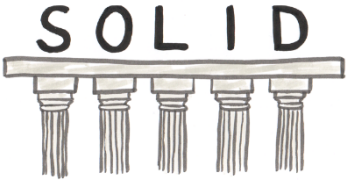
Anexo III: Principios SOLID y patrones de diseño.

Índice de contenidos

1 Principios SOLID.



Los principios SOLID de la POO son un **conjunto de buenas prácticas diseñadas para ayudar a escribir un código más limpio, mantenible y escalable**. Estos **cinco** principios **ayudan a comprender la importancia de ciertos patrones de diseño y la arquitectura de software en general**. No se consideran reglas absolutas, sino soluciones basadas en el sentido común y la experiencia para resolver problemas comunes en el desarrollo de software.

Los principios SOLID fueron introducidos por primera vez por el reconocido científico informático *Robert C. Martin*, conocido como "*Tío Bob*", en uno de sus artículos en el año 2000. Sin embargo, el acrónimo SOLID fue propuesto más tarde por *Michael Feathers* para agrupar estos conceptos de manera sencilla y memorable.

El **propósito principal** de los principios SOLID es **crear un código que sea comprensible, legible y comprobable, facilitando la colaboración entre varios desarrolladores y mejorando la calidad del software**. La **aplicación de estos principios está estrechamente relacionada con el uso de patrones de diseño**, ya que ayudan a mantener una alta cohesión y un bajo acoplamiento en el software, factores clave para un código sostenible y adaptable.

No obstante, **a pesar de su amplia aceptación, algunos profesionales critican los principios SOLID por considerarlos ambiguos y confusos**. Alegan que estos principios pueden complicar el código, ralentizar el proceso de desarrollo e incluso resultar innecesarios en algunos contextos. Estas críticas ponen en evidencia que, como toda metodología, los principios SOLID no son una solución universal, sino una guía que debe aplicarse con criterio según las necesidades específicas de cada proyecto.

A continuación, se exponen los **cinco principios que conforman SOLID**:

- **S – Single Responsibility Principle (SRP) - Principio de responsabilidad única.**

  Este principio **establece que una clase debe tener una única responsabilidad y, por lo tanto, una sola razón para cambiar**. En términos técnicos, solo un cambio específico en la lógica del software (por ejemplo, lógica de *BD*, lógica de registro, etc.) debería afectar a la clase. Es el principio más fundamental y a la vez el más difícil de aplicar en la práctica.

  Aplicar el SRP significa que si una clase actúa como un contenedor de datos, como una clase Libro o Estudiante, esta solo debería modificarse cuando se cambie el modelo de datos relacionado. Respetar este principio es crucial por dos razones principales:
  - En proyectos donde múltiples equipos trabajan en el mismo código, cada equipo podría modificar la misma clase por diferentes motivos, lo que podría derivar en módulos incompatibles.
  - El SRP facilita el control de versiones; por ejemplo, al ver cambios en una clase de persistencia que gestiona operaciones de *BD* en *GitHub*, se sabrá que los cambios están relacionados únicamente con la lógica de almacenamiento y no con otros aspectos del software.

- **O – Open/Closed Principle (OCP) - Principio abierto-cerrado.**

  El OCP **sostiene que las clases deben estar abiertas para la extensión pero cerradas para la modificación**. En otras palabras, se debe poder añadir nuevas funcionalidades sin alterar el código existente. Esto es especialmente importante en el desarrollo de clases, bibliotecas o frameworks, ya que modificar el código existente siempre conlleva el riesgo de introducir errores inesperados.

  La forma más común de implementar este principio es mediante el **uso de interfaces y clases abstractas**, que permiten agregar nuevas funciones a través de la extensión en lugar de la modificación directa del código original.

- **L – Liskov Substitution Principle (LSP) - Principio de sustitución de *Liskov*.**

  Propuesto por *Barbara Liskov*, este principio **establece que las subclases deben ser sustituibles por sus clases base sin alterar el comportamiento del programa**. Es decir, si la clase B es una subclase de la clase A, se debería poder usar un objeto de B en cualquier contexto que espere un objeto de A, sin causar resultados inesperados.

  La idea es que la subclase extiende el comportamiento de la superclase pero nunca lo reduce. No cumplir con este principio puede llevar a errores complicados de detectar, ya que la suposición subyacente de que las subclases actúan de manera coherente con sus superclases se rompe.

- **I – Interface Segregation Principle (ISP) - Principio de segregación de interfaz.**

Este principio **se enfoca en la necesidad de dividir las interfaces en unidades específicas**. La idea es que **muchas interfaces pequeñas y específicas** son preferibles a una interfaz general con muchas funcionalidades. No se debe obligar a los clientes a implementar métodos que no utilizan, ya que esto crea una dependencia innecesaria y complica el mantenimiento del código.

Por lo tanto, es preferible contar con interfaces que definan métodos específicos para cada propósito, reduciendo la sobrecarga de implementación innecesaria y facilitando la reutilización y evolución del código.

- **D – Dependency Inversion Principle (DIP) - Principio de inversión de dependencia.**

El DIP sugiere que **las clases deben depender de abstracciones, como interfaces o clases abstractas, en lugar de depender de implementaciones concretas**. Esto ayuda a reducir las dependencias directas entre módulos del código, promoviendo un diseño más flexible y de bajo acoplamiento.

El objetivo es lograr que los módulos de alto nivel no dependan de los módulos de bajo nivel, sino de abstracciones, permitiendo que los componentes del sistema evolucionen y se adapten con mayor facilidad sin causar efectos negativos en otras partes del software.

2 Patrones de diseño.

Los patrones de diseño son **soluciones habituales a problemas recurrentes en el diseño de software**. Funcionan como planos prefabricados que se pueden adaptar para resolver problemas específicos en el código. Sin embargo, **no se trata de fragmentos de código listos para copiar y pegar**, como si fueran funciones o bibliotecas ya creadas; un patrón **es un concepto general que describe cómo abordar un problema de diseño particular**.

A menudo, los patrones de diseño se confunden con algoritmos, ya que ambos proporcionan soluciones a problemas conocidos. Sin embargo, mientras que un algoritmo define una serie de pasos concretos para alcanzar un objetivo, un patrón de diseño es una descripción de nivel superior sobre cómo estructurar una solución. Por lo tanto, el código resultante de aplicar el mismo patrón puede variar significativamente de un programa a otro.

Una buena analogía sería comparar un algoritmo con una receta de cocina: ambos contienen pasos específicos para lograr un resultado. Por otro lado, un patrón de diseño se asemeja más a un plano arquitectónico: ofrece una visión general del resultado y de sus funciones, pero la implementación exacta depende del desarrollador y del contexto.

