public Employee next() {
54.             if (!hasNext()) {
55.                 throw new RuntimeException("No hay más elementos");
56.             }
57.             return list.get(index++);
58.         }
59.     }
60.
61.     public void setSubordinates(List<Employee> subordinates) {
62.         this.subordinates = subordinates;
63.     }
64.
65.     public void addSubordinate(Employee subordinate) {
66.         if (subordinate == null) {
67.             subordinates = new ArrayList<>();
68.         }
69.         subordinates.add(subordinate);
70.     }
71.
72.     @Override
73.     public String toString() {
74.         return "Employee{" + "name=" + name + ", role=" + role + '}';
75.     }
76.     /** GET and SET */
77. }
```

Analicemos las partes más interesantes de la clase:

- El método *getIterator* (línea 21) nos permitirá crear un iterador para recorrer los elementos de la estructura de empleados, este iterador será de tipo *TreeEmployeeIterator*.
- Definimos una clase interna *TreeEmployeeIterator* (línea 25), la cual será la clase iteradora, esta clase nos permitirá recorre el árbol de empleados de forma pre-orden y crea una lista con los objetos recuperados. El iterador define los siguientes métodos:

- **hashNext:** regresa un valor boolean indicando si existen más elementos en el iterador por recorrer
- **next:** regresa el siguiente elemento del iterador.

Class IteratorMain:

Clase principal de la aplicación desde la cual probaremos la efectividad del patrón Iterator.

```
1.  package oscarblancarte.ipd.iterator;
2.
3.  import oscarblancarte.ipd.iterator.impl.Employee;
4.  import oscarblancarte.ipd.iterator.impl.IIterator;
5.
6.  public class IteratorMain {
7.
8.      public static void main(String[] args) {
9.          Employee employee = new Employee("Juan", "CEO",
10.                     new Employee("Pedro", "Presidente",
11.                         new Employee("Liliana", "VP",
12.                             new Employee("Oscar", "Gerente",
13.                                 new Employee("Mario", "Developer"),
14.                                 new Employee("Rodolfo", "Developer")),
15.                             new Employee("Sofia", "Gerente",
16.                                 new Employee("Adrián", "Sr Developer"),
17.                                 new Employee("Rebeca", "Developer")
18.                             )
19.                         )
20.                     )
21.             );
22.
23.             IIterator<Employee> empIterator = employee.createIterator();
24.             while (empIterator.hasNext()) {
25.                 Employee emp = empIterator.next();
26.                 System.out.println("emp > " + emp.toString());
27.             }
28.         }
29.     }
```

De la línea 9 a la 21 creamos una estructura de empleados en forma de Árbol, los empleados enviados en el constructor de cada empleado representan los subordinados.

En la línea 23 creamos el Iterator de la estructura de empleados.

En la línea 24 se entra en un ciclo para recorrer el iterador hasta que el método *hasNext* nos regrese *false*, esto indicará que no hay más elementos que recorrer.

En la línea 29 utilizamos el iterador para obtener el siguiente elemento de la estructura.

Al finalizar el ciclo *while*, habremos terminado de recorrer el iterador y con esto habremos recorrido toda la estructura de empleados.

La ejecución:

Para probar la aplicación ejecutaremos la clase *IteratorMain* y tendremos el siguiente resultado:

```
run:
emp > Employee{name=Juan, puesto=CEO}
emp > Employee{name=Pedro, puesto=Precidente}
emp > Employee{name=Liliana, puesto=VP}
emp > Employee{name=Oscar, puesto=Gerente}
emp > Employee{name=Mario, puesto=Developer}
emp > Employee{name=Rodolfo, puesto=Developer}
emp > Employee{name=Sofia, puesto=Gerente}
emp > Employee{name=Adrian, puesto=Sr Developer}
emp > Employee{name=Rebeca, puesto=Developer}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 112: Resultado de la ejecución de la clase IteratorMain.

En la pantalla podremos ver que todos los empleados creados en forma de árbol son recorridos de forma lineal y esto lo hicimos sin saber cómo es que está implementada la estructura de datos internamente, sólo fue necesario crear el iterador e iterar los elementos.

Siguientes pasos:

Como hemos visto, el patrón iterator es sumamente útil y versátil ya que nos permite recorridos muy fácilmente, es muy común ver este patrón de diseño implementado en todas las colecciones del lenguaje Java.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios:

- Cambiemos el orden del recorrido del árbol, de pre-orden a in-orden y observemos si el cliente es afectado a la hora de recorrer los elementos.

- Creemos una nueva estructura para los empleados, esta vez utilizando un *Map*, en el cual la clave sea el empleado y como valor tenga una lista de los subordinados, luego creemos un iterador para recorrer la estructura.

Patrón Command

El patrón de diseño Command nos permite ejecutar operaciones sin conocer los detalles de la implementación de la misma. Las operaciones son conocidas como **comandos** y cada operación es implementada como una clase independiente que realiza una acción muy concreta, para lo cual, puede o no recibir parámetros para realizar su tarea. Una de las ventajas que ofrece este patrón es la de poder crear cuántos comandos requerimos y encapsularlos bajo una interface de ejecución.

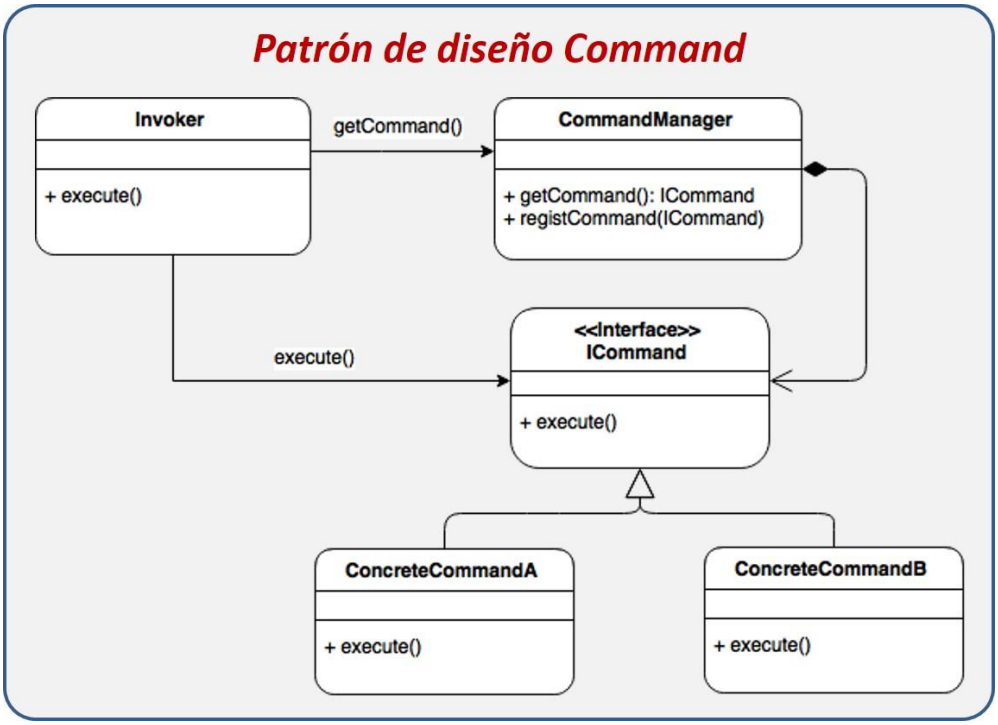


Ilustración 113: Estructura del patrón de diseño Command.

Los elementos del patrón Command se describen a continuación:

- **ICommand:** Interface que describe la estructura de los comandos, la cual define el método de ejecución genérico para todos los comandos.

- **Concrete Command:** Representan los comandos concretos, éstos deberán heredar de *ICommand*. Cada una de estas clases representa un comando que podrá ser ejecutado de forma independiente.
- **Command Manager:** Este componente nos servirá para administrar los comandos que tenemos disponibles en tiempo de ejecución, desde aquí podemos solicitar los comandos o registrar nuevos.
- **Invoker:** El *invoker* representa la acción que dispara alguno de los comandos.

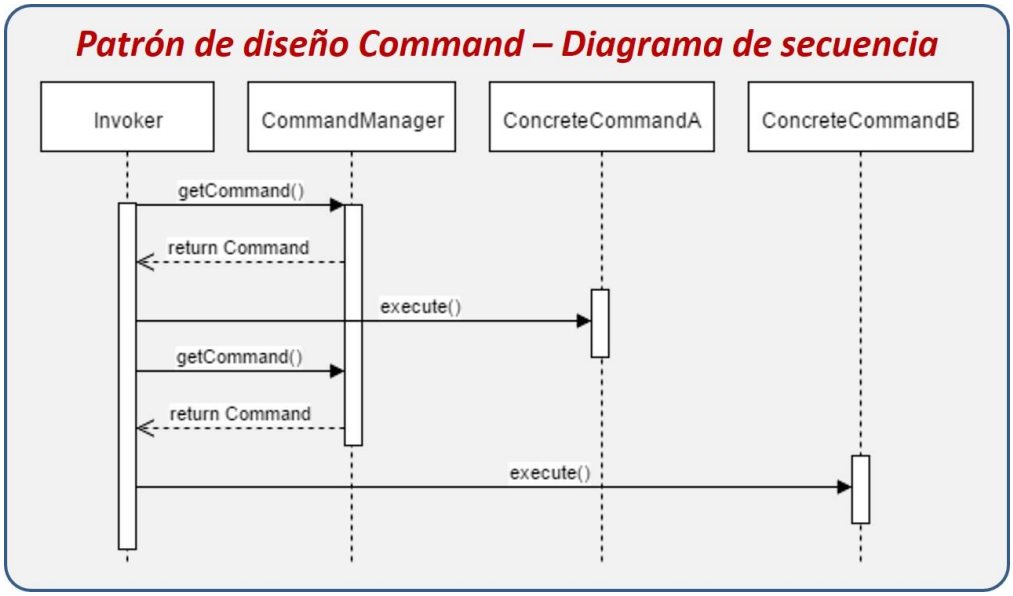


Ilustración 114: Diagrama de secuencia del patrón de diseño Command.

El diagrama anterior se interpreta de la siguiente manera:

1. El *invoker* obtiene un Comando del *CommandManager*.
2. El *invoker* ejecuta el comando.
3. El *invoker* obtiene otro Comando del *CommandManager*.
4. El *invoker* ejecuta el comando.

Observemos que utilizar el patrón de diseño *command* nos permite encapsular acciones mediante los comandos, de esta manera, el invoker ejecuta las acciones sin conocer la implementación concreta del comando. Además, todos los

comandos son ejecutados de la misma manera por lo que no habrá necesidad de conocer cómo ejecutar cada comando.

Cuándo utilizarlo:

- Cuando es necesario encapsular los detalles de ejecución de una serie de operaciones bajo una única interface.
- Cuando es necesario realizar aplicaciones que trabajan con operaciones de alto nivel, donde cada operación deberá ser implementada de forma simple e independiente.

El escenario:

En este escenario implementaremos nuestra propia Consola. Las consolas responden a base de comandos, estos comandos pueden o no necesitar parámetros para funcionar. Lo asombroso aquí es que la consola tiene la característica de saber siempre qué operación va a realizar sin tener que decirle qué clase o qué software utilice para realizar la tarea, ya que la consola es capaz de determinar el comando utilizado y ejecutar el componente o software que realizar la operación.

Otro punto importante es que la consola tiene la característica de agregarle comandos a medida que instalamos nuevo software como es el caso de Java, si intentamos ejecutar el comando java –version, con una computadora la cual no tiene instalado Java, verán que nos arrojará un mensaje de comando no encontrado (El mensaje dependerá del sistema operativo utilizado), sin embargo, si ejecutas de nuevo el comando una vez instalado java, entonces obtendrás un mensaje como el siguiente:
Java version “1.7.0_51″ (por ejemplo).
El patrón Command nos permite registrar nuevos comandos mediante el componente *CommandManager*.

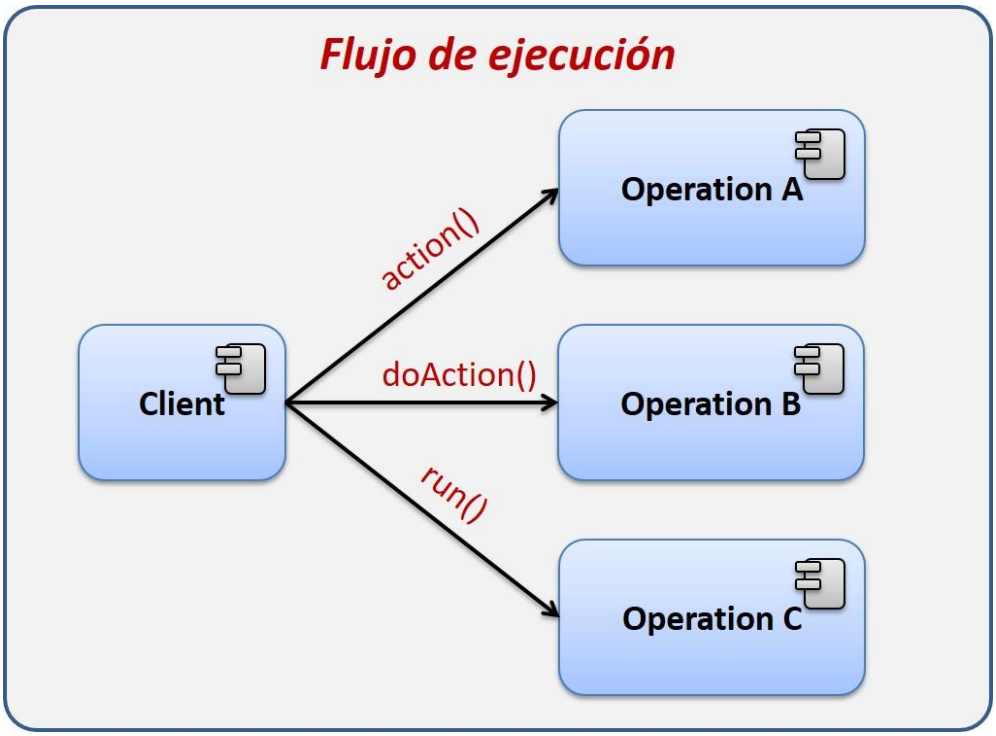


Ilustración 115: Implementación sin utilizar el patrón Command.

Podemos ver cómo el cliente interactúa con tres operaciones distintas. Al no utilizar el patrón de diseño Comando, está obligado a conocer la implementación exacta y la operación que tendrá que ejecutar, obligándonos a tener una secuencia de *elseif* o un *switch* para ejecutar la operación. Además, si una nueva operación es agregada tendríamos que modificar el programa para implementar la lógica de ejecución de la operación.

La consola que implementaremos deberá tener las siguientes operaciones:

- **Exit:** Finaliza la consola y termina con nuestro programa.
- **Memory:** Nos muestra en pantalla la cantidad de memoria total disponible, utilizada y máxima que podemos utilizar.
- **Echo:** Clásico comando utilizado para hacer un eco de lo escrito en pantalla.
- **Date:** Comando que nos imprime en pantalla la hora y la fecha.

- **File:** Comando que nos permite interactuar con archivos implementando operaciones básicas como crear, borrar o escribir.
- **Dir:** Nos permite realizar operaciones con directorios, ya sea crearlos o borrarlos.
- **WaitHello:** Comando que nos permite demostrar la ejecución en segundo plano, este comando al ejecutarse escribe "*Hello*!" Después de un tiempo determinado.
- **Batch:** Comando que ejecuta en lote un conjunto de comandos definidos en un archivo de texto.

Desde luego que los comandos aquí descritos son bastante simples ya que el objetivo es demostrar la utilización del patrón comando. Así que deberemos dejar preparada la consola para agregar nuevos comandos más adelante.

La solución:

Para resolver el escenario utilizaremos el patrón de diseño Comando por dos principales razones, la primera: nos permite definir operaciones que serán ejecutadas como comandos, lo cual significa que podremos ejecutar la operación que sea sin conocer los detalles de la implementación; las segunda: que nos permitirá agregar nuevos comandos sin modificar la consola.

Para iniciar el ejemplo tendremos que pensar primero en la funcionalidad que le daremos a nuestra consola. Las operaciones que deberemos implementar en la consola ya fueron expuestas en la sección del escenario, sin embargo, no hemos hablado de cómo es que estas operaciones deberán ser implementadas para sacar el máximo provecho al patrón de diseño Comando.

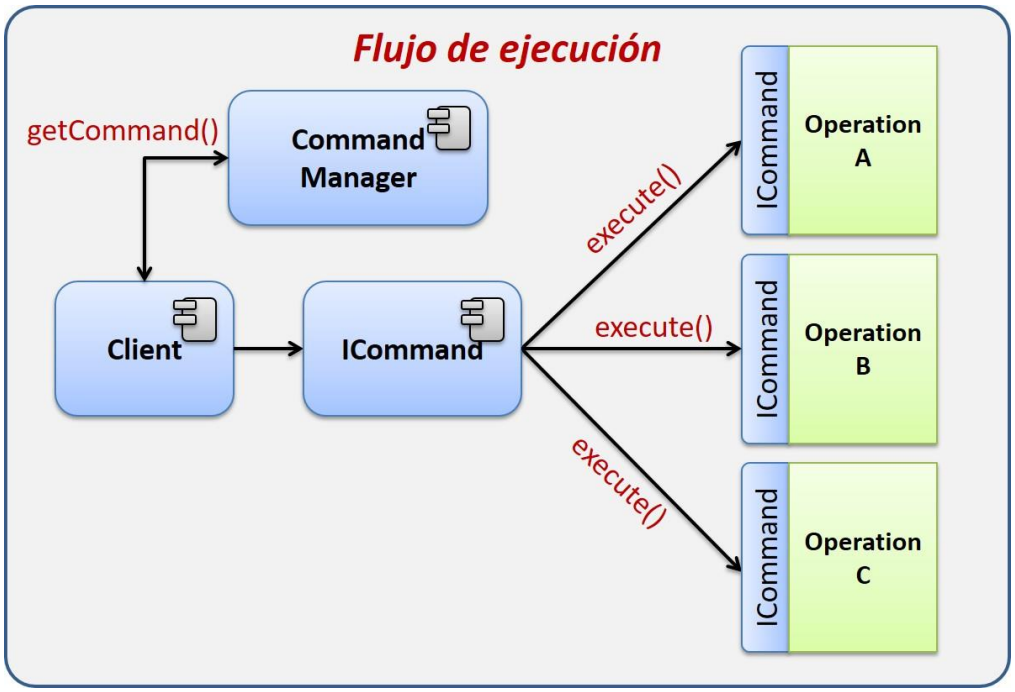


Ilustración 116: Ejecución utilizando el patrón Comando.

En la imagen podemos ver cómo es que se implementaría la consola utilizando el patrón de diseño Comando. Podemos apreciar que todos los comandos implementan la misma interface, esta interface definirá la forma en la que se

ejecutarán todos los comandos sin importar que tan simple o complejo sea. El componente CommandManager será utilizado por el cliente para obtener las instancias de los comandos.

En la siguiente imagen se describen como deberían de quedar implementados los comandos descritos anteriormente.

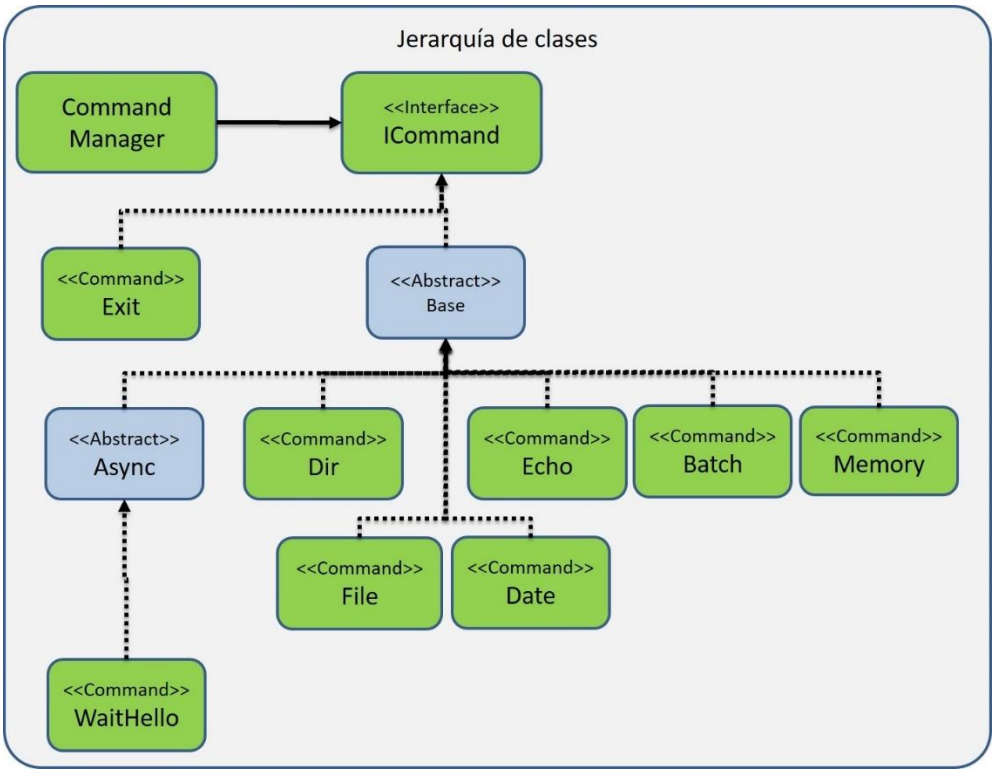


Ilustración 117: Diagrama de clases de los comandos.

Vemos los distintos comandos que deberán existir en la aplicación así como la jerarquía de los mismos. Podemos apreciar que todos los comandos descritos anteriormente están plasmados en el diagrama, pero también podemos ver algunos que no son precisamente comandos pero servirán como base para crear otros.

La clase Base y *Async* son un ejemplo de estos casos, los cuales por si solos no representan un comando dentro de la aplicación pero servirán como base para crear otros de una forma más simple. La clase Base nos servirá para crear todos

aquellos comandos que requieren imprimir en pantalla algún resultado o error. Por otra parte, la clase *Async* servirá para crear comandos que serán ejecutados de forma asíncrona.

La implementación:

Una vez que hemos hablado de cómo deberemos desarrollar la aplicación y que comprendemos los comandos que deberán ser parte de la aplicación, pasaremos a la implementación.

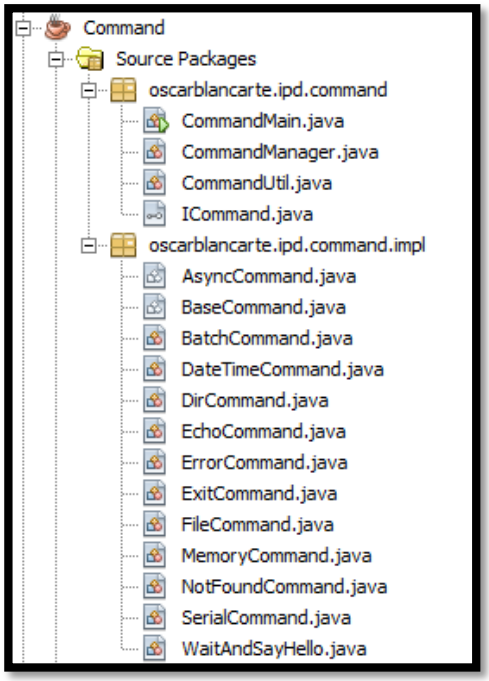


Ilustración 118: Estructura del proyecto Command.

En la imagen podemos apreciar los paquetes que componente el proyecto *Command*:

- **oscarblancarte.ipd.command:** Paquete con las clases base para crear la consola y la implementación del patrón Comando.
- **oscarblancarte.ipd.command.impl:** Paquete que contiene las clases correspondientes a los comandos que serán utilizados en la aplicación.

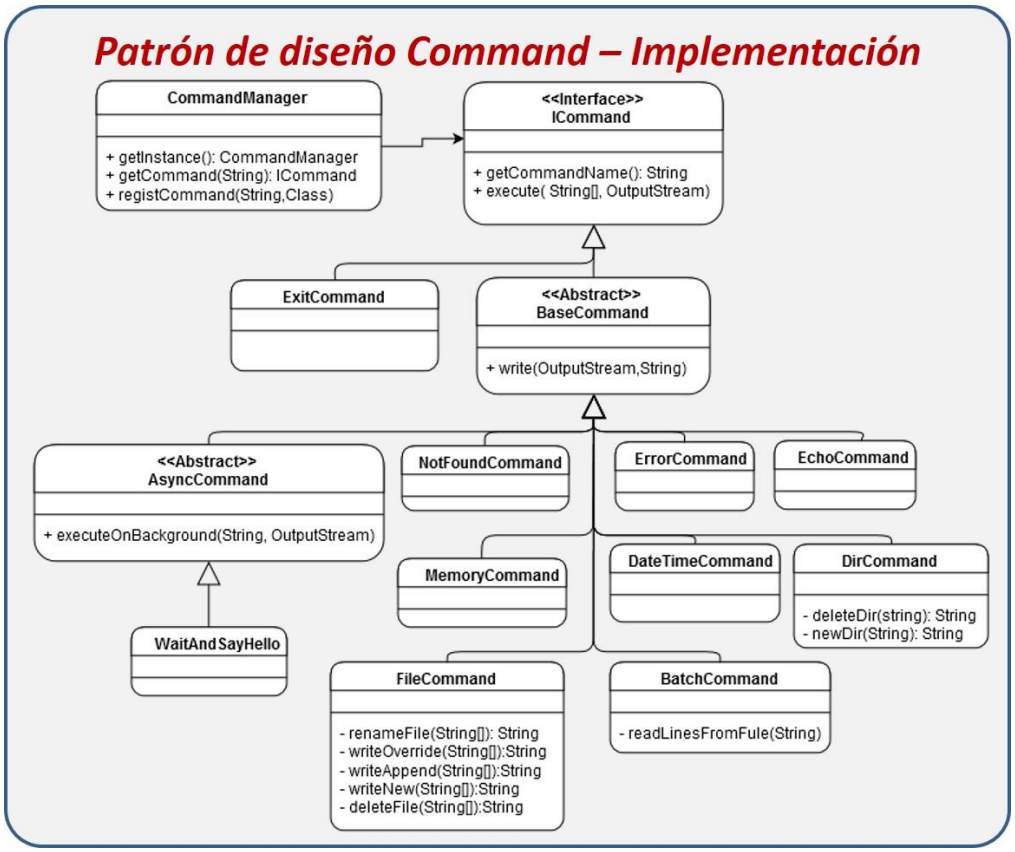


Ilustración 119: Diagrama de clases del proyecto Command.

Clase ICommand:

La interface *ICommand* tiene como propósito definir la estructura mínima de un comando, todos los comandos deberán implementar esta interface y los métodos:

- ***getCommandName*:** Regresa el nombre del comando.
- **execute:** Todos los comandos serán ejecutados por medio de este método, el cual tiene como entrada un arreglo de parámetros y un objeto OutputStream que servirá para imprimir cualquier cosa en pantalla.

```
1. package oscarblancarte.ipd.command;
2.
3. import java.io.OutputStream;
4.
```

```
5.  public interface ICommand {
6.
7.      public String getCommandName();
8.
9.      public void execute(String[] args, OutputStream out);
10.     }
```

Clase CommanManager:

La clase *CommandManager* es la encargada del registro y la creación de los comandos. Cuando el cliente requiera ejecutar algún comando éste lo solicitará al *CommandManager* para que le regrese el objeto adecuado para el comando solicitado.

```
1.  package oscarblancarte.ipd.command;
2.
3.  import java.util.HashMap;
4.  import oscarblancarte.ipd.command.impl.*
5.
6.  public class CommandManager {
7.
8.      private static CommandManager commandManager;
9.
10.         private static final HashMap<String, Class<? extends ICommand>> COMMANDS =
11.             new HashMap<String, Class<? extends ICommand>>();
12.
13.         private CommandManager() {
14.             registCommand(EchoCommand.COMMAN_NAME, EchoCommand.class);
15.             registCommand(DirCommand.COMMAND_NAME, DirCommand.class);
16.             registCommand(DateTimeCommand.COMMAND_NAME, DateTimeCommand.class);
17.             registCommand(MemoryCommand.COMMAN_NAME, MemoryCommand.class);
18.             registCommand(FileCommand.COMMAND_NAME, FileCommand.class);
19.             registCommand(ExitCommand.COMMAND_NAME, ExitCommand.class);
20.             registCommand(BatchCommand.COMMAND_NAME, BatchCommand.class);
21.             registCommand(WaitAndSayHello.COMMAND_NAME, WaitAndSayHello.class);
22.         }
23.
24.         public static synchronized CommandManager getIntance() {
25.             if (commandManager == null) {
26.                 commandManager = new CommandManager();
27.             }
28.
29.             return commandManager;
30.         }
31.
32.         public ICommand getCommand(String commandName) {
```

```
33.
34.         if (COMMANDS.containsKey(commandName.toUpperCase())) {
35.             try {
36.                 return COMMANDS.get(commandName.toUpperCase()).newInstance();
37.             } catch (Exception e) {
38.                 e.printStackTrace();
39.                 return new ErrorCommand();
40.             }
41.         } else {
42.             return new NotFoundCommand();
43.         }
44.     }
45.
46.     public void registCommand(String commandName, Class<? extends ICommand> command) {
47.         COMMANDS.put(commandName.toUpperCase(), command);
48.     }
49. }
```

La clase define los siguientes métodos:

- **getCommand:** Método invocado por el cliente para obtener el objeto del comando, para esto es requerido el nombre del comando a ejecutar. Los comandos son recuperados de un *Map* el cual contiene una relación del nombre del comando con la clase asociada al comando. En caso de que el comando no exista será regresado el comando *CommandNotFound* del cual hablaremos más adelante.
- **registCommand:** Método utilizado para registrar los comandos que tendremos disponibles en tiempo de ejecución. Todo comando deberá estar registrado al momento de ser solicitado de lo contrario no será encontrado.
- **getIntance:** Método que nos permite obtener la instancia Singleton de la clase *CommanManager.*

Observemos que en el constructor de la clase *CommandManager* registramos todos los comandos que estarán disponibles en tiempo de ejecución.

Comandos: Las siguientes clases representan las implementaciones de los comandos que tendremos disponibles en la aplicación, los cuales fueron

registrados en el constructor de *CommandManager*. Las clases siguientes representan los comandos que analizamos en la Ilustración 19.

Clase ExitCommand:

Iniciaremos con el comando *Exit* debido que es el más simple de todos. Este comando implementa *ICommand* y nos permitirá finalizar la aplicación de forma inmediata.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.OutputStream;
4.  import oscarblancarte.ipd.command.ICommand;
5.
6.  public class ExitCommand implements ICommand {
7.
8.      public static final String COMMAND_NAME = "exit";
9.
10.         @Override
11.         public String getCommandName() {
12.             return COMMAND_NAME;
13.         }
14.
15.         @Override
16.         public void execute(String[] args, OutputStream out) {
17.             System.exit(0);
18.         }
19.
20.     }
```

Cuando el comando es utilizado, el método *execute* es llamado finalizando así la aplicación con la instrucción System.exit.

Clase BaseCommand:

La clase *BaseCommand* no es como tal un comando, en su lugar, es una clase base y abstracta que podrá ser extendida por otros comandos para obtener la funcionalidad básica de un comando como por ejemplo, imprimir en la consola.

```
1. package oscarblancarte.ipd.command.impl;
2.
3. import java.io.OutputStream;
4. import oscarblancarte.ipd.command.ICommand;
5.
6. public abstract class BaseCommand implements ICommand {
7.
8.     @Override
9.     public abstract String getCommandName();
10.
11.         @Override
12.         public abstract void execute(String[] args, OutputStream out);
13.
14.         public void write(OutputStream out, String message) {
15.             try {
16.                 out.write(message.getBytes());
17.                 out.flush();
18.             } catch (Exception e) {
19.                 e.printStackTrace();
20.             }
21.         }
22.     }
```

Se define el siguiente método:

- **Write:** Ofrece la posibilidad de imprimir mensajes en pantalla de una forma ordenada. Mediante este método se controla la forma en que los comandos imprimen en pantalla, con la finalidad de que si en el futuro la forma de imprimir en pantalla cambia sólo será necesario modificar esta clase.

Clase DirCommand:

La clase *DirCommand* implementa el comando *dir* y nos permite realizar operaciones sobre los directorios de nuestro sistema de archivos, como crear o borrar una carpeta. El comando recibe como parámetro la operación a realizar (crear o borrar {-N|-D}) más la ruta de la carpeta.

```
1. package oscarblancarte.ipd.command.impl;
2.
3. import java.io.File;
4. import java.io.OutputStream;
5. import java.nio.channels.WritableByteChannel;
6.
```

```
7.  public class DirCommand extends BaseCommand {
8.
9.     public static final String COMMAND_NAME = "dir";
10.
11.        @Override
12.        public String getCommandName() {
13.           return COMMAND_NAME;
14.        }
15.
16.        @Override
17.        public void execute(String[] args, OutputStream out) {
18.           if (args == null || args.length < 2) {
19.              write(out, COMMAND_NAME + " argumentos insuficientes");
20.           }
21.
22.           String operation = args[0];
23.           if ("-D".equals(operation.toUpperCase())) {
24.              write(out, deleteDir(args[1]));
25.           } else if ("-N".equals(operation.toUpperCase())) {
26.              write(out, newDir(args[1]));
27.           } else {
28.              write(out, "Se esperaba una operación correcta -d | -n");
29.           }
30.        }
31.
32.        private String deleteDir(String url) {
33.
34.           try {
35.              File file = new File(url);
36.              if (!file.exists()) {
37.                 return "El archivo no existe";
38.              }
39.
40.              if (!file.canWrite()) {
41.                 return "Privilegios insuficientes";
42.              }
43.
44.              file.delete();
45.              return "";
46.           } catch (Exception e) {
47.              return "ERROR: " + e.getMessage();
48.           }
49.
50.        }
51.
52.        private String newDir(String url) {
53.           try {
54.              File file = new File(url);
55.              if (file.exists()) {
56.                 return "El archivo ya existe";
57.              }
58.
59.              file.mkdirs();
```

```
60.                 return "";
61.             } catch (Exception e) {
62.                 return "ERROR: " + e.getMessage();
63.             }
64.         }
65.
66.     }
```

Se definen los siguientes métodos:

- **deleteDir:** Borra un directorio en nuestro sistema de archivos siempre y cuando ésta exista.
- **newDir:** Crear un nuevo directorio en nuestro sistema de archivos.
- **execute:** Método ejecutado desde el cliente, éste se encarga de delegar la tarea a los métodos anteriores según los parámetros enviados.

FileCommand:

La clase *FileCommand* implementa el comando *file*. Este comando nos permitirá realizar operaciones con archivos. Las operaciones que soporta son: escribir en archivos, borrar, renombrar, sobrescribir y adherir contenido a un archivo existente.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.File;
4.  import java.io.FileOutputStream;
5.  import java.io.FileWriter;
6.  import java.io.OutputStream;
7.  import java.util.Arrays;
8.
9.  public class FileCommand extends BaseCommand {
10.
11.         public static final String COMMAND_NAME = "file";
12.         private static final String WRITE_APPEND = "-WA";
13.         private static final String WRITE_OVERRIDE = "-WO";
14.         private static final String WRITE_NEW = "-WN";
15.         private static final String RENAME_FILE = "-R";
16.         private static final String DELETE_FILE = "-D";
17.
18.         @Override
19.         public void execute(String[] args, OutputStream out) {
20.             if (args.length < 2) {
```

```
21.             write(out, "Parámetros insuficientes");
22.             return;
23.         }
24.
25.         String operation = args[0].toUpperCase();
26.
27.         String[] reduce = Arrays.copyOfRange(args, 1, args.length);
28.         if (WRITE_APPEND.equals(operation)) {
29.             write(out, writeAppend(reduce));
30.         } else if (WRITE_NEW.equals(operation)) {
31.             write(out, writeNew(reduce));
32.         } else if (WRITE_OVERRIDE.equals(operation)) {
33.             write(out, writeOverride(reduce));
34.         } else if (RENAME_FILE.equals(operation)) {
35.             write(out, renameFile(reduce));
36.         } else if (DELETE_FILE.equals(operation)) {
37.             write(out, deleteFile(reduce));
38.         } else {
39.             write(out, "No se encontró la operación {"
40.                 + WRITE_APPEND + "|" + WRITE_NEW + "|" + WRITE_OVERRIDE + "|"
41.                 + RENAME_FILE + "|DELETE_FILE}");
42.         }
43.     }
44.
45.     private String renameFile(String[] args) {
46.         String filePath = args[0];
47.         String newFileName = args[1];
48.         try {
49.             File file = new File(filePath);
50.             file.renameTo(new File(newFileName));
51.             return "";
52.         } catch (Exception e) {
53.             return "ERROR: " + e.getMessage();
54.         }
55.     }
56.
57.     private String writeOverride(String[] args) {
58.         String filePath = args[0];
59.         String fileContent = args[1];
60.
61.         try {
62.             File file = new File(filePath);
63.             if (!file.exists()) {
64.                 if (!file.createNewFile()) {
65.                     return "ERROR: Error al crear el archivo";
66.                 }
67.             }
68.             FileWriter fileW = new FileWriter(file);
69.             fileW.write(fileContent.toCharArray());
70.             fileW.flush();
71.             fileW.close();
72.             return "";
73.         } catch (Exception e) {
```

```
74.             return "ERROR: " + e.getMessage();
75.         }
76.     }
77.
78.     private String writeAppend(String[] args) {
79.         String filePath = args[0];
80.         String fileContent = args[1];
81.
82.         try {
83.             File file = new File(filePath);
84.             if (!file.exists()) {
85.                 return "ERROR: El archivo no existe";
86.             }
87.
88.             FileWriter fileW = new FileWriter(file, true);
89.             fileW.append(fileContent);
90.             fileW.flush();
91.             fileW.close();
92.             return "";
93.         } catch (Exception e) {
94.             return "ERROR: " + e.getMessage();
95.         }
96.     }
97.
98.     private String writeNew(String[] args) {
99.         String filePath = args[0];
100.        String fileContent = args[1];
101.
102.        try {
103.            File file = new File(filePath);
104.            if (file.exists()) {
105.                return "ERROR: El archivo ya existe";
106.            }
107.            if (!file.createNewFile()) {
108.                return "ERROR: No fue posible crear el archivo";
109.            }
110.
111.            FileWriter fileW = new FileWriter(file);
112.            fileW.write(fileContent.toCharArray());
113.            fileW.flush();
114.            fileW.close();
115.            return "";
116.        } catch (Exception e) {
117.            return "ERROR: " + e.getMessage();
118.        }
119.    }
120.
121.    private String deleteFile(String[] args) {
122.        String filePath = args[0];
123.        File file = new File(filePath);
124.        if (!file.delete()) {
125.            return "No fue posible eliminar el archivo";
126.        }
```

```
127.
128.            return "";
129.        }
130.
131.        @Override
132.        public String getCommandName() {
133.            return COMMAND_NAME;
134.        }
135.    }
```

Las operaciones son las siguientes:

- **WRITE_APPEND = -WA:** Permite escribir en un *archivo* conservando el contenido existente previamente. Recibe dos parámetros, el primero corresponde al path del archivo y el segundo el contenido a escribir. Esta operación se define en el método *writeAppend***.**
- **WRITE_OVERRIDE = -WO:** Permite escribir en un archivo sobrescribiendo su contenido. Recibe dos parámetros, el primero corresponde al path del archivo y el segundo el contenido a escribir. Esta operación se define en el método *writeOverride*.
- **WRITE_NEW = -WN:** Permite crear un archivo y luego escribir en él. Recibe dos parámetros, el primero corresponde al path del archivo y el segundo el contenido a escribir. Esta operación se define en el método *writeNew*.
- **RENAME_FILE = -R:** Permite renombrar un archivo existente. Recibe dos parámetros, el primero corresponde al path del archivo y el segundo corresponde al nuevo nombre. Esta operación se define en el método *renameFile*.
- **DELETE_FILE = -D:** Permite borrar un archivo. Recibe únicamente el path del archivo a eliminar. Esta operación se define en el método *deleteFile*.

EchoCommand:

La clase *EchoCommand* implementa el comando *echo*, el cual repite en pantalla lo que se le manda como parámetro.

```
1. package oscarblancarte.ipd.command.impl;
2.
3. import java.io.OutputStream;
4. import java.util.Arrays;
5.
6. public class EchoCommand extends BaseCommand {
7.
8.     public static final String COMMAN_NAME = "echo";
9.
10.         @Override
11.         public void execute(String[] args, OutputStream out) {
12.             String message = getCommandName() + " " + Arrays.toString(args);
13.             write(out, message);
14.         }
15.
16.         @Override
17.         public String getCommandName() {
18.             return COMMAN_NAME;
19.         }
20.     }
```

DateCommand:

La clase *DateTimeCommand* implementa el comando *date* y nos permite conocer la fecha y hora del sistema en el formato que deseemos. El comando únicamente recibe como entrada el formato en el que queremos ver la fecha y hora basado en el estándar Java.
Para ver los formatos válidos visite la página:

https://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html

```
1. package oscarblancarte.ipd.command.impl;
2.
3. import java.io.OutputStream;
4. import java.text.SimpleDateFormat;
5. import java.util.Date;
6.
7. public class DateTimeCommand extends BaseCommand {
8.
9.     public static final String COMMAND_NAME = "date";
10.
11.         @Override
12.         public String getCommandName() {
13.             return COMMAND_NAME;
14.         }
```

```
15.
16.         @Override
17.         public void execute(String[] args, OutputStream out) {
18.
19.             SimpleDateFormat dateFormater = null;
20.             if (args == null) {
21.                 dateFormater = new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");
22.             } else {
23.                 try {
24.                     dateFormater = new SimpleDateFormat(args[0]);
25.                 } catch (Exception e) {
26.                     write(out, "formato inválido");
27.                     return;
28.                 }
29.
30.             }
31.             String date = dateFormater.format(new Date());
32.             write(out, date);
33.         }
34.     }
```

MemoryCommand:

La clase *MemortCommand* implementa el comando *memory*, el cual no tiene parámetros y nos arroja la información de la memoria que utiliza nuestra aplicación.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.OutputStream;
4.
5.  public class MemoryCommand extends BaseCommand {
6.
7.      public static final String COMMAN_NAME = "memory";
8.
9.      @Override
10.         public String getCommandName() {
11.             return COMMAN_NAME;
12.         }
13.
14.         @Override
15.         public void execute(String[] args, OutputStream out) {
16.             double heap = Runtime.getRuntime().totalMemory() / 1000000d;
17.             double heapMax = Runtime.getRuntime().maxMemory() / 1000000d;
18.             double heapFree = Runtime.getRuntime().freeMemory() / 1000000d;
19.
20.             String salida = "Heap: " + heap + "\nMax Heap: "
```

```
21.                 + heapMax + "\nFree Heap: " + heapFree;
22.             write(out, salida);
23.         }
24.
25.     }
```

BatchCommand:

La clase *BatchCommand* implementa el comando *batch* y nos permite abrir un archivo de comandos, este archivo es parecido a los archivos en *batch* ya que podemos escribir un conjunto de comandos que serán ejecutados en el orden en el que se encuentran en el archivo (se asume que cada línea es un comando). El comando únicamente recibe como parámetro el *Path* del archivo a ejecutar.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.File;
4.  import java.io.FileInputStream;
5.  import java.io.OutputStream;
6.  import java.util.Arrays;
7.  import oscarblancarte.ipd.command.CommandManager;
8.  import oscarblancarte.ipd.command.CommandUtil;
9.  import oscarblancarte.ipd.command.ICommand;
10.
11.     public class BatchCommand extends BaseCommand {
12.
13.         public static final String COMMAND_NAME = "batch";
14.
15.         @Override
16.         public String getCommandName() {
17.             return COMMAND_NAME;
18.         }
19.
20.         @Override
21.         public void execute(String[] args, OutputStream out) {
22.
23.             if (args == null || args.length < 1) {
24.                 write(out, "Número de parámetros inválido");
25.                 return;
26.             }
27.
28.             CommandManager manager = CommandManager.getIntance();
29.             String[] lines = readLinesFromFile(args[0]);
30.             for (String line : lines) {
31.                 String[] argsCommand = CommandUtil.tokenizerArgs(line);
32.                 ICommand command = manager.getCommand(argsCommand[0]);
```

```
33.                 String[] reduce = Arrays.copyOfRange(argsCommand, 1, argsCommand.length);
34.                 command.execute(reduce, out);
35.                 write(out, "\n");
36.             }
37.             write(out, "Batch ejecutado");
38.         }
39.
40.         private String[] readLinesFromFile(String filePath) throws RuntimeException {
41.             File file = new File(filePath);
42.             FileInputStream stream = null;
43.             try {
44.                 if (!file.exists()) {
45.                     throw new RuntimeException("Archivo no encontrado");
46.                 }
47.
48.                 stream = new FileInputStream(file);
49.                 byte[] byteArray = new byte[stream.available()];
50.                 stream.read(byteArray);
51.
52.                 String commands = new String(byteArray);
53.                 String[] lines = commands.split("\n");
54.                 return lines;
55.             } catch (Exception e) {
56.                 throw new RuntimeException(e.getMessage());
57.             } finally {
58.                 try {
59.                     stream.close();
60.                 } catch (Exception e2) {
61.                 }
62.             }
63.         }
64.
65.     }
```

La clase define el método *readLinesFromFile* y es utilizado para separar las líneas del archivo *batch* para ser ejecutadas.

AsyncCommand:

Esta clase es parecida a *BaseCommand* y no es como tal un comando ya que es una *clase Abstract*, tiene como propósito demostrar que podemos implementar comandos que se ejecuten en segundo plano o de forma asíncrona.
Esta clase agrega el método *executeOnBackground* en el cual deberemos implementar la funcionalidad que deseamos que realice el comando, ya que el

método original *execute* es sobrescrito para ejecutar el método *executeOnBackground* en un hilo independiente.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.OutputStream;
4.
5.  public abstract class AsyncCommand extends BaseCommand {
6.
7.      public void execute(final String[] args, final OutputStream out) {
8.          new Thread(new Runnable() {
9.              public void run() {
10.                     executeOnBackground(args, out);
11.                 }
12.             }).start();
13.         }
14.
15.         public abstract void executeOnBackground(String[] args, OutputStream out);
16.     }
```

Clase WaitAndSeyHello:

La clase *WaitAndSayHello* extiende de *AsyncCommand* e implementa el comando *waithello*, lo cual lo convierte automáticamente en un comando que se ejecuta en segundo plano. A este comando únicamente se le envía un tiempo en milisegundos que utilizará para esperar antes de imprimir en pantalla *Hello!*
Este comando fue implementado sólo como ejemplo para demostrar que es posible ejecutar comandos en segundo plano.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.OutputStream;
4.
5.  public class WaitAndSayHello extends AsyncCommand {
6.
7.      public static final String COMMAND_NAME = "waithello";
8.
9.      @Override
10.         public void executeOnBackground(String[] args, OutputStream out) {
11.             if (args == null || args.length < 1) {
12.                 write(out, "Parámetros insuficientes");
13.                 return;
14.             }
```

```
15.
16.         Long time = null;
17.         try {
18.             time = Long.parseLong(args[0]);
19.         } catch (Exception e) {
20.             write(out, "Tiempo inválido");
21.             return;
22.         }
23.
24.         try {
25.             Thread.sleep(time.longValue());
26.             write(out, "Hello!!");
27.         } catch (Exception e) {
28.             e.printStackTrace();
29.         }
30.     }
31.
32.     @Override
33.     public String getCommandName() {
34.         return COMMAND_NAME;
35.     }
36.
37. }
```

Clase ErrorCommand:

La clase ErrorCommand no está definida en nuestro diagrama y es utilizada por el *CommandManager* únicamente para regresar un error al intentar instanciar un comando. Este comando no puede ser ejecutado por el cliente ya que no es un comando registrado, sin embargo, *CommandManager* lo maneja como un comando propio del sistema para representar un error.
El cliente nunca se entera de que este comando es retornado ya que simplemente lo ejecuta por *CommandManager* una vez que es devuelto.

```
1. package oscarblancarte.ipd.command.impl;
2.
3. import java.io.OutputStream;
4. import java.nio.channels.WritableByteChannel;
5.
6. public class ErrorCommand extends BaseCommand {
7.
8.     private static final String COMMAND_NAME = "ERROR";
9.
10.     @Override
11.     public String getCommandName() {
```

```
12.             return COMMAND_NAME;
13.         }
14.
15.         @Override
16.         public void execute(String[] args, OutputStream out) {
17.             String message = "Error al invocar el comando";
18.             write(out, message);
19.         }
20.
21.     }
```

Clase NotFoundCommand:

Comando parecido a *ErrorCommand* pero éste es regresado cuando un comando no es encontrado. El cliente nunca se entera de que este comando es retornado ya que simplemente lo ejecuta una vez que es devuelto por *CommandManager*.

```
1.  package oscarblancarte.ipd.command.impl;
2.
3.  import java.io.OutputStream;
4.
5.  public class NotFoundCommand extends BaseCommand {
6.
7.      private static final String COMMAND_NAME = "NOT FOUND";
8.
9.      @Override
10.         public String getCommandName() {
11.             return COMMAND_NAME;
12.         }
13.
14.         @Override
15.         public void execute(String[] args, OutputStream out) {
16.             write(out, "Comando no encontrado");
17.         }
18.     }
```

Clase CommandUtil:

Clase que utilizamos únicamente para procesar los datos tecleados por el usuario y dividir las instrucciones en *tokens*, los *tokens* pueden ser el nombre del comando y los parámetros. La regla que se sigue para dividir los *tokens* es la siguiente:

Cada palabra separada por un espacio es considera como un *token* a excepción que se encuentre entre comillas, por ejemplo: *"Hola Mundo",* en este caso toda la oración es considerada como un *token*.

```
1.  package oscarblancarte.ipd.command;
2.
3.  import java.util.ArrayList;
4.  import java.util.Arrays;
5.  import java.util.List;
6.
7.  public class CommandUtil {
8.
9.      public static String[] tokenizerArgs(String args) {
10.             List<String> tokens = new ArrayList<String>();
11.             char[] charArray = args.toCharArray();
12.
13.             String contact = "";
14.             boolean inText = false;
15.             for (char c : charArray) {
16.                 if (c == ' ' && !inText) {
17.                     if (contact.length() != 0) {
18.                         tokens.add(contact);
19.                         contact = "";
20.                     }
21.                 } else if (c == '"') {
22.                     if (inText) {
23.                         tokens.add(contact);
24.                         contact = "";
25.                         inText = false;
26.                     } else {
27.                         inText = true;
28.                     }
29.                 } else {
30.                     contact += c;
31.                 }
32.             }
33.             if (contact.trim().length() != 0) {
34.                 tokens.add(contact.trim());
35.             }
36.
37.             String[] argsArray = new String[tokens.size()];
38.             argsArray = tokens.toArray(argsArray);
39.             return argsArray;
40.         }
41.
42.         public static void main(String[] args) {
43.             String commanda = "file -an c:/dummy/dummy.txt \"Hola mundo tres veces\"";
44.             String[] tokens = CommandUtil.tokenizerArgs(commanda);
45.             System.out.println(Arrays.toString(tokens));
```

```
46.         }
47.     }
```

Clase CommandMain:

La clase *CommandMain* ejecuta la aplicación creando un hilo indeterminado que estará a la escucha de los comandos que le solicitemos por medio de la consola. Los comandos que le podremos solicitar son los que registramos en el *CommandManager*.

```
1.  package oscarblancarte.ipd.command;
2.
3.  import java.util.Arrays;
4.  import java.util.Scanner;
5.
6.  public class CommandMain {
7.
8.      public static void main(String[] args) {
9.          System.out.println("Command Line is Start");
10.             CommandManager manager = CommandManager.getIntance();
11.
12.             Scanner in = new Scanner(System.in);
13.             while (true) {
14.                 String line = in.nextLine();
15.                 if (line.trim().length() == 0) {
16.                     continue;
17.                 }
18.                 String[] commands = CommandUtil.tokenizerArgs(line);
19.                 String commandName = commands[0];
20.                 String[] commandArgs = null;
21.                 if (commands.length > 1) {
22.                     commandArgs = Arrays.copyOfRange(commands, 1, commands.length);
23.                 }
24.                 ICommand command = manager.getCommand(commandName);
25.                 command.execute(commandArgs, System.out);
26.                 System.out.println("");
27.             }
28.         }
29.     }
```

Los comandos son leídos por el teclado en la línea 14 y son convertidos en *tokens* en la línea 18.

El *token* de la posición cero representa el nombre del comando y es utilizado para buscarlo mediante el *CommandManager* en la línea 24.
Finalmente, el comando es ejecutado en la línea 25, la consola espera la terminación de la ejecución para esperar el siguiente comando.

La Ejecución:

Ya tenemos listo todo, por lo que comenzaremos ejecutando la aplicación para obtener el siguiente resultado:

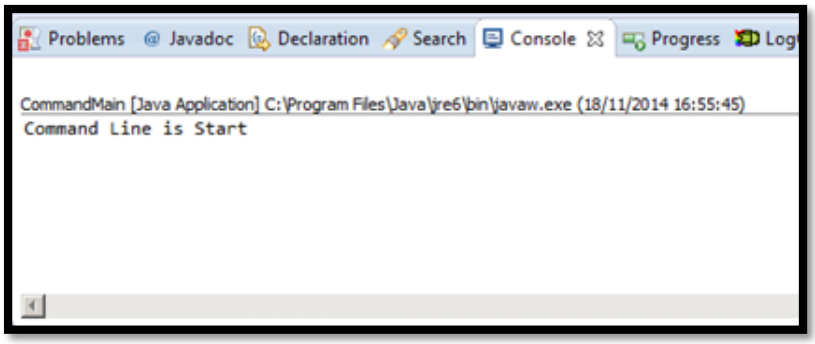


Ilustración 120: Muestra cómo se verá la aplicación una vez ejecutada

En este punto la consola se encuentra en ejecución y está a la espera de que escribamos los comandos.

Ejecución 1: Comando date.

Iniciaremos ejecutando el comando *date* y le pediremos que nos arroje la fecha en el formato "**DD/MM/YYYY**"; seguido, le pediremos la hora en el formato "**HH:MM:SS**" y finalmente le pediremos fecha y hora en formato "**DDMMYYYY HHMMSS**".

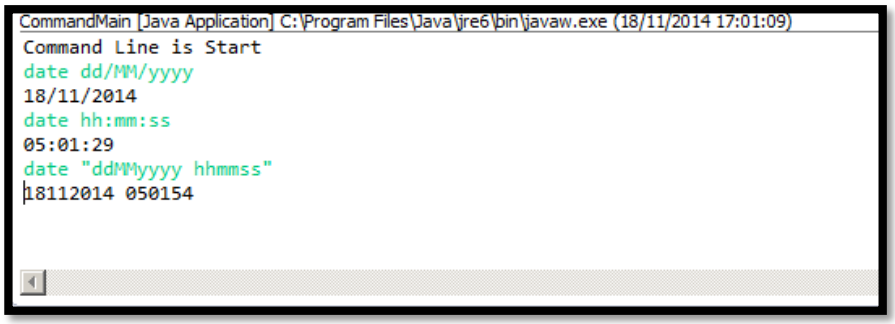


Ilustración 121: Resultado de la ejecución del comando *date*.

Ejecución 2: Comando dir.

Para el siguiente ejemplo utilizaremos el comando *Dir* para crear un nuevo directorio que después borraremos.

Ilustración 122: Ejecución del comando *dir* para crear la carpeta dummy3.

El comando *dir* es ejecutado con el parámetro –n (new) que indica que la carpeta deberá ser creada.

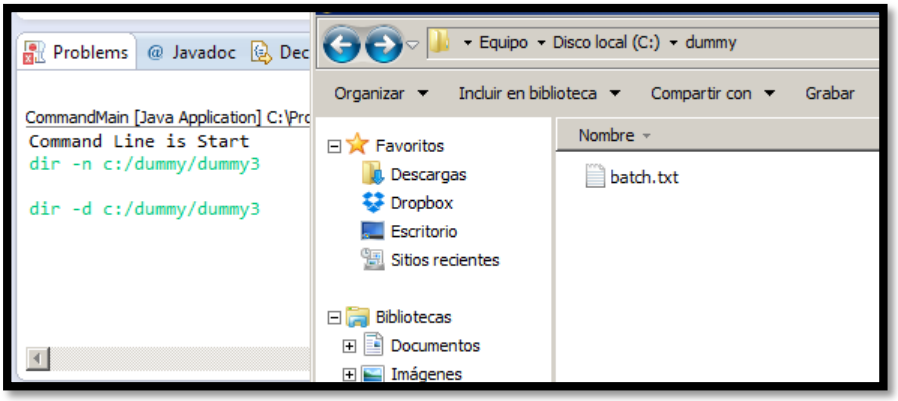


Ilustración 123: Ejecución del comando *dir* para borrar la carpeta dummy3 creada en el paso anterior.

El comando *dir* es ejecutado con el parámetro –d (*delete*) el cual indica que la carpeta deberá ser eliminada.

Ejecución 3: Comando echo.

Para el siguiente ejemplo utilizaremos el comando *echo* para decir "Hola Mundo".

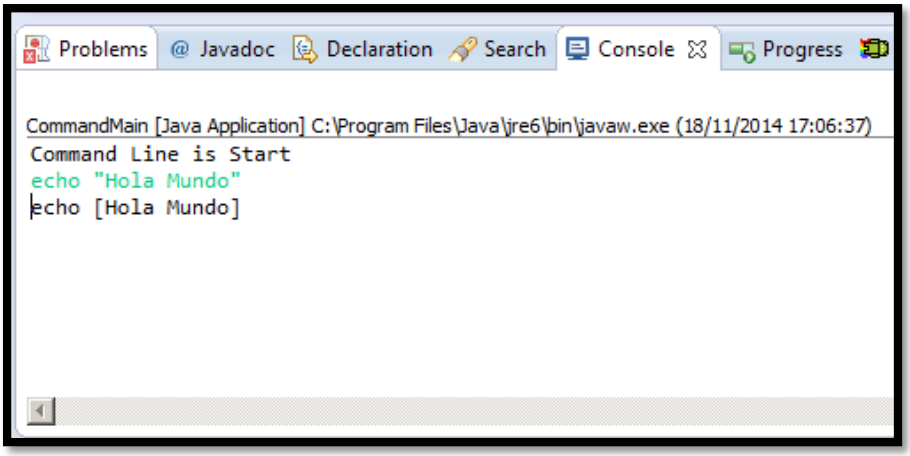


Ilustración 124: Ejecución del comando *echo*, imprime en pantalla Hola Mundo.

Ejecución 4: Comando file.

Para el siguiente ejemplo utilizaremos el comando *File* (que crea un nuevo archivo de texto)**.** El parámetro **–wn** indica que se deberá crear el archivo y escribir en él "**Hola Mundo**".

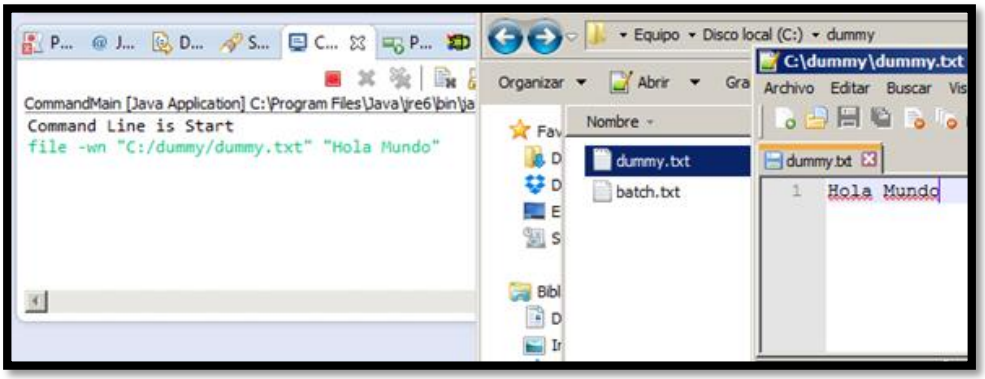


Ilustración 125: Ejecución del comando *file* para crear un nuevo archivo y escribir en él Hola Mundo.

Ejecución 5: Comando memory.

El siguiente comando a utilizar es *memory,* el cual nos dará datos acerca de la memoria de la aplicación.

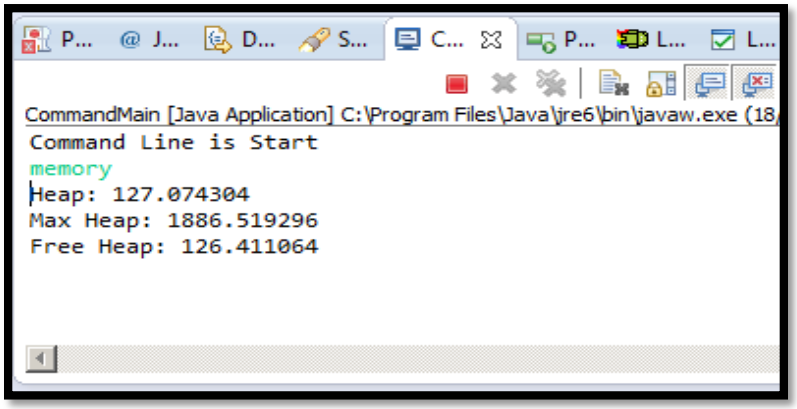


Ilustración 126: Ejecución del comando *memory*

Ejecución 6: Comando file.

Utilizaremos nuevamente el comando *File* para renombrar el archivo dummy.txt a helloword.txt

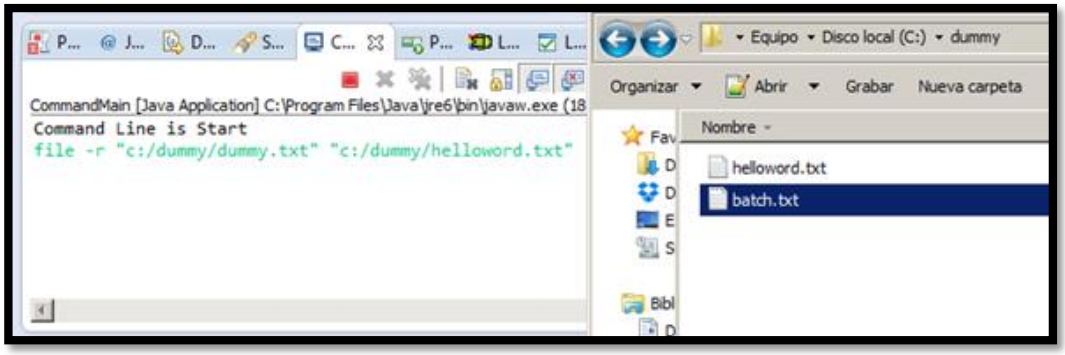


Ilustración 127: Ejecución del comando *file* en el cual se renombra un archivo.

El parámetro –r (raname) del comando *file* es utilizado para solicitar el cambio de nombre de un archivo.

Ejecución 7: Comando waithello.

