**ESTRUCTURAS DE DATOS USANDO PATRÓN DE DISEÑO MODELO VISTA CONTROLADOR.**

****

**Presentado a:**

**Ing. Oswaldo Vélez Langs.**

**Presentado por:**

**Rafael Martínez Bravo.**

**Jhordy Barrera Ballesta.**

**UNIVERSIDAD DE CORDOBA**

**FACULTAD DE INGENIERIAS**

**INGENIERIA DE SISTEMAS**

**2018**

**TABLA DE CONTENIDO**

**Pág.**

[1. INTRODUCCIÓN. 3](#_Toc516036921)

[2. OBJETIVOS. 3](#_Toc516036922)

[3. MARCO TEÓRICO. 4](#_Toc516036923)

[3.1 Modelo Vista Controlador (MVC). 4](#_Toc516036924)

[3.2 Modelo. 4](#_Toc516036925)

[3.3 Vista. 4](#_Toc516036926)

[3.4 Controlador. 4](#_Toc516036927)

[3.5 Estructuras de datos. 4](#_Toc516036928)

[3.6 Listas enlazadas. 4](#_Toc516036929)

[3.7 Listas enlazadas dobles. 5](#_Toc516036930)

[3.8 Pilas. 6](#_Toc516036931)

[4. IMPLEMENTACIÓN 7](#_Toc516036932)

[4.1 Implementación Listas. 7](#_Toc516036933)

[4.1.1 Modelo Listas. (modelo.listas). 7](#_Toc516036934)

[4.2 Ejercicios con Listas Enlazadas. (controlador.ListasControlador). 7](#_Toc516036935)

[4.2.1 Agregar Nodo. (agregarNodo). 7](#_Toc516036936)

[4.2.2 Eliminar Lista. (eliminarLista). 7](#_Toc516036937)

[4.2.3 Máxima distancia. (maximaDistancia). 7](#_Toc516036938)

[4.2.4 Eliminar pares o impares. (delParImpar). 7](#_Toc516036939)

[4.2.5 Edad Promedio. (edadPromedio). 7](#_Toc516036940)

[4.2.6 Llenado automático. (autoLlenar). 8](#_Toc516036941)

[4.3 Implementación Pilas. 8](#_Toc516036942)

[4.4 Ejercicios con Pilas. 8](#_Toc516036943)

[4.4.1 Apilar. (apilar). 8](#_Toc516036944)

[4.4.2 Des apilar. (desapilar). 8](#_Toc516036945)

[4.4.3 Auto apilar. (autoApilar). 8](#_Toc516036946)

[4.4.4 Invertir Pila. (invertirPila). 8](#_Toc516036947)

[4.5 Implementación Colas. 8](#_Toc516036948)

[4.6 Ejercicios con colas. 8](#_Toc516036949)

[5. BIBLIOGRAFIA 9](#_Toc516036950)

# 1. INTRODUCCIÓN.

Antes de comenzar a hablar del MVC y de su uso con el desarrollo Web debemos hablar de las causas para que haya existido la relación entre ambos. Como programadores, generalmente enfrentamos problemas típicos en medio del desarrollo. Problemas que se vuelven más grandes si no hubo una planeación adecuada para nuestro proyecto (Como el diseño de las bases de datos, la estructuración y módulos del programa, que no se documentó el código y tener que buscar entre las miles y miles de líneas que se dejaron a la deriva para buscar el error en el algoritmo, etc.).

Es común ver casos donde tenemos una telaraña enorme de código donde tenemos entre líneas de código que hacemos para la interfaz, mejor conocida como front – end (que significa presentar al usuario esta interfaz para que pueda interactuar y recibir los datos del usuario sin ningún problema), las cuales no conectan con el back – end (donde este recibe los datos del front -end y los procesa para mandar un resultado que mostrara el front – end). También suele ocurrir que después de haber entregado un producto “decente” al cliente que nos pidió el proyecto, se le pueda ocurrir una nueva “idea” que quiera implementarse para una actualización para su proyecto, por lo que sino preparamos el código ya que es de ley que debe soportar los cambios bruscos que no solo pasaran en nuestras manos sino también de los demás programadores que toquen el código, el supuesto tiempo corto que deberíamos pasar para solo modificar un par de líneas de código se vuelve en una pesadilla infernal para los programadores.

Afortunadamente, en la industria contamos con mucha gente que se dedica a desarrollar metodologías para desarrollar correctamente nuestro código con la finalidad de que no termine siendo un monstruo horripilante que no queramos tocar de nuevo. Una de las ideas más aceptadas por varios programadores (si bien suele haber excepciones para cada caso) es el MVC.

MVC (Model – View – Controller, Modelo – Vista – Controlador) es un patrón de arquitectura de software que permite tener todo el código ordenado para que pueda soportar los cambios bruscos que requiera nuestro programa ya sea durante el desarrollo o el soporte que se le debe ir dando. El MVC propone construir tres componentes donde se diferencia cada uno con la separación de los datos, la lógica del negocio y la interfaz del usuario. Se explicará cada uno de estos.