Aunque es posible trabajar durante años como programador sin conocer los patrones de diseño, muchas personas los aplican sin darse cuenta. Entonces, **¿por qué es importante aprenderlos?**:

- ✓ **Soluciones probadas y herramientas útiles**: los patrones de diseño son un conjunto de soluciones probadas a problemas comunes en el diseño de software. Aunque no siempre se enfrenten directamente esos problemas, conocer los patrones enseña a abordar diversas situaciones utilizando principios de diseño orientados a objetos.
- ✓ **Lenguaje común para el equipo**: los patrones de diseño establecen un lenguaje común que facilita la comunicación entre los desarrolladores. Por ejemplo, al decir "usa un singleton para eso", todos los miembros del equipo que conocen el patrón comprenderán la idea sin necesidad de explicaciones adicionales.

Clasificación de los patrones de diseño según su propósito:

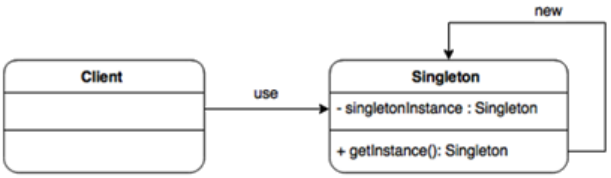
- ✓ **Creacionales**: se enfocan en los mecanismos de creación de objetos, incrementando la flexibilidad y la reutilización del código. Ejemplos: *Factory Method* y *Singleton*.
- ✓ **Estructurales**: describen cómo ensamblar objetos y clases en estructuras más complejas manteniendo la eficiencia y la flexibilidad. Ejemplos: *Adapter* y *Composite*.
- ✓ **De comportamiento**: se encargan de la comunicación efectiva y la asignación de responsabilidades entre objetos, mejorando la interacción y colaboración entre ellos. Ejemplos: *Observer* y *Strategy*.

Los patrones de diseño ofrecen soluciones sólidas y probadas para situaciones específicas, lo que facilita el desarrollo y mejora la mantenibilidad del código. Son una herramienta fundamental para cualquier desarrollador que busque crear software de calidad y trabajar de manera más eficiente en equipo.

2.1 Patrón Singleton.

El patrón Singleton es un **patrón de diseño creacional cuyo principal objetivo es garantizar que una clase tenga una única instancia durante toda la ejecución de la aplicación, y que esta instancia sea accesible de manera global**. Este patrón es útil cuando se necesita exactamente un objeto de una clase para coordinar acciones en todo el sistema.

Para lograr esto, el patrón restringe la creación de nuevas instancias de la clase mediante el operador `new`, imponiendo un constructor privado que impide la creación de instancias fuera de la propia clase. Adicionalmente, se implementa un método estático que gestiona la creación y el acceso a la única instancia disponible.

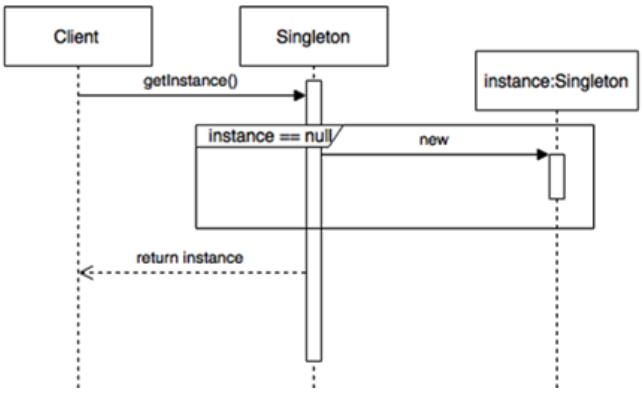


Los **componentes** que conforman el patrón son los siguientes:

- ✓ `Client`: es el componente que solicita obtener una instancia de la clase `Singleton`.
- ✓ `Singleton`: es la clase que implementa el patrón Singleton. Se asegura de que solo exista una única instancia de sí misma durante la vida de la aplicación.

Secuencia de pasos del patrón Singleton:

1. Solicitud de instancia: el cliente solicita la instancia del `Singleton` mediante el método estático `getInstance`.
2. Validación de existencia: el `Singleton` valida si la instancia ya fue creada previamente. Si no existe, se crea una nueva instancia.
3. Retorno de instancia: se devuelve la instancia recién creada o, en su defecto, la ya existente.



Ejemplo: un caso práctico del patrón Singleton es en la gestión de conexiones a una *BD*, asegurando que solo se cree una única conexión, sin importar cuántas veces se solicite desde diferentes partes del programa.

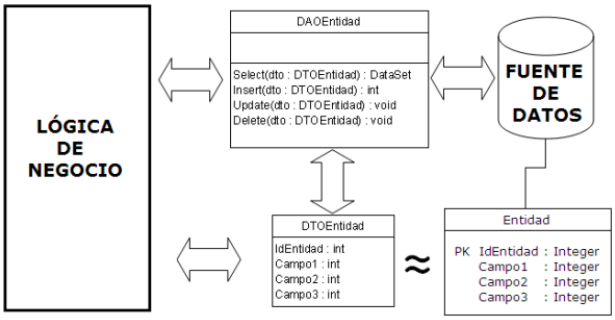
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DbConnection {
    private static DbConnection instance; // Instancia única de la clase
    private final Connection connection; // Conexión a la BD
    // Constructor privado que restringe la creación de instancias
    private DbConnection() throws SQLException, ClassNotFoundException {
        // Configuración de la conexión
        String jdbcDriver = "org.mariadb.jdbc.Driver";
        String dbUrl = "jdbc:mariadb://localhost:3306/base_datos";
        String user = "root";
        String password = "root";
        Class.forName(jdbcDriver); // Carga del driver JDBC
        // Establecimiento de la conexión
        this.connection = DriverManager.getConnection(dbUrl, user, password);
    }
    public Connection getConnection() { // Método para obtener la conexión a la base de datos
        return connection;
    }
    // Método estático para obtener la única instancia de DbConnection
    public static DbConnection getInstance() throws SQLException, ClassNotFoundException {
        // Verificación si la instancia ya existe o si la conexión está cerrada
        if (instance == null || instance.getConnection().isClosed())
            instance = new DbConnection();
        return instance;
    }
}
```

En este ejemplo, la clase `DbConnection` implementa el patrón Singleton para gestionar la conexión a la *BD*. El método `getInstance` se encarga de verificar si ya existe una instancia activa; en caso contrario, se crea una nueva. Esto garantiza que solo haya una instancia de la conexión activa, optimizando los recursos y asegurando consistencia en el acceso a la *BD*.

Ejemplo: implementación de un gestor de registro de logs (*Logger*) en una aplicación. El objetivo de un *Logger* Singleton es asegurarse de que todos los mensajes de registro se manejen a través de una única instancia, evitando la duplicación y garantizando que los logs no se pierdan ni se mezclen.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Logger {
    private static Logger instance; // Instancia única de la clase Logger
    private PrintWriter writer; // Archivo de log

    // Constructor privado que inicializa el archivo de log
    private Logger() throws IOException {
        // Configura el archivo de log con el nombre y la ruta
        String logFile = "app.log";
        writer = new PrintWriter(new FileWriter(logFile, true), true);
    }

    // Método estático para obtener la única instancia del Logger
    public static Logger getInstance() throws IOException {
        if (instance == null)
            instance = new Logger();

        return instance;
    }

    // Método para escribir mensajes en el log con la fecha y hora actual
    public void log(String message) {
        String timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());
        writer.println(timestamp + " - " + message);
    }

    // Método para cerrar el archivo de log al terminar la aplicación
    public void close() {
        writer.close();
    }
}
```

