



DEI BACKEND I

CERTIFIED TECH DEVELOPER

PRIMER AÑO
TERCER BIMESTRE

Contenido

Módulo 1: Patrones de Diseño	4
Patrón Facade.....	4
Propósito y Solución.....	5
Ventajas y Desventajas.....	6
¿Cómo implementar el Patrón Facade?.....	6
Ejemplo de Patrón Facade.....	6
Patrón Template Method.....	9
Propósito y Solución.....	10
Ventajas y Desventajas.....	11
¿Cómo implementar el Patrón Template Method?	11
Ejemplo de Template Method.....	12
Patrón Cadena de Responsabilidad.....	13
Propósito y Solución.....	15
Ventajas y Desventajas.....	16
¿Cómo implementamos el Patrón Cadena de Responsabilidad?	16
Ejemplo de Patrón Cadena de Responsabilidad.....	17
Patrón Proxy	18
Propósito y Solución.....	20
Ventajas y Desventajas.....	21
¿Cómo implementamos el Patrón Proxy?	21
Ejemplo de Patrón Proxy	22
Patrón Flyweight	23
Propósito y Solución.....	24
Ventajas y Desventajas.....	24
¿Cómo implementar el Patrón Flyweight?	24
Ejemplo de Patrón Flyweight	25
Módulo 2: Testeo unitario, logging y acceso a datos.....	26
Pruebas Unitarias JUnit	26
Introducción a Test Unitarios y Test de Integración	29
Test de integración.....	29
Test Unitarios	29
¿Qué es JUnit?	30
¿Cómo configuramos la librería?	31
Testeo Parametrizado y Test Suite	32
Testeo parametrizado	32

Test Suite	33
Logueo	34
Logging	35
Configurar el appender	36
Ejemplo.....	37
Log4j	37
Arquitectura Log4j.....	38
Niveles de registro.....	39
JDBC.....	41
Uso y funcionamiento de un driver JDBC.....	42
Detalle	43
H2	45
Características	45
Conectándonos a H2 desde Java	45
Consultas y transacciones sobre base de datos.....	48
Utilizar JDBC Prepared Statement.....	48
Modificar datos con PreparedStatement.....	49
Transacciones	50
Pasos para invocar Stored Procedures.....	51
¿Cómo loguearnos con log4J en una base de datos?.....	51
Patrón DAO.....	53
Propósito y solución	53
¿Cómo implementamos el patrón DAO?	54
Maven.....	60

Módulo 1: Patrones de Diseño

Patrón Facade

Los patrones de diseño son técnicas para resolver problemas comunes en el desarrollo de software. Hay tres tipos: los **creacionales**, los **estructurales** y los de **comportamiento**. El **patrón Facade** o fachada es un tipo de **patrón estructural**.

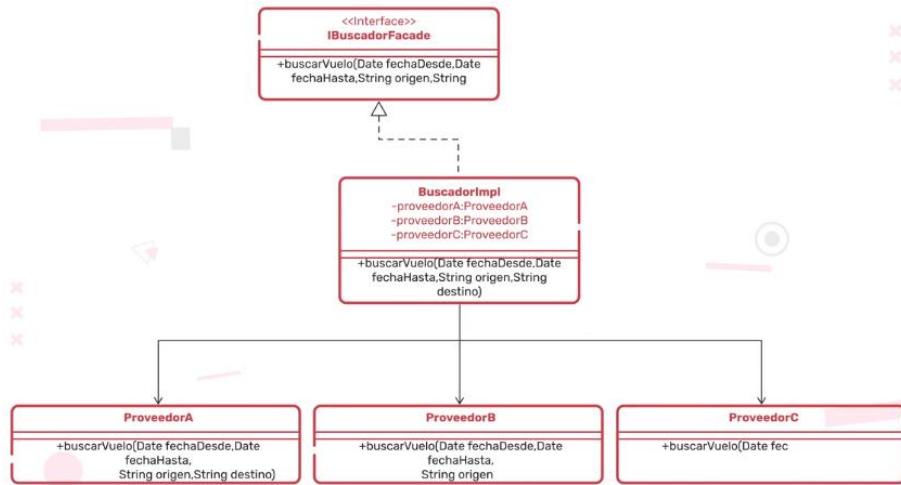
Si pensamos en un portal de turismo, en el momento que ingresamos a una búsqueda, la web se comunica con diferentes agencias y nos presentan las ofertas y las posibilidades existentes. Eso quiere decir que nos proporciona **un único punto** de entrada para comunicarnos con varias agencias al mismo tiempo unificando la información, sin que tengamos que preocuparnos por cada uno de los sitios haciendo la misma búsqueda.

¿CÓMO TRABAJA? Pensemos que tenemos tres subsistemas que son proveedores de información, cada uno sabe comunicarse con una aerolínea diferente y, como clientes, queremos averiguar los precios para volar a Colombia en una fecha determinada. Entonces tenemos que solicitarle a cada uno de esos módulos que nos informe el valor de vuelo para Colombia en esa fecha, es decir, tendríamos que hacer tres consultas y saber cómo realizarlas. Cuando utilizamos el patrón Facade podríamos tener una **clase facade** que recibe la petición del cliente y se comunica con los módulos que saben comunicarse con las aerolíneas. Esto hace mucho más fácil la comunicación y **oculta la complejidad** de interactuar con los diferentes módulos.

¿CÓMO RESOLVER EL PROBLEMA DE LOS VUELOS? Aplicando el patrón implementamos una interfaz para comunicar al cliente con el sistema. Definiríamos la clase fachada que será la encargada de comunicarse con los subsistemas, en este caso las aerolíneas. Como vemos los subsistemas son atributos de la clase de BuscadorImpl o sea nuestro Facade. Así queda nuestro ejemplo implementando el patrón:

```

1  public class BuscadorImpl implements IBuscadorFacade {
2      private ProveedorA proveedorA;
3      private ProveedorB proveedorB;
4      private ProveedorC proveedorC;
5
6      public BuscadorImpl(ProveedorA proveedorA, ProveedorB
7          proveedorB, ProveedorC proveedorC) {
8          this.proveedorA = proveedorA;
9          this.proveedorB = proveedorB;
10         this.proveedorC = proveedorC;
11     }
12     // Vamos a traer los vuelos desde los diferentes
13     proveedores, los agrupamos y los devolvemos
14     @Override
15     public List<Vuelo> buscarVuelo(Date fechaDesde, Date
16     fechaHasta, String origen, String destino) {
17         List<Vuelo> vuelos = new ArrayList();
18
19
20         vuelos.addAll(proveedorA.buscar(fechaDesde,fechaHasta,origen,d
21         estino));
22
23         vuelos.addAll(proveedorB.buscar(fechaDesde,fechaHasta,origen,d
24         estino));
25
26         vuelos.addAll(proveedorC.buscar(fechaDesde,fechaHasta,origen,d
27         estino));
28
29         return vuelos;
30     }
31 }
```



Como intermediario el objeto Facade nos garantiza que el acceso y la comunicación con el resto los subsistemas se simplifican y se reduce al mínimo la dependencia entre el cliente y esos sistemas. Esto nos da algunas **ventas**:

- El software se vuelve más flexible y fácil de expandir
- Reducimos el uso de objetos que tratan directamente con el cliente
- Traducimos el acoplamiento entre cliente y los subsistemas, lo que nos permite modificar los subsistemas sin afectar al cliente.

Su **desventaja** es el alto grado de dependencia de la interfaz de la fachada.

En conclusión, el patrón Facade nos ayuda a reducir la complejidad de interactuar con un conjunto de subsistemas actuando de intermediario entre el cliente y los subsistemas, permitiendo tener una única entrada facilitando la comunicación entre estos.

Propósito y Solución

PROPOSITO: Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

SOLUCIÓN: Se dispone de una interfaz que define como el cliente se va a comunicar con el sistema. Esta clase implementará esta interfaz para poder recibir las peticiones y será la encargada de enviarle la petición del cliente a la clase que corresponda (subsistemas).

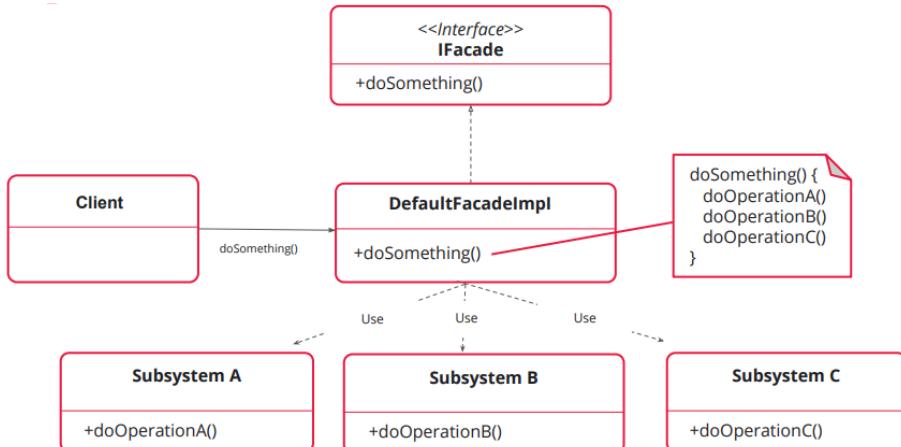


ILUSTRACIÓN 1: DIAGRAMA UML - PATRÓN FACADE

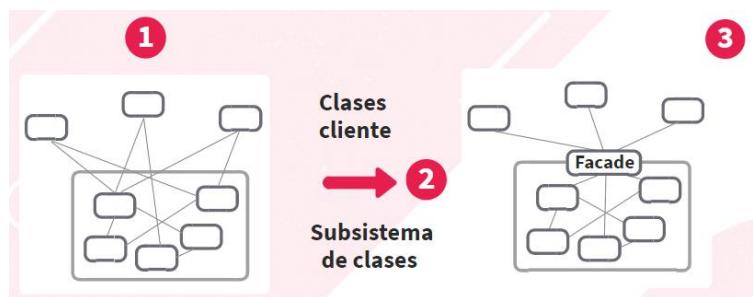
Ventajas y Desventajas

Ventajas	Desventajas
El software se vuelve más flexible y fácilmente de expandir.	Alto grado de dependencia en la interfaz de la fachada.
Reducimos el uso de objetos que tratan directamente con el cliente, haciendo que el sistema sea más fácil de usar.	
Reducimos el acoplamiento entre el cliente y los subsistemas. Esto nos permite modificar los subsistemas sin afectar al cliente.	

TABLA 1: VENTAJAS Y DESVENTAJAS DEL PATRÓN FACADE

¿Cómo implementar el Patrón Facade?

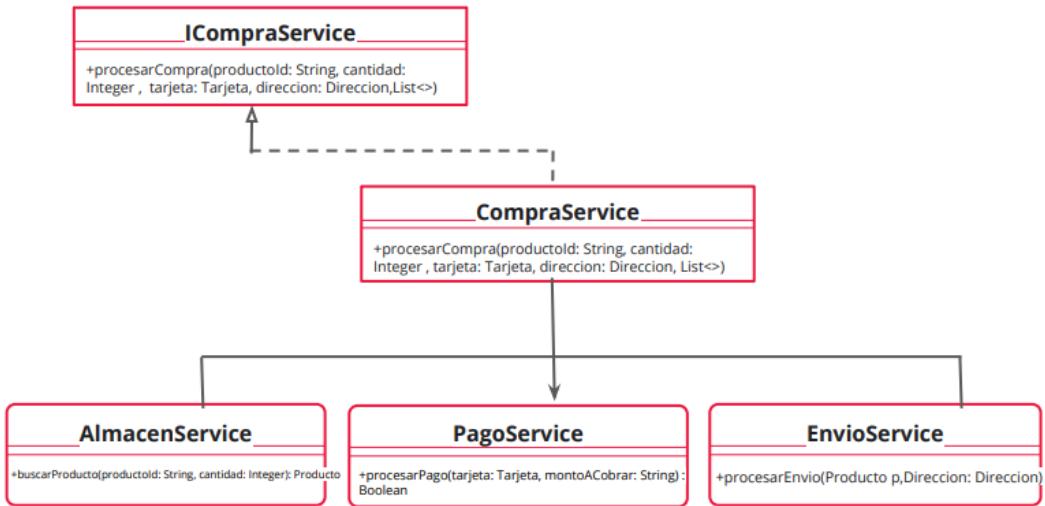
1. Comprueba si es posible proporcionar una clase más simple que la que está proporcionando un subsistema existente. Estamos bien encaminados si esta interfaz hace que el código cliente sea independiente de muchas clases del subsistema
2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. Para aprovechar el patrón máximo, hagamos que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora, el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, solo tendremos que modificar el código de la fachada.
3. Si la fachada se vuelve demasiado grande, pensemos en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.



El patrón facade nos ayuda a reducir la complejidad de interactuar con un conjunto de subsistemas, actuando de intermediario entre el cliente y los subsistemas, permitiéndonos tener una única entrada, facilitando la comunicación entre estos.

Ejemplo de Patrón Facade

Supongamos que tenemos que diseñar un sistema para un e-commerce. Nuestro cliente nos pide que, al momento de efectuar la venta del producto, nuestro sistema debería realizar una serie de pasos, por ejemplo: pedir el producto al almacén, acreditar el pago y enviar el pedido. Veamos cómo podemos resolver este problema aplicando este patrón.



Implementación de las clases del diagrama UML

```

public interface ICompraService {

    public void procesarCompra(String productoId,
        Integer cantidad, Tarjeta tarjeta,
        Direccion direccion, List<Producto> productos);
}

public class CompraService implements ICompraService {

    private AlmacenService almacenService;
    private PagoService pagoService;
    private EnvioService envioService;

    public CompraService() {

        this.almacenService = new AlmacenService();
        this.pagoService = new PagoService();
        this.envioService = new EnvioService();

    }

    public void procesarCompra(String productoId,
        Integer cantidad, Tarjeta tarjeta,
        Direccion direccion, List<Producto>
        productos) {

        Producto prod;
        almacenService.setProductos(productos);
        prod = almacenService.buscarProducto(productoId,cantidad);
        if(prod != null){
            double montoCobrar = producto.getValor() * cantidad;
            if(pagoService.procesarPago(tarjeta,montoCobrar)){
                envioService.procesarEnvio(producto,direccion);
            }
        }
    }
}

```

Código de la interface **ICompraService**

Código de la clase **CompraService** (nuestro facade)

Código de la clase **CompraService** (nuestro facade)

```
public class AlmacenService {  
    private List<Producto> productos;  
  
    public void setProductos(List<Producto> productos) {  
        this.productos = productos;  
    }  
}
```

Código de la clase
AlmacenService
(subsistema)

```
public Producto buscarProducto(String productoId,  
                               Integer cantidad) {  
  
    Producto producto = null;  
  
    for (Producto p : this.productos) {  
        if (p.getProductoId().equals(productoId) &&  
            p.getCantidad() >= cantidad){  
  
            producto = p;  
            p.setCantidad(p.getCantidad() - cantidad);  
            producto.setCantidad(cantidad);  
        }  
    }  
    return producto;  
}
```

Código de la clase
AlmacenService
(subsistema)

```
Public class PagoService {  
  
    public Boolean procesarPago(Tarjeta tarjeta,double montoACobrar){  
        Boolean pagoRealizado = Boolean.FALSE;  
        if(tarjeta != null && tarjeta.getNumerosFrente() != null &&  
            tarjeta.getCodSeguridad() != null)  
  
            System.out.println("Procesando el pago por "+  
                montoACobrar);  
            pagoRealizado = TRUE;  
  
        return pagoRealizado;  
    }  
}
```

Código de la clase
PagoService
(subsistema)

```
public class EnvioService {  
  
    public void procesarEnvio(Producto producto,  
                             Direccion direccion){  
  
        System.out.println("Enviando producto a " +  
            direccion.getCalle() + " " + direccion.getNro() + "," +  
            direccion.getBarrio());  
    }  
}
```

Código de la clase
EnvioService
(subsistema)

Simulamos un caso en el método main para ponerlo a prueba.

```

public class Prueba {
    public static void main(String[] args) {
        List<Producto> productos = new ArrayList<>();
        Producto productoUno = new Producto("1",5,1000,"Mouse");
        Producto productoDos = new Producto("2",5,3000,"Teclado");
        productos.add(productoUno);
        productos.add(productoDos);
        Tarjeta tarjeta = new Tarjeta("1111222233334444","012","2025/07/09");
        Direccion direccion = new Direccion("Av Monroe", "860", "1428", "CABA", "Capital federal");

        ICompraService compraService = new CompraService();
        compraService.procesarCompra("1",2,tarjeta,direccion, productos);
    }
}

```

Conclusión

Mediante una interface definimos cómo el cliente se deberá comunicar con nuestro sistema, definimos la clase CompraService que actuará de fachada, recibiendo las peticiones y comunicándose con los subsistemas para que, en conjunto, se complete la compra.

Clase 2: Patrón Template Method

Patrón Template Method

Imaginemos que en nuestro sistema varias clases hacen lo mismo, pero tienen algunos detalles diferentes, es muy probable que entre estas clases tengamos código repetido.

Existen tres tipos de patrones de diseño que son técnicas para resolver problemas comunes en el desarrollo de software. El **Patrón Template Method** forma parte de los patrones denominados de **comportamiento**, es decir, que nos va ayudar a resolver problemas de interacción entre las clases y objetos.

El patrón define un método esqueleto con el código que estas clases tienen en común, permitiendo que, algunas partes puedan ser modificadas por la subclase que implemente este método esqueleto logrando ubicar en un solo lugar el código repetido. La solución que propone el Patrón Template Method es abstraer todo el comportamiento que comparten las entidades en una clase abstracta de la que, posteriormente, estas entidades extenderán. Esta clase abstracta definirá el esqueleto del método y delegará algunas responsabilidades a las clases hijas mediante métodos abstractos que deberán implementar.

