

Patrones de diseño y JDBC

Introducción

En este cuarto módulo, se va a completar y presentar el desarrollo del proyecto informático que da solución al problema identificado. Esto implica la elección y aplicación de un patrón de diseño, manteniendo la coherencia con las definiciones establecidas en la arquitectura previamente definida.

Para la creación del prototipo, será fundamental que este sea capaz de interactuar con una base de datos MySQL. Esta actividad incluye establecer conexiones, realizar consultas, actualizar registros y presentar resultados en la interfaz.

Como parte del proyecto integrador, se debe asegurar el uso de clases abstractas o interfaces para favorecer la reutilización de métodos y establecer una estructura robusta mediante la gestión de excepciones en Java. Además, se abordará la declaración, inicialización y manipulación de arreglos, así como la utilización de la clase *ArrayList*.

Al final de cada módulo, se aplican los conceptos de forma práctica, utilizando como referencia el proyecto informático planteado como base de trabajo para la materia. Te recomiendo apoyarte en las lecturas y bibliografía propuesta en las materias: INF382–Programación Orientada a Objetos e INF386 – Taller de Algoritmos y Estructuras de Datos I .

1. Patrones de diseño

El esquema de programación orientada a objetos, al tratarse de un problema de modelado de situación, cuenta con diferentes patrones o modelos predefinidos para resolver determinadas situaciones.

Aun cuando las situaciones sean particulares, se tiene la posibilidad de identificarlas con alguno de los patrones predeterminados con los que contamos.

Los patrones son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. Es decir, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares.

En un patrón, se identifican los siguientes elementos: su nombre, el problema que resuelve y que nos ayudará a comprender cuándo aplicar dicho patrón, la solución, los costos y beneficios que conlleva.

En la actualidad, existe un gran número de patrones documentados y, para facilitar la tarea de selección del más adecuado, podemos dividirlos en grupos de acuerdo con sus características principales.

Tipos de patrones

En diferentes fases del problema se utilizan diferentes tipos de patrones. En una etapa inicial, a la cual podemos identificar como fase de análisis/diseño, se aplican los que se conocen como patrones arquitectónicos, ya que en este punto se están planificando las bases para resolver un problema.

Una vez concluida dicha fase, y pasando ya a una etapa de desarrollo/construcción, se encuentran los patrones de diseño. Estos tienen un nivel de abstracción medio y se utilizan para resolver porciones acotadas del problema general. Son sencillos y flexibles, extensibles y reusables.

Los patrones de diseño permiten incrementar vocabulario de diseño, mejorar la documentación y mantenimiento de sistemas, acelerar el proceso de diseño —porque se utilizan soluciones testeadas— y facilitar el aprendizaje de buenas prácticas.

Patrón «modelo-vista-controlador» (MVC)

La realización de la interfaz de usuario de una aplicación resulta un problema complejo. “La principal característica del diseño de una interfaz de usuario es que debe ser lo suficientemente flexible para dar respuesta a las [...] exigencias propias de una interfaz moderna” (Debrauwer, 2013, p. 199).

Los usuarios de la aplicación pueden solicitar cambios a la interfaz para que sea más simple de usar, incluir nuevas funcionalidades, evolucionar la presentación de ventanas, etc. Estos requisitos hacen casi imposible diseñar una interfaz de usuario que pueda aplicarse en el seno del núcleo funcional de la aplicación, y resultará conveniente adoptar el patrón composite MVC.

- **Modelo.** Se trata del núcleo funcional que gestiona los datos manipulados en la aplicación. Es la representación de la información o esquemas de información con el cual va a trabajar nuestro sistema. Interactúa con el controlador y, usualmente, accede a la base de datos.
- **Vista.** Se trata de los componentes destinados a representar la información al usuario. Cada vista va a trabajar con los datos de un modelo, pero un modelo puede estar vinculado a varias vistas. Cada evento disparado por esta será interpretado por el controlador, quien tiene programado el comportamiento. Es el controlador el que regulará las acciones de la vista e interacciones con el modelo.
- **Controlador.** Es un componente que recibe los eventos que provienen del usuario y los traduce en consultas para el modelo o para la vista. Cada controlador responde a las acciones del usuario que vienen desde la vista. A su vez, envía peticiones al modelo cuando se hace alguna solicitud sobre la información, para que este acceda a la base de datos

MVC es una forma de organizar el código de un sitio o una aplicación para ayudar a mantener el orden y estructura de un proyecto. Algunas justificaciones prácticas de por qué utilizar MVC serían, por ejemplo, las siguientes:

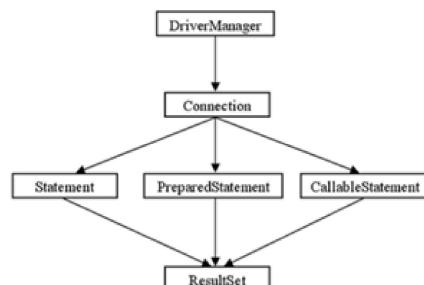
- Si el cliente quiere cambiar los colores de la presentación o el color de un botón determinado, con MVC se tendrá que cambiar únicamente los archivos o recursos que estén implicados en la vista.
- Si cambia algo en la base de datos o si agregamos cierta información en la estructura de una tabla, podremos realizar los cambios de sintaxis en los archivos correspondientes al modelo, sin temer que se hagan modificaciones involuntarias en el controlador.

En MVC aplicado correctamente, la estructura del código se encuentra bien segmentada y ordenada.

JDBC (Java Database Connectivity)

La biblioteca JDBC proporciona un conjunto de clases y, sobre todo, de interfaces que permiten la manipulación de una base de datos desde un programa Java. Estos elementos representan todo lo necesario para acceder a los datos desde una aplicación Java. (Grousard, 2010, p. 146)

Figura 1. Presentación de JDBC



Se cuenta con la clase «`DriverManager`», que es la encargada de asegurar el vínculo con el driver, para obtener la conexión con la base de datos. Esta clase implementa la interfaz «`Connection`»; esa conexión se usa para transmitir las instrucciones a la base de datos. Las peticiones simples se ejecutan mediante la interfaz «`Statement`», las peticiones con parámetros con la interfaz «`PreparedStatement`» y los procedimientos almacenados con la interfaz «`CallableStatement`».

Una vez consultada la base de datos, los registros seleccionados por la instrucción SQL se recogen y devuelven en forma de un elemento «`ResultSet`».

Utilización de la biblioteca JDBC

Antes que nada, se debe contar con el *driver JDBC* adaptado a la base de datos que vamos a utilizar. Este controlador suele estar disponible para su descarga en la página del diseñador de la base de datos.

Para poder utilizar este *driver* en la aplicación, se debe cargar mediante el método `forName` de la clase «`Class`». Este método recibe como parámetro una cadena de caracteres con el nombre del *driver* en cuestión. Si consultamos la documentación del *driver*, podremos obtener fácilmente el nombre de la clase.

El encargado de establecer la conexión es el método `getConnection` de la clase «`DriverManager`», y tiene 3 posibles formas de establecer la conexión. En caso de éxito, el método `getConnection` devuelve una instancia de una clase que implementa la interfaz «`Connection`».

Una vez establecida la conexión, es conveniente comprender cómo manipularla. Una conexión queda abierta desde el momento en que se crea, por lo que no tendremos que preocuparnos por abrirla, sino por cerrarla al terminar de utilizarla. Para cerrar la conexión, contamos con el método `close`, pero una vez cerrada, ya no hay posibilidad de comunicación sin una reconexión previa.