2.2 Patrón Data Access Object (*DAO*).

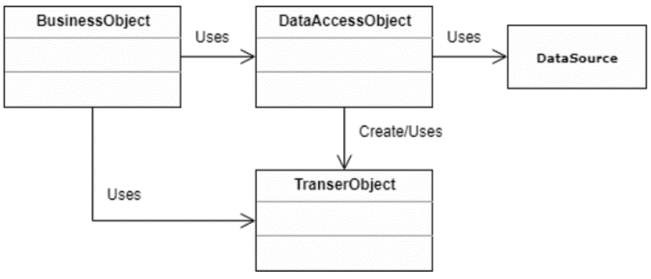


Las **aplicaciones modernas suelen necesitar acceder a diversas fuentes de datos**, como *BD* relacionales, archivos, servicios web, etc. Sin embargo, **la lógica de acceso a estos datos puede variar considerablemente según la fuente**, lo cual podría requerir constantes ajustes y refactorizaciones en el código de la aplicación. Para resolver este problema, se utiliza el **patrón Data Access Object (DAO)**, que **separa la lógica de acceso a datos de los objetos de negocio** (Business Objects), **encapsulando así toda la interacción con las fuentes de datos**.

El **uso del patrón DAO** aporta varios **beneficios**:

- **Desacoplamiento**: separa la lógica de negocio de la lógica de acceso a datos, facilitando la evolución y el mantenimiento del software.
- **Flexibilidad**: permite cambiar la fuente de datos (*BD* relacional, NoSQL, XML, archivos planos, servicios REST, etc.) sin necesidad de modificar la lógica de negocio.
- **Reutilización y mantenimiento**: centraliza las operaciones CRUD, lo que facilita su reutilización y reduce la duplicación de código.

Los **principales componentes** que conforman este patrón son:

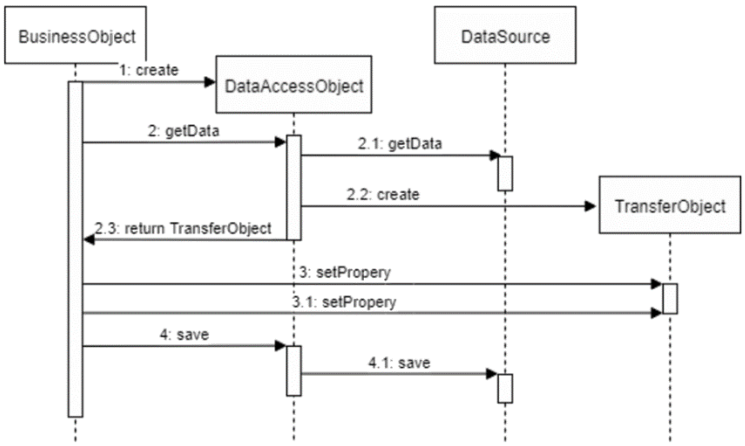


- ✓ **BusinessObject** (Cliente): objeto que necesita acceder a los datos para realizar su lógica de negocio. No interactúa directamente con la fuente de datos, sino a través del DAO.
- ✓ **DataAccessObject** (DAO): encapsula la lógica de acceso a datos, proporcionando una interfaz que oculta la

complejidad y los detalles técnicos de la fuente de datos.

- ✓ **TransferObject** (DTO - Data Transfer Object): objeto que transporta los datos entre el DAO y el BusinessObject. Representa una entidad de dominio.
- ✓ **DataSource**: fuente de datos abstracta, que puede ser una *BD*, servicio web, archivo, etc.

El patrón **funciona de la siguiente manera**:



1. El BusinessObject crea u obtiene una referencia al DataAccessObject.
2. Para solicitar información:
   - El DAO consulta la fuente de datos (DataSource).
   - Recupera los datos y los encapsula en un TransferObject.
   - Devuelve el TransferObject al BusinessObject.
3. El BusinessObject puede modificar los datos del TransferObject según la lógica de negocio.
4. Para guardar los cambios:
   - El BusinessObject envía el TransferObject actualizado al DAO.
   - El DAO actualiza la fuente de datos con la información modificada.

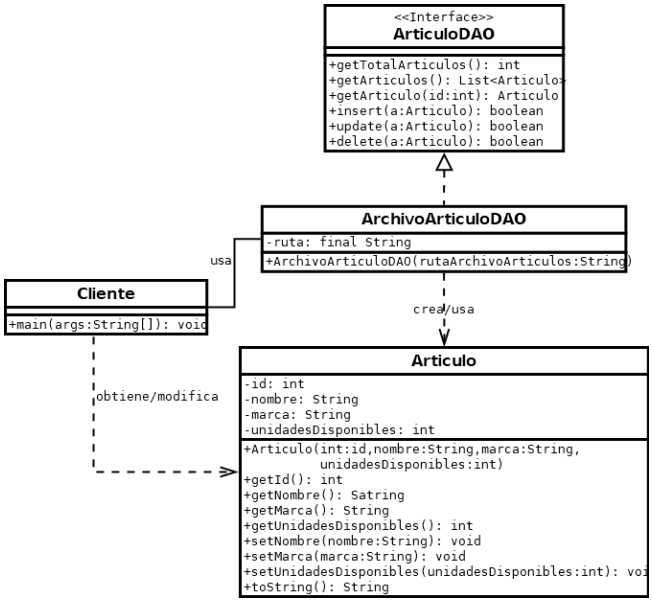
Este esquema permite que la lógica de negocio no se preocupe por los detalles técnicos del acceso a los datos, lo cual facilita el mantenimiento y escalabilidad del sistema.