Si pensamos en un sitio web que nos permite iniciar sesión con la cuenta de Facebook, Twitter o Google. Aplicando el patrón definiremos una clase abstracta, un algoritmo que nos permite Iniciar sesión dejando algunos detalles particulares, para que las subclases implementen este método y lo adapte para comunicarse con Facebook, Google o Twitter.

```

1  public abstract class Login {
2
3      public Boolean procesarUsuario(String usuario, String
4      contrasena) {
5          Boolean usuarioValido = Boolean.FALSE;
6          if (usuario != null & contrasena != null)
7              usuarioValido = iniciarSesion(usuario,
8              contrasena);
9
10         return usuarioValido;
11     }
12     protected abstract Boolean iniciarSesion(String
13     usuario, String contrasena);
14 }
15

```

Twitter

Facebook

Google

```

1  public class TwitterLogin extends Login {
2      @Override
3      protected Boolean iniciarSesion(String
4      usuario, String contrasena) {
5
6          System.out.println("Validando
7          usuario y contraseña con Twitter");
7
8          return
9          validarUsuarioYContraseña(usuario,
10          contrasena);
11      }
12
13
14
15
16
17

```

```

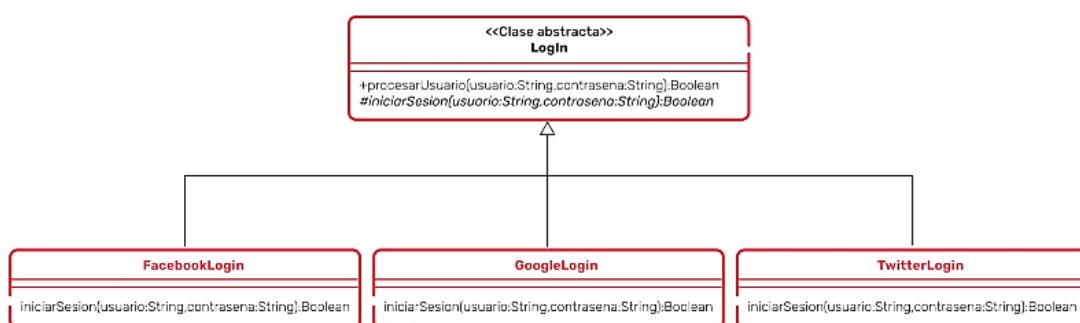
1  public class TwitterLogin extends Login {
2      @Override
3      protected Boolean iniciarSesion(String
4      usuario, String contrasena) {
5
6          System.out.println("Validando
7          usuario y contraseña con Twitter");
7
8          return
9          validarUsuarioYContraseña(usuario,
10          contrasena);
11      }
12
13
14
15
16
17

```

```

1  public class TwitterLogin extends Login {
2      @Override
3      protected Boolean iniciarSesion(String
4      usuario, String contrasena) {
5
6          System.out.println("Validando
7          usuario y contraseña con Twitter");
7
8          return
9          validarUsuarioYContraseña(usuario,
10          contrasena);
11      }
12
13
14
15
16
17

```

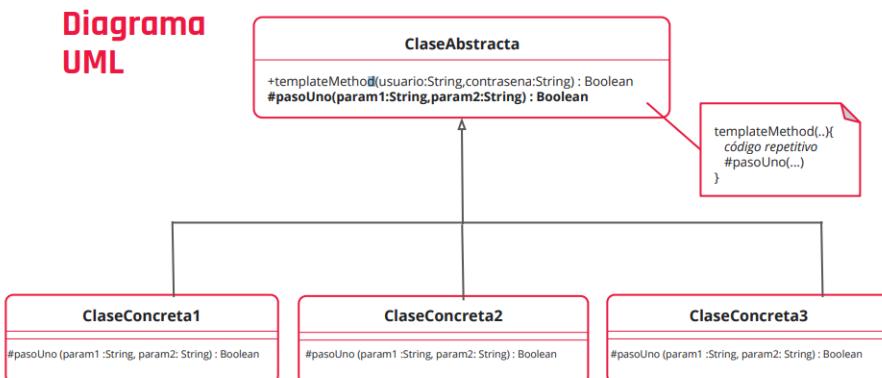


El Patrón Template Method es utilizado por diferentes APIs de Java y también por frameworks como Spring. Al eliminar el código repetido nuestro código será más eficiente, legible y mantenible permitiendo que sea más fácil de extender y de mejorar.

Propósito y Solución

PROPOSITO: Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

SOLUCIÓN: El método esqueleto está conformado por el código que estas clases tienen en común, permitiendo que algunas partes puedan ser modificadas por la subclase que implemente el mismo, logrando ubicar en un solo lugar el código repetido.



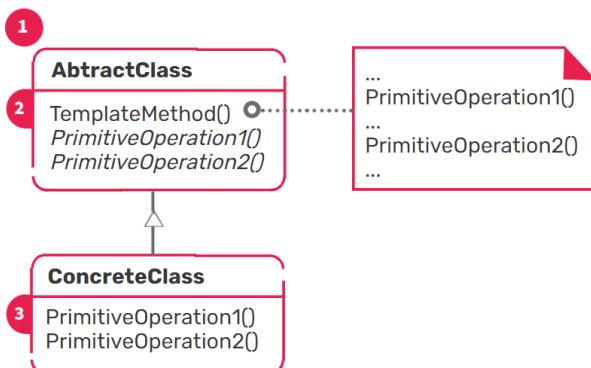
Ventajas y Desventajas

Ventajas	Desventajas
Los clientes pueden sobreescibir ciertas partes de un algoritmo grande para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.	Posee un alto grado de dependencia en la interfaz de la fachada.
Se puede colocar el código duplicado dentro de una superclase.	Algunos clientes pueden verse limitados por el esqueleto proporcionado por el algoritmo.

TABLA 2: VENTAJAS Y DESVENTAJAS DEL PATRÓN TEMPLATE METHOD

¿Cómo implementar el Patrón Template Method?

1. Consideremos qué pasos son comunes a todas las subclases en las que vemos código repetido y cuáles siempre serán únicos.
2. Creamos la clase base abstracta y declaremos el método esqueleto y un grupo de métodos abstractos que representan los pasos del algoritmo a implementar por las subclases.
3. Para cada variación del método, creamos una nueva subclase concreta. Esta debe implementar todos los pasos abstractos.



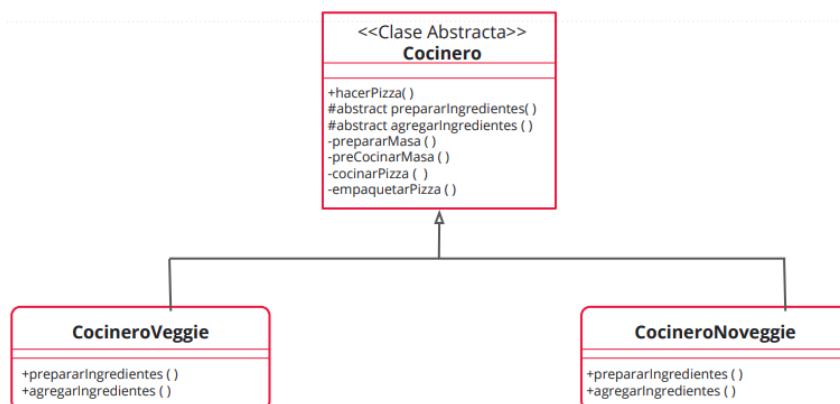
Entonces, ¿para qué nos sirve el patrón template method? **Porque al eliminar el código repetido, nuestro código será más eficiente, legible y mantenible.** Esto hará que sea más fácil de extender y mejorar.

Ejemplo de Template Method

Veamos un ejemplo del patrón haciendo una analogía con un ejemplo de la vida real. Pensemos en una pizzería y en el proceso de preparar diferentes tipos de pizzas. Intentemos identificar los pasos que un cocinero debe realizar para cocinar y entregar una pizza. Podrían ser:



Por cada variedad de pizza, los cocineros tienen que hacer todos esos pasos. Aplicando el patrón template method, podríamos crear el método esqueleto, con los pasos comunes a todas las pizzas y dejar que los cocineros solo se preocupen por los pasos que no son comunes a todas las pizzas, en este caso, preparar los ingredientes y agregar los ingredientes. Veamos cómo representamos esta solución en un diagrama UML.



Implementación de las clases del diagrama UML

```
public abstract class Cocinero {

    public void hacerPizza(){
        prepararMasa();
        preCocinarMasa();
        prepararIngredientes();
        agregarIngredientes();
        cocinarPizza();
        empaquetarPizza();
    }

}
```

Código de la clase abstracta que define el método esqueleto y los métodos abstractos a implementar por las subclases.

```
protected abstract void prepararIngredientes();
protected abstract void agregarIngredientes();
private void prepararMasa(){
    System.out.println("Preparando masa..");
}
private void preCocinarMasa(){
    System.out.println("Pre cocinando masa..");
}
private void cocinarPizza(){
    System.out.println("Enviando al horno la pizza");
}
private void empaquetarPizza(){
    System.out.println("Empaquetando pizza");
}
```

Código de la clase abstracta que define el método esqueleto y los métodos abstractos a implementar por las subclases.

```

public class CocineroNoVeggie extends Cocinero {
    @Override
    protected void prepararIngredientes() {
        System.out.println("Preparando queso y jamón,");
    }

    @Override
    protected void agregarIngredientes() {
        System.out.println("Agregando los ingredientes");
    }
}

```

Código de la
subclase
CocineroNoVeggie

```

public class CocineroVeggie extends Cocinero {

    @Override
    protected void prepararIngredientes() {
        System.out.println("Preparando tomate y quesos");
    }

    @Override
    protected void agregarIngredientes() {
        System.out.println("Agregando quesos y tomate");
    }
}

```

Código de la
subclase
CocineroVeggie
(esta clase
validará los datos
en Facebook)

```

public class Main {

    public static void main(String[] args) {
        Cocinero cocineroVeggie = new CocineroVeggie();
        Cocinero cocineroNoVeggie = new CocineroNoVeggie();

        cocineroVeggie.hacerPizza();
        cocineroNoVeggie.hacerPizza();
    }
}

```

Test

Clase 4: Patrón Cadena de Responsabilidad

Patrón Cadena de Responsabilidad

Imaginemos un equipo de desarrollo de Software que está formado por una persona que desarrolla BackEnd, otra que desarrolla FrontEnd y una persona de infraestructura. Por otro lado, tenemos un cliente que no sabe cuáles son las funciones de cada uno y le solicita a la persona de infraestructura un cambio de color en una pantalla. Esta persona analiza la petición del cliente y, en caso de poder resolverla, lo hace, pero sino la deriva a otro compañero hasta que alguien del equipo pueda hacerlo. Así es cómo funciona el patrón cadena de responsabilidad.

El **Patrón Cadena de Responsabilidad** es de **comportamiento**. Este patrón nos permite establecer una cadena entre objetos a través de la cual se pasa una petición formulada por un objeto emisor que no

conoce cuál es el objeto que podrá responderle. De esta manera, los objetos receptores pasará la petición hasta que alguno pueda brindar una solución.

Este patrón está compuesto por una **clase manejador** que recibirá las peticiones del cliente y por las clases que reasignarán las peticiones que envía el manejador e intentarán procesarla.

Pensemos en este ejemplo, estamos desarrollado una aplicación y queremos escribir y enviar el flujo por el que va pasando la información desde que ingresan los datos, mientras procesamos, y luego le respondemos al cliente. También debemos definir, mediante una escala de importancia, si vamos a escribir por consola en un archivo de texto o enviar un email. Veamos cómo resolvemos este problema aplicando el patrón.

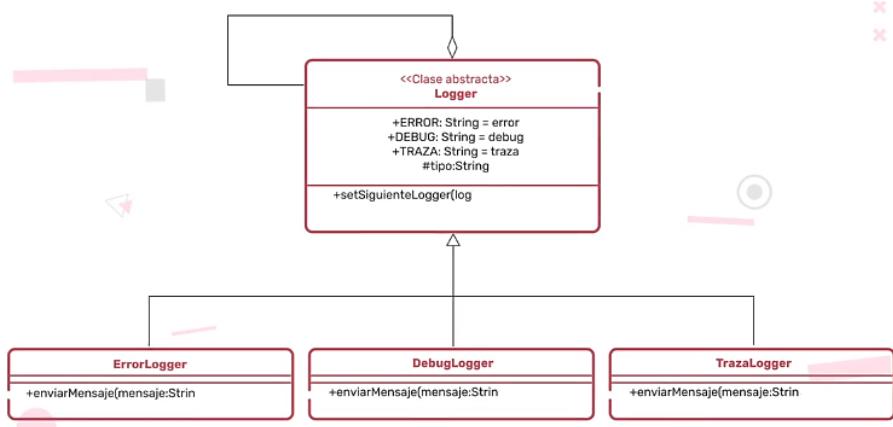
La clase manejadora es responsable de comenzar la cadena y definir como el cliente se comunicará con ella. Las subclases que participarán de la cadena procesan la petición del cliente de acuerdo a un criterio preestablecido. Todas las clases que extienden del Logger saben cómo enviar un mensaje pero cada uno lo hará de acuerdo a un criterio y forma diferente y continuarán la cadena pasando por el resto de las subclases.

```
1  public abstract class Logger {  
2  
3      public static String ERROR = "error";  
4      public static String DEBUG = "debug";  
5      public static String TRAZA = "traza";  
6      protected Logger siguienteLogger;  
7      protected String tipo;  
8  
9      public Logger setSiguienteLogger(Logger  
10 logger){  
11         this.siguienteLogger = logger;  
12         return this;  
13     }  
14     public abstract void  
15     enviarMensaje(String mensaje, String tipo);  
16  
17 }
```

ILUSTRACIÓN 2: CLASE MANEJADORA

```
1  public class DebugLogger extends Logger {  
2      public DebugLogger(String tipo) {  
3          this.tipo = tipo;  
4      }  
5  
6      @Override  
7      public void enviarMensaje(String  
8      mensaje, String tipo) {  
9          if(tipo.equals(this.tipo))  
10             System.out.println("Escribiendo  
11 en DEBUG : "+mensaje);  
12             if(this.siguienteLogger != null)  
13                 this.siguienteLogger.enviarMensaje(mensaje,  
14 tipo);  
15             }  
16         }  
17     }  
18 }  
  
1  public class ErrorLogger extends Logger {  
2  
3      public ErrorLogger(String tipo) {  
4          this.tipo = tipo;  
5      }  
6  
7      @Override  
8      public void enviarMensaje(String  
9      mensaje, String tipo) {  
10         if(tipo.equals(this.tipo))  
11             System.out.println("Escribiendo  
12 en un archivo de texto : "+mensaje);  
13             if(this.siguienteLogger != null)  
14                 this.siguienteLogger.enviarMensaje(mensaje,  
15 tipo);  
16             }  
17         }  
18     }  
19 }  
  
1  public class TrazalLogger extends Logger {  
2      public TrazalLogger(String tipo) {  
3          this.tipo = tipo;  
4      }  
5  
6      @Override  
7      public void enviarMensaje(String  
8      mensaje, String tipo) {  
9          if(tipo.equals(this.tipo))  
10             System.out.println("Escribiendo  
11 en un archivo de texto : "+mensaje);  
12             if(this.siguienteLogger != null)  
13                 this.siguienteLogger.enviarMensaje(mensaje,  
14 tipo);  
15             }  
16         }  
17     }  
18 }  
19 }
```

ILUSTRACIÓN 3: SUBCLASES



Las **ventajas** de utilizar este patrón son que nos brinda mayor flexibilidad a la hora de procesar las peticiones del cliente, podemos agregar objetos que sepan resolver nuevas responsabilidades o modificar las actuales sin afectar al cliente. También nos da menor acoplamiento ya que el patrón permite que un objeto y una petición sepan que va a ser tratada, pero tanto el emisor como el receptor no conocen nada del otro. Una **desventaja** es que puede ser complejo implementar la cadena y si no está bien configurada puede que no se cubran todas las peticiones, es decir, la recepción no está asegurada.

Es beneficioso utilizar este patrón cuando esperamos que nuestra aplicación procese diferentes tipos de solicitudes y responda con diferentes resultados, pero no conocemos de antemano cuáles serán estas solicitudes.

Propósito y Solución

PROPÓSITO: Es un patrón de diseño de comportamiento que permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada uno de ellos, decide si la procesa o la pasa al siguiente manejador.

SOLUCIÓN: Crear una cadena con las clases manejadores para que procesen la solicitud del cliente. Cada uno tiene un campo para almacenar una referencia al siguiente manejador de la cadena. La solicitud viaja por la misma hasta que todos los manejadores hayan tenido la oportunidad de procesarla (los manejadores pueden decidir no pasar la solicitud y detener el procedimiento).

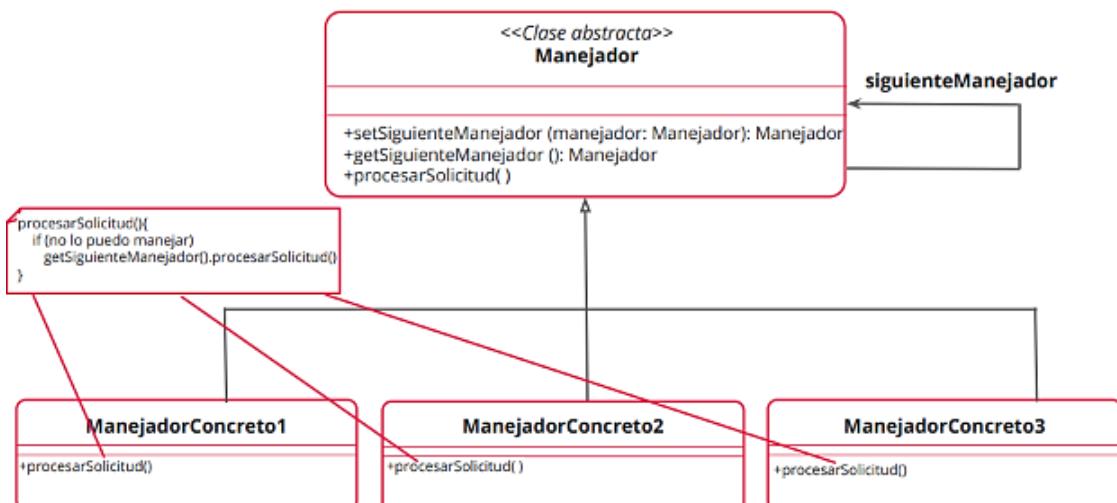


ILUSTRACIÓN 4: DIAGRAMA UML - PATRÓN CADENA DE RESPONSABILIDAD

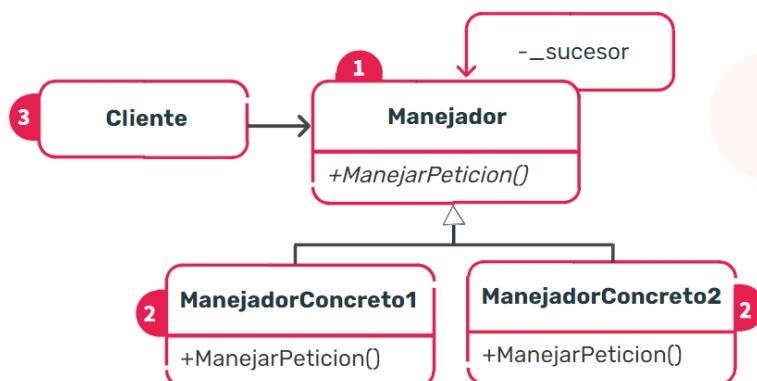
Ventajas y Desventajas

Ventajas	Desventajas
Mayor flexibilidad para procesar las peticiones del cliente: Es posible agregar objetos que sepan resolver nuevas responsabilidades o modificar las actuales sin afectar al cliente.	Puede ser complejo implementar la cadena, y si no está bien configurada, puede que no se cubran todas las peticiones.
Menor acoplamiento: Permite que un objeto envíe una petición y sepa que va a ser tratada, pero tanto el emisor como el receptor no conocen nada del otro.	