Hasta aquí, solamente hemos ejecutado comandos síncronos, pero en este ejemplo ejecutaremos el comando *waithello* implementado por la clase *WaitAndSeyHello*, el cual lo ejecutaremos y le diremos que espere 10 segundos antes de decir *Hello!*

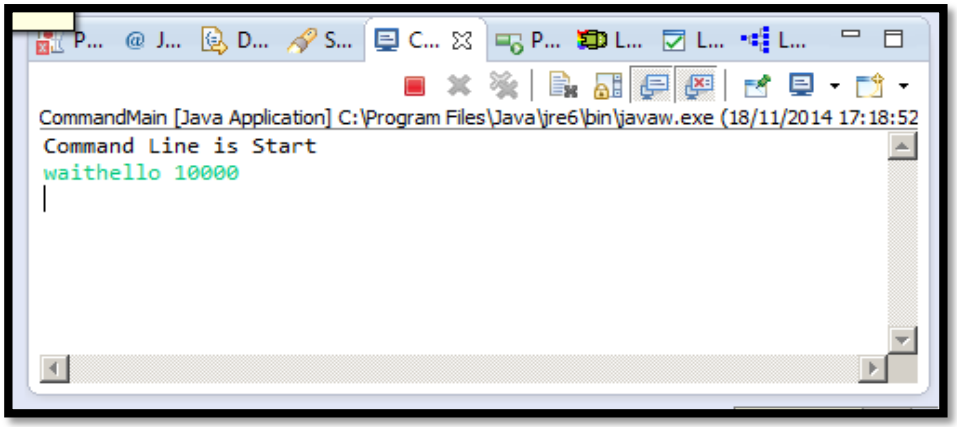


Ilustración 128: El comando waithello es ejecutado con un tiempo de espera de 10 segundos.

Después de esperar 10 segundos tenemos el siguiente resultado:

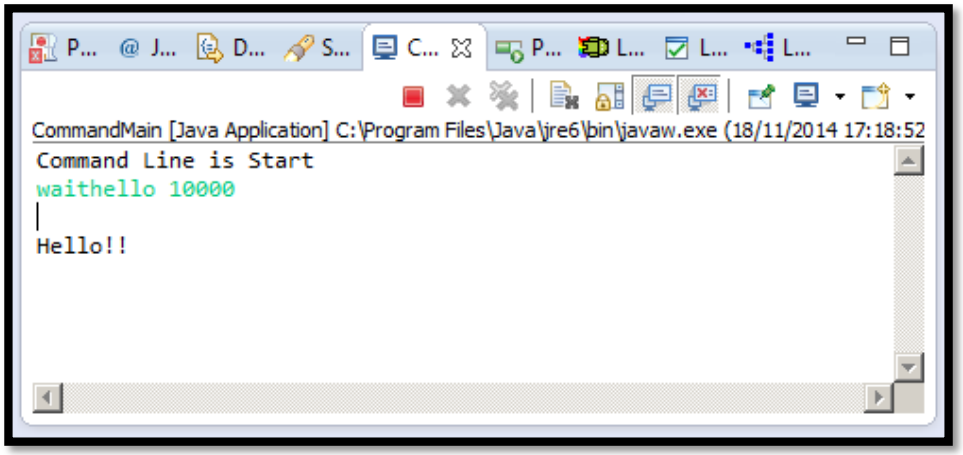


Ilustración 129: Después de esperar 10 segundos el mensaje Hello! es impreso en pantalla.

Ejecución 8: Comando batch.

En el siguiente ejemplo ejecutaremos el comando *batch* que lee un archivo de texto y ejecuta todos los comandos que tenemos en él de forma secuencial, para este ejemplo ejecutaremos el comando y le diremos que lea un archivo en la ruta c:/dummy/batch.txt, el cual tendrá el siguiente contenido:

```
1. date “dd/MM/yyyy hh:mm:ss”
2. dir -n “c:/dummy/dummy2”
```

3. echo “Se creo el archivo dummy2”
4. file -wn “c:/dummy/dummy2/dummy.txt” “Hola mundo con el comando file”
5. memory
6. file -r “c:/dummy/dummy2/dummy.txt” “c:/dummy/dummy2/dummy2.txt”
7. date “dd/MM/yyyy hh:mm:ss”

Lo que esperemos que realice el comando es lo siguiente:
Imprima la fecha y hora (línea 1).
Cree el directorio c: /dummy/dummy2 (línea 2).
Escriba en pantalla: “Se creó el archivo dummy2” (línea 3).
Cree un archivo en “c:/dummy/dummy2/dummy.txt” con el contenido “Hola mundo” con el comando *file* (línea 4).
Imprima en pantalla los datos de la memoria (línea 5).
Renombre el archivo que acabamos de crear como dummy2.txt (línea 6).
Imprima nuevamente la fecha y hora (línea 7).

Ilustración 130: *Resultado de la ejecución del comando bach.*

En la imagen podemos apreciar cómo fueron ejecutados los comandos en el orden que fueron establecidos y los archivos que fueron creados en la ejecución.

Ejecución 9: Comando no encontrado.

En el mundo real, es muy común que nos equivoquemos al escribir un comando o que ejecutemos algunos creyendo que existen, en estos casos la consola nos indica que el comando no existe. Por ejemplo, podemos intentar ejecutar el comando “**Java**”, el cual nos indicará que no fue encontrado ya que no tenemos ningún comando Java registrado en el *CommandManager*.

En estos casos la clase *NotFoundCommand* es la que atiende le petición.

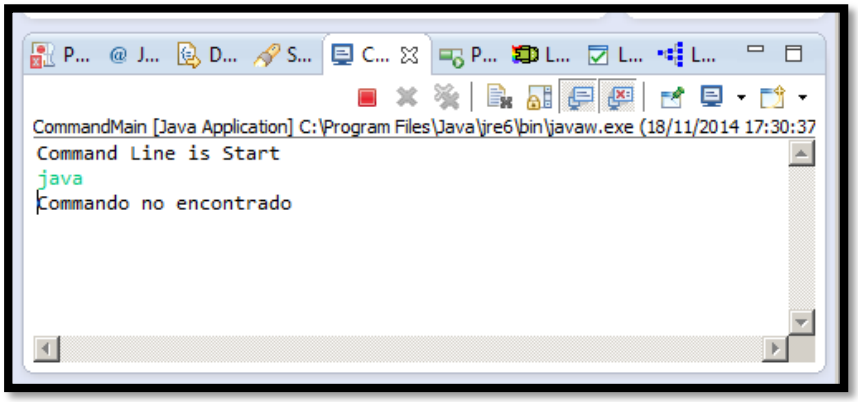


Ilustración 131: Resultado de la ejecución de un comando no encontrado.

Ejecución 10: Comando Exit

Finalmente, ejecutamos el comando *exit* que terminará con la aplicación.

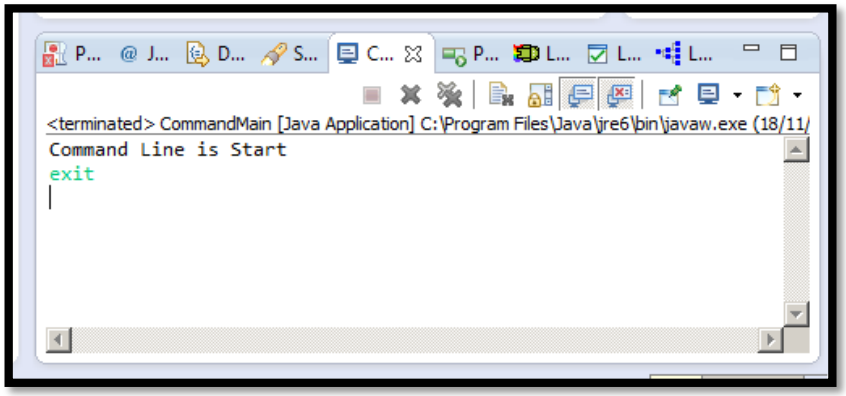


Ilustración 132: Resultado de la ejecución del comando *exit*.

El comando únicamente finaliza la aplicación de una forma rápida, sin embargo, podemos extender la funcionalidad para que no nos permita cerrarla si hay procesos en ejecución en segundo plano o que nos pregunte si realmente queremos cerrarla.

Siguientes pasos:

Hasta este punto ya habremos creado una consola que es capaz de ejecutar comandos, los cuales serán realizados sin saber realmente que clase está efectuando el trabajo. Sin embargo, en la vida real no nos interesa saber cómo es atendido un comando, siempre y cuando realice la tarea que deseamos. Y como ya vimos, con el patrón de diseño *Command* esto es posible.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios:

- Creemos un nuevo comando que nos permita buscar un archivo mediante su nombre en el sistema de archivos. Recibirá como parámetro el *path* desde donde comenzará a buscar y el **nombre** del archivo.

- Creemos un comando que nos permite registrar nuevos comandos de forma dinámica, para lo cual debemos enviar como parámetro el nombre completo de la clase y el nombre con el que se registrará. El comando se encargará de registrar la clase en el *CommandManager* y desde ese momento podrá utilizarse desde la consola.

- Modifiquemos el comando de búsqueda de archivos para que se ejecute de forma asíncrona, para lo cual, debemos implementar la clase *AsyncCommand* e imprimir en pantalla el resultado de la búsqueda una vez que termine.

Patrón Observer

El patrón de diseño Observer permite observar los cambios producidos por un objeto, de esta forma, cada cambio que afecte el estado del objeto observado lanzará una notificación a los observadores; a esto se le conoce como Publicador-Suscriptor. Observer es uno de los principales patrones de diseño utilizados en interfaces gráficas de usuario (GUI), ya que permite desacoplar al componente gráfico de la acción a realizar.

Los objetos observados pueden tener uno o varios observadores, por lo que cuando un cambio ocurra en el objeto, todos los observantes serán notificados del cambio.

El *Observer* está dividido en dos partes: el observable y el observador; el primero es el objeto que deseamos observar y el observador es el objeto que está observando los cambios del Observable.

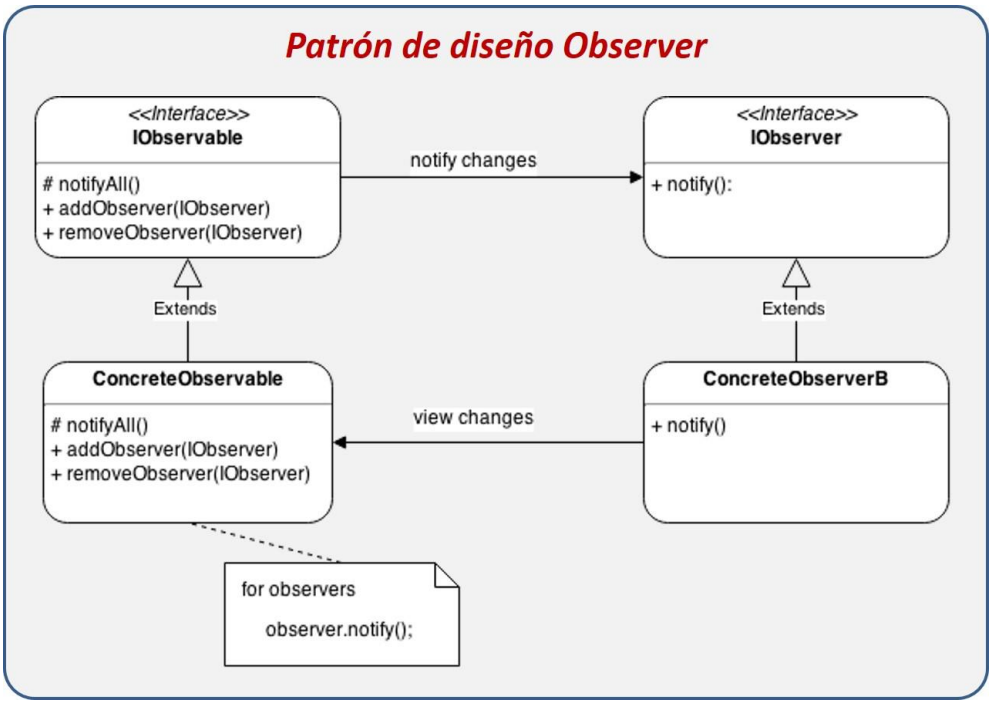


Ilustración 133: Estructura del patrón de diseño Observer.

Los componentes del patrón se explican a continuación:

- **IObservable:** Interface que deben de implementar todos los objetos que quieren ser observados, en ella se definen los métodos mínimos que se deben implementar.
- **ConcreteObservable:** Clase que desea ser observada, ésta implementa IObservable y debe implementar sus métodos.
- **IObserver:** Interfaces que deben implementar todos los objetos que desean observar los cambios de *IObservable*.
- **ConcreteObserver:** Clase concreta que está atenta de los cambios de *IObserver*, esta clase hereda de *IObserver* y debe de implementar sus métodos.

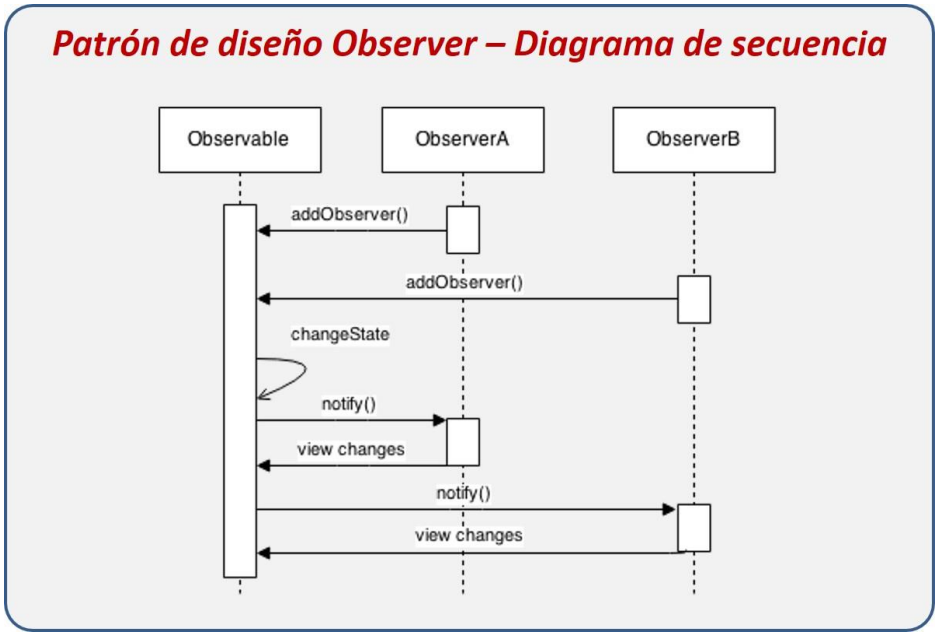


Ilustración 134: Diagrama de secuencia del patrón Observer.

El diagrama anterior se interpreta de la siguiente manera:

1. El ObserverA se registra con el objeto Observable para ser notificado de algún cambio.

2. El ObserverB se registra con el objeto Observable para ser notificado de algún cambio.
3. Ocurre algún cambio en el estado del Observable.
4. Todos los *Observers* son notificados con el cambio ocurrido.

Cuando un cambio ocurre en el *Observable* todos los *Observers* serán notificados, el *Observer* tendrá que validar si la notificación que lanzó el *Target* le son de interés, de no ser así simplemente podrá pasarla por alto, sin embargo, si el cambio ocurrido en el Observable es de interés se podrá realizar una acción.

Cuándo utilizarlo:

- Cuando requieres que una serie de objetos estén enterados sobre los cambios realizados por otro objeto.

- Cuando se requiere desacoplar el objeto analizado de los interesados, permitiendo la variación de ambas partes.

El escenario:

Nos han pedido que desarrollemos una aplicación de escritorio, la cual al iniciar cargue la configuración de Internacionalización (I18N). Entre las configuraciones tenemos el formato de Fecha y Moneda, éstas serán necesarias en el programa para mostrar la fecha y la moneda en un formato configurado por el usuario según su localización geográfica, sin embargo, el usuario puede cambiar esta configuración en cualquier momento; lo que implica que en todas las partes donde se esté utilizando un formato de fecha o moneda deberán ser notificados del cambio para aplicar el formato correspondiente.

Uno de los principales problemas es que cuando alguna configuración cambie, todos los interesados deberán ser notificados de inmediato por lo que se deberá implementar un mecanismo que nos permita hacer esto.

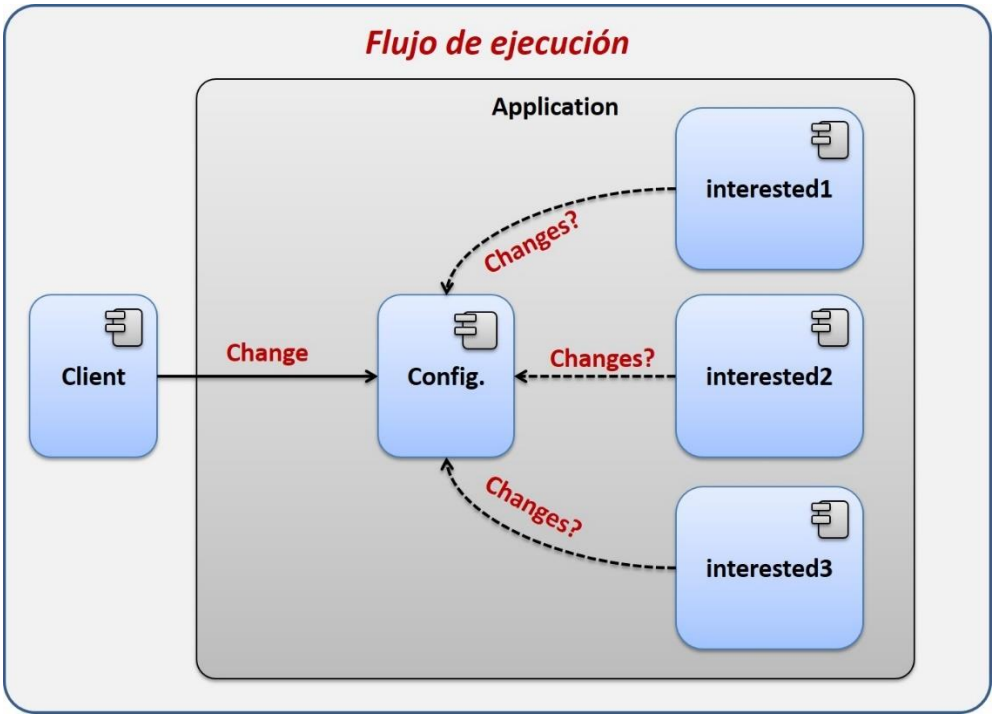


Ilustración 135: Cambio en la configuración sin el patrón Observer.

La imagen anterior demuestra cómo es que normalmente estaríamos observando si se realizó algún cambio en la configuración. Lo primero que vemos es que los interesados son los que tienen que preguntar a la configuración si algún cambio ocurrió, esto provoca tres problemas:

1. No sabremos de inmediato que algún cambio se realizó en la configuración hasta que vayamos a preguntarle.
2. Los interesados están totalmente acoplados a la configuración por lo que tendrán que tener una referencia directa a éste.
3. Tenemos que programar algún mecanismo que le esté preguntando a la configuración si se realizó algún cambio, esto puede ser algún *pooling* o monitoreo constante, lo cual puede resultar complicado de programar.

Por esta razón tenemos que implementar un mecanismo que nos permita que la configuración notifique a los interesados en el momento en que algún cambio ocurra.

La solución:

Para darle solución a este problema se propone implementar el patrón de diseño *Observer* el cual nos permitirá notificar a los interesados si algún cambio ocurrió en la configuración en vez de que los interesados estén preguntando si algún cambio ocurrió. Los Observers deberán implementar una interfaz en común que permita a la configuración notificar a todos los observadores registrados.

La configuración deberá implementar una interfaz que indique que es un Objeto observable, de esta manera los interesados lo sabrán y se podrán suscribir o dar de baja a las notificaciones, así, cuando un Objeto deje de estar interesado en los cambio del Observable, simplemente dejará de recibir notificaciones.

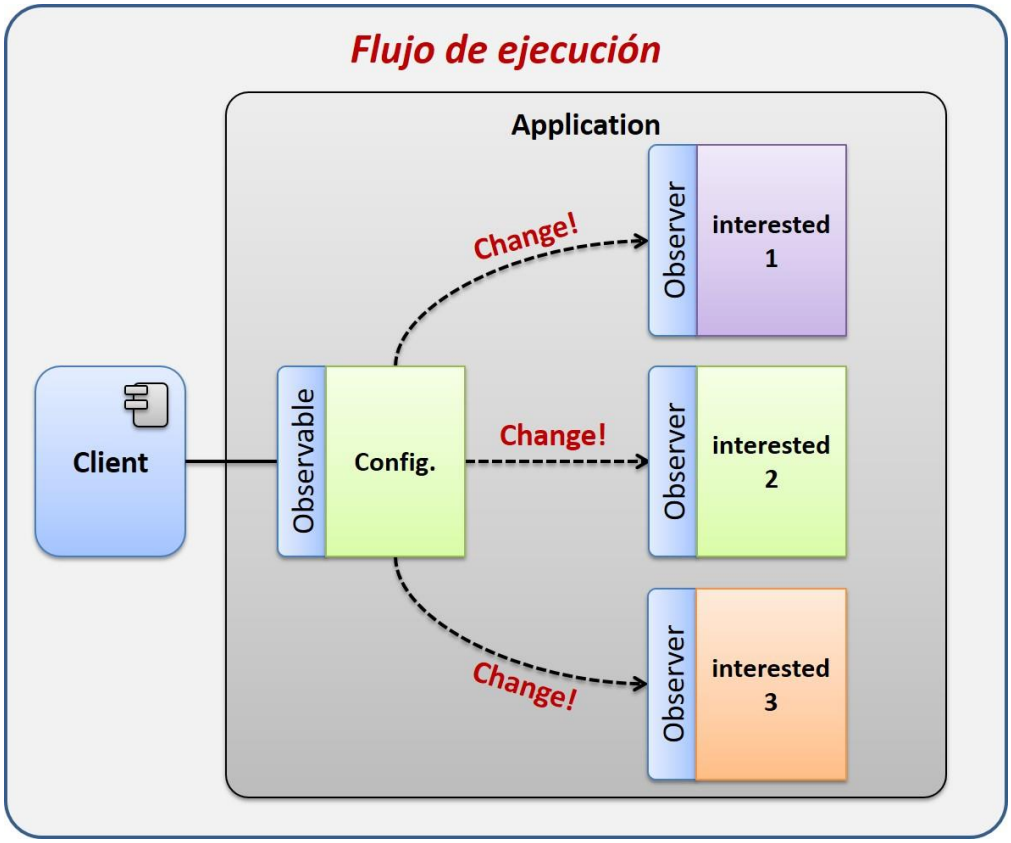


Ilustración 136: Notificación de cambios implementando el patrón Observer.

En la imagen podemos ver algunas diferencias significativas.

La primera es que las fechas de los cambios van de la configuración a los interesados, lo que quiere decir que será la configuración la encargada de notificarle que ocurrió un cambio para que apliquen los nuevos formatos en todas las pantallas que el usuario tiene a la vista.

La segunda es que los interesados implementan la interfaz *Observer* y la configuración de la interfaz Observable.

La tercera es que hemos desacoplado la configuración de los *Observer*, desde luego que la deberán de utilizar pero ya no tendrán una referencia directa con ella, en su lugar, cuando un evento ocurra se les mandará la configuración como parámetro.

La implementación:

Ya hemos analizado el problema y hemos visto cómo resolverlo por lo que hora pasaremos a la implementación.

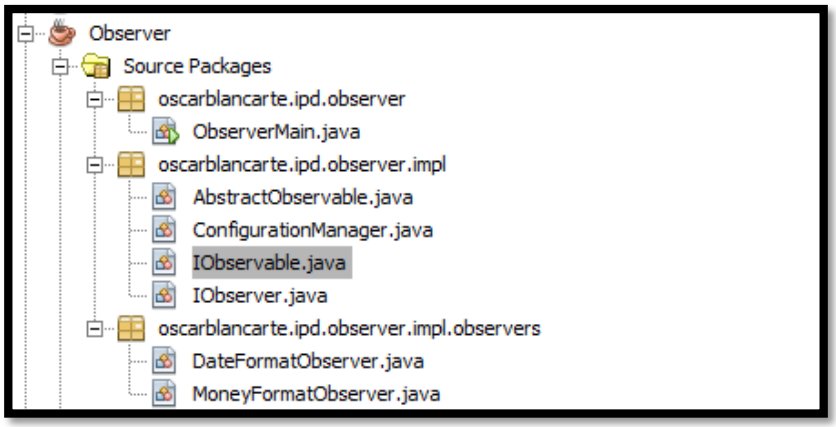


Ilustración 137: Estructura del proyecto Observer.

El proyecto se compone de los siguientes paquetes:

- **oscarblancarte.ipd.observer:** Paquete por *default* del proyecto en el cual se tiene la clase ejecutable del proyecto.
- **oscarblancarte.ipd.observer.impl:** Paquete en donde se definen las clases e interfaces del patrón de diseño Observer.
- **oscarblancarte.ipd.observer.impl.observers:** Paquete en el que se encuentran todas las clases que se utilizarán de *Observer*.

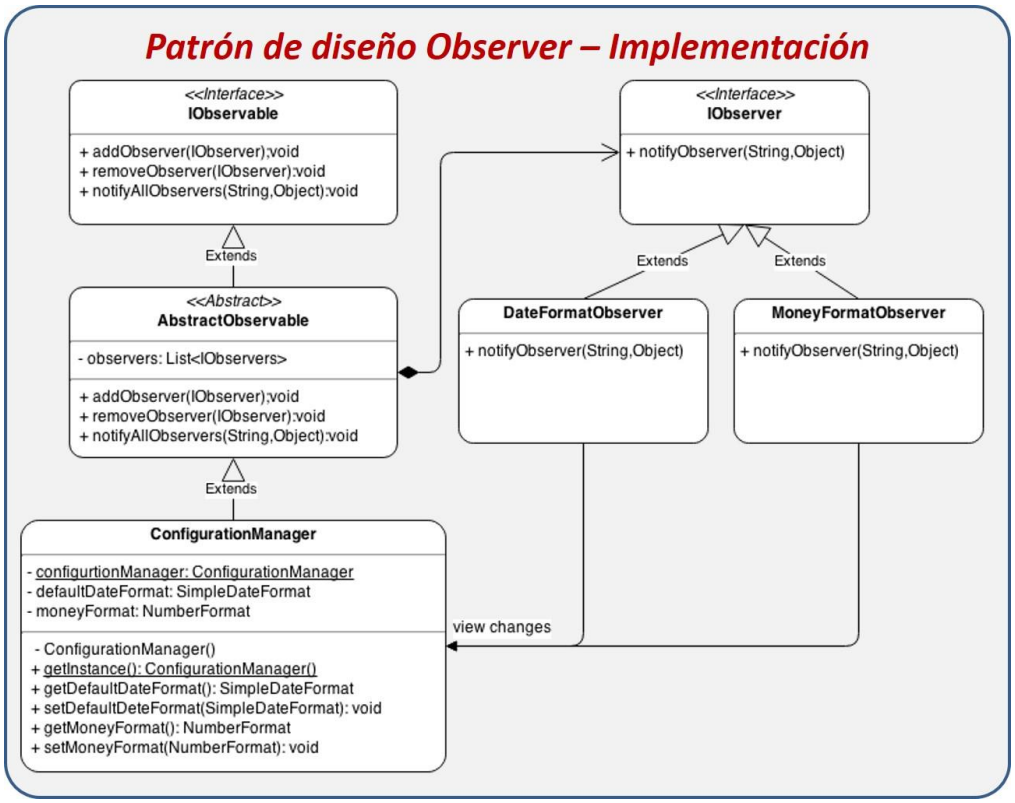


Ilustración 138: Diagrama de clases del proyecto Observer.

Interface IObservable:

Esta es la interface que todos los objetos que desean ser observados deberán implementar.

```
1. package oscarblancarte.ipd.observer.impl;
2.
3. public interface IObservable {
4.
5.     public void addObserver(IObserver observer);
6.     public void removeObserver(IObserver observer);
7.     public void notifyAllObservers(String command, Object source);
8. }
```

La clase define los siguientes métodos:

- **addObserver:** Permite registrar un *Observer* con el fin de ser notificado cuando algún cambio suceda.
- **removeObserver:** Permite a un *Observer* darse de baja, con lo que ya no será notificado cuando algún cambio suceda.
- **notifyAllObservers:** Método desde el cual se notifica un cambio a los *Observers*. Cuando un cambio ocurra en la configuración este método será ejecutado, notificando a todos los *observer* registrados.

Clase AbstractObservable:

Clase abstracta que utilizaremos como base para crear nuevos objetos Observables, la clase hereda de *IObservable* por lo que implementamos sus métodos.

```
1.  package oscarblancarte.ipd.observer.impl;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public abstract class AbstractObservable implements IObservable {
7.
8.      private final List<IObserver> observers = new ArrayList<>();
9.
10.         @Override
11.         public void addObserver(IObserver observer) {
12.             this.observers.add(observer);
13.         }
14.
15.         @Override
16.         public void removeObserver(IObserver observer) {
17.             this.observers.remove(observer);
18.         }
19.
20.         @Override
21.         public void notifyAllObservers(String command, Object source) {
22.             for (IObserver observer : observers) {
23.                 observer.notifyObserver(command, source);
24.             }
25.         }
26.     }
```

Podemos apreciar en la línea 8 que tenemos una lista de *IObserver*, éstos serán los *Observer* que serán notificados cuando un cambio ocurra.

Se implementan los métodos *addObserver* y *removeObserver* con el fin de registrar y dar de baja a los *Observers*.

El método *notifyAllObservers* notificará a todos los *Observer* que se encuentren en la lista *observers*.

Clase ConfigurationManager:

La clase *ConfigurationManager* es la clase que queremos observar debido a que en ella se encuentra la configuración de fecha y moneda. La clase hereda de *AbstractObservable* lo que le permite registrar y dar de baja *Observers*.

```
1.  package oscarblancarte.ipd.observer.impl;
2.
3.  import java.text.NumberFormat;
4.  import java.text.SimpleDateFormat;
5.
6.  public class ConfigurationManager extends AbstractObservable {
7.
8.      private SimpleDateFormat defaultDateFormat;
9.      private NumberFormat moneyFormat;
10.
11.         private static ConfigurationManager configurationManager;
12.
13.         private ConfigurationManager() {
14.         }
15.
16.         public static ConfigurationManager getInstance() {
17.             if (configurationManager == null) {
18.                 configurationManager = new ConfigurationManager();
19.             }
20.             return configurationManager;
21.         }
22.
23.         public SimpleDateFormat getDefaultDateFormat() {
24.             return defaultDateFormat;
25.         }
26.
27.         public void setDefaultDateFormat(SimpleDateFormat defaultDateFormat) {
28.             this.defaultDateFormat = defaultDateFormat;
29.             notifyAllObservers("defaultDateFormat", this);
```

```
30.         }
31.
32.         public NumberFormat getMoneyFormat() {
33.             return moneyFormat;
34.         }
35.
36.         public void setMoneyFormat(NumberFormat moneyFormat) {
37.             this.moneyFormat = moneyFormat;
38.             notifyAllObservers("moneyFormat", this);
39.         }
40.     }
```

Lo primero que podemos observar es que la clase *ConfigurationManager* implementa el patrón *Singleton*, por lo que sólo podremos obtener una instancia de la clase. La instancia se obtiene por medio del método *getInstance*.

Analicemos también los métodos *setDefaultDateFormat* y *setMoneyFormat*, estos dos métodos son bastante parecidos a cualquier método *set* tradicional, sin embargo, éstos lanzan una notificación por medio del método *notifyAllObservers* heredado de *AbstractObserver*.

Interface IObserver:

Interface que deberán implementar todos los *observer* con el fin de ser notificados cuando algún cambio ocurra.

```
1. package oscarblancarte.ipd.observer.impl;
2.
3. public interface IObserver {
4.
5.     public void notifyObserver(String command, Object source);
6. }
```

El método *notifyObserver* será llamado cuando algún cambio ocurra en la configuración, el parámetro *command* indica la propiedad modificada y el *source* representa el objeto *ConfigurationManager*.

Clase DateFormatObserver:

Clase *Observer* la cual hereda de *IObserve*, esta clase estará atenta únicamente de los cambios en el formato de fecha. Cuando un cambio en el formato de fecha ocurra en la configuración, el método *notifyObserver* será llamado.

```
1.  package oscarblancarte.ipd.observer.impl.observers;
2.
3.  import java.util.Date;
4.  import oscarblancarte.ipd.observer.impl.ConfigurationManager;
5.  import oscarblancarte.ipd.observer.impl.IObserver;
6.
7.  public class DateFormatObserver implements IObserver{
8.
9.      @Override
10.         public void notifyObserver(String command, Object source) {
11.             if(command.equals("defaultDateFormat")){
12.                 ConfigurationManager conf = (ConfigurationManager)source;
13.                 System.out.println("Observer ==> DateFormatObserver.dateFormatChange > "
14.                         + conf.getDefaultDateFormat().format(new Date()));
15.             }
16.         }
17.     }
```

En la línea 11 vemos que se valida si la propiedad afectada es *defaultDateFormat*, en caso de ser así obtenemos la referencia de la configuración (línea 16) e imprimimos la fecha con el nuevo formato.

Clase MoneyFormatObserver:

Clase *Observer* la cual hereda de *IObserve*, esta clase estará atenta únicamente de los cambios en el formato de la moneda. Cuando un cambio en el formato de la moneda ocurra en la configuración, el método *notifyObserver* será llamado.

```
1.  package oscarblancarte.ipd.observer.impl.observers;
2.
3.  import oscarblancarte.ipd.observer.impl.ConfigurationManager;
4.  import oscarblancarte.ipd.observer.impl.IObserver;
5.
6.  public class MoneyFormatObserver implements IObserver{
```

```
7.    @Override
8.    public void notifyObserver(String command, Object source) {
9.       if(command.equals("moneyFormat")){
10.              ConfigurationManager conf = (ConfigurationManager)source;
11.              System.out.println("Observer ==> MoneyFormatObserver.moneyFormatChange > "
12.                     + conf.getMoneyFormat().format(1.11));
13.           }
14.        }
15.     }
```

En la línea 9 vemos que se valida si la propiedad afectada es *moneyFormat*, en caso de ser así, obtenemos la referencia de la configuración (línea 10) e imprimimos un ejemplo del nuevo formato de moneda.

Clase ObserverMain:

Clase principal del proyecto la cual utilizaremos para comprobar la funcionalidad del patrón.

```
1.  package oscarblancarte.ipd.observer;
2.
3.  import java.text.DecimalFormat;
4.  import java.text.SimpleDateFormat;
5.  import oscarblancarte.ipd.observer.impl.ConfigurationManager;
6.  import oscarblancarte.ipd.observer.impl.observers.DateFormatObserver;
7.  import oscarblancarte.ipd.observer.impl.observers.MoneyFormatObserver;
8.
9.  public class ObserverMain {
10.
11.         public static void main(String[] args) {
12.            ConfigurationManager conf = ConfigurationManager.getInstance();
13.
14.            //Se establecen los valores por default.
15.            conf.setDefaultDateFormat(new SimpleDateFormat("yyyy/MM/dd"));
16.            conf.setMoneyFormat(new DecimalFormat("##.00"));
17.            System.out.println("Configuración establecida");
18.
19.            //Se dan de alta lo observers
20.            DateFormatObserver dateFormatObserver = new DateFormatObserver();
21.            MoneyFormatObserver moneyFormatObserver = new MoneyFormatObserver();
22.            conf.addObserver(dateFormatObserver);
23.            conf.addObserver(moneyFormatObserver);
24.            System.out.println("");
25.
26.            //Se cambia la configuración
27.            conf.setDefaultDateFormat(new SimpleDateFormat("dd/MM/yyyy"));
28.            conf.setMoneyFormat(new DecimalFormat("###,#00.00"));
```

```
29.         System.out.println("");
30.
31.         //Se realiza otro cambio en la configuración.
32.         conf.setDefaultDateFormat(new SimpleDateFormat("MM/yyyy/dd"));
33.         conf.setMoneyFormat(new DecimalFormat("###,#00"));
34.
35.         conf.removeObserver(dateFormatObserver);
36.         conf.removeObserver(moneyFormatObserver);
37.         System.out.println("");
38.
39.         //Se realiza otro cambio en la configuración.
40.         conf.setDefaultDateFormat(new SimpleDateFormat("MM/yyyy"));
41.         conf.setMoneyFormat(new DecimalFormat("###,##0.00"));
42.     }
43. }
```

Puntos destacados del programa:

- En la línea 12 obtenemos el objeto de configuración.
- De las líneas 15 a la 17 establecemos los formatos de fecha y moneda en la configuración.
- De las líneas 20 a 23 creamos los *observer* y los registramos en la configuración.
- De las líneas 27 a 29 modificamos los formatos de fecha y moneda.
- En las líneas 32 y 33 volvemos a modificar los formatos de fecha y moneda.
- En las líneas 35 y 36 damos de baja los *observer* de la configuración con el fin de ya no recibir notificaciones en caso de que algún cambio ocurra.
- Finalmente, en las líneas 40 y 41 volvemos hacer un cambio en la configuración de la fecha y moneda.
- Cuando ejecutemos la aplicación pongamos atención en que sólo se notificarán los cambios de la fecha y moneda en el lapso que tenemos los observer registrados y una vez que los demos de baja ya no serán notificados.

NOTA: Cuando ejecutemos el programa veremos que cada vez que el formato de fecha y hora cambia, éstos son impresos en pantalla por los Observadores, los cuales los imprimirán como una acción después de ser notificados de un cambio en el *Observer*.

La ejecución:

Ejecutaremos la clase *ObserverMain* para ver el siguiente resultado:

```
run:
Configuration establecida

Observer ==> DateFormatObserver.dateFormatChange > 15/03/2015
Observer ==> MoneyFormatObserver.moneyFormatChange > 01.11

Observer ==> DateFormatObserver.dateFormatChange > 03/2015/15
Observer ==> MoneyFormatObserver.moneyFormatChange > 01

BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 139: Resultado de la ejecución de ObserverMain.

En la primera línea podemos observar que la configuración es creada y los formatos de moneda y fecha son establecidos. En este punto no vemos que los observadores hayan sido notificados debido a que no están registrados en la configuración.

En la siguiente sección vemos que los observadores ya fueron notificados de un segundo cambio en la configuración.

Seguido, un nuevo cambio en la configuración es realizado disparando la notificación de los observadores.

Finalmente, los observadores son retirados del registro del *observer* para después cambiar nuevamente los formatos de fecha y moneda.
Al ya no estar registrados no se ve ningún cambio en la pantalla.

Siguientes pasos:

Hemos visto que *Observer* es un patrón muy bueno cuando requerimos estar atento a lo que sucede en otros objetos. El patrón *observer* nos proporciona un mecanismo reactivo ante los cambios, evitando un fuerte acoplamiento y un desgaste en el performance de la aplicación, al tener que estar monitoreando constantemente el estado de un objeto.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios:

- Creemos una nueva configuración en la clase *ConfigurationManager* la cual nos permita administrar configurar el formato de moneda, es decir, el usuario podrá indicar que su moneda será dólar ($) o euro (€). Además tendremos que crear un nuevo observador que nos permita ser notificados de que el formato ha cambiado.

- Creemos un nuevo observador que procese cualquier cambio en el estado de la configuración, en lugar de filtrar los eventos, tendrá que procesar todos por igual.

- Creemos una nueva interface *ConfigurationObserver* que en vez de tener un sólo método de notificación, deberemos tener un método para cada tipo de configuración, los cuales serán ejecutados cuando esa configuración en específico cambie. Los métodos que debemos de tener son:
    - dateFormatChange
    - moneyFormatChange
    - currencyFormatChange

Patrón Templete Method

El patrón de diseño *templete* centra su funcionalidad en la reutilización de código y se utiliza para implementar algoritmos que realizan los mismos pasos para llegar a una solución. Esto se logra implementando clases bases que definan un comportamiento predeterminado. Usualmente es creado un método para cada paso del algoritmo a implementar, de los cuales algunos serán implementados y otros permanecerán abstractos hasta su ejecución por parte de las subclases.

El patrón *templete* define además, un método público que permita a los usuarios del patrón ejecutar el *templete*, este método se encargará de orquestar la ejecución ordenada de los métodos o pasos del algoritmo.

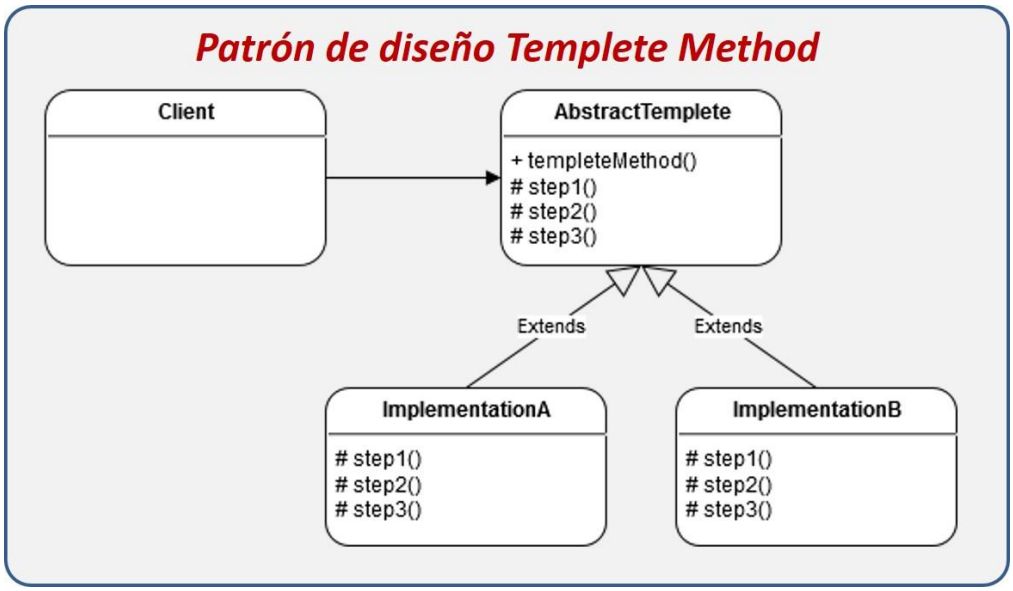


Ilustración 140: Estructura del patrón de diseño Templete Method.

Los componentes del patrón Templete Method se explica a continuación:

- **Client:** Es el componente que acciona la ejecución del templete.
- **AbstractTemplete:** Clase abstracta con una serie de operaciones que definen los pasos para llevar a cabo la ejecución del algoritmo. La clase

tiene el método *templeteMethod* que ejecuta en orden los métodos step1, step2, step3.

- **Implementation:** Clase que representa un temple concreto, para lo cual deberá de heredar de *AbstractTemplete* e implementar los métodos de ésta.

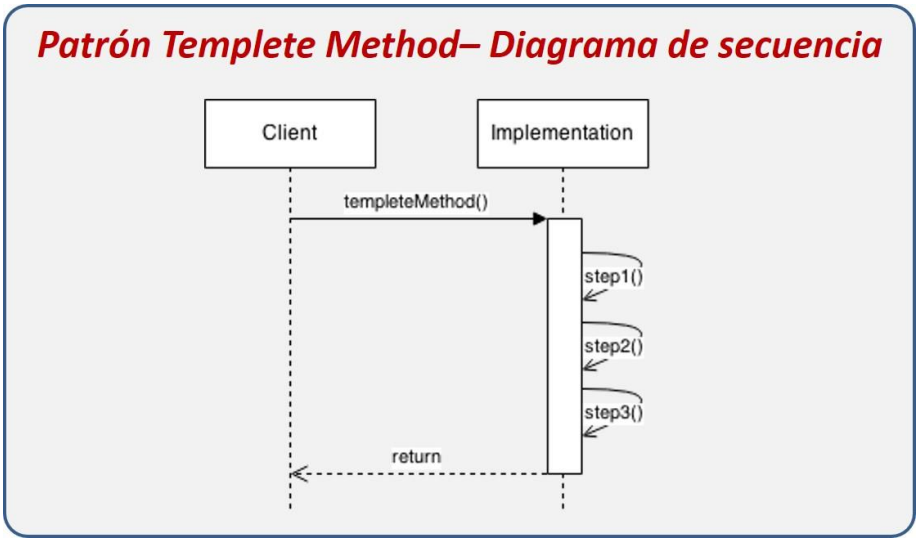


Ilustración 141: Diagrama de secuencia del patrón Templete Method.

El diagrama se interpreta de la siguiente manera:

1. El cliente crea u obtiene una instancia de una implementación del *templete*.
2. El cliente ejecuta el método público *templeteMethod* del *templete*.
3. La implementación por *default* del método *templeteMehod* ejecuta en orden los métodos step1, step2, step3.
4. La implementación retorna un resultado.

Como podemos ver, el patrón *templete* ayuda a la reutilización del código en operaciones que tienen pasos similares, además de que controla el orden en que se ejecutan los pasos impidiendo que el cliente afecte el orden de ejecución.

Cuándo utilizarlo:

- Cuando se ha identificado que una serie de objetos comparten un comportamiento similiar y que además requieren de los mismos pasos para completar su ejecución.
- Cuando es posible reutilizar código mediante la creación de plantillas que implementen gran parte de la funcionalidad desde una clase base.

El escenario:

Si alguno de nosotros ha trabajado alguna vez para empresas de venta de servicios, verá que es muy común que las empresas busquen establecer convenios con otras empresas (como podrían ser farmacias o tiendas de abarrotes) con el fin de que los clientes puedan pagar sus servicios a través de sus sucursales. Un ejemplo muy común es el pago del recibo del agua, luz eléctrica, gas, teléfono, internet, televisión de paga, entre otros. Todas estas compañías dan la opción a sus clientes de pagar estos servicios en las tiendas de su preferencia.

Estos establecimientos cada día tienen que enviarnos el concentrado con todos los pagos que recibieron por parte de nuestros clientes, de tal manera que nosotros podamos aplicar estos pagos en nuestros sistemas. Estos concentrados por lo general son archivos planos de texto donde se concentra básicamente el número de cliente, monto pagado, forma de pago y la fecha en que se realizó el pago.

Pues bien, nuestro cliente nos ha solicitado que implementemos un componente de software que tome estos archivos y aplique los pagos en su sistema. Hasta aquí todo suena bien, sin embargo, tenemos un pequeño problema, cada establecimiento nos entrega la información en un formato distinto.

Los requerimientos para desarrollar este componente son los siguientes:

- Los archivos pueden llegar en diferente formato por establecimiento.
- El formato cambia pero la información minina necesaria siempre viene en el documento.
- Los archivos no se pueden procesar más de una vez.
- El nombre del documento es validado por establecimiento y cada establecimiento puede tener diferente formato.
- Para aplicar el pago se debe validar que el cliente exista en nuestro sistema.
- Una vez procesado el documento se debe generar un archivo de log donde indique los pagos aplicados y los rechazados con su respectiva

explicación del motivo del rechazo, además, el archivo procesado deberá ser movido a una carpeta de archivos procesados.

- Los pagos recibidos por los establecimientos no deberán ser mayores de $200.

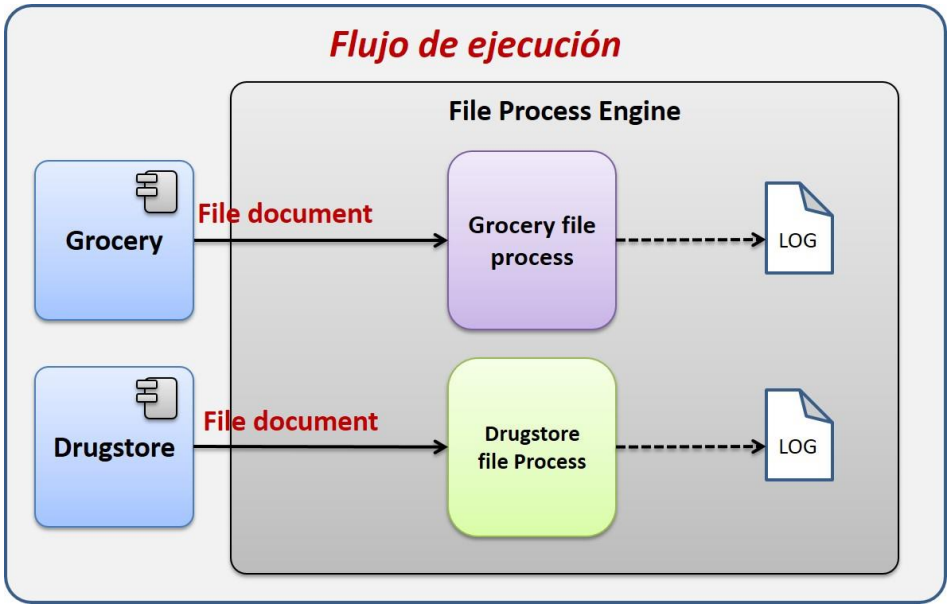


Ilustración 142: Recepción y procesamiento tradiconal de archivos.

En la imagen podemos ver una implementación tradicional de procesamiento de archivos, donde será necesario desarrollar un componente por cada archivo distinto que tengamos que procesar.

Esta solución no es incorrecta pero tenemos el gran problema de la reutilización de código, damos carta abierta a que cada implementación realice los pasos que crea necesarios y en el orden que quiera para procesar el documento. Esto provoca el problema del mantenimiento y la depuración ya que cada implementación trabajaría de forma muy distinta para llegar al mismo resultado.

La solución:

Para darle una mejor solución implementaremos el patrón de diseño *Templete Method*, que nos permitirá establecer los pasos por lo que el procesamiento de archivos deberá pasar sin importar el establecimiento que envíe el documento.

Para lograr esto deberemos definir los pasos requeridos para poder procesar cualquier archivo de pagos:

1. **validateName:** Primer paso, valida el nombre del documento.
2. **validateProcess:** Segundo paso, valida que el documento no haya sido procesado anteriormente.
3. **processFile:** Tercer paso, procesará el documento, analizando cada pago (línea), aplicará el pago y localizará los errores.
4. **createLog:** Cuarto paso, generar un *Log* con los pagos procesados exitosamente y los errores encontrados.
5. **moveDocument:** Quinto paso, moverá el documento a una carpeta de documentos procesados.
6. **markAsProcessFile:** Sexto paso, registrará el documento como procesado impidiendo que un mismo archivo sea procesado más de una vez.

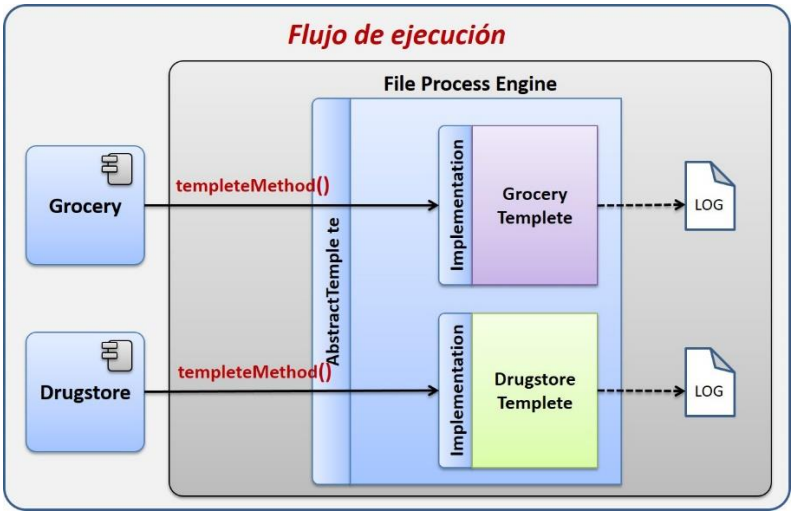


Ilustración 143: Procesamiento de archivos implementando el patrón Templete Method.

En la imagen podemos ver cómo quedaría el componente de procesamiento de archivos implementado con el patrón *Templete Method*. Veamos que es muy parecida a la imagen expuesta en el escenario, sin embargo, hay una gran diferencia y es que los componentes de procesamiento de archivos no son implementados libremente, en su lugar la clase *AbstractTemplete* rige la forma en que éstos deben ser implementados y además controla la ejecución de los componentes.
Cuando el cliente desea procesar el documento sólo necesita mandar llamar al método *templeteMethod*, el cual se encargará de ejecutar los pasos expuestos anteriormente en el orden correcto para obtener el resultado esperado.

Este escenario plantea una gran ventaja y es que para dar mantenimiento o depuración a la aplicación ya sabemos exactamente los pasos del proceso y qué hacer en cada paso.

La implementación:

Una vez que hemos visto lo que tenemos que hacer, pasaremos a la implementación del programa.

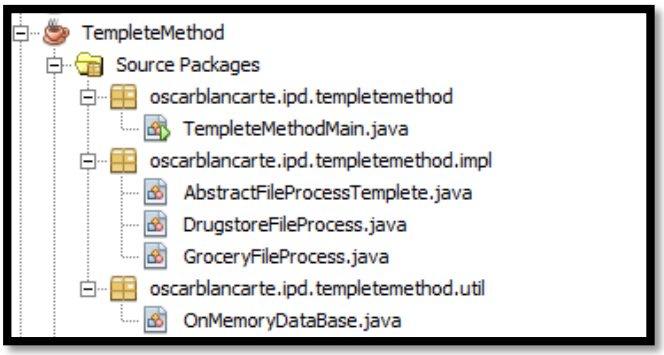


Ilustración 144: Estructura del proyecto Templete Method.

Lo paquetes que componen la aplicación se explican a continuación:

- **oscarblancarte.ipd.templemethod:** Paquete principal de la aplicación que únicamente contiene la clase principal *TempleteMethodMain*.
- **oscarblancarte.ipd.templemethod.impl:** Paquete en el cual está la definición del *templete* y de las implementaciones que servirán para procesar los distintos archivos de pagos.
- **oscarblancarte.ipd.templemethod.util:** Paquete con clases de utilidad que complementan alguna funcionalidad pero no son parte esencial del patrón.

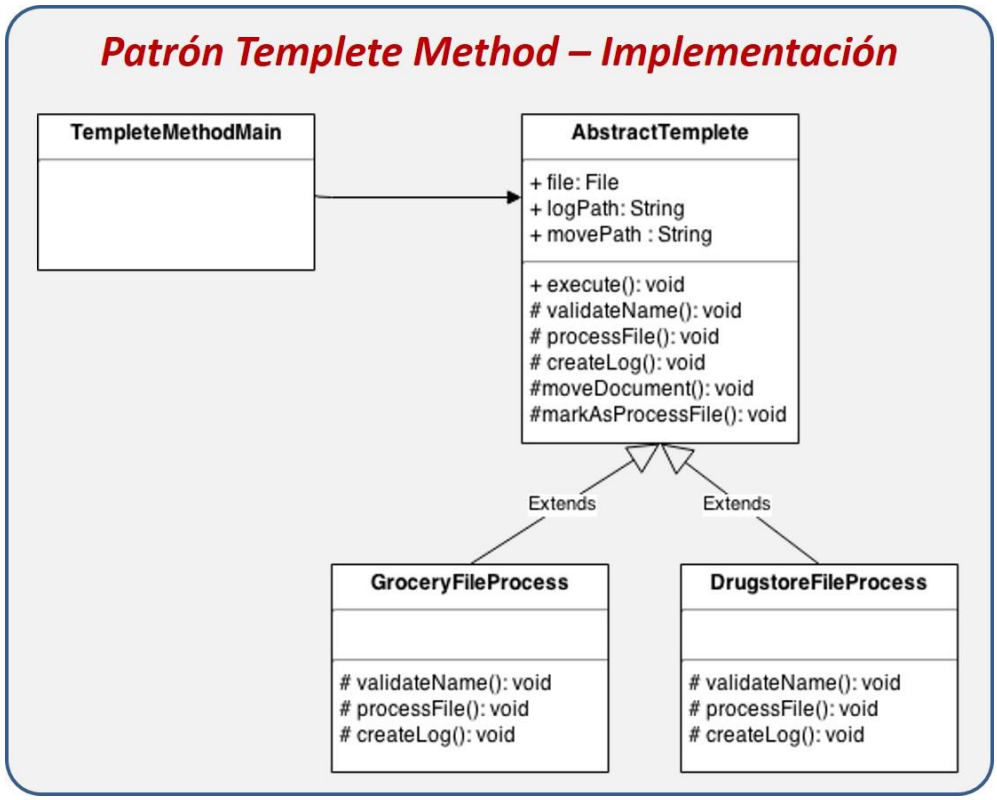


Ilustración 145: Diagrama de clases del proyecto Templete Method.

En la imagen podemos ver las principales clases del proyecto *Templete Method* y cómo es que éstas se relacionan entre sí.

Clase AbstractFileProcessTemplete:

Esta clase corresponde al *templete* abstracto. En esta clase se define los pasos por los que un *templete* debe de pasar y los métodos que deberá implementar para su funcionamiento.

```
1. package oscarblancarte.ipd.templetemethod.impl;
2.
3. import java.io.File;
4. import oscarblancarte.ipd.templetemethod.util.OnMemoryDataBase;
5.
6. public abstract class AbstractFileProcessTemplete {
7.
```

```
8.    protected File file;
9.    protected String logPath;
10.       protected String movePath;
11.
12.       public AbstractFileProcessTemplete(File file, String logPath, String movePath) {
13.          this.file = file;
14.          this.logPath = logPath;
15.          this.movePath = movePath;
16.       }
17.
18.       public final void execute() throws Exception {
19.          validateName();
20.          validateProcess();
21.          processFile();
22.          createLog();
23.          moveDocument();
24.          markAsProcessFile();
25.       }
26.
27.       protected abstract void validateName() throws Exception;
28.
29.       protected void validateProcess() throws Exception {
30.          String fileStatus = OnMemoryDataBase.getFileStatus(file.getName());
31.          if (fileStatus != null && fileStatus.equals("Procesado")) {
32.             throw new Exception("El archivo '" + file.getName() + "' ya fue procesado");
33.          }
34.       }
35.
36.       protected abstract void processFile() throws Exception;
37.
38.       protected abstract void createLog() throws Exception;
39.
40.       private void moveDocument() throws Exception {
41.          String newPath = movePath+"/"+file.getName();
42.          boolean change = file.renameTo(new File(newPath));
43.       }
44.
45.       protected void markAsProcessFile() throws Exception{
46.          OnMemoryDataBase.setProcessFile(file.getName());
47.       }
48.    }
```

Primero que nada, veamos que la clase definen tres propiedades *file*, logPath y movePath, las cuales serán utilizadas para conocer la ruta del archivo a procesar, la url en donde dejaremos el archivo de *log* y la ruta a la que moveremos el archivo una vez que sea procesado.

Estos atributos serán establecidos en el constructor de la clase y deberán ser proporcionados por el cliente al momento de crear un *templete*.

Los métodos de la clase son los siguientes:

- **execute:** Este es el método que el cliente tendrá que ejecutar para procesar los archivos, se encarga de ejecutar el resto de los métodos del *templete* en el orden correcto. Veamos que este método es el único **público** de la clase y que además, es marcado como *final* para evitar que una subclase altere el comportamiento del *templete*.
- **validateName:** Método abstracto el cual deberá ser implementado por las subclases para validar que el nombre del archivo tenga el formato correcto para el establecimiento.
- **validateProcess:** Método que valida si el archivo no ha sido procesado anteriormente. Como podemos determinar de una forma general si un archivo ya fue procesado, podemos implementar este método desde aquí y reutilizarlo en todas las implementaciones, aunque al no ser marcado como final nos da la posibilidad de sobrescribirlo para implementar una estrategia distinta de validación para algún archivo en específico.
- **processFile:** Método encargado de procesar los archivos y aplicar los pagos. Debido a que cada establecimiento envía los archivos en formato distinto, dejaremos este método como abstract con la finalidad de que sea implementado por la subclase de forma adecuada.
- **createLog:** Método encargado de crear un *log* con los movimientos aplicados y rechazados. Quedará como abstracto para que sea definido por las subclases.
- **moveDocument:** Método encargado de mover el documento de directorio una vez termine de procesarlo (esta operación puede ser idéntica para todas las subclases). Debido a que el procedimiento de mover un archivo es idéntico para todas lo archivos, hemos decidido implementar el método desde aquí para reutilizar la funcionalidad.
- **markAsProcesFile:** Método que se encarga de marcar el documento como procesado una vez que termina el procesamiento del archivo. Este método lo hemos definido desde aquí para reutilizar la funcionalidad en las subclases, sin embargo, puede ser sobrescrito por las subclases para personalizar el funcionamiento.

Archivo de la Farmacia:

Éste será el documento que nos envió la farmacia. En él estarán todos los pagos realizados por los clientes. El nombre del archivo es '*10032015123.drug*'

```
1. 0010110010032015
2. 0020205012032015
3. 0030309013032015
4. 0041015013032015
5. 0051120014032015
6. 0062012514032015
7. 0072501014032015
8. 0083011114032015
9. 0094032115032015
10. 0105024116032015
```

Este documento es un archivo plano y está dividido de la siguiente manera:

- **ID del pago:** Los primeros tres caracteres.
- **Número del cliente:** Los siguientes dos caracteres.
- **Los Monto pagado:** Siguientes tres caracteres.
- **Fecha:** Los siguientes ocho caracteres en formato ddmmyyyy.

Este documento será procesado por el templete *DrugstoreFileProcess* del cual hablaremos a continuación.

Clase DrugstoreFileProcess:

Clase que hereda de *AbstractFileProcessTemplete* y tiene la finalidad de procesar los documentos provenientes de la farmacia. Veamos que esta clase no tiene ningún método público, por lo que no podrá ser ejecutado directamente, tendrá que ser ejecutado por medio del método *ejecute* de la súper clase.