La **implementación del patrón DAO** sigue estos **pasos**:

1. **Definir entidades**: crear las clases que representen los objetos de dominio, con propiedades privadas, métodos getters y setters.
2. **Crear la interfaz DAO**: definir una interfaz que especifique las operaciones CRUD para cada entidad.
3. **Gestionar conexiones (opcional)**: implementar patrones como Singleton para gestionar conexiones a *BD*.
4. **Implementar el DAO concreto**: crear la clase que implementa la interfaz DAO, interactuando con la fuente de datos específica.
5. **Configurar una factoría (opcional)**: crear una factoría para obtener instancias del DAO, permitiendo cambiar la fuente de datos sin modificar el código.
6. **Usar DAO en la lógica de negocio**: utilizar los DAOs para las operaciones de acceso a datos, delegando esta responsabilidad para mantener la lógica de negocio enfocada en tareas de alto nivel.

Ejemplo: manejo de artículos con archivos binarios.

Se desarrolla una aplicación para gestionar los artículos de una tienda de informática, almacenados en un archivo binario `tienda_pc.dat`. El archivo no usa serialización y contiene los datos de los artículos en el siguiente orden: `ID`, `nombre`, `marca` y `unidades disponibles`. Se implementa una clase `ArchivoArticuloDAO` que realiza las operaciones de inserción, actualización, eliminación y consulta sobre el archivo.



Los métodos actuarán así:

- Constructor: inicializa la propiedad `ruta` con la ruta donde está el archivo `tienda_pc.dat`. Si no existe el archivo, lo crea y escribe un `0` como señal de que tiene 0 artículos.
- `getTotalArticulos`: devuelve el número de artículos, buscando ese dato en el archivo.
- `getArticulos`: devuelve una lista con todos los objetos `Articulo` que hay en el archivo.
- `insert`: recibe un artículo y, si no existe en el archivo, lo añade al final del mismo.
- `update`: recibe un artículo que se supone es la versión actualizada de un artículo del archivo que tiene ese

mismo *id*. El método reescribe en el archivo el objeto (artículo) que tiene ese `id`.

- `delete`: recibe un artículo y, si existe en el archivo lo borra y devuelve `true`.

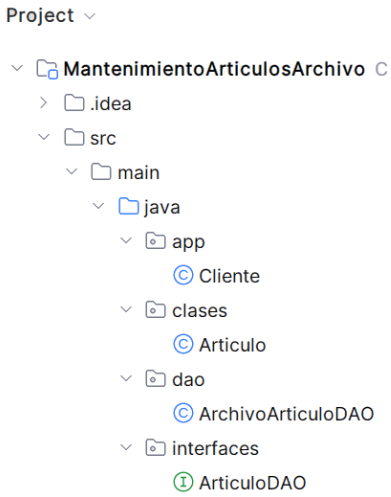
Es una buena costumbre crear una interfaz llamada `ArticuloDAO` que reúna todos los métodos del DAO y hacer que `ArchivoArticuloDAO` implemente dicha interfaz. Así, se podrá sustituir o mejorar el DAO sin cambiar el programa.

Características de la clase `ArchivoArticuloDAO`:

- Bajo consumo de memoria RAM: los artículos no se almacenan en memoria; se leen directamente del archivo cuando se necesitan.
- Alto uso de disco y CPU: debido a que cada acceso requiere buscar en el disco, la operación puede ser lenta.
- Modo desconectado: el archivo se abre solo cuando se necesita y se cierra después de cada operación.
- No es thread-safe: no soporta acceso concurrente.

➢ `Articulo.java`

```java
package clases;

import java.io.Serializable;

public class Articulo implements Serializable {
    private int id;
    private String nombre;
    private String marca;
    private int unidadesDisponibles;

    // Constructor por defecto
    public Articulo() {
    }

    // Constructor con parámetros
    public Articulo(int id, String nombre, String marca, int unidadesDisponibles) {
        this.id = id;
        this.nombre = nombre;
        this.marca = marca;
        this.unidadesDisponibles = unidadesDisponibles;
    }

    // Getters y Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getMarca() {
        return marca;
    }

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

    public int getUnidadesDisponibles() {
        return unidadesDisponibles;
    }

    public void setUnidadesDisponibles(int unidadesDisponibles) {
        this.unidadesDisponibles = unidadesDisponibles;
    }

    @Override
    public String toString() {
        return String.format("Articulo{id=%d, nombre='%s', marca='%s', unidadesDisponibles=%d}",
                id, nombre, marca, unidadesDisponibles);
    }
}
```



- ArticuloDAO.java

```java
package interfaces;

import clases.Articulo;
import java.util.List;

// Interfaz que define las operaciones básicas sobre los artículos
public interface ArticuloDAO {
    /**
     * Obtiene el total de artículos disponibles.
     *
     * @return Número total de artículos.
     * @throws Exception Si ocurre un error al obtener los datos.
     */
    int getTotalArticulos() throws Exception;

    /**
     * Obtiene una lista de todos los artículos.
     *
     * @return Lista de objetos Articulo.
     * @throws Exception Si ocurre un error al obtener los datos.
     */
    List<Articulo> getArticulos() throws Exception;

    /**
     * Obtiene un artículo por su ID.
     *
     * @param id ID del artículo a buscar.
     * @return El artículo correspondiente al ID proporcionado.
     * @throws Exception Si ocurre un error al obtener los datos.
     */
    Articulo getArticulo(int id) throws Exception;

    /**
     * Inserta un nuevo artículo.
     *
     * @param articulo Artículo a insertar.
     * @return true si la inserción fue exitosa, false en caso contrario.
     * @throws Exception Si ocurre un error al insertar los datos.
     */
    boolean insert(Articulo articulo) throws Exception;

    /**
     * Actualiza un artículo existente.
     *
     * @param articulo Artículo a actualizar.
     * @return true si la actualización fue exitosa, false en caso contrario.
     * @throws Exception Si ocurre un error al actualizar los datos.
     */
    boolean update(Articulo articulo) throws Exception;

    /**
     * Elimina un artículo.
     *
     * @param articulo Artículo a eliminar.
     * @return true si la eliminación fue exitosa, false en caso contrario.
     * @throws Exception Si ocurre un error al eliminar los datos.
     */
    boolean delete(Articulo articulo) throws Exception;
}
```