TABLA 3: VENTAJAS Y DESVENTAJAS DEL PATRÓN CADENA DE RESPONSABILIDAD

¿Cómo implementamos el Patrón Cadena de Responsabilidad?

1. Crearemos una clase abstracta que será la encargada de almacenar una referencia al siguiente manejador de la cadena y un método para enviar la solicitud del cliente al elemento siguiente de la cadena. Los manejadores concretos podrán utilizar este comportamiento invocando al método padre.
2. Una a una, creemos subclases manejadoras concretas e implementemos los métodos. Cada manejador debe tomar dos decisiones cuando recibe una solicitud:
 - Si procesa la solicitud.
 - Si pasa la solicitud al siguiente eslabón de la cadena.
3. El cliente puede activar cualquier manejador de la cadena, no solo el primero. La solicitud se pasará a lo largo de la cadena hasta que algún manejador se rehúse a pasarlo o hasta que llegue al final de la cadena. Debido a la naturaleza dinámica de la cadena, el cliente debe estar listo para gestionar los siguientes escenarios:
 - La cadena puede consistir en un único vínculo.
 - Algunas solicitudes pueden no llegar al final de la cadena.
 - Otras pueden llegar hasta el final de la cadena sin ser gestionadas.



Entonces, podemos concluir que es beneficioso utilizar el patrón cadena de responsabilidad cuando esperamos que nuestra aplicación procese diferentes tipos de solicitudes y responda con diferentes resultados, pero no conocemos de antemano cuáles son estas solicitudes.

Ejemplo de Patrón Cadena de Responsabilidad

Imaginemos que estamos desarrollando un sistema para el área de créditos en un banco y queremos que al momento de que un cliente solicite un crédito se envíe un pedido a los diferentes encargados de autorizarlo. El banco nos indicó que:

- Si el monto no supera los 60.000, entonces, el ejecutivo de cuenta puede aprobarlo.
- Si el monto está entre los 60.000 y 200.000, entonces, el gerente es quien lo aprueba.
- Si el monto se encuentra por encima de 200.000 lo aprueba el director.

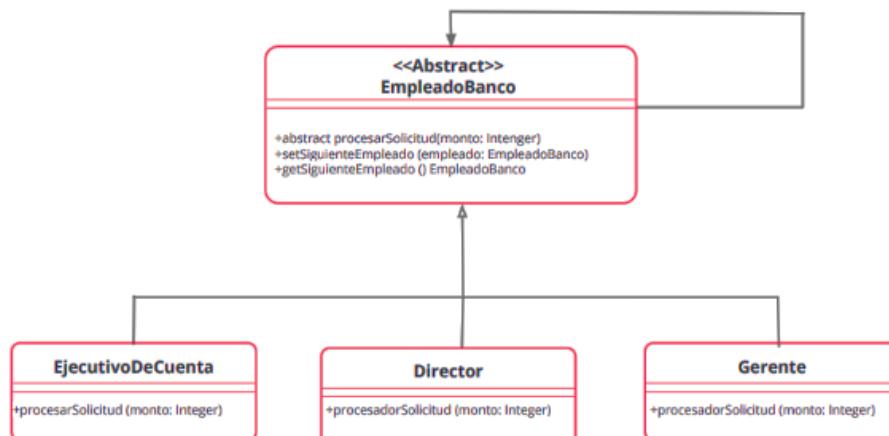


ILUSTRACIÓN 5: DIAGRAMA UML

Implementación de las clases del diagrama UML

```
public abstract class EmpleadoBanco {

    private EmpleadoBanco sigEmpleadoBanco;
    public abstract void procesarSolicitud(Integer monto);

    public void setSigEmpleadoBanco(EmpleadoBanco emp) {
        sigEmpleadoBanco = emp;
    }
    public EmpleadoBanco getSigEmpleadoBanco() {
        return sigEmpleadoBanco;
    }
}
```

La clase **manejadora** es la responsable de comenzar la cadena con ella, en este caso, **EmpleadoBanco**.

```
public class Director extends EmpleadoBanco {
    @Override
    public void procesarSolicitud(Integer monto) {
        if (monto > 200000)
            System.out.println("Yo me encargo de gestionarlo.
                Director");
        else if (getSigEmpleadoBanco() != null)
            getSigEmpleadoBanco().procesarSolicitud(monto);
    }
}
```

Subclases en las que definiremos bajo qué criterio y cómo procesarán la solicitud.

```

public class EjecutivoCuenta extends EmpleadoBanco {
    @Override
    public void procesarSolicitud(Integer monto) {
        if (monto < 60000)
            System.out.println("Yo me encargo de gestionarlo.
                Ejecutivo de cuenta");
        else if (getSigEmpleadoBanco() != null)
            getSigEmpleadoBanco().procesarSolicitud(monto);
    }
}

```

Subclases en las que definiremos bajo qué criterio y cómo procesarán la solicitud.

```

public class Gerente extends EmpleadoBanco {

    @Override
    public void procesarSolicitud(Integer monto) {
        if (monto >= 60000 && monto <= 200000)
            System.out.println("Yo me encargo de gestionarlo.
                Gerente");
        else if (getSigEmpleadoBanco() != null)
            getSigEmpleadoBanco().procesarSolicitud(monto);

    }
}

```

Subclases en las que definiremos bajo qué criterio y cómo procesarán la solicitud.

```

public static void main(String[] args) {

    EmpleadoBanco empleado1 = new EjecutivoCuenta();
    EmpleadoBanco empleado2 = new Gerente();
    EmpleadoBanco empleado3 = new Director();

    empleado2.setSigEmpleadoBanco(empleado3);
    empleado1.setSigEmpleadoBanco(empleado2);

    empleado1.procesarSolicitud(78000);

}

```

Test

Clase 5: Patrón Proxy

Patrón Proxy

¿QUÉ ES UN PROXY? Un servidor Proxy es un ordenador que puede conectarse como interfaz entre dos ordenadores o redes, es decir que asume la función de intermediario recibiendo las peticiones y transmisiéndolas con su propia dirección IP a otra red.

El **Patrón Proxy** es del tipo **estructural**, este patrón cumple la función de ser un intermediario que agrega funcionalidad a una clase sin tocar la misma. La definición del patrón Proxy se pone en práctica a través del concepto **Proxy Object**, podemos decir que este es un objeto que agrega lógica adicional por encima del objeto al que se desea llegar sin necesidad de modificar ese objeto determinado.

¿CÓMO ESTÁ COMPUESTO EL PATRÓN? Imaginemos que estamos desarrollando una aplicación para descargar videos de YouTube. Recibimos una petición de video, la buscamos en YouTube y se la enviamos al usuario, este proceso será altamente costoso si recibimos muchas peticiones en simultáneo. Acá es donde aparece el patrón.

Primero definimos una interfaz, tenemos la clase que busca los videos en YouTube que implementa la interfaz y tenemos la clase Proxy que también implementa la interfaz y utiliza la clase que sabe comunicarse con YouTube. Nuestro proxy realiza lo mismo que el proveedor de videos, pero le agregamos lógica extra, en este caso, siempre vamos a buscar los videos en nuestra memoria caché y si no tenemos entonces le pedimos al proveedor de los mismos.

```
1 public interface IProveedorDeVideo {  
2  
3     Map<String, Video> popularVideos();  
4  
5     Video getVideo(String videoId);  
6 }  
7
```

ILUSTRACIÓN 6: INTERFAZ

```
1 public class ProveedorVideoYoutube implements IProveedorDeVideo {  
2     @Override  
3     public Map<String, Video> popularVideos() {  
4         return getRandomVideos();  
5     }  
6  
7     @Override  
8     public Video getVideo(String videoId) {  
9         return new Video(videoId, "video_ejemplo.mp4");  
10    }  
11    private HashMap<String, Video> getRandomVideos() {  
12        System.out.print("Descargando... ");  
13        Map<String, Video> hmap = new HashMap<String, Video>();  
14        hmap.put("1", new Video("sadghasgdas", "Catzzzz.avi"));  
15        hmap.put("2", new Video("mkafiksangas", "mascotas.mp4"));  
16        hmap.put("3", new Video("asdfas3ffasd", "baile_video.mp4"));  
17        hmap.put("4", new Video("dlsdk5jfslaf", "Barcelona_vs_RealM.mov"));  
18        hmap.put("5", new Video("3sdfgsdij333", "Programing_lesson#1.avi"));  
19        System.out.print("Completo!" + "\n");  
20        return hmap;  
21    }  
22}
```

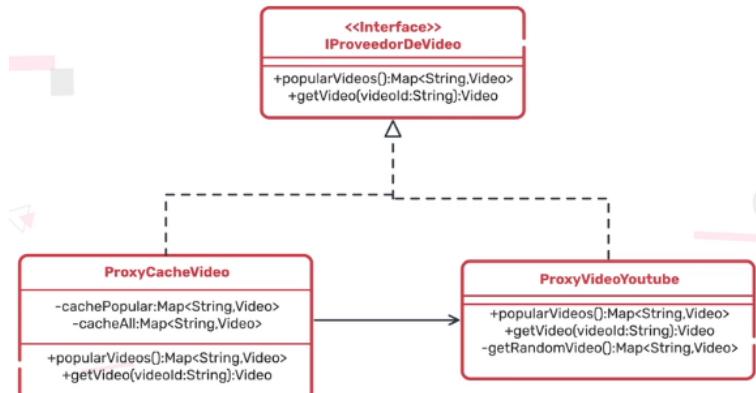
ILUSTRACIÓN 7: CLASE BUSCARVIDEOS

```

1 public class ProxyCacheVideo implements IProveedorDeVideo {
2     private IProveedorDeVideo proveedorDeVideo;
3     private Map<String, Video> cachePopular;
4     private Map<String, Video> cacheAll;
5
6     public ProxyCacheVideo(IProveedorDeVideo proveedorDeVideo) {
7         this.proveedorDeVideo = proveedorDeVideo;
8         this.cachePopular = new HashMap<>();
9         this.cacheAll = new HashMap<>();
10    }
11
12    @Override
13    public Map<String, Video> popularVideos() {
14        if(this.cacheAll.isEmpty())
15            cachePopular = proveedorDeVideo.popularVideos();
16
17        return cachePopular;
18    }
19    @Override
20    public Video getVideo(String videoId) {
21        Video video = cacheAll.get(videoId);
22        if(video == null)
23            video = proveedorDeVideo.getVideo(videoId);
24            cacheAll.put(videoId,video);
25        return video;
26    }
27}

```

ILUSTRACIÓN 8: CLASE PROXY



El patrón Proxy es de los patrones más utilizados por el framework Spring principalmente para controlar el acceso a determinados objetos y para instanciar objetos que son computacionalmente costosos.

Propósito y Solución

PROPOSITO: Tiene por objetivo desarrollar la función de ser un intermediario que agrega funcionalidad a una clase, sin tocar la misma.

SOLUCIÓN: Definir una clase Proxy con la misma interfaz que el objeto de servicio original. Después, se debe actualizar nuestra aplicación para que los clientes se comuniquen con el proxy y no con el servicio destino. Al recibir la solicitud de un cliente, el proxy le enviará al servicio, pero como intermediario podremos realizar operaciones antes o después de enviarle la solicitud.

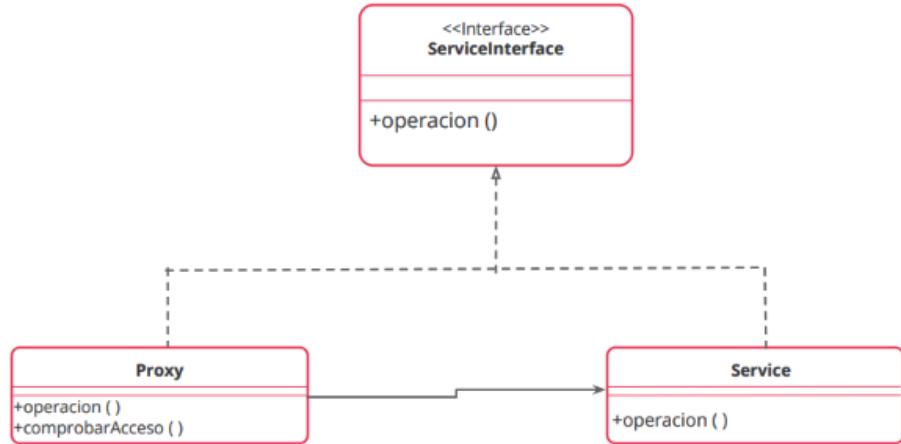


ILUSTRACIÓN 9: DIAGRAMA UML - PATRÓN PROXY

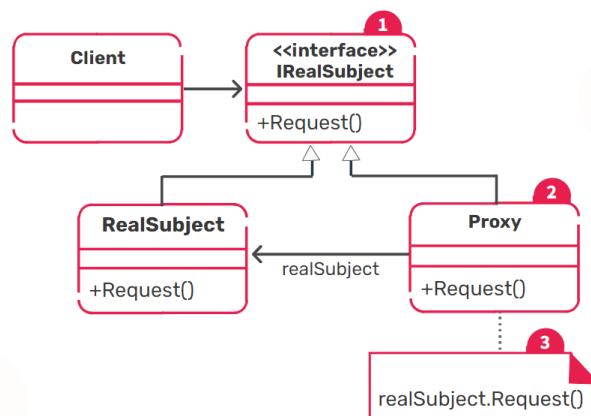
Ventajas y Desventajas

Ventajas	Desventajas
El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.	Al agregar una capa más entre el cliente y el servicio real, la respuesta puede retrasarse.
Principio de abierto/cerrado: Podemos introducir nuevos proxies sin cambiar el servicio o los clientes.	

TABLA 4: VENTAJAS Y DESVENTAJAS DEL PATRÓN PROXY

¿Cómo implementamos el Patrón Proxy?

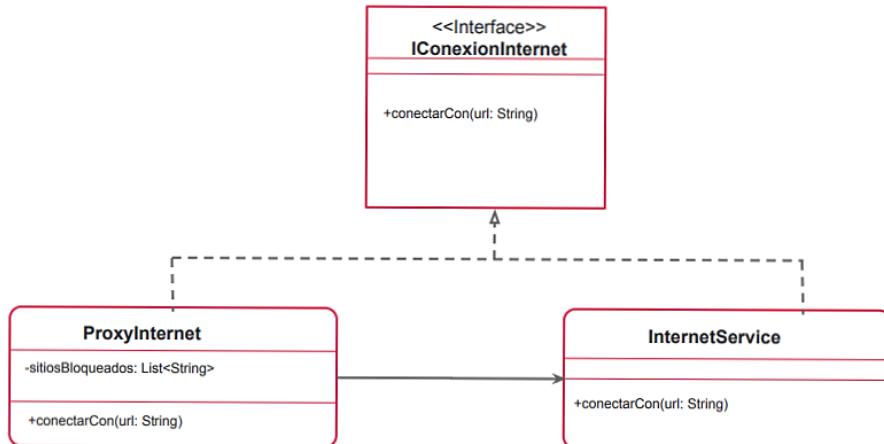
1. Si no hay una interfaz de servicio preexistente, creamos una para que los objetos de proxy y de servicio sean intercambiables.
2. Creamos la clase proxy. Debe tener un campo para almacenar una referencia al servicio.
3. Implementemos los métodos del proxy según sus propósitos. En la mayoría de los casos, después de hacer cierta labor, el proxy debería delegar el trabajo a un objeto de servicio.



Entonces, podemos mencionar que es beneficioso utilizar el patrón proxy cuando queremos agregar funcionalidad adicional sin modificar los servicios actuales.

Ejemplo de Patrón Proxy

Pensemos en un escenario de la vida real, en algunos lugares, como la universidad o el trabajo, la red que nos conecta a Internet está limitada y no tenemos acceso a todos los sitios. Puede ser que tengamos un proxy que esté restringiendo el acceso y en lugar de conectarnos directamente a Internet, estemos comunicándonos con el proxy y sea este el que finalmente se comunica con Internet.



Implementación de las clases del diagrama UML

```
public interface IConexionInternet {  
    public void conectarCon(String url);  
}
```

Definimos la interface

```
public class InternetService implements IConexionInternet {  
  
    @Override  
    public void conectarCon(String url) {  
        System.out.println("Conectando con " + url);  
    }  
}
```

Definimos el servicio que nos conecta a Internet.

```
public class ProxyInternet implements IConexionInternet {  
    private InternetService internetService;  
    private List<String> sitiosBloqueados;  
  
    public ProxyInternet(List<String> sitiosBloqueados,  
                        InternetService internetService) {  
        this.sitiosBloqueados = sitiosBloqueados;  
        this.internetService = internetService;  
    }  
}
```

El proxy realiza lo mismo que el servicio Internet, pero le agregamos lógica extra.



Clase 7: Patrón Flyweight

Patrón Flyweight

Imaginemos que tenemos en el sistema una clase con pocos atributos y que necesitamos instanciar una gran cantidad de veces, el patrón Flyweight nos va a permitir tener muchas menos instancias y además nos da un número de posibilidades para “decorar”.