# **2. OBJETIVOS**.

* Conocer y hacer uso del patrón de diseño MVC que permite organizar el acceso a datos y garantizar la integridad de estos.
* Conocer y manejar estructuras de datos dinámicas y sus funcionalidades.

# 3. MARCO TEÓRICO.

## 3.1 Modelo Vista Controlador (MVC).

Es un patrón de diseño que ha ganado popularidad en los últimos tiempos, el cual consiste en separar el acceso a datos de los datos, y a su vez de la presentación de estos.

## 3.2 Modelo.

También conocido como la lógica de negocio, la cual está conformada por los métodos y atributos que satisfacen el diseño base de cada componente de nuestra aplicación, así como también, la validación de estos. (Atributos, gets, sets, etc.).

## 3.3 Vista.

Es el encargado de interactuar con el usuario, realizando las tareas de captura de información o acciones, para solicitar al controlador los datos que son de interés del usuario, después de recibir una respuesta se encarga de actualizar la vista correspondiente del usuario.

## **3.4 Controlador**.

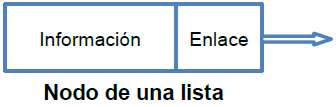
Este realiza las acciones de comunicador entre la vista y el modelo de datos, pero más concretamente es quien se ocupa de la seguridad e integridad de los datos, validar que los datos que se desean obtener estén permitidos, es posible crear un controlador de acceso para cada modelo, así como también controladores que no estén asociados a estos.

## 3.5 Estructuras de datos.

Existen estructuras de datos dinámicas que se utilizan para almacenar datos que están cambiando constantemente. A diferencia de los vectores, las estructuras de datos dinámicas se expanden y se contraen haciéndolas más flexibles a la hora de añadir o eliminar información.

## 3.6 Listas enlazadas.

Las listas enlazadas permiten almacenar información en posiciones de memoria que no sean contiguas. Estas listas para almacenar la información, contienen elementos llamados nodos. Estos nodos poseen dos campos, uno para almacenar la información o valor del elemento y otro para el enlace que determina la posición del siguiente elemento o nodo de la lista.



Al insertar o borrar información no es necesario realizar un desplazamiento; para esto los nodos de la lista cuentan con punteros o enlaces que contienen la posición o dirección del otro nodo que le sigue. Por esta razón no es necesario que los elementos de la lista se almacenen en posiciones contiguas.



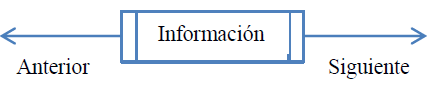
Cuando en una lista enlazada no hay ningún elemento quiere decir que la lista está vacía, además existe un puntero de cabecera para acceder al primer nodo de la lista.

Nos enfocaremos en el modelo de lista clásico, esto quiere decir que el primero que llega es el primero de la lista, y los nodos siguientes siempre se anexan al final. Esto no significa que no se puedan realizar las operaciones de inserción entre nodos, inserción al comienzo, etc.

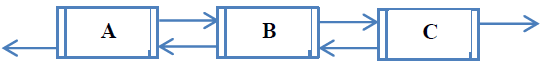
## 3.7 Listas enlazadas dobles.

Muchas veces las listas enlazadas simples no soportan la función de recorrer una lista desde el último nodo de la lista hasta la cabeza o inicio de la misma. Esto teniendo en cuenta que la opción de retroceder en las listas no se puede realizar con un solo enlace o referencia (siguiente).

En este caso si el objetivo es recorrer la lista en ambos sentidos, se deberían tener dos referencias o enlaces en cada nodo; es decir, cada nodo además de la información tendrá un enlace al siguiente nodo de la lista y otro al nodo anterior.



Como la implementación de estas listas tienen dos enlaces o referencias en cada nodo, reciben el nombre de doblemente enlazadas. En este sentido se podrán realizar búsquedas y recorridos en ambos sentidos.



## 3.8 Pilas.

Las pilas son estructuras de datos que almacenan un conjunto de valores y donde las operaciones de agregar y eliminar elementos se efectúan siempre por el tope o cima de la pila; por esta razón los primeros elementos en ser ingresados en la pila son los últimos en ser eliminados, y los últimos elementos en ser añadidos serán los primeros en ser eliminados. Debido a este comportamiento las pilas también se les conoce como listas LIFO (Last In, First Out), el último elemento en entrar, será primero en salir; o FILO (Fisrt In, Last Out), el primero en entrar, será el último en salir (Zahonero y Joyanes Aguilar, 1999).

Se pueden tomar como ejemplo una pila de libros, en la que un nuevo libro se pone encima de los otros y si usted va a quitar un libro de la pila procura retirar el que está más arriba, que corresponde al último que puso en la pila de libros.

La estructura de datos pila no tiene un medio propio para almacenar los datos o elementos que la componen, por lo que dependen de otras estructuras para guardar sus datos. El mecanismo utilizado para almacenar la información contenida en una pila se conoce como proceso de implantación de la pila. Para implementar la pila se utilizan arreglos o listas enlazadas simples como estructura auxiliar.