La función «`isClosed`» permite comprobar si la conexión está cerrada. En el caso de que se obtenga un «`false`» de esta función, no siempre quiere decir que la conexión está abierta y funcional, ya que hay un tercer estado de la conexión en el cual no se encuentra cerrada, pero tampoco permite transferir instrucciones hacia el servidor. Para comprobar la disponibilidad de la conexión, se cuenta con el método `isValid`, el cual comprueba la conexión al enviar una instrucción SQL y verificar la respuesta del servidor.

Si se desea efectuar operaciones de lectura únicamente, se puede optimizar la comunicación al precisar que la conexión será de solo lectura, mediante el método `setReadOnly` y se puede comprobar si una conexión está funcionando de este modo mediante el método `isReadOnly`.

Durante la ejecución de una instrucción SQL, el servidor puede detectar problemas y, por lo tanto, generar advertencias. Es posible recuperar estas advertencias mediante el método `getWarnings` de la conexión. Este método devuelve un objeto `SQLWarning` representativo del problema encontrado por el servidor. Si el servidor encuentra varios problemas, genera varios objetos `SQLWarning` encadenados entre sí. El método `getNextWarning` permite obtener el elemento siguiente o `NULL`, si se ha alcanzado el final de la lista. Se puede vaciar la lista con el método `clearWarnings`.

En general, en el momento de la creación de la URL de conexión, uno de sus parámetros determina el nombre de la base de datos con la que queremos establecer una conexión. La modificación del nombre de la base de datos a la que estamos conectados se lleva a cabo con el método `setCatalog`, al cual hay que proporcionarle el nombre de otra base de datos presente en el mismo servidor. Por supuesto, hace falta que la cuenta con la que se abrió la sesión disponga de permisos de acceso suficientes para esta base de datos. Cabe señalar que, con este método, cambiamos de base de datos, pero la conexión se sigue refiriendo al mismo servidor. No hay ningún medio para cambiar de servidor sin crear una nueva conexión.

Hay muchos métodos disponibles que pueden ser de utilidad. No obstante, la meta principal de una conexión es permitir la ejecución de instrucciones SQL. Por lo tanto, es ella la que facilitará los objetos necesarios para la ejecución de estas instrucciones.

2. Arreglos, arraylist, archivos y manejo de excepciones

Arreglos en Java

Un arreglo o *array* es una colección de variables del mismo tipo de datos, identificados por un nombre en común. Pueden contener tipos de datos primitivos o tipos de datos por referencia (objetos), aunque el arreglo propiamente dicho es siempre un objeto. Este objeto es especial, ya que no tiene métodos.

Cada una de las variables con datos que conforman el arreglo se denomina elemento. En Java, los arreglos pueden tener una o más dimensiones y se usan para una variedad de propósitos, porque ofrecen una forma conveniente de agrupar, ordenar y manipular variables relacionadas.

Aunque los arreglos en Java se pueden usar como las matrices en otros lenguajes de programación, tienen una característica especial: se implementan como objetos. Al hacerlo, se obtienen varias ventajas importantes, una de las cuales es que aquellos objetos que no son utilizados pueden ser recolectados.

Los elementos de un arreglo se enumeran consecutivamente: 0, 1, 2, 3, etc. Estos números se denominan valor índice y localizan la posición del elemento dentro del arreglo, proporcionando acceso directo a este.

Arreglo unidimensional: representa una lista de variables relacionadas, indicando un solo índice, al estar compuesto por una única dimensión.

En la siguiente tabla podemos observar un arreglo unidimensional con 5 elementos representados por valores numéricos enteros:

Tabla 1. Arreglo unidimensional con 5 elementos representados por valores numéricos enteros

Índice	0	1	2	3	4
Elemento	15	7	24	55	9

Fuente: elaboración propia

Arreglo bidimensional: representa una lista de variables relacionadas, indicando dos índices, al estar compuesto por dos dimensiones. Se los denomina también matriz o tabla.

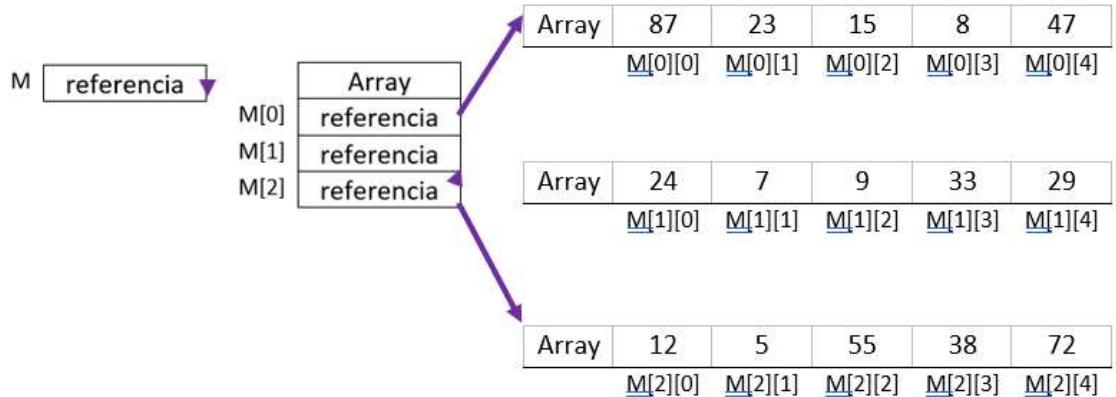
En la siguiente figura se puede observar un arreglo bidimensional con una matriz de 3 filas por 4 columnas

Figura 2. Arreglo bidimensional con una matriz de 3 filas por 4 columnas

Filas	0	0	1	2	3	4
	0	87	23	15	8	47
	1	24	7	9	33	29
	2	12	5	55	38	72

A efectos prácticos, cuando trabajamos con arreglos bidimensionales podemos pensar en una tabla como la presentada anteriormente, donde los elementos están distribuidos en filas y columnas. Pero, en realidad, una matriz en Java es un arreglo de arreglo, cuya disposición real en memoria es la siguiente:

Figura 3. Matriz en Java



`M` es el nombre del arreglo. Contiene la dirección de memoria (referencia) de un arreglo unidimensional de 3 elementos. Cada elemento de este arreglo unidimensional contiene la dirección de memoria de otro arreglo unidimensional. Cada uno de estos últimos arreglos unidimensionales contiene los valores de cada fila de la matriz.

Se puede observar que la primera fila y columna de la matriz se identifica como 0. Así, para acceder a cada elemento de la matriz se utilizan dos índices: el primero indica la fila y el segundo, la columna.

«`M[0][2] = 15;` //» asigna el valor 15 al elemento situado en la primera fila (fila 0) y tercera columna (fila 2).

Declaración de un arreglo

Se declara de forma similar a otros tipos de datos, primitivos o por referencia (objetos), pero indicando al compilador que se trata de un arreglo, mediante el uso de corchetes que pueden ubicarse a la derecha o izquierda del identificador.

La sintaxis de declaración de variables arreglo en Java es la siguiente:

```
tipo [] identificador;  
  
tipo identificador[];
```

Java no permite indicar el número de elementos en la declaración de un arreglo; por ejemplo: en la declaración «`int numero[12]`», el compilador producirá un error.