De los tres tipos de patrones de diseño, el Patrón Flyweight es **tipo estructural** y les permite a los programas soportar grandes cantidades de objetos manteniendo un bajo uso de memoria ya que devuelva objetos en caché en lugar de crear nuevos. El patrón logra esto compartiendo partes del estado del objeto entre varios objetos, es decir, abstrae las partes reutilizables y, en lugar de crear objetos cada vez que sea requerido, podemos reutilizar objetos creados por otras instancias permitiendo reducir la capacidad de memoria requerida por la aplicación.

Este patrón cuenta con varios componentes:

- El cliente: Es el objeto que dispara la ejecución
- El FlyweightFactory: Es la fábrica que utilizaremos para crear los objetos flyweight u objetos ligeros.
- El Flyweight: Corresponde a los objetos que deseamos reutilizar para que los mismos sean más ligeros.

El patrón Flyweight reutiliza el objeto evitando cargar la memoria y además ayuda muchísimo en la performance, pero ¡Atención! Debemos tener cuidado de nunca usarlo cuando los objetos tienen

muchos estados porque si tenemos que modificar muchos estados en tiempo de ejecución entonces puedo ocupar mucho tiempo de procesamiento y sería difícil de mantener.

Propósito y Solución

PROPÓSITO: El patrón logra lo dicho anteriormente compartiendo partes del estado del objeto entre varios objetos. Es decir, abstrae las partes reutilizables y, en lugar de crear objetos cada vez que sea requerido, podemos reutilizar objetos creados por otras instancias. Esto permite reducir la capacidad de memoria requerida por la aplicación.

SOLUCIÓN: Este patrón cuenta con varios componentes: el cliente es el objeto que dispara la ejecución. El FlyweightFactory es la fábrica que utilizaremos para crear los objetos flyweight u objetos ligeros. El flyweight corresponde a los objetos que deseamos reutilizar para que estos sean más ligeros.

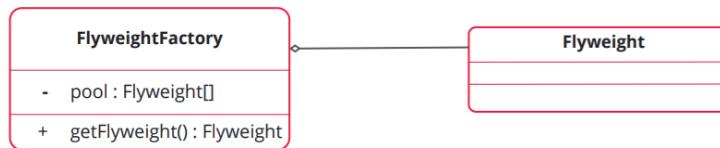


ILUSTRACIÓN 10: DIAGRAMA UML - PATRÓN FLYWEIGHT

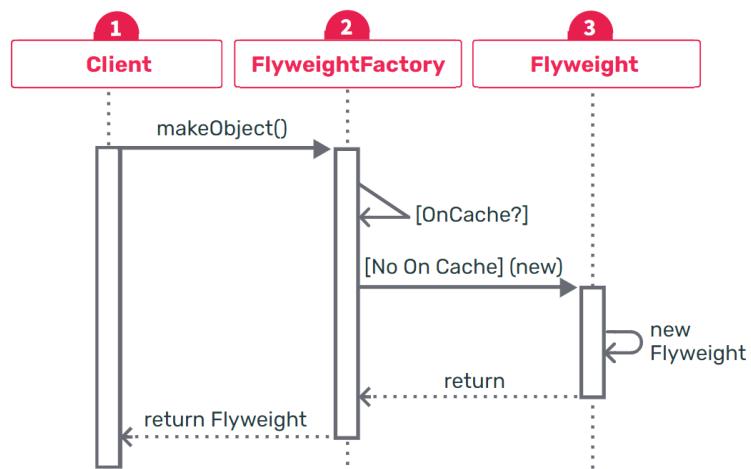
Ventajas y Desventajas

VENTAJA: Reduce una gran cantidad el peso de los datos en el servidor.

DESVENTAJA: Consumo un poco más de tiempo para realizar las búsquedas.

¿Cómo implementar el Patrón Flyweight?

1. Objeto que dispara la ejecución
2. Fábrica que utilizaremos para crear los objetos flyweight u objetos ligeros.
3. Corresponde a los objetos que deseamos reutilizar con el fin de que nuestros objetos sean más ligeros



Podemos mencionar que este patrón es utilizado cuando la **optimización de los recursos es algo primordial**, ya que elimina la redundancia de objetos con propiedades idénticas.

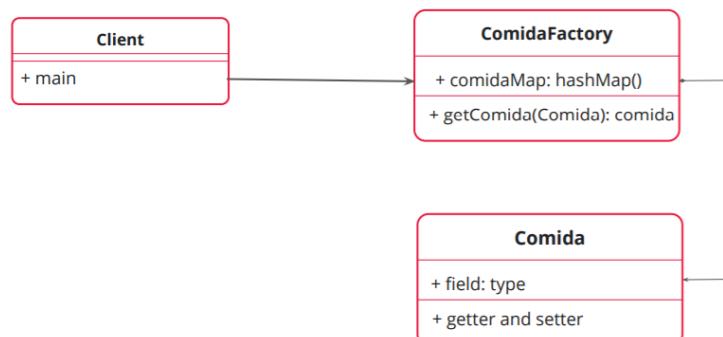
Ejemplo de Patrón Flyweight

Supongamos que tenemos un recetario que contiene recetas que están en diferentes colecciones como carne; sopas; ensaladas; o en diferentes categorías como comida italiana; mexicana; argentina; rápidas.

La receta para una hamburguesa podría estar en varias secciones: americana, carnes, rápidas, etc. Si necesitáramos tener un objeto receta hamburguesa en cada una de las colecciones sería muy poco performante y se usaría mucha memoria.

Entonces, el cliente pide un objeto a la FlyweightFactory, esta se fija si existe en el caché y en caso contrario crea uno nuevo. Flyweight comparte el estado de los objetos.

Veamos cómo representamos esta solución en un diagrama UML.



Implementación del diagrama UML

Código de la **clase** que define el método 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.

```
import java.util.HashMap;
public class ComidaFactory {
    private static final HashMap<String, Comida> comidaMap = new HashMap();

    public static Comida getComida(String tipoComida) {
        Comida comida = (Comida) comidaMap.get(tipoComida);

        if (comida == null) {
            comida = new Comida(tipoComida);
            comidaMap.put(tipoComida, comida);
            System.out.println("Creando objeto comida de tipo: "
+ tipoComida);
        }
        return comida;
    }
}
```

Código de la clase

```

public class Comida {
    private String tipoComida;
    private int precio;
    private boolean tieneLechuga;

    public Comida(String tipoComida) {
        this.tipoComida = tipoComida;
    }

    public String getTipoComida() {
        return tipoComida;
    }
}

```

Objeto: comida con atributos precio, tipo de comida y si tiene lechuga.

```

public int getPrecio() {
    return precio;
}

public void setPrecio(int precio) {
    this.precio = precio;
}
public boolean isTieneLechuga() {
    return tieneLechuga;
}

public void setTieneLechuga(boolean tieneLechuga) {
    this.tieneLechuga = tieneLechuga;
}

```

Objeto: comida con atributos precio, tipo de comida y si tiene lechuga.

```

public void descripcionDeLaComida () {
    System.out.println("Es un/una " + getTipoComida()
+ " que sale: " + getPrecio());
}

```

Objeto: comida con atributos precio, tipo de comida y si tiene lechuga.

Módulo 2: Testeo unitario, logging y acceso a datos

Pruebas Unitarias JUnit

¿Cómo podemos comprobar que nuestro código funciona en todos los escenarios? Para comprobarlo utilizaremos JUnit que es la manera de testear nuestro código y saber que funciona correctamente.

Para poder testear nuestro código tenemos la opción de hacerlo de manera específica, o sea, cubriendo una parte del código mediante un test unitario o de manera global, con los test de integración que cubren un módulo de test. **JUnit** nos la da posibilidad de **generalizar lo que sucede en cada test** y para eso tenemos las siguientes anotaciones que debemos utilizar:

- **@BeforeAll:** Se va a ejecutar una sola vez antes de todos los textos unitarios y es un buen momento para, por ejemplo, inicializar variables.

- **@BeforeEach**: Se va a ejecutar antes de cada test y también nos sirve para inicializar variables comunes a todos los test.
- **@Test**: Sirve para generar un test unitario
- **@Disabled**: Lo usamos cuando queremos que el test no se ejecute.
- **@AfterEach**: Ejecuta un método después de la ejecución de cada test
- **@AfterAll**: Ejecuta un método después de la ejecución de todos los test.

```

1 import org.junit.jupiter.api.*;
2
3 public class CicloVidaTest {
4     @BeforeAll
5     static void initAll() {
6     }
7
8     @BeforeEach
9     void init() {
10    }
11
12    @Test
13    void regular_testi_method() {
14        ...
15    }
16
17    @Test
18    @Disabled("este test no se ejecuta")
19    void skippedTest() {
20        // not executed
21    }
22
23    @AfterEach
24    void tearDown() {
25    }
26
27    @AfterAll
28    static void tearDownAll() {
29    }
30 }
```

Además de las anotaciones contamos con algunas funcionalidades particulares justamente para realizar validaciones adentro del test:

- **assertEquals()**: Funciona para comparar si dos resultados son iguales
- **assertTrue()**: Para saber si el resultado es verdadero o falso
- **thrown.expect()**: Nos aseguramos que recibimos una excepción por ejemplo, la división por cero debería devolver un error.

```

1 import org.junit.jupiter.api.Test;
2
3 import static
4 org.junit.jupiter.api.Assertions.*;
5
6 class AssertionsTest {
7
8     @Test
9     void standardAssertions() {
10         assertEquals(2, 2);
11         assertEquals(4, 4, "Ahora el mensaje opcional de
12 la aserción es el último parámetro.");
13         assertTrue(edad == 2, "Los números son
14 iguales?");
15
16     @Rule public ExpectedException thrown=
17     ExpectedException.none();
18
19     @Test
20     public void myTest() {
21         thrown.expect( Exception.class );
22         thrown.expectMessage("Init Gold must be >= 0");
23
24         rodgers = new Pirate("Dread Pirate Rodgers", -100);
25     }
26 }
```

Los **TEST UNITARIOS** son aplicables a todo el código y mientras más tengamos más cubierta vamos a tener nuestra aplicación.

Con los **TEST DE INTEGRACIÓN** podemos asegurarnos que los diferentes flujos del código funcionan correctamente, por ejemplo, podemos tener un módulo de usuarios en el que podemos crear, eliminar, actualizar o buscar un usuario. Podemos hacer un test unitario por cada uno y armar una suite con todos los test unitarios para corroborar que el módulo usuario funciona correctamente como un todo.

Los test unitarios y de integración son importantes cuando necesitamos saber si nuestro sistema cumple con las especificaciones y, más importante aún, cuando necesitamos actualizar una parte del código o mejorar la performance. Si el test sigue corriendo bien significa que nuestro cambio está bien hecho es importante. Además, debemos estar atentos a un buen diseño de test para no tener falsos positivos.

Tal como pudimos dar cuenta, JUnit provee una gran variedad de assertions que se encuentran ubicadas en ***org.junit.jupiter.api.Assertions***, por ejemplo:

- `assertArrayEquals`
- `assertEquals`
- `assertTrue` and `assertFalse`
- `assertNull` and `assertNotNull`
- `assertSame` and `assertNotSame`
- `assertAll`
- `assertNotEquals`
- `assertIterableEquals`
- `assertThrows`
- `assertTimeout` and `assertTimeoutPreemptively`
- `assertLinesMatch`

```

assertEquals(4, 4);
assertNotEquals(3, 4);
assertTrue(true);
assertFalse(false);
assertNull(null);
assertNotNull("Hello");
assertNotSame(originalObject, otherObject);
```

Introducción a Test Unitarios y Test de Integración

Test de integración

Las unidades individuales se integran para formar componentes más grandes, por ejemplo, dos unidades que ya fueron probadas se combinan en un componente integrado y se prueba la interfaz entre ellas. Esto nos permite cubrir un área mayor de código, del que a veces no tenemos control.

Por lo tanto, podemos concluir que este tipo de test tiene por objetivo validar la interacción entre los módulos de software.

Test Unitarios

Los test o pruebas unitarias tienen por objetivo tomar una pequeña parte de software, **aislándola** del resto del código, para determinar si se comporta/funciona tal como esperamos.

Cada unidad se prueba por separado antes de ser integrada en los componentes para probar las interfaces entre las unidades.

Cabe aclarar que, cualquier dependencia del módulo bajo prueba debe sustituirse por un mock o un stub, para acotar la prueba específicamente a esa unidad de código.

Para llevar a cabo un correcto test unitario se debe seguir un proceso conocido como 3A:

ARRANGE (ORGANIZAR): En este paso se **definen los requisitos** que debe cumplir el código.

ACT (ACTUAR): Aquí se **ejecuta el test** que dará lugar a los resultados que debemos analizar

ASSERT (AFIRMAR): Se **comprueban si los resultados obtenidos** son los esperados. Si es así, se valida y se continúa. Caso contrario, se corrige el error hasta que desaparezca.

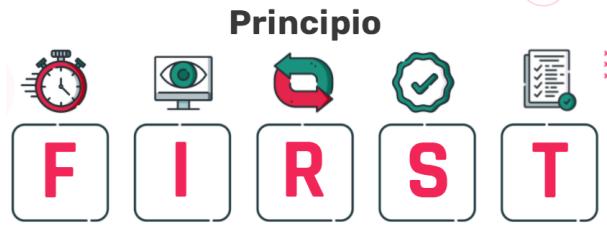
Ventajas de los test unitarios

Veamos algunos beneficios de utilizar este tipo de test:

- **Facilitar los cambios en el código:** Al detectar el error rápidamente es más fácil cambiarlo y volver a probar.
- **Encontrar bugs:** Probando componentes individuales antes de la integración. Esto genera que no impacten en otra parte del código.
- **Proveen documentación:** Ayudan a comprender qué hace el código y cuál fue la intención al desarrollarlo.
- **Mejoran el diseño y la calidad del código:** Invitan al desarrollador a pensar en el diseño antes de escribirlo (Test Driven Development - TDD).



Continuando con los test unitarios, vamos a conocer las cinco características que deben tener para ser considerados tests con calidad. Estas características son conocidas como el principio F.I.R.S.T.



FAST (RÁPIDOS): Es posible tener miles de tests en nuestro proyecto y deben ser rápidos de correr.

ISOLATED/INDEPENDENT (AISLADOS/INDEPENDIENTES): Un método de test debe cumplir con los «3A» (arrange, act, assert). Además, no debe ser necesario que sean corridos en un determinado orden para funcionar, es decir, deben ser independientes unos de los otros.

REPEATABLE (REPETIBLES): Resultados determinísticos. No deben depender de datos del ambiente mientras están corriendo —por ejemplo, la hora del sistema—.

SELF-VALIDATING (AUTOVALIDADOS): No debe ser requerida una inspección manual para validar los resultados.

THOROUGH (COMPLETOS): Deben cubrir cada escenario de un caso de uso y no solo buscar una cobertura del 100%. Probar mutaciones, edge cases, excepciones, errores, entre otros.

¿Qué es JUnit?

JUnit¹ es el framework open source de testing para Java más utilizado. Nos permite escribir y ejecutar tests automatizados. Es soportado por todas las IDEs (Eclipse, IntelliJ IDEA), build tools (Maven, Gradle) y por frameworks como Spring. Conozcamos su arquitectura.

VINTAGE: Contiene el motor de JUnit 3 y 4 para correr tests escritos en estas versiones (Old test).

JUPITER: Es el componente que implementa el nuevo modelo de programación y extensión: API para escribir tests y motor para ejecutarlos (New tests).

COMPONENTES DE TERCEROS: La idea es que otros frameworks puedan ejecutar sus propios casos de prueba realizando una extensión de la plataforma.

PLATFORM: Es un componente que actúa de ejecutor genérico de los tests. La platform-launcher es usada por las IDEs y los build tools.

¹ <http://junit.org>

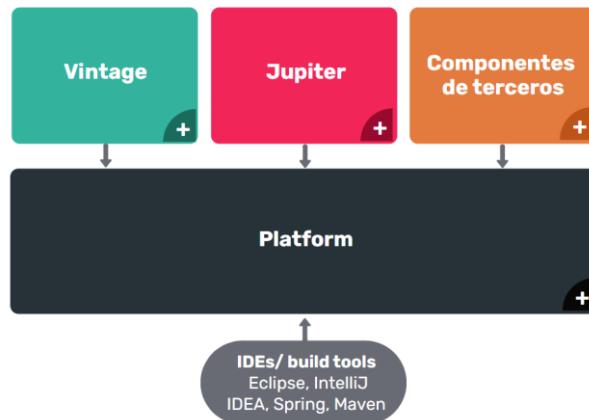
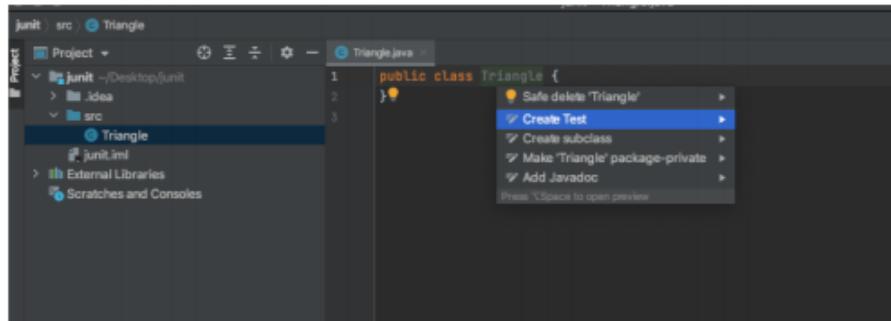


ILUSTRACIÓN 11: ARQUITECTURA DE JUNITS

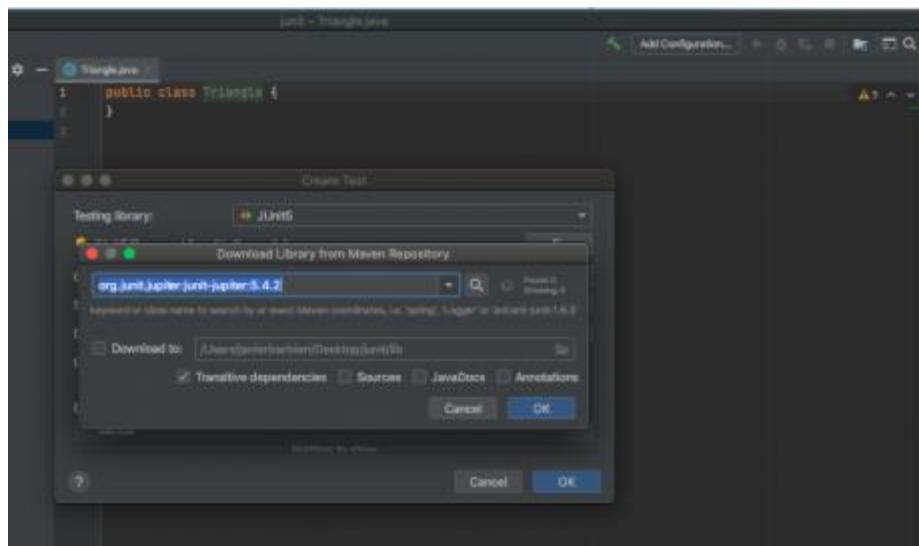
¿Cómo configuramos la librería?