```
1. package oscarblancarte.ipd.templetemethod.impl;
2.
3. import java.io.File;
4. import java.io.FileInputStream;
5. import java.io.FileOutputStream;
6. import oscarblancarte.ipd.templetemethod.util.OnMemoryDataBase;
```

```
7.
8.  public class DrugstoreFileProcess extends AbstractFileProcessTemplete {
9.
10.         private String log = "";
11.
12.         public DrugstoreFileProcess(File file, String logPath, String movePath) {
13.             super(file, logPath, movePath);
14.         }
15.
16.         @Override
17.         protected void validateName() throws Exception {
18.             String fileName = file.getName();
19.             if (!fileName.endsWith(".drug")) {
20.                 throw new Exception("Nombre el archivo no válido"
21.                         + ", debe terminar con .drug");
22.             }
23.
24.             if (fileName.length() != 16) {
25.                 throw new Exception("El documento no tiene el formato esperado");
26.             }
27.         }
28.
29.         @Override
30.         protected void processFile() throws Exception {
31.             FileInputStream input = new FileInputStream(file);
32.             try {
33.                 byte[] fileContect = new byte[input.available()];
34.                 input.read(fileContect);
35.                 String message = new String(fileContect);
36.                 String[] lines = message.split("\n");
37.                 for (String line : lines) {
38.                     String id = line.substring(0, 3);
39.                     String customer = line.substring(3, 5);
40.                     double amount = Double.parseDouble(line.substring(5, 8));
41.                     String date = line.substring(8, 16);
42.                     boolean exist = OnMemoryDataBase.customerExist(
43.                             Integer.parseInt(customer));
44.
45.                     if (!exist) {
46.                         log += id + " E" + customer + "\t\t" + date
47.                                 + " El cliente no existe\n";
48.                     } else if (amount > 200) {
49.                         log += id + " E" + customer + "\t\t" + date
50.                                 + " El monto excede el máximo\n";
51.                     } else {
52.                         log += id + " E" + customer + "\t\t" + date
53.                                 + " Aplicado exitosamente\n";
54.                     }
55.                 }
56.             } finally {
57.                 try {
58.                     input.close();
59.                 } catch (Exception e) {
```

```
60.             }
61.         }
62.     }
63.
64.     @Override
65.     protected void createLog() throws Exception {
66.         FileOutputStream out = null;
67.         try {
68.             File outFile = new File(logPath + "/" + file.getName());
69.             if (!outFile.exists()) {
70.                 outFile.createNewFile();
71.             }
72.             out = new FileOutputStream(outFile, false);
73.             out.write(log.getBytes());
74.             out.flush();
75.         } finally {
76.             out.close();
77.         }
78.     }
79. }
```

En la línea 16 tenemos el constructor de la clase, el cual recibe el archivo a procesar, la ruta donde dejará el *log* y la ruta en la que tendrá que dejar el documento una vez procesado.

Los métodos implementados son:

- **validateName:** Método que valida el nombre del archivo. En la línea 19 valida que el archivo termine con la extensión .drug. En la línea 24 se valida que el nombre tenga una longitud de 12 caracteres, los cuales corresponde al formato ddmmyyyy (ocho caracteres) + ID de tres caracteres + .drug (cinco caracteres).
- **processFile:** Método de procesamiento del archivo. De las líneas 31 a la 34 cargamos el documento a la memoria, en la línea 36 y 37 separamos el documento en líneas e iteramos cada una. De las líneas 38 a la 41 sacamos las secciones de la línea como el id, cliente, monto y fecha (los utilizaremos para aplicar el pago). En la línea 42 a la 46 validamos que el cliente exista. De la línea 48 a la 50 validamos que el monto pagado no sea mayor de $200. Podemos apreciar que a medida que avanzamos vamos llenando la variable *log*, la cual va registrando los pagos aplicados y los errores detectados.

- **createLog:** En este método tomamos la variable *log* y la escribimos en la ruta definida en la variable *logPath* con el mismo nombre, este archivo será el *log* de la ejecución del archivo.

Archivo de la tienda de abarrotes:

Este será el documento que nos envié la tienda de abarrotes. En él estarán todos los pagos realizados por los clientes que realicen los pagos en las tiendas. El nombre del archivo es *'123.gry'*.

```
1.  001,01,100,10/03/2015
2.  002,02,050,12/03/2015
3.  003,03,090,13/03/2015
4.  004,10,150,13/03/2015
5.  005,11,200,14/03/2015
6.  006,20,125,14/03/2015
7.  007,25,010,14/03/2015
8.  008,30,111,14/03/2015
9.  009,40,321,15/03/2015
10. 010,50,241,16/03/2015
```

Este documento es un archivo plano que está dividido de la siguiente manera:

- **ID del pago:** Corresponde al primer *token*.
- **Número del cliente:** Corresponde al segundo *token*.
- **Monto pagado:** Corresponde al tercer *token*.
- **Fecha:** Corresponde al cuarto *token* en formato dd/mm/yyyy.

Este documento será procesado por el *templete GroceryFileProcess* el cual analizaremos a continuación.

Clase GroceryFileProcess:

Clase que hereda de *AbstractFileProcessTemplete* y tiene la finalidad de procesar los documentos provenientes de la tienda de abarrotes. Veamos que esta clase no tiene ningún método público, por lo que no podrá ser ejecutado directamente, tendrá que ejecutarse por medio del método *ejecute* de la súper clase.

```
1.  package oscarblancarte.ipd.templetemethod.impl;
2.
3.  import java.io.File;
4.  import java.io.FileInputStream;
5.  import java.io.FileOutputStream;
6.  import oscarblancarte.ipd.templetemethod.util.OnMemoryDataBase;
7.
8.  public class GroceryFileProcess extends AbstractFileProcessTemplete {
9.
10.         private String log = "";
11.
12.         public GroceryFileProcess(File file, String logPath, String movePath) {
13.             super(file, logPath, movePath);
14.         }
15.
16.         @Override
17.         protected void validateName() throws Exception {
18.             String fileName = file.getName();
19.             if (!fileName.endsWith(".gry")) {
20.                 throw new Exception("Nombre del archivo no válido"
21.                         + ", debe terminar con .gry");
22.             }
23.
24.             if (fileName.length() != 7) {
25.                 throw new Exception("El documento no tiene el formato esperado");
26.             }
27.         }
28.
29.         @Override
30.         protected void processFile() throws Exception {
31.             FileInputStream input = new FileInputStream(file);
32.             try {
33.                 byte[] fileContect = new byte[input.available()];
34.                 input.read(fileContect);
35.                 String message = new String(fileContect);
36.                 String[] lines = message.split("\n");
37.                 for (String line : lines) {
38.                     String[] parts = line.split(",");
39.                     String id = parts[0];
40.                     String customer = parts[1];
41.                     double amount = Double.parseDouble(parts[2]);
42.                     String date = parts[3];
43.                     boolean exist = OnMemoryDataBase.customerExist(
44.                             Integer.parseInt(customer));
45.                     if (!exist) {
46.                         log += id + " E" + customer + "\t\t" + date
47.                                 + " El cliente no existe\n";
48.                     } else if (amount > 200) {
49.                         log += id + " E" + customer + "\t\t" + date
50.                                 + " El monto excede el máximo\n";
51.                     } else {
52.                         //@TODO Aplicar el pago.
53.                         log += id + " E" + customer + "\t\t" + date
```

```
54.                         + " Aplicado exitosamente\n";
55.                 }
56.             }
57.         } finally {
58.             try {
59.                 input.close();
60.             } catch (Exception e) {
61.             }
62.         }
63.     }
64.
65.     @Override
66.     protected void createLog() throws Exception {
67.         FileOutputStream out = null;
68.         try {
69.             File outFile = new File(logPath + "/" + file.getName());
70.             if (!outFile.exists()) {
71.                 outFile.createNewFile();
72.             }
73.             out = new FileOutputStream(outFile, false);
74.             out.write(log.getBytes());
75.             out.flush();
76.         } finally {
77.             out.close();
78.         }
79.     }
80. }
```

Esta implementación es bastante similar a la de *DrugstoreFileProcess*, sin embargo, cuenta con algunas diferencias necesarias para procesar los archivos de la tienda de abarrotes.

- **validateName:** Valida el nombre mediante dos reglas; el nombre debe tener la extensión '**gry**' y debe de exceder de siete caracteres incluyendo la extensión.
- **processFile:** Mediante este método procesaremos los archivos entrantes. Una de las principales diferencias del archivo de la farmacia contra el de la tienda de abarrotes es que el de abarrotes separa los campos del archivo mediante comas, por lo que el archivo será procesado por líneas. Cada línea es un pago y cada *token* separado por coma será un atributo del pago.
- **createLog:** Finalmente, en este método se crea el *log* en disco.

Clase OnMemoryDataBase:

Clase que simula ser una base de datos en memoria, la cual guarda los ID de los clientes del sistema y los documentos procesados.

```
1.  package oscarblancarte.ipd.templetemethod.util;
2.
3.  import java.util.HashMap;
4.  import java.util.Map;
5.
6.  public class OnMemoryDataBase {
7.
8.      public static final Map<String, String> PROCESS_DOCUMENTS = new HashMap<>();
9.      public static final int[] CUSTOMERS = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9,
10.             10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
11.
12.         public static String getFileStatus(String fileName) {
13.             if (PROCESS_DOCUMENTS.containsKey(fileName)) {
14.                 return PROCESS_DOCUMENTS.get(fileName);
15.             }
16.             return null;
17.         }
18.
19.         public static void setProcessFile(String fileName) {
20.             PROCESS_DOCUMENTS.put(fileName, "Procesado");
21.         }
22.
23.         public static boolean customerExist(int id) {
24.             for (int i : CUSTOMERS) {
25.                 if (i == id) {
26.                     return true;
27.                 }
28.             }
29.             return false;
30.         }
31.     }
```

La variable *PROCESS_DOCUMENTS* guarda la relación de los documentos procesados, como llave se guarda el nombre del documento y como valor el *status* del documento.
El arreglo *CUSTOMER* almacena los clientes válidos de la aplicación.

Los métodos definidos son:

- **getFileStatus:** Regresa el *status* del documento. Cuando un documento ya fue procesado éste regresa *"Procesado".*
- **setProcessFile:** Método que marca un documento como procesado.
- **customerExist:** Indica si un cliente existe en la aplicación.

Clase TempleteMethodMain:

Clase ejecutable que utilizaremos para procesar los documentos de las farmacias y tiendas de abarrotes.

```
1.  package oscarblancarte.ipd.templetemethod;
2.
3.  import java.io.File;
4.  import java.util.Date;
5.  import java.util.Timer;
6.  import java.util.TimerTask;
7.  import oscarblancarte.ipd.templetemethod.impl.DrugstoreFileProcess;
8.  import oscarblancarte.ipd.templetemethod.impl.GroceryFileProcess;
9.
10.     public class TempleteMethodMain extends TimerTask {
11.
12.         private static final String[] PATHS =
13.             new String[]{"C:/files/drugstore", "C:/files/grocery"};
14.         private static final String LOG_DIR = "C:/files/logs";
15.         private static final String PROCESS_DIR = "C:/files/process";
16.
17.         public static void main(String[] args) {
18.             new TempleteMethodMain().start();
19.         }
20.
21.         public void start() {
22.             try {
23.                 Timer timer = new Timer();
24.                 timer.schedule(this, new Date(), (long) 1000 * 10);
25.                 System.in.read();
26.             } catch (Exception e) {
27.                 e.printStackTrace();
28.             }
29.         }
30.
31.         @Override
32.         public void run() {
33.             System.out.println("> Monitoring start");
34.             File f = new File(PATHS[0]);
35.             if(!f.exists()){
36.                 throw new RuntimeException("El path '"+PATHS[0]+"' no existe");
37.             }
```

```
38.         File[] drugstoreFiles = f.listFiles();
39.         for (File file : drugstoreFiles) {
40.             try {
41.                 System.out.println("> File found > " + file.getName());
42.                 new DrugstoreFileProcess(file,LOG_DIR,PROCESS_DIR).execute();
43.                 System.out.println("Archivo procesado > " + file.getName());
44.             } catch (Exception e) {
45.                 System.err.println(e.getMessage());
46.             }
47.         }
48.
49.         f = new File(PATHS[1]);
50.         if(!f.exists()){
51.             throw new RuntimeException("El path "'+PATHS[1]+'" no existe");
52.         }
53.         System.out.println("> Read Path " + PATHS[1]);
54.         File[] groceryFiles = f.listFiles();
55.         for (File file : groceryFiles) {
56.             try {
57.                 System.out.println("> File found > " + file.getName());
58.                 new GroceryFileProcess(file,LOG_DIR,PROCESS_DIR).execute();
59.                 System.out.println("Archivo procesado > " + file.getName());
60.             } catch (Exception e) {
61.                 System.err.println(e.getMessage());
62.             }
63.         }
64.     }
65. }
```

En la línea 12 establecemos las rutas del sistema de archivos que estaremos monitoreando por nuevo archivos. Tenemos dos rutas: la primera para los archivos de las tiendas de abarrotes y la segunda para las farmacias.

En la línea 14 definimos la ruta donde deberemos dejar los archivos de log.

En la línea 15 definimos dónde debemos colocar los archivos que ya fueron procesados.

Los métodos definidos son los siguientes:

- **start:** Crea un temporizador que ejecutará el método *run* cada 10 segundos.

- **run:** Cada vez que es ejecutado busca nuevos archivos de pago para ser procesados. Cuando un archivo es encontrado inicia el procesamiento de éste mediante el *templete* correspondiente según el tipo de archivo.

Cuando un archivo de farmacia es encontrado será procesado por la clase *DrugstoreFileProcess* (línea 42), en cambio, cuando un archivo de abarrotes es encontrado es procesado por la clase *GroceryFileProcess* (línea 58).

La ejecución:

Antes de continuar con la ejecución del programa debemos asegurarnos de que todas las rutas que definimos para los archivos y *logs* existan, estas carpetas deberán tener la siguiente estructura:

Nombre	Fecha de modificación	Tipo
drugstore	28/03/2015 04:06 p...	Carpeta de archivos
grocery	26/03/2015 03:57 p...	Carpeta de archivos
logs	26/03/2015 03:58 p...	Carpeta de archivos
process	28/03/2015 04:06 p...	Carpeta de archivos

Ilustración 146: Estructura de carpetas.

Una vez que tengamos la estructura de carpetas correctamente configurada, nos aseguramos de tener los archivos de pagos en las carpetas correspondientes, el archivo *10032015123.drug* deberá estar en la carpeta ***drugstore*** y el archivo *123.gry* deberá estar en la carpeta ***grocery***.

Ya con los archivo en las carpetas correctas ejecutaremos la clase *TempleteMethodMain* para obtener el siguiente resultado.

```
run:
> Monitoring start
> File found > 10032015123.drug
Archivo procesado > 10032015123.drug
> Read Path C:/files/grocery
> File found > 123.gry
Archivo procesado > 123.gry
```

Ilustración 147: Resultado de la ejecución.

Podemos ver que los archivos de la farmacia y la tienda de abarrotes fueron procesados exitosamente, sin embargo, tendremos que ver que sucedió en la ejecución. Lo primero que validamos es que los archivos fueron eliminados de su respectiva carpeta y puesto en la carpeta de procesados (process).

Nombre	Fecha de modificación	Tipo
123.gry	25/03/2015 10:43 a...	Archivo GRY
10032015123.drug	25/03/2015 08:09 p...	Archivo DRUG

Ilustración 148: Carpeta de archivos procesados.

Y finalmente, tendremos que ir a la carpeta de *logs* y abrir los archivos generados.

El log de la tienda de abarrotes quedaría de la siguiente manera:

```
1.  001 01   10/03/2015 Aplicado exitosamente
2.  002 02   12/03/2015 Aplicado exitosamente
3.  003 03   13/03/2015 Aplicado exitosamente
4.  004 10   13/03/2015 Aplicado exitosamente
5.  005 11   14/03/2015 El cliente no existe
6.  006 20   14/03/2015 Aplicado exitosamente
7.  007 25   14/03/2015 El cliente no existe
8.  008 30   14/03/2015 Aplicado exitosamente
9.  009 40   15/03/2015 El monto excede el máximo
10. 010 50   16/03/2015 El monto excede el máximo
```

Veamos que las líneas 5 y 7 no fueron procesadas por que el cliente que realizó el pago no existe. También las líneas 9 y 10 no fueron procesados por que el pago es mayor al monto máximo permitido.

El log de la farmacia quedaría de la siguiente manera:

```
1.  001 01   10032015 Aplicado exitosamente
2.  002 02   12032015 Aplicado exitosamente
3.  003 03   13032015 Aplicado exitosamente
4.  004 10   13032015 Aplicado exitosamente
5.  005 11   14032015 El cliente no existe
6.  006 20   14032015 Aplicado exitosamente
7.  007 25   14032015 El cliente no existe
8.  008 30   14032015 Aplicado exitosamente
9.  009 40   15032015 El monto excede el máximo
10. 010 50   16032015 El monto excede el máximo
```

El *log* de la farmacia es idéntico al de la tienda de abarrotes debido a que la información enviada era la misma pero en diferente formato.

Observemos que los pasos para procesar los archivos son los mismos a pesar de tener archivos distintos, cada paso del procesamiento de los archivos puede variar en su implementación pero los pasos requeridos para su procesamiento siempre serán los mismos, es aquí donde utilizar el patrón Templete Method cobra sentido. Si en el futuro un nuevo archivo es agregado, simplemente se tendría que implementar una nueva subclase del templete con las reglas de procesamiento para ese archivo específico.

Siguientes pasos:

Hasta aquí ya hemos implementado el patrón Templete Method y hemos visto las grandes ventajas que nos ofrece, por lo cual continuaremos con algunos ejercicios para reafirmar los conocimientos adquiridos:

- Agreguemos un nuevo archivo que nos permita recibir los pagos realizados por medio de cargos automáticos, este archivo nos lo envía el banco y contiene los cargos realizados a los clientes directamente sobre su tarjeta de crédito/débito. El formato del archivo será en XML y tendrá la estructura que más te guste. El procesamiento y el log deberán tener las reglas ya conocidas.

- Agreguemos un paso extra al *templete*, el cual cree un nuevo archivo con todos los movimientos que no pudieron ser procesados, este archivo deberá tener el mismo formato que el archivo de entrada más el motivo por el cual no puedo ser procesado. Luego debemos implementarlo en todas las subclases.

- Creemos un nuevo *templete* más simple, esta vez sólo se deberá procesar el archivo y moverlo de carpeta, para lo cual tendremos que crear la clase templete y las implementaciones para procesar los archivos.

Patrón Strategy

El patrón de diseño *strategy* permite establecer en tiempo de ejecución el rol de comportamiento de una clase. *Stretagy* se basa en el polimorfismo para implementar una serie de comportamientos que podrán ser intercambiados durante la ejecución del programa, logrando con esto que un objeto se pueda comportar de forma distinta según la estrategia establecida.

Este patrón suele trabajar en conjunto con el patrón *Factory* Method para la creación dinámica de las estrategias, pero no es requerido para su implementación.

Strategy está compuesto por el contexto y las estrategias.

El contexto es una clase que encapsula la estrategia y su comportamiento está totalmente ligado a la estrategia concreta que contenga, por otra parte están las estrategias, las cuales son clases que definen un comportamiento concreto por lo que entre más estrategias existan más funcionalidad podrá tener el contexto.

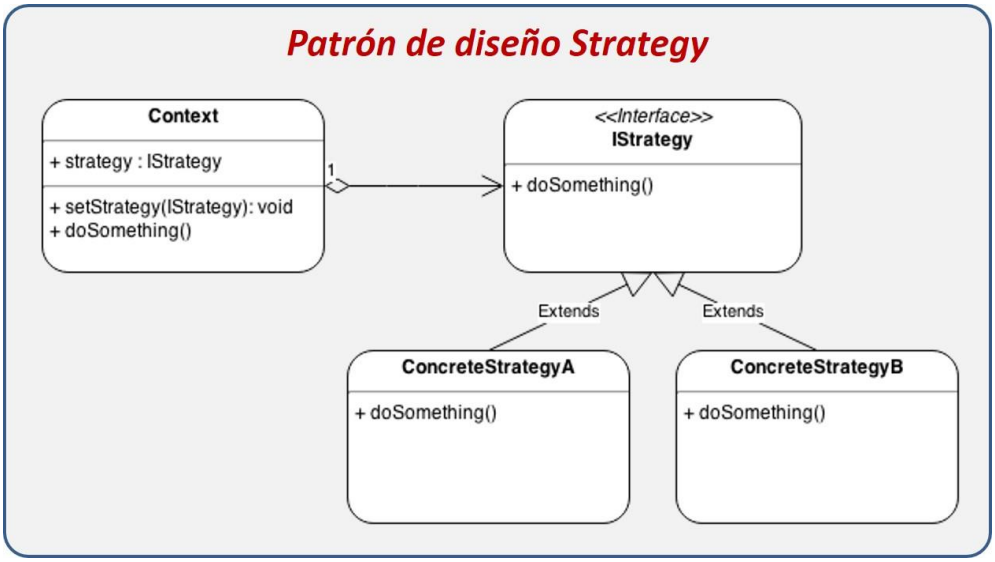


Ilustración 149: Estructura del patrón de diseño Strategy

.

Los componentes del patrón son los siguientes:

- **Context:** Componente que encapsula la estrategia a utilizar, tiene como característica que se puede establecer la estrategia a utilizar en tiempo de ejecución.
- **IStrategy:** Interface en común que todas las estrategias deberán implementar. En esta interface se definen las operaciones que las estrategias deberán implementar.
- **ConcreteStrategy:** Representa las estrategias concretas, las cuales heredan de *IStrategy*.

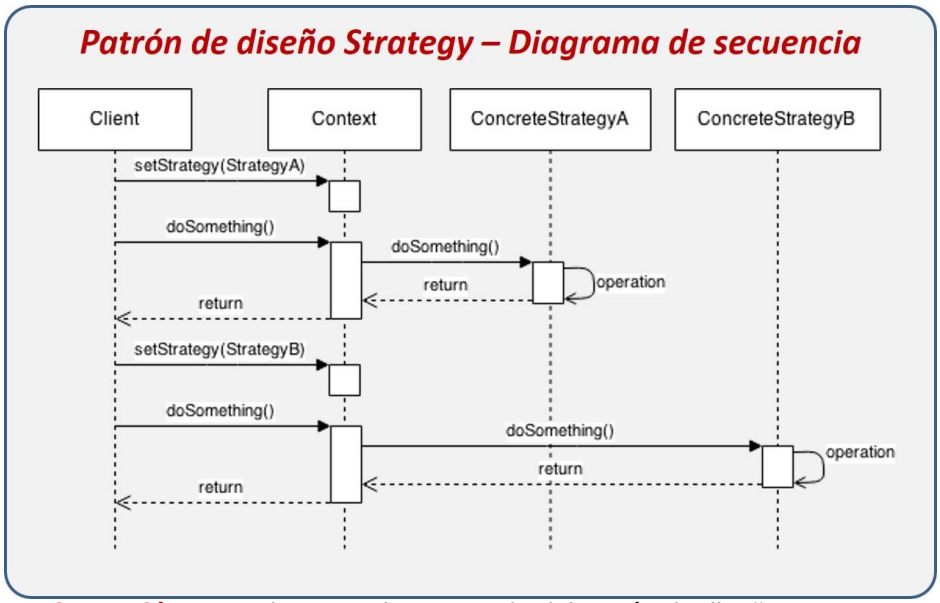


Ilustración 150: Diagrama de secuencia del patrón de diseño Strategy.

El diagrama anterior se interpreta de la siguiente manera:

1. El cliente crea un nuevo contexto y establece la estrategia A.
2. El cliente ejecuta la operación *doSomething*.
3. *Context* a su vez delega esta responsabilidad a *ConcreteStrategyA*.
4. *ConcreteStrategyA* realiza la operación y regresa el resultado.
5. *Context* toma el resultado y lo regresa al cliente.
6. El cliente le cambia la estrategia a *Context* en tiempo de ejecución.
7. El cliente ejecuta nuevamente la operación *doSomething*.

8. *Context* a su vez delega esta responsabilidad a *ConcreteStrategyB*.
9. *ConcreteStategyB* realiza la operación y regresa el resultado.
10. *Context* toma el resultado y lo regresa al cliente.

Como podemos ver en el diagrama, el patrón estrategia nos permite cambiar dinámicamente la estrategia en tiempo de ejecución para resolver un problema, las estrategias pueden ser intercambiadas por distintas razones, aunque el motivo más común es que sean intercambiadas según las configuraciones de nuestro aplicativo.

Cuándo utilizarlo:

- Cuando se requiere que el comportamiento de una clase puede variar según el estado de la aplicación o simplemente por configuración.

- Cuando un conjunto de clases se diferencia únicamente por su comportamiento. El comportamiento se podría extraer en una estrategia para después ser establecida según las circunstancias.

El escenario:

Con la finalidad de comprender mejor lo que acabamos de explicar expondremos un escenario bastante común en las aplicaciones.

Imaginemos que estamos diseñando una aplicación que hace cualquier cosa pero nos han pedido que esta aplicación permita definir la forma en la que los usuario se podrán autentificar con la aplicación, de entrada nuestro cliente nos dice que les dará la opción de que los usuarios se puedan autentificar de tres formas, sin embargo, nos ha dejado en claro que requiere que la aplicación esté preparada para soportar más formas de autentificación.
Los métodos de autentificación que la aplicación deberá de soportar son:

- **XML Authentification:** Permite al administrador dar de alta a los usuarios de la aplicación mediante un archivo XML.
- **SQL Authentification:** Permitirá al administrador dar de alta usuarios mediante el registro de éstos en una tabla de la base de datos.
- **OnMemory Authentification:** Permitirá al administrador dar de alta a nuevos usuarios que vivan en la memoria del programa.

Los puntos a tomar en cuenta son los siguientes:

- La autentificación se realizará mediante usuario y contraseña y el autenticador nos deberá indicar el rol del usuario dentro de la aplicación.
- La aplicación le deberá indicar el método de autentificación que desea utilizar para el logeo.

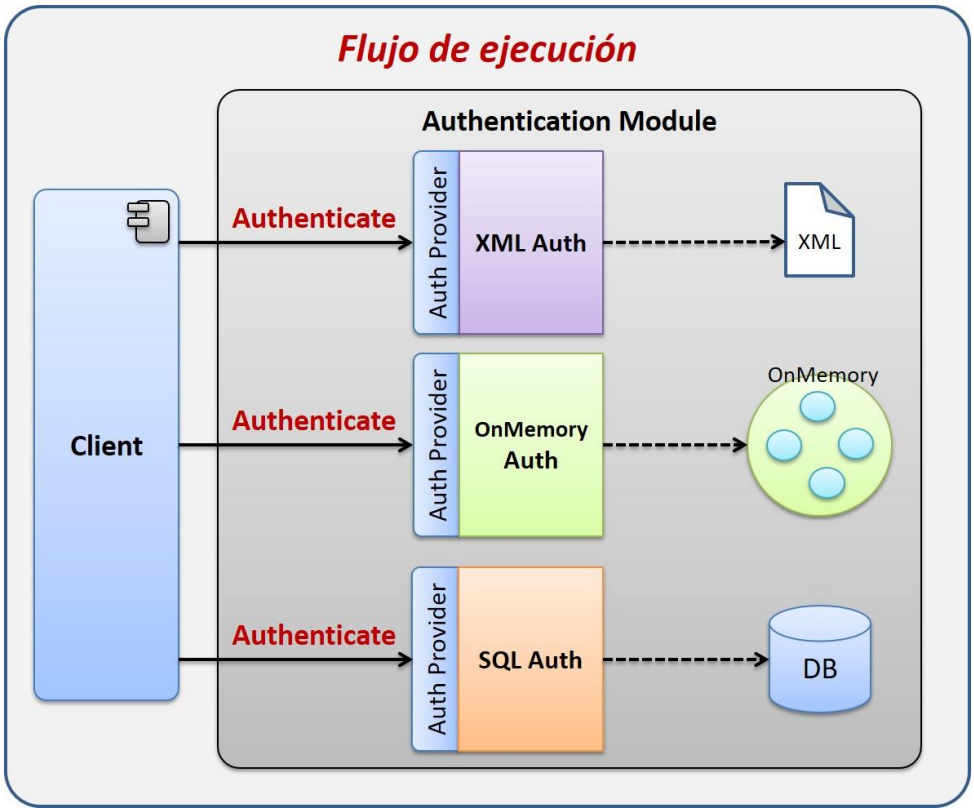


Ilustración 151: Autentificación sin utilizar el patrón de diseño Strategy.

En la imagen podemos apreciar un escenario de autentificación sin utilizar el patrón estrategia, en la imagen podemos apreciar los tres proveedores de autentificación disponibles. Si bien todos implementan la misma interfaz, este escenario presenta algunos problemas.

El primero es que el cliente debe estar atento cuando la estrategia de autentificación cambie con el fin de utilizar un proveedor distinto.

Segundo, es necesario conocer las implementaciones concretas utilizadas para realizar la autentificación.

Tercero, es que no siempre podemos notificar al cliente en tiempo de ejecución para indicarle la nueva estrategia.

El cuarto problema es que tenemos que configurar al cliente cada vez que una nueva estrategia sea dada de alta, por ejemplo una autentificación por medio de un servidor LDAP o por WebService.

La solución:

Para darle una solución a los requerimientos que nos han solicitado, hemos decidido implementar el patrón de diseño *Strategy*.
Como hemos visto tenemos que soportar varios tipos de autentificación y dejar preparada la aplicación para soportar nuevas implementaciones.

Lo primero que debemos tomar en cuenta es la necesidad de crear un componente que se encargue de la autentificación y deberá contener la estrategia a utilizar para dicha autentificación, este componente lo conocemos como el *Context* y en nuestro caso será el ***AuthentificationProvier*** que delegará la responsabilidad de autentificación a una estrategia previamente definida, además de que permitirá el intercambio de esta estrategia en tiempo de ejecución.

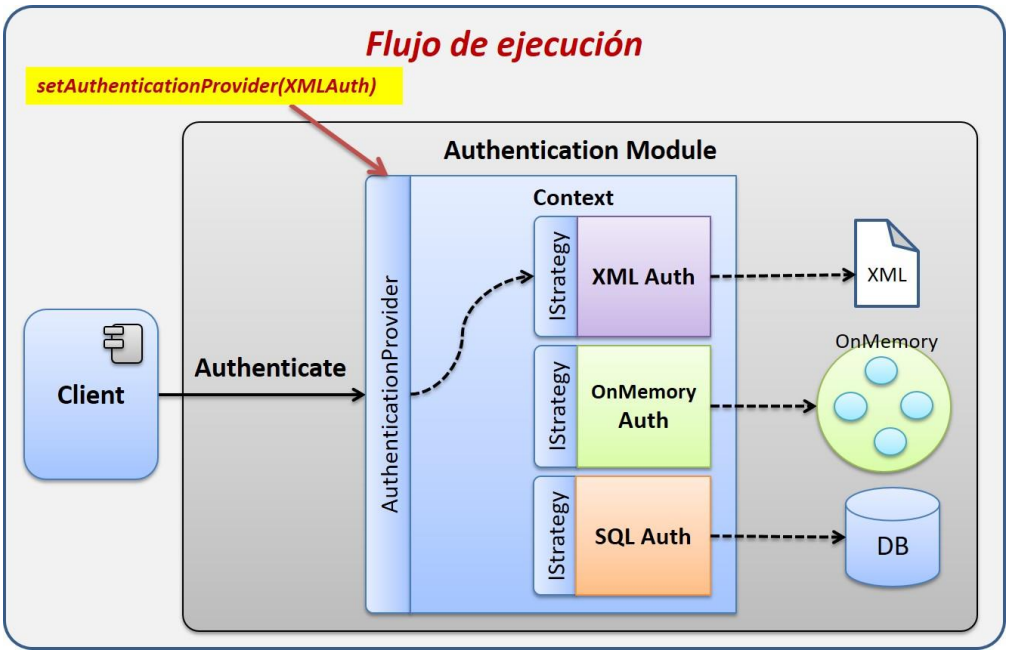


Ilustración 152: Autentifcación por medio del patrón de diseño Strategy.

En la imagen podemos apreciar que un nuevo componente es colocado, el componente *AuthentificationProvider* será el encargado de realizar la autentificación de usuarios por medio de una de las estrategias establecidas. En la imagen también se establece la estrategia XML *Authentification* por lo que

cuando el cliente intenta realizar el logeo al *AuthentificationProvider* éste automáticamente delega la responsabilidad a la estrategia de logeo por XML.

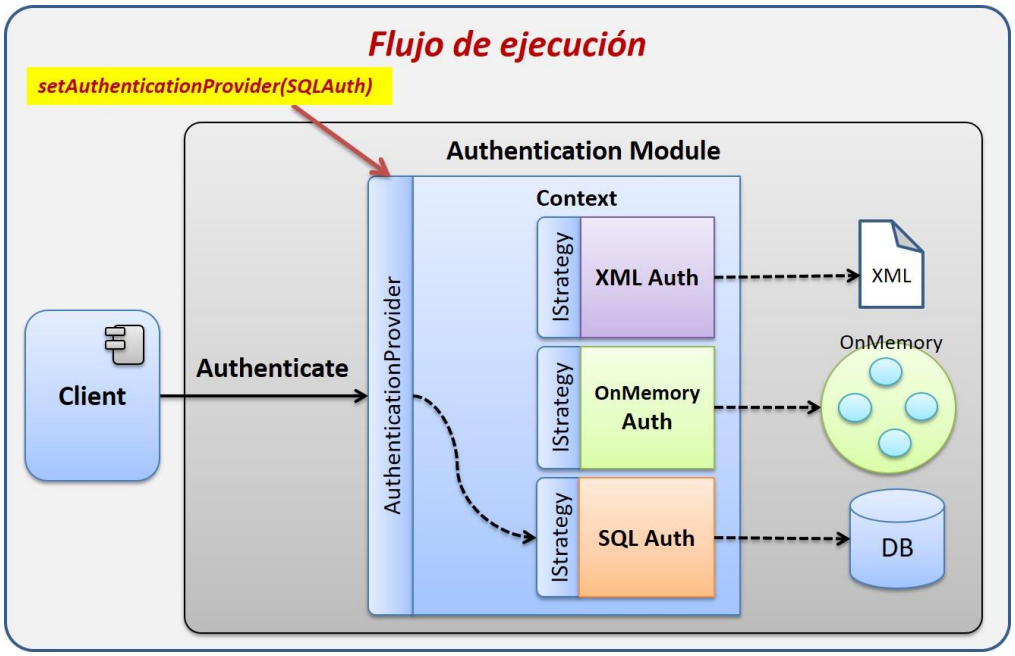


Ilustración 153: Imagen que muestra el intercambio de la estrategia.

En la imagen podemos apreciar algo muy interesante; se establece una nueva estrategia de autentificación, lo que provoca que cuando el usuario intenta autentificarse en la aplicación esta vez su petición es procesada por la estrategia SQL *Authentication*, esto desde luego que tiene grandes ventajas.
Primero, el cliente jamás se entera del cambio de implementación; dos, si cambia la estrategia esto no le afectará ya que el cambio lo realiza el *AuthentificationProvider* en vez de hacerlo el cliente; tres, si se agregan nuevas estrategias de autenticación, el cliente no se tendrá que preocupar por conocerlas ya que esa será tarea del componente *AuthentificationProvider*.

La implementación:

Una vez que conocemos lo que tenemos que hacer iniciaremos con la implementación del proyecto:

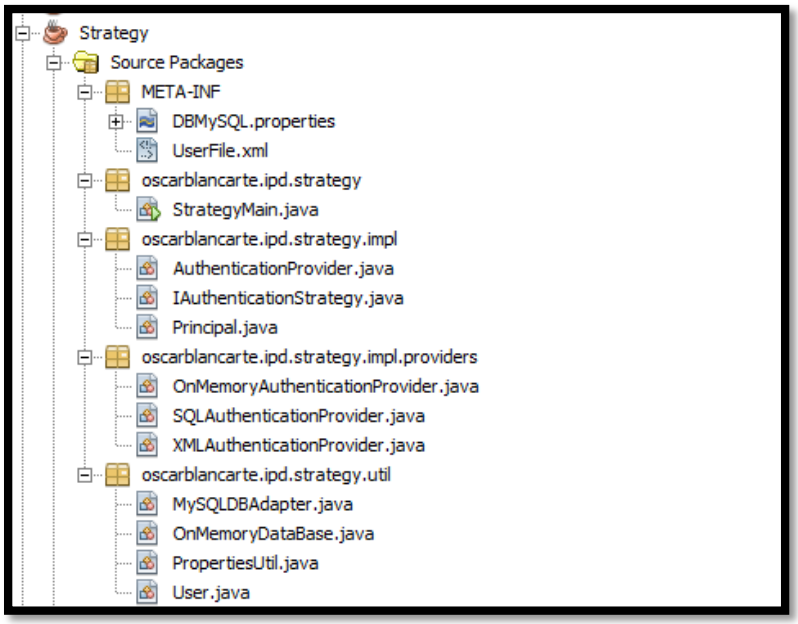


Ilustración 154: Estructura del proyecto Strategy.

En la imagen podemos apreciar los paquetes que componen la aplicación:

- **META-INF:** Paquete en donde dejamos los archivos de configuración necesarios para la ejecución del programa.
- **oscarblancarte.ipd.strategy:** Paquete principal del proyecto donde sólo tenemos la clase ejecutable *StrategyMain*.
- **oscarblancarte.ipd.strategy.impl:** Paquete donde tenemos los componentes básicos del patrón, la interfaz de las estrategias y la clase que utilizaremos como *Context*.
- **oscarblancarte.ipd.strategy.impl.providers:** Paquete en el que están todas las estrategias concretas de autentificación.
- **oscarblancarte.ipd.strategy.util:** Paquete con clases de utilidad que utilizaremos en el programa.

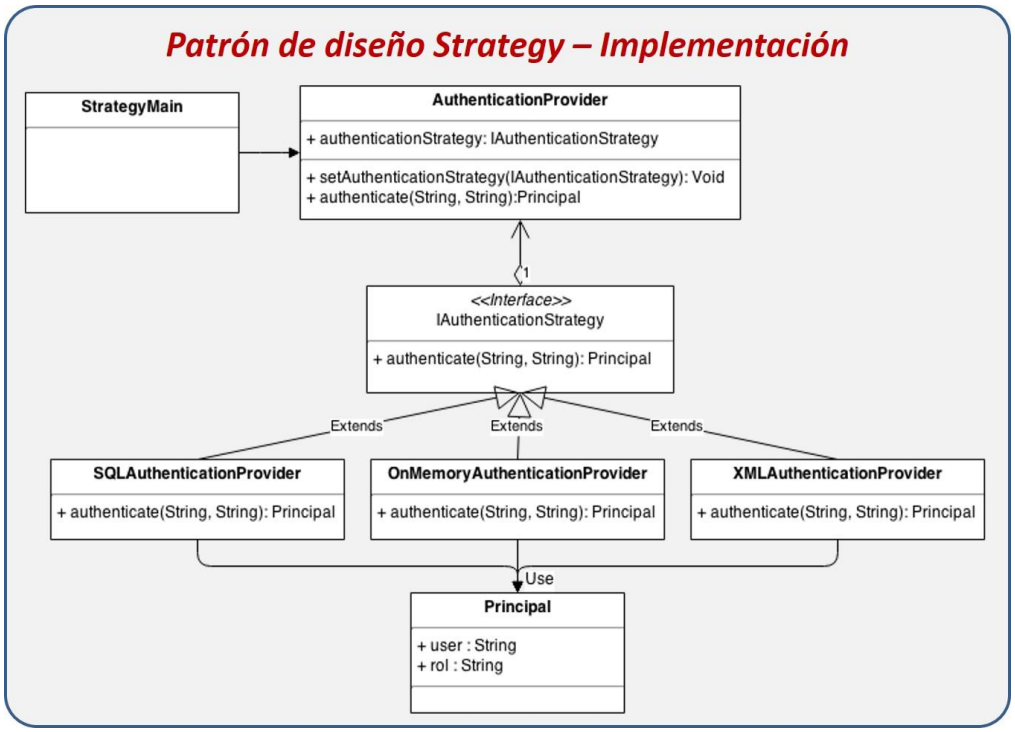


Ilustración 155: Diagrma de clases del proyecto Strategy.

En la imagen podemos ver las principales clases que se utilizarán para el proyecto, también podemos apreciar cómo se relacionan entre sí.

Interface IAuthenticationStrategy:

Interface que deberán implementar todas las estrategias de autentificación, se define el método *authenticate* que sirve para determinar si el usuario se puedo autentificar y el rol al que pertenece.

```
1. package oscarblancarte.ipd.strategy.impl;
2.
3. public interface IAuthenticationStrategy {
4.
5.     public Principal authenticate(String userName, String passwrd);
6. }
```

Clase Principal:

Clase utilizada para representar la autentificación del usuario, las propiedades son *userName* y *rol* los cuales indican el nombre de usuario y el rol al que pertenece.

```
1.  package oscarblancarte.ipd.strategy.impl;
2.
3.  public class Principal {
4.
5.      private String userName;
6.      private String rol;
7.
8.      public Principal(String userName, String rol) {
9.          this.userName = userName;
10.         this.rol = rol;
11.     }
12.    /** GET AND SET */
13.
14.     @Override
15.     public String toString() {
16.         return "Principal{" + "userName=" + userName + ", rol=" + rol + '}';
17.     }
18.
19. }
```

Clase AuthentificationProvider:

Clase que se utilizada para llevar a cabo la autentificación, en la clase podemos ver que se tiene una referencia hacia un *AuthentificationStrategy*, él cual indica la estrategia de autentificación que se llevará a cabo.

```
1.  package oscarblancarte.ipd.strategy.impl;
2.
3.  public class AuthenticationProvider {
4.      private IAuthenticationStrategy authenticationStrategy;
5.
6.      public void setAuthenticationStrategy(IAuthenticationStrategy strategy){
7.          this.authenticationStrategy = strategy;
8.      }
9.
10.     public Principal authenticate(String userName, String password){
11.         if(authenticationStrategy==null){
12.             throw new RuntimeException("Estrategia de autenticación no definida");
13.         }
```

```
14.         return authenticationStrategy.authenticate(userName, password);
15.     }
16. }
```

Esta clase representa el *Context* y tiene los métodos:

- **setAuthenticationStrategy:** Que se utilizará para cambiar la estrategia de autentificación en tiempo de ejecución.
- **authenticate:** Método que utilizará el cliente para llevar a cabo la autentificación. Este método delega la tarea a la estrategia establecida previamente.

Clase OnMemoryAuthenticationProvider:

Esta clase representa la estrategia de autentificación con usuarios en memoria.

```
1. package oscarblancarte.ipd.strategy.impl.providers;
2.
3. import oscarblancarte.ipd.strategy.impl.*;
4. import oscarblancarte.ipd.strategy.util.*;
5.
6. public class OnMemoryAuthenticationProvider implements IAuthenticationStrategy{
7.
8.     @Override
9.     public Principal authenticate(String userName, String passwrd) {
10.         User user = OnMemoryDataBase.findUserByName(userName);
11.         if(user!=null && user.getPassword().equals(passwrd)){
12.             return new Principal(user.getUserName(), user.getRol());
13.         }
14.
15.         return null;
16.     }
17. }
```

El método *authentication* utiliza la clase de utilidad *OnMemoryDataBase* para buscar al usuario en memoria, si lo encuentra entonces regresa una instancia de *Principal* con el nombre del usuario y el rol, en caso de no encontrarlo regresa *null*.

Clase OnMemoryDataBase:

Clase utilizada para almacenar a los usuarios en memoria.

```
1.  package oscarblancarte.ipd.strategy.util;
2.
3.  import java.util.HashMap;
4.
5.  public class OnMemoryDataBase {
6.
7.      private static final HashMap<String, User> USER_MAP = new HashMap<>();
8.
9.      static {
10.         USER_MAP.put("oblancarte",new User("oblancarte", "1234", "Admin"));
11.         USER_MAP.put("rlopez",new User("rlopez", "2345", "Cajero"));
12.         USER_MAP.put("lcastro",new User("lcastro", "2345", "Supervisor"));
13.     }
14.
15.     public static User findUserByName(String name){
16.         return USER_MAP.get(name);
17.     }
18. }
```

Los usuarios válidos son almacenados en el Map *USER_MAP*, también se proporciona el método *findUserByName* que busca los usuarios válidos que están en memoria.

Clase User:

Clase utilizada para representar un usuario en memoria, contiene los atributos nombre de usuario, *password* y rol.

```
1.  package oscarblancarte.ipd.strategy.util;
2.
3.  public class User {
4.
5.      private String userName;
6.      private String password;
7.      private String rol;
8.
9.      public User(String userName, String password, String rol) {
10.         this.userName = userName;
11.         this.password = password;
```

```
12.         this.rol = rol;
13.     }
14.
15.     /** GET AND SET */
16. }
```

Clase SQLAuthenticationProvier:

Esta clase representa la estrategia de autentificación con usuarios por medio de base de datos.

```
1. package oscarblancarte.ipd.strategy.impl.providers;
2.
3.
4. import java.sql.*;
5. import oscarblancarte.ipd.strategy.impl.*;
6. import oscarblancarte.ipd.strategy.util.MySQLDBAdapter;
7.
8. public class SQLAuthenticationProvider implements IAuthenticationStrategy{
9.     private static final String USER_QUERY = "SELECT userName,rol from users" +
10.                 "where userName='%s' and password = '%s'";
11.     private MySQLDBAdapter mysqlAdapter;
12.
13.     public SQLAuthenticationProvider(){
14.         mysqlAdapter = new MySQLDBAdapter();
15.     }
16.
17.     @Override
18.     public Principal authenticate(String user, String passwrd) {
19.         try {
20.             Connection connection = mysqlAdapter.getConnection();
21.             Statement statement = connection.createStatement();
22.             String queryUser = String.format(USER_QUERY, user,passwrd);
23.             ResultSet query = statement.executeQuery(queryUser);
24.             while(query.next()){
25.                 return new Principal(query.getString("userName"),
26.                         query.getString("rol"));
27.             }
28.             return null;
29.         } catch (Exception e) {
30.             e.printStackTrace();
31.             return null;
32.         }
33.     }
34. }
```

En el constructor de la clase instanciamos a la clase *MySQLDBAdapter* la cual nos permitirá obtener conexiones a la base de datos MySQL, esta clase ya la analizamos con el patrón *FactoryMethod* por lo que entraremos en detalles.

El método *authenticate* obtiene una conexión a la base de datos y lanza una consulta a la tabla de usuarios para determinar si el usuario existe en la base de datos. La definición de la tabla *users* la vemos a continuación.

Tabla Users:

Definición de la tabla *Users* con tres usuarios válidos. El *script* es para MySQL.

```
1.  CREATE TABLE `users` (
2.    `userName` varchar(45) NOT NULL,
3.    `password` varchar(45) NOT NULL,
4.    `rol` varchar(45) NOT NULL,
5.    PRIMARY KEY (`userName`)
6.  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
7.
8.  insert into users(userName,password,rol) values('oblancarte','1234','Admin');
9.  insert into users(userName,password,rol) values('amedina','2345','Cajero');
10. insert into users(userName,password,rol) values('iolvera','3456','Supervidor');
```

Clase XMLAuthenticationProvider:

Esta clase representa la estrategia de autentificación por medio de un archivo XML.

```
1.  package oscarblancarte.ipd.strategy.impl.providers;
2.
3.  import java.io.IOException;
4.  import java.io.InputStream;
5.  import javax.xml.parsers.DocumentBuilder;
6.  import javax.xml.parsers.DocumentBuilderFactory;
7.  import javax.xml.xpath.XPath;
8.  import javax.xml.xpath.XPathFactory;
9.  import org.w3c.dom.Document;
10. import oscarblancarte.ipd.strategy.impl.IAuthenticationStrategy;
11. import oscarblancarte.ipd.strategy.impl.Principal;
12.
13. public class XMLAuthenticationProvider implements IAuthenticationStrategy {
```

```
14.
15.     private static final String rolXPath = "Users/User[@userName='%s'" +
16.                                         "and @password='%s']/@rol";
17.
18.     @Override
19.     public Principal authenticate(String userName, String passwrd) {
20.         InputStream file = null;
21.         try {
22.             file = getClass().getResourceAsStream("/META-INF/UserFile.xml");
23.             DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
24.             DocumentBuilder builder = builderFactory.newDocumentBuilder();
25.             Document xmlDocument = builder.parse(file);
26.             XPath xPath = XPathFactory.newInstance().newXPath();
27.
28.             String xpath = String.format(rolXPath, userName,passwrd);
29.             String rol = xPath.compile(xpath).evaluate(xmlDocument);
30.             if(rol!=null && !rol.isEmpty()){
31.                 return new Principal(userName, rol);
32.             }
33.             return null;
34.         } catch (Exception e) {
35.             e.printStackTrace();
36.             return null;
37.         } finally {
38.             try { file.close(); } catch (IOException ex) {}
39.         }
40.     }
41. }
```

Mediante esta clase se podrán autenticar usuarios por medio de un archivo de usuarios en XML. El archivo que se utilizará para autentificar a los usuarios se encuentra en el paquete ***META-INF/UserFile.xml.***

De las líneas 22 a la 25 cargamos el XML con los usuarios, en la línea 29 se ejecuta un *XPath* sobre el XML para buscar al usuario, si se encuentra se regresa una instancia de la clase *Principal***,** de lo contrario regresa *null*.

En la línea 15 podemos apreciar el *XPath* utilizado para buscar el usuario.

XML UserFile.xml:

A continuación el contenido del archivo UserFile.xml que tiene los usuarios válidos para la aplicación.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <Users>
3.     <User userName='oblancarte' rol='Admin' password='1234' />
4.     <User userName='jperez' rol='Cajero' password='2345' />
5.     <User userName='fsuarez' rol='Supervisor' password='3456' />
6. </Users>
```

La ejecución:

Para probar la aplicación ejecutaremos la clase *StrategyMain* que nos arrojará el siguiente resultado:

```
run:
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
```

Ilustración 156: Ejecución de la clase StrategyMain.

Lo primero que nos indicará la aplicación es que elijamos el método de autentificación, para este ejemplo elegimos XML *Authentication* (opción 3).

```
run:
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
3
XML Authentication Select >


Favor de autenticarse:
Usuario:
```

Ilustración 157: Resultado de la elección de la estrategia de autentificación.

En este punto no está pidiendo que pongamos el usuario y *password*.

```
run:
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
3
XML Authentication Select >


Favor de autenticarse:
Usuario:
oblancarte
Pasword:
1234
Autentificación exitosa
Principal{userName=oblancarte, rol=Admin}
BUILD SUCCESSFUL (total time: 4 minutes 25 seconds)
```

Ilustración 158: Autentificación exitosa.

Una vez que ponemos un usuario y *password* válidos el sistema, nos indica que la autentificación fue exitosa y nos dice el rol del usuario, en este caso es Admin.

Prueba dos:

Corremos nuevamente la aplicación y elegimos la opción 3, ponemos un usuario inválido y nos sale lo siguiente:

```
run:
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
3
XML Authentication Select >


Favor de autenticarse:
Usuario:
amedina
Pasword:
2345
Usuario o password inválido.
¿Desea cambiar el metodo de autentificación? (S/N)
```

Ilustración 159: Authentificación fallida.

Ponemos el usuario amedida/2345 que no está registrado en el XML, luego nos pregunta si deseamos cambiar de método de autentificación y le decimos que sí (S). Nos pedirá que volvamos a elegir la estrategia y elegimos la opción 2 (SQL *Authentication*).

```
Usuario o password inválido.
¿Desea cambiar el metodo de autentificación? (S/N)
s
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
2
SQL Authentication Select >


Favor de autenticarse:
Usuario:
```

Ilustración 160: Cambio de estrategia de autentificación.

En la imagen podemos apreciar que hemos cambiado en tiempo de ejecución la estrategia de autentificación a SQL *Authentication* por lo que volvemos a reintentar con el mismo usuario amedina/2345.

```
Introdusca el tipo de autentificación a utilizar.
1.-OnMemory Authentication
2.-SQL Authentication
3.-XML Authentication
2
SQL Authentication Select >


Favor de autenticarse:
Usuario:
amedina
Pasword:
2345
ConnectionString ==> jdbc:mysql://localhost:3306/pos?user=root&password=1234
Connection class ==> com.mysql.jdbc.JDBC4Connection
Autentificación exitosa
Principal{userName=amedina, rol=Cajero}
BUILD SUCCESSFUL (total time: 3 minutes 47 seconds)
```

Ilustración 161: Autentificación exitosa con SQLAuthentication.

La autentificación se realizó exitosamente con el mismo usuario que nos había marcado error, sin embargo, cambiamos la estrategia de autentificación a SQL *Authentication* donde **amedina** sí era un usuario válido y no permitió el acceso.

Siguientes pasos:

Como ya vimos, el patrón estrategia nos sirve para cambiar dinámicamente las estrategias para resolver un problema, en este caso hemos resulto el problema de autenticación mediante tres formas distintas. Sin embargo, no hemos probado con *OnMemoryAuthentication* por lo que no lo quedaremos de tarea.

Finalmente, para fortalecer los conocimientos realizaremos lo siguientes ejercicios:

- Como primer ejercicio debemos crear nuestro propio usuario en todos los *Authentication Provider* que hemos creado. Correremos la aplicación y podremos logearnos con el proveedor que sea.

- Creemos un nuevo *Authentication Provider* que permita logearnos por medio de un servicio web (WebService) el cual recibirá también el usuario y *password*. Esta estrategia deberá salir como la opción 4 del menú.

- Creemos un nuevo *Authentication Provider* que busque logearse en todos los *authentication provider* existentes, de tal manera que busque al usuario en la estrategia OnMemoryAuthenticationProvider, si no encuentra al usuario continuará la búsqueda con la estrategia *SQLAuthenticationProvider*, si nuevamente el usuario no es encontrado debemos buscarlo en la estrategia *XMLAuthenticationProvider*. Si el usuario no es encontrado en ninguna estrategia debemos regresar *null*, en caso contrario regresaremos el Objeto Principal con el rol del usuario. (Este nuevo proveedor podría seguir el patrón *composite*).

Patrón Chain of Reponsability

El patrón de diseño Cadena de responsabilidad (Chain of Responsability) es un patrón que se distingue por su versatilidad, permitiendo resolver problemas donde no estamos muy seguros de qué objeto deberá procesar una solicitud concreta; este patrón de diseño resuelve problemas fácilmente donde la herencia no puede. Apoyándose de una estructura en forma de cadena donde una secuencia de objetos tratan de atender una petición.

Mediante este patrón se evita el acoplamiento entre el que envía y el que recibe una petición. Por lo cual una serie de objetos encadenados reciben las peticiones, el mensaje pasa a través de la cadena hasta que alguno de los objetos encadenados sea capaz de procesar el mensaje.

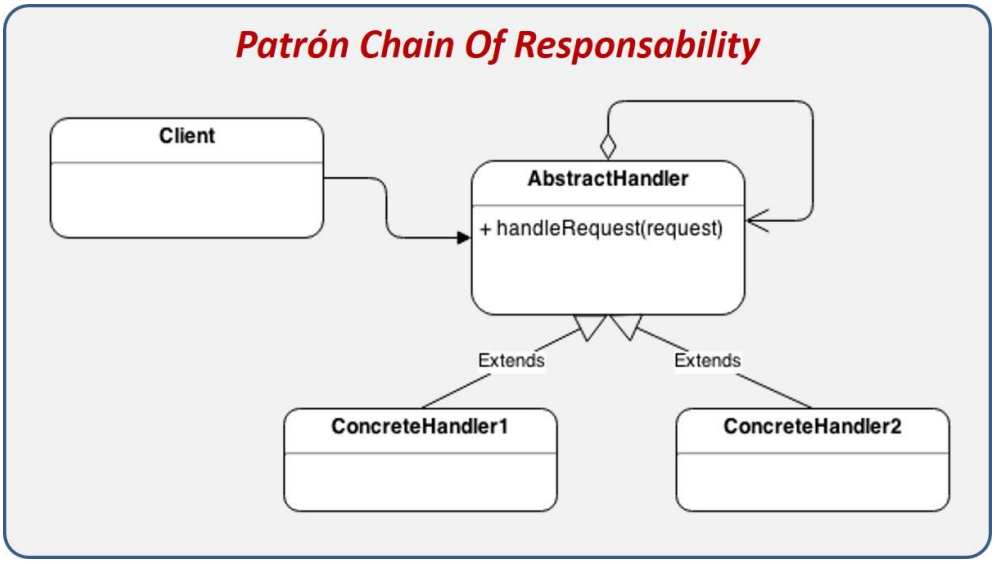


Ilustración 162: Patrón de diseño Chain of Responsability.

Los componentes del patrón se explican a continuación:

- **Client:** Usuario o sistema que inicia la ejecución del patrón.
- **AbstractHandler**: Clase base utilizada para definir la estructura de todos los *ConcreteHandler*. Esta clase es una agregación de sí misma, lo que le

permite contener otro *AbstractHandler* que continuará con la cadena de ejecución.

- **ConcreteHandler:** Representan las implementaciones concretas de *AbstractHandler* las cuales se utilizarán para procesar las solicitudes.

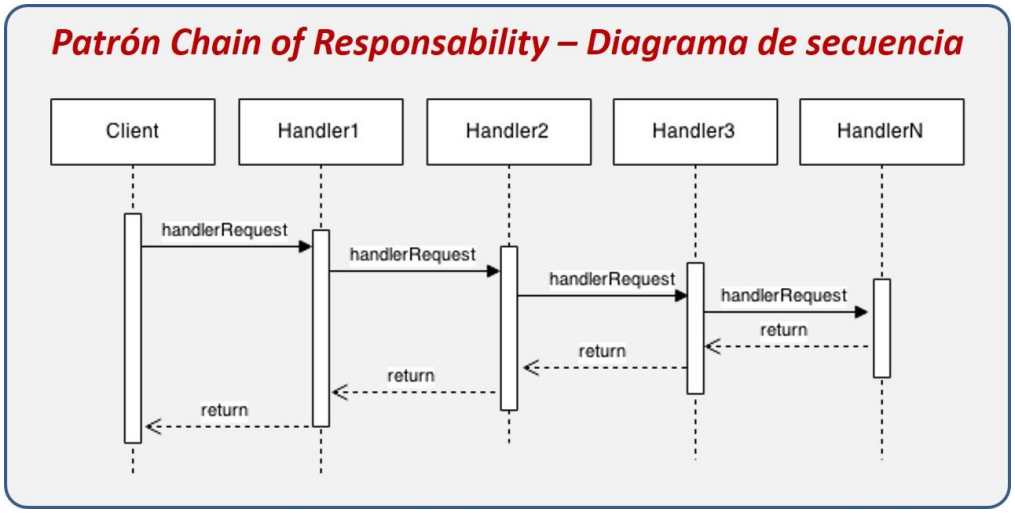


Ilustración 163: Diagrama de secuencia del patrón Chain of Responsability.

El diagrama se interpreta de la siguiente manera:

1. El cliente solicita el procesamiento de una solicitud a una cadena de responsabilidad.
2. El **primer *Handler*** intenta procesar el mensaje, sin embargo, no es capaz de procesarlo por alguna razón y envía el mensaje al siguiente *handler* de la cadena.
3. El **segundo *Handler*** intenta procesar el mensaje sin éxito, por lo que envía el mensaje al siguiente *Handler* de la cadena.
4. El **tercer *Handler*** también intenta procesar el mensaje sin éxito y envía el mensaje al siguiente *Handler* de la cadena.
5. El ***HandlerN*** (Algun *handler* de la secuencia) por fin es capaz de procesar el mensaje exitosamente y regresa una respuesta (opcional) de tal forma que la respuesta es replicada por todos los *Handlers* pasados hasta llegar al Cliente.

Este patrón nos permite crear una cadena de objetos que son ejecutados en secuencia hasta encontrar algún objeto que es capaz de procesar el mensaje, de tal forma que el mensaje pueda ser procesado por el primer objeto de la cadena o hasta el último; desde luego que el cliente nunca sabrá quién fue realmente el que procesó su mensaje.

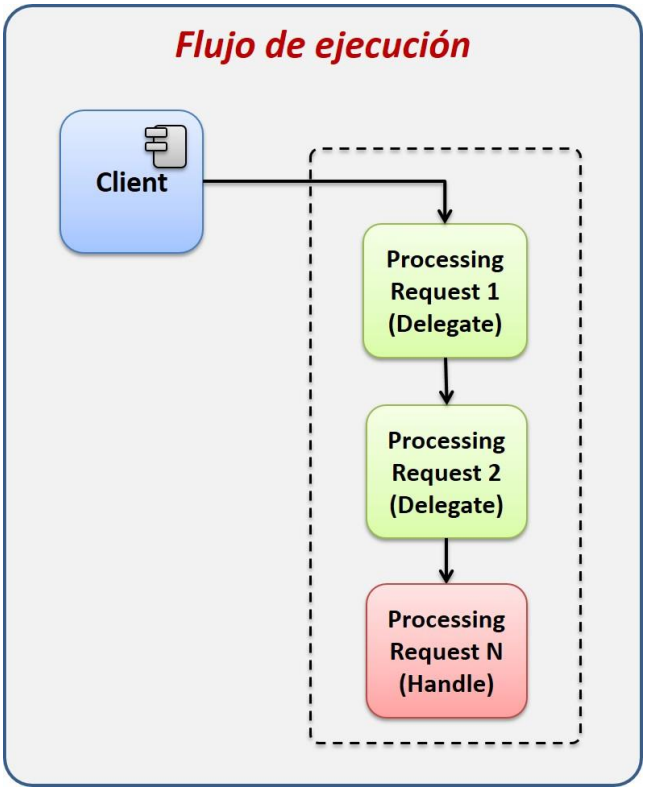


Ilustración 164: Proceso de encadenamiento de objetos.

La siguiente imagen trata de ilustrar de otra manera cómo es que el patrón permite encadenar *Handler* con la esperanza de que alguno pueda procesar los mensajes entrantes. Es importante notar que se puede dar el caso de que ninguno de los Handler sea capaz de procesar el mensaje por lo que será necesario implementar algún Handler por *default*.

Cuándo utilizarlo:

- Cuando en tiempo de diseño no somos capaces de determinar de un conjunto de objetos cuál de ellos podrá atender una petición dada.
- Cuando un conjunto de objetos podría procesar una petición y éstos pueden agregarse dinámicamente.
- Cuando se requiere lanzar una petición sin saber a quién solicitarlo, desacoplando la relación existente entre el objeto invocador y el que procesará la petición.

El escenario:

Nos han asignado para desarrollar un nuevo sistema CRM y que dentro de su funcionalidad está la creación de órdenes de venta, las órdenes de venta serán objetos que representan los artículos que le venderemos a nuestro cliente y la información de nuestro cliente como su teléfono, dirección, información crediticia, etcétera.

Asumiremos que el sistema ya está implementado, pero nos han pedido que diseñemos un mecanismo que nos permita validar que la orden está correctamente armada.
Las reglas para la creación de una orden son las siguientes:

- El cliente deberá tener el estatus de Activo.
- Se validará la dirección del cliente dejando como obligatorio el campo *direccion1*, *cp*, *país*.
- Se validará el teléfono del cliente por lo cual se tendrá que tener la *lada* a tres dígitos, y el *teléfono* a siete dígitos.
- Validación de la línea de crédito del cliente, por lo que el total de la venta, más el saldo actual no deberá ser mayor que el límite de crédito del cliente.
- Se validarán las líneas de la orden asegurándonos de que la cantidad de productos sea mayor que cero, el precio de venta del producto no sea 20% menor al precio de lista.

Nuestro cliente sólo nos ha solicitado de momento estas reglas, sin embargo, nos indica que las órdenes son objetos muy complejos que pueden tener más cosas dentro y que de seguro podrían necesitar validaciones adicionales, es por eso que nos han pedido que diseñemos una solución que le permita realizar las validaciones y estar diseñado para agregar nuevas en un futuro próximo.