Si se desea declarar un arreglo de más de una dimensión, se deben agregar tantos pares de corchetes como dimensiones tenga el arreglo. Por ejemplo, si declaramos un arreglo de dos dimensiones con objetos del tipo «Estudiante», sería así:

```
Estudiante [][] listado;
```

También, se lo podría haber declarado de la siguiente forma:

```
Estudiante listado [][];
```

Creación de un arreglo

Java considera que un arreglo es una referencia a un objeto, por lo que se debe invocar su constructor para asignar la memoria y crear el objeto.

Para la creación de un arreglo, Java debe conocer cuánto espacio de memoria debe reservar, por lo que se debe especificar el tamaño, haciendo referencia a la cantidad de elementos que este puede almacenar.

Para crear realmente el arreglo, utilizamos la palabra reservada «new», seguida por el tipo de dato del arreglo y los corchetes. La sintaxis para declarar y definir un arreglo de un número de elementos determinado es la siguiente:

```
tipo nombreArreglo [] = new tipo[numeroDeElementos];
```

O bien:

```
tipo nombreArreglo [];
```

```
nombreArreglo = new tipo[numeroDeElementos];
```

También, se puede declarar y crear un arreglo en una sola sentencia:

```
int[] nota = new int[3];
```

```
Estudiante[] listado = new Estudiante[3];
```

Al utilizar el operador «new», como en el ejemplo anterior, estamos construyendo o instanciando un objeto arreglo. Cuando construimos el arreglo «listado», utilizando el operador «new», no estamos invocando al constructor de la clase «Estudiante», ni creando instancias de tipo «Estudiante».

En la creación de un arreglo, siempre se debe indicar el tamaño, ya que de lo contrario aparecerá un error en la etapa de compilación. Como dijimos, la JVM necesita conocer cuánto espacio de memoria debe reservar para almacenar el arreglo.

Inicialización de un arreglo

Un arreglo puede almacenar valores primitivos o bien objetos, mediante variables de referencia. La inicialización consiste en asignarle valores del tipo definido a los elementos.

Inicialización de forma explícita

Esta metodología se puede aplicar cuando el número de elementos del arreglo es pequeño y se utiliza una serie de valores entre llaves. Esto solo puede realizarse para inicializar el arreglo y no en sentencias de asignación posteriores.

```
int cuenta[] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos enteros a cuenta y, además, es capaz de determinar su tamaño.

Figura 4. Compilador

Índice	0	1	2	3	4
Elemento	15	25	-45	0	50

Fuente: elaboración propia

Inicialización mediante un índice

El acceso a los elementos del arreglo se puede realizar mediante un índice que indica la posición exacta dentro de este. El índice del primer elemento es cero.

Entonces, supongamos que tenemos definida la clase «Estudiante» y declaramos un arreglo que contiene elementos de tipo «Estudiante»:

```
Estudiante alumno[] = new Estudiante[3];
```

El objeto «alumno» puede almacenar elementos que representan referencias a objetos de tipo «Estudiante»; pero, como vimos anteriormente, hasta el momento solo hemos realizado la creación del arreglo, por lo que las tres referencias son nulas.

Para inicializar el arreglo, podemos crear objetos del tipo «Estudiante» y asignarlos a las posiciones del arreglo «alumno».

```
alumno[0] = new Estudiante("Monica");  
  
alumno[1] = new Estudiante("Eduardo");  
  
alumno[2] = new Estudiante("Laura");
```

Si declaramos y creamos un arreglo, sus elementos quedan inicializados con los valores por defecto, correspondientes al tipo de datos del arreglo.

Tabla 2. Declaración de un arreglo

Tipo de dato	Valor por defecto
objeto	null
byte, short, int, long	0
float, double	0.0
boolean	false

char	'\u0000'
------	----------

Fuente: elaboración propia

Inicialización con sentencias iterativas

Esta alternativa resulta de mucha utilidad, principalmente para inicializar arreglos de gran tamaño. Para identificar la ventaja, supongamos que tenemos un arreglo «String» de 50 elementos y deseamos inicializarlo con la cadena «Test». Si usamos la inicialización clásica por índice, deberíamos hacerlo de la siguiente forma:

```
String[] st = new String[50];

st[0] = new String("Test");

st[1] = new String("Test");

st[2] = new String("Test");

...

st[49] = new String("Test");
```

El uso de las sentencias iterativas nos permite repetir ciertas sentencias de código un número determinado de veces o mientras se cumpla una condición. Adicionalmente, los arreglos tienen un atributo denominado «length», que identifica el número de elementos que puede almacenar. En nuestro ejemplo, el valor de «st.length» sería igual a 50.

Colecciones

Las clases «Vector» y «ArrayList» de Java proporcionan colecciones, que son un grupo de clases que almacenan secuencias de objetos. Una de las diferencias principales entre ambos es que la clase «Vector» y sus métodos están sincronizados y, una vez que se invoca un método, no se puede invocar el mismo método, a menos que la llamada anterior haya finalizado. En el caso de la clase «ArrayList» es no sincronizada.

La seguridad de subprocessos está relacionada con el uso de un mecanismo de bloqueo. El subprocesso seguro proporciona protección al acceso de datos, ya que no permite que otros subprocessos puedan realizar cambio y, por lo tanto, no habrá inconsistencia de datos o contaminación de datos.

La clase «Vector» se puede utilizar, incluso, es probable que nos encontremos con código que la tenga disponible y, por ello, la vamos a analizar. Sin embargo, excepto casos especiales, actualmente se prefiere el empleo de «ArrayList».

«Vector» y «ArrayList» pueden aumentar automáticamente la capacidad, según sea necesario. «Vector» se duplica durante la expansión de la capacidad, y «ArrayList» se incrementa en un 50 % durante la expansión de la capacidad. «Vector» usa la enumeración y el iterador para atravesar una matriz; mientras que una «ArrayList» solo usa iterador para atravesar una matriz.

Clase «ArrayList»

La clase «ArrayList» permite almacenar datos en memoria de forma similar a los arreglos, pero su tamaño puede variar dinámicamente. Además, dispone de muchos métodos que nos permiten añadir, eliminar y modificar elementos de forma simple.

Pertenece a la lista de clases de colección estándar. Se define dentro del paquete java.util, extiende la clase «AbstractList», que también es una

clase de colección estándar, y también implementa «List», una interfaz definida en las interfaces de colección.

«ArrayList» es una clase contenedora genérica que puede implementar arreglos dinámicos de objetos que pueden ser de tipos distintos. Como esto puede traer complicaciones en el momento de trabajar con sus datos, existe la posibilidad de indicar en la declaración el tipo de objetos que contiene.

Diferencias entre un arreglo y «ArrayList»

En Java, los arreglos estándar son siempre de una longitud fija o estática y, una vez creados, no es posible hacer crecer o disminuir su tamaño. Por lo tanto, debemos tener el conocimiento previo de la longitud del arreglo, y esto a veces recién lo conocemos en tiempo de ejecución. Esto no ocurre en «ArrayList», que puede variar su tamaño de forma dinámica y dispone de muchos métodos que nos permiten añadir, eliminar y modificar elementos de forma simple.

El arreglo puede ser de varias dimensiones, mientras que el «ArrayList» es unidimensional. La variable de longitud se utiliza para determinar el tamaño de la matriz. El método size() se utiliza para determinar el tamaño del «ArrayList».

Un arreglo es una estructura que admite objetos de una clase o datos primitivos, pero siempre del mismo tipo, especificado en la declaración. Si se intenta almacenar un tipo de datos diferente, al que se especificó al crear el objeto «Array», se lanza la excepción 'ArrayStoreException'. «ArrayList», en cambio, es compatible con genéricos, pero solo puede almacenar objetos y usa la clase envolvente para trabajar con datos primitivos.

Un arreglo es un componente de programación nativo en Java y utiliza el operador de asignación para almacenar elementos que se almacenan en ubicaciones de memoria contiguas. «ArrayList» usa el método add() para insertar elementos y, al ser una clase del marco de colecciones de Java, los objetos no se almacenan en ubicaciones contiguas.

El rendimiento puede variar, dependiendo de la operación que se esté realizando. Sin embargo, iterar sobre un arreglo es algo más rápido que iterar sobre un arraylist. Si el programa involucra un gran número datos primitivos, la matriz tendrá un rendimiento mejor que el «ArrayList», tanto en términos de tiempo como de memoria. Las matrices son un lenguaje de programación de bajo nivel que se puede usar en implementaciones de colecciones.

«ArrayList» requiere más memoria para almacenar igual cantidad de datos que un arreglo, principalmente cuando se trabaja con datos primitivos, donde usa su envolvente.

Creación de un arraylist

Se utiliza el operador «new» de igual forma que se crea cualquier objeto; la clase «ArrayList» dispone de diversos constructores:

Tabla 3. Creación de un arraylist

Prototipo de constructor	Descripción
ArrayList()	Este es el constructor predeterminado de la clase ArrayList. Crea una clase ArrayList vacía con tamaño 10.
ArrayList (int capacidad)	Este constructor sobrecargado construye un objeto ArrayList vacío con una capacidad inicial definida. Cuando se supera la capacidad, esta se incrementa un 50 %.
	Se crea un objeto ArrayList con los elementos iniciales de la colección especificada c. La expresión «Collection<?>

ArrayList (Collection<? extends E>c)

extends E>c» significa que puede ser cualquier estructura de datos que extienda de la clase «Collection». El «?» se utiliza como un comodín.

Fuente: elaboración propia

Tomando el constructor predeterminado, se podría declarar y crear un *arraylist* con la siguiente sintaxis. También, siempre, podríamos separar la declaración y creación.