- ArchivoArticuloDAO.java

```java
package dao;

import clases.Articulo;
import interfaces.ArticuloDAO;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;

public class ArchivoArticuloDAO implements ArticuloDAO {
    private final String ruta;

    // Constructor que verifica la existencia del archivo y lo crea si no existe.
```

```
public ArchivoArticuloDAO(String rutaArchivoArticulos) throws IOException {
    ruta = rutaArchivoArticulos;
    Path path = Paths.get(ruta);
    if (!Files.exists(path)) {
        try (RandomAccessFile file = new RandomAccessFile(ruta, "rw")) {
            file.writeInt(0); // Inicializa el archivo con 0 artículos.
        }
    }
}

// Obtiene el total de artículos.
@Override
public int getTotalArticulos() throws IOException {
    try (RandomAccessFile file = new RandomAccessFile(ruta, "r")) {
        return file.readInt();
    }
}

// Obtiene todos los artículos del archivo.
@Override
public List<Articulo> getArticulos() throws IOException {
    List<Articulo> lista = new ArrayList<>();
    try (RandomAccessFile file = new RandomAccessFile(ruta, "r")) {
        int total = file.readInt();
        for (int i = 0; i < total; i++) {
            int id = file.readInt();
            String nombre = file.readUTF();
            String marca = file.readUTF();
            int unidadesDisponibles = file.readInt();
            lista.add(new Articulo(id, nombre, marca, unidadesDisponibles));
        }
    }
    return lista;
}

// Obtiene un artículo por su ID.
@Override
public Articulo getArticulo(int id) throws IOException {
    try (RandomAccessFile file = new RandomAccessFile(ruta, "r")) {
        int total = file.readInt();
        for (int i = 0; i < total; i++) {
            int idLeido = file.readInt();
            String nombre = file.readUTF();
            String marca = file.readUTF();
            int unidadesDisponibles = file.readInt();
            if (id == idLeido)
                return new Articulo(id, nombre, marca, unidadesDisponibles);
        }
    }
    return null; // Retorna null si no encuentra el artículo.
}

// Inserta un nuevo artículo si no existe.
@Override
public boolean insert(Articulo articulo) throws IOException {
    try (RandomAccessFile file = new RandomAccessFile(ruta, "rw")) {
        int total = file.readInt();
        for (int i = 0; i < total; i++) {
            int id = file.readInt();
            file.readUTF(); // Nombre
            file.readUTF(); // Marca
            file.readInt(); // Unidades
            if (id == articulo.getId())
                return false; // El artículo ya existe.
        }
        // Añade el nuevo artículo.
        file.seek(file.length());
        file.writeInt(articulo.getId());
        file.writeUTF(articulo.getNombre());
        file.writeUTF(articulo.getMarca());
        file.writeInt(articulo.getUnidadesDisponibles());
        file.seek(0);
        file.writeInt(total + 1); // Actualiza el total de artículos.
        return true;
```

```
        }
    }

    // Actualiza un artículo existente.
    @Override
    public boolean update(Articulo articulo) throws IOException {
        Path tempFilePath = Files.createTempFile("articulo", ".tmp");
        boolean actualizado = false;

        try (RandomAccessFile file = new RandomAccessFile(ruta, "r");
                RandomAccessFile tempFile = new RandomAccessFile(tempFilePath.toFile(), "rw")) {
            int total = file.readInt();
            tempFile.writeInt(total);
            for (int i = 0; i < total; i++) {
                int id = file.readInt();
                String nombre = file.readUTF();
                String marca = file.readUTF();
                int unidadesDisponibles = file.readInt();

                if (id == articulo.getId()) {
                    // Actualiza el artículo.
                    tempFile.writeInt(articulo.getId());
                    tempFile.writeUTF(articulo.getNombre());
                    tempFile.writeUTF(articulo.getMarca());
                    tempFile.writeInt(articulo.getUnidadesDisponibles());
                    actualizado = true;
                } else {
                    // Copia el artículo original.
                    tempFile.writeInt(id);
                    tempFile.writeUTF(nombre);
                    tempFile.writeUTF(marca);
                    tempFile.writeInt(unidadesDisponibles);
                }
            }
        }

        if (actualizado) {
            Files.delete(Paths.get(ruta));
            Files.move(tempFilePath, Paths.get(ruta));
        } else
            Files.delete(tempFilePath);

        return actualizado;
    }

    // Elimina un artículo.
    @Override
    public boolean delete(Articulo articulo) throws IOException {
        Path tempFilePath = Files.createTempFile("articulo", ".tmp");
        boolean eliminado = false;

        try (RandomAccessFile file = new RandomAccessFile(ruta, "r");
                RandomAccessFile tempFile = new RandomAccessFile(tempFilePath.toFile(), "rw")) {
            int total = file.readInt();
            tempFile.writeInt(total - 1); // Asume que el artículo se eliminará.

            for (int i = 0; i < total; i++) {
                int id = file.readInt();
                String nombre = file.readUTF();
                String marca = file.readUTF();
                int unidadesDisponibles = file.readInt();

                if (id != articulo.getId()) {
                    tempFile.writeInt(id);
                    tempFile.writeUTF(nombre);
                    tempFile.writeUTF(marca);
                    tempFile.writeInt(unidadesDisponibles);
                } else
                    eliminado = true; // Indica que se encontró y se eliminará.
            }
        }

        if (eliminado) {
            Files.delete(Paths.get(ruta));
            Files.move(tempFilePath, Paths.get(ruta));
        } else
            Files.delete(tempFilePath);
```

```
            return eliminado;
        }
    }
```