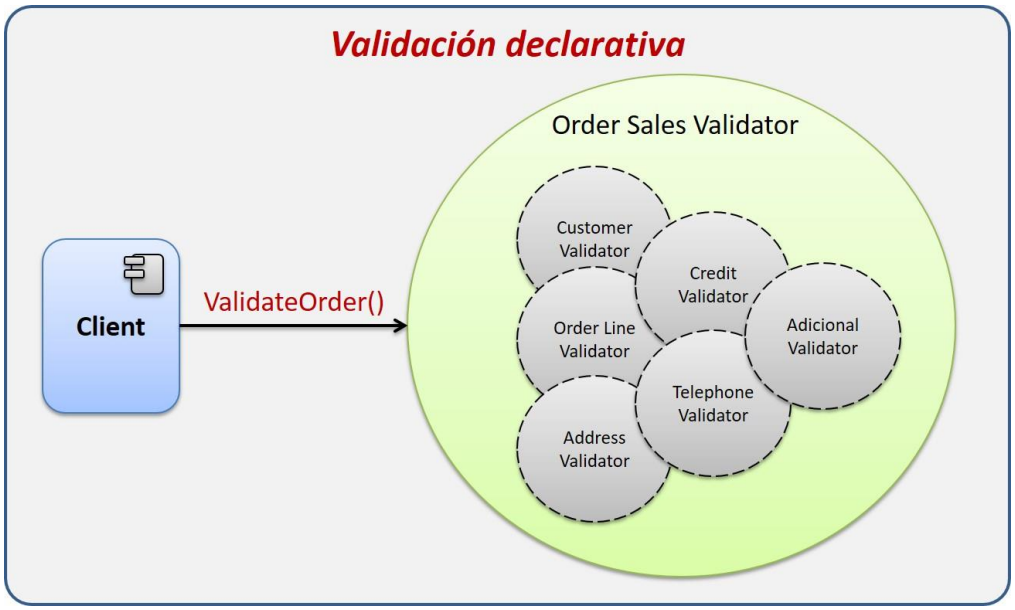


Ilustración 165: Validaciones de una orden.

Normalmente cuando creamos un objeto de validación, intentamos introducir todas las reglas dentro de este mismo objeto, en algunas ocasiones tenemos todo en un mismo método y en otras, para ser más ordenados, creamos distintos métodos para validar secciones de nuestro objeto como es el caso de la imagen anterior.

Aquí, un sólo objeto tiene todos los métodos necesarios para validar la orden, sin embargo, éste representa un problema ya que es una estructura estática y será necesario modificar la clase cada vez que una nueva validación es solicitada. Sin mencionar que las validaciones no son reutilizables. Ya que sólo funcionan para el contexto en el que fueron creadas.

La solución:

Como ya vimos, una orden es un objeto complejo que puede tener varias cosas dentro que necesitarán validación y todas estas reglas junto con las nuevas deberán ser soportadas sin tener un gran impacto sobre la solución.

Para darle un poco más de emoción a este ejemplo hemos decidido hacer una pequeña variación del patrón original y denominarlo *Tree of Responsability* (Árbol de Responsabilidad) el cual en lo único en que varía es que en vez de tener un sólo *Handler* como siguiente objeto a procesar el mensaje, esta variante nos permite tener varios *Handler* como siguiente objeto a procesar el mensaje. De tal forma que en lugar de tener una cadena, formamos un árbol de responsabilidades.

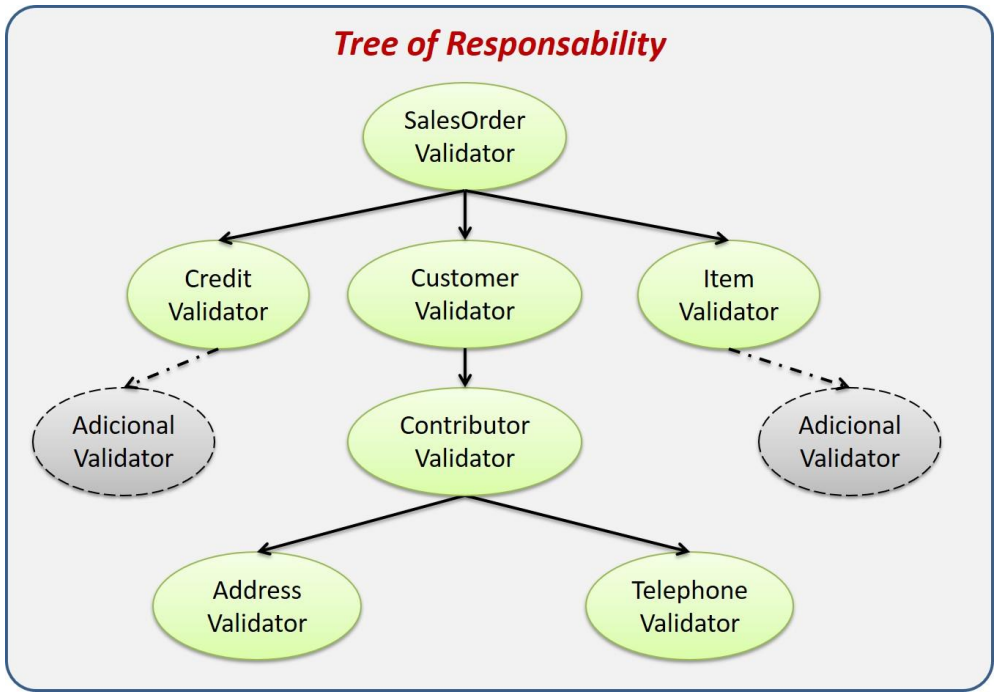


Ilustración 166: Estructura de la validación utilizando Tree of Responsability.

En la imagen vemos como hemos implementado una cadena de responsabilidad formando un Árbol, este árbol no sólo validará la orden sino que además se apoyará de cada Handler para validar una pequeña sección de la orden.

Originalmente el patrón pasa el mensaje por los *Handler* hasta que finalmente un pueda procesar **todo** el mensaje, sin embargo, en nuestra implementación cada Handler de la cadena validará una pequeña sección de la orden, de tal forma que cuando todo el árbol sea recorrido estaremos seguros de que la orden está perfectamente creada.

En la imagen vemos círculos verdes y grises, los verdes representan validaciones que el cliente ya nos solicitó y que tenemos que implementar, los grises representan *Handlers* nuevos que podemos encadenar en cualquier parte de la validación y así validar nuevas secciones de la orden de forma dinámica sin afectar lo que ya está construido.

La implementación:

Una vez que conocemos los requerimientos, pasaremos a la implementación de la solución.

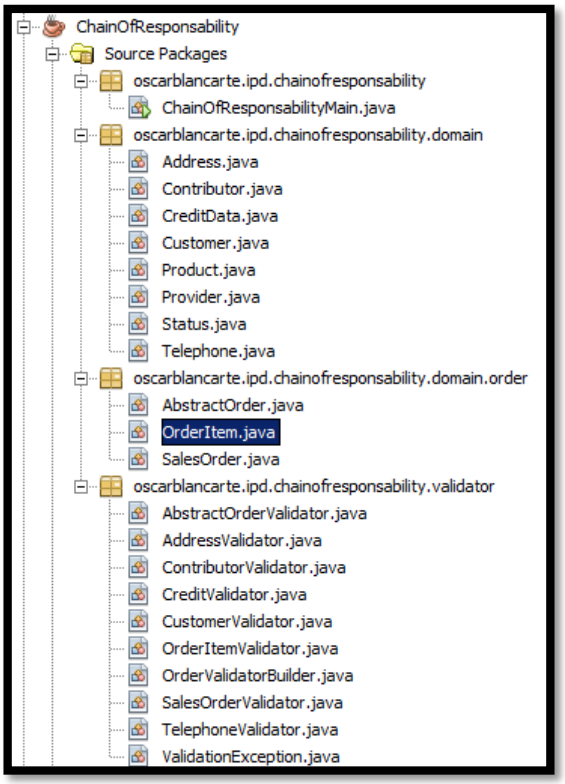


Ilustración 167: Estructura del proyecto ChainOfResponsability.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.chainofresponsability:** Paquete principal que contiene únicamente la clase principal del proyecto *ChainOfResponsabilityMain*.
- **oscarblancarte.ipd.chainofresponsability.domain:** Contiene las clases de dominio que serán necesarias para la creación de la orden.
- **oscarblancarte.ipd.chainofresponsability.domain.order:** Paquete que contiene las clases relacionadas con las órdenes.

- **oscarblancarte.ipd.chainofresponsability.validator:** Contiene las clases de validación que implementan el patrón *Chain of Responsability*.

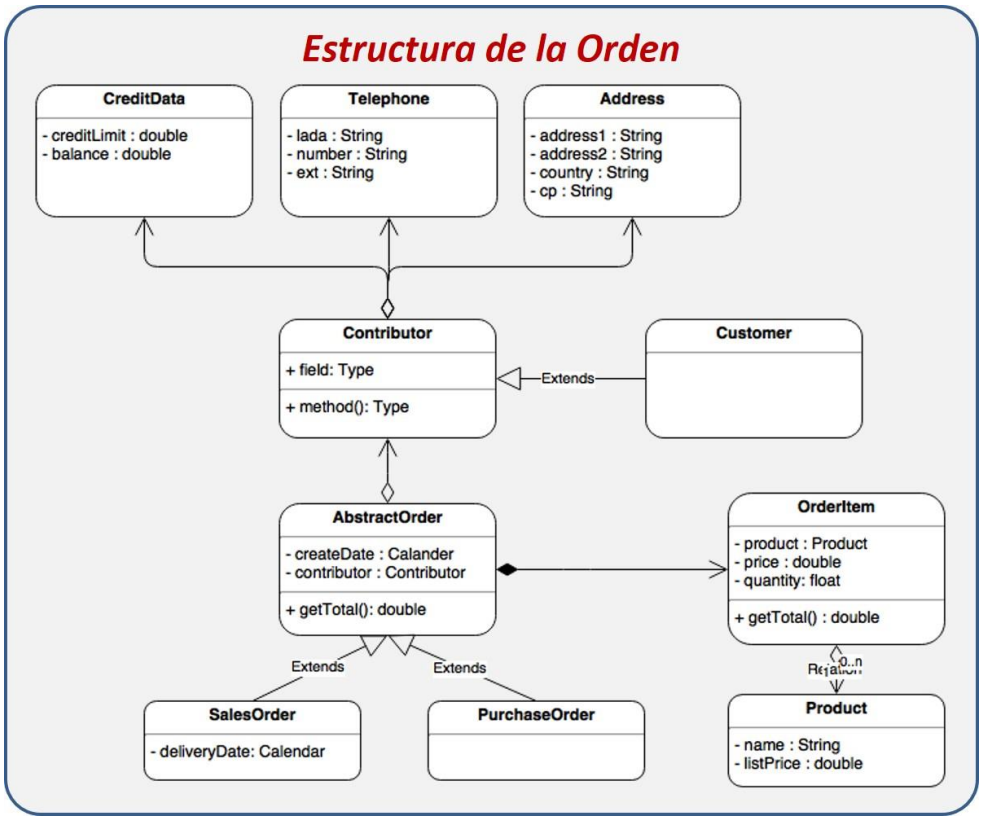


Ilustración 168: Diagrama de clases que modela la estructura de la Orden.

En la imagen podemos ver las clases que componen una Orden, este diagrama nos puede ayudar a entender cómo es que debe estar formada una Orden y nos da una idea de cómo es que debemos implementar las validaciones.

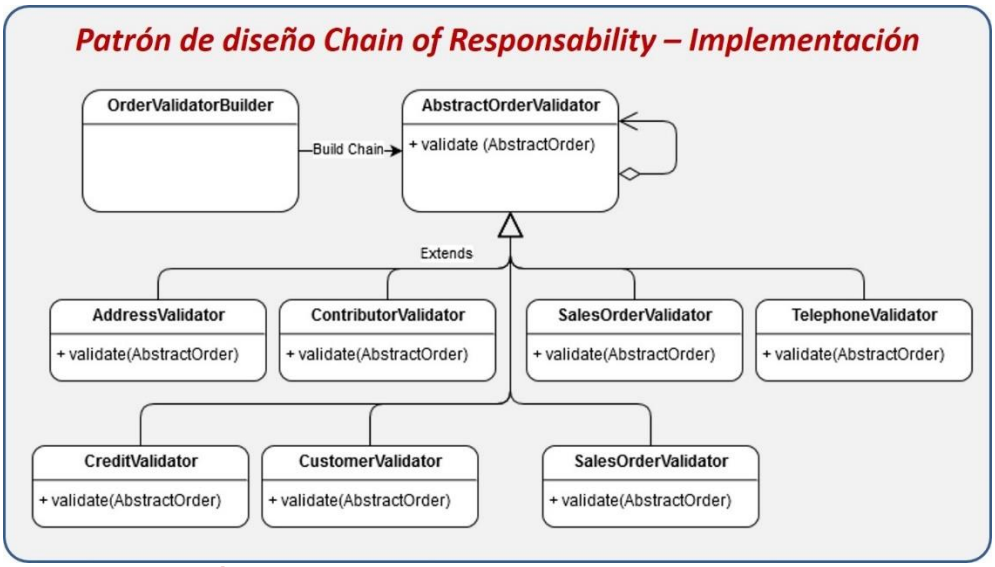


Ilustración 169: Diagrama que muestra los Handler del proyecto.

En la imagen podemos ver todas las clases que conforman el Árbol de responsabilidades, las clases que vemos aquí serán parte de la cadena de responsabilidad.

Clase Address:

Clase que corresponde a la dirección del cliente, se establece la dirección, el país y el código postal.

```
1.  package oscarblancarte.ipd.chainofresponsability.domain;
2.
3.  public class Address {
4.
5.      private String address1;
6.      private String address2;
7.      private String country;
8.      private String CP;
9.
10.     /** GET AND SET */
11. }
```

Clase Contributor:

Clase abstracta que sirve de base para crear clientes o proveedores, las propiedades que contiene son nombre (*name*), rfc, estatus (*status*), dirección (*address*), teléfono (*telephone*) e información de crédito (*creditData*).

```
1.  package oscarblancarte.ipd.chainofresponsability.domain;
2.
3.  public abstract class Contributor {
4.
5.      private String name;
6.      private String rfc;
7.      private Status status;
8.
9.      private Address address;
10.     private Telephone telephone;
11.     private CreditData creditData;
12.
13.     /** GET AND SET */
14. }
```

Clase CreditData:

Clase que contiene la información de crédito del cliente, las propiedades que contiene son límite de crédito (*creditLimit*) y saldo (*balance*).

```
1.  package oscarblancarte.ipd.chainofresponsability.domain;
2.
3.  public class CreditData {
4.
5.      private double creditLimit;
6.      private double balance;
7.
8.      /** GET AND SET */
9.  }
```

Clase Customer:

Clase que hereda de *Contributor* y representa a un cliente de la orden de venta. Como vemos, no agrega propiedades adicionales a las de la súper clase.

```
1. package oscarblancarte.ipd.chainofresponsability.domain;
2.
3. public class Customer extends Contributor{
4.
5. }
```

Clase Product:

Clase que corresponde a un producto, esta clase contiene el nombre del producto (*name*) y el precio de lista (*listPrice*).

```
1. package oscarblancarte.ipd.chainofresponsability.domain;
2.
3. public class Product {
4.     private String name;
5.     private double listPrice;
6.
7.     /** GET AND SET */
8. }
```

Enumeración Status:

Enumeración que contiene los *status* posibles que puede tener un cliente, los cuales son ACTIVO y BAJA.

```
1. package oscarblancarte.ipd.chainofresponsability.domain;
2.
3. public enum Status {
4.     ACTIVO,BAJA
5. }
```

Clase Telephone:

Clase que representa el teléfono del cliente. Las propiedades que contiene la clase son *lada*, número (*number*) y extensión (*ext*).

```
1. package oscarblancarte.ipd.chainofresponsability.domain;
2.
```

```
3. public class Telephone {
4.
5.     private String lada;
6.     private String number;
7.     private String ext;
8.
9.     /** GET AND SET */
10. }
```

Clase AbstractOrder:

Clase abstracta que sirve como base para crear distintos tipos de órdenes, en este ejemplo sólo crearemos órdenes de venta.

Las propiedades de la clase son fecha de creación (*createDate*), contribuidor (*contributor*) y lista de Orden Items (*orderItems*) correspondiente el detalle de la orden.

```
1. package oscarblancarte.ipd.chainofresponsability.domain.order;
2.
3. import java.util.Calendar;
4. import java.util.List;
5. import oscarblancarte.ipd.chainofresponsability.domain.Contributor;
6.
7. public abstract class AbstractOrder {
8.     private Calendar createDate;
9.     private Contributor contributor;
10.     private List<OrderItem> orderItems;
11.
12.     public Calendar getCreateDate() {
13.         return createDate;
14.     }
15.
16.     public double getTotal(){
17.         double total = 0d;
18.         for (OrderItem abstractOrderItem : orderItems) {
19.             total+=abstractOrderItem.getTotal();
20.         }
21.         return total;
22.     }
23.     /** GET AND SET */
24. }
```

Apreciemos que el método *getTotal* obtiene el total mediante la suma de los *OrderItem*.

Clase SalesOrder:

Clase que representa una orden de venta, la clase hereda de *AbstractOrder* y solamente define la propiedad fecha de entrega (*deliveryDate*).

```
1.  package oscarblancarte.ipd.chainofresponsability.domain.order;
2.
3.  import java.util.Calendar;
4.
5.  public class SalesOrder extends AbstractOrder{
6.      protected Calendar deliveryDate;
7.
8.      public Calendar getDeliveryDate() {
9.          return deliveryDate;
10.     }
11.
12.     public void setDeliveryDate(Calendar deliveryDate) {
13.         this.deliveryDate = deliveryDate;
14.     }
15.
16. }
```

Clase OrderItem:

Esta clase representa el detalle de la orden. Dentro del detalle de la orden tenemos la información relevante con respecto a los productos vendidos, el detalle está compuesto por las propiedades: producto (*product*), precio (*price*) y cantidad (*quantity*).

```
1.  package oscarblancarte.ipd.chainofresponsability.domain.order;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.Product;
4.
5.  public class OrderItem {
6.
7.      private Product product;
8.      private double price;
9.      private float quantity;
10.
```

```
11.     /** GET AND SET */
12.
13.     public void setPrice(double price) {
14.         this.price = price;
15.     }
16.
17.     public double getTotal(){
18.         return price*quantity;
19.     }
20. }
```

Clase AbstractOrderValidator:

Clase base utilizada para construir el resto de validadores. Esta clase representa al *AbstractHandler* por lo que contiene una lista de objetos de sí misma con la finalidad de crear una lista de encadenamiento.

La clase define los métodos:

- **Valídate:** Método abstracto que deberá ser implementado por las subclases para procesar el mensaje o enviarlo al siguiente en la cadena de procesamiento en caso de no ser la clase indicada para procesar ese mensaje.
- **addValidate:** Método utilizado para agregar más *Handler* a la clase y así armar la cadena de ejecución. Cada *handler* agregado incrementa la cadena, agregando nuevas posibilidades para procesar el mensaje.

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
6.
7.  public abstract class AbstractOrderValidator {
8.
9.      protected List<AbstractOrderValidator> validators = new ArrayList<>();
10.
11.     public abstract void validate(AbstractOrder order) throws ValidationException;
12.
13.     public void addValidator(AbstractOrderValidator validator){
14.         validators.add(validator);
15.     }
```

```
16. }
```

Clase AddressValidator:

Clase que hereda de *AbstractOrderValidator* y permite validar los datos de un domicilio mediante el método *validate* el cual es implementado para validar los siguientes puntos.

- *Addres1* obligatorio.
- CP obligatorio a 4 dígitos.
- País obligatorio.

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.Address;
4.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
5.
6.  public class AddressValidator extends AbstractOrderValidator{
7.
8.      @Override
9.      public void validate(AbstractOrder order) throws ValidationException{
10.         Address address = order.getContributor().getAddress();
11.         if(address.getAddress1()==null || address.getAddress1().length()==0){
12.             throw new ValidationException("La dirección 1 es obligatoria");
13.         }else if(address.getCP()==null || address.getCP().length()!=4){
14.             throw new ValidationException("El CP debe de ser de 4 dígitos");
15.         }else if(address.getCountry()==null || address.getCountry().length()==0){
16.             throw new ValidationException("El país es obligatorio");
17.         }
18.     }
19. }
```

En la línea 10 se valida que el domicilio1 no sea nulo y que el campo no esté vacío.
En la línea 13 se valida el código postal de igual manera que el domicilio.
Finalmente, en la línea 15 se valida que el país sea diferente de nulo y que no esté vacío.

Si cualquiera de la validaciones no es superada, se lanza una excepción de tipo *ValidationException*, la cual será propagada hasta el cliente para informarle la causa del error.

Clase ContributorValidator:

Clase que hereda de *AbstractOrderValidator* y será utiliza para validar los datos del cliente, para lo cual implementa el método *validate* que se encargará de validar lo siguiente:

- El estatus correcto (ACTIVO) (línea 14)
- Encadenar la validación al siguiente *Handler* de la cadena de responsabilidad con la finalidad de continuar la validación (línea 12).

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.Contributor;
4.  import oscarblancarte.ipd.chainofresponsability.domain.Status;
5.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
6.
7.  public class ContributorValidator extends AbstractOrderValidator{
8.
9.      @Override
10.     public void validate(AbstractOrder order) throws ValidationException {
11.         for(AbstractOrderValidator validator : validators){
12.             validator.validate(order);
13.         }
14.         Contributor contributor = order.getContributor();
15.         if(Status.ACTIVO != contributor.getStatus()){
16.             throw new ValidationException("El contribuyente no está activo");
17.         }
18.     }
19. }
```

Cualquier error producido durante la validación lanzará la excepción *ValidateException* la cual será propagada hasta el cliente.

Clase CreditValidator:

Clase que hereda de *AbstractOrderValidator* y será utilizada para validar el crédito del cliente evitando una compra que supere su límite establecido. El método *validate* es utilizado para validar los datos de crédito del cliente, tomando el precio total de la orden más el saldo actual del cliente (línea 12) y se compara contra el límite de crédito, de tal forma que si lo supera entonces arroja la excepción *ValidateException* (línea 14) la cual será propagada hasta el cliente.

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.CreditData;
4.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
5.
6.  public class CreditValidator extends AbstractOrderValidator{
7.
8.      @Override
9.      public void validate(AbstractOrder order)throws ValidationException {
10.         double total = order.getTotal();
11.         CreditData creditData = order.getContributor().getCreditData();
12.         double newBalance = creditData.getBalance() + total;
13.         if(newBalance > creditData.getCreditLimit()){
14.             throw new ValidationException("El monto de la orden excede el "
15.                     + "límite de crédito disponible");
16.         }
17.     }
18. }
```

Clase CustomerValidator:

Clase que hereda de *AbstractOrderValidator* y será utilizada para validar que el contribuyente de la orden sea un cliente, esto se debe a que la clase *Contributo* es la clase base para crear clientes y proveedores. El método *validate* se encarga únicamente de validar que el contribuyente de la orden es una instancia de *Customer* (línea 14) y pasa la orden a los siguientes *Handler* de la cadena de responsabilidad (línea 11).

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
```

```
3. import oscarblancarte.ipd.chainofresponsability.domain.Customer;
4. import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
5.
6. public class CustomerValidator extends AbstractOrderValidator{
7.
8.     @Override
9.     public void validate(AbstractOrder order) throws ValidationException{
10.         for(AbstractOrderValidator validator : validators){
11.             validator.validate(order);
12.         }
13.
14.         if(!(order.getContributor() instanceof Customer)){
15.             throw new ValidationException("El contribuyente no es un cliente");
16.         }
17.     }
18. }
```

Nuevamente, la excepción es propagada al cliente si una validación no es superada.

Clase OrderItemValidator:

Clase que hereda de *AbstractOrderValidator* y será utilizada para validar el detalle de la orden, de tal manera que cada producto agregado a la orden será validado por esta clase. El método *validate* se implementa para validar lo siguiente:

- La cantidad del producto por línea debe ser mayor que cero (línea 15).
- El precio del producto no supere en un 80% al precio de lista (línea 23).

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import java.util.List;
4.  import oscarblancarte.ipd.chainofresponsability.domain.Product;
5.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
6.  import oscarblancarte.ipd.chainofresponsability.domain.order.OrderItem;
7.
8.  public class OrderItemValidator extends AbstractOrderValidator {
9.
10.     @Override
11.     public void validate(AbstractOrder order) throws ValidationException {
12.         List<OrderItem> orderItems = order.getOrderItems();
13.         for (OrderItem item : orderItems) {
14.             Product product = item.getProduct();
15.             if (item.getQuantity() <= 0) {
```

```
16.             throw new ValidationException("El producto '"
17.                     + product.getName() + "' no tiene cantidad");
18.         }
19.
20.         double listPrice = item.getProduct().getListPrice();
21.         double price = item.getPrice();
22.         double priceLimit = listPrice - (listPrice*0.20d);
23.         if(price < priceLimit){
24.             throw new ValidationException("El precio del producto '"+
25.                     product.getName()+"' sobrepasa el límite permitido");
26.         }
27.     }
28. }
29. }
```

Clase TelephoneValidator:

Clase que hereda de *AbstractOrderValidator* y será utilizada para validar el teléfono del cliente, para lo cual se implementa el método *validate* que validará lo siguiente:

- Lada obligatoria a tres dígitos (línea 16).
- Número de teléfono obligatorio a siete dígitos (línea 14).

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.Telephone;
4.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
5.
6.  public class TelephoneValidator extends AbstractOrderValidator{
7.
8.      @Override
9.      public void validate(AbstractOrder order) throws ValidationException{
10.         Telephone tel = order.getContributor().getTelephone();
11.         if(null == tel){
12.             throw new ValidationException(
13.                     "El teléfono del contribuyente es obligatorio");
14.         }else if(tel.getNumber().length()!= 7){
15.             throw new ValidationException("Número de teléfono inválido");
16.         }else if(tel.getLada().length()!=3){
17.             throw new ValidationException("Lada inválida");
18.         }
19.     }
20. }
```

Clase SalesOrderValidator:

Clase que hereda de *AbstractOrderValidator* y permite validar el tipo de orden, en este caso se valida que la orden sea de venta, para lo cual se implementa el método *validate que* valida lo siguiente:

- El tipo de orden es de venta (línea 10).
- Encadena la validación del resto de la orden con los siguientes *Handlers* de la cadena. (línea 15).

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  import oscarblancarte.ipd.chainofresponsability.domain.order.AbstractOrder;
4.  import oscarblancarte.ipd.chainofresponsability.domain.order.SalesOrder;
5.
6.  public class SalesOrderValidator extends AbstractOrderValidator {
7.
8.      @Override
9.      public void validate(AbstractOrder order) throws ValidationException {
10.         if (!(order instanceof SalesOrder)) {
11.             throw new ValidationException("Se esperaba una orden de Venta");
12.         }
13.
14.         for(AbstractOrderValidator validator : validators){
15.             validator.validate(order);
16.         }
17.     }
18. }
```

Clase ValidationException:

Excepción *custom* que es lanzada cuando se encuentra algún error al realizar la validación de la orden con la finalidad de notificar al cliente de los errores encontrados.

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  public class ValidationException extends Exception {
4.
5.      public ValidationException(String message) throws ValidationException {
```

```
6.        super(message);
7.    }
8. }
```

Clase OrderValidatorBuilder:

Clase que utilizamos como utilidad para crear la estructura del Árbol de validación, en esta clase determinamos las validaciones que tendrán la orden y la secuencia en las que se ejecutarán. Los métodos definidos son:

- **buildContributorValidator:** Método utilizado para crear el árbol de validación del contribuyente, este método encadena las clases *AddressValidator* y *TelephoneValidator*.
- **buildCustomerValidator:** Método utilizado para crear el árbol de validación del cliente, este método encadena la clase *ContributorValidator* construida por el método *buildContributorValidator*.
- **buildSalesOrderValidator:** Método utilizado para crear el árbol de validaciones de la orden de venta, este método encadena las clases *OrderItemValidator*, *CreditValidator* y la clase *CustomerValidator* construido por el método *buildCustomerValidator*.

```
1.  package oscarblancarte.ipd.chainofresponsability.validator;
2.
3.  public class OrderValidatorBuilder {
4.      public static final AbstractOrderValidator buildSalesOrderValidator(){
5.          AbstractOrderValidator validator = new SalesOrderValidator();
6.          validator.addValidator(buildCustomerValidator());
7.          validator.addValidator(new OrderItemValidator());
8.          validator.addValidator(new CreditValidator());
9.          return validator;
10.     }
11.
12.     private static AbstractOrderValidator buildCustomerValidator(){
13.         AbstractOrderValidator validator = new CustomerValidator();
14.         validator.addValidator(buildContributorValidator());
15.         return validator;
16.     }
17.
18.     private static AbstractOrderValidator buildContributorValidator(){
19.         AbstractOrderValidator validator = new ContributorValidator();
20.         validator.addValidator(new AddressValidator());
21.         validator.addValidator(new TelephoneValidator());
```

```
22.         return validator;
23.     }
24. }
```

Clase ChainOfResponsabilityMain:

Finalmente, la clase principal del proyecto con la que probaremos la aplicación. En el método *main* hemos creado una orden con todos los valores validados exitosamente.

```
1.  package oscarblancarte.ipd.chainofresponsability;
2.
3.  import java.util.*;
4.  import oscarblancarte.ipd.chainofresponsability.domain.*;
5.  import oscarblancarte.ipd.chainofresponsability.domain.order.*;
6.  import oscarblancarte.ipd.chainofresponsability.validator.*;
7.
8.  public class ChainOfResponsabilityMain {
9.
10.     public static void main(String[] args) {
11.         SalesOrder salesOrder = new SalesOrder();
12.         salesOrder.setDeliveryDate(Calendar.getInstance());
13.         Customer customer = new Customer();
14.         customer.setName("Oscar Blancarte");
15.         customer.setRfc("XXX0000000X0");
16.         customer.setStatus(Status.ACTIVO);
17.
18.         Telephone phone = new Telephone();
19.         phone.setExt("123");
20.         phone.setLada("999");
21.         phone.setNumber("1234567");
22.         customer.setTelephone(phone);
23.
24.         Address address = new Address();
25.         address.setAddress1("Address 1");
26.         address.setAddress2("Address 2");
27.         address.setCP("1234");
28.         address.setCountry("México");
29.         customer.setAddress(address);
30.
31.         CreditData creditData = new CreditData();
32.         creditData.setBalance(1000);
33.         creditData.setCreditLimit(13000);
34.         customer.setCreditData(creditData);
35.
36.         salesOrder.setContributor(customer);
37.
38.         List<OrderItem> orderItems = new ArrayList<OrderItem>();
39.         for (int c = 0; c < 10; c++) {
```

```
40.             OrderItem item = new OrderItem();
41.             item.setPrice((c + 1) * 30);
42.             Product product = new Product();
43.             product.setListPrice((c + 1) * 32);
44.             product.setName("Product " + (c + 1));
45.             item.setProduct(product);
46.             item.setQuantity(c + 1);
47.             orderItems.add(item);
48.         }
49.         salesOrder.setOrderItems(orderItems);
50.         System.out.println("Total orden > " + salesOrder.getTotal());
51.         try {
52.             AbstractOrderValidator validator
53.                     = OrderValidatorBuilder.buildSalesOrderValidator();
54.             validator.validate(salesOrder);
55.             System.out.println("Validación exitosa");
56.         } catch (Exception e) {
57.             System.err.println(e.getMessage());
58.             e.printStackTrace();
59.         }
60.
61.     }
62. }
```

Las secciones más importantes son las siguientes:

- De las líneas 13 a la 16 creamos el cliente.
- De las líneas 18 a la 22 creamos el teléfono del cliente.
- De las líneas 24 a la 29 creamos la dirección del cliente.
- De las líneas 31 a la 34 creamos los datos de crédito del cliente.
- De las líneas 38 a la 47 creamos el detalle de la Orden.
- De las líneas 42 a la 44 creamos los productos.
- En las línea 52 creamos el validador de órdenes mediante la clase *OrderValidatorBuilder*.
- Finalmente, en la línea 54 realizamos la validación de la orden; de salir algún error este es manejado en el bloque *catch*.

La ejecución:

Una vez que comprendemos como está conformado el proyecto y hemos implementado todas las clases, podremos iniciar con la ejecución del proyecto, para lo cual ejecutaremos la clase *ChainOfResponsabilityMain* para obtener el siguiente resultado:

```
run:
Total orden > 11550.0
Validación exitosa
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 170: Resultado de le ejecución de la clase ChainOfResponsabilityMain.

Si ejecutamos la clase tal y como está, no tendremos ningún problema en validar la Orden debido a que todos los datos están correctamente establecidos, sin embargo, realicemos un pequeño ajuste al código y modifiquemos la siguiente línea de código:

```
24. phone.setLada("9991");
```

Modificamos la lada del cliente agregándole un dígito extra y ejecutamos la aplicación para obtener el siguiente resultado:

```
run:
Total orden > 11550.0
Lada invalida
oscarblancarte.ipd.chainofresponsability.validator.ValidationException: Lada invalida
        at oscarblancarte.ipd.chainofresponsability.validator.TelephoneValidator.validate(TelephoneValidator.java:21)
        at oscarblancarte.ipd.chainofresponsability.validator.ContributorValidator.validate(ContributorValidator.java:16)
        at oscarblancarte.ipd.chainofresponsability.validator.CustomerValidator.validate(CustomerValidator.java:15)
        at oscarblancarte.ipd.chainofresponsability.validator.SalesOrderValidator.validate(SalesOrderValidator.java:19)
        at oscarblancarte.ipd.chainofresponsability.ChainOfResponsabilityMain.main(ChainOfResponsabilityMain.java:58)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 171: Error al validar la Lada.

Como vemos, la Lada del cliente no ha pasado la regla de validación de tres dígitos.

Para realizar una prueba regresamos la línea anterior como estaba y modificaremos la siguiente línea:

```
37. creditData.setCreditLimit (10000);
```

Disminuimos el límite de crédito del cliente a 10,000 y ejecutamos la aplicación nuevamente para ver el siguiente resultado.

```
run:
Total orden > 11550.0
El monto de la orden excede el límite de crédito disponible
oscarblancarte.ipd.chainofresponsability.validator.ValidationException: El monto de la orden excede el límite de crédito disponible
        at oscarblancarte.ipd.chainofresponsability.validator.CreditValidator.validate(CreditValidator.java:18)
        at oscarblancarte.ipd.chainofresponsability.validator.SalesOrderValidator.validate(SalesOrderValidator.java:19)
        at oscarblancarte.ipd.chainofresponsability.ChainOfResponsabilityMain.main(ChainOfResponsabilityMain.java:58)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 172: Error al validar el límite de crédito.

La aplicación nos ha lanzado el error: *'El monto de la orden excede el límite de crédito disponible'*, este error es porque establecimos el saldo del cliente en 1,000 (balance) y el límite de crédito a 10,000
El monto de la orden es de 11,550 más el saldo actual da un total de 12,550, lo cual es mayor que el límite de crédito del cliente.

Para finalizar realizaremos una prueba más, nuevamente regresamos el código como estaba originalmente y modificamos la siguiente línea:

```
50. item.setQuantity(0);
```

Modificamos la cantidad de los productos a cero y ejecutamos la aplicación para obtener el siguiente resultado:

```
run:
Total orden > 0.0
El producto 'Product 1' no tiene cantidad
oscarblancarte.ipd.chainofresponsability.validator.ValidationException: El producto 'Product 1' no tiene cantidad
        at oscarblancarte.ipd.chainofresponsability.validator.OrderItemValidator.validate(OrderItemValidator.java:21)
        at oscarblancarte.ipd.chainofresponsability.validator.SalesOrderValidator.validate(SalesOrderValidator.java:19)
        at oscarblancarte.ipd.chainofresponsability.ChainOfResponsabilityMain.main(ChainOfResponsabilityMain.java:58)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 173: Error al validar la cantidad del producto.

Al ejecutar la aplicación nos marca un error, el producto 1 no tiene cantidad debido a que establecimos la cantidad de los productos a cero.

Podríamos seguir realizando combinaciones pero se podría extender mucho más este capítulo por lo que nos tocará a nosotros seguir realizando más pruebas desde nuestra computadora y validar los resultados.

Siguientes pasos:

Hemos comprobado que el patrón *Chain of Responsability* nos permite crear estructuras dinámicas que delegan la responsabilidad al siguiente *Handler* de la cadena hasta que un *handler* adecuado es capaz de procesar el mensaje.
También vimos que este tipo de problemas sería muy complicado de resolver mediante la herencia y nos obligaría a crear estructuras rígidas que no podrían ser creadas dinámicamente.

Para fortalecer nuestro aprendizaje realizaremos los siguientes ejercicios:

- Creemos un nuevo tipo de orden, esta orden será de Compra, las compras no tienen cliente por lo que tendremos que crear una clase proveedor en la cual definiremos las propiedades que consideremos necesarias. Y finalmente crearemos un nuevo Handler que permite validar nuestro nuevo tipo de orden. **Nota:** debemos reutilizar lo más posible los validadores existentes.

- Los validadores actualmente terminan su ejecución cuando un error es encontrado, en ese momento el cliente es informado del error. Por lo que modificaremos las clases para que en lugar de regresar el primer error, éstos sean almacenados para ser devueltos todos los errores al usuario. Se deberá crear un método que le permita al cliente obtener los errores encontrados durante la validación, en caso de no existir se regresaría un *null* en otro caso la lista de errores encontrados.

- En este ejemplo hemos creado las cadenas de responsabilidad a través de la clase de utilidad *OrderValidatorBuilder*, sin embargo, debemos diseñar un XML que permita configurar la forma en que las cadenas de responsabilidad se deberían de generar con la finalidad de crear un validador totalmente configurable. Adicionalmente deberemos crear una clase *builder* que interprete el XML y nos regrese el Validador correctamente formado.

Patrón Interpreter

El patrón de diseño interpreter es utilizado para evaluar un lenguaje definido como Expresiones, este patrón nos permite interpretar un lenguaje como Java, C#, SQL o incluso un lenguaje inventado por nosotros el cual tiene un significado; y darnos una respuesta tras evaluar dicho lenguaje.
Interpreter es uno de los patrones de diseño más complejos debido a que para su funcionalidad debe combinar técnicas de programación orientada a objetos avanzada y su interpretación puede ser algo confusa, las principales cosas con las que nos enfrentaremos son la Herencia, Polimorfismo y la Recursividad.

Es importante entender qué es un lenguaje y no me refiero a entender el código si no a entender cómo es que un lenguaje se define, un lenguaje es definido mediante una nomenclatura en formato **Backus-Naur form** (BNF) la cual trata de expresar un lenguaje de una forma simple y precisa.

Más adelante entraremos en detalle pero por ahora nos basta con saber que un lenguaje es un conjunto de expresiones que tienen una gramática definida las cuales, colocadas en cierto orden, tiene un significado. Estas expresiones deberán ser reutilizables y serán analizadas mediante algoritmos recursivos.

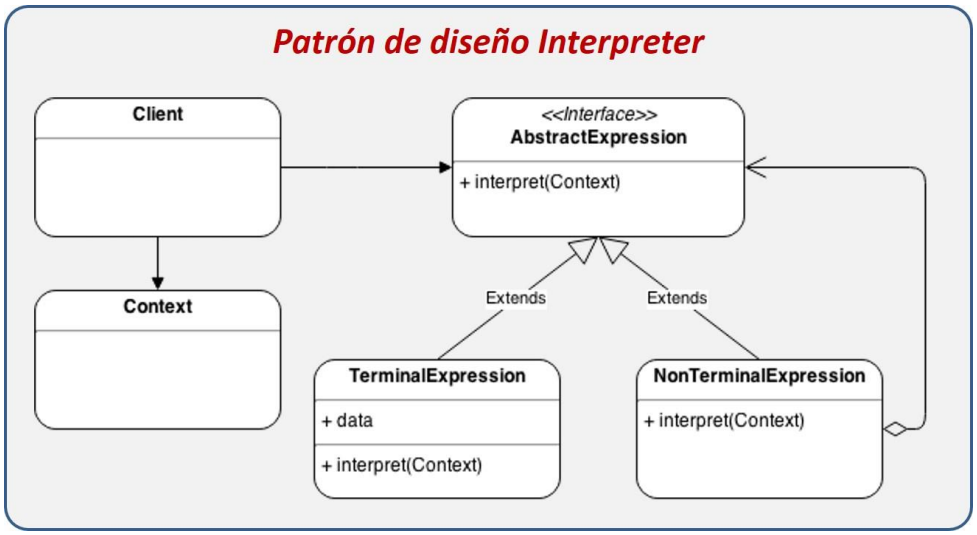


Ilustración 174: Estructura del patrón de diseño Interpreter.

Los componentes del patrón *Interpreter* se explican a continuación:

- **Client:** Actor que dispara la ejecución del *interpreter*.
- **Context:** Objeto con información global que será utilizada por el intérprete para leer y almacenar información global entre todas las clases que conforman el patrón, este es enviado al interpreter el cual lo replica por toda la estructura.
- **AbstractExpression:** Interface que define la estructura mínima de una expresión.
- **TerminalExpression**: Se refiere a expresiones que no tienen más continuidad y al ser evaluadas o interpretadas terminan la ejecución de esa rama. Estas expresiones marcan el final de la ejecución de un sub-árbol de la expresión.
- **NonTerminalExpression:** Son expresiones compuestas y dentro de ellas existen más expresiones que deben ser evaluadas. Estas estructuras son interpretadas utilizando recursividad hasta llegar a una expresión Terminal.

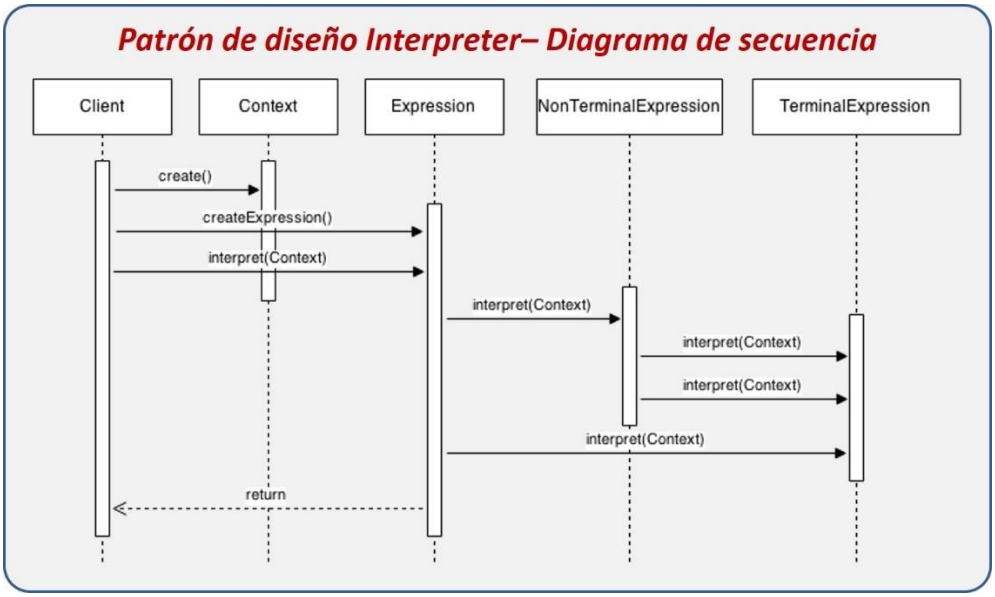


Ilustración 175: Diagrama de secuencia del patrón Interpreter.

El diagrama se interpreta de la siguiente manera:

1. El cliente crea el contexto para la ejecución del *interpreter*.
2. El cliente crea u obtiene la expresión a evaluar.
3. El cliente solicita la interpretación de la expresión al interpreter y le envía el contexto.
4. La Expresión manda llamar a las Expresiones No Terminales que contiene.
5. La Expresión No Terminal manda llamar a todas las Expresiones Terminales.
6. La Expresión Raíz solicita la interpretación de una Expresión Terminal.
7. La expresión se evalúa por completo y se tiene un resultado de la interpretación de todas las expresiones terminales y no terminales.

Un punto importante a resaltar es que las expresiones pueden ser tan complejas como queramos y que una expresión no terminal puede tener dentro varias expresiones no terminales y que éstas a su vez también puedan tener otras expresiones no terminales y así sucesivamente hasta llegar a todas a las expresiones terminales que nos indican que la rama ha llegado a su fin.

Cuándo utilizarlo:

- Cuando, dada una gramática, nos ayuda a generar expresiones que representan al lenguaje, junto con un *interpreter* que es capaz de darle un significado a dicha gramática y arrojar un resultado tras su interpretación.

- Utilizada como *interpreters* en tiempo de ejecución como la máquina virtual de Java o JavaScript.

El escenario:

Imaginemos que se nos ha ocurrido crear una nueva base de datos que tenga como forma de persistencia un archivo de Excel, este archivo deberá ser leído mediante consultas SQL, por lo que debemos crear un mecanismo que nos ayude a interactuar con el Excel.
Lo primero será definir el lenguaje a utilizar, lo segundo será descomponer la consulta en Expresiones y tercero, tener un mecanismo que nos ayude a interpretar las Expresiones y que nos arroje los resultados.

Con la finalidad de desarrollar el ejemplo de forma clara y simple nos enfocaremos únicamente en desarrollar las consultas del *SELECT* dejando de tarea desarrollar el resto de operaciones como *UPDATE* y *DELETE*.

Para empezar, debemos definir nuestro lenguaje que nos ayudará a interpretar nuestras Expresiones, para esto utilizaremos la nomenclatura BFN y quedaría de la siguiente forma:

```
1.  <select>      ::= <target> <from> <where>
2.  <target>      ::= "select" <literal>?
3.  <from>        ::= "from" <literal>
4.  <where>       ::= "where" <statement>
5.  <statement>   ::= <and> | <or> | <boolExp>
6.  <and>         ::= <statement> "and" <statement>
7.  <or>          ::= <statement> "or" <statement>
8.  <boolExp>     ::= <literal> <mathOp> <literal>
9.  <literal>     ::= <date> | <numeric> | <text>
10. <date>        ::= //Date Formate regex
11. <numeric>     ::= //Numeric Formate regex
12. <text>        ::= .? //Any entry
13. <mathOp>      ::= "=" | "!=" | "<>" | "<" | "<=" | ">" | ">="
```

Este lenguaje puede resultar enredoso pero en realidad es muy simple de interpretar; veamos que, del lado izquierdo, tenemos el nombre de todas las expresiones que utilizaremos para identificarlas y reutilizarlas, del lado derecho tenemos la definición de la expresión, por ejemplo:

```
1. <select>      ::= <target> <from> <where>
```

Expresión llamada ***select*** la cual está formada de la expresión ***<target>***, seguida de la expresión ***<form>*****,** seguida de la expresión ***<where>*****.**

```
1. <target>    ::= "select" <literal>?
```

Expresión llamada ***target*** la cual está formada de la palabra **"*select*",** seguida de una o más expresiones: **<literal>** (? = 1 o más)**.**

```
1. <literal>    ::= <date> | <numeric> | <text>
```

Expresión llamada literal la cual está formada por la Expresión ***<date>*** o ***<numeric>*** o ***<text***> pero nunca más de una. (| = OR).

No entraremos en detalle para no extender mucho este tema ya que con los ejemplos explicados es suficiente para comprender la interpretación del resto del lenguaje.

Definido el lenguaje podrías crear expresiones como las siguientes:

1. SELECT ID,NAME,ADDRESS FROM EMPLOYESS WHERE ID > 10
2. SELECT NAME FROM EMPLOYESS
3. SELECT ID,NAME FROM EMPLOYESS WHERE ID > 10 OR NAME = 'Oscar' AND LASTA_NAME <> 'Blancarte'

Si validamos las expresiones contra nuestro lenguaje veremos que son perfectamente válidas.

NOTA: Un paso importante para la conversión del lenguaje (consultas SQL) a expresiones es desarrollar un analizador Léxico y Sintáctico que valide que lo expresado cumpla perfectamente la gramática de nuestro lenguaje, como resultado de este proceso podemos obtener el **Árbol Sintáctico Abstracto** el cual es el que necesitamos para el patrón *Interpret*.

Los pasos de generación del analizador léxico y semántico están fuera del alcance de este libro debido a que son temas bastantes extensos.

Para esto existen libros y herramientas que nos pueden ayudar a entenderlos y construir el código a partir de la nomenclatura BNF como lo son JavaCC (*Java Compiler Compiler*) el cual está disponible en la página https://javacc.java.net/, también, podemos basarnos en libros como *Compiladores - principios, técnicas y herramientas* de los autores Alfred V. Aho, Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman.

Para este ejemplo partiremos del Árbol Sintáctico Abstracto.

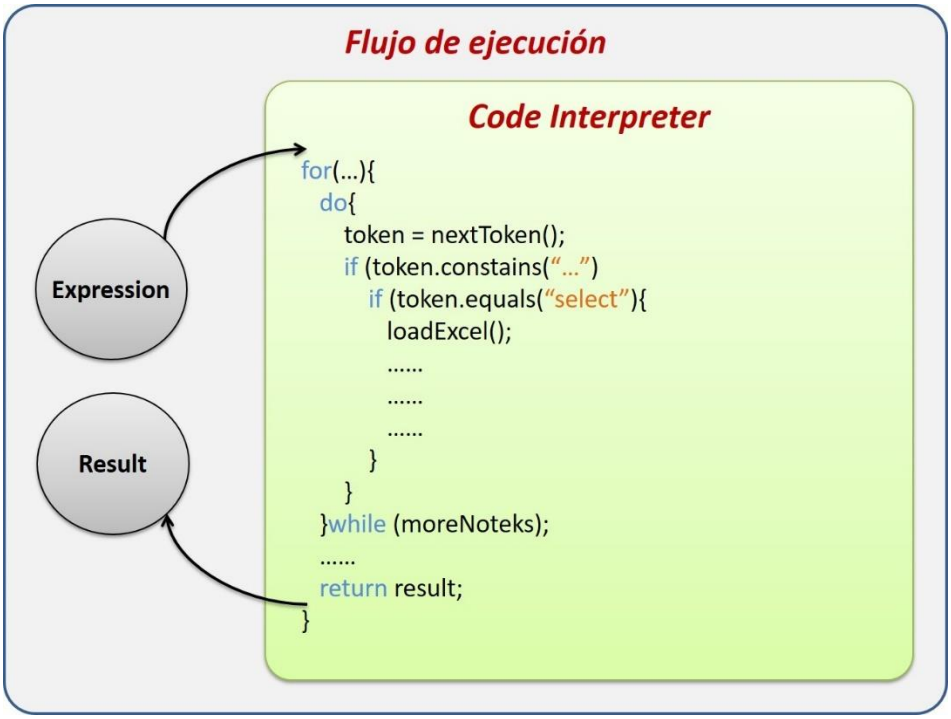


Ilustración 176: Implementación de intérpretes sin utilizar expresiones.

En la imagen se aprecia cómo es que generalmente se implementa un mal intérprete, el cual es un conjunto de código que va recorriendo la expresión original (SELECT SQL) y al mismo tiempo la va interpretando. Este diseño tiene un gran problema: hace un código sumamente complejo y que es difícilmente mantenible por ello, hay que modificarlo con "pinzas" para no afectar lo que ya está construido. Al final de la ejecución tienes el resultado pero en realidad nunca sabemos con precisión la secuencia que llevó a su resolución.

La solución:

Como ya vimos, el patrón *interprete* nos sirve para interpretar un lenguaje que puede ser muy simple o muy complejo, además vimos que una solución con un código sin un Árbol sintáctico abstracto es algo muy complicado, por lo cual decidiremos implementar uno de tal forma que cada expresión sea representada por una clase y que esa sea polimórficamente capaz de interpretar el lenguaje.

Analicemos las siguientes consultas:

```
1. SELECT ID, BORN_DATE, DEPARTMENT, FIRST_NAME, LAST_NAME FROM EMPLOYEES WHER
   E BORN_DATE <> '10/20/1980'
```

Y esta otra:

```
1. SelectExpression select = new SelectExpression(
2. new TargetExpression(
3.     new LiteralExpression("ID"),
4.     new LiteralExpression("BORN_DATE"),
5.     new LiteralExpression("DEPARTMENT"),
6.     new LiteralExpression("FIRST_NAME"),
7.     new LiteralExpression("LAST_NAME")
8. ),
9.     new FromExpression(
10.      new LiteralExpression("EMPLOYEES")),
11.      new WhereExpression(
12.        new BooleanExpression(
13.        new LiteralExpression("BORN_DATE"),
14.        new LiteralExpression("!="),
15.        new DateExpression("10/20/1980")
16.      )
17. )
18. );
```

La primera representa una consulta expresada en gramática del lenguaje SQL la cual no es difícil de interpretar para nosotros (ojo humano).

La segunda representa exactamente la misma consulta pero está expresada mediante un Árbol Sintáctico Abstracto y cada selección de la consulta es una Expresión. Si vemos a detalle esta estructura, forma un Árbol donde la raíz es la expresión ***Select*** y las ramas son todas las expresiones terminales, a primera vista (ojo humano) esto es mucho más difícil de interpretar debido a la gran

cantidad de objetos que son necesarios crear y anidarlos correctamente para que la consulta tenga sentido. Pero la pregunta aquí es ¿qué crees que sea más sencillo de interpretar para un programa?
Veámoslo así, para un programa la consulta SQL es sólo un *string*, no sabe dónde inicia una columna y dónde están las condiciones, para lo cual tiene que empezar a evaluar toda la consulta, clasificando los *tokens* y descartar consultas inválidas, y sobre todo esto, interpretar y dar un resultado, en cambio con un Árbol Sintáctico Abstracto ya todo está clasificado, cada elemento de la expresión tiene un sentido claro para el programa, de antemano éste sabe exactamente que hay en cada objeto, sabe si se trata de una tabla, una expresión o una columna, además de que el analizador léxico, sintáctico y semántico ya se aseguraron de que la consulta SQL original estuviera perfectamente armada.

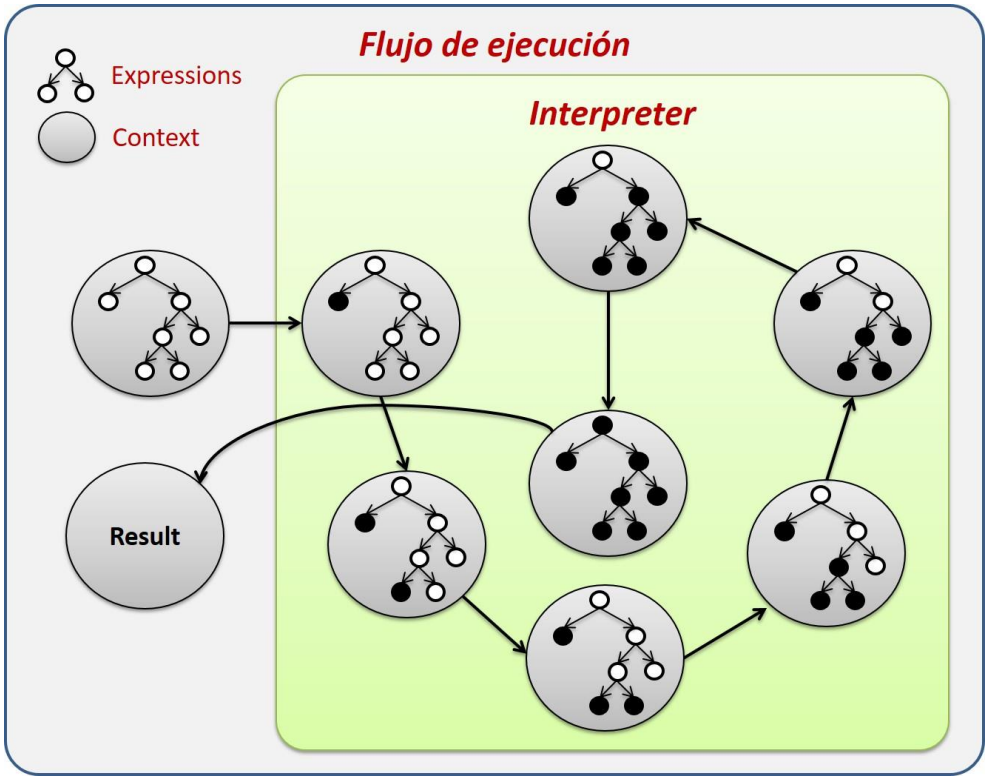


Ilustración 177: Interpretación con un Árbol Sintactico Abstracto.

La imagen anterior ilustra cómo es que el intérprete va interpretando paso a paso cada sección del Árbol Sintáctico Abstracto hasta llegar a una respuesta.

El Árbol es interpretado de forma recursiva iniciando con la Raíz pero procesándola al final, es decir se realiza un recorrido Post-Order.
Cada expresión será resuelta cuando todas las expresiones que la componen sean resueltas. Como resultado de la evaluación de la expresión se obtendrá un resultado, el cual puede ser un valor o una acción, en nuestro caso esperamos las filas que correspondan con los criterios de búsqueda.

La implementación:

Una vez que conocemos lo que tenemos que hacer es hora de empezar con la implementación, para lo cual empezaremos analizando el proyecto *Interpreter*.

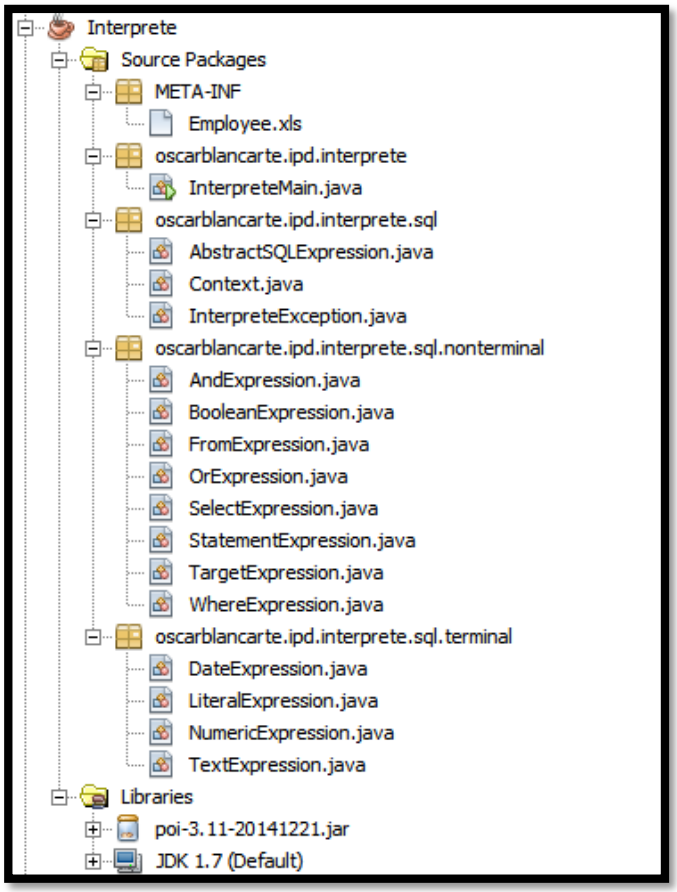


Ilustración 178: Estructura del proyecto Interpreter.

En el proyecto podemos ver los siguientes paquetes:

- **META-INF:** Folder donde se encuentra el archivo de Excel que utilizaremos como base de datos.
- **Oscarblancarte.ipd.interprete:** Paquete principal del proyecto el cual tiene únicamente la clase principal del proyecto *InterpreterMain*.

- **Oscarblancarte.ipd.interprete.sql:** Contiene las clases base para crear el *interpreter* y su contexto.
- **Oscarblancarte.ipd.interprete.sql.nonterminal:** Paquete en el que se encuentran las expresiones no terminales.
- **Oscarblancarte.ipd.interprete.sql.terminal:** Paquete con expresiones terminales.
- **Librerías:** Este proyecto hace uso de la librería Apache POI la cual es utilizada para leer los archivos de Excel, la librería la puedes encontrar en el paquete libs en la raíz del proyecto.

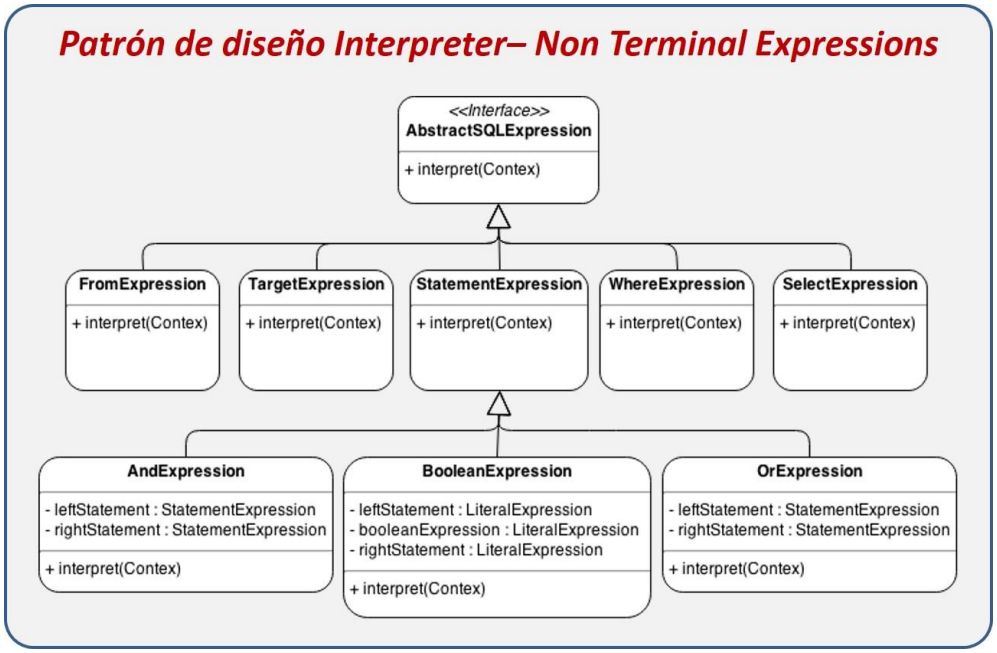


Ilustración 179: Diagrama de clases de las Expresiónes no terminales.

Las expresiones no terminales son aquellas que están compuestas de otras expresiones, por ejemplo: *SELECT NAME FROM EMPLOYEE*.
SELECT es una expresión no terminal ya que está representada por la clase *SelectExpresion* la cual es un compuesto de *TargetExpression*, *FromExpresion* y *WhereExpresion*.

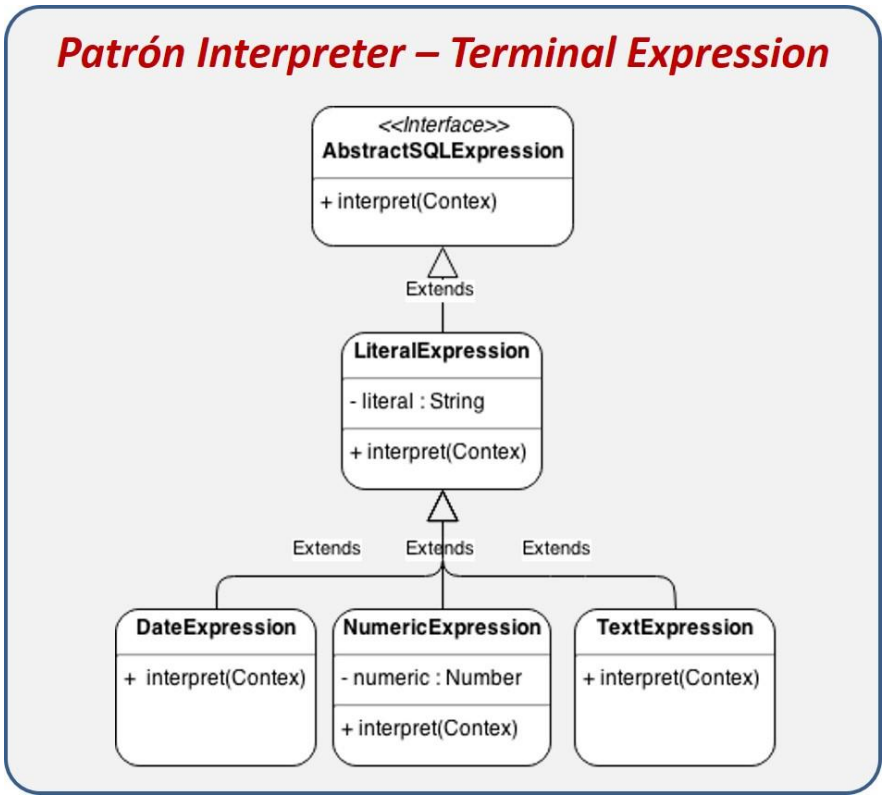


Ilustración 180: Diagrama de clases de las expresiones terminales.

Las expresiones terminales son aquellas donde el árbol de expresiones termina, no precisamente que termine todo el árbol, sino que termine una rama. Por ejemplo *SELECT NAME FROM EMPLOYEE, NAME y EMPLOYEE* son terminales porque son representadas por la Expresión *LiteralExpresión* la cual no tiene más ramificaciones.

Lo primero que haremos será analizar el Excel que utilizaremos como base de datos.

	A	B	C	D	E	F	G
1	ID	FIRST_NAME	LAST_NAME	DEPARTMENT	BORN_DATE	STATUS	TELEPHONE
2	1	Oscar	Blancarte	Sistemas	10/20/1981	Activo	1234567890
3	2	Javier	Iturralde	Contabilidad	10/20/1990	Baja	2345678901
4	3	Liliana	Castro	Sistemas	10/20/1960	Activo	3456789012
5	4	Juan	Perez	Finanzas	10/20/1985	Activo	4567890123
6	5	Manuel	Benitez	Compras	10/20/2001	Activo	5678901234
7	6	Adrian	Garcia	Ventas	10/20/1992	Baja	6789012345
8	7	Pablo	Picapiedra	Contabilidad	10/20/1954	Activo	7890123456
9	8	Pedro	Wilson	Finanzas	10/20/1923	Activo	8901234567
10	9	Rebeca	de la Torre	Ventas	10/20/1997	Baja	9012345678
11	10	Iridiam	Galindo	Call center	10/20/1921	Activo	1123456789
12	11	Fernanda	Lopez	Sistemas	10/20/1956	Activo	2234567890
13	12	Gabriela	Rios	Contabilidad	10/20/1965	Activo	3345678902
14	13	Mauricio	Olvera	Contabilidad	10/20/1978	Baja	4456789012
15	14	Fernando	Buelna	Sistemas	10/20/1987	Activo	5567890123
16	15	Wendy	Solano	Call center	10/20/1990	Baja	6678901234

EMPLOYEES Hoja2 Hoja3

Ilustración 181: Contenido del Excel Employee.xls.

Tomemos en cuenta lo siguiente:

- La primera fila es utilizada para poner los nombres de las columnas, el intérprete las analizará para determinar las columnas disponibles, de tal forma que si intentamos buscar una columna que no exista nos arrojará un error, también tiene la ventaja de que no importa en qué posición esté la columna ya que será analizada en tiempo de ejecución.
- El nombre de la Hoja será utilizado como el nombre de la Tabla, de tal forma que la hoja 1 es la tabla de Empleados (*EMPLOYESS*) y de no encontrarse nos arrojará un error.
- Para este ejemplo no se soportan valores Null o vacíos por lo que debemos dejar todos los campos con un valor determinado.
- Las fechas deberán estar siempre con el mismo formato.
- Cada fila del Excel representa un registro como en las bases de datos relacionales.

Clase InterpreteException:

Excepción personalizada que utilizaremos para lanzar cualquier error producido durante la interpretación de las Expresiones.