```
ArrayList nombreArray = new ArrayList();
```

Esta instrucción crea el *arraylist* «nombreArray» vacío y, así declarado, puede contener objetos de cualquier tipo. Es decir, un Ar *arraylist* puede contener objetos de **tipos distintos**.

```
ArrayList a = new ArrayList();

a.add("Palabra");

a.add(5);

a.add('c');

a.add(21.5);
```

El primer objeto que se añade es el *string* «Palabra». El resto no son objetos, son datos de tipos primitivo que el compilador convierte automáticamente en objetos de su clase envolvente (clase contenedora o *wrapper*) antes de añadirlos al *array*.

Un arreglo al que se le pueden asignar objetos de distinto tipo podría traer alguna complicación a la hora de trabajar sus datos. Una alternativa a esta declaración es indicar el tipo de objetos que contiene.

La siguiente declaración y creación de un *arraylist* solo soportará objetos de la clase <tipo>:

```
ArrayList<tipo> nombreArray = new ArrayList<>();
```

<tipo> debe ser una clase e indica el tipo de objetos que contendrá el array.

La sentencia utiliza el operador diamante <> en reemplazo de <tipo> y limita el objeto que se debe utilizar. Igualmente, funcionaría si se coloca <tipo>, pero es redundante. En un *arraylist* no se pueden usar tipos primitivos, pero sí su clase envolvente, por ejemplo:

```
ArrayList<Integer> numero = new ArrayList<>();
```

Iteradores

La forma más sencilla de recorrer un *arraylist* es a través de un bucle «for» y accediendo a la propiedad «size». Esto es útil, pero, a medida que el programa gana en complejidad, es necesario evitar errores.

Una alternativa a esta situación es usar un iterador. En Java, un iterador es una construcción que se utiliza para recorrer la colección. Para utilizar un iterador, debemos obtener el objeto del iterador, utilizando la interfaz que funciona como un cursor, para recorrer la colección de objetos mediante sus métodos *hasNext()* y *next()*. (Álvarez Caules, 2020, <https://lc.cx/sDm8K5>)

El uso de operaciones sobre colecciones usando un «for» tradicional también puede dar lugar a resultados no deseados. En general, operar sobre una colección al mismo tiempo que la recorremos puede ser problemático. Este problema queda salvado mediante el uso de los objetos tipo «Iterator». Un objeto de tipo «Iterator» funciona como una copia para recorrer una colección, es decir, el recorrido no se basa en la colección real, sino en una copia. De este modo, al mismo tiempo que hacemos el recorrido, podemos manipular la colección real, añadiendo, eliminando o modificando elementos de esta. Para poder usar objetos de tipo «Iterator», se debe importar el paquete java.util.Iterator. La sintaxis para poder usar los objetos del tipo «Iterator» es:

```
Iterator <TipoARecorrer> it = nombreDeLaColección.iterator();
```

Dado que no disponemos de constructor, usamos un método del que disponen todas las colecciones, denominado iterator() y que, cuando es invocado, devuelve un objeto de tipo «Iterator» con una copia de la colección. Por tanto, hemos de distinguir:

1. «Iterator» con mayúsculas, que define un tipo.
2. El método «iterator» (con minúscula), que está disponible para todas las colecciones y que, cuando es invocado, devuelve un objeto de tipo «Iterator» con una copia de la colección.

Es decir, «Iterator» es un tipo genérico, porque requiere que definamos un tipo complementario, cuando declaramos objetos. Los objetos de tipo «Iterator» tienen como métodos los siguientes:

Tabla 4. Métodos de los objetos de tipo «iterator»

Método	Descripción
E next()	Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción
boolean hasNext()	Indica si hay un elemento siguiente (y así evita la excepción).
void remove()	Elimina el último elemento devuelto por next

Fuente: elaboración propia

Excepciones en Java

Una excepción es una situación anormal ocurrida durante la ejecución de un programa que altera su flujo normal. En un programa Java, pueden producirse excepciones por distintos motivos. Por ejemplo, en el caso de que un programa ejecute una división entre cero, si intenta escribir ciertos datos en un archivo y ocurre un error, si intenta acceder a un elemento de un arreglo utilizando un índice incorrecto, entre otros.

Existen muchas otras situaciones en las que se puede producir una excepción. En algunos casos, las excepciones indican errores en el programa y en otros indican algún problema grave ocurrido en la JVM, el sistema operativo o en el hardware utilizado.

Escribir un código que verifique y gestione las posibles situaciones anómalas que podría tener un programa es muy importante. La detección de posibles errores y la determinación de cómo debería reaccionar la aplicación ante esos errores, es un aspecto fundamental que definirá qué tan robusta y confiable es la aplicación. La plataforma de desarrollo Java proporciona algunos mecanismos que permiten a un programa Java detectar las excepciones y recuperarse de los posibles problemas ocurridos. Los programas Java también pueden generar excepciones, por

ejemplo, para señalar alguna situación anómala detectada por el propio programa.

Mediante el manejo de excepciones, los desarrolladores pueden detectar posibles eventos anormales e indicarle a la aplicación cómo se debe comportarse ante su aparición. Una excepción es representada por un objeto que la máquina virtual de Java instancia en tiempo de ejecución, si sucede algún evento anormal. Permite al programador intervenir en esta situación, si lo considera necesario, para recuperarse del estado anormal y evitar que la aplicación finalice de forma abrupta.

Manejo de excepciones en Java

De acuerdo con la obligación o no de definir el tratamiento de excepciones, estas se pueden clasificar en verificadas en compilación y no verificadas en compilación. Para las excepciones que son verificadas, es obligatorio definir un código que haga un tratamiento. Las excepciones que derivan de «Exception» son verificadas, excepto las «RuntimeException».

Una ventaja principal del manejo de excepciones es que automatiza gran parte del código de manejo de errores que previamente debía realizar el programador en cualquier programa grande. El manejo de excepciones agiliza la gestión de errores, ya que, si se produce un error, será detectado, se lanza la excepción y es procesada por el manejador de excepciones. Este bloque de código, llamado «manejador de excepciones», se ejecuta automáticamente cuando se detecta que ocurre un error y no es necesario verificar manualmente el éxito o el fracaso de cada operación específica. El manejo de errores usando excepciones no evita errores, pero permite la detección y eventual corrección de estos en tiempo de ejecución.

El manejo de excepciones es importante y se lo considera una buena práctica, ya que permite gestionar errores comunes del programa. Para responder a estos errores, el programa debe vigilar y manejar estas excepciones. Además, la biblioteca API de Java hace un uso extensivo de excepciones.

El modelo para el manejo de excepciones en Java consta fundamentalmente de cinco nuevas palabras reservadas: *try*, *throw*, *throws*, *catch* y *finally*.

- **try:** es un bloque para detectar un error y lanzar excepciones.
- **catch:** es un manejador para capturar y realizar el tratamiento de los errores que provocaron la excepción.
- **throw:** es una expresión para lanzar excepciones.
- **throws:** permite declarar la lista de excepciones que un método puede llegar a lanzar.
- **finally:** es un bloque opcional situado después de los catch de un try, que se ejecuta siempre.

Captura de excepciones

Cuando se presenta una excepción, el código responsable de hacer algo con dicha excepción se denomina gestor de la excepción. Se dice que este código captura la excepción y la ejecución del programa se transfiere al manejador de la excepción apropiado.

La palabra reservada **try** designa un bloque de código a evaluar para intentar capturar un error y lanzar la excepción para que pueda ser gestionada por el manejador **catch**.

La palabra reservada **catch** designa un bloque de código con el conjunto de instrucciones necesarias para el tratamiento del error capturado por **try**. En la sintaxis, el argumento de **catch** podría ser «*Exception ex*»; lo que significa que, cuando se produce un error, Java genera un objeto «*ex*» de tipo «*Exception*», con la información sobre el error, y este objeto se envía al bloque **catch**. Estos errores solo pueden ser detectados en tiempo

de ejecución del programa. Los bloques *catch* siguen inmediatamente a bloques *try* o a otro manejador *catch* con diferente argumento.

Al contrario que el bloque *try*, *catch* solo se ejecuta en circunstancias especiales, cuando el argumento es la excepción que puede ser capturada, una referencia a un objeto de una clase ya definida o bien derivada de la clase base «Exception». Cuando ocurre una excepción en una sentencia durante la ejecución del bloque *try*, el programa comprueba cada bloque *catch*, comenzando por el primero, hasta que encuentra un manejador cuyo argumento coincide con el tipo de excepción. Puede haber n bloques *catch* a continuación de un *try*.

Java proporciona un manejador especial denominado *finally*, que es opcional y se escribe después del último *catch*. En un bloque *try*, se ejecutan sentencias de todo tipo, llamadas a métodos, creación de objetos, pedido de recursos, procesamiento de recursos, apertura de archivos, etc. Todos estos recursos tienen que ser liberados cuando termine el bloque *try*, o también se puede requerir ejecutar un bloque de sentencias siempre. Para ello, existe la cláusula *finally*, que se ejecuta siempre, se haya presentado o no una excepción.

Captura de excepciones de subclase

La cláusula *catch* para una superclase también coincidirá con cualquiera de sus subclases. La superclase de todas las excepciones es *Throwable*. Para atrapar todas las excepciones posibles, se debe capturar *Throwable*.

Si deseamos capturar excepciones de un tipo de superclase y un tipo de subclase, debemos colocar primero la subclase en la secuencia de *catch*. Si no lo hacemos, la captura de la superclase también atrapará todas las clases derivadas. Esta regla se autoejecuta, porque poner primero la superclase hace que se cree un código inalcanzable, ya que la cláusula *catch* de la subclase nunca se puede ejecutar, y en Java el código inalcanzable es un error.

Lanzamiento de excepciones

Cuando en un método necesitamos lanzar una excepción en tiempo de ejecución, utilizaremos la palabra clave *throw*, seguida de una instancia de la excepción a lanzar. Cuando el programa la encuentra, puede comunicar que la excepción ocurrió por un lanzamiento. La sintaxis a utilizar es la siguiente:

```
throw objeto;
```

El objeto debe ser de la clase «Exception» o una clase derivada, porque lanzar una excepción hace que termine el bloque *try* y que las sentencias que siguen no se ejecuten.

El objeto puede instanciarse y luego lanzarse la excepción:

```
Exception e = new Exception("Dividendo ingresado negativo!");
throw e;
```

O hacer ambos pasos en la misma línea:

```
throw new Exception("Dividendo ingresado negativo!");
```

El manejador *catch* que captura una excepción realiza un proceso con ella y puede decidir devolver control, *return*, o continuar la ejecución en el mismo método, a continuación del último *catch*.

Especificación de excepciones

Cuando se produce una excepción, la máquina virtual interrumpe la ejecución normal del programa y busca un bloque de código adecuado para tratar la situación. Si no encuentra este código en el método actual, la excepción se propaga hacia el método que lo haya invocado y se busca allí el código que la trate. Si tampoco ese método dispone del código adecuado, se propagará, a su vez, al que lo haya invocado, y así sucesivamente.

Java ofrece la posibilidad de declarar en la cabecera de los métodos las excepciones que esta puede llegar generar, lo que se conoce como especificación de excepciones. Consiste en una lista de las excepciones que el método puede lanzar, lo que garantiza que el método no lance ninguna otra excepción, salvo las del tipo `RuntimeException` o `Error`, que son no verificadas y, por lo tanto, no es necesario especificarlas.

La cláusula `throws`, a continuación del método, permite declarar la lista de excepciones que el método puede llegar a generar. La sintaxis es la siguiente:

```
tipo nombreMetodo(listaParametro) throws listaExcepciones {  
    // Cuerpo  
}
```

Aquí, «`listaExcepciones`» es una lista de excepciones separadas por comas, que el método podría lanzar fuera de sí mismo.

Manipulación de archivos en Java

Entrada y salida de datos

El paquete `java.io` contiene todas las clases relacionadas con las funciones de entrada (*input*) y salida (*output*). En términos de programación, se denomina entrada a la posibilidad de introducir datos hacia un programa; salida sería la capacidad de un programa de mostrar información al usuario. (Sánchez Asenjo, 2010, p. 6)

Archivos y flujos

Un **archivo** es una colección de registros relacionados entre sí con aspectos comunes y organizados para contener datos, de tal modo que puedan recuperarse fácilmente, borrarse, actualizarse o almacenarse de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se organizan los registros, el acceso a los registros del archivo se puede realizar de forma secuencial y directa. El primero implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro. El segundo supone el acceso a un registro determinado, sin que ello involucre la consulta de los registros previos.

En Java, un archivo es una secuencia de *bytes*, que son la representación de los datos almacenados. Dispone de clases para trabajar las secuencias de *bytes*, como datos de tipos primitivos o de referencia. Un **flujo** es una abstracción que se refiere a una corriente de datos que van desde un origen hasta un destino. Entre uno y otro debe existir una conexión o canal (*pipe*) por donde circulan los datos.

Al comenzar la ejecución de un programa en Java, se crean automáticamente tres objetos de flujo, canales por los que pueden fluir datos de entrada o salida. Estos son objetos definidos en la clase `«System»`:

- **System.in**: de entrada estándar; permite el ingreso al programa de flujos de *bytes* desde el teclado.
- **System.out**: de salida estándar; permite al programa imprimir datos por pantalla.
- **System.err**: de salida estándar de errores; permite al programa imprimir errores por pantalla.

Clase «File»

En el paquete `java.io` se encuentra la clase `«File»`, pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos: listado de archivos, crear carpetas, borrar archivos, cambiar nombre, etc. Un objeto `«File»` representa un archivo o un directorio y sirve para obtener información y para navegar por la estructura de archivos. (Sánchez Asenjo, 2010, p. 6)

Flujos de datos en Java

Todo el proceso de entrada y salida en Java se realiza a través de *streams*, que son flujos de datos binarios accesibles mediante una secuencia de *bytes*. Estos flujos no representan ni textos, ni objetos, sino datos binarios puros.

En los programas, entonces, hay que crear objetos *stream* y, para ello, Java utiliza dos clases que derivan directamente de Object: «*InputStream*» y «*OutputStream*». Ambas son abstractas y declaran métodos que deben redefinirse en sus clases derivadas. La primera es la base de todas las clases definidas para flujos de entrada; la segunda es la base de todas las clases definidas para flujos de salida.

Estas dos clases abstractas definen las funciones básicas de lectura y escritura de un *stream* (sin estructurar). Poseen numerosas subclases; de hecho, casi todas las clases preparadas para la lectura y la escritura derivan de estas.

En el caso de las excepciones, todas las que provocan las excepciones de E/S son derivadas de *IOException* o de sus derivadas. Además, son habituales, ya que la entrada y salida de datos es una operación crítica y, por lo tanto, la mayoría de las operaciones deben acompañarse por un manejador de excepciones *try-catch*.

Archivos de bajo nivel: *FileInputStream* y *FileOutputStream*

Todo archivo de entrada y de salida se puede considerar como una secuencia de *bytes* de bajo nivel, a partir de la cual se construyen flujos de más alto nivel para proceso de datos complejos, desde tipos básicos hasta objetos. Las clases «*FileInputStream*» y « *FileOutputStream*» se utilizan para leer o escribir *bytes* en un archivo. Los objetos de estas dos clases son los flujos de entrada y salida a nivel de *bytes*. Los constructores de ambas clases permiten crear flujos u objetos asociados a un archivo que se encuentra en cualquier dispositivo.

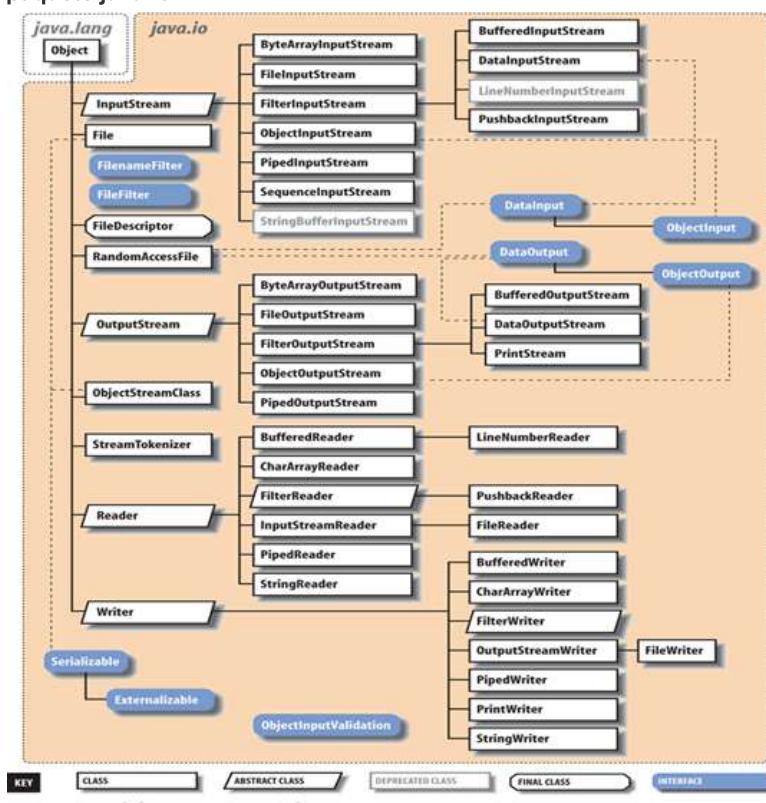
Archivos de datos: *DataInputStream* y *DataOutputStream*

A veces, resulta complejo trabajar de manera directa con flujos de *bytes*; ya que los datos que se escriben o se leen en los archivos son enteros, cadenas, etc. Las clases «*DataInputStream*» y «*DataOutputStream*» derivan de *FileInputStream* y « *FileOutputStream*». Estas organizan las secuencias de *bytes* para formar datos primitivos o *string*. Así, se pueden escribir o leer directamente datos de tipo *char*, *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* y *string*.

Flujos *PrintStream*

La clase «*PrintStream*» deriva directamente de «*FilterOutputStream*». Su característica más importante es disponer de métodos que agregan la marca de fin de línea. Los flujos de tipo «*PrintStream*» son de salida y se asocian con otro de bajo nivel de bytes, que, a su vez, se crea asociado a un archivo externo. Los métodos de esta clase *print()* y *println()* se sobrecargan para escribir desde cadenas hasta cualquier dato primitivo; *println()* escribe un dato y, a continuación, agrega la marca de fin de línea.

Figura 5. Clases e interfaces del paquete java.io



3. Desarrollo del sistema utilizando Java

Recuerda que te encuentras trabajando en el equipo de desarrollo de ECOViento y te solicitan liderar un proyecto informático para diseñar y desarrollar un sistema de monitoreo y control de centrales eólicas.

En el tercer módulo, se aplicarán los conceptos de patrones de diseño, conexión de Java con una base de datos MySQL, manejo de estructura de datos, excepciones y archivos. Los entregables son los siguientes:

- Selección del patrón de diseño y justificación.
- Explicación del desarrollo, usando un patrón MVC.
- Presentación del desarrollo en Java.
- Entrega del proyecto integrador, completo.
- Presentación del proyecto en un video con una duración, aproximadamente, de 3 minutos.

Selección del patrón de diseño y justificación

Un patrón de diseño de software proporciona una estructura predefinida que ayuda a resolver de manera eficiente los problemas que se presentan en el desarrollo de una aplicación.

Cada patrón de diseño tiene un propósito específico y se define al momento de crear la arquitectura. Para el desarrollo del sistema de gestión y monitoreo de centrales eólicas, se definió utilizar el patrón MVC (modelo-vista-controlador).

Sin embargo, para el prototipo se adoptaron algunas simplificaciones. Solo se va a aplicar el concepto, omitiendo la utilización de *frameworks*, *servlets*, JSP, HTML, entre otros. Adicionalmente, se va a crear un paquete independiente que utiliza este patrón solo para el registro de las condiciones del viento, sobre una lista de turbinas ya almacenadas en la base de datos MySQL. Esto permite cumplir con el requerimiento RNF06.

Tabla 5. Requerimiento RNF06

Requerimiento Sistema	Descripción
RNF06	Para la persistencia y consulta de datos en la BD, se debe utilizar un patrón MVC (modelo-vista-controlador).

Fuente: elaboración propia

La selección del patrón MVC permite trabajar con aplicaciones que disponen de una estructura clara y organizada. Pensando en el sistema final, es fundamental contar con una metodología que facilite la gestión, el mantenimiento y la escalabilidad.

El patrón MVC se basa en tres componentes esenciales que trabajan de manera conjunta, pero independiente, para lograr un diseño de software eficiente y modular. Cada uno de estos componentes desempeña un papel específico en el ciclo de vida de una aplicación, lo que simplifica la gestión de la lógica de negocio y la presentación de datos. El modelo se encarga de la lógica y el almacenamiento de datos, la vista se ocupa de la presentación visual y el controlador gestiona la interacción del usuario. Esta división facilita el desarrollo y mantenimiento del proyecto, al evitar la mezcla de estas áreas críticas.

Explicación del desarrollo usando un patrón MVC

Inicialmente, es importante recordar que, antes de comenzar a escribir el código, se requiere realizar un análisis y diseño detallado. En este proyecto se está utilizando el proceso unificado de desarrollo (PUD) y, a los fines de simplificación, se omite la presentación de los artefactos generados.

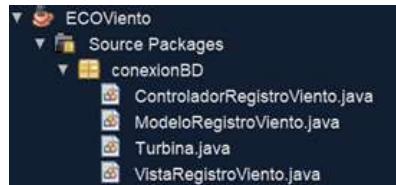
Para el prototipo, se va a aplicar el patrón MVC solo para un paquete conexiónBD independiente que contiene las clases que permiten registrar las condiciones del viento, sobre una lista de turbinas ya almacenadas en la base de datos MySQL.

Adicionalmente, se va a crear una clase turbina en este paquete. En el sistema final, esto puede no ser necesario, si se implementan las interfaces correspondientes. Se realiza también a los fines de simplificar la presentación

Paquete conexiónBD

Contiene las clases que permiten registrar las condiciones del viento, sobre una lista de turbinas ya almacenadas en la base de datos MySQL.

Figura 6. Paquete conexiónBD



Fuente: captura de pantalla de MySQL (Oracle Corporation, 2000)

Clase «Turbina»

Genera objetos de turbina eólica y tiene cuatro atributos privados, un constructor y los métodos *getter/setter* (se omite la presentación).

- **id**: identifica la turbina eólica de manera única.
- **modelo**: identifica el modelo de la turbina eólica.
- **potenciaMax**: representa la potencia máxima de la turbina.
- **estado**: indica si la turbina está funcionando o no.

Figura 7. Clase «Turbina»