- Cliente.java

```
package app;

import clases.Articulo;
import dao.ArchivoArticuloDAO;
import interfaces.ArticuloDAO;
import java.util.List;

public class Cliente {
    public static void main(String[] args) {
        try {
            ArticuloDAO dao = new ArchivoArticuloDAO("tienda_pc.dat");

            System.out.println("\n-----------------------------------------------------------");
            System.out.println("Insertando tres artículos: ");
            System.out.println("-----------------------------------------------------------");
            insertarArticulo(dao, new Articulo(1, "Ratón óptico Logitech G203", "Logitech", 10));
            insertarArticulo(dao, new Articulo(2, "Keychron K5 Pro Wireless", "Keychron", 13));
            insertarArticulo(dao, new Articulo(3, "MacBook Pro M2 PRO", "Apple", 3));

            System.out.println("\n-----------------------------------------------------------");
            System.out.println("Listado de artículos antes de modificar el nombre del 2:");
            System.out.println("-----------------------------------------------------------");
            listarArticulos(dao);

            // Obtener y modificar el artículo con id 2
            modificarArticulo(dao, 2, "Teclado Keychron K5 Pro Wireless");

            System.out.println("\n-----------------------------------------------------------");
            System.out.println("Listado de artículos tras modificar el nombre del 2:");
            System.out.println("-----------------------------------------------------------");
            listarArticulos(dao);

            // Borrar un artículo por id
            eliminarArticulo(dao, 2);

            System.out.println("\n---------------------------------------");
            System.out.println("Listado de artículos tras eliminar el 2:");
            System.out.println("---------------------------------------");
            listarArticulos(dao);

            System.out.println("\n---------------------------------------");
            System.out.printf("Total de artículos: %d%n", dao.getTotalArticulos());
            System.out.println("---------------------------------------");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace(); // Opcional: para más detalles sobre el error
        }
    }

    // Método para insertar un artículo
    private static void insertarArticulo(ArticuloDAO dao, Articulo articulo) {
        try {
            if (dao.insert(articulo))
                System.out.println("Insertado artículo: " + articulo.getNombre());
            else
                System.out.println("El artículo ya existe: " + articulo.getNombre());
        } catch (Exception e) {
            System.out.println("Error al insertar artículo: " + e.getMessage());
        }
    }

    // Método para listar todos los artículos
    private static void listarArticulos(ArticuloDAO dao) {
        try {
            List<Articulo> lista = dao.getArticulos();
            for (Articulo articulo : lista)
                System.out.println(articulo);
        } catch (Exception e) {
            System.out.println("Error al listar artículos: " + e.getMessage());
        }
    }

    // Método para modificar un artículo
    private static void modificarArticulo(ArticuloDAO dao, int id, String nuevoNombre) {
```

```
            try {
                Articulo articulo = dao.getArticulo(id);
                if (articulo != null) {
                    articulo.setNombre(nuevoNombre);
                    if (dao.update(articulo))
                        System.out.println("Modificado artículo: " + articulo.getNombre());
                } else
                    System.out.println("Artículo no encontrado para modificar, id: " + id);
            } catch (Exception e) {
                System.out.println("Error al modificar artículo: " + e.getMessage());
            }
        }

        // Método para eliminar un artículo por id
        private static void eliminarArticulo(ArticuloDAO dao, int id) {
            try {
                Articulo articulo = new Articulo(id, null, null, 0);
                if (dao.delete(articulo))
                    System.out.println("Eliminado artículo con id " + id + ".");
                else
                    System.out.println("No se encontró el artículo para eliminar, id: " + id);
            } catch (Exception e) {
                System.out.println("Error al eliminar artículo: " + e.getMessage());
            }
        }
    }
```

Ejemplo: gestión de artículos de una tienda de informática usando una *BD* relacional.

Para gestionar los productos de una tienda de informática utilizando una *BD* relacional, basta con modificar el DAO actual (`ArchivoArticuloDAO`), que trabaja con archivos, y reemplazarlo por uno que opere con *BD* relacionales, como `BdArticuloDAO`. En la clase cliente, simplemente cambiar la instancia del DAO por la nueva (esto se puede resolver usando un patrón Factory, como se explica en el siguiente apartado).

Al trabajar con una *BD* relacional, es necesario manejar la conexión correspondiente.

- Crear la *BD* y la tabla correspondiente (por ejemplo con *MariaDB*).

```
DROP USER IF EXISTS articulos;
CREATE USER articulos IDENTIFIED BY "articulos";
CREATE OR REPLACE DATABASE articulos;
GRANT ALL PRIVILEGES ON articulos.* TO articulos;
USE articulos;
CREATE TABLE articulo (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL UNIQUE,
    marca VARCHAR(50) NOT NULL,
    stock INT UNSIGNED NOT NULL
);
```

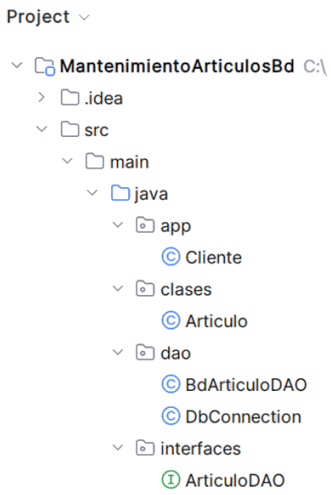
- Agregar dependencias al proyecto (ejemplo de dependencias en el caso de un proyecto *Maven*).

```
<dependencies>
    <dependency>
        <groupId>com.googlecode.sli4j</groupId>
        <artifactId>sli4j-slf4j-simple</artifactId>
        <version>2.0</version>
    </dependency>
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
        <version>3.3.2</version>
    </dependency>
</dependencies>
```