```
1. package oscarblancarte.ipd.interprete.sql;
2.
3. public class InterpreteException extends Exception{
4.
5.     public InterpreteException(String message) {
6.         super(message);
7.     }
8. }
```

Interface AbstractSQLExpression:

Interface que define la estructura mínima de las expresiones terminales y no terminales. En ella se define el método *interpret* que es utilizado para realizar la interpretación de la expresión. Para realizar la tarea requiere que se le envié el *Context* y pueda lanzar la excepción *InterpreteException* si encuentra algún problema durante la interpretación.

```
1.  package oscarblancarte.ipd.interprete.sql;
2.
3.  public interface AbstractSQLExpression {
4.      public Object interpret(Context context) throws InterpreteException;
5.  }
```

Clase Context:

Esta clase contiene información global para todo el *Interpreter*, en ella se almacena la información que todas las expresiones comparten, en nuestro caso se almacena lo siguiente:

- **woorkbook:** Clase que representa al archivo de Excel y es parte del API de Apache POI.

- **sheet:** Representa una hoja del Excel y la clase también es parte del API de Apache POI.
- **dateFormat:** Formato de fecha que se utilizará para leer las fechas del Excel.
- **columns**: Arreglo de *strings* con los nombres de las columnas colocados en orden.
- **dbPath:** URL donde se encuentra el archivo de Excel.
- **tableIterator:** Iterador utilizado para iterar sobre las filas del Excel.
- **resultRow:** Lista con todas las filas que pasarán las reglas de filtrado.
- **resultColumnCount:** Indica el número de columnas que se espera como respuesta.
- **currentRow:** Fila que se está procesando es este momento, a medida que las filas del Excel son procesadas esta variable se va modificando con la fila siguiente.
- **result:** Lista con todos los registros a devolver como resultado de la interpretación de la expresión, esta lista sólo contiene las columnas solicitadas.

Los métodos definidos son los siguientes:

- **createResultArray:** Método que inicia el arreglo de resultados y establece el número de columnas a regresar, el número de columnas es calculado mediante el análisis del archivo de Excel.
- **getDateFormat y setDateFormat:** Método para el establecimiento y obtención del formato de fechas.
- **getResultArray:** Método que regresa el resultado final de la evaluación de toda la expresión.
- **createRow:** Cuando un registro es candidato a ser devuelto como resultado de la evaluación del *where*, un nuevo registro es creado en los resultados finales, este método crea un arreglo del mismo tamaño de las columnas esperadas para la respuesta y lo agrega al arreglo de resultados.

- **getResultRow:** Este método regresa las filas que serán devueltas como resultado, sin embargo, éste contiene toda la fila sin filtrar las columnas que no son esperadas como resultado.
- **initiateFileRead:** Carga el archivo de Excel desde el directorio especificado.
- **cretateRowIterator:** Crea un iterador que permite recorrer el Excel fila a fila.
- **getRowIterator:** Regresa el iterador con todas las filas del Excel.
- **getCurrentRow:** Regresa la fila con la que se está trabajando actualmente.
- **nextRow:** Método que carga la siguiente fila del Excel siempre y cuando exista una más.
- **addCurrentRowToResults:** Agrega la fila actual a la lista de resultados, este método se ejecuta cuando la fila ha cumplido todos los filtros.
- **Destroy:** Cierra la conexión con el Excel al finalizar la evaluación de la expresión.
- **tableExist:** Valida que existe una hoja de Excel con el mismo nombre del parámetro de entrada.
- **loadTable:** Carga la hoja de Excel con el nombre de la tabla y analiza la primera fila para obtener el nombre de las columnas así como su posición dentro del Excel.
- **tableColumn:** Valida si una columna existe en el Excel.
- **columnIndex:** Regresa la posición de la columna.

Como vemos, esta clase guarda muchos datos que son alimentados o modificados a medida que la expresión es evaluada e interpretada para finalmente tener un resultado.

```
1.  package oscarblancarte.ipd.interprete.sql;
2.
3.  import java.io.IOException;
4.  import java.text.SimpleDateFormat;
5.  import java.util.ArrayList;
6.  import java.util.Arrays;
7.  import java.util.Iterator;
8.  import java.util.List;
9.  import org.apache.poi.hssf.usermodel.HSSFSheet;
10.  import org.apache.poi.hssf.usermodel.HSSFWorkbook;
```

```
11. import org.apache.poi.ss.usermodel.Cell;
12. import org.apache.poi.ss.usermodel.Row;
13.
14. public class Context {
15.
16.     private SimpleDateFormat dateFormat;
17.     private HSSFWorkbook workboook;
18.     private HSSFSheet sheets;
19.     private String[] columns;
20.     private final String dbPath;
21.     private Iterator<Row> tableIterator;
22.     private final List<Row> resultRows = new ArrayList<>();
23.     private Row currentRow;
24.     private List<Object[]> result;
25.     private int resultColumnCount;
26.
27.     public Context(String dbPath) {
28.         this.dbPath = dbPath;
29.         initiateFileRead();
30.     }
31.
32.     public void createResultArray(int columns){
33.         this.resultColumnCount = columns;
34.         result = new ArrayList<>();
35.     }
36.
37.     public SimpleDateFormat getDateFormat() {
38.         return dateFormat;
39.     }
40.
41.     public void setDateFormat(SimpleDateFormat dateFormat) {
42.         this.dateFormat = dateFormat;
43.     }
44.
45.     public List<Object[]> getResultArray(){
46.         return result;
47.     }
48.
49.     public Object[] createRow(){
50.         Object[] row = new Object[resultColumnCount];
51.         result.add(row);
52.         return row;
53.     }
54.
55.     public List<Row> getResultRow(){
56.         return resultRows;
57.     }
58.
59.
60.
61.     private void initiateFileRead() {
62.         try {
63.             System.out.println("dbPath > " + dbPath);
```

```
64.             workboook = new HSSFWorkbook(Context.class.getClassLoader()
65.                     .getResourceAsStream(dbPath));
66.         } catch (IOException e) {
67.             e.printStackTrace();
68.         }
69.     }
70.
71.     public void createRowIterator() {
72.         tableIterator = sheets.iterator();
73.         tableIterator.next();
74.     }
75.
76.     public Row getCurrentRow() {
77.         return currentRow;
78.     }
79.
80.     public boolean nextRow() {
81.         if (tableIterator.hasNext()) {
82.             currentRow = tableIterator.next();
83.             return true;
84.         }
85.         return false;
86.     }
87.
88.     public Iterator<Row> getRowIterator() {
89.         return tableIterator;
90.     }
91.
92.     public void addCurrentRowToResults() {
93.         resultRows.add(currentRow);
94.     }
95.
96.  public void destroy() {
97.         try {
98.             workboook.close();
99.         } catch (IOException e) {
100.            e.printStackTrace();
101.        }
102.    }
103.
104.    public boolean tableExist(String table) {
105.        return loadTable(table);
106.    }
107.
108.    private boolean loadTable(String table) {
109.
110.        if (columns != null) {
111.            return true;
112.        }
113.
114.        sheets = workboook.getSheet(table);
115.        if (sheets == null) {
116.            return false;
```

```
117.        }
118.        for (Row row : sheets) {
119.            int lastRow = row.getLastCellNum();
120.            System.out.println("lastRow > " + lastRow);
121.            columns = new String[lastRow];
122.            int index = 0;
123.            for (Cell cell : row) {
124.                columns[index++] = cell.getStringCellValue();
125.            }
126.            System.out.println("Table > '" + table + "' Colums > '"
127.                    + Arrays.toString(columns) + "'");
128.            break;
129.        }
130.        return true;
131.    }
132.
133.    public boolean tableColumn(String column) {
134.        for (String col : columns) {
135.            if (col.equalsIgnoreCase(column)) {
136.                return true;
137.            }
138.        }
139.        return false;
140.    }
141.
142.    public int columnIndex( String column) {
143.        for (int c = 0; c < columns.length; c++) {
144.            if (columns[c].equalsIgnoreCase(column)) {
145.                return c;
146.            }
147.        }
148.        throw new RuntimeException("Columna '" + column + "' no existe");
149.    }
150.}
```

Expresiones Terminales:

Las siguientes clases representan las expresiones terminales.

Clase LiteralExpression:

Esta clase representa todo lo que debe de ser manejado de forma “literal” es decir, a que como está en la expresión debe ser tomado, el nombre de la columna debe ser guardado tal cual está en la expresión. Esta clase recibe como parámetro el literal y el método *interpret* simplemente regresa este literal cuando es evaluado.

```
1.  package oscarblancarte.ipd.interprete.sql.terminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.Context;
4.  import oscarblancarte.ipd.interprete.sql.InterpreteException;
5.  import oscarblancarte.ipd.interprete.sql.AbstractSQLExpression;
6.
7.  public class LiteralExpression implements AbstractSQLExpression {
8.
9.      protected String literal;
10.
11.     public LiteralExpression(String literal) {
12.         this.literal = literal;
13.     }
14.
15.     @Override
16.     public Object interpret(Context context) throws InterpreteException {
17.         return literal;
18.     }
19.
20.     @Override
21.     public String toString() {
22.         return literal;
23.     }
24. }
```

Clase NumericExpression:

Esta clase hereda de *LiteralExpression*, lo que indica que su valor también debe ser tratado como literal, sin embargo, esta clase le indica al *interpreter* que el valor contenido es un Número en vez de simple texto.

```
1.  package oscarblancarte.ipd.interprete.sql.terminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.Context;
4.
5.  public class NumericExpression extends LiteralExpression{
6.
7.      private final Number number;
8.
9.      public NumericExpression(Number num){
10.         super(null);
11.         this.number = num;
12.     }
13.
14.     @Override
15.     public Object interpret(Context context) {
16.         return number;
```

```
17.     }
18.
19.     @Override
20.     public String toString() {
21.         return number.toString();
22.     }
23.
24. }
```

Cuando la expresión sea interpretada sólo se obtendrá el valor numérico de la propiedad *number*.

Clase TextExpression:

Esta clase hereda de *LiteralExpression* lo que indica que también su valor deberá ser tratado como "literal", sin embargo, esta clase le indica al *interpreter* que su contenido es texto y el texto puede ser una cadena de texto con separación de líneas y desde la consulta se encuentra entre comillas. Por ejemplo *"Call Center"* que al estar entre comillas le indica al *interpreter* que debe ser tomado como un sólo *token* y no separe *Call* y *Center* como dos *token* independientes.

```
1. package oscarblancarte.ipd.interprete.sql.terminal;
2.
3. import oscarblancarte.ipd.interprete.sql.Context;
4.
5. public class TextExpression extends LiteralExpression {
6.
7.     public TextExpression(String literal) {
8.         super(literal);
9.     }
10.
11.     @Override
12.     public String interpret(Context context) {
13.         return literal;
14.     }
15.
16.     @Override
17.     public String toString() {
18.         return "'"+literal+"'";
19.     }
20. }
```

El método *interpret* sólo regresa el valor literal al ser interpretado.

Clase DateExpression:

Clase que hereda de *LiteralExpression*, lo que indica que su valor deberá ser tratado como literal, sin embargo, esta clase le indica al *interpreter* que el contenido es una fecha y no sólo texto.

```
1.  package oscarblancarte.ipd.interprete.sql.terminal;
2.
3.  import java.text.SimpleDateFormat;
4.  import oscarblancarte.ipd.interprete.sql.Context;
5.  import oscarblancarte.ipd.interprete.sql.InterpreteException;
6.
7.  public class DateExpression extends LiteralExpression {
8.
9.      public DateExpression(String literal) {
10.         super(literal);
11.     }
12.
13.     @Override
14.     public Object interpret(Context context) throws InterpreteException {
15.         try {
16.             SimpleDateFormat dateFormat = context.getDateFormat();
17.             return dateFormat.parse(literal);
18.         } catch (Exception e) {
19.             throw new InterpreteException("Formato de fecha inválido '"+literal+"'");
20.         }
21.     }
22.
23.     @Override
24.     public String toString() {
25.         return "'"+literal+"'";
26.     }
27. }
```

Cuando la clase es interpretada ésta regresa la fecha formateada.

Expresiones No Terminales:

Las siguientes clases representan las clases no terminales.

Clase SelectExpression:

Clase que representa a un *select* de SQL, es una clase no terminal y además está compuesta de otras expresiones no terminales como *TargetExpression*,

FromExpression y *WhereExpression* las cuales deberán ser evaluadas primero antes de tener un resultado final.

Podemos apreciar el método *evaluate* que evalúa al *target* (columnas), luego al *from* (tabla Origen) y finalmente, evalúa al *where* (condiciones de filtrado), una vez evaluado todo, el contexto tiene las filas filtradas listas para ser devueltas al cliente.

En resumen lo que sucede es lo siguiente:

1. El *form* carga el Excel y carga la hoja donde se encuentra la información, junto con esto se cargan las columnas válidas y las posiciones de éstas.
2. El *where* itera todos los registros de la hoja de Excel y evalúa si la fila cumple con las cláusulas de filtrado, si es así agrega los registros a los resultados, si no, simplemente continúa con la siguiente fila y repite el proceso.
3. El *target* toma los resultados filtrados por el *where* y simplemente toma las columnas que fueron solicitadas en el *query* y genera un nuevo arreglo de resultados finales.
4. Los resultados son devueltos al cliente.

```
1.  package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.AbstractSQLExpression;
4.  import java.util.List;
5.  import oscarblancarte.ipd.interprete.sql.Context;
6.  import oscarblancarte.ipd.interprete.sql.InterpreteException;
7.
8.  public class SelectExpression implements AbstractSQLExpression{
9.
10.     private final TargetExpression target;
11.     private final FromExpression from;
12.     private final WhereExpression where;
13.
14.     public SelectExpression(TargetExpression columns,
15.             FromExpression table,WhereExpression where){
16.         this.target = columns;
17.         this.from = table;
18.         this.where = where;
19.     }
20.
```

```
21.     public SelectExpression(TargetExpression columns,
22.             FromExpression table){
23.         this.target = columns;
24.         this.from = table;
25.         this.where = new WhereExpression(null);
26.     }
27.
28.     @Override
29.     public List<Object[]> interpret(Context context)
30.             throws InterpreteException{
31.         from.interpret(context);
32.         where.interpret(context);
33.         target.interpret(context);
34.         return context.getResultArray();
35.     }
36.
37.     @Override
38.     public String toString() {
39.         return "SELECT " + target.toString() + " " +
40.                 from.toString() + " " +  where.toString();
41.     }
42. }
```

Cuando el método *interpreter* termine, ya habremos consultado al Excel con un *Select* SQL. ¿Eso fue fácil no? En parte sí, pero aún falta analizar el resto de clases que son las que finalmente hacen el trabajo.

Clase FromExpression:

Esta clase representa el apartado "***FROM <TABLE>***" de la expresión por lo que el nombre de la tabla queda almacenado dentro de una *LiteralExpresion*.

El método *interpret* lo primero que hace es interpretar la expresión del nombre de la tabla (líneas 17 y 18) la cual regresa el nombre de la tabla como un *String*, seguido, se valida que la tabla exista mediante el método *tableExist* (línea 19) del *Context*; de no existir, un error se regresará indicando que la tabla no existe (línea 19).

Esta expresión no regresa nada debido a que con la llamada al método *tableExist* se crea en el *Context* la información de la tabla junto con sus columnas.

```
1.  package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.*;;
4.  import oscarblancarte.ipd.interprete.sql.terminal.*;
5.
6.  public class FromExpression implements AbstractSQLExpression {
7.
8.      private LiteralExpression table;
9.
10.     public FromExpression(LiteralExpression from){
11.         this.table = from;
12.     }
13.
14.     @Override
15.     public Object interpret(Context context)
16.             throws InterpreteException{
17.         String tableName =
18.                 table.interpret(context).toString();
19.         if(!context.tableExist(tableName)){
20.             throw new InterpreteException(
21.                     "La tabla '"+tableName+"' no existe");
22.         }
23.         return null;
24.     }
25.
26.     @Override
27.     public String toString() {
28.         return "FROM " + table.toString();
29.     }
30. }
```

Observemos que nos apoyamos de la clase *Context* para determinar si la tabla (Hoja) existe en nuestro Excel. (línea 19).

Clase WhereExpression:

La clase *WhereExpression* representa toda la sección del "*WHERE*" y se encarga de determinar si una fila del Excel deberá ser parte del resultado final, para lo cual evalúa todas las expresiones de filtrado y agrega únicamente las que cumplen con el criterio de búsqueda.

El método *interpret,* al ser evaluado, creará un iterador de todos los registros del Excel (línea 15) para posteriormente recorrer todos los registros (línea 16), notemos que con la llamada al método *nextRow* se crea en el *Context* la referencia al registro actual, el cual más adelante será utilizado para determinar si el registro cumple con todas las reglas de filtrado, mientras tanto, en la línea 20 se evalúan

todas las expresiones del *where* que deberán ser evaluadas a "*true*" para que sean agregadas a los resultados.

Finalmente, si la expresión *where* no es declarada nos aseguramos de agregar todos los registros como parte del resultado final (línea 17).

```
1.  package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.*;
4.
5.  public class WhereExpression implements AbstractSQLExpression {
6.
7.      private StatementExpression statement;
8.
9.      public WhereExpression(StatementExpression statement) {
10.         this.statement = statement;
11.     }
12.
13.     @Override
14.     public Void interpret(Context context) throws InterpreteException {
15.         context.createRowIterator();
16.         while (context.nextRow()) {
17.             if (statement == null) {
18.                 context.addCurrentRowToResults();
19.             } else {
20.                 Boolean result = (Boolean) statement.interpret(context);
21.                 if (result) {
22.                     context.addCurrentRowToResults();
23.                 }
24.             }
25.         }
26.         return null;
27.     }
28.
29.     @Override
30.     public String toString() {
31.         if (statement != null) {
32.             return "WHERE " + statement.toString();
33.         } else {
34.             return "";
35.         }
36.     }
37. }
```

Clase TargetExpression:

Esta clase interpreta las columnas seleccionadas de la tabla y representan las columnas a seleccionar. Cuando el método *interpret* es ejecutado sucede lo siguiente:

1. Se establece el número de columnas seleccionadas en el *Context* (línea 19).
2. En la línea 21 obtiene todas las finales filtradas por el select mediante el método *getResultRow* del *Context*.
3. En la línea 22 itera los resultados con el fin de filtrar las columnas solicitadas.
4. En la línea 23 crea un nuevo arreglo donde se establecerán únicamente las columnas solicitadas en el *select*.
5. De las líneas 26 a la 32 se iteran las expresiones del *TargetExpression* con el fin de seleccionar únicamente las columnas que fueron solicitadas en el *request*, para lo cual se obtiene el índice de las columnas solicitadas en el Excel y en la línea 32 se obtiene el valor de la columna, finalmente, en la línea 33 se agrega el valor de la columna al arreglo de resultados finales.

En resumen, al ser evaluada esta expresión se obtiene un arreglo de resultados únicamente con las columnas solicitas en el *select*.

```
1.  package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3.  import oscarblancarte.ipd.interprete.sql.AbstractSQLExpression;
4.  import java.util.*;
5.  import org.apache.poi.ss.usermodel.*;
6.  import oscarblancarte.ipd.interprete.sql.*;
7.  import oscarblancarte.ipd.interprete.sql.terminal.LiteralExpression;
8.
9.  public class TargetExpression implements AbstractSQLExpression {
10.
11.     private List<LiteralExpression> targets = new ArrayList<LiteralExpression>();
12.
13.     public TargetExpression(LiteralExpression... expressions) {
14.         this.targets = Arrays.asList(expressions);
15.     }
16.
17.     @Override
```

```
18.     public Void interpret(Context context) throws InterpreteException{
19.         context.createResultArray(targets.size());
20.
21.         List<Row> resultRow = context.getResultRow();
22.         for (Row row : resultRow) {
23.             Object[] result = context.createRow();
24.             int col = 0;
25.
26.             for (LiteralExpression literalExpression : targets) {
27.                 String column = literalExpression.interpret(context).toString();
28.                 context.tableColumn(column);
29.                 int columnIndex = context.columnIndex(column);
30.                 Cell cell = row.getCell(columnIndex);
31.                 cell.setCellType(Cell.CELL_TYPE_STRING);
32.                 String value = cell.getStringCellValue();
33.                 result[col++] = value;
34.             }
35.         }
36.         return null;
37.     }
38.
39.     @Override
40.     public String toString() {
41.         String output = "";
42.         for (LiteralExpression literalExpression : targets) {
43.             output += ", " + literalExpression.toString();
44.         }
45.         return output.substring(2);
46.     }
47. }
```

Interface StatementExpression:

Esta interface nos sirve para crear condiciones de búsqueda como el *AND*, *OR* y las expresiones *Booleanas* como =, >, >=, <, <=, <>, esta interface será utilizada por el intérprete para asegurarnos de que únicamente subclases de *StatementExpression* lleguen al *WhereExpression*.

```
1. package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3. import oscarblancarte.ipd.interprete.sql.*;
4.
5. public interface StatementExpression extends AbstractSQLExpression {
6.
7.     @Override
8.     public Object interpret(Context context)throws InterpreteException;
9. }
```

Clase BooleanExpression:

La clase *BooleanExpression* evalúa expresiones que tiene como resultado un booleano, esta clase es utilizada para evaluar expresiones como la siguiente: ID > 10, FIRT_NAME = “oscar”, las expresiones son procesadas de forma distinta dependiendo del tipo de subclase de la variable *rightExpression*, la cual determina si el campo es texto (*TextExpression*), número (*NumericExpression*) o fecha (*DateExpression*).

Cuando el método *interpret* es ejecutado lo primero que hay que hacer es interpretar las tres partes que componen una expresión booleana (líneas 26 a la 28) dos Operandos y un Operador, en este caso *leftExpression* y *rightExpression* corresponde a los Operandos y el operador es el *booleanExpression*.

Si el *rightExpression* es una subclase de *NumericExpression* entonces las comparaciones contra el valor del Excel se harán mediante operaciones numéricas (líneas 33 a la 52).
Para el caso de los numéricos utilizamos los operadores nativos del lenguaje como =, >, >=, <, <=, <> para realizar las comparaciones. Este procesamiento se hace dentro del *if* de la línea 33.

Si el *rightExpression* es una subclase de *DateExpression* entonces las comparaciones contra el valor del Excel se harán mediante comparación de tiempo en vez de texto (líneas 53 a 80). En el caso de las fechas lo primero que hacemos es convertir la fecha en un Objeto *Date* mediante el *DateFormat* que tenemos en el *Context* y seguido, obtenemos su representación en *Long*, después, podemos utilizar los operadores nativos del lenguaje =, >, >=, <, <=, <> para realizar las comparaciones. Este procesamiento se hace dentro del if de la línea 53.

Si el *rightExpression* es una subclase de *LiteralExpression* o *TextExpression* entonces las comparaciones contra los valores del Excel se harán como texto. En el caso de ser texto o literal entonces utilizamos los métodos estándar *compareToIgnoreCase* y *equalsIgnoreCase* para realizar las comparaciones, el primero nos servirá para determinar cuál es mayor o menor y el segundo lo

utilizaremos para saber si es igual o diferente (negando la expresión). Este procesamiento se hace dentro del *if* de la línea 81.

```
1.  package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3.  import java.util.Date;
4.  import org.apache.poi.ss.usermodel.*;
5.  import oscarblancarte.ipd.interprete.sql.*;
6.  import oscarblancarte.ipd.interprete.sql.terminal.*;
7.
8.  public class BooleanExpression implements StatementExpression {
9.
10.     private final LiteralExpression leftExpression;
11.     private final LiteralExpression booleanOperator;
12.     private final LiteralExpression rightExpression;
13.
14.     public BooleanExpression(LiteralExpression leftExp,
15.             LiteralExpression operator, LiteralExpression rightExp) {
16.         this.leftExpression = leftExp;
17.         this.booleanOperator = operator;
18.         this.rightExpression = rightExp;
19.     }
20.
21.     @Override
22.     public Boolean interpret(Context context)
23.             throws InterpreteException {
24.         Row currentRow = context.getCurrentRow();
25.
26.         String left = leftExpression.interpret(context).toString();
27.         String opr = booleanOperator.interpret(context).toString();
28.         Object right = rightExpression.interpret(context);
29.
30.         int columnIndex = context.columnIndex(left);
31.         Cell cell = currentRow.getCell(columnIndex);
32.
33.         if (rightExpression instanceof NumericExpression) {
34.             cell.setCellType(Cell.CELL_TYPE_NUMERIC);
35.             double cellValue = cell.getNumericCellValue();
36.             double rightVal = ((Number) right).doubleValue();
37.             if (opr.equals("=")) {
38.                 return cellValue == rightVal;
39.             } else if (opr.equals("<>") || opr.equals("!=")) {
40.                 return cellValue != rightVal;
41.             } else if (opr.equals(">")) {
42.                 return cellValue > rightVal;
43.             } else if (opr.equals(">=")) {
44.                 return cellValue >= rightVal;
45.             } else if (opr.equals("<")) {
46.                 return cellValue < rightVal;
47.             } else if (opr.equals("<=")) {
```

```
48.                 return cellValue <= rightVal;
49.             } else {
50.                 throw new RuntimeException(
51.                         "Operador no esperado '" + opr + "'");
52.             }
53.         } else if (rightExpression instanceof DateExpression) {
54.             cell.setCellType(Cell.CELL_TYPE_STRING);
55.             String cellValue = cell.getStringCellValue();
56.             long cellDateLong = 0;
57.             long expressionDateLong = ((Date) right).getTime();
58.             try {
59.                 Date date = context.getDateFormat().parse(cellValue);
60.                 cellDateLong = date.getTime();
61.             } catch (Exception e) {
62.                 throw new InterpreteException(
63.                         "Fecha inválida > " + cellValue);
64.             }
65.             if (opr.equals("=")) {
66.                 return cellDateLong == expressionDateLong;
67.             } else if (opr.equals("<>") || opr.equals("!=")) {
68.                 return cellDateLong != expressionDateLong;
69.             } else if (opr.equals(">")) {
70.                 return cellDateLong > expressionDateLong;
71.             } else if (opr.equals(">=")) {
72.                 return cellDateLong >= expressionDateLong;
73.             } else if (opr.equals("<")) {
74.                 return cellDateLong < expressionDateLong;
75.             } else if (opr.equals("<=")) {
76.                 return cellDateLong <= expressionDateLong;
77.             } else {
78.                 throw new RuntimeException(
79.                         "Operador no esperado '" + opr + "'");
80.             }
81.         } else if (rightExpression instanceof LiteralExpression
82.                 || rightExpression instanceof TextExpression) {
83.             cell.setCellType(Cell.CELL_TYPE_STRING);
84.             String cellValue = cell.getStringCellValue();
85.             String rightVal = right.toString();
86.             if (opr.equals("=")) {
87.                 return cellValue.equals(rightVal);
88.             } else if (opr.equals("<>") || opr.equals("!=")) {
89.                 return !cellValue.equalsIgnoreCase(rightVal);
90.             } else if (opr.equals(">")) {
91.                 int result = cellValue.toString().
92.                         compareToIgnoreCase(rightVal.toString());
93.                 return result > 0;
94.             } else if (opr.equals(">=")) {
95.                 int result = cellValue.toString().
96.                         compareToIgnoreCase(rightVal.toString());
97.                 return result >= 0;
98.             } else if (opr.equals("<")) {
99.                 int result = cellValue.toString().
100.                        compareToIgnoreCase(rightVal.toString());
```

```
101.             return result < 0;
102.         } else if (opr.equals("<=")) {
103.             int result = cellValue.toString().
104.                     compareToIgnoreCase(rightVal.toString());
105.             return result <= 0;
106.         } else {
107.             throw new RuntimeException(
108.                     "Operador no esperado '" + opr + "'");
109.         }
110.     } else {
111.         throw new RuntimeException("Tipo de celda no soportada "
112.                 + right.getClass().getSimpleName());
113.     }
114. }
115.
116. @Override
117. public String toString() {
118.     return leftExpression.toString() + " "
119.             + booleanOperator.toString() + " "
120.             + rightExpression.toString();
121. }
122.}
```

Clase AndExpression:

Esta clase representa la instrucción *AND* de SQL, mediante esta clase podemos evaluar dos expresiones del operador *AND*. La instrucción es verdadera siempre y cuando las dos expresiones sean verdaderas.

Esta clase tiene dos variables de tipo *StatementExpression* las cuales serán evaluadas en el método *interpreter* obteniendo un *booleano* como resultado de su interpretación, finalmente, estos dos resultados son evaluados con el operador & (*and*) y el resultado es devuelto.

```
1. package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3. import oscarblancarte.ipd.interprete.sql.*;
4.
5. public class AndExpression implements StatementExpression {
6.
7.     private final StatementExpression leftStatement;
8.     private final StatementExpression rightStatement;
9.
10.     public AndExpression(StatementExpression leftStatement,
11.             StatementExpression rightStatement) {
12.         this.leftStatement = leftStatement;
```

```
13.         this.rightStatement = rightStatement;
14.     }
15.
16.     @Override
17.     public Object interpret(Context context)throws InterpreteException {
18.         Boolean leftReslt = (Boolean) leftStatement.interpret(context);
19.         Boolean rightReslt = (Boolean) rightStatement.interpret(context);
20.         return leftReslt && rightReslt;
21.     }
22.
23.     @Override
24.     public String toString() {
25.         return "( " + leftStatement.toString()
26.                 + " AND " + rightStatement.toString() + " )";
27.     }
28. }
```

Observemos las líneas 18 y 19 en las que evaluamos las expresiones internas para determinar el resultado de la evaluación, dicha evaluación deberá regresar un resultado booleano, luego los dos booleanos son comparados con el operador && (AND) para determinar si las dos expresiones son verdaderas.

Clase OrExpression:

Esta clase representa la instrucción OR de SQL, mediante esta clase podemos evaluar dos expresiones mediante el operador OR. La instrucción es verdadera si cualquiera de las dos expresiones es verdadera.

Esta clase tiene dos variables de tipo *StatementExpression* las cuales serán evaluadas en el método *interpreter* obteniendo un booleano como resultado de su interpretación, finalmente, estos dos resultados son evaluados con el operador | (or) y el resultado es devuelto.

```
1. package oscarblancarte.ipd.interprete.sql.nonterminal;
2.
3. import oscarblancarte.ipd.interprete.sql.*;
4.
5. public class OrExpression implements StatementExpression{
6.     private StatementExpression leftStatement;
7.     private StatementExpression rightStatement;
8.
9.     public OrExpression(StatementExpression leftStatement,
```

```
10.             StatementExpression rightStatement) {
11.         this.leftStatement = leftStatement;
12.         this.rightStatement = rightStatement;
13.     }
14.
15.     @Override
16.     public Object interpret(Context context)throws InterpreteException {
17.         Boolean leftReslt = (Boolean) leftStatement.interpret(context);
18.         Boolean rightReslt = (Boolean) rightStatement.interpret(context);
19.         return leftReslt || rightReslt;
20.     }
21.
22.     @Override
23.     public String toString() {
24.         return "( "+leftStatement.toString()
25.                 + " OR " + rightStatement.toString() + " )";
26.     }
27. }
```

Observemos las líneas 17 y 18 en las cuales las expresiones booleanas son evaluadas para determinar si por lo menos una es verdadera para regresar el resultado.

Clase InterpreterMain:

Clase principal que utilizaremos para probar la aplicación, en la siguiente sección analizaremos la clase, por lo pronto veamos que tenemos tres métodos para crear diferentes expresiones, los cuales son:

- **getById:** Crea una Expresión que consulta la tabla de Empleado mediante su ID regresando las columnas *FIRS_NAME* y *LAST_NAME*.
- **findByDate:** Crea una Expresión que permite buscar a los Empleados con fecha de nacimiento mayor al '*01/01/1990*'.
- **findByTwoID:** Crea un Expresión que permite buscar Empleados mediante dos ID, para lo cual se utiliza la Expresión OR.
- **complexQuery:** Crea una Expresión para buscar a los Empleados con Estatus '*Activo*', fecha de nacimiento (BORN_DATE) menor o igual al '*01/01/1981*' y que pertenezcan al departamento de Sistemas o de Contabilidad.

Observemos que los métodos descritos se encargan de crear los árboles sintácticos abstractos correspondientes a las expresiones.

En la línea 13 mandaremos llamar el método de la expresión que queremos utilizar.

En la línea 14 llamamos el método *toString* para ver cómo es que se está generando el *query*, este método sólo es informativo y no representa ninguna funcionalidad dentro del programa.

En las líneas 15 y 16 creamos el contexto del *interpreter* para lo cual se establece la URL del archivo de Excel y después se establece el formato de fecha utilizado para hacer el *parsers* de las fechas del Excel.

En la línea 18 se hace la interpretación de la expresión y obtenemos el resultado de la misma en arreglo.

En las líneas 19 y 20 se imprimen los valores del arreglo de resultados.

```
1.  package oscarblancarte.ipd.interprete;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.*;
5.  import oscarblancarte.ipd.interprete.sql.*;
6.  import oscarblancarte.ipd.interprete.sql.nonterminal.*;
7.  import oscarblancarte.ipd.interprete.sql.terminal.*;
8.
9.  public class InterpreteMain {
10.
11.     public static void main(String[] args) {
12.         //Arbol Sintactico Abstracto
13.         SelectExpression select = getById();
14.         System.out.println(select.toString());
15.         Context context = new Context("META-INF/Employee.xls");
16.         context.setDateFormat(new SimpleDateFormat("dd/MM/yyyy"));
17.         try {
18.             List<Object[]> output = select.interpret(context);
19.             for (Object[] object : output) {
20.                 System.out.println(Arrays.toString(object));
21.             }
22.         } catch (InterpreteException e) {
23.             e.printStackTrace();
24.         } finally {
```

```
25.             context.destroy();
26.         }
27.     }
28.
29.     public static SelectExpression getById() {
30.         SelectExpression select = new SelectExpression(
31.             new TargetExpression(
32.                 new LiteralExpression("FIRST_NAME"),
33.                 new LiteralExpression("LAST_NAME")
34.             ),
35.             new FromExpression(
36.                 new LiteralExpression("EMPLOYEES")),
37.             new WhereExpression(
38.                 new BooleanExpression(
39.                     new LiteralExpression("ID"),
40.                     new LiteralExpression("="),
41.                     new NumericExpression(10)
42.                 )
43.             )
44.         );
45.         return select;
46.     }
47.
48.     public static SelectExpression findByDate() {
49.         SelectExpression select = new SelectExpression(
50.             new TargetExpression(
51.                 new LiteralExpression("ID"),
52.                 new LiteralExpression("BORN_DATE"),
53.                 new LiteralExpression("DEPARTMENT"),
54.                 new LiteralExpression("FIRST_NAME"),
55.                 new LiteralExpression("LAST_NAME")
56.             ),
57.             new FromExpression(
58.                 new LiteralExpression("EMPLOYEES")),
59.             new WhereExpression(
60.                 new BooleanExpression(
61.                     new LiteralExpression("BORN_DATE"),
62.                     new LiteralExpression(">"),
63.                     new DateExpression("01/01/1990")
64.                 )
65.             )
66.         );
67.         return select;
68.     }
69.
70.     public static SelectExpression findByTwoID() {
71.         SelectExpression select = new SelectExpression(
72.             new TargetExpression(
73.
74.                 new LiteralExpression("FIRST_NAME"),
75.                 new LiteralExpression("DEPARTMENT"),
76.                 new LiteralExpression("ID")
77.             ),
```

```
78.             new FromExpression(
79.                 new LiteralExpression("EMPLOYEES")),
80.             new WhereExpression(
81.                 new OrExpression(
82.                     new BooleanExpression(
83.                         new LiteralExpression("ID"),
84.                         new LiteralExpression("="),
85.                         new NumericExpression(5)
86.                     ),
87.                     new BooleanExpression(
88.                         new LiteralExpression("ID"),
89.                         new LiteralExpression("="),
90.                         new NumericExpression(14)
91.                     )
92.                 )
93.             )
94.         );
95.         return select;
96.     }
97.
98.     public static SelectExpression complexQuery() {
99.         SelectExpression select = new SelectExpression(
100.        new TargetExpression(
101.            new LiteralExpression("FIRST_NAME"),
102.            new LiteralExpression("LAST_NAME")
103.        ),
104.        new FromExpression(
105.            new LiteralExpression("EMPLOYEES")),
106.        new WhereExpression(
107.            new AndExpression(
108.                new BooleanExpression(
109.                    new LiteralExpression("STATUS"),
110.                    new LiteralExpression("="),
111.                    new TextExpression("Activo")
112.                ),
113.                new AndExpression(
114.                    new BooleanExpression(
115.                        new LiteralExpression("BORN_DATE"),
116.                        new LiteralExpression("<="),
117.                        new DateExpression("01/01/1981")
118.                    ),
119.                    new OrExpression(
120.                        new BooleanExpression(
121.                            new LiteralExpression("DEPARTMENT"),
122.                            new LiteralExpression("="),
123.                            new TextExpression("Sistemas")
124.                        ),
125.                        new BooleanExpression(
126.                            new LiteralExpression("DEPARTMENT"),
127.                            new LiteralExpression("="),
128.                            new TextExpression("Contabilidad")
129.                        )
130.                    )
```

```
131.                )
132.            )
133.        )
134.        );
135.        return select;
136.    }
137.}
```

La ejecución:

Una vez que hemos analizado el código y sabemos cómo es que debe de funcionar, pasaremos a ejecutarlo y analizar los resultados de la ejecución. Lo primero que haremos es asegurarnos de tener la línea 13 de la siguiente manera:

```
1.      SelectExpression select = findByID();
```

Cuando ejecutemos la aplicación tendremos el siguiente resultado:

```
run:
SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEES WHERE ID = 10
Table > 'EMPLOYEES' Colums > '[ID, FIRST_NAME, LAST_NAME, DEPARTMENT, BORN_DATE, STATUS, TELEPHONE]'
[Iridiam, Galindo]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 182: Resultado de la ejecución.

Lo que podemos ver es la consulta que se generaría con la expresión, sin embargo, recordemos que esta consulta es sólo informativa pues no es utilizada para nada; en su lugar una consulta como ésta es la necesaria para crear la expresión.

Lo segundo que vemos es el análisis de la tabla *EMPLOYEE*, la cual nos indica las columnas que tiene y el orden en el que están.

Al final vemos el resultado, en este caso nos indica que '*Iridiam Galindo*' fue la única fila que concordó con el criterio de búsqueda.

Segunda prueba:

Para esta segunda prueba, lo que haremos es asegurarnos de tener la línea 13 de la siguiente manera:

```
17.      SelectExpression select = findByDate();
```

Cuando ejecutemos la aplicación tendremos el siguiente resultado:

```
run:
SELECT ID, BORN_DATE, DEPARTMENT, FIRST_NAME, LAST_NAME FROM EMPLOYEES WHERE BORN_DATE > '01/01/1990'
Table > 'EMPLOYEES' Colums > '[ID, FIRST_NAME, LAST_NAME, DEPARTMENT, BORN_DATE, STATUS, TELEPHONE]'
[2, 10/20/1990, Contabilidad, Javier, Iturralde]
[5, 10/20/2001, Compras, Manuel, Benitez]
[6, 10/20/1992, Ventas, Adrian, Garcia]
[9, 10/20/1997, Ventas, Rebeca, de la Torre]
[15, 10/20/1990, Call center, Wendy, Solano]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 183: Resultado de la ejecución.

En este ejemplo hemos buscado a los Empleados que tienen una fecha de nacimiento mayor al '*01/01/1990*' y los resultados los podemos ver al final de la ejecución. En este caso los registros que concuerda con los criterios de búsqueda son 5.

Tercera prueba:

Para esta tercera prueba lo que haremos es asegurarnos de tener la línea 13 de la siguiente manera:

```
17.        SelectExpression select = complexQuery();
```

Cuando ejecutemos la aplicación tendremos el siguiente resultado:

```
SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEES
WHERE ( STATUS = 'Activo' AND ( BORN_DATE <= '01/01/1981' AND ( DEPARTMENT = 'Sistemas' OR DEPARTMENT = 'Contabilidad' ) ) )
Table > 'EMPLOYEES' Colums > '[ID, FIRST_NAME, LAST_NAME, DEPARTMENT, BORN_DATE, STATUS, TELEPHONE]'
[Liliana, Castro]
[Pablo, Picapiedra]
[Fernanda, Lopez]
[Gabriela, Rios]
```

Ilustración 184: Resultado de la ejecución.

Este ejemplo es más complejo porque realizamos una búsqueda donde los empleados tengan *status* activo y fecha de nacimiento menor o igual al '*01/01/2981*' y pertenezcan al departamento de Sistemas o Contabilidad.
Para estos criterios de búsqueda sólo obtenemos 4 resultados.

Como ejercicio podemos crear más expresiones para nuestro programa y validar que los resultados sean correctos, también podemos hacer consultas más avanzadas.

Siguientes pasos:

Hemos construido un interpreter compatible con SQL de una forma muy simple, sin embargo, existen instrucciones básicas (como el *insert* y *delete)* que podríamos agregar.

Para fortalecer los conocimientos realicemos los siguientes ejercicios:

- Agreguemos la instrucción *order by* que nos permita ordenar los resultados por una serie de columnas, para esto, el BNF sería el siguiente: *ORDER_BY: "ORDER BY" (<COLUMN> <"DESC"| "ASC>)?* O la palabra *ORDER BY* seguido de al menos una columna y la instrucción "DESC" o "ASC".

- Agreguemos la instrucción *group by* que agrupará los resultados según las columnas enviadas como parámetro. El BNF es el siguiente: *GROUP_BY: "GROUP BY" <COLUMN>?* O la palabra *GROUP BY* seguido de al menos una columna.

- Implementemos una nueva funcionalidad, podemos escoger la que más nos guste (*INSERT, DELETE, UPDATE*), debemos implementar en su totalidad y definir el BNF para dicha instrucción.

Patrón Mediator

El patrón de diseño Mediator se encarga de gestionar la forma en que un conjunto de clases se comunican entre sí, Mediator es especialmente útil cuando tenemos una gran cantidad de clases que se comunican de forma directa, ya que mediante la implementación de este patrón podemos crear una capa de comunicación bidireccional, en la cual las clases se pueden comunicar con el resto de ellas por medio de un objeto en común que funge como un mediador o intermediario.

En proyectos grandes podemos llegar a tener un problema, y es que el número de clases aumenta y con esto también aumentan las relaciones que tienen las clases con el resto de clases del proyecto. Esto puede suponer un grave problema de acoplamiento con el resto de clases de nuestro proyecto, sobre todo por que creamos canales de comunicación directos y difíciles de rastrear o depurar.

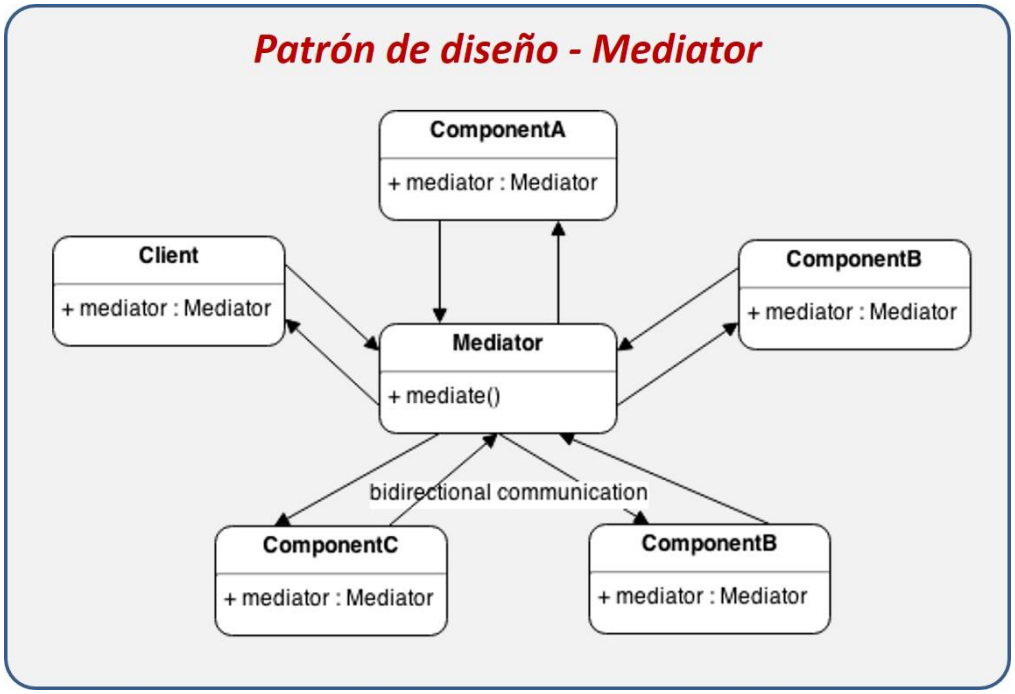


Ilustración 185: Estructura del patrón de diseño Mediator.

Los componentes que conforman el patrón se explican a continuación:

- **Client:** Componente que inicia la comunicación con el resto de los componentes por medio del mediador.
- **Components:** Componentes que son parte de la red de comunicación por medio del mediador, éstos pueden ser diversos objetos que comparten el mismo mediador para comunicarse.
- **Mediador:** Componente que sirve de mediador entre el resto de componentes, tiene como principal rol canalizar los mensajes entrantes al destinatario correspondiente.

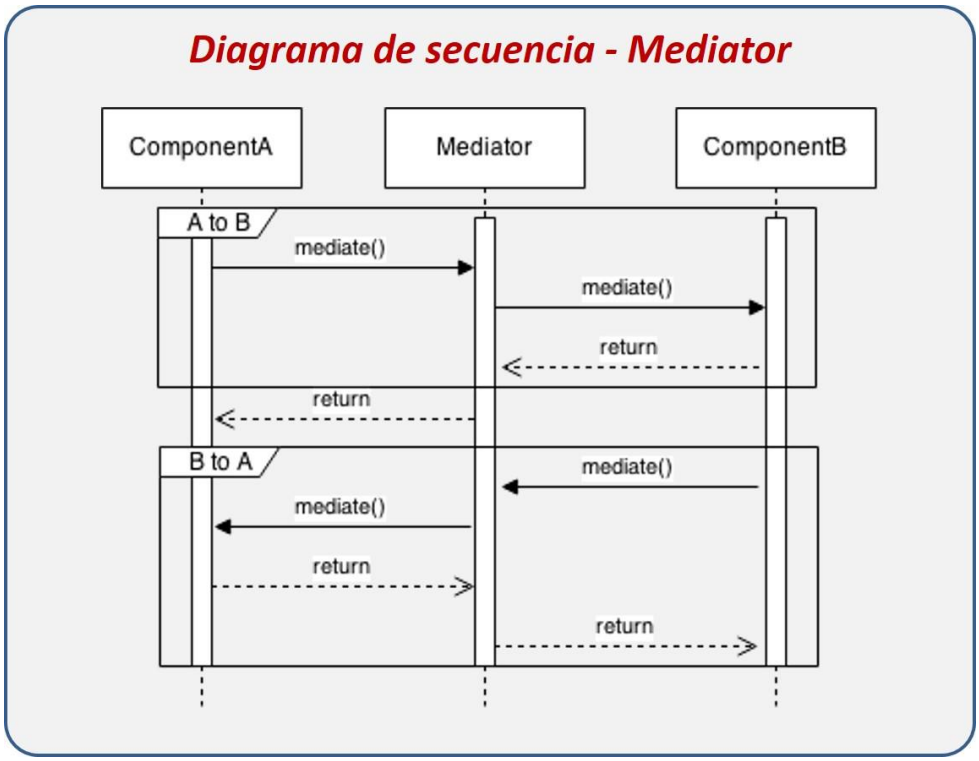


Ilustración 186: Diagrama de secuencia del patrón Mediator.

El diagrama se interpreta de la siguiente manera:

1. El ComponenteA desea comunicarse con el ComponenteB y le envía un mensaje por medio del mediador.

2. El mediador puede analizar el mensaje con fines de depuración, seguimiento o para canalizar el mensaje al destinatario.
3. El mensaje es entregado al destinatario y regresa una respuesta al mediador.
4. El mediador recibe la respuesta y la reedirecciona al ComponenteA.
5. De igual forma, el proceso se puede repetir del ComponentB al ComponentA repitiendo los pasos anteriores logrando una comunicación bidireccional.

Los componentes A y B se comunican por medio del Mediator, evitando tener conexión de forma directa, esta comunicación puede ser ventajosa si lo que buscamos es reducir la dependencia que existen entre los dos.

Cuándo utilizarlo:

- Cuando una gran cantidad de clases se comunican directamente de una forma muy conocida pero que es difícil entender qué clases se comunican con quién.
- Cuando es necesario desacoplar la forma en que se comunican las clases permitiendo que ambas partes puedan cambiar sin afectar directamente al resto.
- Cuando requerimos centralizar en un solo componente la lógica en que todos los objetos se comunican.

El escenario:

Para comprender mejor cómo es que las clases se comunican entre sí a través de mediator, expondremos un escenario donde nos han pedido que desarrollemos una plataforma en la que además de la funcionalidad estándar de la misma, nos han solicitado que sea posible agregarle nuevos módulos (estos módulos pueden ser tan simple o complejo como queramos), lo importante es que la aplicación final pueda llegar a tener una gran cantidad de módulos instalados. Además de esto, los módulos pueden necesitar comunicarse entre sí, por lo que definir un sistema de comunicación entre los distintos módulos es muy importante.

Los módulos instalados en nuestra aplicación requerirán de una comunicación bidireccional por lo que es necesario contemplar esto a la hora de desarrollar nuestra plataforma.

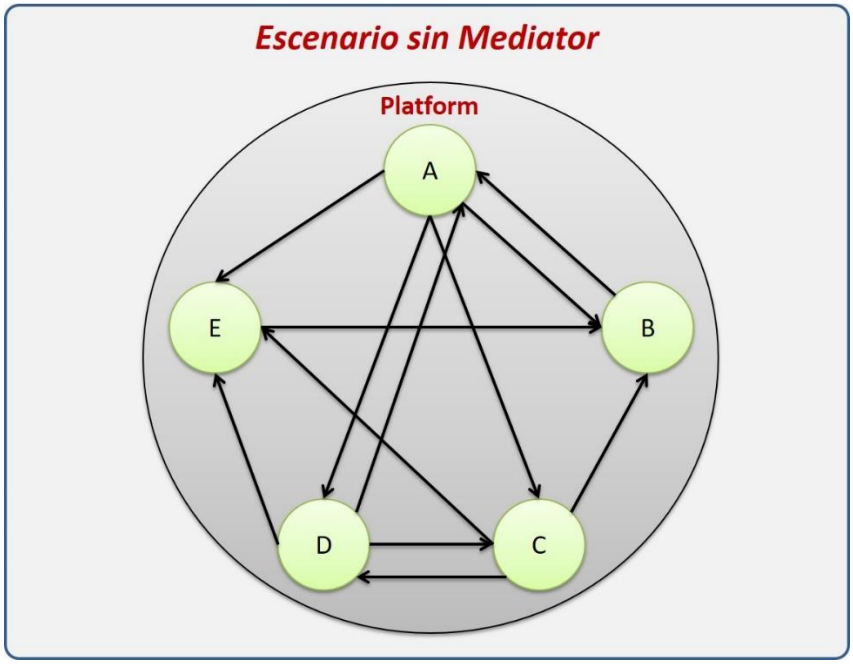


Ilustración 187: Escenario de comunicación de módulos sin mediador.

En la imagen anterior podemos apreciar una plataforma que contiene 5 módulos instalados. Cada módulo requiere comunicarse con otros módulos, lo que obliga a crear relaciones directas con cada uno de ellos, este escenario puede no

parecer tan malo, sin embargo, imaginemos este diagrama con al menos unos 10 módulos más, veremos que las relaciones se hacen cada vez más complejas por el gran número de éstos, también podemos apreciar que cada módulo está fuertemente acoplado al resto, por lo que si eliminamos uno de ellos las relaciones existentes se verán afectadas gravemente.

Para darle un poco más de emoción al ejemplo, simularemos un conjunto de módulos que procesen ventas por internet, el proceso se muestra a continuación:

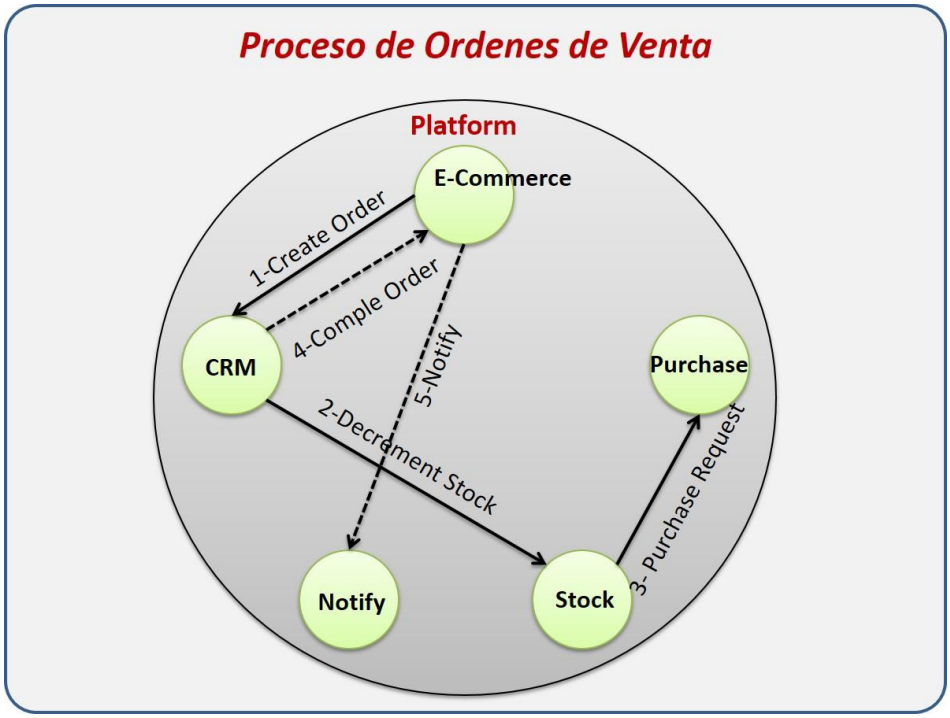


Ilustración 188: Procesamiento de órdenes de venta.

Los pasos son los siguientes:

1. Cuando un cliente realiza una compra por el *E-Commerce*, éste solicita la creación de una Orden de venta al CRM.
2. El CRM procesa la orden y solicita al módulo de inventario el decremento de las existencias.

3. El módulo de almacén realiza una solicitud de compra cuando el inventario baja por el nivel mínimo de existencias.
4. *E-Commerce* recibe la respuesta del CRM con el ID de la Orden creada.
5. En este punto CRM sigue procesando el cierre de la Orden, al pasar un tiempo determinado está completo el procesamiento de la Orden y envía la notificación del cierre al *E-Commerce*.
6. *E-Commerce* realiza una notificación al usuario por *E-mail* por medio del módulo de Notificaciones.
7. FIN.

Es importante notar que los pasos 4 y 5 son iniciados por el módulo de CRM, ya que una vez que la orden es completada le es notificado al *E-Commerce.*

La solución:

Como ya vimos, una solución con comunicación directa puede ser mala idea, por lo que nos inclinaremos a implementar el patrón Mediator para que nos ayude a mediar la comunicación entre los módulos, logrando con esto un programa mucho más claro, fácil de mantener y sobre todo eliminamos la dependencia directa entre los módulos.

Esta solución propone desarrollar la plataforma utilizando el patrón de diseño Mediator que nos permitirá hacer un componente central en la comunicación entre los módulos, evitando la comunicación directa y facilitando con ello el monitoreo y depuración de la aplicación.

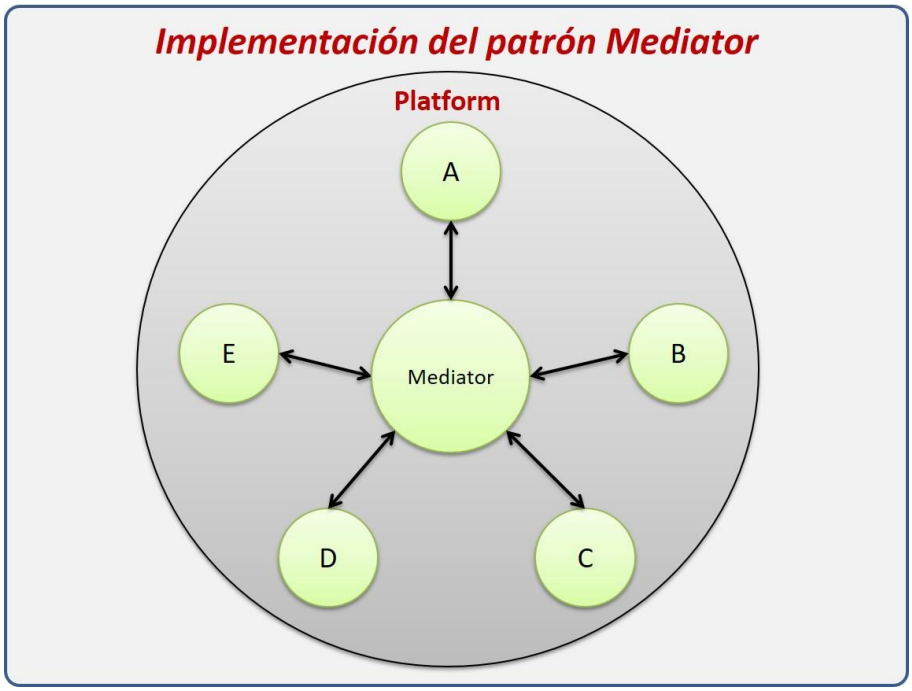


Ilustración 189: Implementación de la plataforma con el patrón Mediator.

En la imagen podemos apreciar cómo utilizamos al Mediator como pieza angular de la comunicación entre los distintos módulos de la aplicación, cuando un módulo quiere comunicarse con otro utiliza al mediador para establecer la comunicación con éste, de la misma manera, el módulo destino contesta al módulo origen a

través del mediador, es el mediador el que se encarga de enviar los mensajes al destinatario correcto.

Este diseño representa grandes ventajas y se puede apreciar claramente con sólo ver la imagen y es que la comunicación se realiza siempre por el mismo canal evitando tener que identificar cómo es que un módulo se comunica con el otro.
Una de las mayores ventajas que ofrece este diseño es que podemos desarrollar un componente que monitoreé la actividad del mediador, para analizar cómo los módulos se comunican. Podemos generar estadísticas o agregar esquemas de seguridad para impedir que ciertos módulos se comuniquen con otros o simplemente para mantener un control y que los desarrolladores no se puedan comunicar con otro módulo sin la autorización del administrador del sistema.

La implementación:

Una vez que sabemos lo que tenemos que hacer, pasaremos a la implementación de la solución.

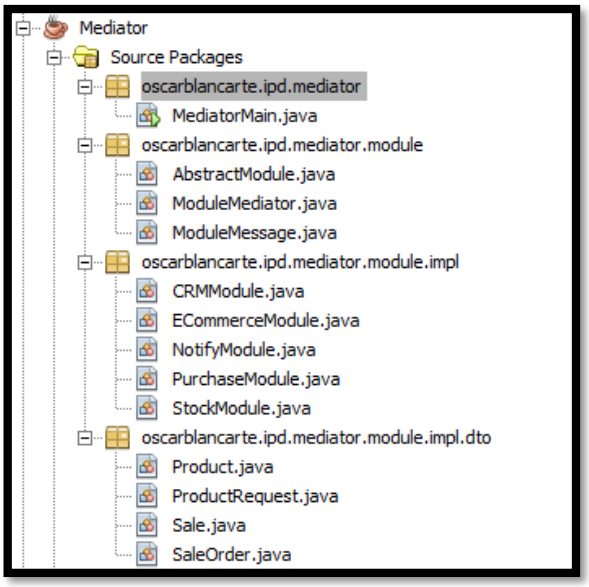


Ilustración 190: Estructura del proyecto Mediator.

El proyecto está dividido de la siguiente manera:

- **oscarblancarte.ipd.mediator:** Paquete principal del proyecto que contiene la clase ejecutable del proyecto *MediatorMain*.
- **oscarblancarte.ipd.mediator.module:** Paquete que contiene las clases base para crear los módulos de la aplicación.
- **oscarblancarte.ipd.mediator.module.impl:** Paquete que tiene todo los módulos disponibles en la aplicación.
- **oscarblancarte.ipd.mediator.module.impl.dto:** Paquete que contiene las entidades utilizadas para la comunicación entre los distintos módulos de la aplicación.

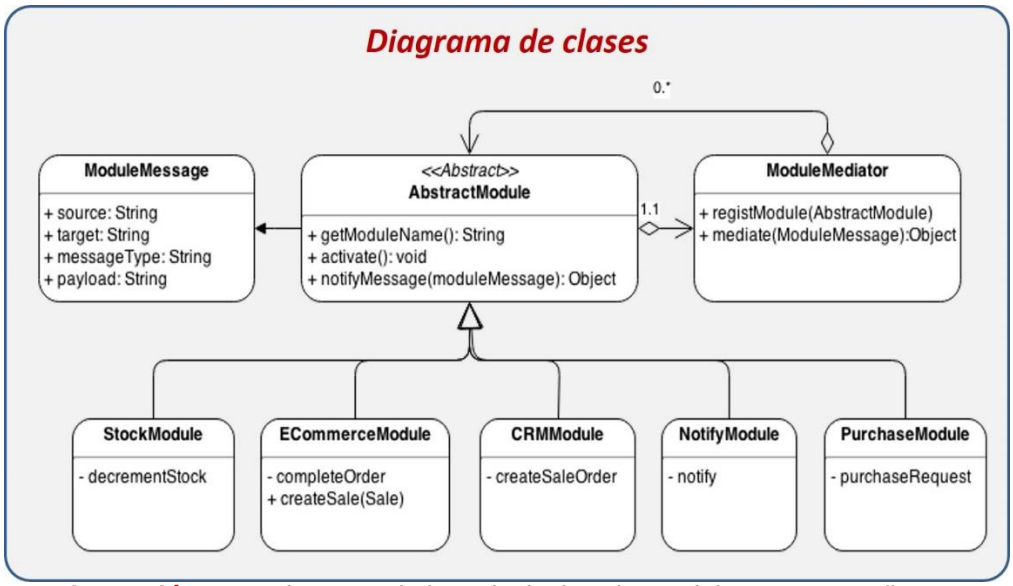


Ilustración 191: Diagrama de las principales clases del proyecto Mediator.

En el diagrama anterior podemos apreciar claramente las principales clases que componen el proyecto Mediator, los módulos que se utilizarán en la aplicación y cómo está relacionado el Mediador con los módulos.

Clase AbstractModule:

Clase que representa un módulo de forma abstracta, todos los módulos que utilicemos en la aplicación deberán heredar de esta clase.

En la clase encontramos los siguientes métodos:

- ***getModulName***: Método abstracto que deberán implementar los módulos con el fin de regresar el nombre del módulo.
- ***activate***: Método que sirve para activar el módulo dentro de la aplicación, este método registra el módulo con el Mediador con la finalidad de enviar y recibir mensajes de otros módulos.
- ***notifyMessage***: Mediante este método el módulo será notificado cuando un nuevo mensaje le sea enviado. Cada módulo deberá implementar la forma exacta en que procesará los mensajes.