```
package conexiónBD;

public class Turbina {
    private int id;
    private String modelo;
    private double potenciaMax;
    private String estado;

    public Turbina(int id, String modelo, double potenciaMax, String estado) {
        this.id = id;
        this.modelo = modelo;
        this.potenciaMax = potenciaMax;
        this.estado = estado;
    }
}
```

Fuente: elaboración propia

Clase «ControladorRegistroViento»

Es parte del patrón MVC y se encarga de controlar las interacciones relacionadas con las mediciones de viento. El controlador se sitúa entre el modelo, que gestiona la base de datos, y la vista, que gestiona la interfaz de usuario.

En esta clase, se realiza la importación de paquetes para gestionar conexiones y dar tratamiento a las excepciones. Además, importa la interfaz

«List» del paquete java.util, que permite trabajar con colecciones en forma de lista.

Contiene dos atributos privados y el constructor.

- **modelo**: es una instancia de la clase «ModeloRegistroViento» y se utiliza para gestionar los datos relacionados con los registros de viento.
- **vista**: es una instancia de la clase «ModeloRegistroViento» y se utiliza para manejar la presentación de información y la interacción con el usuario.

Figura 8. Clase «ControladorRegistroViento»

```
package conexionBD;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

public class ControladorRegistroViento{
    private ModeloRegistroViento modelo;
    private VistaRegistroViento vista;

    public ControladorRegistroViento(Connection conexion) {
        this.modelo = new ModeloRegistroViento(conexion);
        this.vista = new VistaRegistroViento();
    }
}
```

Fuente: elaboración propia

Adicionalmente, cuenta con los siguientes dos métodos:

- Método ejecutar()

Figura 9. Método ejecutar()

```
public void ejecutar() throws SQLException {
    int idTurbina = vista.solicitarIdTurbina();
    double velocidadViento = vista.solicitarVelocidadViento();
    String direccionViento = vista.solicitarDireccionViento();

    modelo.insertarRegistroViento(idTurbina, velocidadViento, direccionViento);
    System.out.println("Medición insertada en base de datos");
}
```

Fuente: elaboración propia

- Método mostarTurbinasDisponibles()

Figura 10. Método mostarTurbinasDisponibles()

```
public void mostarTurbinasDisponibles() {
    List<Turbina> turbinasDisponibles = modelo.consultarTurbinasDisponibles();
    vista.mostrarTurbinas(turbinasDisponibles);
}
```

Fuente: elaboración propia

Clase «ModeloRegistroViento»

Es parte del patrón MVC y se utiliza para manejar la lógica que permite la interacción con la base de datos. El modelo se encarga de representar y gestionar los datos relacionados con las mediciones de viento.

En esta clase se realiza la importación de paquetes que contienen las clases necesarias para establecer conexiones, ejecutar consultas SQL, manejar errores y gestionar conjuntos de resultados.

Además, se trabaja con un *arraylist*, que es una implementación de la interfaz «List» en Java y se utiliza para crear colecciones dinámicas, que pueden crecer o disminuir, según sea necesario. Se utiliza para gestionar los datos que se utilizan en las consultas SQL.

Hay que recordar que *list* es una abstracción que define operaciones comunes para trabajar con listas, independientemente de la implementación. *arraylist* es una implementación específica, que utiliza arreglos dinámicos para almacenar elementos.

Figura 11. Clase «ModeloRegistroViento»

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

Fuente: elaboración propia

Por otro lado, esta clase contiene atributos privados y el constructor.

- **conexión**: permite mantener una única instancia de conexión a la base de datos, que se utilizará en los métodos para interactuar con la base de datos.

Figura 12. Atributo «conexión»

```
public class ModeloRegistroViento {
    private Connection conexion;

    public ModeloRegistroViento(Connection conexion) {
        this.conexion = conexion;
    }
}
```

Fuente: elaboración propia

Adicionalmente, cuenta con los siguientes dos métodos:

- Método *insertarRegistroViento()*

Figura 14. Método consultarTurbinasDisponibles()

```
public List<Turbina> consultarTurbinasDisponibles() {
    List<Turbina> turbinas = new ArrayList();
    PreparedStatement statement = null;
    ResultSet resultSet = null;

    try {
        String consultaSQL = "SELECT * FROM turbinas_eolicas WHERE estado = 'Activa'";
        statement = conexion.prepareStatement(consultaSQL);
        resultSet = statement.executeQuery();

        while (resultSet.next()) {
            int id = resultSet.getInt("turbina_id");
            String modelo = resultSet.getString("modelo");
            double potenciaMax = resultSet.getDouble("potencia_max");
            String estado = resultSet.getString("estado");