Para configurar la librería de JUnit en nuestro entorno de desarrollo (IntelliJ IDEA) debemos seguir los siguientes pasos:

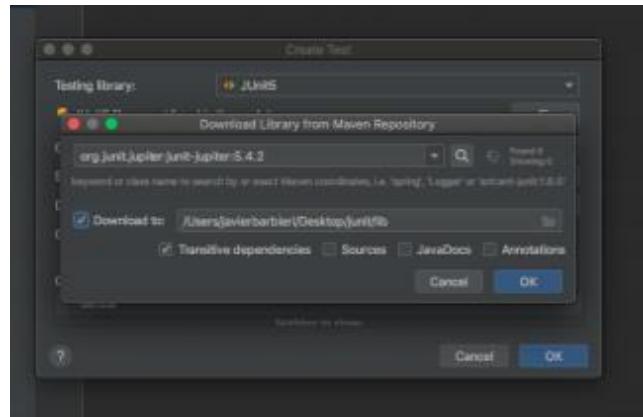
Nos paramos sobre la clase que queremos testear, hacemos alt + Enter y hacemos click en Create Test.



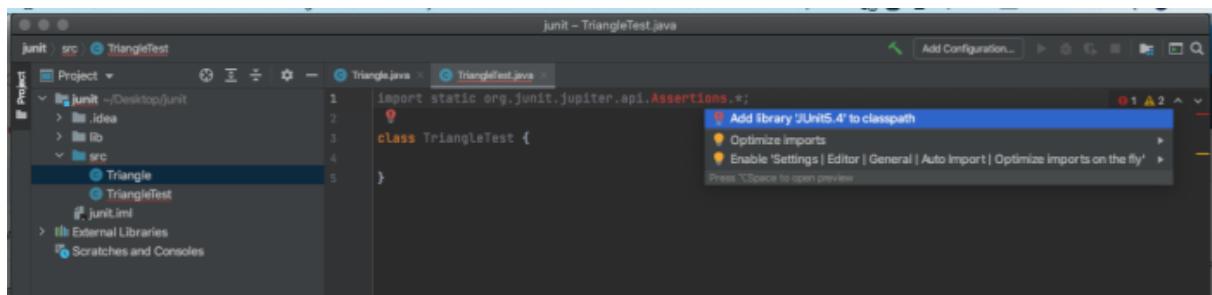
Luego el IDE nos guiará automáticamente para agregar la librería de JUnit de la siguiente manera:



A continuación, simplemente hacemos click sobre el botón OK y se descargará la librería en la ruta recomendada.



Como último paso debemos agregarla a lo que se denomina classpath. La manera más fácil es agregar el paquete, como vemos en la imagen, y en la recomendación del IDE aparece cómo agregarla.



Testeo Parametrizado y Test Suite

Testeo parametrizado

En nuestros test se realizan múltiples comprobaciones simplemente para probar diferentes casos. Esto nos lleva a repetir código, como, por ejemplo:

```
import org.junit.Assert;
import org.junit.Test;
public class MultiplicarTest {
    @Test
    public void debemosCorroborarMultiplicaciones() {
        Assert.assertEquals(4, 2*2);
        Assert.assertEquals(6, 3*2);
        Assert.assertEquals(5, 5*1);
        Assert.assertEquals(10, 5*2);
    }
}
```

Para construir test parametrizados, JUnit utiliza un custom runner que es Parametrized. El cual nos permite definir los parámetros de varias ejecuciones de un solo test.

```

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;

@RunWith(Parameterized.class)
public class MultiplicarTest {
    @Parameterized.Parameters
    public static Iterable data(){
        return Arrays.asList(new Object[][]{
            {4,2,2},{6,3,2},{5,5,1},{10,5,2}
        });
    }

    private int multiplierOne;
    private int expected;
    private int multiplierTwo;

    public MultiplicarTest(int expected, int multiplierOne, int multiplierTwo) {
        this.multiplierOne = multiplierOne;
        this.expected = expected;
        this.multiplierTwo = multiplierTwo;
    }

    @Test
    public void debeMultiplicarElResultado(){
        Assert.assertEquals(expected,multiplierOne*multiplierTwo);
    }
}

```

En la línea de código que aparece debajo estamos indicando que vamos a utilizar el runner de Parameterized, el cual se encargará de ejecutar el test las veces necesarias dependiendo del número de parámetros configurado.

`@RunWith(Parameterized.class)`

La anotación `@Parameters` indica cuál es el método que nos va a devolver el conjunto de parámetros a utilizar por el runner.

Lo que necesitamos es un constructor que permita ser inicializado con los objetos que tenemos en cada elemento de la colección.

Finalmente, se ejecutará el test utilizando los datos que hemos recogido en el constructor.

Test Suite

JUnit test suite nos permite agrupar y ejecutar los tests en grupo. Las suites de prueba se pueden crear y ejecutar con estas anotaciones: `@RunWith` o `@SuiteClasses`

```

private int multiplierOne;
private int expected;
private int multiplierTwo;

public MultiplicarTest(int expected, int multiplierOne, int multiplierTwo) {
    this.multiplierOne = multiplierOne;
    this.expected = expected;
    this.multiplierTwo = multiplierTwo;
}

@Test
public void debeMultiplicarElResultado(){
    Assert.assertEquals(expected,multiplierOne*multiplierTwo);
}
}

```

```

import junit.framework.Assert;

import org.junit.Test;

public class TestFeatureDos {
    @Test
    public void testSegundoFeature()
    {
        Assert.assertTrue(true);
    }
}

```

```

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

import com.TestFeaturePrimero;
import com.TestFeatureSegundo;

@RunWith(Suite.class)
@SuiteClasses({ TestFeaturePrimero.class, TestFeatureSegundo.class })
public class TestFeatureSuite {
    //
}

```

Clase 10: Logging (Trazas y Debug)

Logueo

El logueo es una herramienta para tener documentadas las ejecuciones dentro del sistema y, además, para poder entender que pasa cuando hay un error. Para esto usamos una librería que se llama **LOG4J** qué es la encargada de escribir en nuestro archivo de log lo que consideramos importante además de los errores. Para que esto funcione necesitamos tener un archivo que se llame log4j.properties y las librerías de apache también nuestro class path.

```

1 # Logger con opción root
2 log4j.rootLogger=DEBUG, stdout, file
3 log4j.logger.infoLogger=DEBUG
4 log4j.additivity.infoLogger = false
5
6 # Redirigir mensajes por consola
7 log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
8 log4j.appenders.stdout.Target=System.out
9 log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
10 log4j.appenders.stdout.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}] [%-5p] [%c{1}:%L] %m%n
11
12 # Redirigir los mensajes a un fichero de texto
13 # soportando file rolling
14 log4j.appenders.file=org.apache.log4j.RollingFileAppender
15 log4j.appenders.file.File=demoApplication.log
16 log4j.appenders.file.MaxFileSize=5MB
17 log4j.appenders.file.MaxBackupIndex=10
18 log4j.appenders.file.layout=org.apache.log4j.PatternLayout
19 log4j.appenders.file.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}] [%-5p] [%c{1}:%L] %m%n

```

Un ejemplo de cómo podemos agregar contenido al archivo log es de la siguiente manera:

```

1 private static final Logger logger = Logger.getLogger(TestLog.class);
2 logger.info("Hello World, This is an information message.");
3 logger.error("Hello World, This is a error message.");
4 logger.warn("Hello World, This is a warning message.");
5 logger.debug("Hello World, This is a debugging message.");
6 logger.fatal("Hello World, This is a fatal message.");
7

```

En este caso, instanciamos la clase logger desde get.Logger y elegimos que tipo de log vamos a agregar. Los tipos son: info, error, warn, debug y fatal. Desde nuestro archivo de configuración podemos elegir cuál vamos a mostrar. Es importante filtrar algunos de estos mensajes en producción ya que los logs suelen crecer muy rápido.

Un archivo de log se implementa en cualquier clase JAVAy es tarea del developer agregarlo y darle valor al archivo log. Mientras **más detalles** agreguemos **mejor** va a ser nuestra **lectura sobre los logs**.

No loguear en nuestras clases es bastante problemático porque si surge un error prácticamente tenemos que adivinar qué está pasando. Por lo tanto, es de vital importancia usar logs en las áreas fundamentales con un buen detalle sobre qué está pasando ya que esto nos va a dar muchas herramientas a la hora de entender que sucede en el sistema, especialmente, si es algo que otros desarrolladores van a chequear.

Logging

Para poder loguear necesitamos agregar un archivo de configuración. Lo agregaremos en el root del proyecto, es decir, en la carpeta principal. Por ejemplo, si nuestro proyecto tiene el nombre app, en esa carpeta creamos un archivo de configuración con el nombre log4j.properties con el siguiente contenido:

En la primera línea estamos indicando el nivel mínimo de logging y los appenders que vamos a emplear. En este caso usaremos un nivel de logging establecido en DEBUG y creamos dos appenders, stdout y file.

```
log4j.rootLogger=DEBUG, stdout, file
```

La segunda línea sirve para configurar en qué nivel se empezarán a mostrar las advertencias tanto por consola como a almacenarse en el fichero.

```
log4j.logger.infoLogger=DEBUG
```

Y, con la tercera línea evitamos que los appenders hereden la configuración de sus appenders padres, en caso de que los hubiera (en el nuestro, sería el appender principal así que no tenemos ese problema).

```
log4j.additivity.infoLogger = false
```

Crear la configuración para imprimir mensajes por consola

En la primera línea indicamos qué tipo de logger será, haciendo referencia a la clase que imprimirá los mensajes (¡Recordá la sección Appenders!).

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
```

En la segunda línea, le decimos que queremos imprimirlo directamente por la consola.

```
log4j.appender.stdout.Target=System.out
```

Y las dos últimas líneas son para configurar la plantilla que tendrá cada mensaje. Podés ver todas las posibles opciones de configuración en la página de ayuda de Oracle.

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}][%-5p][%c{1}:%L] %m%n
```

El mensaje de salida sería algo tal que así:

```
[2018-08-03 11:48:39] [INFO] [App:29] Esto es una prueba desde App class
```

Configurar el appender

En estas líneas vamos a hacer exactamente lo mismo que antes, pero configurando el appender para que salga a través de un fichero.

En la primera línea configuraremos la clase como RollingFileAppender, lo que significa que se crearán distintos ficheros al cumplirse determinadas condiciones que tratamos en las siguientes líneas.

```
log4j.appender.file=org.apache.log4j.RollingFileAppender
```

En la siguiente línea indicamos el nombre (con ruta incluida) que queremos que tenga nuestro fichero de log.

```
log4j.appender.file.File=avisos.log
```

Con MaxFileSize establecemos el tamaño máximo que tendrá nuestro fichero, y con MaxBackupIndex indicamos cuántos archivos podemos tener usando el mismo log. A partir de llegar al máximo, comenzarán a sobrescribirse empezando por el más antiguo.

```
log4j.appender.file.MaxFileSize=5MB
```

Y, por último, al igual que por consola, indicamos que plantilla tendrán nuestros mensajes.

```
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.file.layout.ConversionPattern=[%d{yyyy-MM-dd HH:mm:ss}][%-5p][%c{1}:%L] %m%n
```

Concluyendo...

La clase que hemos visto nos da la posibilidad de loguear distintos tipos de logeo, que van de la mano de qué tipo de información queremos mostrar.

Los tipos de logs nos dan información más exacta de qué está pasando, por ejemplo, si usamos:

```
logger.info("Este es un mensaje solo con información");
```

Vemos que solo nos sirve para agregar un comentario de qué pasó en ese momento, por lo tanto, no nos da mucho valor.

Por otro lado, si agregamos algo como:

```
logger.error("El usuario es invalido", e);
```

Claramente el logs nos dice que tenemos un error, y la letra 'e' como segundo parámetro nos va a imprimir en el log, el stacktrace del error. Un ejemplo sería:

```
Exception in thread "main" java.lang.NullPointerException at  
com.example.myproject.Book.getTitle(Book.java:16) at  
com.example.myproject.Author.getBookTitles(Author.java:25) at  
com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Podemos observar que el error principal fue NullPointerException y vemos donde se generó.

Como vimos anteriormente, los logs levels son: FATAL, ERROR, WARN, INFO, DEBUG, TRACE y ALL.

Ahora podremos ver la visibilidad, dependiendo de cuál elegimos en:

```
log4j.logger.infoLogger=DEBUG
```

Ejemplo

En el siguiente ejemplo veremos cómo instanciar un objeto logger que nos permitirá loguear mensajes en diferentes prioridades. Los más usados son error, warning, debug e info.

```
import org.apache.log4j.Logger;  
  
public class TestLog {  
    private static final Logger logger = Logger.getLogger(TestLog.class);  
    public static void main(String[] args) {  
        logger.info("Empezamos nuestro metodo MAIN");  
        try {  
            String variable = "Hola";  
            int division = 1 / 0;  
        } catch (Exception e) {  
            logger.error("Error por dividir por cero ", e);  
        }  
        logger.warn("Advertencia el metodo MAIN esta por finalizar");  
        logger.debug("Esto va a mostrarse solo si el infoLogger esta en DEBUG");  
        logger.info("Finalizamos el thread MAIN");  
    }  
}
```

Log4j

Log4j es una librería desarrollada en Java por la Apache Software Foundation que permite a los desarrolladores elegir la salida y el nivel de granularidad de los mensajes o logs en tiempo de ejecución. En otras palabras, es utilizada para generar mensajes de logging de una forma limpia, sencilla, permitiendo filtrarlos por importancia y pudiendo configurar su salida tanto por consola, fichero u otras diferentes.

VENTAJA: Permite tener un registro de lo que está pasando en nuestros sistemas, lo que nos posibilita entender mejor los errores.

DESVENTAJA: La única desventaja es que a veces los archivos se hacen muy grandes y ocupan mucho espacio. Es por ello que debemos elegir bien qué tipo de información queremos almacenar.

Arquitectura Log4j

La API de Log4j sigue una arquitectura en capas donde cada una proporciona diferentes objetos para realizar diferentes tareas. Esta estructura hace que el diseño sea flexible y fácil de ampliar en el futuro.

Los dos tipos de objetos disponibles con el marco Log4j son:

OBJETOS BASE: Objetos de marco obligatorios y necesarios para utilizar el marco.

OBJETOS DE SOPORTE: Objetos de marco opcionales que soportan objetos básicos para realizar tareas adicionales.

Objetos base

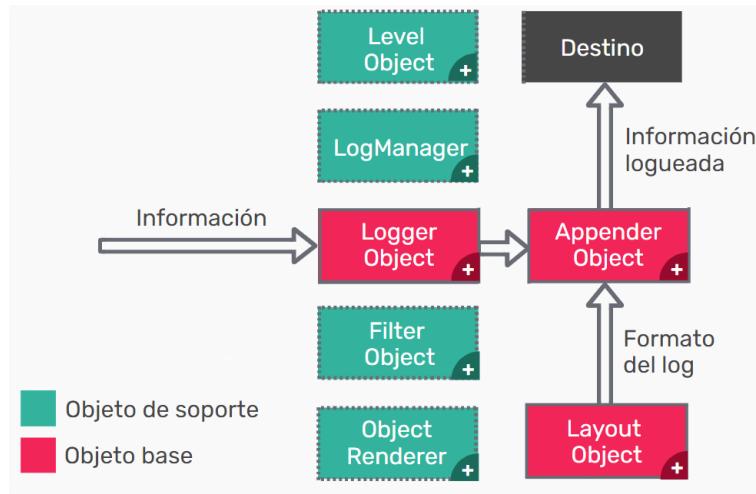
Los objetos base incluyen los siguientes tipos de objetos:

- **LOGGER OBJECT:** La capa de nivel superior es el Logger que proporciona el objeto Logger. Este es responsable de capturar la información de registro y se almacena en una jerarquía de espacio de nombres.
- **APPENDER OBJECT:** Esta es una capa de nivel inferior que proporciona el objeto appender. Este es responsable de publicar información de registro.
- **LAYOUT OBJECT:** La capa de diseño proporciona objetos que se utilizan para formatear la información del registro en diferentes estilos. Los objetos de diseño juegan un papel importante en la publicación de información de registro de una manera que sea legible.

Objetos de soporte

Mientras que los objetos de soporte incluyen los siguientes tipos de objetos:

- **LEVEL OBJECT:** El objeto de nivel define la granularidad y prioridad de cualquier información de registro. Hay siete niveles de registro definidos en la API: OFF, DEBUG, INFO, ERROR, WARN, FATAL y ALL.
- **LOGMANAGER:** Administra el marco de registro. Es responsable de leer los parámetros de configuración inicial de una configuración de todo el sistema, un archivo de configuración o una class de configuración.
- **FILTER OBJECT:** El objeto de filtro se utiliza para analizar la información de registro y tomar otras decisiones sobre si esa información debe registrarse o no.
- **OBJECT RENDER:** Se especializa en proporcionar una representación de cadena de varios objetos pasados a la infraestructura de registro. Los objetos layout utilizan este objeto para preparar la información de registro final.



Niveles de registro

Por defecto Log4j tiene niveles de prioridad para los mensajes, entre ellos se encuentran:

- OFF: Este es el nivel de mínimo detalle, deshabilita todos los logs.
- FATAL: Se utiliza para mensajes críticos del sistema, generalmente después de guardar el mensaje, el programa se cierra.
- ERROR: Indica eventos de error que aún podrían permitir que la aplicación continúe ejecutándose.
- WARN: Se utiliza para mensajes de alerta sobre eventos.
- INFO: Se refiere a mensajes informativos que resaltan el progreso de la aplicación en un nivel aproximado.
- DEBUG: Designa los eventos informativos detallados más útiles para depurar una aplicación.
- TRACE: Se utiliza para mostrar mensajes con un mayor nivel de detalle que debug.
- ALL: Es el nivel de máximo detalle, habilita todos los logs.

	FATAL	ERROR	WARN	INFO	DEBUG	TRACE	ALL
FATAL	X						
ERROR	X	X					
WARN	X	X	X				
INFO	X	X	X	X			
DEBUG	X	X	X	X	X		
TRACE	X	X	X	X	X	X	
ALL	X	X	X	X	X	X	X

Clase 11: Acceso a Base de Datos

Conectarse a una base de datos es un paso fundamental en nuestro desarrollo de sistemas ya que los datos son tan importantes como el código que lo genera. Desde JAVA nos conectamos a una base de datos mediante JDBC.

¿Cómo guardar la información de un programa orientado a objetos en una base relacional? Para esto, debemos encontrar la manera de que la clase quede guardada en la tabla con su mismo nombre. Esta relación se hace respetando que cada atributo de la clase corresponde a una columna de la tabla, es

decir, **debemos respetar el tipo**. La manera de conectarnos es por JDBC, un protocolo que nos ayuda a conectar un programa Java con una base de datos relacional. Para nuestro ejemplo vamos a utilizar una base de datos h2 qué es totalmente en JAVA.

Imaginemos que tenemos un usuario y queremos guardar los registros en nuestra base de datos, para eso creamos la clase usuario junto a dos simples atributos: ID y name. El ID es un campo obligatorio ya que cada tabla debe tener único ID que debe ser de tipo numérico.