```
1.  package oscarblancarte.ipd.mediator.module;
2.
3.  public abstract class AbstractModule {
4.
5.      protected ModuleMediator mediator;
6.
7.      public abstract String getModulName();
8.
9.      public void activate() {
10.         mediator = ModuleMediator.getInstance();
11.         mediator.registModule(this);
12.     }
13.
14.     public abstract Object notifyMessage(ModuleMessage message);
15. }
```

Observemos que el mediador se obtiene como una clase *Singleton* (línea 10), lo que nos permitirá compartir el mismo mediador para todos los módulos.

Clase ModuleMediator:

Esta clase es la pieza angular de este ejemplo ya que será la que se encargue de realizar la comunicación entre todos los módulos, con esto en mente ya sólo nos falta mencionar que esta clase será nuestro Mediador.

Esta clase sigue al patrón *Singleton* y sólo permitirá la creación de una única instancia, la cual será compartida por todos los módulos.

Los métodos de la clase son:

- **getInstance:** Método por el cual se obtiene o crea la única instancia de esta clase.
- **registModule:** Método desde el cual los módulos se registran al mediador con el fin de que éste sepa de su existencia y pueda dirigir los mensajes a los módulos correspondientes. Los módulos registrados son referenciados en la propiedad *modules*.
- **Mediate:** Método utilizado por los módulos para comunicarse con el resto de los módulos, este método se encarga de averiguar a quien va dirigido el mensaje y lo reenvía al destinatario correcto.

```
1.  package oscarblancarte.ipd.mediator.module;
2.
3.  import java.util.HashMap;
4.  import java.util.Map;
5.
6.  public class ModuleMediator {
7.
8.      private static ModuleMediator mediator;
9.
10.     private static final Map<String, AbstractModule> modules = new HashMap<>();
11.
12.     private ModuleMediator() {
13.     }
14.
15.     public static synchronized ModuleMediator getInstance() {
16.         if (mediator == null) {
17.             mediator = new ModuleMediator();
18.         }
19.         return mediator;
20.     }
21.
22.     public void registModule(AbstractModule module) {
23.         modules.put(module.getModulName(), module);
24.     }
25.
26.     public Object mediate(ModuleMessage mediateEvent) {
27.         if (!modules.containsKey(mediateEvent.getTarget())) {
28.             throw new RuntimeException("El módulo '"
29.                     + mediateEvent.getTarget() + "' no está registrado");
30.         }
31.         System.out.println("Mediate source > '" + mediateEvent.getSource()
32.                 + "', target > '" + mediateEvent.getTarget()
33.                 + "', messagetType > '"+mediateEvent.getMessageType()+"'");
34.         AbstractModule module = modules.get(mediateEvent.getTarget());
35.         return module.notifyMessage(mediateEvent);
36.     }
37. }
```

Observemos la línea 27, la cual se asegura que el destinatario del mensaje esté registrado en el Mediador, de lo contrario lanza un error.

También podemos ver que de las líneas 31 a la 33 podemos hacer acciones que nos permitan depurar la aplicación, es este caso sólo se imprimen en pantalla los eventos, sin embargo, podrías crear mecanismos de monitoreo mucho más complejos que nos ayuden a depurar o auditar tiempos de respuesta, valores enviados y valores devueltos en cada ejecución, etcétera.

En la línea 34 finalmente es localizado el módulo al que va dirigido el mensaje para ser enviado en la línea 35, de igual forma la respuesta del destinatario es devuelta al módulo remitente.

Clase ModuleMessage:

Esta clase es complementaria de la clase *ModuleMediator* ya que es utilizada por los módulos para enviar mensajes de un módulo a otro por medio del Mediador.

La clase contiene las siguientes propiedades:

- **Source:** Indica el nombre del módulo que envío el mensaje.
- **Target:** Indica el nombre del módulo al que va dirigido el mensaje.
- **messageType:** Indica el tipo de mensaje que se envió con el fin de poder ser atendido de forma esperada.
- **Payload:** Objeto que representa el mensaje como tal, podría ser cualquier objeto en tiempo de ejecución, por lo que el *messageType* juega un papel importante para determinar el tipo de mensaje que fue enviado.

```
1.  package oscarblancarte.ipd.mediator.module;
2.
3.  public class ModuleMessage {
4.      private String source;
5.      private String target;
6.      private String messageType;
7.      private Object payload;
8.
9.      public ModuleMessage(String source, String target, String messageType,
10.             Object payload) {
11.         this.source = source;
12.         this.target = target;
13.         this.messageType = messageType;
14.         this.payload = payload;
15.     }
16.
17.     /** GET AND SET */
18. }
```

Clase Product:

Clase que representa un producto. La única propiedad de esta clase es el nombre del producto.

```
1.  package oscarblancarte.ipd.mediator.module.impl.dto;
2.
3.  public class Product {
4.      private String name;
5.
6.      public Product(String name) {
7.          this.name = name;
8.      }
9.
10.     /** GET AND SET */
11. }
```

Clase Sale:

Clase que representa una venta, la clase únicamente tiene un listado de productos vendidos.

```
1.  package oscarblancarte.ipd.mediator.module.impl.dto;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class Sale {
7.
8.      protected List<Product> productos = new ArrayList<>();
9.
10.     public List<Product> getProductos() {
11.         return productos;
12.     }
13.
14.     public void addProduct(Product product) {
15.         this.productos.add(product);
16.     }
17.
18.     public void setProductos(List<Product> productos) {
19.         this.productos = productos;
20.     }
21.
22. }
```

Clase ProductRequest:

Clase que es utilizada para solicitar la creación de una orden de compra al módulo de compras tras la venta de artículos por medio del *ECommerce*.

```
1.  package oscarblancarte.ipd.mediator.module.impl.dto;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class ProductRequest {
7.      public List<Product> products = new ArrayList<>();
8.
9.      public List<Product> getProducts() {
10.         return products;
11.     }
12.
13.     public void setProducts(List<Product> products) {
14.         this.products = products;
15.     }
16. }
```

Clase SaleOrder:

Clase que utilizaremos para crear la orden de venta, ésta extiende a la clase *Sale* y le agrega únicamente un ID para identificar cada Orden. Esta clase será utilizada por el CRM para crear la Orden de venta a partir de la venta generada por el *ECommerce*.

```
1.  package oscarblancarte.ipd.mediator.module.impl.dto;
2.
3.  public class SaleOrder extends Sale {
4.
5.      private String id;
6.
7.      public SaleOrder(String id) {
8.          this.id = id;
9.      }
10.
11.     /** GET AND SET */
12. }
```

Clase ECommerceModule:

Este es el módulo principal de este ejemplo ya que será por medio de él que dispararemos la ejecución al resto de los demás módulos.

Las propiedades que tenemos son las siguientes:

- **MODULE_NAME:** Nos indica el nombre del módulo.
- **OPERATION_COMPLETE_ORDER:** Constante que indica el nombre de la operación disponible para este módulo.

Los métodos disponibles son los siguientes:

- **getModuleName:** Regresa el nombre del módulo.
- **notifyMessage:** Método que el Mediador utilizará para notificar el módulo de un nuevo mensaje. Utiliza la propiedad de *messageType* (línea 19) para identificar qué operación se realizará con el mensaje entrante, si el tipo de mensaje no es identificado entonces se lanza un error.
- **createSale:** Método utilizado para crear una nueva venta, en las líneas 40 y 41 vemos cómo se crea un *ModuleMessage* con el fin de enviar un mensaje al módulo de CRM, para lo cual se establece en la propiedad *source* el nombre de este módulo (módulo que envía) y como *target* el nombre del módulo de CRM (Módulo destinatario), también indicamos el *operationType* con el nombre de la operación disponible por CRM y finalmente enviamos el *payload* que es la venta como tal.
- **completeOrder:** Esta operación es llamada por CRM cuando la orden de venta se ha completado para continuar con el flujo de notificación al usuario.

```
1.  package oscarblancarte.ipd.mediator.module.impl;
2.
3.  import oscarblancarte.ipd.mediator.module.*;
4.  import oscarblancarte.ipd.mediator.module.impl.dto.*;
5.
6.  public class ECommerceModule extends AbstractModule {
7.
8.      public static final String MODULE_NAME = "ECommerce";
```

```
9.
10.     public static final String OPERATION_COMPLETE_ORDER = "CompleteOrder";
11.
12.     @Override
13.     public String getModulName() {
14.         return MODULE_NAME;
15.     }
16.
17.     @Override
18.     public Object notifyMessage(ModuleMessage message) {
19.         switch (message.getMessageType()) {
20.             case OPERATION_COMPLETE_ORDER:
21.                 return compleOrder(message);
22.             default:
23.                 throw new RuntimeException("Operación no soportada '" +
24.                         message.getMessageType() + "'");
25.         }
26.     }
27.
28.     private String compleOrder(ModuleMessage message){
29.         SaleOrder saleOrder = (SaleOrder)message.getPayload();
30.         System.out.println("Orden completada exitosamente > " );
31.
32.         ModuleMessage crmMessage = new ModuleMessage(MODULE_NAME,
33.                 NotifyModule.MODULE_NAME, NotifyModule.OPERATION_NOTIFY,
34.                 saleOrder);
35.         mediator.mediate(crmMessage);
36.         return saleOrder.getId();
37.     }
38.
39.     public String createSale(Sale sale){
40.         ModuleMessage crmMessage = new ModuleMessage(MODULE_NAME,
41.                 CRMModule.MODULE_NAME, CRMModule.OPERATION_CREATE_ORDER, sale);
42.         return mediator.mediate(crmMessage).toString();
43.     }
44.
45. }
```

Clase CRMModule:

Esta clase representa el módulo de CRM que utilizaremos para crear una orden de venta a partir de una venta del *ECommerce*. Las propiedades de la clase son las siguientes:

- **MODULE_NAME:** Constante con el nombre del módulo.
- **OPERATION_CREATE_ORDER:** Constante utilizada para identificar la operación de creación de la orden.

Las operaciones son las siguientes:

- **getModuleName:** Constante que representa el nombre del módulo.
- **notifyMessage:** Método utilizado por el Mediador para informar al módulo de un nuevo mensaje. Este módulo utiliza la propiedad *messageType* para identificar el tipo de mensaje y utilizar la operación adecuada para procesarlo.
- **createSalesOrder:** Método utilizado para crear una orden de venta a partir de una venta. Esta operación crea un objeto de tipo *SaleOrder* (línea 34) y solicita al módulo de Almacén la reducción del inventario (líneas 37 a 40).

Después espera 10 segundos simulando que la orden se está cerrando en segundo plano (línea 46), mientras este tiempo transcurre el modulo regresa el ID de la Orden creada (línea 59), después de 10 segundos CRM envía un mensaje al *ECommerce* notificándole que la orden se ha cerrado con éxito (líneas 47 a 51).

```
1.  package oscarblancarte.ipd.mediator.module.impl;
2.
3.  import java.util.UUID;
4.  import java.util.logging.*;
5.  import oscarblancarte.ipd.mediator.module.*;
6.  import oscarblancarte.ipd.mediator.module.impl.dto.*;
7.
8.  public class CRMModule extends AbstractModule {
9.
10.     public static final String MODULE_NAME = "CRM";
11.
12.     public static final String OPERATION_CREATE_ORDER = "CreateOrder";
13.
14.     @Override
15.     public String getModulName() {
16.         return MODULE_NAME;
17.     }
18.
19.     @Override
20.     public Object notifyMessage(ModuleMessage message) {
21.         switch (message.getMessageType()) {
22.             case OPERATION_CREATE_ORDER:
23.                 return createSaleOrder(message);
24.             default:
```

```
25.                 throw new RuntimeException("Operación no soportada '"
26.                         + message.getMessageType() + "'");
27.         }
28.     }
29.
30.     private String createSaleOrder(ModuleMessage event) {
31.         Sale sale = (Sale) event.getPayload();
32.         String ID = UUID.randomUUID().toString();
33.         System.out.println("Orden de venta creada exitosamente");
34.         final SaleOrder saleOrder = new SaleOrder(ID);
35.         saleOrder.setProductos(sale.getProductos());
36.
37.         ModuleMessage stockEvent = new ModuleMessage(MODULE_NAME,
38.                 StockModule.MODULE_NAME, StockModule.OPERATION_DECREMENT_STOCK,
39.                 saleOrder);
40.         mediator.mediate(stockEvent);
41.
42.         new Thread(new Runnable() {
43.             @Override
44.             public void run() {
45.                 try {
46.                     Thread.sleep(1000 * 10);
47.                     ModuleMessage stockEvent = new ModuleMessage(MODULE_NAME,
48.                             ECommerceModule.MODULE_NAME,
49.                             ECommerceModule.OPERATION_COMPLETE_ORDER,
50.                             saleOrder);
51.                     mediator.mediate(stockEvent);
52.                 } catch (InterruptedException ex) {
53.                     Logger.getLogger(CRMModule.class.getName())
54.                             .log(Level.SEVERE, null, ex);
55.                 }
56.             }
57.         }).start();
58.
59.         return ID;
60.     }
61.
62. }
```

Clase StockModule:

Clase que representa el módulo de Almacén, mediante este módulo se simula afectar los inventarios tras una orden de compra.

Los atributos de la clase son:

- **MODULE_NAME:** Constante que contiene el nombre del módulo.

- **OPERATION_DECREMENT_STOCK:** Constante con la que identificaremos la operación de decremento de existencia.

Las operaciones son las siguientes:

- **getModuleName:** Operación que regresa el nombre del módulo.
- **notifyMessage:** Método utilizado por el Mediador para informar al módulo de un nuevo mensaje. Este módulo utiliza la propiedad *messageType* para identificar el tipo de mensaje y utilizar la operación adecuada para procesarlo.
- **decrementStock:** Operación que simula afectar el inventario y además solicita al módulo de Compras la reposición de los artículos vendidos (líneas 36 a 39).

```
1.  package oscarblancarte.ipd.mediator.module.impl;
2.
3.  import oscarblancarte.ipd.mediator.module.*;
4.  import oscarblancarte.ipd.mediator.module.impl.dto.*;
5.
6.  public class StockModule extends AbstractModule {
7.
8.     public static final String MODULE_NAME = "Stock";
9.     public static final String OPERATION_DECREMENT_STOCK = "DecrementStock";
10.
11.    @Override
12.    public String getModulName() {
13.       return MODULE_NAME;
14.    }
15.
16.    @Override
17.    public Object notifyMessage(ModuleMessage message) {
18.       switch (message.getMessageType()) {
19.          case OPERATION_DECREMENT_STOCK:
20.             return decrementStock(message);
21.          default:
22.             throw new RuntimeException("Operación no soportada '"
23.                   + message.getMessageType() + "'");
24.       }
25.    }
26.
27.    private Void decrementStock(ModuleMessage message) {
28.       SaleOrder saleOrder = (SaleOrder) message.getPayload();
29.       for (Product product : saleOrder.getProductos()) {
30.          System.out.println("decrement product > " + product.getName());
```

```
31.         }
32.
33.         ProductRequest productRequest = new ProductRequest();
34.         productRequest.setProducts(saleOrder.getProductos());
35.
36.         ModuleMessage purchaseMessage = new ModuleMessage(MODULE_NAME,
37.                 PurchaseModule.MODULE_NAME,
38.                 PurchaseModule.OPERATION_PURCHASE_REQUEST, productRequest);
39.         mediator.mediate(purchaseMessage);
40.         return null;
41.
42.     }
43. }
```

Clase PurchaseModule:

Clase que representa el módulo de compras.

Las propiedades son las siguientes:

- **MODULE_NAME:** Constante que contiene el nombre del módulo.
- **OPERATION_PURCHASE_REQUEST:** Constante con la que identificaremos la operación de requerimiento de compra.

Las operaciones son las siguientes:

- **getModuleName:** Operación que regresa el nombre del módulo.
- **notifyMessage:** Método utilizado por el Mediador para informar al módulo de un nuevo mensaje. Este módulo utiliza la propiedad *messageType* para identificar el tipo de mensaje y utilizar la operación adecuada para procesarlo.
- **purchaseRequest:** En esta operación no realizamos nada, solamente regresamos *null*, esta operación es el final de la ejecución iniciada desde *ECommerce*.

```
1.  package oscarblancarte.ipd.mediator.module.impl;
2.
3.  import oscarblancarte.ipd.mediator.module.*;
4.
```

```
5.  public class PurchaseModule extends AbstractModule {
6.
7.      public static final String MODULE_NAME = "Chopping";
8.      public static final String OPERATION_PURCHASE_REQUEST = "PurchaseRequest";
9.
10.     @Override
11.     public String getModulName() {
12.         return MODULE_NAME;
13.     }
14.
15.     @Override
16.     public Object notifyMessage(ModuleMessage message) {
17.         switch (message.getMessageType()) {
18.             case OPERATION_PURCHASE_REQUEST:
19.                 return purchaseRequest(message);
20.             default:
21.                 throw new RuntimeException("Operación no soportada '"
22.                         + message.getMessageType() + "'");
23.         }
24.     }
25.
26.     private Void purchaseRequest(ModuleMessage message){
27.         return null;
28.     }
29. }
```

Clase MotifyModule:

Clase que representa al módulo de notificaciones, éste simula enviar correos electrónicos a los clientes una vez que la orden de compra se ha cerrado con éxito.

Las propiedades son las siguientes:

- **MODULE_NAME:** Constante que contiene el nombre del módulo.
- **OPERATION_NOTIFY:** Constante con la que identificaremos la operación de notificación.

Las operaciones son las siguientes:

- **getModuleName:** Operación que regresa el nombre del módulo.
- **notifyMessage:** Método utilizado por el Mediador para informar al módulo de un nuevo mensaje. Este módulo utiliza la propiedad

messageType para identificar el tipo de mensaje y utilizar la operación adecuada para procesarlo.

- **notify:** En esta operación sólo imprimimos en pantalla la confirmación de que fue enviada la notificación al cliente.

```
1.  package oscarblancarte.ipd.mediator.module.impl;
2.
3.  import oscarblancarte.ipd.mediator.module.*;
4.
5.  public class NotifyModule extends AbstractModule {
6.
7.      public static final String MODULE_NAME = "Notification";
8.      public static final String OPERATION_NOTIFY = "Notify";
9.
10.     @Override
11.     public String getModulName() {
12.         return MODULE_NAME;
13.     }
14.
15.     @Override
16.     public Void notifyMessage(ModuleMessage message) {
17.         switch (message.getMessageType()) {
18.             case OPERATION_NOTIFY:
19.                 return notify(message);
20.             default:
21.                 throw new RuntimeException("Operación no soportada '"
22.                         + message.getMessageType() + "'");
23.         }
24.     }
25.
26.     private Void notify(ModuleMessage message) {
27.         System.out.println("Notificación por EMail enviada");
28.         return null;
29.     }
30.
31. }
```

Clase MediatorMain:

Clase principal del proyecto que utilizaremos para comprobar la efectividad del patrón de diseño Mediator.

```
1.  package oscarblancarte.ipd.mediator;
2.
3.  import oscarblancarte.ipd.mediator.module.impl.*;
4.  import oscarblancarte.ipd.mediator.module.impl.dto.*;
5.
6.  public class MediatorMain {
7.
8.      public static void main(String[] args) {
9.          new CRMModule().activate();
10.         new NotifyModule().activate();
11.         new StockModule().activate();
12.         new PurchaseModule().activate();
13.
14.         ECommerceModule client = new ECommerceModule();
15.         client.activate();
16.
17.         Sale sale = new Sale();
18.         for(int c=0;c<5;c++){
19.             sale.addProduct(new Product("Product" + (c+1)));
20.         }
21.         client.createSale(sale);
22.     }
23.
24. }
```

En las líneas 9 a la 12 creamos los módulos utilizados en la aplicación y los activamos mediante el método *actívate* el cual, como ya hablamos, sirve para iniciar el módulo y registrarlo con el Mediador.

En la líneas 14 y 15 creamos y activamos el módulo de *ECommerce* y que utilizaremos para iniciar le ejecución de toda la comunicación con el resto de módulos.

De las líneas 17 a la 20 creamos la venta mediante la clase *Sale* y le agregamos 5 productos auto generados. Finalmente, en la línea 21 mandamos llamar el método *createSale* y enviamos la venta creada anteriormente, esto iniciará automáticamente la ejecución con el resto de módulos por medio del Mediador.

La ejecución:

En este punto hemos terminado de implementar el programa y comenzaremos con la ejecución. Para esto iniciaremos el programa por medio de la clase *MediatorMain* y veremos el siguiente resultado:

```
run:
Mediate source > 'ECommerce', target > 'CRM', messagetType > 'CreateOrder'
Orden de venta creada exitosamente
Mediate source > 'CRM', target > 'Stock', messagetType > 'DecrementStock'
decrement product > Product1
decrement product > Product2
decrement product > Product3
decrement product > Product4
decrement product > Product5
Mediate source > 'Stock', target > 'Chopping', messagetType > 'PurchaseRequest'
```

Ilustración 192: Resultado de la ejecución.

Podemos apreciar como *ECommerce* envía un mensaje a CRM mediante la operación *CreateOrder*, la cual toma el mediador y la envía al CRM.

CRM indica que la orden fue creada exitosamente.

CRM envía a su vez un mensaje al módulo de Almacén (*Stock*) mediante la operación *DecrementStock* y es enviada al mediador para que éste a su vez, le envíe el mensaje al módulo destino.

Seguido, vemos cómo el módulo de almacén decrementa los artículos del mismo.

Despúes, vemos cómo el módulo de almacén envía un mensaje al módulo de compras para solicitar la reposición de los artículos vendidos.

Luego de ver esto, podemos esperar 10 segundos con la finalidad de que CRM termine de cerrar la orden y enviar la notificación a *ECommerce*.

```
Mediate source > 'CRM', target > 'ECommerce', messagetType > 'CompleteOrder'
Orden completada exitosamente >
Mediate source > 'ECommerce', target > 'Notification', messagetType > 'Notify'
Notificación por EMail enviada
BUILD SUCCESSFUL (total time: 10 seconds)
```

Ilustración 193: Finalización de la ejecución.

Finalmente, cuando la orden se cierra, podemos ver cómo CRM envía un mensaje al *ECommerce* por medio de la operación *CompleteOrder* por medio del mediador. *ECommerce* envía una notificación al cliente por medio del módulo *Notify* mediante la operación *Notification*.

Como ya vimos, toda la comunicación se llevó a cabo por medio del mediador y los demás módulos no tienen por qué conocer las clases de los demás módulos, lo que sin duda es una gran ventaja.

Algo que nos puede llegar a parecer complicado es el método *mediate*, ya que debe de recibir el tipo de objeto, el destinatario y el remitente. Esto de entrada puede parecer molesto, sin embargo, podemos crear los mediadores de tal forma que tengan firmas concretas para cada operación de la aplicación, en este caso se implementa de esta manera debido a que es una forma genérica de comunicación.

Utilizando un mensaje genérico evitamos la necesidad de modificar al Mediador cuando una nueva operación es agregada al sistema, en otro caso, tendríamos que modificar el mediador para agregar la nueva firma de la operación. Sin embargo, las dos formas son válidas y dependerá de tu criterio para determinar que es para ti lo más conveniente.

Siguientes pasos:

Como podemos apreciar, la implementación del Mediador es una solución bastante simple con grandes beneficios, entre los que destacan la reducción de la dependencia entre las clases, depuración de aplicaciones y sobre todo que controlamos la forma en que las clases de comunican entre sí.

Para fortalecer estos conocimientos realizaremos los siguientes ejercicios:

- Agreguemos un nuevo módulo de distribución que programe el envío del producto a nuestro cliente. El módulo deberá ser llamado por el CRM una vez que la orden es completada.

- Modifiquemos la clase mediadora para determinar el tiempo de ejecución de cada operación, de esta manera podremos identificar qué operaciones tardan más y cuáles menos.

- Modifiquemos el programa para que el mediador defina métodos específicos por cada operación, de esta forma tendremos una operación por cada operación de cada módulo y los parámetros y respuestas serán acorde a cada uno.

Patrón Memento

Memento es un patrón diseñado que nos permite capturar el estado de un objeto en un momento determinado con la finalidad de regresar a este estado en cualquier momento. Este patrón es utilizado cuando tenemos objetos que cambian en el tiempo y por alguna razón necesitamos restaurar su estado en un momento determinado.

Un ejemplo claro de éstos es un editor de texto. Cada vez que escribimos una letra el editor de texto guarda un estado del documento, el cual nos permite viajar en los estados deshaciendo o rehaciendo los cambios que hemos hecho.

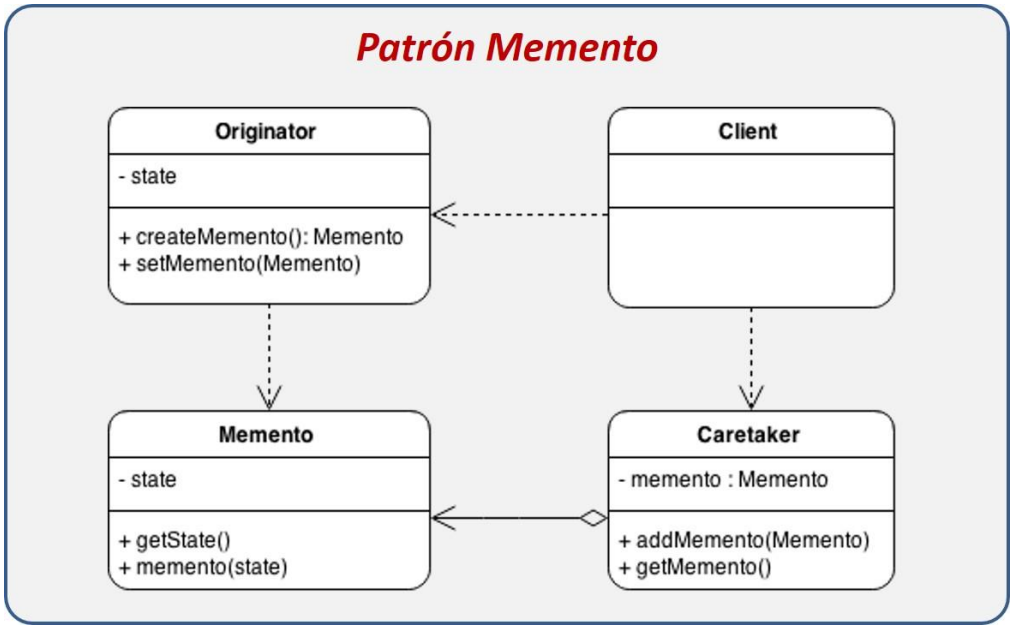


Ilustración 194: Patrón de diseño Memento.

Los componentes del patrón se explican a continuación:

- **Client:** Componente que afecta al *Originator* y registra el nuevo estado con el *Caretaker*. En otras palabras, es quien realiza el cambio sobre el objeto y registra el estado.
- **Originator:** Es el componente que cambia de estado.

- **Memento:** Componente que almacena el estado del *Originator* en un momento determinado.
- **Caretaker:** Componente que registra los cambios del *Originator*. Este componente nos permite viajar entre los distintos estados del *Originator*.

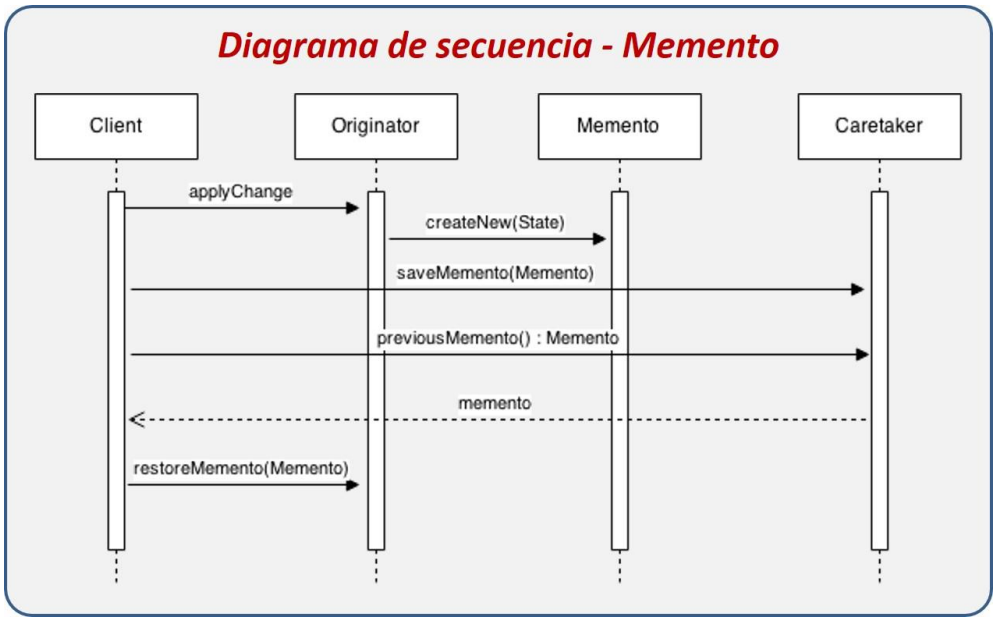


Ilustración 195: Diagrama de secuencia del patrón Memento.

El diagrama anterior se interpreta de la siguiente manera:

- El Cliente aplica un cambio sobre el *Originator*.
- El *Originator* crea un nuevo Memento que representa su estado actual.
- El Cliente guardar el Memento en el *Caretaker* para posteriormente poder cambiar entre los estados del *Originator*.
- Después de un tiempo, el Cliente solicita al *Caretaker* el estado previo del *Originator*.
- El Cliente restaura el estado del *Originator* mediante el Memento obtenido del *Caretaker*.

Como podemos apreciar en el diagrama, el patrón Memento permite guardar los estados del *Originator* para posteriormente viajar entre los distintos estados

generados durante un tiempo determinado. Una de las características que debe tener el objeto *Originator* es la funcionalidad de regresar a un estado mediante un objeto Memento.

Cuándo utilizarlo:

- Es necesario restaurar el estado pasado de un sistema.

- Cuando tenemos objetos que cambian con el tiempo y es necesario guardar su estado para poder regresar a él en cualquier momento.

El escenario:

Para este ejemplo desarrollaremos una aplicación simple de alta de Empleados la cual nos permita capturar alguno de los datos de los empleados y guardarlos. Para demostrar la utilidad del patrón incorporaremos un par de botones que nos permitan regresar a los estados anteriores de Empleados pero también regresar hasta el último estado.

En el ejemplo solicitaremos solamente tres datos del empleado: nombre, apellido y número de empleado; tendremos un formulario como el siguiente:

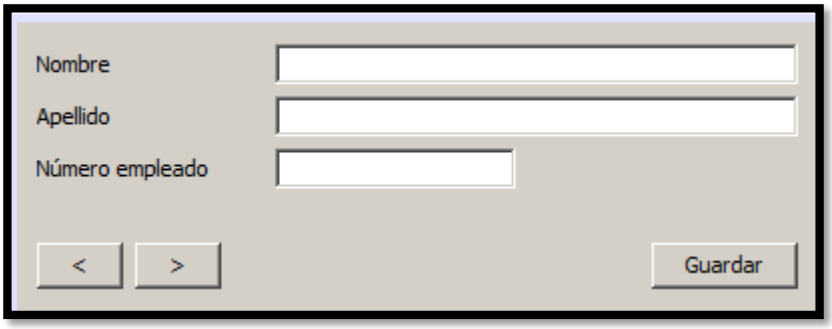


Ilustración 196: Formulario de alta de Empleados.

Cuando el usuario presione el botón de guardar, el empleado será modificado con los valores capturados y un nuevo estado se generará, éste será registrado en el *Caretaker* para posteriormente, viajar entre los estados y lo haremos con los dos botones ubicados en la parte posterior izquierda.

La solución:

Para poder implementar un mecanismo que nos ayude a guardar los estados del Empleado y posteriormente viajar entre éstos, implementaremos el patrón de diseño Memento.

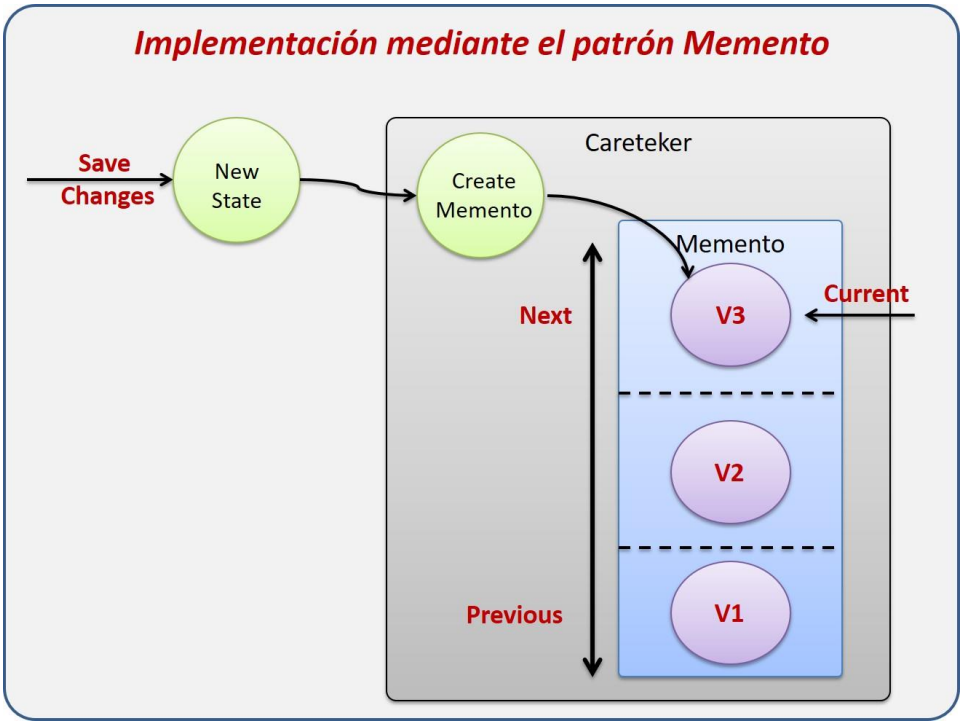


Ilustración 197: Implementación utilizando Memento.

En la imagen podemos apreciar cómo un nuevo estado es generado tras afectar el estado del Empleado, este nuevo estado es capturado por el *Caretaker* que lo almacena en una lista ligada, esta lista guardará al inicio el primer estado conocido del Empleado y al final el último estado.
Cuando necesitemos obtener los estados anteriores del Empleado viajaremos a los estados previos del *Caretaker* y para ver los estos más actuales viajaremos hacia adelante en la lista.

En este caso la misma aplicación (GUI) jugará como el cliente, pues será éste el que modifique al empleado y registre los cambios de estado.

La implementación:

Con una idea clara de lo que tenemos que hacer iniciaremos con la implementación de la solución.

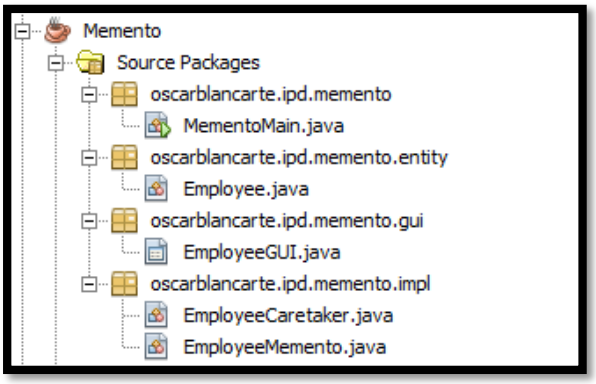


Ilustración 198: Estructura del proyecto Memento.

Los paquetes que conforman el proyecto son los siguientes:

- **oscarblancarte.ipd.memento:** Paquete principal que únicamente contiene la clase ejecutable del proyecto.
- **oscarblancarte.ipd.memento.entity:** Paquete con las Entidades del proyecto.
- **oscarblancarte.ipd.memento.gui:** Paquete con las clases de interface gráfica.
- **oscarblancarte.ipd.memento.impl:** Paquete con las clases bases del patrón Memento.

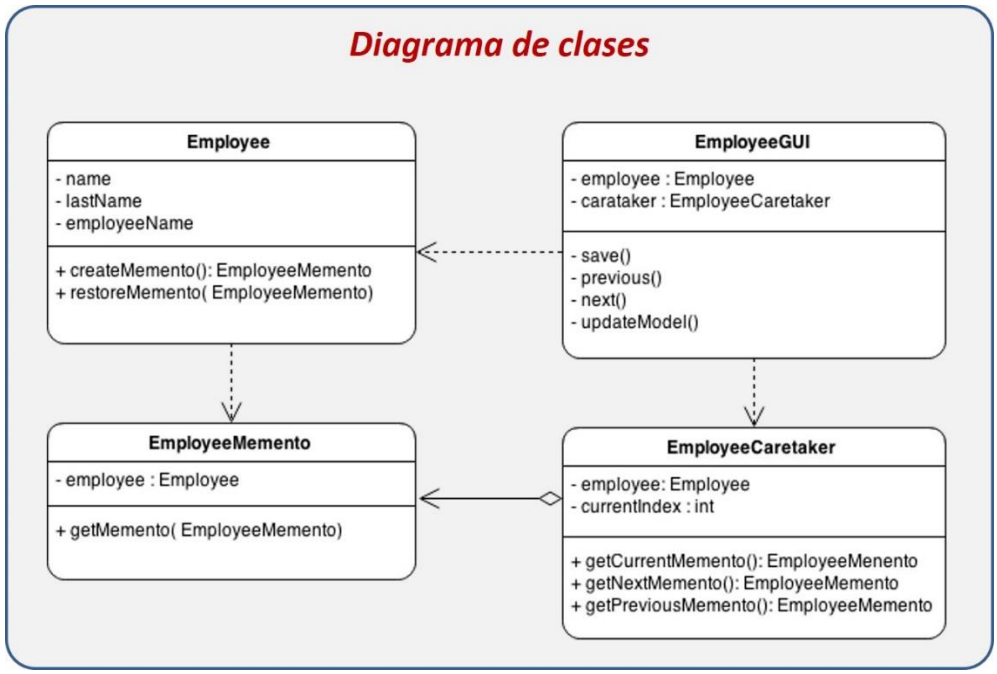


Ilustración 199: Diagrama de clases del proyecto Memento.

Clase Employee:

La clase *Employee* representa al *Originator* debido a que es la clase que cambia de estado tras ser modificada desde el formulario.

La clase tiene los atributos: nombre (*name*), apellido (*lastName*) y número de empleado (*employeeNumber*); los cuales corresponden a los campos del formulario.

Los métodos de la clase son los siguientes:

- **createMemento:** Método que crea un nuevo estado del Empleado basado en el estado actual, para esto crea una instancia de la clase Memento a partir del estado actual del empleado.
- **restoreMemento:** Método que restaura el Empleado a un estado conocido, para esto le envía el Memento con el estado que se desea restaurar.

```
1.  package oscarblancarte.ipd.memento.entity;
2.
3.  import oscarblancarte.ipd.memento.impl.EmployeeMemento;
4.
5.  public class Employee implements Cloneable{
6.      private String name;
7.      private String lastName;
8.      private String employeeNumber;
9.
10.     public Employee(String name, String lastName, String employeeNumber) {
11.         this.name = name;
12.         this.lastName = lastName;
13.         this.employeeNumber = employeeNumber;
14.     }
15.
16.     public Employee() {
17.     }
18.
19.     /** GET AND SET */
20.
21.     public EmployeeMemento createMemento(){
22.         try {
23.             return new EmployeeMemento((Employee)this.clone());
24.         } catch (Exception e) {
25.             throw new RuntimeException("Error al clonar el empleado");
26.         }
27.     }
28.
29.     public void restoreMemento(EmployeeMemento memento){
30.         Employee employee = memento.getMemento();
31.         this.name = employee.name;
32.         this.lastName = employee.name;
33.         this.employeeNumber = employee.employeeNumber;
34.     }
35. }
```

Clase EmployeeMemento:

Clase utilizada para guardar un estado del Empleado, esta clase recibe como parámetro del constructor el objeto Empleado que utilizará como el estado.

```
1.  package oscarblancarte.ipd.memento.impl;
2.
3.  import oscarblancarte.ipd.memento.entity.Employee;
4.
5.  public class EmployeeMemento {
6.
7.      public Employee employee;
```

```
8.
9.     public EmployeeMemento(Employee employee){
10.         this.employee = employee;
11.     }
12.
13.     public Employee getMemento(){
14.         return employee;
15.     }
16. }
```

Clase EmployeeCaretaker:

Esta clase representa el *Caretaker* que se encarga de almacenar los estados por lo que ha pasado el Empleado, por otra parte nos permite ver los estados anteriores y posteriores del Empleado.

Los atributos de la clase son los siguientes:

- **currentIndex:** Indica el estado actual del Empleado. A medida que avanzamos a estados más nuevos, el índice aumenta y disminuye al retroceder en los estados.
- **States:** Lista con todos los estados registrados del Empleado.

Los métodos son los siguientes:

- **addNewMemento:** Operación que nos permite registrar nuevos estados del Empleado.
- **getCurrentMemento:** Regresa el estado actual del Empleado.
- **getNextMemento**: Permite obtener el estado más reciente del Empleado.
- **getPreviousMemento:** Permite obtener un estado posterior del Empleado.

```
1.  package oscarblancarte.ipd.memento.impl;
2.
3.  import java.util.LinkedList;
4.
5.  public class EmployeeCaretaker {
6.      private int currentIndex = -1;
7.
```

```
8.      private final LinkedList<EmployeeMemento> states = new LinkedList<>();
9.
10.     public void addNewMemento(EmployeeMemento m) {
11.         states.add(m);
12.         currentIndex++;
13.     }
14.     public EmployeeMemento getCurrentMemento() {
15.         return states.get(currentIndex);
16.     }
17.
18.     public EmployeeMemento getNextMemento() {
19.         int newIndex = currentIndex +1;
20.         if( newIndex >= states.size()){
21.             return null;
22.         }
23.         currentIndex = newIndex;
24.         return states.get(currentIndex);
25.     }
26.
27.     public EmployeeMemento getPreviousMemento() {
28.         int newIndex = currentIndex-1;
29.
30.         if(newIndex  <= -1 || newIndex >= states.size()-1){
31.             return null;
32.         }
33.         currentIndex= newIndex;
34.         return states.get(currentIndex);
35.     }
36. }
```

Clase EmployeeGUI:

Esta clase la utilizamos para crear la interface gráfica del formulario, sin embargo, hemos omitido cómo crearla para dejar únicamente lo relevante en el ejemplo.

Los atributos relevantes de la clase son:

- **employee:** Objeto con los datos del empleado, esta clase representa el *Originator* y será la que se afecte tras guardar los cambios desde el formulario.
- **caretaker:** Objeto que utilizaremos para guardar los estados del Empleado.

Los métodos relevantes son:

- **Save:** Operación que registra los cambios del formulario en el objeto empleado (líneas 17 a 19) seguido, crea un Memento del empleado y lo registra en el *caretaker* (línea 21). Esta operación es ejecutada al presionar el botón de guardar.
- **Previous:** Operación que permite retroceder en los estados registrados mediante la obtención del estado anterior (línea 28), seguido de restaurar el estado del empleado con el Memento obtenido (línea 33) y finalmente, actualizar la pantalla con el nuevo estado con ayuda del método updateModel (línea 34).
- **Next:** Operación que permite adelantarse en los estados registrados mediante la obtención del estado siguiente (línea 40), seguido de restaurar el estado del empleado con el Memento obtenido (línea 45) y finalmente actualizar la pantalla con el nuevo estado mediante el método updateModel (línea 46).
- **UpdateModel:** Operación que actualiza los campos del formulario con el nuevo estado del objeto empleado.

```
1.  package oscarblancarte.ipd.memento.gui;
2.
3.  import oscarblancarte.ipd.memento.entity.Employee;
4.  import oscarblancarte.ipd.memento.impl.EmployeeCaretaker;
5.  import oscarblancarte.ipd.memento.impl.EmployeeMemento;
6.
7.  public class EmployeeGUI extends javax.swing.JFrame {
8.      Employee employee = new Employee();
9.      private EmployeeCaretaker caretaker = new EmployeeCaretaker();
10.
11.     public EmployeeGUI() {
12.         initComponents();
13.     }
14.
15.
16.     private void save(java.awt.event.ActionEvent evt) {
17.         employee.setName(nameTxt.getText());
18.         employee.setLastName(lastNameTxt.getText());
19.         employee.setEmployeeNumber(empNumberTxt.getText());
20.
21.         caretaker.addNewMemento(employee.createMemento());
22.
```

```
23.         JOptionPane.showMessageDialog(this, "Estado guardado");
24.     }
25.
26.     private void previous(java.awt.event.ActionEvent evt) {
27.
28.         EmployeeMemento menento = caretaker.getPreviousMemento();
29.         if(menento==null){
30.             JOptionPane.showMessageDialog(this, "No existen más estados");
31.             return;
32.         }
33.         employee.restoreMemento(menento);
34.         updateModel();
35.     }
36.
37.     private void next(java.awt.event.ActionEvent evt) {
38.         // TODO add your handling code here:
39.
40.         EmployeeMemento memento = caretaker.getNextMemento();
41.         if(memento==null){
42.             JOptionPane.showMessageDialog(this, "No existen más estados");
43.             return;
44.         }
45.         employee.restoreMemento(memento);
46.         updateModel();
47.     }
48.
49.     private void updateModel(){
50.         nameTxt.setText(employee.getName());
51.         lastNameTxt.setText(employee.getLastName());
52.         empNumberTxt.setText(employee.getEmployeeNumber());
53.     }
54.
55. }
```

Clase MementoMain:

Clase principal del proyecto utilizada únicamente para iniciar la interface gráfica.

```
1.  package oscarblancarte.ipd.memento;
2.
3.  import oscarblancarte.ipd.memento.gui.EmployeeGUI;
4.
5.  public class MementoMain {
6.
7.      public static void main(String[] args) {
8.          java.awt.EventQueue.invokeLater(new Runnable() {
9.              public void run() {
10.                 new EmployeeGUI().setVisible(true);
```

```
11.             }
12.         });
13.     }
14. }
```

La ejecución:

Para empezar a utilizar la aplicación, lo primero que tendremos que hacer será ejecutar la clase *MementoMain* para ver una pantalla como la siguiente:

Ilustración 200: Formulario de Empleados.

Lo siguiente que haremos será crear un empleado con los datos: Juan, Perez, 1111 y guardaremos el registro.

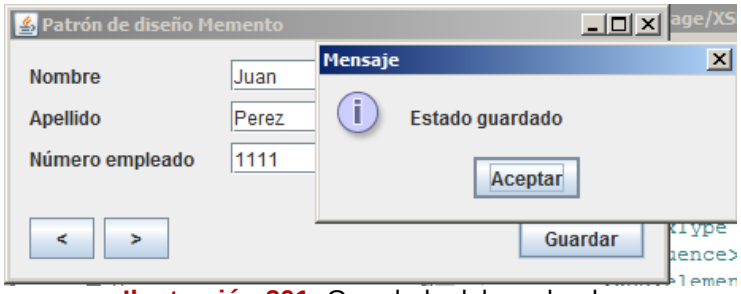


Ilustración 201: Guardado del empleado.

Luego, editar los campos del cliente con algún otro valor y guardarlo (repetiremos esta acción las veces que queramos).

Cuando hayamos guardado los datos del empleado al menos unas 2 veces utilizaremos los botones **<** y **>** para ver cómo es que podemos cambiar al estado que estábamos hace un momento, los botones nos permitirán ver desde el primer estado hasta el último.

Siguientes pasos:

El patrón de diseño Memento puede ser un buen aliado cuando necesitamos crear aplicaciones que cambian constantemente el estado de sus objetos y es necesario tener un mecanismo que nos ayude a retroceder o avanzar en los cambios que hemos realizado en un tiempo determinado.

Para fortalecer los conocimientos adquiridos realizaremos las siguientes actividades:

- Agreguemos una propiedad más al empleado, que guarde el número de cliente. Éste deberá verse reflejado en la pantalla y deberá mostrarse cuando un estado sea restaurado.

- Agreguemos un nivel más de complejidad a la clase de empleado, por lo cual agregaremos dentro de la clase una referencia a un objeto de dirección del cliente. La dirección deberá ser una nueva clase que tenga los campos: dirección 1, dirección 2, ciudad, código postal y el país. Estos campos deberán reflejarse en la pantalla y actualizarse cuando un estado sea establecido.

- Agreguemos un nuevo botón a la pantalla que nos permita borrar el estado actual, para lo cual debemos de eliminar dicho estado del *Caretaker*.

Patrón Null Object

El patrón de diseño Null Object nace de la necesidad de evitar los valores nulos que puedan originar error en tiempo de ejecución. Básicamente lo que este patrón propone es utilizar instancias que implementen la interface requerida pero con un cuerpo vacío en lugar de regresar un valor *null*.

Este patrón de diseño nos puede ayudar a evitar algunos problemas en tiempo de ejecución como el clásico *NullPointerException* o *NullReferenceException* de los lenguajes como Java o C# y nos ahorra la tediosa tarea de preguntar si un objeto es nulo antes de usarlo, como en el siguiente código:

```
1.  public void action() {
2.      Employee object = EmployeeFinder.findEmployee();
3.      if (object != null) {
4.          System.out.println("name > " + object.getName());
5.      }
6.
7.      if (object.getAddress() != null) {
8.          System.out.println("address > " + object.getAddress());
9.      }
10. }
```

Como podemos apreciar en el ejemplo anterior, hay que preguntar si el objeto o sus atributos no son nulos antes de usarlos, ya que de no hacerlo el programa podría terminar inesperadamente por una excepción no controlada como las ya mencionadas.

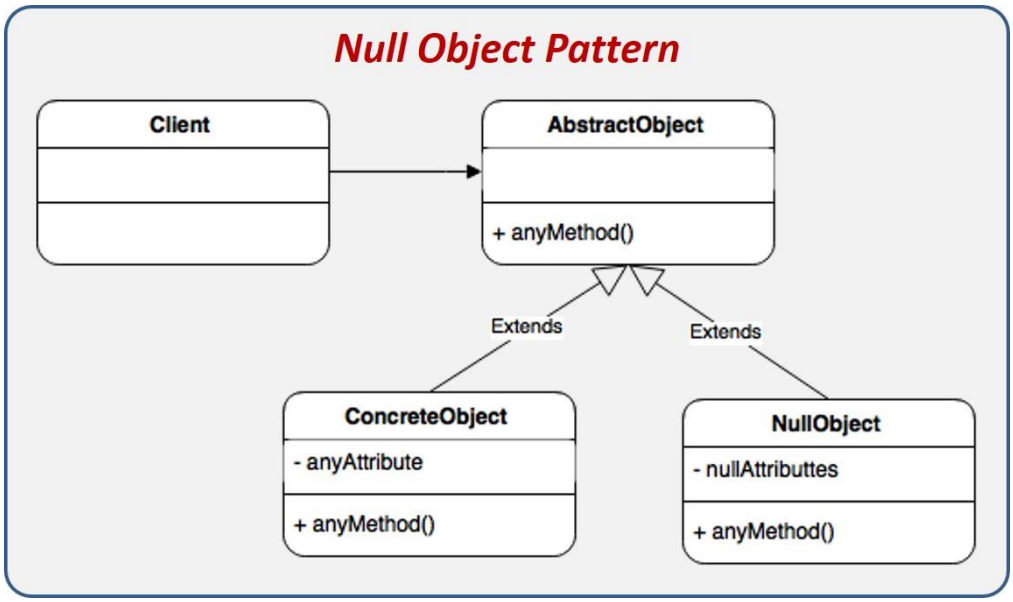


Ilustración 202: Estructura del patrón de diseño Null Object.

Los componentes que conforman el patrón son los siguientes:

- **Client:** Componente que utiliza las instancias que puede ser afectado por un objeto nulo.
- **AbstractObject:** Interface común entre el objeto real y su objeto de representación nula.
- **ConcreteObject:** Clase que representa al objeto real, el cual tiene una implementación concreta; será creado siempre y cuando el objeto buscado exista.
- **NullObject:** Objeto con implementación de sus métodos vacíos, devuelto en lugar de regresar una referencia nula. Será creado únicamente cuando el objeto buscado no exista.

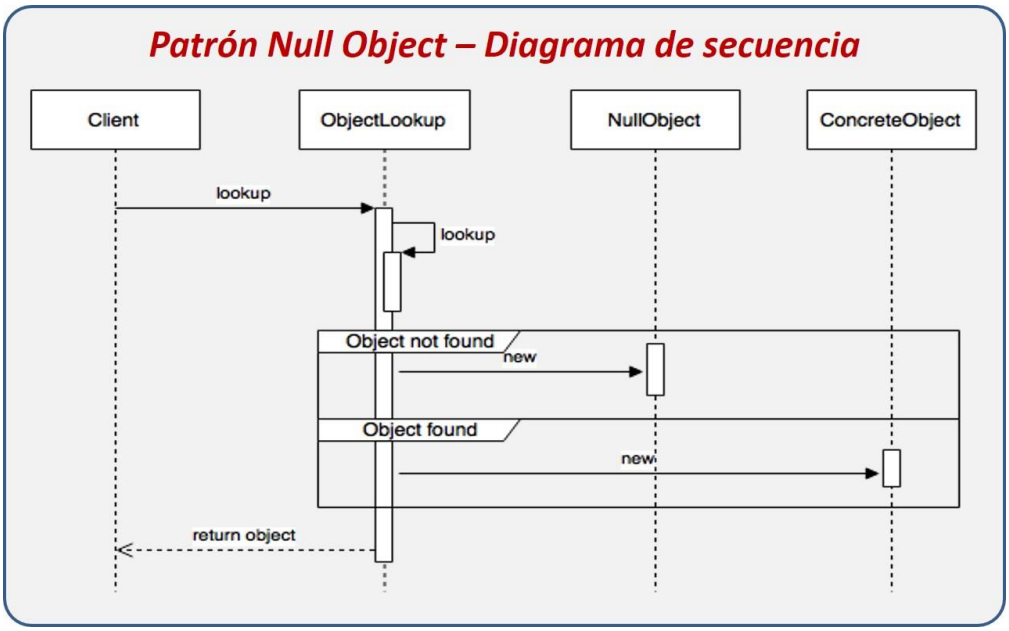


Ilustración 203: Diagrama de secuencia del patrón null object.

El diagrama de interpreta de la siguiente manera:

1. El cliente intenta buscar un objeto determinado.
2. El *ObjectLookup* busca si el objeto solicitado existe.
    a. Si el objeto solicitado no existe entonces regresa una instancia de *NullObject*.
3. Por otra parte, si el objeto es localizado entonces se regresa una instancia del *ConcreteObject*.
4. Se regresa al cliente cualquiera de las dos instancias anteriores, sin embargo, éste nunca obtendrá una referencia nula en caso de no encontrarse.

El escenario:

Para comprender mejor cómo funciona este patrón de diseño, desarrollaremos un servicio de búsqueda de empleados, el cual por medio del ID sea capaz de localizar al empleado, sin embargo, hay que tener cuidado cuando este servicio regrese un valor nulo indicando que no localizó al empleado. En este caso se nos ha solicitado que hagamos algo que nos ayude a evitar este problema.

Un problema muy habitual al obtener valores nulos no esperados, es la finalización del programa o el hilo que se esté ejecutando al momento de que una excepción es lanzada tras intentar utilizar una referencia nula.

La siguiente imagen muestra cómo es que esto ocurre:

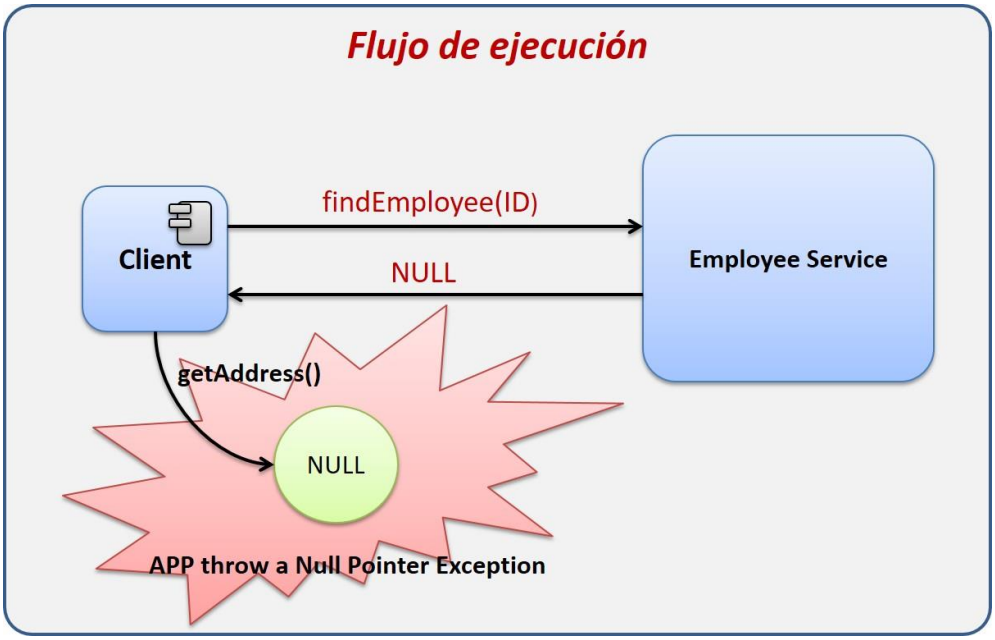


Ilustración 204: Ejecución con referencias nulas.

El cliente busca a un empleado mediante el servicio de empleados, no lo encuentra y regresa un valor nulo, el cliente intenta acceder a las propiedades del Empleado, sin embargo, este es nulo y se arroja la excepción de *NullPointerException*.

La solución:

Para evitar este tipo de situaciones y que el programa termine de forma inesperada, podemos optar por implementar el patrón de diseño *Null Object*, mediante la implementación de este patrón podríamos regresar un Objeto Empleado con sus métodos sin cuerpo o podríamos regresar un valor fijo que nos ayude a identificar qué es un valor por *default*.

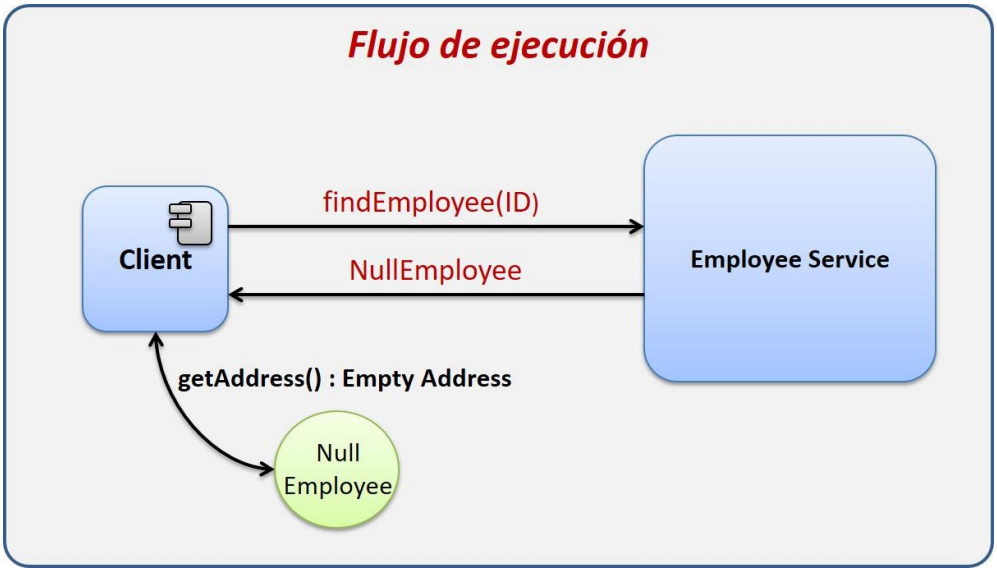


Ilustración 205: Implementación del patrón Null Object.

El cliente intenta obtener una instancia de un empleado mediante su ID a través del servicio de Empleados. El empleado con el ID solicitado no es encontrado y en su lugar una instancia que cumple con el patrón *Null Object*, es retornado.
Cuando el cliente recibe esta instancia intentará acceder a sus propiedades o métodos. El cliente, al recibir una instancia vacía, no lanzará una excepción al encontrarse con que el cliente no existe; en su lugar una representación vacía se obtendrá al leer cualquier método o propiedad del objeto Empleado.

Una buena práctica para implementar los *NullObject*, es crear la instancia nula como una constante, de esta manera siempre utilizaremos la misma constante para todos los objetos.

La implementación:

Una vez que conocemos cómo funciona el patrón y hemos expuesto el escenario, pasaremos a la implementación.

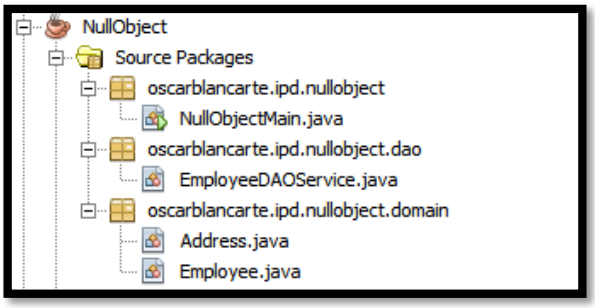


Ilustración 206: Estructura del patrón Null Object.

Los paquetes del proyecto son los siguientes:

- **Oscarblancarte.ipd.nullobject:** Paquete principal del proyecto contiene la clase ejecutable *NullObjectMain*.
- **Oscarblancarte.ipd.nullobject.dao:** Paquete con las clases para buscar a los empleados.
- **Oscarblancarte.ipd.nullobject.domain:** Contiene las clases de dominio de la aplicación, empleado y domicilio.

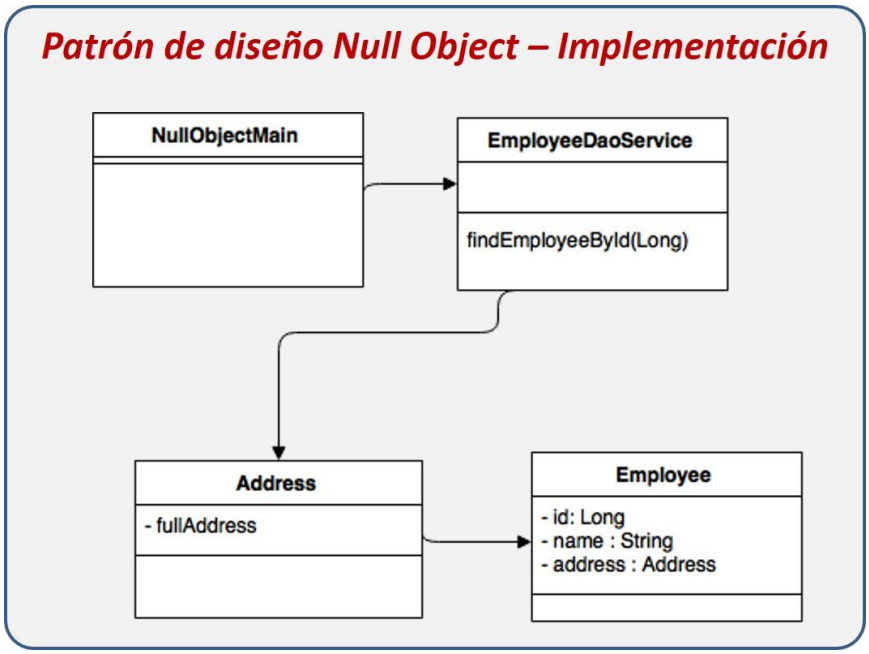


Ilustración 207: Diagrama de clases del proyecto NullObject.

En el diagrama podemos ver las clases del proyecto y la forma en que están relacionadas.

Clase Employee:

La clase Empleado es la clase más importante del proyecto, pues en ésta centraremos nuestra atención para entender el funcionamiento del patrón *Null Object*.

La clase cuenta con una variable de si misma llamada *NULL_EMPLOYEE* (línea 5) y que usaremos para representar un empleado nulo, podemos apreciar que regresan valores por *default* en todos sus métodos. El método *getAddress* puede ser interesante porque regresa un *Null Object* de la dirección con la finalidad de prevenir excepciones cuando el cliente intente acceder a las propiedades del domicilio del empleado.

La clase cuenta, además, con un constructor para crear nuevas instancias (línea 24) y recibe el ID, Nombre y Domicilio.