            Turbina turbina = new Turbina(id, modelo, potenciaMax, estado);
            turbinas.add(turbina);
        }
    } catch (SQLException e) {
        System.err.println("Error al consultar las turbinas disponibles: " + e.getMessage());
    }
    return turbinas;
}
```

Fuente: elaboración propia

Clase «VistaRegistroViento»

Es parte del patrón MVC y se utiliza para manejar la interacción con el usuario con el fin de registrar las mediciones de viento.

En esta clase, se realiza la importación de la interfaz «List» del paquete *java.util*. Se utiliza para representar colecciones ordenadas de elementos, en este caso, la lista de turbinas.

Adicionalmente, se realiza la importación de la clase «Random», que también es parte del paquete java.util y proporciona métodos para generar números aleatorios. Se va a utilizar para simular valores aleatorios, como la velocidad del viento o la dirección del viento.

Contiene el constructor y los siguientes métodos:

- Método solicitarIdTurbina()

Figura 15. Método solicitarIdTurbina()

```
public int solicitarIdTurbina() {  
    System.out.print("Seleccionar turbina a registrar ");  
    return scanner.nextInt();  
}
```

Fuente: elaboración propia

- Método solicitarVelocidadViento()

Figura 16. Método solicitarVelocidadViento()

```
public double solicitarVelocidadViento() {  
  
    double velocidadViento = Math.random() * 50;  
    System.out.println("Velocidad detectada por el sensor: " + velocidadViento);  
    return velocidadViento;  
}
```

Fuente: elaboración propia

- Método mostrarTurbinas()

Figura 17. Método mostrarTurbinas()

```
public void mostrarTurbinas(List<Turbina> turbinas) {  
    System.out.println("_____");  
    System.out.println("Turbinas Disponibles:");  
    System.out.println("_____");  
    System.out.println("ID | Modelo | Pot. Máxima | Estado");  
  
    for (Turbina turbina : turbinas) {  
        System.out.println(turbina.getId() + " | " + turbina.getModelo() + " | " +  
                           turbina.getPotenciaMax() + " | " + turbina.getEstado());  
    }  
    System.out.println("_____");  
}
```

Fuente: elaboración propia

- Método solicitarDireccionViento()

Figura 18. Método solicitarDireccionViento()

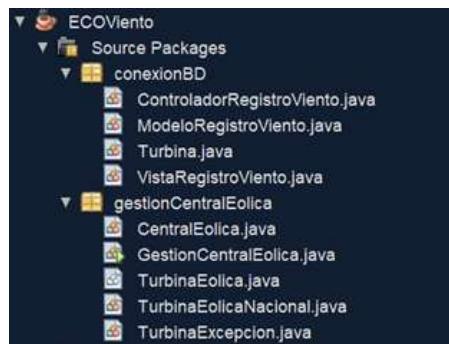
```
public String solicitarDireccionViento() {  
    String[] array = {"Norte", "Sur", "Este", "Oeste"};  
    Random random = new Random();  
    int indiceAleatorio = random.nextInt(array.length);  
  
    System.out.println("Dirección detectada por el sensor: " + array[indiceAleatorio]);  
    return array[indiceAleatorio];  
}
```

Fuente: elaboración propia

Clase «GestionCentralEolica»

Es la clase principal, que permite gestionar el menú para realizar distintas opciones propias del MVP que se está desarrollando. Recordemos que esta clase se encuentra en el paquete gestionCentralEolica. La estructura del proyecto es la siguiente:

Figura 19. Estructura del proyecto



En esta estructura se agrega una nueva opción que va a permitir que persistan, en la base de datos MySQL, los datos de condiciones del viento.

Figura 20. Nueva opción para persistir los datos de condiciones del viento

```
public class GestionCentralEolica {
    public static void main(String[] args) throws TurbinaExpcion {
        Scanner scanner = new Scanner(System.in);

        CentralEolica central = new CentralEolica("ECOViento");
        boolean ejecutando = true;

        while (ejecutando) {
            System.out.println("_____");
            System.out.println("MENU PRINCIPAL - ECOViento");
            System.out.println("_____");
            System.out.println("1. Agregar Turbina");
            System.out.println("2. Iniciar Turbina(s)");
            System.out.println("3. Detener Turbina(s)");
            System.out.println("4. Listar turbinas");
            System.out.println("5. Calcular Energía Generada");
            System.out.println("6. Almacenar mediciones de viento");
            System.out.println("7. Salir");
            System.out.println("_____");
            System.out.print("Elige una opción: ");
        }
    }
}
```

Fuente: elaboración propia

También, se agrega el manejo de una excepción, para que el usuario solo pueda seleccionar un número, evitando la interrupción del programa si se introduce, por ejemplo, una letra.

Figura 21. Manejo de una excepción

```
int opcion = -1;

try {
    opcion = scanner.nextInt();
} catch (java.util.InputMismatchException e) {
    System.err.println("Opción no válida, ingresa un número");
    scanner.nextLine();
}
```

Fuente: elaboración propia

Esta nueva opción establece una conexión a una base de datos MySQL, maneja posibles errores de conexión a la base de datos y utiliza un controlador del tipo «ControladorRegistroViento» para mostrar las turbinas disponibles y registrar mediciones de viento.

Figura 22. Uso de «ControladorRegistroViento»

```
case 6 -> {
    try (Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/ecoviento", "root", "")) {
        ControladorRegistroViento controlador = new ControladorRegistroViento(conexion);
        controlador.mostrarTurbinasDisponibles();
        controlador.ejecutar();
    } catch (SQLException e) {
        System.err.println("Error de conexión a la base de datos: " + e.getMessage());
    }
}
```

Fuente: elaboración propia

Clase «GestionCentralEolica»

Luego de registrar algunas mediciones de las condiciones del viento en la tabla «registros_viento» de la base de datos MySQL del proyecto, se puede realizar una consulta para confirmar que los registros se hayan impactado correctamente.

Figura 23. Clase «GestionCentralEolica»

SELECT * FROM registros_viento;				
viento_id	turbina_id	fecha_registro	velocidad_viento	direccion_viento
9	1	2023-10-07 10:13:24	15,0	Norte
10	1	2023-10-07 10:32:57	31,03	Este
11	3	2023-10-07 10:34:42	23,06	Norte
12	4	2023-10-07 10:44:40	18,9	Norte
13	1	2023-10-14 13:06:49	37,85	Este
14	3	2023-10-15 21:48:19	14,53	Sur
15	4	2023-10-15 21:48:31	35,27	Oeste
16	1	2023-10-15 21:48:40	47,34	Oeste

Fuente: elaboración propia

Presentación del desarrollo en Java

En esta instancia, ya estás en condiciones de presentar una nueva versión del prototipo, para lo cual te propongo utilizar GitHub: <https://github.com/>.

A continuación, se adjuntan las clases Java utilizadas para que puedas crear un proyecto en una IDE y comprobar la salida con la ejecución del programa desarrollado.

Paquete «GestionCentralEolica»: actualizar la clase «GestionCentralEolica»



Paquete conexionBD



Referencias

Álvarez Caules, C. (2020). Java ArrayList for y sus opciones. <https://www.arquitectura.rajava.com/java-arraylist-for-y-sus-opciones/>

Debrauwer, L. (2013). Patrones de diseño en Java. Ediciones ENI.

GitHub, (2021). La jerarquía de clases de E/S en Java. https://github.com/Suryakant-Bharti/Important-Java-Concepts/blob/master/_moreReadMe/input_output/README.md#the-hierarchy-of-io-class-in-java