- `DbConnection.java` (ver código de ejemplo del punto 2.1) → Modificar el nombre de la *BD*, usuario y clave.
- `Articulo.java` (ver código de ejemplo anterior).
- `ArticuloDAO.java` (ver código de ejemplo anterior).
- `BdArticuloDAO.java`

```
package dao;

import clases.Articulo;
import interfaces.ArticuloDAO;
import java.sql.Connection;
```

```java
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

public class BdArticuloDAO implements ArticuloDAO {
    @Override
    public int getTotalArticulos() throws Exception {
        String query = "SELECT COUNT(*) AS n_articulos FROM articulo";
        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query);
                ResultSet rs = st.executeQuery()) {

            if (rs.next())
                return rs.getInt("n_articulos");
        } catch (Exception e) {
            System.err.println("Error al obtener el total de artículos: " + e.getMessage());
            throw e;
        }
        return 0; // Si no se encuentra ningún resultado
    }

    @Override
    public List<Articulo> getArticulos() throws Exception {
        List<Articulo> listaArticulos = new ArrayList<>();
        String query = "SELECT * FROM articulo";

        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query);
                ResultSet rs = st.executeQuery()) {

            while (rs.next()) {
                Articulo art = new Articulo();
                art.setId(rs.getInt("id"));
                art.setNombre(rs.getString("nombre"));
                art.setMarca(rs.getString("marca"));
                art.setUnidadesDisponibles(rs.getInt("stock"));
                listaArticulos.add(art);
            }
        } catch (Exception e) {
            System.err.println("Error al listar artículos: " + e.getMessage());
            throw e;
        }
        return listaArticulos;
    }

    @Override
    public Articulo getArticulo(int id) throws Exception {
        Articulo articulo = null;
        String query = "SELECT * FROM articulo WHERE id = ?";

        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query)) {
            st.setInt(1, id);
            try (ResultSet rs = st.executeQuery()) {
                if (rs.next()) {
                    articulo = new Articulo();
                    articulo.setId(rs.getInt("id"));
                    articulo.setNombre(rs.getString("nombre"));
                    articulo.setMarca(rs.getString("marca"));
                    articulo.setUnidadesDisponibles(rs.getInt("stock"));
                }
            }
        } catch (Exception e) {
            System.err.println("Error al obtener artículo con id " + id + ": " + e.getMessage());
            throw e;
        }

        return articulo;
    }

    @Override
    public boolean insert(Articulo a) throws Exception {
        String query = "INSERT INTO articulo(nombre, marca, stock) VALUES(?, ?, ?)";

        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query)) {
```



```
            st.setString(1, a.getNombre());
            st.setString(2, a.getMarca());
            st.setInt(3, a.getUnidadesDisponibles());
            int rowsAffected = st.executeUpdate();
            return rowsAffected > 0; // Retorna true si se insertó al menos un registro
        } catch (Exception e) {
            System.err.println("Error al insertar artículo: " + e.getMessage());
            throw e;
        }
    }
    @Override
    public boolean update(Articulo a) throws Exception {
        String query = "UPDATE articulo SET nombre = ?, marca = ?, stock = ? WHERE id = ?";
        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query)) {
            st.setString(1, a.getNombre());
            st.setString(2, a.getMarca());
            st.setInt(3, a.getUnidadesDisponibles());
            st.setInt(4, a.getId());
            int rowsAffected = st.executeUpdate();
            return rowsAffected > 0; // Retorna true si se actualizó al menos un registro
        } catch (Exception e) {
            System.err.println("Error al actualizar artículo con id " + a.getId()
                                                                    + ": " + e.getMessage());
            throw e;
        }
    }
    @Override
    public boolean delete(Articulo a) throws Exception {
        String query = "DELETE FROM articulo WHERE id = ?";
        try (Connection con = DbConnection.getInstance().getConnection();
                PreparedStatement st = con.prepareStatement(query)) {
            st.setInt(1, a.getId());
            int rowsAffected = st.executeUpdate();
            return rowsAffected > 0; // Retorna true si se eliminó al menos un registro
        } catch (Exception e) {
            System.err.println("Error al eliminar artículo con id " + a.getId() + ": "
                                                                + e.getMessage());
            throw e;
        }
    }
}
```

- `Cliente.java` (ver código de ejemplo anterior) → Cambiar la siguiente línea

  `ArticuloDAO dao = new ArchivoArticuloDAO("tienda_pc.dat");`

  por `ArticuloDAO dao = new BdArticuloDAO();`

  y agregar la importación de la clase `import dao.BdArticuloDAO;`

2.3 Patrón Repository.

El patrón Repository es un patrón de diseño de software que **se utiliza comúnmente en el desarrollo de aplicaciones para separar la lógica de negocio de la lógica de acceso a datos**. Este patrón establece una capa intermedia entre la lógica de la aplicación y el sistema de almacenamiento de datos, proporcionando una interfaz uniforme para acceder a los datos, independientemente de su almacenamiento o recuperación. A diferencia del patrón DAO (Data Access Object), el patrón Repository **tiende a ofrecer una interfaz más rica que incluye operaciones específicas del dominio, en lugar de limitarse a las operaciones CRUD** (Crear, Leer, Actualizar, Eliminar) **básicas**.

Elección entre Repository y DAO:

- ✓ **Repository**: se elige cuando **se busca una abstracción más rica y se desea desacoplar completamente la lógica de negocio de la capa de persistencia**. El patrón Repository se enfoca en operaciones de alto nivel y en emular una colección de objetos, proporcionando una interfaz más orientada al dominio y a las necesidades específicas del negocio.
- ✓ **DAO**: se elige cuando **la simplicidad y la eficiencia en las operaciones CRUD son más importantes y la abstracción de alto nivel no es una prioridad**. El patrón DAO se centra en la manipulación directa de datos

y suele ser adecuado para aplicaciones donde se requiere una interfaz más sencilla y directa con la capa de persistencia.

El patrón Repository facilita una abstracción más rica, mientras que el patrón DAO proporciona una interfaz más cercana al almacenamiento de datos.

Ejemplo: clase `ArticuloRepository` que usará `ArticuloDAO`. Ofrece al programa las siguientes funcionalidades:

- Grabar un artículo nuevo → Esto ya lo hace el DAO.
- Actualizar las unidades disponibles de un artículo.
- Borrar un artículo → Esto ya lo hace el DAO.
- Consultar los datos de un artículo individual a partir de su *id*.
- Consultar todos los artículos que hay en la tienda → Esto ya lo hace el DAO.
- Consultar todos los artículos que tengan agotadas sus existencias.

Hay tres cosas que ya las hace el DAO. Aun así, el Repository también deberá permitir hacerlas, porque la idea es que el programa principal ya no va a usar el DAO, sino que el programa usa el Repository.

```
package repository;

import clases.Articulo;
import interfaces.ArticuloDAO;
import java.util.ArrayList;
import java.util.List;

public class ArticuloRepository {
    private final ArticuloDAO dao; // ArticuloDAO es una interfaz (no está ligado a archivos ni BD)

    public ArticuloRepository(ArticuloDAO dao) {
        this.dao = dao;
    }

    public boolean insert(Articulo a) throws Exception {
        return dao.insert(a); // Delega en el DAO
    }

    public boolean delete(Articulo a) throws Exception {
        return dao.delete(a); // Delega en el DAO
    }

    public boolean update(Articulo a) throws Exception {
        return dao.update(a); // Delega en el DAO
    }

    public List<Articulo> getArticulos() throws Exception {
        return dao.getArticulos(); // Delega en el DAO
    }

    public Articulo getArticulo(int id) throws Exception {
        return dao.getArticulo(id); // Delega en el DAO
    }

    public boolean modificarUnidades(Articulo a, int unidades) throws Exception {
        Articulo articuloModificado = new Articulo(
                a.getId(), // El artículo modificado conserva id, nombre y marca
                a.getNombre(),
                a.getMarca(),
                unidades // El artículo modificado tiene las unidades actualizadas
        );
        return dao.update(articuloModificado); // Usa el DAO para actualizar el artículo
    }

    public List<Articulo> getArticulosSinStock() throws Exception {
        List<Articulo> lista = new ArrayList<>();
        for (Articulo a : dao.getArticulos()) {
            // Añade a la lista si el artículo recorrido no tiene unidades
            if (a.getUnidadesDisponibles() == 0)
                lista.add(a);
        }
        return lista;
    }
}
```

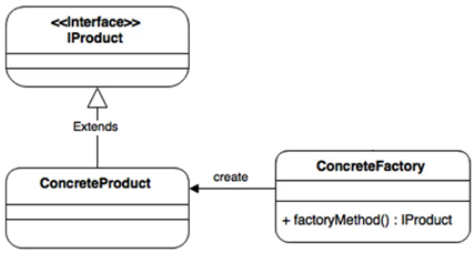
Como se puede observar en el ejemplo, la clase `ArticuloDAO` proporciona métodos para operaciones básicas de inserción, consulta, borrado y modificación de objetos Articulo. Sin embargo, los requisitos del programa pueden ser ligeramente diferentes. Por esta razón, el patrón Repository se utiliza para "subir de nivel" y ofrecer al programa los métodos específicos que necesita, separando así la lógica de negocio de la lógica de acceso a datos y proporcionando una interfaz más adecuada para el dominio.