```
1.  package oscarblancarte.ipd.nullobject.domain;
2.
3.  public class Employee {
4.
5.      public static final Employee NULL_EMPOYEE = new Employee() {
6.
7.          public Long getId() {
8.              return 0L;
9.          }
10.
11.         public String getName() {
12.             return "UNKNOW EMPLOYEE";
13.         }
14.
15.         public Address getAddress() {
16.             return Address.NULL_ADDRESS;
17.         }
18.     };
19.
20.     private Long id;
21.     private String name;
22.     private Address address;
23.
24.     public Employee(Long id, String name, Address address) {
25.         this.id = id;
26.         this.name = name;
27.         this.address = address;
28.     }
29.
30.     public Employee() {
31.     }
32.
33.     public Long getId() {
34.         return id;
35.     }
36.
37.     public void setId(Long id) {
38.         this.id = id;
39.     }
40.
41.     public String getName() {
42.         return name;
43.     }
44.
45.     public void setName(String name) {
46.         this.name = name;
47.     }
48.
49.     public Address getAddress() {
50.         return this.address;
51.     }
52.
53.     public void setAddress(Address address) {
```

```
54.         this.address = address;
55.     }
56. }
```

Clase Address:

La clase Domicilio (*Address*) la utilizaremos para representar un domicilio del Empleado. Tiene la propiedad *fullAddres* que representa el domicilio completo del empleado (línea 11).
En la clase se encuentra también una instancia de si misma llamada *NULL_ADDRESS*, que será utilizada para representar un domicilio nulo.

```
1.  package oscarblancarte.ipd.nullobject.domain;
2.
3.  public class Address {
4.
5.      public static final Address NULL_ADDRESS = new Address() {
6.          public String getFullAddress() {
7.              return "NOT ADDRESS";
8.          }
9.      };
10.
11.     public String fullAddress;
12.
13.     public Address(String fullAddress) {
14.         this.fullAddress = fullAddress;
15.     }
16.
17.     public Address() {
18.         this.fullAddress = fullAddress;
19.     }
20.
21.     public String getFullAddress() {
22.         return fullAddress;
23.     }
24.
25.     public void setFullAddress(String fullAddress) {
26.         this.fullAddress = fullAddress;
27.     }
28. }
```

Clase EmployeeDaoService:

Clase que utilizaremos para simular un servicio de búsqueda de empleados que implementa:

- **findEmployeeById:** Busca a un empleado mediante el ID, en caso de no encontrarlo éste regresa un *Null Object* del Empleado (línea 23), los empleados son precargados desde el constructor de la clase (línea 11).

Observemos que la variable *Employee*.*NULL_EMPOYEE* es una constante, por lo que siempre regresaremos la misma instancia evitando tener varias instancias vacías y ahorrar un poco de memoria.

```
1.  package oscarblancarte.ipd.nullobject.dao;
2.
3.  import java.util.*;
4.  import oscarblancarte.ipd.nullobject.domain.Address;
5.  import oscarblancarte.ipd.nullobject.domain.Employee;
6.
7.  public class EmployeeDAOService {
8.
9.      private List<Employee> EMPLOYEE_LIST = new ArrayList<>();
10.
11.     public EmployeeDAOService(){
12.         Employee emp1 = new Employee(1L, "Oscar Blancarte",
13.                 new Address("Reforma 150 int 20, México D.F."));
14.         EMPLOYEE_LIST.add(emp1);
15.     }
16.
17.     public Employee findEmployeeById(Long id){
18.         for(Employee emp : EMPLOYEE_LIST){
19.             if(emp.getId().longValue() == id.longValue()){
20.                 return emp;
21.             }
22.         }
23.         return Employee.NULL_EMPOYEE;
24.     }
25. }
```

Para este ejemplo el único ID válido de un empleado será el 1, por lo que cualquier otro regresará un Empleado Nulo.

Clase NullObjectMain:

Esta es la clase ejecutable del proyecto y desde aquí comprobaremos la utilidad de este patrón.

Veamos que en la línea 13 se crea una instancia de la clase *EmployeeDAOService* que utilizamos en la línea 14 para buscar al Empleado con el ID=1. Seguido, imprimimos la dirección completa de éste (línea 15).
Si regresamos a la clase *EmployeeDAOService***,** el empleado con ID ya se encuentra registrado, por lo que estamos seguros de que la búsqueda tendrá como resultado un empleado real.

En la línea 17 intentamos buscar el siguiente Empleado con ID=2, el cual sabemos que no está registrado, e imprimimos su dirección, aquí es donde hay que poner atención ya que el empleado al no existir y en caso de regresar un valor nulo, el programa terminaría con un *NullPointerException*, sin embargo, esto no ocurrirá debido a que hemos implementado el patrón *Null Object*.

```
1.  package oscarblancarte.ipd.nullobject;
2.
3.  import java.io.IOException;
4.  import java.util.logging.Level;
5.  import java.util.logging.Logger;
6.  import oscarblancarte.ipd.nullobject.dao.EmployeeDAOService;
7.  import oscarblancarte.ipd.nullobject.domain.Employee;
8.
9.  public class NullObjectMain {
10.
11.     public static void main(String[] args) {
12.         EmployeeDAOService service = new EmployeeDAOService();
13.         Employee emp1 = service.findEmployeeById(1L);
14.         System.out.println(emp1.getAddress().getFullAddress());
15.
16.         Employee emp2 = service.findEmployeeById(2L);
17.         System.out.println(emp2.getAddress().getFullAddress());
18.     }
19. }
```

Si al ejecutar el programa no tuviéramos implementado el patrón *Null Object*, el programa terminaría con un *NullPointerException* en la línea 17.

La ejecución:

Para comprobar la funcionalidad del patrón ejecutaremos la clase *NullObjectMain* para ver el siguiente resultado:

```
run:
Reforma 150 int 20, Mexico D.F.
NOT ADDRESS
BUILD SUCCESSFUL (total time: 2 seconds)
```

Ilustración 208: Resultado de la ejecución.

En la imagen podemos apreciar que el primer Empleado tiene la dirección "*Reforma 150 int 20, México D.F.*", sin embargo, el segundo Empleado buscado no fue encontrado y en su lugar se regresa un *Null Object* de Empleado, éste al solicitarle su dirección regresó un *Null Object* de la dirección, de esta manera el programa ha podido continuar la ejecución y regresa únicamente la leyenda *"NOT ADDRESS"* como dirección.

Siguientes pasos:

Hemos analizado las ventajas que nos trae implementar el patrón *Null Object*, sin embargo, cabe mencionar que no siempre será necesario o precisamente lo mejor, ya que existen situaciones en las que regresar un valor nulo puede ser excelente. Este patrón se utiliza mucho en conjunto con el patrón *Estrategy* utilizándolo cuando no queremos manejar ninguna estrategia pero queremos evitar errores o tener que validar constantemente si la estrategia es nula.

Para fortalecer los conocimientos adquiridos realizaremos los siguientes ejercicios:

- Creemos una nueva clase llamada *Department* que representará el departamento al que pertenece el empleado dentro de la empresa. Adicionalmente, modificaremos la clase *Employee* para que agregue una referencia a la clase *Department*. Finalmente, nos aseguraremos de que el empleado regrese una representación *Null Object* del departamento en caso de no existir el empleado o que el empleado no tenga departamento.

- Agregaremos una nueva clase llamada *Customer* que representará a nuestros clientes. Crearemos una clase que busque a nuestros clientes por medio del ID, en caso de no encontrar al cliente regresará una representación nula.

Patrón State

El patrón de diseño State se caracteriza por modificar su comportamiento dependiendo del estado en el que se encuentra la aplicación. Para lograr esto es necesario crear una serie de clases que representarán los distintos estados por los que puede pasar la aplicación; es decir, se requiere de una clase por cada estado por el que la aplicación pueda pasar.

Este patrón de diseño se puede confundir con el patrón de diseño Memento pero existen diferencias claves que lo hacen diferente.
En primer lugar, el patrón Memento guarda los estados de un Objeto en particular con la finalidad de poder restaurar el estado en un tiempo determinado, y en segundo lugar, Memento no altera su comportamiento según el estado del objeto.
Por otro lado, el patrón State centra su comportamiento en tiempo de ejecución dependiendo el estado en el que esté, de manera que el comportamiento de algunas de sus operaciones podría funcionar de formas muy distintas según el estado en el que se encuentre la aplicación al momento de realizar la acción.

Un ejemplo simple de esto son las máquinas de refrescos, cuando presionas el botón para que te dé un refresco sin haber puesto una moneda, la maquina te pedirá que insertes una moneda.
Ahora bien, si repetimos el proceso pero esta vez depositando la moneda, la máquina nos dará el refresco.
Otro escenario es si depositamos la moneda y presionamos el botón, pero el refresco que deseamos se agotó, la máquina nos arrojará el mensaje y nos pedirá que elijamos otro producto o la devolución del dinero.
Pongamos atención en que siempre presionamos el mismo botón, pero su funcionamiento dependió totalmente del estado en el que estaba la máquina al momento de presionar el botón.

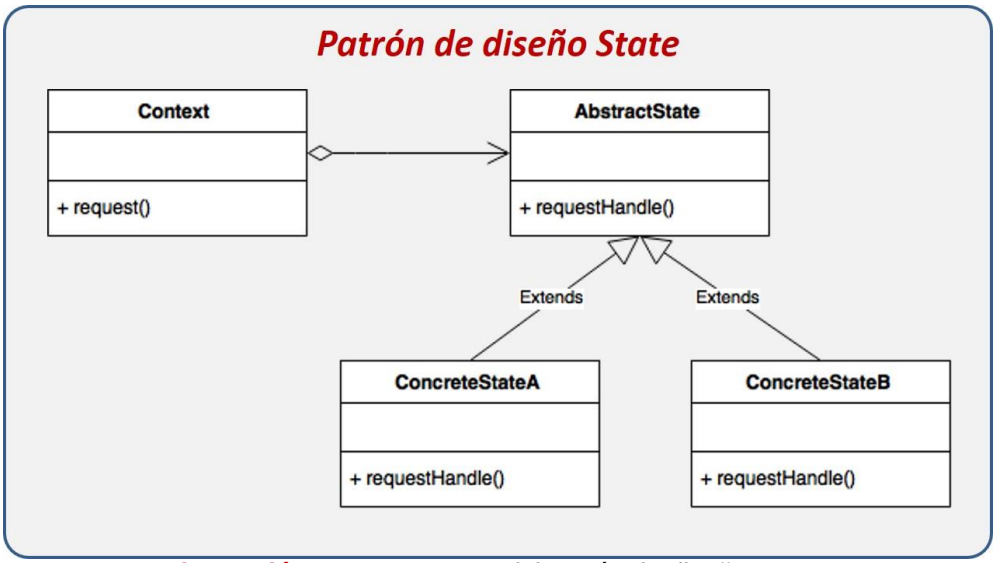


Ilustración 209: Estructura del patrón de diseño State.

A continuación se explican los componentes que conforman el patrón de diseño *State*:

- **Context:** Representa el componente que puede cambiar de estado, el cual tiene entre sus propiedades el estado actual. En el ejemplo de la máquina de refrescos, éste sería la máquina como tal.
- **AbstractState:** Clase base para la generación de los distintos estados. Se recomienda que sea una clase abstracta en lugar de una interface debido a que podemos definir comportamientos por *default* y así afectar el funcionamiento de todos los estados.
- **ConcreteState:** Cada uno de estos componentes representa un posible estado por el cual la aplicación puede pasar, por lo que tendremos un *ConcreteState* por cada estado posible. Esta clase debe de heredar de *AbstractState*.

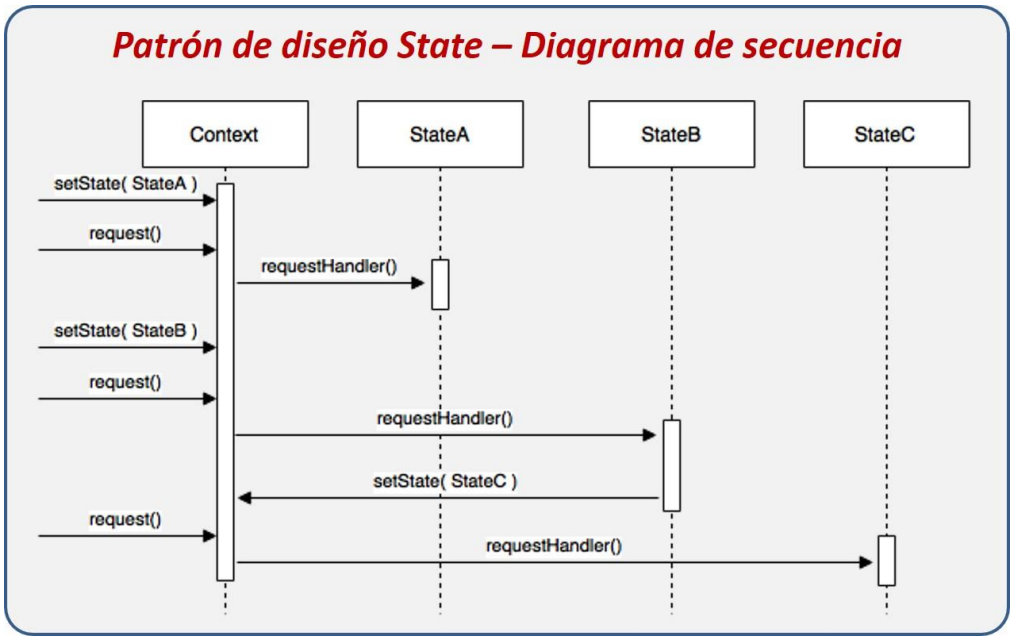


Ilustración 210: Diagrama de secuencia del patrón de diseño State.

El diagrama anterior se interpreta de la siguiente manera:

1. Se establece un estado por *default* al *Context,* el cual es StateA.
2. Se ejecuta la operación *request* sobre el *Context,* la cual delega la ejecución al estado actual (StateA).
3. El Context cambia del estado A al estado B.
4. Se ejecuta nuevamente la operación *request* sobre el *Context* que delega la ejecución al estado actual (StateB).
5. La ejecución del StateB da como resultado un cambio de estado al StateC.
6. Se ejecuta nuevamente la operación *request* sobre el *Context* que delega la ejecución al estado actual (StateC).

Como podemos observar, los consumidores del *Context* siempre ejecutan la misma operación pero el comportamiento de la ejecución de esta operación está asociado al estado que tenga el *Context* al momento de la ejecución.

El estado puede ser establecido directamente desde fuera del *Context* pero, también la ejecución de un método de algún estado puede ocasionar un cambio de estado en el contexto. Retomando el ejemplo de la máquina de refrescos, cuando alguien presiona el botón para comprar un refresco éste le entregará el refresco pero como parte de la operación de servir el producto, también cambiará el estado de la máquina a su estado inicial.

El escenario:

Para comprender mejor cómo funciona este patrón simularemos la creación de un servidor, este servidor tendrá como objetivo atender las solicitudes que le mandemos. El servidor deberá pasar por distintos estados durante todo su ciclo de vida, de esta manera los mensajes enviados al servidor serán atendidos de forma distinta según el estado en el que se encuentre el servidor al momento de enviar la petición.

Los estados por los que deberá pasar el servidor son:

- **Detenido:** El servidor no aceptará ninguna petición.
- **Iniciando:** El servidor aceptará las peticiones pero no las atenderá hasta que el servidor inicie.
- **Iniciado:** El servidor acepta las peticiones y empieza a procesar las solicitudes.
- **Saturado:** Cuando el servidor reciba mas de 5 solicitudes, se saturará y no aceptará más solicitudes hasta que el número de peticiones baje a 4.

Algunas reglas a tomar en cuenta son las siguientes:

1. El servidor no podrá ser apagado mientras está en estado **Iniciando**.
2. El servidor pasará a estado **Iniciado** a 10 segundos de entrar en estado **Iniciando**, logrando con esto simular que está cargando.
3. Cuando el servidor reciba 5 o más solicitudes, mientras esté en estado **Iniciando,** pasará directamente a estado **Saturado** sin pasar por **Iniciado**. Cuando el servidor baje el número de peticiones a 4 nuevamente pasará a estado **Iniciado** y aceptará peticiones.

Un punto importante es que nosotros sólo podremos establecer los estados **Detenido** e **Iniciando**, el resto de estados serán controlados internamente en consecuencia de las reglas definidas.

Ilustración 211: Protopito de la pantalla.

La pantalla solamente tendrá dos botones, en la parte superior tendremos el botón de iniciar que establecerá el estado del servidor en **Iniciando**, si lo volvemos a presionar establecerá el estado **Detenido**. El segundo botón es *sendMessage* el cual se encuentra en la parte baja de la pantalla y lo que hace es enviar una petición al servidor.

Para estar enterado de lo que está pasando dentro del servidor, éste arrojará mensajes en el área de texto ubicado al centro de la pantalla. Un mensaje aparecerá cada vez que el estado cambie o una nueva petición se envíe al servidor.

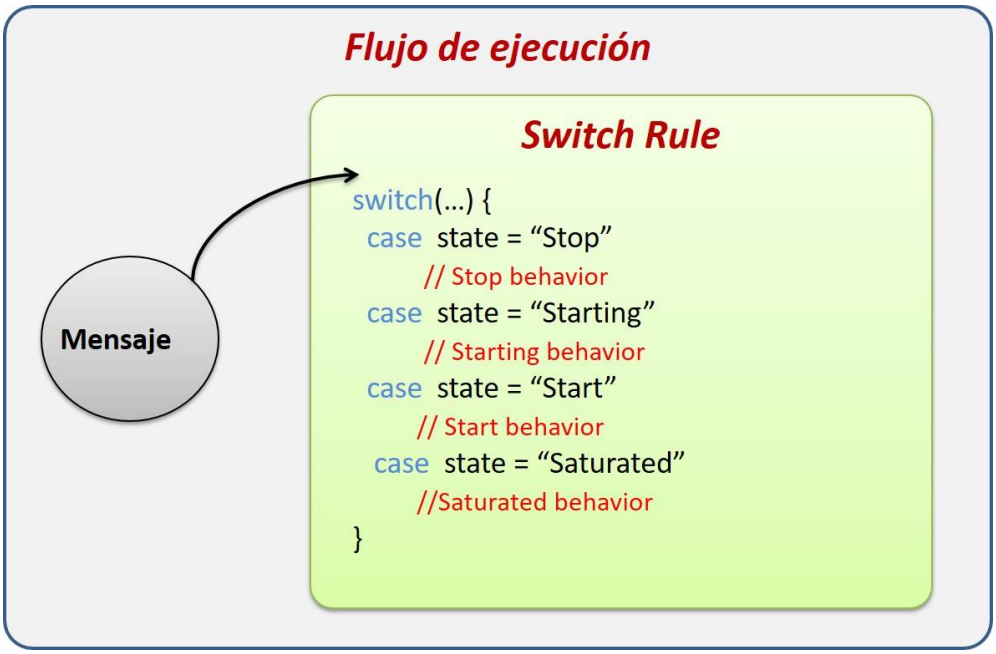


Ilustración 212: Ejecución sin utilizar el patrón de diseño State.

Uno de los errores más comunes en este tipo de implementaciones es utilizar *switch* o una serie de *else if*… *else if*, para determinar el estado de la aplicación y con base en esto realizar la acción adecuada, sin embargo, este es un diseño poco mantenible y fomenta que implementemos toda la lógica en esta misma clase.

La solución:

Para solucionar este problema de una forma más sencilla y organizada hemos decidido implementar la solución utilizando el patrón de diseño *State*, el cual nos permitirá definir los estados por los que podrá pasar el servidor y al mismo tiempo definir el comportamiento del mismo a través de los distintos estados de la aplicación.

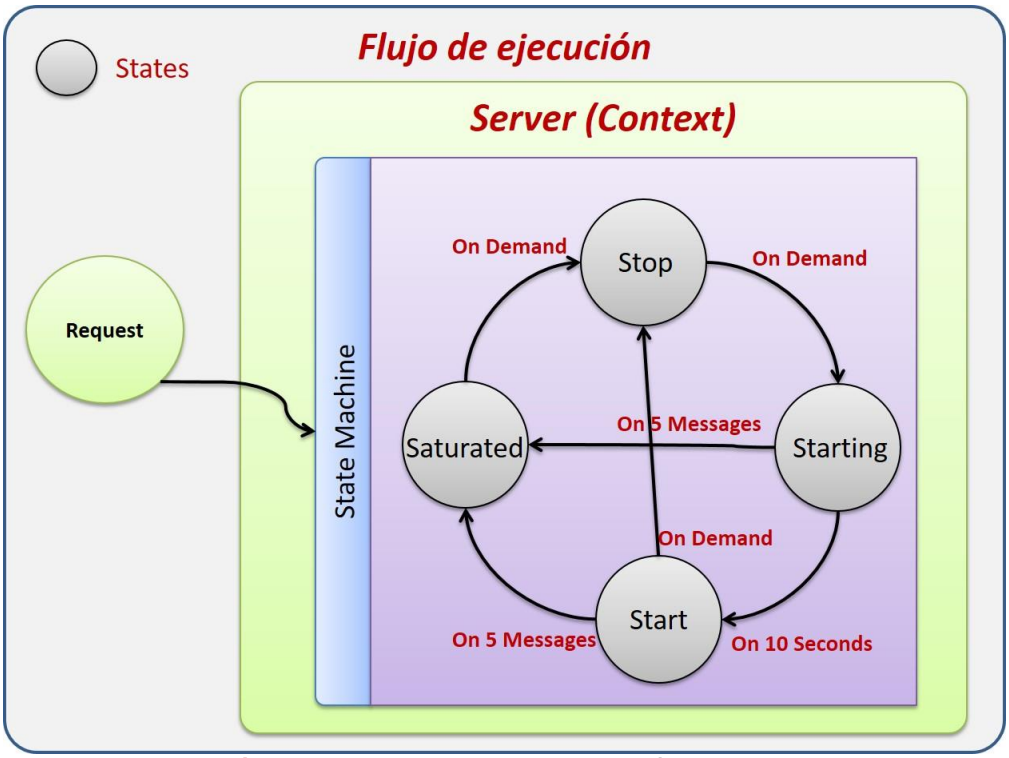


Ilustración 213: Flujo de trabajo con el patrón de diseño State.

En la imagen podemos apreciar la forma en que un mensaje es procesado, cuando una solicitud llega al servidor es delegada al estado actual del servidor, la cual será establecida por acciones del usuario o por las circunstancias dadas en tiempo de ejecución.

Lo importante de este modelo es que está perfectamente clara la forma en que trabaja nuestra aplicación, incluso podemos saber los estados por lo que podrá pasar y el orden de éstos, que a su vez, modificarán el comportamiento de la aplicación.

Por ejemplo, cuando un mensaje llegue al servidor y el estado de éste sea **Apagado**, el servidor rechazará el mensaje inmediatamente; en cambio, si el servidor se encuentra en estado **Iniciado**, entonces el mensaje será recibido y procesado.

La implementación:

Una vez que hemos comprendido cómo funciona el patrón de diseño *State* y conocemos el escenario a desarrollar, pasaremos a la implementación del mismo.

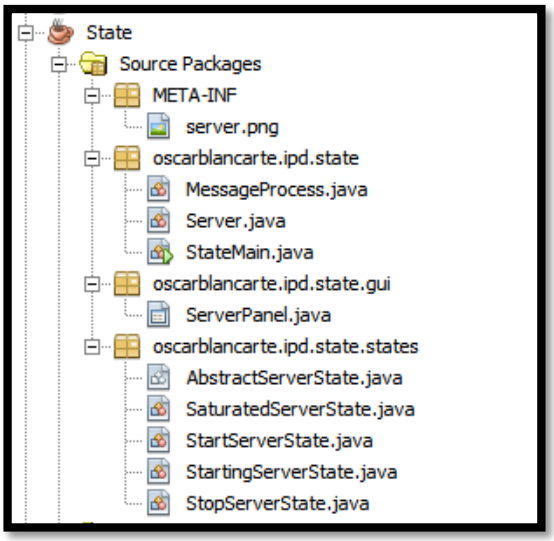


Ilustración 214: Estructura del proyecto State.

Los paquetes del proyecto se explican a continuación:

- **oscarblancarte.ipd.state:** Paquete principal del proyecto.
- **oscarblancarte.ipd.state.gui:** Paquete con las clases de interface gráfica.
- **oscarblancarte.ipd.state.states**: Paquete que contiene los estados posibles de la aplicación.

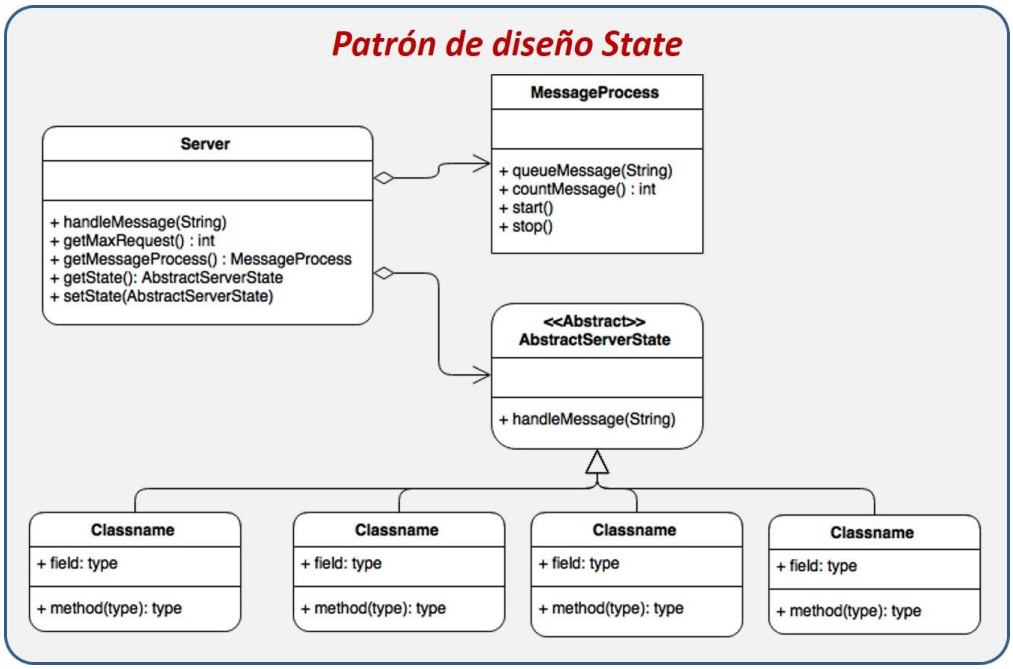


Ilustración 215: Diagrama de clases del proyecto State.

La imagen anterior muestra las clases del proyecto y cómo es que éstas se relacionan entre sí.

Clase Server:

El servidor representa al *Context* del patrón de diseño ya que es el que puede cambiar de estado.

El servidor contiene las siguientes propiedades:

- **maxRequest:** Constante que indica el número máximo de solicitudes que puede recibir antes de entrar en estado **Saturado**.
- **messageProcess:** Clase encargada del procesamiento de los mensajes, más adelante la analizaremos.
- **state:** Esta variable contiene el estado actual del servidor y será en esta propiedad en la que centraremos nuestra atención pues es el estado el que define el comportamiento del servidor.

Las operaciones más importantes del servidor son:

- **handleMessage:** Esta operación es la que utilizaremos para enviarle mensajes al servidor. Si vemos con cuidado nos damos cuenta de que en realidad el servidor no procesa el mensaje, en su lugar le envía el mensaje al estado actual por medio de la operación *handleMessage*. Más adelante analizaremos cómo los distintos estados procesan el mensaje.
- **setState:** Operación que nos permitirá establecerle un estado al servidor. El estado puede ser determinado por algún agente externo al servidor o internamente como resultado de la ejecución del estado actual.
- En este método establecemos una restricción que nos impedirá apagar el servidor cuando esté en estado **Iniciando**.
- **getState:** Regresa el estado actual del servidor.

```
1.  package oscarblancarte.ipd.state;
2.
3.  import oscarblancarte.ipd.state.states.AbstractServerState;
4.  import oscarblancarte.ipd.state.states.StartingServerState;
5.  import oscarblancarte.ipd.state.states.StopServerState;
6.
7.  public class Server {
8.
9.      private static final int maxRequest = 5;
10.     private final MessageProcess messageProcess
11.             = new MessageProcess(this);
12.     private AbstractServerState state;
13.
14.     public Server() {
15.         setState(new StopServerState(this));
16.     }
17.
18.     public void handleMessage(String message) {
19.         state.handleMessage(this, message);
20.     }
21.
22.     public int getMaxRequest() {
23.         return maxRequest;
24.     }
25.
26.     public MessageProcess getMessageProcess() {
27.         return messageProcess;
28.     }
29.
30.     public AbstractServerState getState() {
```

```
31.         return state;
32.     }
33.
34.     public void setState(AbstractServerState state) {
35.         if (this.state instanceof StartingServerState
36.                 && state instanceof StopServerState) {
37.             System.out.println("Server is starting, "
38.                     + "cannot change state");
39.             return;
40.         }
41.         this.state = state;
42.         System.out.println("Server change state > " +
43.                 this.state.getClass().getSimpleName());
44.     }
45. }
```

Clase MessageProcess:

La clase *MessageProcess* es la encargada de procesar los mensajes, esta clase atenderá los mensajes uno a uno con un intervalo de 10 segundos.

Las operaciones de esta clase son:

- **queueMessage:** Operación utilizada para almacenar una solicitud para ser procesada. Antes de aceptar un mensaje, valida que el número de mensajes almacenados no superen el máximo de permitidos por el *Server*, de ser así, el mensaje es rechazado.
- **start:** Inicia con el procesamiento de mensajes. Se crea un hilo que procesa un mensaje cada 10 segundos.
- **stop:** Detiene el procesamiento de mensajes.

```
1.  package oscarblancarte.ipd.state;
2.
3.  import java.util.LinkedList;
4.  import java.util.Queue;
5.  import java.util.logging.Level;
6.  import java.util.logging.Logger;
7.
8.  public class MessageProcess {
9.
10.     private Thread sendThread;
11.     private final Server server;
12.     private final Queue<String> messageQueue
13.             = new LinkedList<>();
```

```
14.
15.     public MessageProcess(Server server) {
16.         this.server = server;
17.     }
18.
19.     public boolean queueMessage(String message) {
20.         if (messageQueue.size() >= server.getMaxRequest()) {
21.             System.out.println("Requests reached maximum ("
22.                     + messageQueue.size() + ")");
23.             return false;
24.         }
25.         messageQueue.add(message);
26.         System.out.println("Queue message ("
27.                 + messageQueue.size() + ")");
28.         return true;
29.     }
30.
31.     public int countMessage() {
32.         return messageQueue.size();
33.     }
34.
35.     public void start() {
36.         this.sendThread = new Thread(new Runnable() {
37.             public void run() {
38.                 try {
39.                     while (true) {
40.                         Thread.sleep(1000 * 10);
41.                         if (!messageQueue.isEmpty()) {
42.                             String message = messageQueue.poll();
43.                             System.out.println("Message process > "
44.                                     + message + "(" + messageQueue.size()
45.                                     + ")");
46.                         }
47.                     }
48.                 } catch (InterruptedException ex) {
49.                     System.out.println(ex.getMessage());
50.                 }
51.             }
52.         });
53.
54.         this.sendThread.start();
55.     }
56.
57.     public void stop() {
58.         if (this.sendThread != null) {
59.             this.sendThread.stop();
60.             this.sendThread = null;
61.         }
62.     }
63. }
```

Clase AbstractServerState:

Clase base que sirve para crear todos los estados. Esta clase se recomienda que sea abstracta para poder establecer comportamiento por *default* en caso de ser necesario.
La clase únicamente define el método *handleMessage* que será utilizado por el servidor para enviar los mensajes a procesar.

```
1.  package oscarblancarte.ipd.state.states;
2.
3.  import oscarblancarte.ipd.state.Server;
4.
5.  public abstract class AbstractServerState {
6.
7.      public AbstractServerState() {
8.      }
9.
10.     public abstract void handleMessage(
11.             Server server, String message);
12.
13. }
```

Clase StopServerState:

Esta clase representa el estado apagado del servidor.
La operación *handleMessage* está implementada para ignorar el mensaje, cuando un mensaje llega, éste solamente imprime un mensaje indicando que el servidor se encuentra apagado.

```
1.  package oscarblancarte.ipd.state.states;
2.
3.  import oscarblancarte.ipd.state.Server;
4.
5.  public class StopServerState extends AbstractServerState {
6.
7.      public StopServerState(Server server){
8.          server.getMessageProcess().stop();
9.      }
10.
11.     @Override
12.     public void handleMessage(Server server, String message) {
13.         System.out.println("The server is stopped");
```

```
14.     }
15. }
```

Clase StartingServerState:

Esta clase representa el estado **Iniciando**. Implementa el método *handleMessage* para aceptar los mensajes entrantes pero no serán procesados hasta que el servidor entre en estado **Iniciado** o **Saturado**.
El constructor de esta la clase crea un hilo que espera 10 segundos antes de pasar el servidor a estado **Iniciado** o **Saturado** según el número de mensajes en espera.

```
1.  package oscarblancarte.ipd.state.states;
2.
3.  import oscarblancarte.ipd.state.Server;
4.
5.  public class StartingServerState extends AbstractServerState {
6.
7.      public StartingServerState(final Server server) {
8.          new Thread(new Runnable() {
9.
10.             @Override
11.             public void run() {
12.                 try {
13.                     System.out.println("Server Starting");
14.                     Thread.sleep(1000 * 10);
15.                     if (server.getMessageProcess().countMessage()
16.                             >= server.getMaxRequest()) {
17.                         server.setState(
18.                                 new SaturatedServerState(server));
19.                     } else {
20.                         server.setState(
21.                                 new StartServerState(server));
22.                     }
23.                     System.out.println("Server Start");
24.                 } catch (Exception e) {
25.                 }
26.             }
27.         }
28.         ).start();
29.     }
30.
31.     @Override
32.     public void handleMessage(Server server, String message) {
33.         server.getMessageProcess().queueMessage(message);
34.     }
```

```
35.
36. }
```

Clase StartServerState:

Clase que representa el estado iniciado del servidor, cuando el servidor cambia a este estado indica que ha iniciado correctamente e inicia con el procesamiento de mensajes. Este estado es establecido automáticamente tras un tiempo en estado ***Starting***.
Los métodos definidos son:

- ***handleMessage***: Recibe los mensajes entrantes para ser procesados. Los mensajes entrantes son enviados al *MessageProcess* para ser atendidos. Cuando el número de mensajes llega a 5, se cambia el estado del *server* a **Saturado** de forma automática.

En el constructor de la clase se inicial el *MessageProcess* mediante la operación *start* para empezar con el procesamiento de mensajes.

```
1.  package oscarblancarte.ipd.state.states;
2.
3.  import oscarblancarte.ipd.state.Server;
4.
5.  public class StartServerState extends AbstractServerState {
6.
7.      public StartServerState(Server server){
8.          server.getMessageProcess().start();
9.      }
10.
11.     @Override
12.     public void handleMessage(Server server, String message) {
13.         boolean isSaturated =
14.                 server.getMessageProcess().queueMessage(message);
15.         if(!isSaturated){
16.             server.setState(new SaturatedServerState(server));
17.         }
18.     }
19. }
```

En el constructor de la clase se inicia el procesamiento de mensajes mediante el método *start* de la clase *MessageProcess* (línea 8).

Clase SaturatedServerState:

Clase que representa el estado **Saturado** y cuando el *server* entra en este estado ya no recibe más mensajes hasta que el número baja a 4. El cambio de este estado se hace de forma automática como parte del ciclo de vida de la aplicación.

El método *handleMessage* es implementado para rechazar los mensajes entrantes debido a que cuando el servidor se encuentra en este estado es que ha llegado al máximo de peticiones soportadas.

El constructor se implementa para monitorear el número de mensajes que hay por procesar. Cuando éste detecta que el número de mensajes baja a 4, automáticamente cambia el estado del servidor a **Iniciado** para continuar recibiendo nuevos mensajes.

```
1.  package oscarblancarte.ipd.state.states;
2.
3.  import oscarblancarte.ipd.state.Server;
4.
5.  public class SaturatedServerState extends AbstractServerState {
6.
7.      private Thread monitoringThread;
8.
9.      public SaturatedServerState(final Server server) {
10.         server.getMessageProcess().start();
11.         monitoringThread = new Thread(new Runnable() {
12.
13.             @Override
14.             public void run() {
15.                 try {
16.                     while (true) {
17.                         if (server.getMessageProcess()
18.                                 .countMessage() <
19.                                 server.getMaxRequest()) {
20.                             server.setState(
21.                                     new StartServerState(server));
22.                             break;
23.                         }
24.                     }
25.                 } catch (Exception e) {
```

```
26.                 System.out.println(e.getMessage());
27.             }
28.         }
29.     });
30.     monitoringThread.start();
31. }
32.
33. @Override
34. public void handleMessage(Server server, String message) {
35.     System.out.println("Cann´t process request, Server Saturated");
36. }
37. }
```

Clase StateMain:

Clase principal de la aplicación desde la cual iniciaremos el panel de control para interactuar con el servidor. Esta clase manda llamar a la clase *ServerPanel* desde la cual se crea la interface gráfica.
No entraremos en detalle de esta clase debido a que los temas de interface gráfica están fuera del alcance de este libro.

```
1. package oscarblancarte.ipd.state;
2.
3. import javax.swing.JFrame;
4. import oscarblancarte.ipd.state.gui.ServerPanel;
5.
6. public class StateMain {
7.
8.     public static void main(String[] args) {
9.         JFrame frame = new JFrame();
10.        frame.setTitle("State pattern");
11.        frame.setDefaultCloseOperation(
12.                JFrame.EXIT_ON_CLOSE);
13.        frame.add(new ServerPanel());
14.        frame.pack();
15.        frame.setVisible(true);
16.    }
17. }
```

Clase ServerPanel:

Clases de interface gráfica desde la cual se monitorea y controla el servidor.

No explicaremos los detalles de la construcción de la interface pues son temas que están fuera del alcance de este libro, pero sí explicaremos las secciones relevantes para entender su funcionamiento.

```
1. package oscarblancarte.ipd.state.gui;
2.
3. import java.io.OutputStream;
4. import java.io.PrintStream;
5. import oscarblancarte.ipd.state.Server;
6. import oscarblancarte.ipd.state.states.*;
7.
8. public class ServerPanel extends javax.swing.JPanel {
9.
10.     private Server server;
11.     private int messageCouter;
12.
13.     public ServerPanel() {
14.         initComponents();
15.
16.         PrintStream defaultPrinter = System.out;
17.         PrintStream interceptor = new TextAreaPrinter(defaultPrinter);
18.         System.setOut(interceptor);
19.
20.         server = new Server();
21.     }
22.
23.     private class TextAreaPrinter extends PrintStream {
24.
25.         public TextAreaPrinter(OutputStream out) {
26.             super(out, true);
27.         }
28.
29.         @Override
30.         public void print(String line) {
31.             println(line);
32.         }
33.
34.         @Override
35.         public void println(String line) {
36.             line = line + "\n";
37.             super.print(line);
38.             jTextArea1.append(line);
39.             jTextArea1.setCaretPosition(jTextArea1.getDocument().getLength());
40.         }
41.
42.     }
43.
44.     private void initComponents() {
45.     }
```

```
46.
47.     private void sendMessageEvent(java.awt.event.ActionEvent evt)
48.         server.handleMessage("Send Message + " + ++messageCouter);
49.     }
50.
51.     private void startAction(java.awt.event.ActionEvent evt)
52.         AbstractServerState state = server.getState();
53.         if (state instanceof StopServerState) {
54.             btnStart.setText("Stop");
55.             server.setState(new StartingServerState(server));
56.         } else {
57.             if (state instanceof StartingServerState) {
58.                 server.setState(new StopServerState(server));
59.             } else {
60.                 btnStart.setText("Start");
61.                 server.setState(new StopServerState(server));
62.             }
63.         }
64.
65.     }
66. }
```

En el constructor pasan varias cosas; la primera es la inicialización de la interface gráfica mediante la ejecución del método *initComponent* (línea 14), lo segundo es la configuración de la salida estándar para, en lugar de ser mostrada en la salida estándar, ésta sea enviada al área de texto de nuestra pantalla (líneas 16 a 18), finalmente, instanciamos la clase *Server* que representa al *server* en sí.

Los métodos definidos en la clase son:

- **initComponents:** Clase de inicialización de interface gráfica, no lo implementamos en esta sección pero está disponible toda su implementación desde el código fuente.
- **startAction:** Método ejecutado por el botón de inicio del servidor, se comporta de distinta manera según el estado del servidor. Si el servidor se encuentra apagado éste iniciará el servidor. Si el servidor se encuentra iniciando o ya inició, éste detiene al servidor.
- **sendMessageEvent:** Método que envía un mensaje al servidor, tras presionar el botón *sendMessage*.

La clase interna *TextAreaPrinter* es utilizada como una clase de utilidad para mandar los mensajes de la salida estándar al área de texto de la aplicación.

La ejecución:

Para probar nuestra aplicación ejecutaremos la clase *StateMain* y veremos la siguiente pantalla:

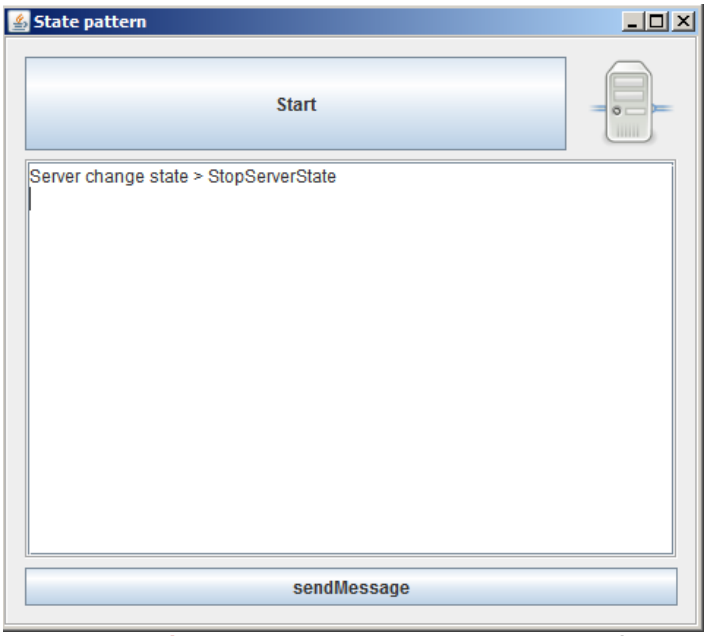


Ilustración 216: Pantalla pricipal de la aplicación.

Podemos apreciar que el primer mensaje que se muestra es del estado por *default* **StopServerState**.

La aplicación tiene dos botones, el de arriba inicia y apaga el servidor y el botón de abajo envía mensajes al servidor.

A continuación, iniciaremos realizando algunas pruebas para comprobar el funcionamiento del patrón *State*.

1. Antes de iniciar el servidor enviaremos un mensaje.
2. Iniciaremos el servidor
3. Esperaremos a que inicie el servidor (tardará 10 segundos).
4. Enviaremos 6 mensajes seguidos.
5. Esperaremos que termine de procesar todos los mensajes.

6. Apagamos el servidor.

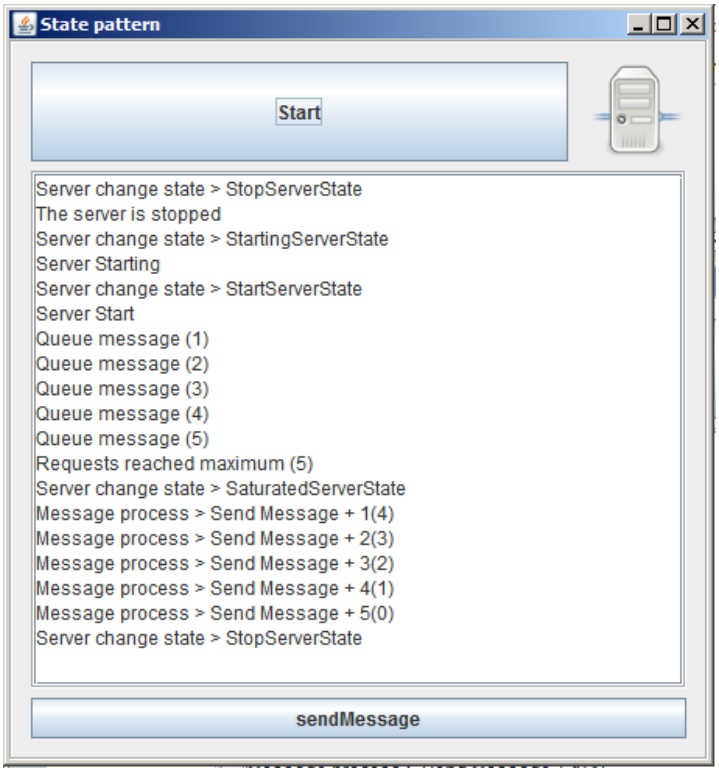


Ilustración 217: Resultado de la ejecución.

Analicemos el resultado por líneas:

1. El servidor comienza con el estado **Detenido**.
2. Se imprime el estado del servidor "***Stopped***".
3. Se presiona el botón para iniciar el servidor, éste cambia su estado a **Iniciando**.
4. Se imprime el estado actual del servidor.
5. Después de 10 segundos el servidor cambia automáticamente al estado **Iniciado**.
6. Se imprime el estado actual del servidor "***Start***".
7. Se envía el mensaje 1.
8. Se envía el mensaje 2.
9. Se envía el mensaje 3.
10. Se envía el mensaje 4.

11. Se envía el mensaje 5.
12. Se envía el mensaje 6 pero el número máximo de solicitudes se alcanzó.
13. El servidor cambia al estado de **Saturado**.
14. El servidor procesa el mensaje 1.
15. El servidor procesa el mensaje 2.
16. El servidor procesa el mensaje 3.
17. El servidor procesa el mensaje 4.
18. El servidor procesa el mensaje 5.
19. Se apaga el servidor y no se aceptan más mensajes.

Como vemos, el patrón *State* se encargó de controlar el comportamiento del servidor en tiempo de ejecución, por ahora podemos realizar más pruebas y ver cómo el servidor se comporta a medida que interactuamos con él.

Siguientes pasos:

Hemos demostrado la utilidad del patrón *State* en escenarios donde la aplicación se tendrá que comportar de formas distintas según el estado en el que se encuentre la aplicación, un escenario común donde podemos implementar este patrón es donde vemos código que basa su funcionamiento en un *switch* o en una serie de instrucciones *else if*.

Por hora hemos terminado de implementar el server pero ¿qué pasaría si decidimos agregar nuevos estados?
Para fortalecer los conocimientos adquiridos agregaremos nuevos estados al servidor como ejercicios para este capítulo:

- Apagado Seguro: Una vez establecido no permite enviar más mensajes y el servidor no debe apagarse hasta que todos los mensajes que entraron (antes de establecer el estado) sean procesados. Una vez que el servidor se apague cambiará su estado al de Apagado.

- Justo a tiempo (Just o time): Este inicio permitirá que el servidor inicie de inmediato pasando directamente a Iniciado sin pasar por Iniciando, sin embargo, el servidor tendrá que pasar por el estado iniciando cuando el primer mensaje llegue, de esta manera creamos un servidor que inicie sus componentes sólo en el momento que son requeridos. Tras pasar el periodo de inicialización, el servidor deberá pasar nuevamente al estado Iniciado.

Patrón Visitor

El patrón de diseño *Visitor* se utiliza para separar la lógica u operaciones que se pueden realizar sobre una estructura compleja. En ocasiones nos podemos encontrar con estructuras de datos que requieren realizar operaciones sobre ella, pero estas operaciones pueden ser muy variadas e incluso se pueden desarrollar nuevas a medida que la aplicación crece.
A medida que estas operaciones crecen, el número de operaciones que deberá tener la estructura también crecerá haciendo que administrar la estructura sea muy complejo. Por esta razón el patrón de diseño *Visitor* propone la separación de estas operaciones en clases independientes llamadas Visitantes, las cuales son creadas implementando una interface común y no requiere modificar la estructura inicial para agregar la operación.

El visitante opera, por lo general, sobre estructuras de datos jerárquicas o en forma de Árbol. Estas estructuras deberán cumplir con una simple regla y es que todos los objetos de la estructura implementarán un método en común que permita visitar el objeto. Este método por lo general es llamado *accept* y tiene como único parámetro al visitante.

Una de las principales desventajas de este patrón, es que si agrega un nuevo objeto dentro de la estructura, nos obligará a modificar todos los visitantes que tengamos implementados si es que queremos que tome en cuenta los nuevos objetos agregados a la estructura.

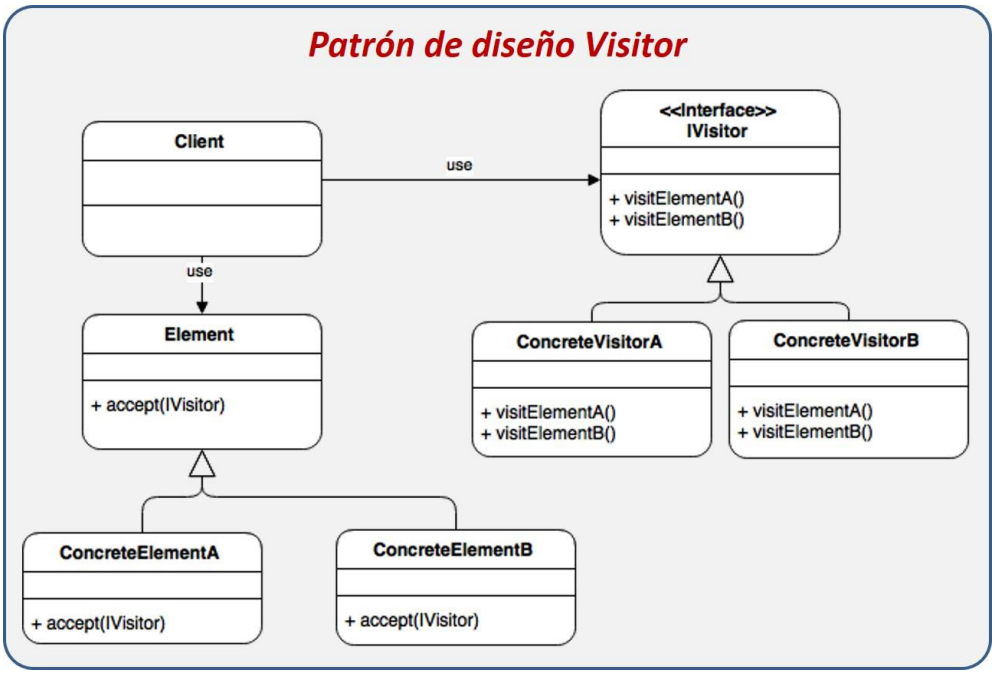


Ilustración 218: Estructura del patrón Visitor.

A continuación se explican los componentes que conforman el patrón.

- **Cliente:** Componente que interactúa con la estructura (*element*) y con el Visitante, éste es responsable de crear los visitantes y enviarlos al elemento para su procesamiento.
- **Element:** Representa la raíz de la estructura, en forma de árbol, sobre la que utilizaremos el Visitante. Este objeto por lo general es una interface que define el método *accept* y deberán implementar todos los objetos de la estructura.
- **ConcreteElement:** Representa un hijo de la estructura compuesta, la estructura completa puede estar compuesta de un gran número de estos objetos y cada uno deberá implementar el método *accept*.
- **IVisitor:** Interface que define la estructura del visitante, la interface deberá tener un método por cada objeto que se requiera analizar de la estructura (*element*).
- **ConcreteVisitor:** Representa una implementación del visitante, esta implementación puede realizar una operación sobre el *element*. Es

posible tener todos los *ConcreteVisitor* necesarios para realizar las operaciones que necesitemos.

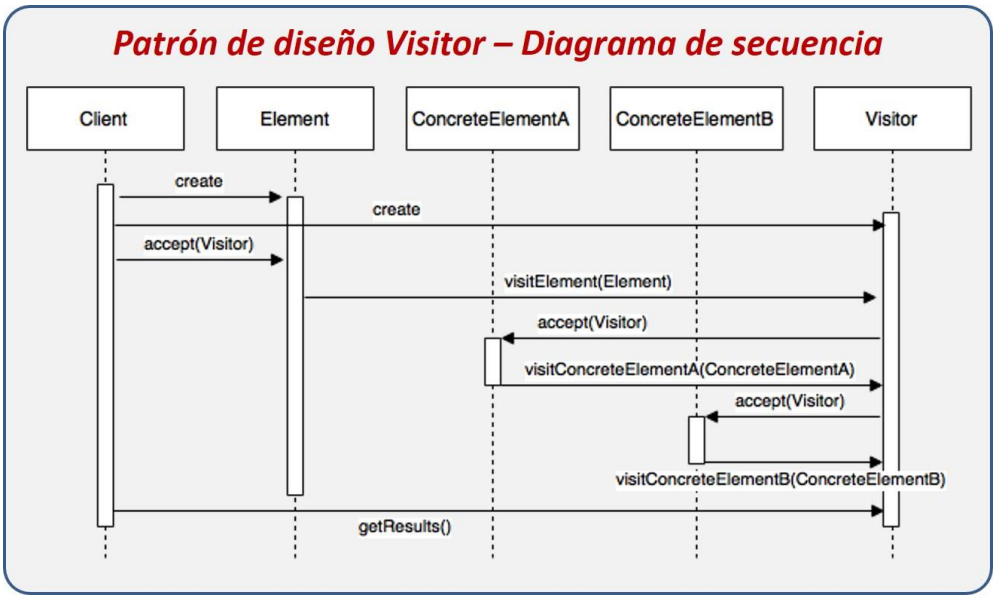


Ilustración 219: Diagrama de secuencia del patrón Visitor.

El diagrama anterior se interpreta de la siguiente manera:

1. El cliente crea la estructura (*Element*).
2. El cliente crea la instancia del Visitante a utilizar sobre la estructura.
3. El cliente ejecuta el método *accept* de la estructura y la envía al Visitante.
4. El *Element* le dice al Visitante con que método lo debe procesar. El Visitante deberá tener un método para cada tipo de clase de la estructura.
5. El Visitante analiza al *Element* mediante su método *visitElement* y repite el proceso de ejecutar el método *accept* sobre los hijos del *Element*. Nuevamente el Visitante deberá tener un método para procesar cada clase hija de la estructura.
6. El ConcreteElementA le indica al Visitante con qué método debe procesarlo, el cual es *visitElementA*.
7. La visitante continúa con los demás hijos de *Element* y esta vez ejecuta el método *accept* sobre el *ConcreteElementB*.

8. El *ConcrteElementB* le indica al Visitante con qué método debe procesarlo, el cual es *visitElementB*.
9. Finalmente el Visitante termina la operación sobre la estructura cuando ha recorrido todos los objetos, obteniendo un resultado que es solicitado por el cliente mediante el método *getResults* (el resultado es opcional ya que existen operaciones que no arrojan resultados).

Podemos apreciar que en este patrón la estructura tiene que decirle al visitante qué operación del visitante debe utilizar para procesarlo. Esto puede resultar muy extraño, sin embargo, tiene grandes ventajas que analizaremos más adelante.

También, vemos que el Visitante debe tener un método para procesar cada clase de la estructura, no obstante, pueden existir métodos que procesen más de un tipo de clase, esto se puede dar en clases que heredan del mismo padre y que por tanto se pueden tratar de forma similar por medio del polimorfismo, por lo que hay que tener cuidado en estos casos.

El escenario:

Para comprender mejor el funcionamiento de este patrón analizaremos la estructura de un proyecto de construcción que se asemeja bastante a los proyectos de software.
Tenemos un proyecto que tiene una serie de actividades y éstas tienen un precio al cliente y un costo. Esta estructura se parece mucho a las que hacemos en Microsoft Project, la cual es una estructura en árbol con una serie de actividades jerárquicas donde cada actividad es hija de otra.

Nos han solicitado que dada una estructura de proyecto obtengamos los siguientes datos:

- **Costo total del proyecto:** Calculado sumando el costo de todas las actividades del proyecto.
- **Precio total al cliente:** Calculado sumando el precio de todas las actividades del proyecto.
- **Utilidad del proyecto:** Precio menos el costo (Utilidad = Precio – Costo).
- Listado de todos los empleados involucrados en el proyecto así como el total de lo que se le tendrá que pagar a cada uno en todo el proyecto, se obtiene sumando el costo de cada actividad y agrupándolo por empleado.

Otra de las características es que nuestro cliente quiere tener la posibilidad de obtener más datos del proyecto sin realizar muchos cambios.

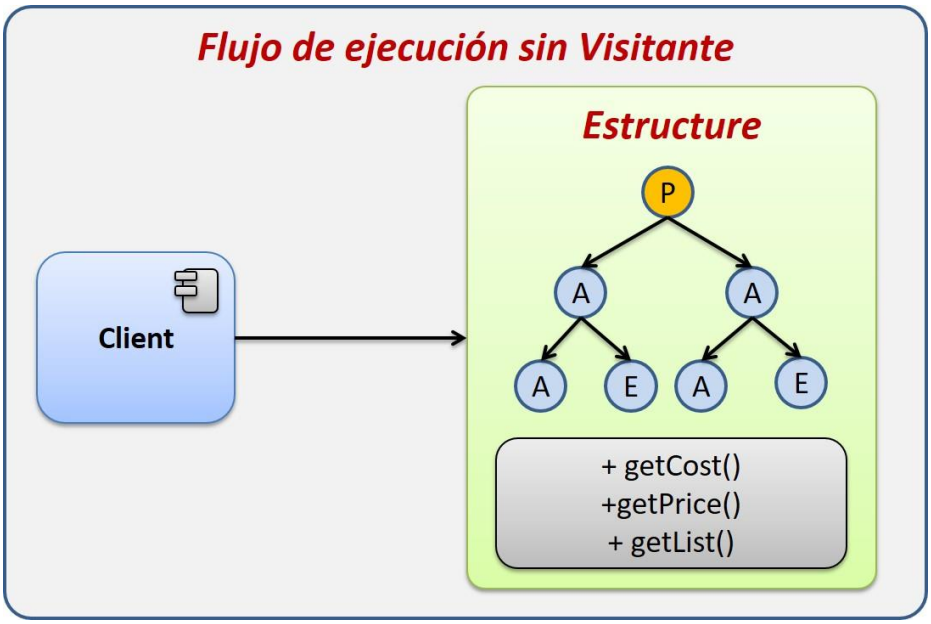


Ilustración 220: Implementación de la solución sin utilizar el patrón.

En la imagen podemos apreciar cómo sería una implementación típica sin utilizar el patrón Visitante en la cual, la misma estructura define las operaciones para obtener los datos necesarios. Desde luego que no estamos diciendo que definir operaciones en la estructura sea malo, sin embargo, tiene algunos inconvenientes.

Imaginemos que el cliente empieza a solicitar más y más operaciones sobre el proyecto y terminamos teniendo una aplicación tan compleja como un Microsoft Project, el cual nos permite gran cantidad de operaciones sobre el proyecto como asignar fechas, ruta crítica, asignar dependencias de actividades, entre otras cosas. Cuando esto suceda nuestra estructura será tan grande que será muy difícil de mantener, habrá tantas operaciones que será difícil identificar para qué es cada una.

En la siguiente sección veremos cómo resolver esto separando las operaciones de la estructura mediante el patrón de diseño Visitante.

La solución:

Ya analizamos algunos de los problemas que ocasiona implementar toda la funcionalidad directamente sobre la estructura, por lo que hora veremos cómo solucionar este problema utilizando el patrón de díseño Visitante.

Como ya vimos este patrón nos permite separar los algoritmos de la estructura, evitando tener toda la funcionalidad del lado de la estructura y delegando esta responsabilidad a los Visitantes. Veamos la siguiente imagen para comprender cómo es que esto quedaría:

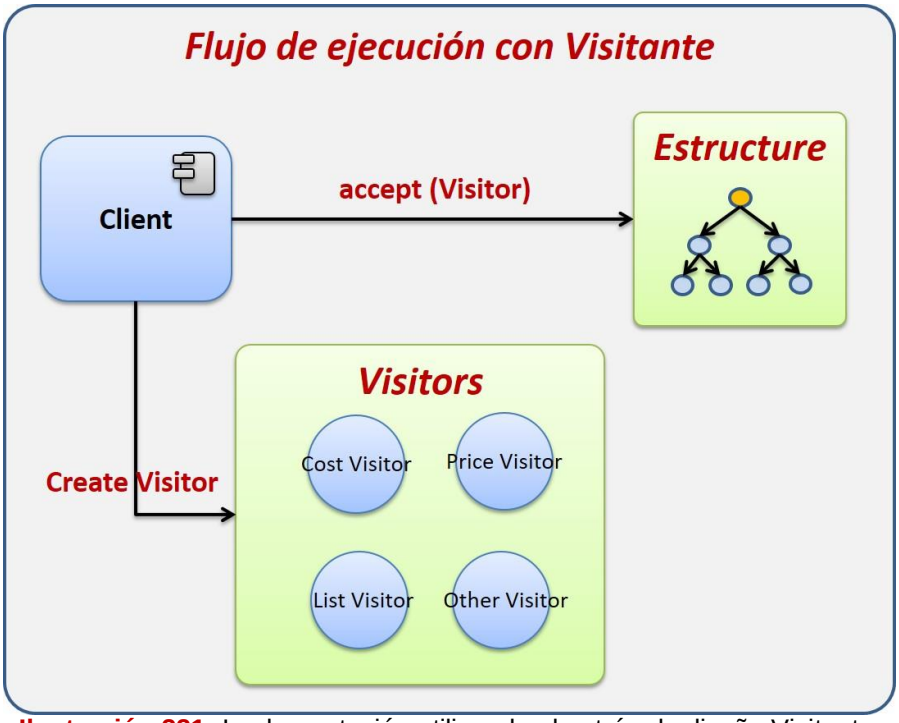


Ilustración 221: Implementación utilizando el patrón de diseño Visitante.

Una gran diferencia que podemos apreciar en esta imagen es que la estructura se limita únicamente a almacenar y organizar la información y por otra parte, tenemos un conjunto de visitantes diseñados para ejecutar ciertos algoritmos sobre la estructura. Esto ofrece grandes ventajas, sobre todo porque nuestro programa está más organizado, es mucho más fácil documentar y es más simple

mantener pequeñas clases que ejecutan una operación sobre la estructura que una gran clase con una infinidad de operaciones.

Una de las principales características que ofrece este patrón es que si requerimos realizar una nueva operación sobre el modelo, sólo necesitamos crear un nuevo Visitante en lugar de tener que modificar la estructura.

La implementación:

Una vez explicado el escenario y que comprendemos lo que tenemos que hacer, iniciaremos con la implementación de la solución.

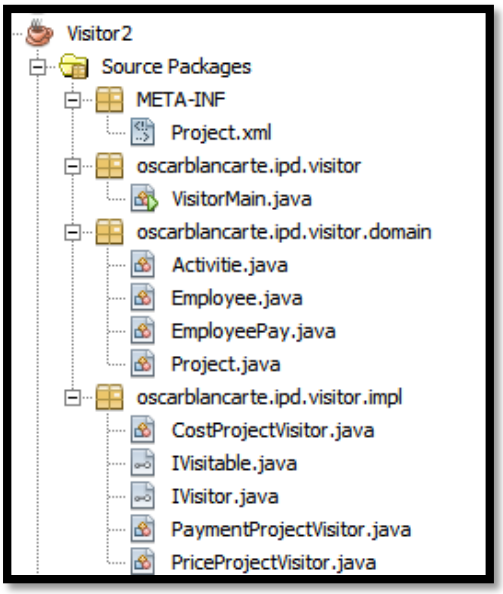


Ilustración 222: Estructura del proyecto Visitor.

Los paquetes del proyecto son los siguientes:

- **oscarblancarte.ipd.visitor:** Paquete principal del proyecto que contiene la clase ejecutable.
- **oscarblancarte.ipd.visitor.domain:** Contiene las clases de dominio utilizadas para representar la estructura del proyecto.
- **oscarblancarte.ipd.visitor.impl:** Contiene las clases relevantes para implementar el patrón *Visitor*.
- **META-INF:** Contiene el XML que contiene la configuración del proyecto, el cual será pasado a objetos en tiempo de ejecución.

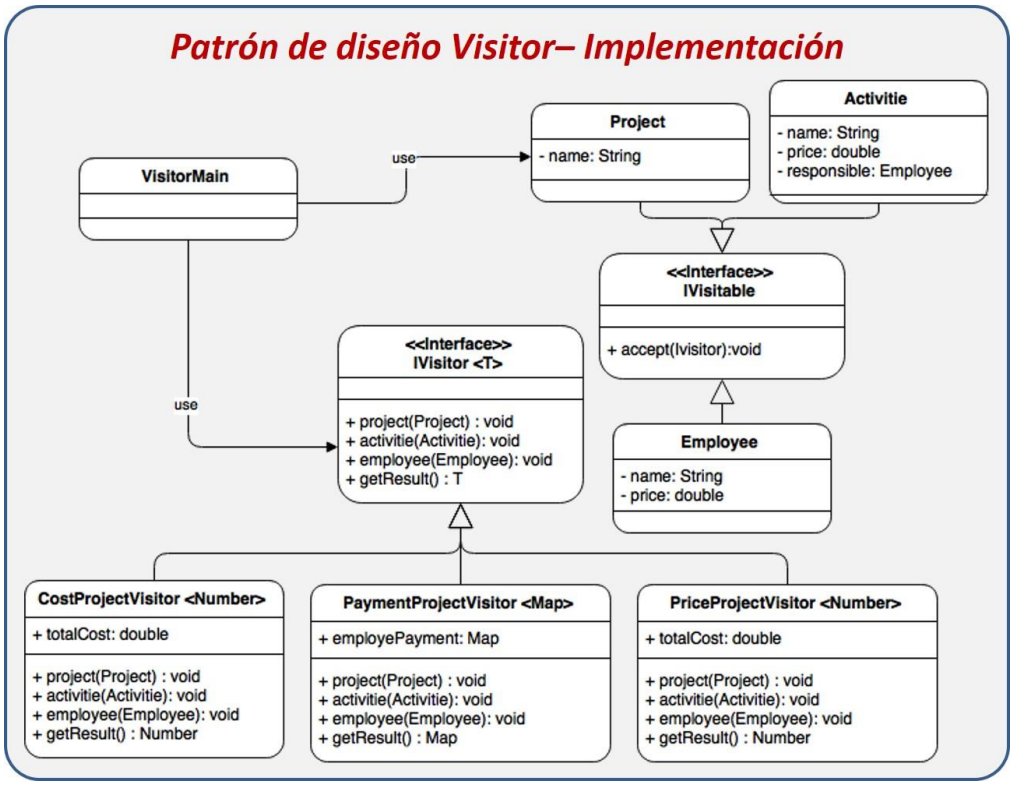


Ilustración 223: Diagrama de clases del proyecto Visitor.

En la imagen se muestran las clases del proyecto así como la forma en que están relacionadas entre sí.

Clase Project:

Esta clase corresponde a la estructura del proyecto y dentro de ella se encuentran todos los objetos que analizaremos mediante el patrón Visitante. Esta clase al igual que todas las que conforman la estructura, tiene que heredar de *IVisitable* que agrega el método accept indispensable para que el Visitante pueda visitar todos los objetos de la estructura del proyecto.

NOTA: Todas las clases del proyecto tiene anotaciones (Metadatos) los cuales los utilizaremos únicamente para convertir el XML *Project.xml* del paquete META-INF en estas clases, por lo tanto no afectan en nada el funcionamiento del patrón.

No entraremos en detalle al hacer estas anotaciones debido a que están fuera del alcance de este libro.

Se definen las siguientes propiedades:

- **name:** Indica el nombre del proyecto.
- **activities:** Lista con todas las actividades que conforman el proyecto.

Los métodos relevantes son las siguientes:

- **accept:** Es forzoso implementar este método debido a que está definido por la interface *IVisitable*, se utilizará más adelante para visitar a este objeto. Todas las clases lo deberán implementar y le tendrán que decir al visitante con qué método debe procesarlo, es por eso que se recibe al visitante y luego ejecuta el método Project (*this*), con esto le estamos indicando al Visitante cómo es que deberá procesar a ella misma.