```
1 public class Usuario {  
2     private Long id;  
3     private String name;  
4  
5     public Usuario(Long id, String name) {  
6         this.id = id;  
7         this.name = name;  
8     }  
9  
10    public Long getId() {  
11        return id;  
12    }  
13  
14    public void setId(Long id) {  
15        this.id = id;  
16    }  
17  
18    public String getName() {  
19        return name;  
20    }  
21  
22    public void setName(String name) {  
23        this.name = name;  
24    }  
25 }
```

Después procedemos a crear nuestra conexión a la base de datos para poder, por ejemplo, insertar los valores que necesitemos. El **objeto Connection** nos va ayudar a preparar las instrucciones para ejecutar en la base de datos.

```
1 Class.forName("org.h2.Driver").newInstance();  
2  
3 Connection con =  
4     DriverManager.getConnection("jdbc:h2:" +  
5         "./Database/my", "root", "myPassword");  
6
```

Después tenemos que crear un **Statement**, siempre es recomendable utilizar prepareStatement ya que nos permite agregar los datos en el query de una manera genérica usando el símbolo de interrogación en dónde va el dato. Por otro lado, si usamos la interfaz Statement debemos poner los datos de una manera manual.

```
1 Statement stmt = con.createStatement();  
2  
3 String createSql = "CREATE TABLE USUARIO(ID  
4     INT PRIMARY KEY, NAME VARCHAR(255));";  
5  
6 String insertSql = "INSERT INTO USUARIO  
7     VALUES(1, 'Diego');"  
8  
9 String sql = "select * from FROM USUARIO  
10    where id = 1";  
11  
12 String sqlConPrepareStatement = "select *  
13     from FROM USUARIO where id = ?";  
14
```

Veamos un ejemplo:

1. Creamos el statement desde la conexión

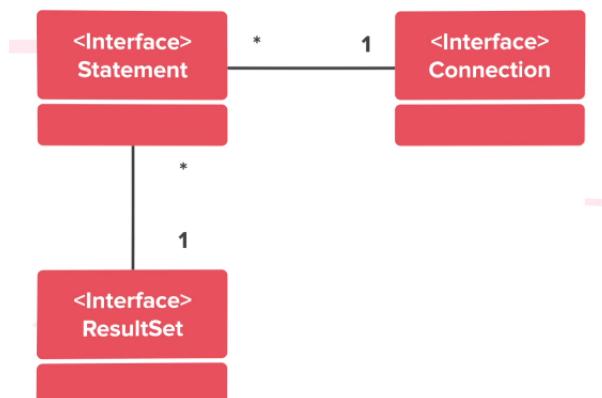
2. Creamos varias Querys. La primera es para definir nuestra tabla, la segunda inserta en nuestra tabla un nuevo registro.
3. Retornamos esos valores desde el resultSet. `rd.getInt(n)` nos va a dar la n columna de nuestra tabla.

```

1 //Ejemplo Statement
2 stmt.execute(createSql);
3 stmt.execute(insertSql);
4 ResultSet rd = stmt.executeQuery(sql);
5
6 while(rd.next()) {
7     System.out.println(rd.getInt(1) +
8     rd.getString(2));
9 }
10

```

JDBC suele ser engorroso cuando se trabajan con grandes tablas que contienen muchos elementos, pero suelen ser usadas cuando debemos hacer grandes querys ya que el tiempo de ejecución es más corto que en un ORM.



Ese Gran importancia que manejemos el acceso a la base de datos ya que no existen proyectos que no necesiten manipular los datos. Generalmente cada tabla debe tener su propio crud, estas son las operaciones básicas en la tabla: lectura, escritura, borrado y creación.

JDBC

JDBC, por las siglas de Java DataBase Connectivity, es un **framework** que consiste en múltiples interfaces y solo algunas clases de soporte. Esto se debe a que la idea detrás de JDBC es que cualquiera pueda crear su propia implementación del framework y adaptarla a sus necesidades. Dado que se trata de un conjunto de interfaces, cualquier código que interactúe con el framework no se verá afectado si se altera la implementación.

Es por esto que JDBC define interfaces que solo declaran el comportamiento que debe llevarse a cabo para conectarse e interactuar con una base de datos. Así, nos encontramos con interfaces tales como: Connection (abstracción del comportamiento de una conexión), Statement (define el comportamiento para realizar sentencias contra una base de datos, sean queries y otras instrucciones), ResultSet (que abstrae el comportamiento para extraer resultados de las consultas), entre otras. Todas estas clases e interfaces están dentro del **paquete java.sql.***. Entonces, para interactuar con los diferentes motores de base de datos, debemos tener una implementación de estas y otras interfaces, es decir, una clase concreta que implemente cada interfaz.

Uso y funcionamiento de un driver JDBC

Si bien cualquiera puede implementar las interfaces que JDBC provee, puede ser un trabajo arduo y extenso, ya que requiere de un gran conocimiento sobre motores de bases de datos. Por lo tanto, hay una implementación de JDBC.

Cada implementación del framework consiste en un paquete de clases, típicamente empaquetados en archivos .jar. Se pueden descargar de los sitios oficiales de cada motor de base de datos y son conocidos con el nombre de driver jdbc. Es decir que tenemos un driver jdbc para Oracle, uno para MySQL, etc.

Cada driver es un paquete de clases. Como tal, debemos incorporarlo en forma de librería a nuestro programa java. De esta forma, tendremos accesibles en el CLASSPATH, las clases que implementan las interfaces del framework. Si bien es posible hacerlo manualmente, típicamente hacemos esto mediante el IDE.

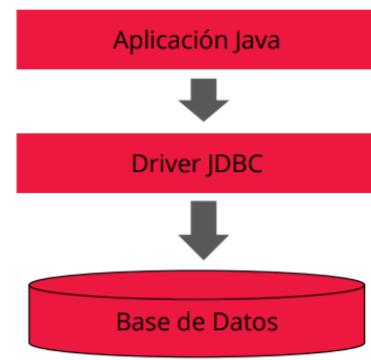
Una vez disponible el driver en el CLASSPATH podremos empezar a trabajar con el driver. En su expresión más básica, cada vez que queramos interactuar con la base de datos necesitamos realizar una serie de pasos:

1. Cargar el Driver
2. Obtener la conexión
3. Ejecutar la consulta contra la base
4. Cerrar la conexión

Veamos, en código, cómo se llevan a cabo estos pasos. Supongamos una clase de prueba, con un método main en el que ponemos:

```
Connection c = null;
try {
    Class.forName("org.h2.Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    System.exit(0);
}

try {
    String url =
"jdbc:h2:tcp://localhost//D:/base_de_datos/ejemplo";
    c = DriverManager.getConnection(url, "sa", "sa");
    c.setAutoCommit(false);
} catch (SQLException e) {
    e.printStackTrace();
    System.exit(0);
}
```



```

try {
    Statement s = c.createStatement();
    s.execute("AQUÍ_SENTENCIA_SQL");
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        c.close();
    } catch(SQLException e) {
        e.printStackTrace();
    }
}

```

Detalle

<<Interface>> DriverManager

`getConnection() : Connection`. Con la clase disponible, hacemos uso de la clase DriverManager que es el registro de los drivers jdbc que tenemos configurados y obtenemos una conexión a la base de datos.

DriverManager.getConnection necesita tres parámetros:

- URL de conexión: es una URL para conectarse al servidor, el formato varía ligeramente de proveedor a proveedor, pero siempre se puede encontrar en la documentación oficial de cada uno.
- Usuario: si el motor soporta acceso multiusuario, acá se pone el nombre de usuario.
- Password: si el motor soporta acceso multiusuario, acá se pone la contraseña de usuario.

<<Interface>> Connection

`setAutoCommit(boolean)`. Con este método indicamos si esta conexión debe manejar las transacciones automáticamente (true) o manualmente (false). Por el momento, lo pondremos en true.

`createStatement() : Statement`. Con este método, preparamos el camino para ejecutar una sentencia contra la base de datos. Aquí nuevamente, se obtiene una instancia de una clase que implementa la interfaz Statement.

`close()`. Para cerrar la conexión. Siempre que usamos algún recurso deberemos liberarlo y, según vimos, es uno de los usos típicos del bloque finally en un esquema try-catch-finally.

<<Interface>> Statement

`execute(String) : ResultSet`. Lo que hacemos es ejecutar la sentencia contra la base de datos. Toma como argumento un String con la sentencia SQL propiamente dicha. Cabe aclarar que hay diferentes maneras de ejecutar sentencias:

- `boolean execute(String)`. Se utiliza para ejecutar cualquier sentencia a la base de datos.
- `int executeUpdate(String)`. Se utiliza para ejecutar sentencias DML, o sea, sentencias que manipulen datos (insert, update, delete). Devuelve un entero con la cantidad de filas afectadas por la sentencia
- `ResultSet executeQuery(String)`. Se utiliza para ejecutar consultas a base de datos.

<<Interface>> ResultSet

Se utiliza para obtener resultados de una consulta a la base de datos. Para recorrer los resultados ResultSet tiene una serie de operaciones fundamentales:

`next ()`. Devuelve true o false, indicando si hay una tupla siguiente para analizar o no. Si la hay, avanza el puntero una posición.

`getXYZ (nombreCampo: String)`. Son métodos para obtener los valores de cada campo, dependiendo de su tipo. Por ejemplo:

- `getInt("id")` devolvería el entero que corresponde al campo “id”.
- `getString("nombre")` devolvería el String con el valor del campo “nombre”.
- `getDate("fecha")` devolvería un java.sql.Date almacenado en el campo fecha

Y la lista sigue: getBoolean, getLong, getDouble, getByte, etc.

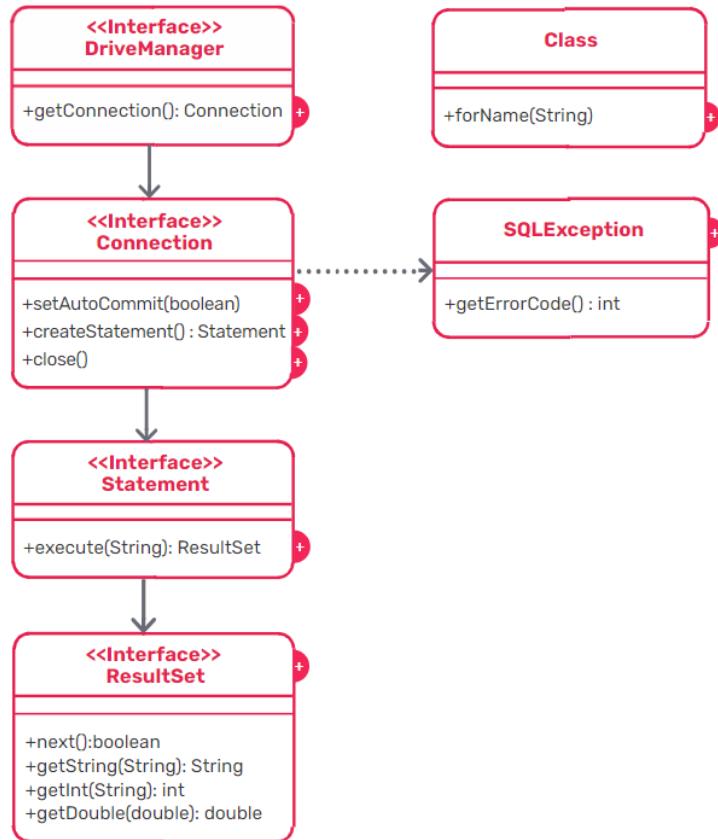
Cabe aclarar que `getXYZ` también acepta un número de columna como un entero —es decir, podríamos hacer `getString(2)` para obtener el valor del nombre del usuario, si el índice de columna está basado en 1 y no en 0—. Si se utilizan “alias” en la consulta, el String que se pase al `getXYZ` será el alias.

Class

`forName (String)`. Con este método cargamos o “registramos” la clase principal del driver. Esto hace disponible al DriverManager para poder administrar los drivers jdbc e ir instanciando las clases —que implementan las interfaces— del driver. Como lo que estamos haciendo es buscar una clase por su nombre, mediante un parámetro String, el compilador puede tomar esa sintaxis como válida, pero al momento de la ejecución, puede que la clase no se encuentre en el CLASSPATH. Por eso, este método arroja una ClassNotFoundException que debe manejarse adecuadamente.

SQLException

`getErrorCode () : int`. Es una checked exception como cualquier otra y tiene un atributo muy útil cuyo valor podemos consultar con el método `getErrorCode()`. Este método nos da el código de error que arrojó la base de datos, es decir que, además de la excepción, podemos saber a más bajo nivel qué está ocurriendo.



H2

H2 es una base de datos open source escrita en Java que permite integrar aplicaciones en Java o ejecutarse en modo cliente-servidor. Principalmente, se puede configurar para que se ejecute como una base de datos en memoria. Entonces, los datos no persistirán en el disco, debido a que la base de datos no se utiliza para el desarrollo de producción, sino principalmente para el desarrollo y las pruebas.

Características

ALTA INTEGRACIÓN: Debido a que está implementada en Java, su integración con cualquier aplicación en este lenguaje es total (mediante API JDBC o ODBC).

MULTIPLATAFORMA: Se puede utilizar en diferentes plataformas.

MODO “EN MEMORIA”: Realiza la gestión de los datos directamente sobre la memoria, lo que acelera enormemente las operaciones realizadas.

MODO EMBEBIDO: Permite el funcionamiento en modo embebido realizando la gestión de los datos en archivos haciendo uso de una pequeña parte de memoria.

TAMAÑO REDUCIDO: Ocupa muchísimo menos que otras bases de datos.

RÁPIDA: Obtiene su gran velocidad gracias a su estrategia de optimización basada en costos que utiliza un algoritmo genético para consultas complejas, sin administración.

Conectándonos a H2 desde Java

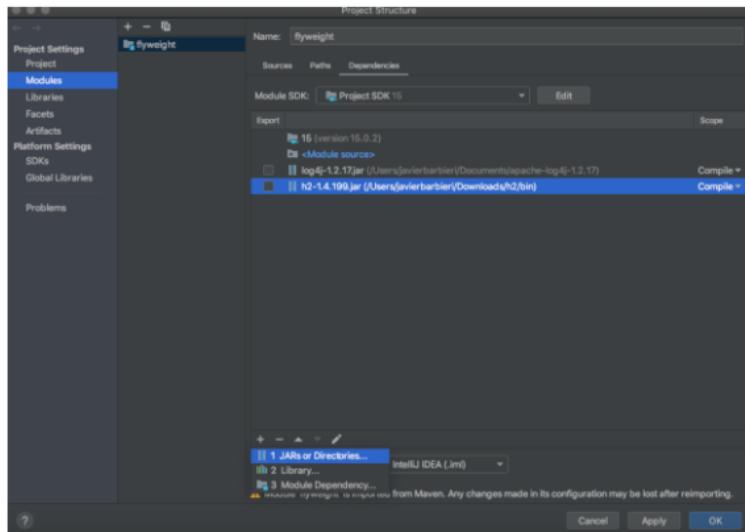
Con el objetivo de conectarnos, vamos a crear una Tabla TEST en la BD. Para eso en la consola ejecutamos:

```
DROP TABLE IF EXISTS TEST;
```

```
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
```



Ahora vamos a agregar el jar h2-1.4. que está en nuestra carpeta bin, para llegar a la carpeta bin, debemos extraer el zip que bajamos previamente.



Hagamos un pequeño ejemplo para entender el Statement. Si queremos ver estos datos, tenemos que ejecutar un ResultSet.

```
public class TestLog {
    public static void main(String[] args) throws Exception {
        Class.forName("org.h2.Driver").newInstance();
        Connection con = DriverManager.getConnection("jdbc:h2:" + "./Database/my", "root", "myPass");
        Statement stmt = con.createStatement();
        String createSql = "DROP TABLE IF EXISTS TEST;\n" +
                           "CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));\n" +
                           "INSERT INTO TEST VALUES(1, 'Hello');\n" +
                           "INSERT INTO TEST VALUES(2, 'World');\n";
        String sql = "select * from TEST";
        stmt.execute(createSql);
        ResultSet rd = stmt.executeQuery(sql);
        while(rd.next()) {
            System.out.println(rd.getInt(1) + rd.getString(2));
        }
    }
}
```

Clase 13: Consultas y Transacciones sobre Base de Datos

Para hablar de las consultas a la base de datos usamos la palabra Query, hacer una Query significa consultar a la base de datos. Por ejemplo, tenemos una tabla de usuario y guardamos dos registros, cada registro corresponde a una fila en la tabla de la base de datos. Si queremos saber si los registros están podemos hacer el siguiente statement, dentro de los ResultSet vamos a tener los dos registros.