2.4 Patrón Factory.

El patrón Factory es un patrón de diseño que **proporciona una interfaz para crear objetos de una clase específica sin exponer la lógica de creación al cliente**. Este patrón es particularmente útil cuando no se conoce, en tiempo de diseño, el tipo exacto de objeto que se va a crear, o cuando se desea delegar la responsabilidad de la creación de objetos a una clase Factory. Al utilizar el patrón Factory, **se pueden instanciar objetos de manera dinámica en función de la configuración establecida** en archivos de texto, XML, propiedades, etc.

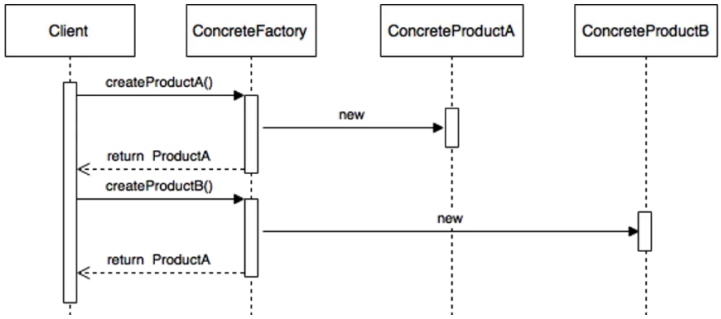
Componentes del patrón Factory:

- ✓ **IProduct**: representa de manera abstracta el objeto que se desea crear. A través de esta interfaz, se define la estructura que tendrán los objetos creados.
- ✓ **ConcreteProduct**: representa una implementación concreta de la interfaz IProduct. Este tipo de producto se crea a través de una ConcreteFactory.
- ✓ **ConcreteFactory**: representa una fábrica concreta utilizada para la creación de instancias de ConcreteProduct.

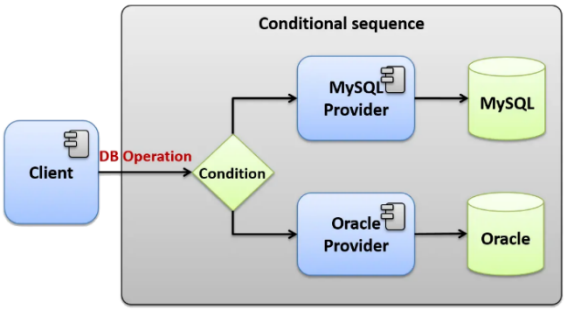


Funcionamiento del patrón Factory:

1. El cliente solicita a ConcreteFactory la creación de un ProductA.
2. ConcreteFactory localiza la implementación concreta de ProductA y crea una nueva instancia.
3. ConcreteFactory devuelve el ConcreteProductA creado.
4. El cliente solicita a ConcreteFactory la creación de un ProductB.
5. ConcreteFactory localiza la implementación concreta de ProductB y crea una nueva instancia.
6. ConcreteFactory devuelve el ConcreteProductB creado.

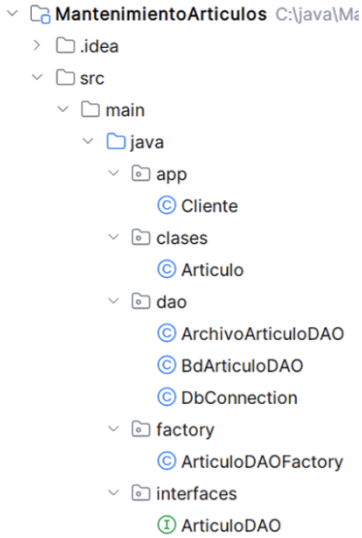


Con la implementación del patrón Factory, es posible desarrollar aplicaciones que puedan conectarse a múltiples *BD* y cambiar entre ellas con solo realizar una simple configuración, sin necesidad de modificar el código adicionalmente.



Ejemplo: uso del patrón Factory para la creación de DAOs.

- ➢ Crear la *BD* y la tabla correspondiente (ver código de ejemplo del punto 2.2).
- ➢ Agregar dependencias al proyecto (ver código de ejemplo del punto 2.2).
- ➢ `DbConnection.java` (ver código de ejemplo del punto 2.1) → Modificar el nombre de la *BD*, usuario y clave.
- ➢ `Articulo.java` (ver código de ejemplo del punto 2.2) .
- ➢ `ArticuloDAO.java` (ver código de ejemplo del punto 2.2).
- ➢ `BdArticuloDAO.java` (ver código de ejemplo del punto 2.2).
- ➢ `ArchivoArticuloDAO.java` (ver código de ejemplo del punto 2.2).
- ➢ `ArticuloDAOFactory.java`




```
package factory;

import dao.ArchivoArticuloDAO;
import dao.BdArticuloDAO;
import interfaces.ArticuloDAO;

public class ArticuloDAOFactory {
```

```
        public static ArticuloDAO getArticuloDAO(String tipoArticuloDAO) throws Exception {
            return switch (tipoArticuloDAO) {
                case "Archivo" -> new ArchivoArticuloDAO("tienda_pc.dat");
                case "MariaDB" -> new BdArticuloDAO();
                default -> null;
            };
        }
    }
```

- `Cliente.java` (ver código de ejemplo del punto 2.2) → Cambiar la siguiente línea:

```
        // ArticuloDAO dao = new ArchivoArticuloDAO("tienda_pc.dat");
        ArticuloDAO dao = ArticuloDAOFactory.getArticuloDAO("Archivo");
```

o

```
        // ArticuloDAO dao = new BdArticuloDAO();
        ArticuloDAO dao = ArticuloDAOFactory.getArticuloDAO("MariaDB");
```

Con estos ajustes, la aplicación será más flexible y mantenible, permitiendo la adición de nuevas implementaciones de `ArticuloDAO` con facilidad.