```
1.  package oscarblancarte.ipd.visitor.domain;
2.
3.  import java.util.*;
4.  import javax.xml.bind.annotation.*;
5.  import oscarblancarte.ipd.visitor.impl.*;
6.
7.  @XmlRootElement(name = "Project")
8.  @XmlAccessorType(XmlAccessType.FIELD)
9.  public class Project implements IVisitable{
10.
11.     @XmlAttribute(name = "name")
12.     private String name;
13.     @XmlElement(name = "Activities")
14.     private List<Activitie> activities;
15.
16.     public Project() {
17.     }
18.
19.     public Project(String name) {
20.         this.name = name;
21.     }
22.
23.     public String getName() {
24.         return name;
25.     }
26.
```

```
27.     public void setName(String name) {
28.         this.name = name;
29.     }
30.
31.     public List<Activitie> getActivities() {
32.         if(this.activities==null){
33.             this.activities = new ArrayList<>();
34.         }
35.         return activities;
36.     }
37.
38.     public void setActivities(List<Activitie> activities) {
39.         this.activities = activities;
40.     }
41.
42.     public void addActivitie(Activitie activitie){
43.         if(this.activities ==null){
44.             this.activities = new ArrayList<>();
45.         }
46.         this.activities.add(activitie);
47.     }
48.
49.     @Override
50.     public void accept(IVisitor visitor) {
51.         visitor.project(this);
52.     }
53. }
```

Clase Activitie:

Una actividad puede tener dentro otras actividades y esta clase representa las distintas actividades que puede tener un proyecto.

Se definen las siguientes propiedades:

- **name:** Nombre de la actividad
- **price:** Precio final al cliente por esta actividad
- **responsable:** Responsable de realizar la actividad
- **activities:** Lista de actividades que componen esta actividad.

Como único método relevante se tiene accept que, como ya platicamos, deberá indicarle al visitante con qué método debe de procesarlo. En este caso se utiliza el método activitie.

```
1.  package oscarblancarte.ipd.visitor.domain;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import javax.xml.bind.annotation.XmlAccessType;
6.  import javax.xml.bind.annotation.XmlAccessorType;
7.  import javax.xml.bind.annotation.XmlAttribute;
8.  import javax.xml.bind.annotation.XmlElement;
9.  import javax.xml.bind.annotation.XmlRootElement;
10.  import oscarblancarte.ipd.visitor.impl.IVisitable;
11.  import oscarblancarte.ipd.visitor.impl.IVisitor;
12.
13.  @XmlRootElement(name = "Activitie")
14.  @XmlAccessorType(XmlAccessType.FIELD)
15.  public class Activitie implements IVisitable{
16.
17.      @XmlAttribute(name = "name")
18.      private String name;
19.      @XmlAttribute(name = "price")
20.      private double price;
21.      @XmlElement(name = "responsible")
22.      private Employee responsible;
23.      @XmlElement(name = "Activities")
24.      private List<Activitie> activities;
25.
26.      public Activitie() {
27.      }
28.
29.      public Activitie(String name, double price, Employee responsible) {
30.          this.name = name;
31.          this.price = price;
32.          this.responsible = responsible;
33.      }
34.
35.      /**
36.       Método GET y SET.
37.      **/
38.
39.      public List<Activitie> getActivities() {
40.          if(this.activities == null){
41.              this.activities = new ArrayList<>();
42.          }
43.          return activities;
44.      }
45.
46.      public void setActivities(List<Activitie> activities) {
47.          this.activities = activities;
48.      }
49.
50.      public void addActivitie(Activitie activitie) {
51.          if (this.activities == null) {
52.              this.activities = new ArrayList<>();
53.          }
```

```
54.         this.activities.add(activitie);
55.     }
56.
57.     public void removeActivitie(Activitie activitie) {
58.         if (this.activities == null) {
59.             this.activities = new ArrayList<>();
60.         }
61.         this.activities.remove(activitie);
62.     }
63.
64.     @Override
65.     public void accept(IVisitor visitor) {
66.         visitor.activitie(this);
67.     }
68. }
```

Clase Employee:

Clase que representa un empleado, los empleados serán asignados como responsables de realizar las actividades.

Se definen las siguientes propiedades:

- **name:** Nombre del empleado
- **price:** El costo del empleado por cada actividad que realice.

Como único método relevante se tiene accept el cual, como ya platicamos, deberá indicarle al visitante con qué método lo deberá de procesar.

```
1.  package oscarblancarte.ipd.visitor.domain;
2.
3.  import java.util.Objects;
4.  import javax.xml.bind.annotation.*;
5.  import oscarblancarte.ipd.visitor.impl.*;
6.
7.  @XmlRootElement(name = "Responsible")
8.  @XmlAccessorType(XmlAccessType.FIELD)
9.  public class Employee implements IVisitable {
10.
11.     @XmlAttribute(name = "name")
12.     private String name;
13.     @XmlAttribute(name = "price")
14.     private double price;// for Hour
15.
16.     public Employee() {
```

```
17.     }
18.
19.     public Employee(String name, double price) {
20.         this.name = name;
21.         this.price = price;
22.     }
23.
24.     public String getName() {
25.         return name;
26.     }
27.
28.     public void setName(String name) {
29.         this.name = name;
30.     }
31.
32.     public double getPrice() {
33.         return price;
34.     }
35.
36.     public void setPrice(double price) {
37.         this.price = price;
38.     }
39.
40.     @Override
41.     public void accept(IVisitor visitor) {
42.         visitor.employee(this);
43.     }
44.
45.     @Override
46.     public int hashCode() {
47.         int hash = 7;
48.         hash = 37 * hash + Objects.hashCode(this.name);
49.         return hash;
50.     }
51.
52.     @Override
53.     public boolean equals(Object obj) {
54.         if (obj == null) {
55.             return false;
56.         }
57.         if (getClass() != obj.getClass()) {
58.             return false;
59.         }
60.         final Employee other = (Employee) obj;
61.         if (!Objects.equals(this.name, other.name)) {
62.             return false;
63.         }
64.         return true;
65.     }
66.
67. }
```

Interface IVisitable:

Interface necesaria para indicar que un objeto puede ser visitado por el Visitante, por lo que todos los objetos que conforman el proyecto (Clase) deberán implementar esta interface. En ésta se define el método accept que tiene como único parámetro el visitante.

```
1. package oscarblancarte.ipd.visitor.impl;
2.
3. public interface IVisitable {
4.     public void accept(IVisitor visitor);
5. }
```

Interface IVisitor:

Esta es la clase clave del patrón, este interface será utilizada para crear los Visitantes concretos. Notemos que se define un método para cada clase de la estructura. Por ejemplo: el método *project* recibe como parámetro la clase *Project*, el método *activitie* recibe como parámetro la clase *Activitie* y así sucesivamente para cada objeto que tenga nuestra estructura.

Adicional a estos métodos he definido un método extra, *getResult*, que nos servirá para obtener el resultado que arroje el visitante tras haber recorrido la estructura. Este método regresa un tipo genérico que debemos definir en el visitante concreto.

NOTA: Si un nuevo objeto es agregado a la estructura, será necesario modificar esta interface para agregar el método correspondiente y esto afectará a todos los visitantes concretos que tengamos definidos.

```
1. package oscarblancarte.ipd.visitor.impl;
2.
3. import oscarblancarte.ipd.visitor.domain.Activitie;
4. import oscarblancarte.ipd.visitor.domain.Employee;
5. import oscarblancarte.ipd.visitor.domain.Project;
6.
7. public interface IVisitor<T> {
8.     public void project(Project project);
9.     public void activitie(Activitie activitie);
10.     public void employee(Employee employee);
```

```
11.     public T getResult();
12. }
```

Clase CostProjectVisitor:

Esta clase es un Visitante *concrete* y la utilizaremos para obtener el costo total del proyecto. Observemos que hereda de *IVisitor* y que define su tipo genérico como *Number*, esto nos indica que su resultado será numérico.

Analicemos como implementa los métodos:

- **project:** El proyecto únicamente le indica a sus actividades que acepten al visitante y termina.
- **activitie:** La actividad le solicita a su responsable que acepte al visitante y luego le solicita a todas las actividades que la componen, que también acepten al visitante.
- **Employee:** Para este visitante el empleado tiene el dato que le interesa: el costo, es por eso que este método suma el costo en una variable local que será el resultado final tras evaluar toda la estructura.
- **getResult:** Regresa el costo total del proyecto una vez que la estructura fue totalmente visitada.

```
1.  package oscarblancarte.ipd.visitor.impl;
2.
3.  import oscarblancarte.ipd.visitor.domain.*;
4.
5.  public class CostProjectVisitor implements IVisitor<Number> {
6.
7.      private double totalCost;
8.
9.      @Override
10.     public void project(Project project) {
11.         for(Activitie act : project.getActivities()){
12.             act.accept(this);
13.         }
14.     }
15.
16.     @Override
17.     public void activitie(oscarblancarte.ipd.visitor.domain.Activitie activitie) {
18.         activitie.getResponsible().accept(this);
19.         for(Activitie act : activitie.getActivities()){
20.             act.accept(this);
```

```
21.         }
22.     }
23.
24.     @Override
25.     public void employee(Employee employee) {
26.         totalCost += employee.getPrice();
27.     }
28.
29.     @Override
30.     public Number getResult() {
31.         return totalCost;
32.     }
33. }
```

Veamos que todo se ejecuta en cadena, el proyecto le indica a sus actividades que acepten al visitante, la actividad le indican a sus responsables que acepten al visitante del cual se obtiene el costo del proyecto, todo esto se hace de manera recursiva por lo que toda la estructura del proyecto se recorrerá.

Clase PriceProjectVisitor:

Esta clase es parecida al *CostProjectVisitor* por lo que no entraremos mucho en detalle, en general es muy similar, la única diferencia es que ésta obtiene el precio total del proyecto, para esto se suma el precio de todas las actividades en lugar del costo.

Observemos que el método *employee* no lo implementamos y esto se debe a que para este visitante no necesitamos nada del empleado, sin embargo, en el método *activitie* se realiza la suma del precio de cada actividad que es visitada, de esta manera obtenemos el precio total del proyecto.

```
1.  package oscarblancarte.ipd.visitor.impl;
2.
3.  import oscarblancarte.ipd.visitor.domain.*;
4.
5.  public class PriceProjectVisitor implements IVisitor<Number> {
6.
7.      private double totalPrice;
8.
9.      @Override
10.     public void project(Project project) {
11.         for(Activitie act : project.getActivities()){
12.             act.accept(this);
13.         }
14.     }
```

```
15.
16.     @Override
17.     public void activitie(oscarblancarte.ipd.visitor.domain.Activitie activitie) {
18.         totalPrice += activitie.getPrice();
19.         for(Activitie act : activitie.getActivities()){
20.             act.accept(this);
21.         }
22.     }
23.
24.     @Override
25.     public void employee(Employee employee) {
26.         // Not Interesting for this Visitor
27.     }
28.
29.     @Override
30.     public Number getResult() {
31.         return totalPrice;
32.     }
33. }
```

Clase PeymentProjectVisitor:

Este visitante es un poco distinto a los otros dos, ya que regresará una lista de todos los empleados del proyecto así como el total a pagarles durante toda su participación. Para lograrlo será necesario sumar el costo de cada empleado por cada actividad y agruparlo por empleado.

```
1.  package oscarblancarte.ipd.visitor.impl;
2.
3.  import java.util.*;
4.  import oscarblancarte.ipd.visitor.domain.*;
5.
6.  public class PaymentProjectVisitor
7.  implements IVisitor<List<EmployeePay>> {
8.
9.      private Map<String, Number> employeePayment = new HashMap<>();
10.
11.     @Override
12.     public void project(Project project) {
13.         for (Activitie act : project.getActivities()) {
14.             act.accept(this);
15.         }
16.     }
17.
18.     @Override
19.     public void activitie(Activitie activitie) {
20.         activitie.getResponsible().accept(this);
21.         for (Activitie act : activitie.getActivities()) {
```

```
22.             act.accept(this);
23.         }
24.     }
25.
26.     @Override
27.     public void employee(Employee employee) {
28.         String resp = employee.getName();
29.         if (employeePayment.containsKey(resp)) {
30.             employeePayment.put(resp,
31.                     employeePayment.get(resp).doubleValue()
32.                     + employee.getPrice());
33.         } else {
34.             employeePayment.put(resp, employee.getPrice());
35.         }
36.     }
37.
38.     @Override
39.     public List<EmployeePay> getResult() {
40.         List<EmployeePay> response = new ArrayList<>();
41.
42.         Set<String> keys = employeePayment.keySet();
43.         for (String key : keys) {
44.             response.add(new EmployeePay(key,
45.                     employeePayment.get(key).doubleValue()));
46.         }
47.
48.         return response;
49.     }
50.
51. }
```

Observemos que el tipo de dato retornado por el método *getResult* cambió de un dato numérico a una lista de objetos *EmployeePay*, los cuales únicamente agrupan el nombre del empleado y total a pagarle en todo el proyecto.

Todos los métodos son similares a los visitantes anteriores, con la diferencia de que el método *employee* sí trabaja de forma distinta.
Éste va sumando el costo de los empleados por actividad y los va agregando a un *Map*, donde la llave es el nombre del empleado y como valor guarda la suma de lo que se le tiene que pagar. Si el empleado ya existe en el *Map* entonces suma el monto, de lo contrario agrega al empleado con el monto inicial.

Finalmente, el método *getResult* convierte este *Map* en una lista de objetos *EmployeePay* para ser retornada.

Clase EmployeePay:

Clase utilizada por el Visitante *PaymentProjectVisitor* para regresar el total a pagar por empleado.

Se definen las siguientes propiedades:

- **employeeName:** Nombre del empleado
- **totalPay:** Total a pagar por todo el proyecto.

```
1.  package oscarblancarte.ipd.visitor.domain;
2.
3.  public class EmployeePay {
4.      public String employeeName;
5.      public double totalPay;
6.
7.      public EmployeePay(String employeeName, double totalPay) {
8.          this.employeeName = employeeName;
9.          this.totalPay = totalPay;
10.     }
11.
12.     /** Métodos GET y SET **/
13.
14.
15. }
```

XML Project.xml:

El siguiente XML se encuentra en el paquete *META-INF* que será cargado por la aplicación y convertido en un objeto *Project*. La finalidad de este XML es ver cómo es que está configurado el proyecto de una forma más amigable. Más adelante analizaremos cómo convertir este XML en un objeto *Project*.
Por el momento analicemos la información, veamos que el primer elemento del XML es el *tag Project* que representa el nombre del proyecto; como hijos al proyecto tenemos una serie de actividades que están formadas por más actividades y un Responsable. El responsable tendrá un precio, el cual se traduce al costo por la actividad. De igual forma, las actividades tienen un nombre y un precio que se traduce al cliente alfinal de la venta.

```
1.  <Project name="Departamento de 3 pisos">
```

```
2.    <Activities name="Supervisión de Arqutectura" price="160000.0">
3.      <responsible name="Oscar Blancarte" price="1500.0" />
4.      <Activities name="Primera planta" price="150000.0">
5.        <responsible name="Juan Perez" price="1300.0" />
6.        <Activities name="Obra negra" price="40000.0">
7.          <responsible name="Alejandro Lopez" price="300.0" />
8.        </Activities>
9.        <Activities name="Obra gris" price="30000.0">
10.         <responsible name="Vladimir Torres" price="300.0" />
11.       </Activities>
12.       <Activities name="Acabados" price="70000.0">
13.         <responsible name="Sofia Galindo" price="300.0" />
14.       </Activities>
15.     </Activities>
16.     <Activities name="Segunda planta" price="160000.0">
17.       <responsible name="Liliana Castro" price="1000.0" />
18.       <Activities name="Obra negra" price="50000.0">
19.         <responsible name="Laura Sanchez" price="1000.0" />
20.       </Activities>
21.       <Activities name="Obra gris" price="40000.0">
22.         <responsible name="Alejandro Lopez" price="300.0" />
23.       </Activities>
24.       <Activities name="Acabados" price="80000.0">
25.         <responsible name="Sofia Galindo" price="300.0" />
26.       </Activities>
27.     </Activities>
28.     <Activities name="Tercera planta" price="200000.0">
29.       <responsible name="Juan Perez" price="1300.0" />
30.       <Activities name="Obra negra" price="50000.0">
31.         <responsible name="Laura Sanchez" price="1000.0" />
32.       </Activities>
33.       <Activities name="Obra gris" price="40000.0">
34.         <responsible name="Vladimir Torres" price="300.0" />
35.       </Activities>
36.       <Activities name="Acabados" price="80000.0">
37.         <responsible name="Sofia Galindo" price="300.0" />
38.       </Activities>
39.     </Activities>
40.   </Activities>
41. </Project>
```

Clase VisitorMain:

Finalmente, tenemos a la clase principal del proyecto. Desde aquí comprobaremos el resultado de los distintos visitantes. Por hora analicemos el código para entender cómo es que el Visitante interactúa con la estructura.

```
1.  package oscarblancarte.ipd.visitor;
2.
3.  import java.util.List;
4.  import javax.xml.bind.*;
5.  import oscarblancarte.ipd.visitor.domain.*;
6.  import oscarblancarte.ipd.visitor.impl.*;
7.
8.  public class VisitorMain {
9.
10.     public static void main(String[] args) {
11.
12.         Project project = null;
13.         try {
14.             JAXBContext context =
15.                     JAXBContext.newInstance(Project.class);
16.             Unmarshaller unmarshaller = context.createUnmarshaller();
17.             project = (Project)unmarshaller
18.                     .unmarshal(VisitorMain.class
19.                             .getResourceAsStream("/META-INF/Project.xml"));
20.         } catch (Exception e) {
21.             System.out.println(e.getMessage());
22.             e.printStackTrace();
23.         }
24.         //Obtener el costo total del proyecto
25.         CostProjectVisitor costVisitor = new CostProjectVisitor();
26.         project.accept(costVisitor);
27.         double cost = costVisitor.getResult().doubleValue();
28.         System.out.println("Costo total > " + cost);
29.
30.         //Obtener el precio de venta del proyecto
31.         PriceProjectVisitor priceVisitor = new PriceProjectVisitor();
32.         project.accept(priceVisitor);
33.         double price = priceVisitor.getResult().doubleValue();
34.         System.out.println("Precio total > " + price );
35.         System.out.println("Ganancia Total > " + (price - cost));
36.
37.         //Mostrar el total a pagar por empleado
38.         System.out.println("\n:::::::: Pagar a los trabajadores :::::::");
39.         PaymentProjectVisitor paymentVisitor =
40.                 new PaymentProjectVisitor();
41.         project.accept(paymentVisitor);
42.         List<EmployeePay> result = paymentVisitor.getResult();
43.         for(EmployeePay pay : result){
44.             System.out.println(pay.getEmployeeName()
45.                     + " > " + pay.getTotalPay());
46.         }
47.     }
48. }
```

Lo primero que realizaremos será cargar el archivo *Project.xml* y convertirlo en la clase *Project*, esto lo hacemos en las líneas 14 a 19 utilizando la librería JAXB, propias del lenguaje Java.
Para no entrar en detalle acerca de esta API, ya que esta fuera del alcance de este libro, sólo mencionaré una librería especializada en la conversión de clases Java a XML y viceversa, a través de anotaciones en las clases utilizadas.

En la línea 25 creamos el visitante *CostProjectVisitor* para obtener el costo y en la línea 26 se lo enviamos al proyecto mediante el método *accept*. Cuando éste finalice tendremos el resultado mediante el método *getResponse*.

En las líneas 31 y 32 hacemos el mismo procedimiento pero para obtener el precio del proyecto, lo hacemos mediante el visitante *PriceProjectVisitor*.

En la línea 35 calculamos la utilidad del proyecto mediante la fórmula Precio – Costo.

De las líneas 38 a la 46 obtenemos el total a pagar por empleado mediante el visitante *PeymentProjectVisitor*.

Podemos apreciar que sobre la misma estructura implementamos el número de algoritmos que creemos convenientes con la gran ventaja de que podemos separarlo de la estructura.
Si en el futuro deseamos realizar nuevas operaciones sobre la estructura, solo tendríamos que crear un nuevo Visitante e implementar la lógica pertinente sin la necesidad de modificar la estructura.

La ejecución:

Ya hemos implementado la solución, por lo que ahora sólo nos falta ejecutar el programa para ver los resultados. Ejecutemos la clase *VisitorMain* para ver:

```
run:
Costo total > 9200.0
Precio total > 1150000.0
Ganancia Total > 1140800.0

:::::::: Pagar a los trabajadores :::::::
Vladimir Torres > 600.0
Laura Sanchez > 2000.0
Juan Perez > 2600.0
Oscar Blancarte > 1500.0
Liliana Castro > 1000.0
Alejandro Lopez > 600.0
Sofia Galindo > 900.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 224: Resultado de la ejecución.

Podemos apreciar los resultados arrojados por los visitantes, compararlos con nuestro XML para validar que son correctos.
El visitante *CostProjectVisitor* nos dice que el costo total del proyecto es de 9,200.00; el visitante *PriceProjectVisitor* nos dice que el precio total del proyecto es de 1,150,000.00; por lo tanto la utilidad es de 1,140,800.00.
Finalmente, tenemos el desglose del total a pagar por empleado y si lo sumamos deberá corresponder con el costo total del proyecto.

Como siguientes pruebas podemos modificar los montos por actividad o el costo de cada empleado, también podemos agregar más actividades e incluso anidar actividades dentro de otras y analizar los resultados.

Siguientes pasos:

Hemos comprobado la utilidad que tiene el visitante sobre estructuras complejas pero podríamos complicar más las cosas, realizando los siguientes ejercicios:

- Agregar fecha de inicio y terminación a cada actividad para determinar si un empleado está asignado al mismo tiempo en más de una actividad.

- Realizar un Visitante que recalcule el precio de las actividades padre mediante la suma del precio de todos sus hijos.

- Reemplazar al responsable de todas las actividades de un empleado por otro.

Podemos apreciar que el visitante nos brinda una gran cantidad de posibilidades sobre la estructura sin comprometer el funcionamiento de ésta.

A – Programación Orientada a Objetos

En esta sección hablaremos de los aspectos básicos de programación orientada a objetos que serán fundamentales para comprender el contenido de este libro. Si bien aquí se explicarán los conceptos claves, leer un libro especializado en programación orientada a objetos siempre será más recomendable.

Iniciaremos con las definiciones:

Clase: Una clase se podrá ver como el molde con el cual se crean los objetos, cada clase define la estructura y propiedades que deberán tener todos los objetos creados a partir de dicha clase.
Por ejemplo: Imaginemos que vamos a hacer gelatinas, para ello es necesario un molde, este molde definirá la forma en que la gelatina será creada.
En Java una clase es creada con el operador *class* seguida del nombre; por ejemplo:

```
1. public class Molde{ }
```

Entre las llaves debería de ir todo el cuerpo de la clase como son las propiedades y métodos.

Objeto: Formalmente un objeto es la instancia de una clase, es creado a partir de una clase y se pueden crear todos los objetos como queramos de una misma clase. Cuando un objeto es creado toma la estructura definida por la clase de la cual fue creada.
Para comprender mejor el concepto retomaremos el ejemplo de la gelatina; decíamos que el molde es la clase, pero en este caso la gelatina como tal es el objeto, de tal manera podríamos crear todas las gelatinas que queramos a partir del mismo modelo y todas las gelatinas compartirán la misma estructura o, en este caso, la forma.
Los objetos son creados con el operador *new* sobre la clase a instanciar.

```
1. Molde gelatina = new Molde();
```

En este caso estamos haciendo dos cosas; del lado izquierdo definimos la variable *gelatina* y le decimos que es de tipo *Model*, del lado derecho instanciamos la clase *Molde* con el operador *new* y la asignamos a la variable *gelatina*.
De esta manera podríamos crear el número de instancias que sean requeridas con tan sólo volver a utilizar el operador *new* sobre la clase *Model*.

Herencia: En programación orientada a objetos, se conoce como herencia a la característica de heredar las propiedades de otra clase. Esto permite que las clases que hereden de otras, hereden también su estructura y comportamiento. Así como nosotros heredamos nuestras características de nuestros padres.

En Java, al igual que en otros lenguajes de programación, sólo se permite la herencia de una sola clase y de varias interfaces.
Un ejemplo básico de herencia son los animales mamíferos, todos ellos tienen características básicas parecidas, pero cada especie tiene propiedades únicas. Por ejemplo, todos los mamíferos son vertebrados y de sangre caliente pero cada especie tiene sus propias características, por ejemplo, los perros tienen cuatro patas y caminan, mientras que los murciélagos tienen alas y vuelan, pero los dos siguen siendo mamíferos.

En POO a esta relación se le conoce como herencia y para que ésta se dé, debe de existir una clase base en común desde la cual todos los mamíferos hereden. En Java la herencia se realiza utilizando la instrucción *extends* de una clase a otra, por ejemplo:

```
1. public class Mammalian{}
2.
3. public class Dog extends Mammalian{}
4.
5. public class Bat extends Mammalian{}
```

Veamos que existe una clase Perro y otra Murciélago las cuales extienden de Mamífero, por lo tanto todos los métodos y propiedades que definamos en la clase Mamífero serán heredados por la clase Perro y Murciélago. Además, las clases Mamífero y Murciélago podrán definir las propiedades y métodos que los hagan únicos.

Polimorfismo: El polimorfismo es la propiedad de POO más complicada de entender debido a que tiene sentido hasta en tiempo de ejecución. Esta propiedad permite implementar una misma operación pero con comportamiento distinto.
Retomaremos el ejemplo de los mamíferos. Sabemos que todos los animales se comunican, todos utilizan algún lenguaje de sonidos que les ayuda a comunicarse con otros de su especie por lo que podremos crear un método abstracto en la clase Mamífero llamado comunicarse (*communicate*), el cual deberá ser implementado por cada especie.

```
1. public abstract class Mammalian{
2.    public abstract void communicate();
3. }
4.
5. public class Dog extends Mamifero{
6.    public abstract void communicate(){
7.       System.out.println("Ladrar");
8.    }
9. }
10.
11. public class Bat extends Mamifero{
12.    public abstract void communicate(){
13.       System.out.println("Utrasound )))" );
14.    }
15. }
```

Veamos el ejemplo anterior; tanto la clase Murciélago como la clase Perro implementan el método *communicate*, la firma del método es idéntica, por lo tanto el método es el mismo, sin embargo, su comportamiento es distinto.

Abstracción: La abstracción nos permite tratar a los objetos de la forma más básica posible abstrayendo todas aquellas características que no son relevantes en el momento. Para ello utilizamos las clases bases para manipularlos en lugar de la clase concreta. Analicemos el siguiente ejemplo:

```
1. Mammalian animal1 = new Dog();
2. animal1.communicate();
3. Mammalian animal2 = new Bat();
4. animal2.communicate();
```

Podemos ver que instanciamos la clase *Dog* y *Bat* pero la asignamos a una clase Mamífero, si ejecutamos el método *communicate* de los dos objetos, podremos saber el resultado que arrojarán debido a que conocemos su tipo de dato:

```
1. Mammalian animal = getNextMammalian();
2. animal.communicate();
```

Supongamos que el método *getNextMammalian* regresa cualquier mamífero de las más de 5,000 especies conocidas. ¿Sabrías que sonido arrojará el método *communicate*? Seguramente la respuesta es no y es la correcta, porque pasa esto: digamos que de forma polimórfica la clase verdadera detrás de la variable animal tendrá implementado el sonido adecuado, pero debido a que asignamos el tipo de dato a una súper clase estamos realizando una Abstracción.
De esta manera tratamos al verdadero mamífero de la forma más simple conocida; como un mamífero. A esta característica se le conoce como Abstracción.

Encapsulamiento: Es la característica de los objetos de ocultar sus propiedades y brindar métodos de acceso para manipular estas propiedades, evitando así que los demás objetos modifiquen directamente sus propiedades dejándolo en un estado inconsistente.

```
1. public class Dog{
2.    private String name;
3.
4.    public String getName(){
5.       return name;
6.    }
7.
8.    public void setName(String name){
9.       this.name = name;
10.    }
11. }
```

En la clase *Dog* podemos ver que la propiedad *name* tiene el modificador de acceso privado lo que impide que nadie pueda modificar su valor directamente. Por su parte, la clase *Dog* proporciona los métodos *getName* y *setName* que sirven para acceder y modificar el valor de la propiedad. De esta manera logramos el encapsulamiento, la clase *Dog* controla la forma en que sus propiedades son

modificadas. Por ejemplo, podríamos agregar validación al método *setName* para que no permita un nombre vacío o mayor a N caracteres.

Acoplamiento: El acoplamiento es la fuerza con la que dos clases o componentes están unidos, dicho de otra manera, indica el nivel de dependencia que tiene uno del otro. Entre mayor sea el acoplamiento entre dos componentes, mayor será la dependencia que existe entre éstos.
En la ingeniería de software siempre se busca el menor acoplamiento posible entre componentes, pero esto no siempre es posible, por eso existen técnicas y patrones de diseño que ayudan a desacoplar los componentes.
Se sabe que un componente está acoplado a otro cuando el primero no puede funcionar sin que el segundo esté presente y, del lado contrario, se dice que un componente está desacoplado cuando el primero puede seguir operando sin que el segundo esté presente.

Para comprender mejor este concepto imaginemos un escenario hipotético, donde tenemos un componente encargado de imprimir un mensaje en pantalla, este componente se llamará *Printer* e internamente utiliza otra clase que es la que realmente imprime el mensaje.

```
1.  public class Printer {
2.
3.      private ConsolePrinter strategy;
4.
5.      public Printer(ConsolePrinter strategy){
6.          this.strategy = strategy;
7.      }
8.
9.      public void generateMessage(){
10.         strategy.process("Hello world!");
11.     }
12.
13.     public static void main(String[] args) {
14.         Printer a = new Printer(new ConsolePrinter());
15.         a.generateMessage();
16.     }
17. }
```

La clase *Printer* recibe como parámetro en su constructor la instancia de la clase utilizada para imprimir los mensajes, en este caso recibe la clase *ConsolePrinter*.

```
1. public class ConsolePrinter {
2.     public void process(String message){
3.         System.out.println(message);
4.     }
5. }
```

La clase *ConsolePrinter* es la que realmente realiza la impresión en pantalla y está íntimamente relacionada con la clase *ConsolePrinter* y, si ésta no existe, tendríamos grandes problemas. **A este tipo de relación se les conoce como alto acoplamiento.**

Ahora bien, ¿qué tendríamos que hacer para desacoplar estas dos clases, de tal manera que la segunda no tenga o tenga muy poca dependencia de la *segunda* clase?
La solución es simple, lo primero que tenemos que hacer es crear una *Interface* que defina las operaciones básicas de la clase *ConsolePrinter*, seguido la clase *Printer* deberá recibir como parámetro esta interface en lugar de la clase concreta. Veamos cómo quedaría dicha interface:

```
1. public interface IPrinterStategy {
2.     public void process(String message);
3. }
```

Observemos que la interface tiene la operación *process*, la cual será ejecutada por la clase *Printer*.

```
1. public class ConsolePrinter implements IPrinterStategy{
2.     public void process(String message){
3.         System.out.println(message);
4.     }
5. }
```

La clase anterior es la ya conocida *ConsolePrinter* pero esta vez implementa la interface *IPrinterStrategy*, la cual obliga a implementar el método *process*.

```
1. public class Printer {
2.
3.     private IPrinterStategy strategy;
4.
5.     public Printer(IPrinterStategy strategy){
6.         this.strategy = strategy;
7.     }
```

```
8.
9.     public void generateMessage(){
10.        strategy.process("Hello world!");
11.    }
12.
13.    public static void wmain(String[] args) {
14.        Printer a = new Printer(new ConsolePrinter());
15.        a.generateMessage();
16.
17.        Printer b = new Printer(new GUIPrinter());
18.        b.generateMessage();
19.    }
20. }
```

Finalmente, veamos que en el constructor de la clase *Printer* recibe la interface *IPrinterStrategy* en lugar de la clase concreta *ConsolePrinter*. Pero, ¿qué ventajas ofrece esto? Primero que nada nos permite que la clase *Printer* pueda funcionar sin saber los detalles de la clase *ConsolePrinter*, sólo le interesa recibir como parámetro una implementación; segundo, la clase concreta puede variar sin afectar al Printer. Por ejemplo, veamos la línea 17 y 18, hemos creado una segunda estrategia para imprimir de forma gráfica un pop-up con el mensaje, pero en la líneas 14 y 15 seguimos trabajando con la consola. Con esto logramos desacoplar la clase *Printer* con la clase concreta para imprimir el mensaje.

Veamos cómo quedaría la clase *GUIPrinter*:

```
1. public class GUIPrinter implements IPrinterStategy{
2.
3.     @Override
4.     public void process(String message) {
5.         JOptionPane.showMessageDialog(null, message);
6.     }
7.
8. }
```

Como mencionamos anteriormente, la ingeniaría de software busca lograr, lo mayor posible, que los componentes estén desacoplados, esto se debe principalmente a que permite que el código sea más mantenible y permite que las dos componentes puedan variar sin afectar al otro. Pero recordemos que esto no siempre será posible y en no en todos los casos es lo mejor. También abusar de esta estrategia puede ser un error. Es importante evaluar las ventajas y

desventajas que nos ofrecerá en cada caso y decidir si tiene sentido desacoplar los componentes.

Cohesión: Así como el acoplamiento tiene que ver con cómo las clases se relacionan entre sí, la cohesión tiene que ver con la forma en que agrupamos unidades de software. Con la cohesión buscamos que todas las unidades de software que creemos estén diseñadas para atacar un solo problemas aumentando su reutilización. Pero para ser más claro veámoslo así: Cohesión es la forma en la que agrupamos funciones en una librería, o la forma en la que agrupamos métodos en una clase, o la forma en la que agrupamos clases en una librería, etcétera.
De esta manera nos aseguramos de que todo lo que construimos está íntimamente relacionado para resolver un problema específico y no lo mezclamos con otras cosas, de esta manera aumenta la reutilización de componentes de software que utilizamos.
¿Te imaginas una librería que sirva para procesar XML y para compresión de archivo? Creo que no, siempre que buscamos una librería sirve para otra cosa, por lo que si queremos una librería para procesar archivos XML y otra para compresión de archivos, encontraremos dos librerías independientes donde cada una sólo tenga como objetivo realizar una tarea. De esta manera, todas las clases que estén en la librería de XML serán exclusivamente para cumplir con el objetivo de procesar archivos XML y del mismo modo, para la librería de compresión de archivos. A este agrupamiento se le conoce como Cohesión.

Cuando empezamos a crear clases o agregar métodos que no son precisamente para resolver el problema, se dice que la cohesión es baja. En la ingeniería de software se busca que siempre exista una alta cohesión.

Analicemos el siguiente ejemplo:

```
1. public interface Calculator {
2.     public Number sum(Number a, Number b);
3.     public Number sub(Number a, Number b);
4.     public Number mult(Number a, Number b);
5.     public Number div(Number a, Number b);
6.     public void createGUI();
7. }
```

Aquí tenemos la interface de una calculadora que tiene métodos para las operaciones básicas. Tiene una operación para sumar, restar, multiplicar y dividir. Hasta aquí todo bien, queda claro que todas las operaciones están relacionadas y que atacan el mismo problema de implementar una calculadora. Pero después llega el método createGUI que se utiliza para crear la interface gráfica de usuario. ¿Qué pasaría si otro programa requiere utilizar esta clase para manejar sus operaciones? el método createGUI y todos los programas que lo implementen, quedarían de sobra ya que sólo sirven para un escenario concreto.
Por lo tanto, para aumentar la cohesión tendríamos que separar este método de la interface dejando únicamente los métodos de operaciones matemáticas.

```
1. public interface Calculator {
2.     public Number sum(Number a, Number b);
3.     public Number sub(Number a, Number b);
4.     public Number mult(Number a, Number b);
5.     public Number div(Number a, Number b);
6. }
```

Esta nueva variante de la interface tiene más sentido, pues todos sus métodos son relacionados. A esto se le conoce como alta cohesión.

Como conclusión podemos decir que siempre deberíamos de buscar que todos los componentes de software que desarrollemos tengan alta cohesión y bajo acoplamiento.

B – Introducción a UML

El Lenguaje Unificado de Modelado (o UML por sus siglas en inglés) es el lenguaje de modelos de sistemas de software más popular hasta la fecha. Esta notación nace de la unión de Rumbaugh, Grady Booch e Ivar Jacobson llamada los tres amigos, los cuales ya eran conocidos por proponer distintas metodologías de diseño. Pero fue hasta que *Rational* les solicitó la creación de un único lenguaje abierto que se juntaron para diseñar UML que es un lenguaje gráfico utilizado para especificar y documentar sistemas.
Es importante recalcar que UML es sólo un lenguaje de modelo y no un lenguaje de programación, por lo que únicamente nos sirve para modelar nuestros sistemas antes de su construcción, pero también sirve para documentarlos. UML es respaldado por la *Object Management Group* (OMG) y desde el año 2005 es un estándar aprobado por la ISO.

Una vez que ya hemos platicado un poco de la historia y orígenes de UML pasemos a la acción. Antes de iniciar, me gustaría aclarar que esta sección no pretende ser una guía completa de UML, en su lugar se realizará una introducción de los elementos necesarios para comprender los ejemplos expuestos en este libro. Si lo que deseas es aprender UML a profundidad, te recomendaría estudiar a fondo otras obras literarias como *UML y Patrones* de Craig Larman el cual me fue muy útil para comprender UML.

Diagrama de clases: Los diagramas de clases se utilizan para diseñar las diversas clases del sistema y cómo es que éstas se relacionan con el resto. Los diagramas de clases están clasificados como diagrama estático, debido a que no diagraman comportamiento, en su lugar sólo están definidas las clases y cómo se relacionan.
Un diagrama de clases está compuesto de los siguientes elementos:

- **Clases:** Son las clases reales que tendrá el proyecto, en ellas se pueden modelar los atributos, métodos y la visibilidad.

- **Relaciones:** Representan la forma en que las clases se relacionan entre sí. Los tipos de relación son Herencia, Composición, Agregación y Asociación.

Clase: La clase es la unidad básica que encapsula toda la información de un objeto y en UML es representada como un rectángulo con tres divisiones.

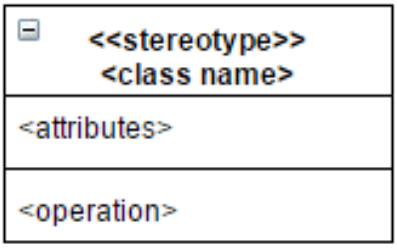


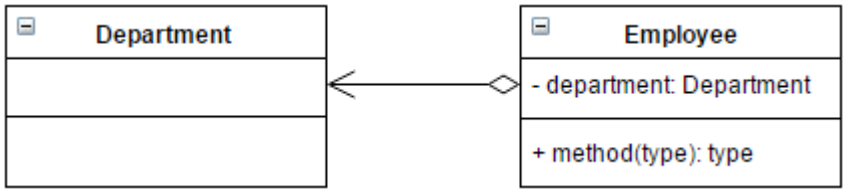
Ilustración 225: Figura de una clase.

Las secciones se describen a continuación:

- ***Class name* (primera sección):** En esta sección se describe el nombre real de la clase que deberá seguir los mismos estándares de programación como el *Camel Case* o cualquier otro utilizado. Cabe mencionar que sobre el nombre se puede agregar una etiqueta entre llaves francesas (« »), que son distintas a las dobles << >>. Un estereotipo sirve para denotar un significado que por UML no es posible dar, por ejemplo, podríamos poner «Abstract» para indicar que se trata de una clase abstracta o «Interface» para indicar que es una interface.
- ***Attributes* (Segunda sección):** En esta sección de describen todos los atributos que tiene la clase, lo atributos pueden ser declarados con los modificadores de acceso privados, protegidos o públicos. Más adelante hablaremos de los modificadores de acceso.
- ***Operations* (Tercera sección):** En esta sección se describen las operaciones o métodos que tiene la clase, los métodos nos dan una idea de cómo ésta interactúa con el resto de clases del diagrama.

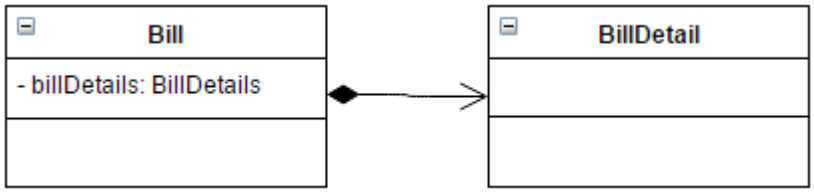
Relaciones: las relaciones definen las forma en que dos clases o más se relacionan entre sí, en UML existen diversos tipos de relaciones, las cuales tiene características articulares. Analicemos éstas relaciones:

- **Agregación:** Representada con la figura de un rombo y una flecha contínua que indica que la clase A contiene a B, pero B no comparte el tiempo de vida de A, por lo que cuando A deje de existir B continuará existiendo.



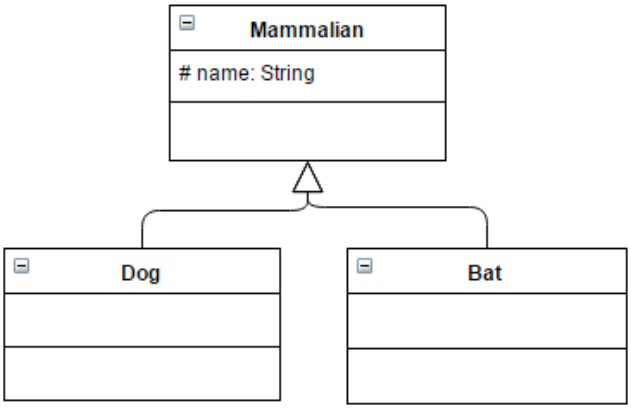
En esta relación un Empleado pertenece a un departamento y está relacionado con una agregación, lo que significa que si el empleado sale de la compañía y es eliminado, el departamento seguirá existiendo.

- **Composición:** Representada con un rombo relleno y una fecha contínua. Esta relación es parecida a la agregación, pero este tipo de relación es más fuerte, ya que si A contiene a B, B tendrá el mismo tiempo de vida que A, por lo tanto, si A deja de existir, también lo hará B.



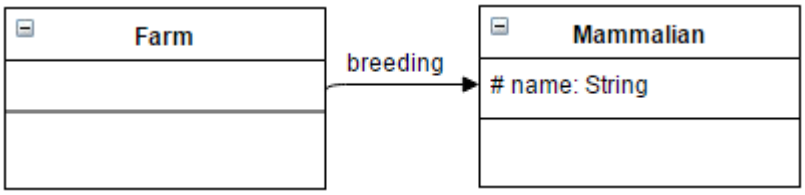
En esta relación una factura (*Bill*) puede tener N movimientos (*BillDetail*), pero los detalles son totalmente dependientes de la factura, por lo que si la factura es eliminada los detallas también deberán de ser eliminados, ya que por separado no tienen sentido de existir.

- **Herencia:** Representada por una flecha vacía contínua. La herencia, al igual que en la programación, indica que una subclase hereda las propiedades y comportamiento de una súper clase.



En esta relación estamos indicando que tanto la clase Perro (*Dog*) como Murciélago (*Bat*) heredan de una superclase llamada Mamífero (*Mammalian*). Así que, tanto el perro como el murciélago son mamíferos; además de que heredan sus propiedades y comportamiento, también heredan la propiedad nombre (*name*) de la clase mamífero.

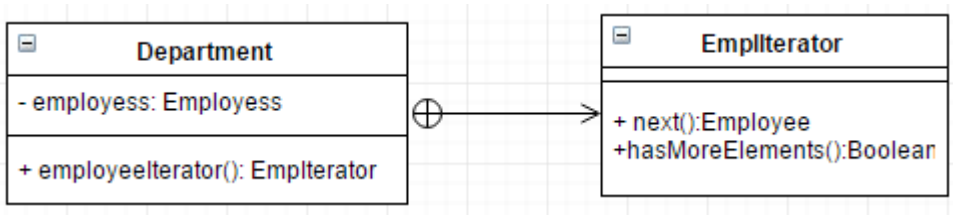
- **Asociación:** Representada como una flecha vacía contínua muy parecida a la de la herencia, pero la punta es mucho más chica. El significado de esta asociación es representar una colaboración entre las dos clases enlazadas, esto indica que puede existir una comunicación unidireccional o bidireccional. No es una relación fuerte, por lo que una clase A no comparte el tiempo de vida de B, si B deja de existir A continuará existiendo.



En esta relación tenemos una asociación entre la clase Granja (*Farm*) y la clase Mamífero (*Mammalian*). La relación por lo general nos dice el tipo de interacción que tiene, en este caso la asociación indica que la Granja cría (*breeding*)

Mamíferos. Podemos ver que no es una operación como tal, más bien denotamos de qué manera están interrelacionadas.

- **Inner class (Clase interna):** Representado con un círculo y una flecha contínua. Esta relación indica que una clase está declarada dentro de otra.



En esta asociación estamos indicando que la clase Departamento (*Department*) tiene dentro de ella a otra clase llamada *EmpIIterator*, el cual crea un iterador de tipo *EmpIIterator* con todos los empleados del departamento. Este tipo de relación se utiliza cuando no tiene sentido crear la clase de forma independiente y la creamos dentro de otra.

Modificadores de acceso: Los modificadores de acceso indican la visibilidad que tienen las propiedades y operaciones, es decir, cómo es que las demás clases las pueden ver. Los modificadores de acceso son mostrados antes de la definición de una propiedad u operación y son representados con un carácter. En UML podemos representar los siguientes modificadores de acceso:

- ***Public* (Público):** Representado con el carácter (+). Este modificador indica que cualquier clase puede ver la operación o propiedad.
- ***Private* (Privado):** Representado con el carácter (-). Este modificador indica que la operación o propiedad sólo puede ser visible para ella misma y el resto de clases no podrán verla.
- ***Protected* (Protegido):** Representado con el carácter (#). Este modificador indica que las propiedades y operaciones sólo serán visibles para ella misma y todas las subclases que hereden de ella.

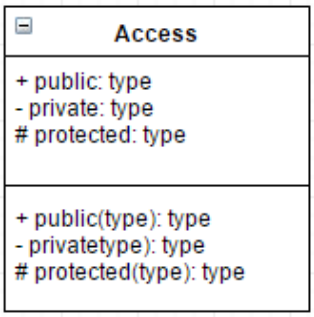


Ilustración 226: Ejemplos de modificadores de acceso.

Finalmente, se muestra una clase con operaciones y atributos con los distintos modificadores de acceso.

Diagrama de secuencia: Los diagramas de secuencias son utilizados para representar el intercambio de mensajes entre diversas clases, actores, componentes y subsistemas. En este tipo de diagramas se pueden identificar los mensajes que son enviados de cada clase en el tiempo.
Los diagramas de secuencia están clasificados como diagramas dinámicos, ya que en éstos se modela el comportamiento en tiempo de ejecución. Es decir, podemos diagramar los mensajes que se envían y el tiempo en que lo hacen, las respuestas que arrojan e incluso hasta cuándo un objeto deja de existir.

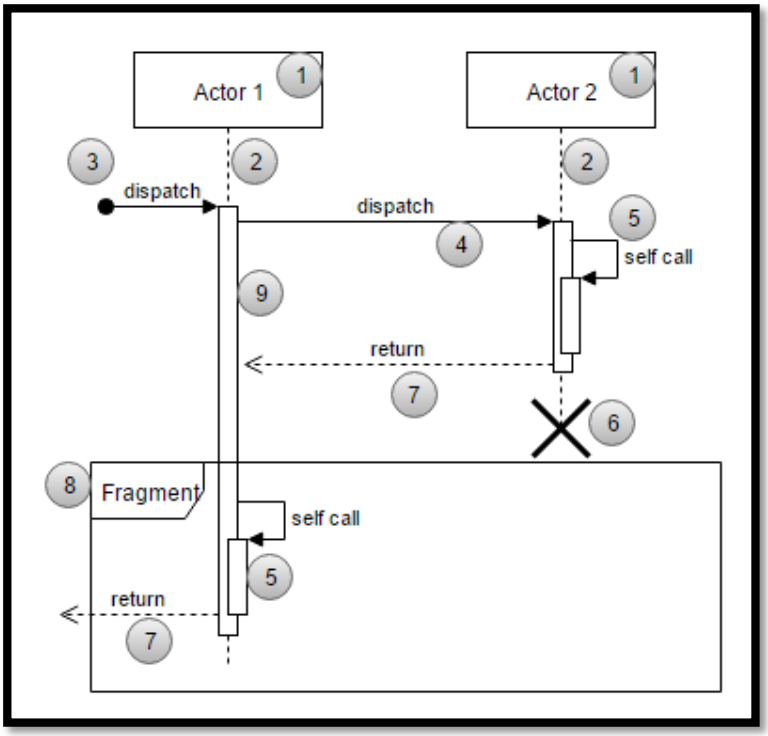


Ilustración 227: Diagrama de secuencia.

Un diagrama de secuencia esta conformado por muchos componentes así que pondré un pequeño ejemplo de un diagrama y analizaremos cada componente por separado.

1. **Actores:** Se representan como una caja en la parte superior del diagrama, un actor puede ser una clase, un componente o incluso otros subsistemas.
2. ***Lifetime*:** El *lifetime* o tiempo de vida de un actor es representado como una línea punteada que se extiende desde la caja del actor hacia abajo. Esta línea indica el tiempo que vivirá un actor. Entre mayor sea esta línea, más tiempo vivará el actor dentro del diagrama.
3. **Evento o mensaje de entrada:** Es un mensaje que entra desde un lugar deconocido, este mensaje es el que inicia la ejecución del diagrama.
4. **Llamadas (*Call*):** Representan la forma en que los actores se comunican, en estas líneas se especifica el mensaje enviado y la operación ejecutada.

5. **Auto llamada (*Self Call*):** Es una llamada que el mismo objeto hace a sí mismo, como por ejemplo, cuando un método del objeto manda llamar a otro método del mismo.
6. **Destructor (*Destruct*):** Esta tache indica que el objeto es destruido y que no será utilizado más el diagrama.
7. **Retorno (*Return*):** Esta flecha indica que se envía una respuesta tras ejecutar la operación solicitada.
8. **Fragmento:** Los fragmentos son utilizados para agrupar una sección o para darle un funcionamiento distinto, son utilizados con regularidad para indicar un ciclo o una condición.
9. **Ejecución:** Representa el tiempo de vida de la ejecución de una operación, entre más larga sea la línea más tiempo dura la ejecución de la operación en el diagrama.

Es importante mencionar que el diagrama se leé de arriba abajo, entre más abajo nos encontremos del diagrama más tiempo habrá transcurrido. Durante la ejecución del diagrama, una actividad deberá terminar para que la siguiente inicie, siempre y cuando sean invocaciones síncronas, de lo contrario el flujo continúa con el siguiente paso.

Este tipo de diagramas es utilizado con regularidad para diagramar el comportamiento de los sistemas, ya que permite identificar los componentes que interactúan y la forma en que se comunican entre sí.
Este tipo de diagramas son muy sencillos de interpretar y dejan muy claro el comportamiento del componte modelado. A lo largo de este libro utilizamos éstos diagramas para ilustrar cómo es que todos los componentes de los patrones de diseño interactúan entre sí para lograr el resultado esperado.

CONCLUSIONES

¡En hora buena! si has llegado hasta aquí es porque seguramente ya has terminado de leer este libro y eso te convierte en uno de los pocos programadores con una formación en patrones de diseño.

Como ya te habrás dado cuenta, a lo largo de este libro los patrones de diseño nos abren la mente no sólo para resolver los problemas aquí planteados, sino que a medida que aprendemos y practicamos seguramente se te habrá venido a la mente aquel sistema o aquel proyecto de la universidad que desarrollaste y que pudiste haber resuelto con alguno de los patrones de diseño que aquí estudiamos.

Aprender patrones de diseño es uno de los primeros pasos para convertirte en un programador de excelencia y un requisito obligatorio para ser un **Arquitecto de software**, es por eso que yo insisto a mis alumnos y a la gente que mentoreo en el trabajo, sobre la importancia de aprender y perfeccionar los patrones de diseño. A medida que más los practicas y los implementas en tu día a día será más fácil identificar con certeza dónde pueden ser aplicados e incluso podrás crear variantes de ellos sin equivocarte.

Recuerda que el mundo de las tecnologías es un ambiente muy dinámico donde lo que aprendemos hoy probablemente mañana no sirve, por lo que te invito a no dejar de aprender y seguir estudiando diariamente.

Finalmente, no me queda más que agradecerte por permitirme ser parte de tu crecimiento profesional y por la confianza depositada en mí al momento de adquirir este libro.

Si crees que este libro alcanzó tus expectativas y te fue de utilidad, puedes dejar tu reseña en *Amazon.com* y compartir con tus amigos la referencia a este libro.

GRACIAS.

BIBLIOGRAFÍA

Alfred V. Aho, M. S. *Compiladores: Principios, Tecnicas y Herramientas* (2 ed.). Pearson.

Cay S. Horstann, G. C. *Core Java 2* (7 ed., Vol. 2). Pearson.

Eric Freeman, B. B. *Head First Design Patterns* (1 ed.). O'Reilly.

Erich Gamma, R. H. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional.

Larman, C. *UML y Patrones* (2 ed.). Pearson.

Program, H. t. (2015). *Paul Deitel, Harvey Detail* (8 ed.). Pearson.

W. Clay Richardson, D. A. *Profesional Java JDK 6* (1 ed.). Wrox.