```
1 String sql = "select * from USUARIO";
2 stmt.execute(createSql);
3 ResultSet rs = ps.executeQuery();
```

Existen querys muy complejas que pueden afectar la performance de la aplicación. Imaginemos una consulta en una tabla con miles de registros, una buena query nos dará la información que necesitamos y nos ahorrará tiempo de ejecución en la aplicación. Las consultas a la base de datos se implementan con **JDBC** que es una manera de conectar nuestra aplicación a la base de datos. Veamos por ejemplo una consulta simple a la tabla empleado:

```
1 Class.forName("org.h2.Driver").newInstance();
2 Connection con =
3 DriverManager.getConnection("jdbc:h2:" +
4 "./Database/my", "root", "myPassword");
5
6 String sql = "select * from EMPLEADO";
7
8 Statement stmt = con.createStatement();
9
10 ResultSet rd = stmt.executeQuery(sql);
11
12 while(rd.next()) {
13     System.out.println(rd.getInt(1) +
14     rd.getString(2));
15 }
16
17
```

Por otro lado, una transacción es un conjunto de operaciones sobre una base de datos que deben ejecutarse como una unidad. Si una de las operaciones tiene un error obtendremos un rollback, esto significa que podemos volver al estado anterior, o sea, al último commit hecho. La parte más importante de este código es el auto commit en false, esto debería iniciar la ejecución de sentencias sin commitear al final cuando termine todas esas sentencias. Si hay un error, disparará una excepción donde haremos el rollback.

```
1 String sql = "INSERT INTO EMPLEADOS(2,'Juan')";
2 conn.setAutoCommit(false);
3 Statement stmt =
4 conn.createStatement();
5
6         stmt.execute(sql);
7
8         // end transaction block, commit
9 changes
10 conn.commit();
11
12         // good practice to set it back to
13 default true
14 conn.setAutoCommit(true);
15
16     } catch (Exception e) {
17         conn.rollback();
18     }
19
20 }
21
22 }
```

Pensemos en el siguiente ejemplo: necesitamos extraer dinero de un cajero automático. En el momento en que introducimos la tarjeta y solicitamos retirar el monto de dinero se **genera una transacción** que finaliza cuando el cajero nos entrega el dinero. ¿Qué sucede si hay un error en el cajero y no nos da nuestro dinero? Si no está aplicada la transacción en la base de datos no contaremos con rollback y no se nos permitirá restituir al estado anterior. Entonces, el saldo en nuestra cuenta sería distinto al que realmente debería ser ya que no recibimos el dinero.

Para las transacciones **debemos pensar si el estado es importante** después de ejecutar de manera secuencial en la base de datos. Por ejemplo, consultar el saldo, restar al saldo 10, sumar al saldo 14 y extraer dinero. Es fundamental que todas esas operaciones se mantengan en una transacción y que en el caso de que, por ejemplo, se presente un error por un corte de luz al extraer el dinero entonces la transacción debería volver al Estado original.

Lo más importante es que estos temas están totalmente relacionados con TI, no importa en qué tecnología trabajemos, todas, en su 99% tienen conexión a base de datos.

Consultas y transacciones sobre base de datos

Utilizar JDBC Prepared Statement

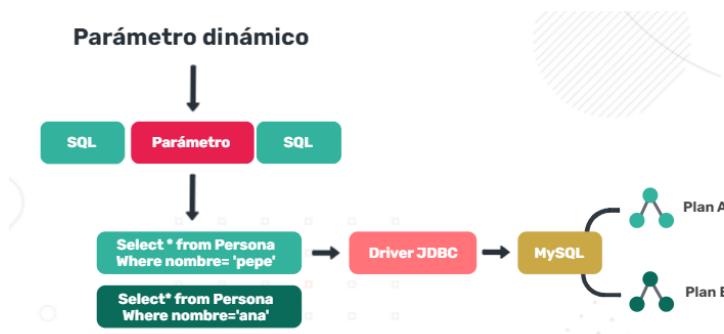
Lo primero que tenemos que entender es cuál es la diferencia entre un **Statement** de JDBC y un **Prepared Statement**. Cuando nosotros construimos una consulta normal de JDBC utilizamos un Statement, o sentencia, que se encarga de definir una consulta SQL a ejecutar contra el motor de la base de datos.

Statement

En este caso estamos construyendo una sentencia y aportando un parámetro a la consulta de forma dinámica. Esto básicamente se convierte en una consulta SQL que nosotros ejecutamos vía el driver JDBC contra la base de datos.

```
Connection conexion = DriverManager.getConnection("jdbc:h2:" +  
        "./Database/my", "root", "myPassword");  
Statement sentencia = conexion.createStatement();  
String nombre="pepe";  
String consulta = "select * from Persona where nombre='"+nombre+"'";  
ResultSet rs=sentencia.executeQuery(consulta);
```

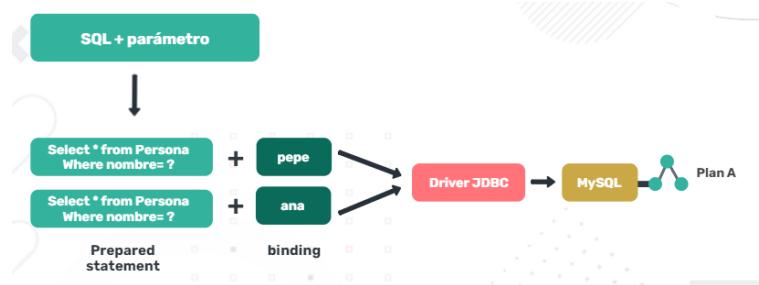
Muchas veces se nos olvida que para cada consulta SQL que construimos contra la base de datos se construye un plan de ejecución en el que la base de datos decide cómo esa consulta se ejecuta.



PreparedStatement

A dos consultas diferentes, se crean dos planes de ejecución diferentes, aunque, ambas consultas sean realmente muy similares y únicamente entre en juego el valor del parámetro que les pasamos. Para solventar este problema existen, los JDBC Prepared statement. Estas estructuras permiten mantener

las consultas neutras sin tener en cuenta los parámetros que se las pasa, ya que realizan un binding de ellos.



De esta forma, cuando la base de datos genera un hash para el plan de ejecución, ambas consultas —la que consulta por Pepe y la que consulta por Ana— devuelven el mismo hash y comparten el plan de ejecución. Vamos a ver esto en código:

```
String consulta = "select * from Persona where nombre = ?";  
Connection conexion= DriverManager.getConnection("jdbc:h2:" +  
        "./Database/my", "root", "myPassword");  
PreparedStatement sentencia=  
    conexion.prepareStatement(consulta);  
sentencia.setString(1, "pepe");  
ResultSet rs = sentencia.executeQuery();
```

A partir de lo visto anteriormente, podemos concluir que no solo nos estaremos ahorrando la construcción de planes de ejecución, sino que también estamos evitando que nos inyecten SQL, ya que, al parametrizar la consulta, la API de JDBC nos protege contra este tipo de ataques. Normalmente el uso de consultas parametrizadas mejora el rendimiento entre un 20 y un 30 % a nivel de base de datos.

Modificar datos con PreparedStatement

Update

Ahora veremos cómo ejecutar un update. Para ello debemos preparar una query y agregarle el símbolo "?" donde vamos a reemplazar el dato. El método executeUpdate() devuelve un entero indicando el número de filas afectadas por la modificación. Por ejemplo:

```
String query = ("UPDATE PERSONAS SET NOMBRE=? WHERE APELLIDO=?");  
try (PreparedStatement pstmt = con.prepareStatement(query)) {  
  
    pstmt.setString(1, "Mariano");  
    pstmt.setString(2, "Martinez");  
    pstmt.executeUpdate();  
  
}
```

Insert

```
String query = ("INSERT INTO PERSONAS (NOMBRE, APELLIDO) VALUES (?,?)");  
try (PreparedStatement pstmt = con.prepareStatement(query)) {  
    pstmt.setString(1, "Mariano");  
    pstmt.setString(2, "Martinez");  
    pstmt.executeUpdate();  
  
}
```

Delete

```
String query = ("DELETE FROM PERSONAS WHERE APELLIDO=?");
try (PreparedStatement pstmt = con.prepareStatement(query)) {
    pstmt.setString(1, "Martinez");
    pstmt.executeUpdate();
}
catch (SQLException ex) {
}
```

Transacciones

Una transacción es un conjunto de operaciones sobre una base de datos que se deben ejecutar como una unidad.

Hay ocasiones en las que es necesario que varias operaciones sobre la base de datos se realicen en bloque, es decir, que se ejecuten o todas o ninguna, pero que no que se realicen solo algunas.

Si se ejecutan parcialmente hasta que una da error, el estado de la base de datos puede quedar inconsistente. En este caso necesitaríamos un mecanismo para devolverla a su estado anterior, pudiendo deshacer todas las operaciones realizadas.

Ejemplo

Supongamos que vamos a reservar un vuelo desde Alicante hasta Sydney. Para llegar a Sydney tenemos que realizar varias escalas. Tendremos que reservar un vuelo de Alicante a Madrid, otro de Madrid a Dubai y por último, volar desde Dubai a Sydney. Si representamos cada reserva como un insert en la base de datos tendremos las siguientes instrucciones:

```
try {
    .....
    s.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
VALUES('Felipe', 'Alicante', 'Madrid')");
    s.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino) VALUES
('Felipe', 'Madrid', 'Dubai')");
    s.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
VALUES('Felipe', 'Dubai', 'Sydney')");
    .....
} catch(SQLException e) {
    // Se ha producido un error pero no sabemos donde
    // ¿Qué operaciones se han realizado? ¿Cómo deshacerlas?
}
```

Un objeto Connection por defecto realiza automáticamente cada operación sobre la base de datos. Esto significa que cada vez que se ejecuta una instrucción, se refleja en la base de datos y no puede ser deshecha. Por defecto está habilitado el modo auto-commit en la conexión.

Los siguientes métodos en la interfaz Connection son utilizados para gestionar las transacciones en la base de datos:

```
void setAutoCommit(boolean valor)
void commit()
void rollback()
```

Para iniciar una transacción deshabilitamos el modo auto-commit mediante el método `setAutoCommit(false)`. Esto nos da el control sobre lo que se realiza y cuándo se realiza.

Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit()`.

Una llamada a rollback() deshará todos los cambios realizados desde el último commit(). Una vez se ha emitido una instrucción commit(), esas transacciones no pueden deshacerse con rollback().

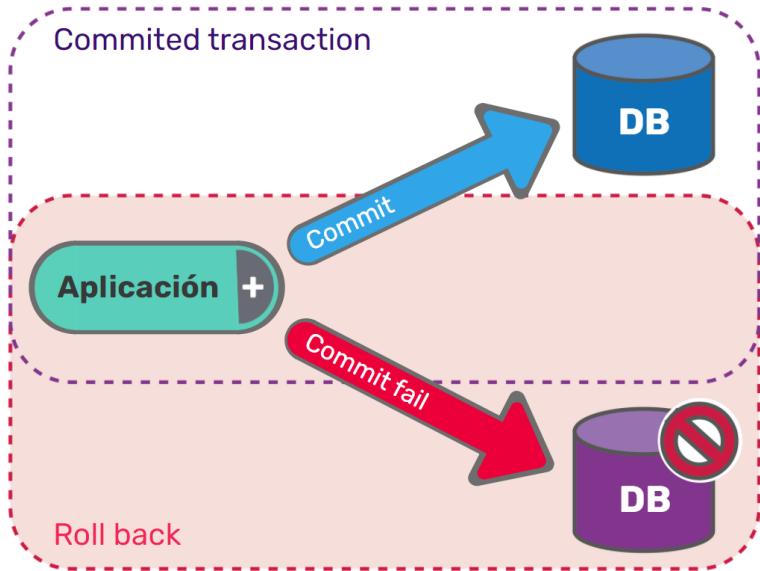


ILUSTRACIÓN 12: LO MISMO EN UN GRAFICO

Pasos para invocar Stored Procedures

1. Las llamadas a los procedimientos almacenados al igual que las PreparedStatements y las consultas simples se hacen sobre la conexión, en este caso, con el método prepareCall() que nos devuelve un CallableStatement.
2. La llamada al procedimiento almacenado, además de ir entre comillas por ser un string, tiene que ir también entre llaves y con el siguiente formato: «{call nombre_procedimiento(?, ?, ...)}» —con tantos signos de interrogación como parámetros tenga el procedimiento almacenado—.
3. Los parámetros de entrada, como con las PreparedStatements, se definen con los métodos setXXX().
4. Hay que definir el tipo de los parámetros de salida con registerOutParameter() donde debemos indicar el tipo del que será ese dato en la base de datos y no en Java.
5. El procedimiento se ejecuta cuando llamamos al método execute(); y, como es lógico, en el momento en el que se ejecute tienen que estar definidos todos los parámetros tanto de entrada como de salida.
6. Finalmente, una vez ejecutado el procedimiento almacenado, podemos obtener los valores que devuelve usando el método getXXX() adecuado para cada caso, recordando que ahora estamos obteniendo los valores en Java, por lo que para obtener un varchar usaremos getString() y no getVarchar().

¿Cómo loguearnos con log4J en una base de datos?

PASO 1: Configurar el appender JDBCAppender en el archivo log4j.properties:

```
# Define the file appender  
log4j.appender.sql=org.apache.log4j.jdbc.JDBCAppender
```

```

log4j.appender.sql.URL=jdbc:h2:/Database/my

# Set Database Driver

log4j.appender.sql.driver=org.h2.Driver

# Set database user name and password

log4j.appender.sql.user=root

log4j.appender.sql.password=password

# Set the SQL statement to be executed.

log4j.appender.sql.sql=INSERT INTO LOGS VALUES ('%x', now(), '%C', '%p', '%m')

# Define the xml layout for file appender

log4j.appender.sql.layout=org.apache.log4j.PatternLayout

```

PASO 2: Crear una tabla Logs en la base de datos:

```

CREATE TABLE LOGS
(
    USER_ID VARCHAR(20) NOT NULL,
    DATED   DATETIME NOT NULL,
    LOGGER  VARCHAR(50) NOT NULL,
    LEVEL   VARCHAR(10) NOT NULL,
    MESSAGE  VARCHAR(1000) NOT NULL
);

```

Clase 14: Patrón DAO (Data Access Object)

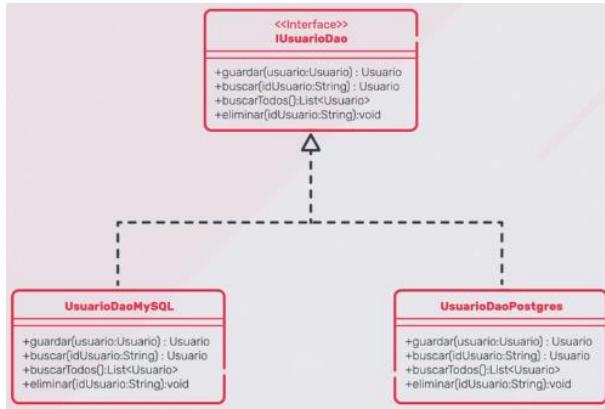
¿Podemos modelar un sistema de tal manera que nos permita cambiar la forma en la que guardamos los datos, pero sin afectar la lógica de negocio?

Pensemos en la siguiente situación: nos encontramos trabajando en un equipo de desarrollo en el que actualmente se utiliza H2 como motor de base de datos, desde el backend implementado todas las operaciones crud para este motor. Pero la empresa decide que todos los equipos migren a MySQL. Necesitamos cambiar de base de datos sin afectar nuestra lógica de negocio, es decir, que el sistema siga funcionando independientemente de la base de datos que estamos utilizando. Veamos como el patrón DAO nos ayudará en esta situación.

El patrón DAO (Data Access Object) nos permitirá aislar en nuestro sistema la capa de negocio, es decir, donde tenemos nuestra lógica principal de la capa de persistencia qué es la que se encarga de realizar las operaciones crud generalmente en una base de datos.

El objetivo principal es que nuestra capa de negocio no se entere los detalles de la implementación de la capa de persistencia de esta manera, a la capa de negocio le es completamente indiferentes si estamos guardando los datos en H2 o MySQL.

Veamos cómo nos queda el modelo en un diagrama:



El patrón nos indica que definamos una interfaz con las operaciones que necesitamos realizar y que, creemos implementaciones adaptando los detalles para utilizar las diferentes opciones de persistencia. De esta manera para la capa de negocio será lo mismo utilizar cualquier implementación.

DAO intenta ir un poco más allá de la solución aportada por JDBC y es acoplar un poco más. Si con JDBC no estamos atados a un motor de base de datos, con DAO no estaremos más atados a las diferencias en la forma de guardar los datos. Al abstraer nuestra lógica de negocio de nuestra capa de persistencia estamos logrando con nuestro sistema pueda evolucionar de una manera mucho más sencilla.

Patrón DAO

Como sabemos, con JDBC tenemos la posibilidad de utilizar diferentes motores de base de datos, sin que nuestro código se vea afectado. Si respetamos los contratos, es decir, programamos siempre contra las interfaces y nunca contra las implementaciones, seremos inmunes a los cambios de driver. Con solo dos toques podremos abandonar MySQL y migrar a PostgreSQL, ya que los dos motores utilizan el estándar SQL.

Pero lamentablemente, no todos los proveedores respetan el estándar, esto significa que las consultas que estábamos usando en MySQL probablemente fallen en PostgreSQL, ya que hay detalles que cambian. Veamos cómo el patrón DAO nos ayudará a cambiar de motor de base de datos sin afectar la lógica principal de nuestro sistema.

Propósito y solución

PROPOSITO: Con el patrón DAO desacoplamos los datos propiamente dichos del lugar donde se almacenan o de la tecnología de almacenamiento. Es decir, para nuestro sistema será indiferente si utilizamos PostgreSQL o MySQL ya que en cualquiera de los dos casos la forma de comunicarse será la misma.

SOLUCIÓN: Este patrón nos propone:

1. Crear una interfaz en la que definamos todas las operaciones que queremos realizar, generalmente las más utilizadas son crear, actualizar, eliminar y leer.
2. Crear las diferentes implementaciones de esta interfaz, por ejemplo, una implementación para PostgreSQL y otra para MySQL.

De esta manera la capa de negocio, es decir donde se encuentra la lógica principal de nuestro sistema, se comunicará con la capa de persistencia, pero no conocerá los detalles de implementación ni se enterará si estamos utilizando PostgreSQL o MySQL ya que cualquiera sea la implementación, tendrá los mismos métodos que la interfaz.

Cuando hablamos de “capas”, ¿a qué nos referimos? En un sistema se denomina “**CAPA**” a una agrupación de componentes (clases) de iguales responsabilidades.

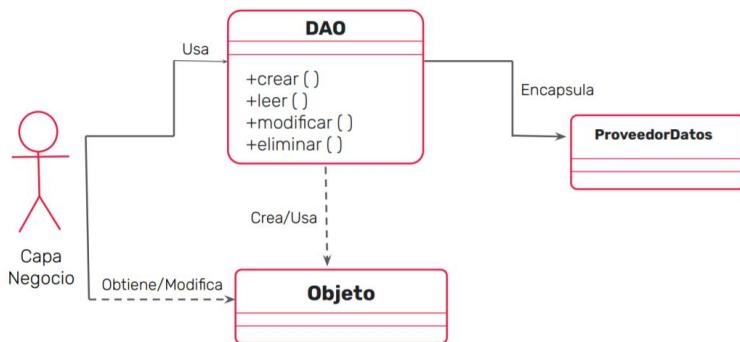
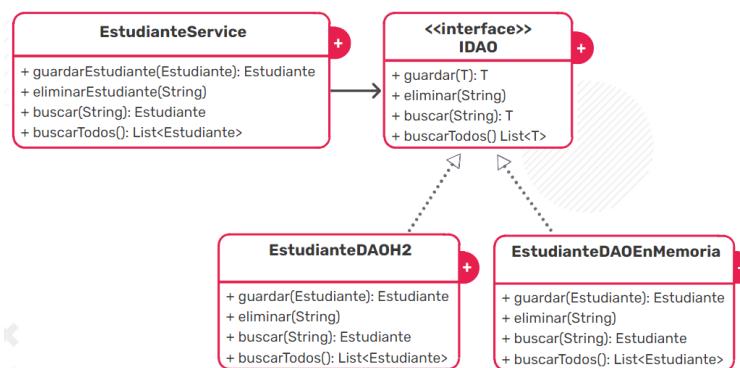


ILUSTRACIÓN 13: DIAGRAMA UML – PATRÓN DAO

El patrón DAO nos permite abstraer la lógica de negocio de nuestra capa de persistencia, logrando que el sistema sea mucho más fácil de evolucionar.

¿Cómo implementamos el patrón DAO?

Supongamos que estamos modelando un sistema para una academia y necesitamos realizar un ABM para los estudiantes. Queremos tener la posibilidad de cambiar la forma de persistir los estudiantes, sin afectar el funcionamiento de la capa de servicio.



Implementación del diagrama UML

En la **capa de negocio**, creamos un servicio, que utilice estas implementaciones. Como vemos, el servicio **EstudianteService** usa la interfaz **IDao** y le es indiferente cuál es la implementación que hay por detrás, ya que cualquiera que sea se comunicará de la misma forma.

```

public class EstudianteService {

    private IDao<Estudiante> estudianteDao;

    public EstudianteService() {
    }

    public EstudianteService( IDao<Estudiante> estudianteDao) {
        this.estudianteDao = estudianteDao;
    }

    public void setEstudianteDao( IDao<Estudiante> estudianteDao) {
        this.estudianteDao = estudianteDao;
    }

    public Estudiante guardarEstudiante(Estudiante estudiante) {
        estudianteDao.guardar(estudiante);
        return estudiante;
    }
    public List<Estudiante> buscarTodos() {
        return estudianteDao.buscarTodos();
    }

    public Estudiante buscar(String id){
        return estudianteDao.buscar(id);
    }

    public void eliminarEstudiante(String id){
        estudianteDao.eliminar(id);
    }
}

```

Definimos la **interfaz DAO** con las operaciones más comunes, que luego tendremos que implementar.

```

public interface IDao<T> {

    public T guardar(T t);
    public void eliminar(String id);
    public T buscar(String id);
    public List<T> buscarTodos();
}

```

Para evitar tener una interfaz por entidad (por ejemplo: IEstudianteDao, IMateriaDao), nos conviene definir una única interfaz y utilizar los genéricos de Java.

Después, debemos crear las implementaciones de esta interfaz, en nuestro caso crearemos dos, una para conectarnos a una base de datos H2 y otra para persistir en una Lista.

```

public class EstudianteDaoH2 implements IDao<Estudiante> {

    private ConfiguracionJDBC configuracionJDBC;

    public EstudianteDaoH2(ConfiguracionJDBC configuracionJDBC) {
        this.configuracionJDBC = configuracionJDBC;
    }

    @Override
    public Estudiante guardar(Estudiante estudiante) {
        Connection connection = configuracionJDBC.conectarConBaseDeDatos();
        Statement stmt = null;
        String query = String.format("INSERT INTO estudiantes
VALUES ('%s', '%s', '%s')", estudiante.getId(), estudiante.getNombre(), estudiante.getApellido());
        try {
            stmt = connection.createStatement();
            stmt.executeUpdate(query);
            stmt.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        return estudiante;
    }

    @Override
    public void eliminar(String id) {
        Connection connection = configuracionJDBC.conectarConBaseDeDatos();
        Statement stmt = null;
        String query = String.format("DELETE FROM estudiantes where id = %s", id);
        try {
            stmt = connection.createStatement();
            stmt.executeUpdate(query);
            stmt.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }
}

```

```

@Override
public Estudiante buscar(String id) {
    Connection connection = configuracionJDBC.conectarConBaseDeDatos();
    Statement stmt = null;
    String query = String.format("SELECT id,nombre,apellido  FROM estudiantes where id = '%s'", id);
    Estudiante estudiante = null;
    try {
        stmt = connection.createStatement();
        ResultSet result = stmt.executeQuery(query);
        while (result.next()) {
            String idEstudiante = result.getString("id");
            String nombre = result.getString("nombre");
            String apellido = result.getString("apellido");
            estudiante = new Estudiante(idEstudiante, nombre, apellido);
        }
        stmt.close();
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    return estudiante;
}

@Override
public List<Estudiante> buscarTodos() {
    Connection connection = configuracionJDBC.conectarConBaseDeDatos();
    Statement stmt = null;
    String query = "SELECT *  FROM estudiantes";
    List<Estudiante> estudiantes = new ArrayList<>();
    try {
        stmt = connection.createStatement();
        ResultSet result = stmt.executeQuery(query);
        while (result.next()) {
            String id = result.getString("id");
            String nombre = result.getString("nombre");
            String apellido = result.getString("apellido");
            estudiantes.add(new Estudiante(id, nombre, apellido));
        }
        stmt.close();
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    return estudiantes;
}

```

```

public class EstudianteDaoEnMemoria implements IDao<Estudiante> {
    private List<Estudiante> estudianteRepositorio;

    public EstudianteDaoEnMemoria(List<Estudiante> estudianteRepositorio) {
        this.estudianteRepositorio = estudianteRepositorio;
    }

    @Override
    public Estudiante guardar(Estudiante estudiante) {
        estudianteRepositorio.add(estudiante);
        return estudiante;
    }

    @Override
    public void eliminar(String id) {
        estudianteRepositorio.removeIf(estudiante -> estudiante.getId().equals(id));
    }

    //Utilizamos streams para buscar una materia por id, y tomamos el primero si hay, si no, devolvemos null.
    @Override
    public Estudiante buscar(String id) {
        return estudianteRepositorio.stream().filter(estudiante ->
estudiante.getId().equals(id)).findFirst().orElseGet(null);
    }

    @Override
    public List<Estudiante> buscarTodos() {
        return estudianteRepositorio;
    }
}

@Test
public void eliminarEstudianteTest() {
    estudianteDao.eliminar("1");
    Assert.assertEquals(estudianteDao.buscar("1"),null);
}
}

```

Implementación en JUnit

En primer lugar, creamos una instancia de EstudianteService, guardamos un estudiante, cambiamos la implementación por otra y volvemos a guardar otro estudiante.

```

private IDao<Estudiante> estudianteDaoEnMemoria = new EstudianteDaoEnMemoria(new ArrayList());
private IDao<Estudiante> estudianteDaoH2 = new EstudianteDaoH2(new ConfiguracionJDBC());
private EstudianteService estudianteService = new EstudianteService();

@Before
public void cargarEstudiantesCambiandoImplementacionDAO() {
    estudianteService.setEstudianteDao(estudianteDaoEnMemoria);
    estudianteService.guardarEstudiante(new Estudiante("1", "EstudianteUno", "ApellidoUno"));
    estudianteService.setEstudianteDao(estudianteDaoH2);
    estudianteService.guardarEstudiante(new Estudiante("2", "EstudianteDos", "ApellidoDos"));
}

```

Como podemos observar, independientemente de la implementación que estemos utilizando por detrás, siempre el servicio se comportará de la misma manera.

```

    @Test
    public void buscarEstudiantesCambiandoImplementacionDAO(){
        estudianteService.setEstudianteDao(estudianteDaoEnMemoria);
        Estudiante estudiante = estudianteService.buscar("1");
        Assert.assertEquals(estudiante.getId(),"1");
        Assert.assertEquals(estudiante.getNombre(),"EstudianteUno");
        Assert.assertEquals(estudiante.getApellido(),"ApellidoUno");
        estudianteService.setEstudianteDao(estudianteDaoH2);
        estudiante = estudianteService.buscar("2");
        Assert.assertEquals(estudiante.getId(),"2");
        Assert.assertEquals(estudiante.getNombre(),"EstudianteDos");
        Assert.assertEquals(estudiante.getApellido(),"ApellidoDos");

    }

```

Clase 18: Maven

Imaginemos que estamos trabajando en una aplicación y necesitamos enviarles notificaciones a las personas usuarias. Existen varias librerías que resuelven este paso, pero, ¿dónde las buscamos? ¿cómo las agregamos a nuestro proyecto? La herramienta Maven nos ayudará a incluir librerías automáticamente.

Pensemos unos minutos en el proceso de desarrollo de Software. Supongamos que ya tenemos una idea y queremos desarrollar una aplicación, abrimos nuestro IDE, comenzamos escribir la aplicación en Java y llega el momento en el que tenemos que agregar una funcionalidad para enviar notificaciones a diferentes dispositivos. Ante esta situación tenemos dos opciones:

1. Investigar cómo funcionan los diferentes sistemas operativos como Android y iOS, entender cómo comunicarnos y escribir el código para enviar notificaciones.
2. Buscar una librería que ya haya implementado la comunicación y solamente utilizarla.

Esta última opción es la más conveniente, como ya mencionamos, Maven nos ayudará a incluir librerías de forma automática. Es una herramienta de desarrollo de Software que automatiza varios procesos relacionados con la construcción de software. Uno de ellos, la gestión de dependencias.

Podemos arrancar a usarla definiendo un archivo llamado Proyect Object Model (POM), utilizando la estructura de nuestro proyecto como, por ejemplo, name, version, description y las librerías a utilizar llamadas dependencias.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6   https://maven.apache.org/xsd/maven-4.0.0.xsd">
7     <modelVersion>4.0.0</modelVersion>
8
9     <groupId>com.dh.mavenEjemplo</groupId>
10    <artifactId>maven-ejemplo</artifactId>
11    <version>0.0.1-SNAPSHOT</version>
12    <name>maven-ejemplo</name>
13    <description>Pom de ejemplo</description>
14    <properties>
15      <java.version>11</java.version>
16    </properties>
17

```

```

1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.12</version>
6     <scope>test</scope>
7   </dependency>
8   <dependency>
9     <groupId>com.h2database</groupId>
10    <artifactId>h2</artifactId>
11    <version>1.4.200</version>
12    <scope>test</scope>
13  </dependency>
14 </dependencies>
15 </project>
16
17

```

Y, además de ayudarnos en la gestión de dependencias, Maven nos ayudará en la ejecución de diferentes etapas del ciclo de vida del proyecto, desde la compilación, el testeo, el empaquetado, la instalación hasta el despliegue de nuestra aplicación.

¿A que nos referimos con esto? Maven nos ayuda a incluir librerías al proyecto, pero también nos guía en el desarrollo de nuestras propias librerías definiendo una serie de fases en nuestro proyecto que deberá completar para asegurar su correcto funcionamiento y despliegue. Dicho esto, veamos tres de las fases más importantes del ciclo de vida de nuestro desarrollo:

COMPILACIÓN: Maven compila nuestro código y convierten nuestros archivos .Java en .class

TEST: Luego de compilar se ejecutarán los tests unitarios que definimos en nuestra aplicación.

EMPAQUETADO: Una vez que tenemos el código compilado, que ya se encuentra testeado, Maven creará un archivo que contendrá todo el proyecto y podrá ser ejecutado en un servidor.

Utilizar Maven en nuestros proyectos nos permitirá centrarnos en lo que realmente nos importa: la lógica de la aplicación. Además, fomenta la reutilización de código y se encarga de la mayoría de las tareas relacionadas con la construcción de nuestro sistema.

Maven

Estamos trabajando en equipo en un sistema que se encargará del login de un sitio, necesitamos utilizar una base de datos para registrar los usuarios y sus datos. Buscamos la librería para conectarnos con MySQL y la agregamos a nuestro proyecto, escribimos el código necesario para guardar a los usuarios en la base de datos y le enviamos el código a todo el equipo.

Ahora, cuando alguien del equipo descarga el código con los últimos cambios e intenta correrlo este falla, ya que las librerías que se agregaron estaban en nuestra computadora. Por lo tanto, al querer buscarlas no las encontrará entonces, cada una de las personas del equipo, tendrá que repetir los pasos que hicimos nosotros: buscarla y agregarla. Bastante molesto ¿no? Además, al agregar las librerías para que el código funcione correctamente, uno de nuestros compañeros nos comenta que con los últimos cambios que hicimos empezó a fallar parte de la funcionalidad desarrollada con anterioridad. Por lo tanto, tendremos que descubrir en dónde está fallando y arreglarlo, provocando retrasos en el desarrollo.

¿Cómo nos puede ayudar Maven ante esta situación?

Utilizando Maven podremos agregar automáticamente las librerías al proyecto. Este se encargará de buscarlas y descargarlas.

De esta manera, cualquier persona que quiera ejecutar nuestro código, no necesitará buscar y agregar cada librería manualmente.

Por otro lado, se encargará de compilar nuestro código y ejecutar todos los tests que tengamos. Así tendremos la certeza de que a medida que vamos agregando código, el resto del proyecto sigue funcionando correctamente.

Apache Maven es una herramienta que nos ayudará en la construcción de proyectos JAVA, para la compilación, el testeo, el empaquetado y el despliegue de nuestro proyecto, ya sea en un servidor remoto o local. Y además automatizará la gestión de dependencias (librerías)

¿Cómo funciona?

Maven se centra en el concepto de archivos POM (Project Object Model). Un archivo POM es cómo representamos un proyecto ante Maven, este archivo contendrá toda la configuración mínima para que se ejecute correctamente. El archivo POM que creemos heredará del Super POM, de esta manera, nosotros solamente tendremos que agregar algunos detalles de configuración. ¡Veamos un ejemplo!

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.empresaproject-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.16</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.empresa.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.16</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>

```

GroupId: se utiliza para indicar el nombre de la organización o empresa y el equipo.

ArtifactId: se utiliza para asignarle un id al proyecto.

Versión: se utiliza para indicar la versión del proyecto.

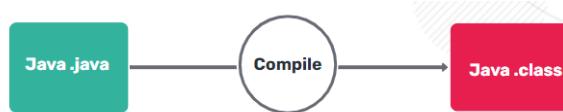


Ciclo de vida de un proyecto Maven

Maven define una serie de fases por la que pasará nuestro código. Las que más utilizaremos son:

MAVEN VALIDATE: Se encargará de validar que nuestro proyecto tiene toda la información necesaria para ser procesado.

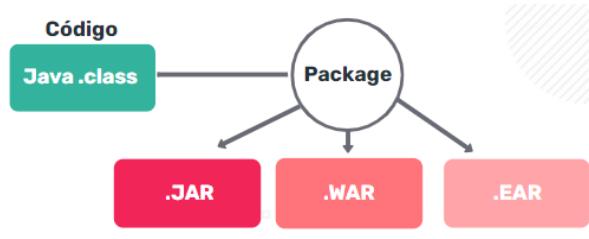
MAVEN COMPILE: Compilará nuestro código, transformando los archivos .java en .class y enviándolos a una carpeta de destino.



MAVEN TEST: Se ejecutarán los test unitarios que tengamos en el proyecto para asegurar el correcto funcionamiento del código que estamos produciendo.

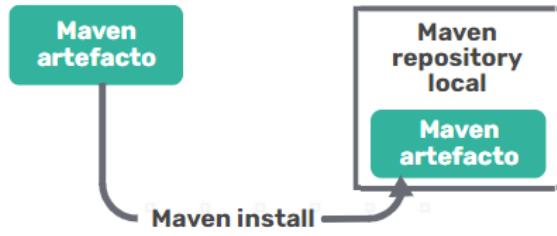


MAVEN PACKAGE: Empaquetará el proyecto en un formato estándar de Java que permitirá luego ser ejecutado. Los posibles formatos del empaquetado serán .jar, .war y .ear. Aunque el más utilizado es el .jar

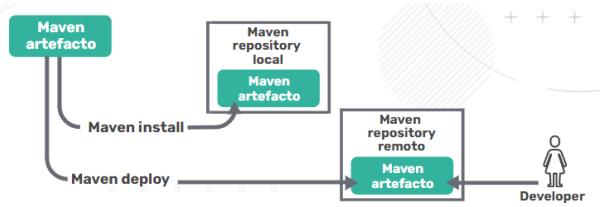


MAVEN VERIFY: Se ejecutarán los test de integración para asegurar el correcto funcionamiento del proyecto.

MAVEN INSTALL: Enviará nuestro proyecto a un repositorio local para que otros proyectos puedan hacer uso de él agregándolo a sus dependencias.



MAVEN DEPLOY: Enviará nuestro proyecto a un repositorio remoto para que otros desarrolladores puedan descargarlo y hacer uso de él agregándolo a sus dependencias.



Repositorios

Como vimos, en las fases de install y deploy, Maven envía nuestro proyecto a repositorios, ya sean locales o remotos.

Los repositorios son el lugar donde Maven irá a buscar las dependencias que nosotros agregamos en el POM.