**3.9 Colas.**

Las colas son una estructura de datos que almacenan un conjunto de valores para los cuales las operaciones de agregar y eliminar datos se realizan en por los extremos opuestos de la estructura, de modo que los primeros elementos en ser ingresados en una cola son los primeros en ser extraídos o eliminados de ella; y así mismo los últimos elementos en ser agregados serán los últimos en ser sacados o eliminados.

A las colas se les conoce también como listas FIFO (First In, First Out) primero en entrar, primero en salir (Zahonero y Joyanes Aguilar, 1999).

Un ejemplo puede ser, el caso de la cola de atención de un banco o la cola para comprar las boletas para entrar a un juego de fútbol, en donde la primera persona en llegar es la primera en ser atendida y en consecuencia la primera en salir de la cola.

La estructura de datos cola, no tiene un medio propio con el cual almacenar sus datos, por lo que depende de otra estructura auxiliar para hacerlo. El proceso empleado para representar o guardar los datos de una cola se le llama implantación de la cola. Para esto lo más habitual es usar arreglos o listas enlazadas que permitan guardar los datos de la cola; de manera que la cola procesa los datos del arreglo o la lista de una forma particular; haciendo que las operaciones de agregar y eliminar datos, se hagan en extremos opuestos de la estructura con la cual se implanta.

# 4. IMPLEMENTACIÓN

## 4.1 Implementación Listas.

### 4.1.1 Modelo Listas. (modelo.listas).

La implementación de una lista parte del diseño de nodo, la información contenida por cada nodo ha sido realizada basada de un objeto Persona a partir de modelo.Persona, con su respectivo apuntador al siguiente nodo, además de los pertinentes constructores, getters y setters de cada clase.

## 4.2 Ejercicios con Listas Enlazadas. (controlador.ListasControlador).

### 4.2.1 Agregar Nodo. (agregarNodo).

Permite agregar un nodo a la lista, si está vacía permite agregar el primer nodo de la lista, por lo contrario, este se agregará al final de la lista.

### 4.2.2 Eliminar Lista. (eliminarLista).

Elimina la lista eliminando el apuntador del inicio de la lista, en java no es necesario preocuparse por rutinas de limpieza de memoria, ya que este se encarga de eliminar la memoria cuando deja de usarse.

### 4.2.3 Máxima distancia. (maximaDistancia).

Encuentra la máxima distancia entre 2 nodos con la misma edad, si se encuentra una sola vez no existe máxima distancia, de encontrar un segundo se guarda la máxima distancia y se continua hasta verificar el final de la lista.

### 4.2.4 Eliminar pares o impares. (delParImpar).

Dado un número par, se deben eliminar todos los nodos que contenga un numero de documento par, si se da un número impar se deben eliminar los nodos con números de documento impares.

### 4.2.5 Edad Promedio. (edadPromedio).

Calcular el promedio de todos los nodos de la lista, se debe validar si la lista está vacía para evitar divisiones por cero, de lo contrario sumamos todas las edades y las dividimos entre el total de nodos.

### 4.2.6 Llenado automático. (autoLlenar).

Permite el llenado automático de nodos, dada una cantidad de nodos, se agregarán automáticamente partiendo de la letra ‘a’ para nombre y apellido, y desde el número 1 para documento, teléfono, y edad.

## 4.3 Implementación Pilas.

Como las pilas no tienen su estructura de datos propia hacemos uso de listas para implementarla. Heredando de la clase ListaSimple, e implementando los métodos que se necesitan para realizar las operaciones.

## 4.4 Ejercicios con Pilas.

### 4.4.1 Apilar. (apilar).

Consiste en agregar un nuevo nodo a la pila, siempre por la cima. Si la pila está vacía se agrega el primer nodo de la pila, de lo contrario se agregará el siguiente a la cima de la pila.

### 4.4.2 Des apilar. (desapilar).

Se encarga de eliminar el nodo que se encuentra en la cima. Como se había mencionado antes, en las operaciones sobre pila solo se puede eliminar el nodo de la cima a la vez.

### 4.4.3 Auto apilar. (autoApilar).

Genera automáticamente nodos como el caso de las listas, en este caso siempre se añaden los nodos por la cima.

### 4.4.4 Invertir Pila. (invertirPila).

Como en el ejemplo de la pila de libros, para invertir la pila se pasa el nodo de la cima a una nueva pila hasta completar todos los elementos.

## 4.5 Implementación Colas.

Como las colas tampoco tienen su estructura de datos propias hacemos uso de listas para su implementación. Como en el ejemplo de la cola de un banco la cola se llena por el ultimo nodo y se elimina por el comienzo de esta.

### 4.6 Ejercicios con colas.

### 4.6.1 Encolar. (encolar).

Consiste en agregar un nuevo nodo a la cola, siempre final. Si la cola está vacía se agrega el primer nodo de la cola, de lo contrario se agregará el siguiente al final de la cola.

### 4.6.2 Des encolar. (desencolar).

Se encarga de eliminar el nodo que se encuentra al frente de la cola. Como se había mencionado antes, en las operaciones sobre cola se elimina el nodo del comienzo de la cola.

### 4.6.3 Auto encolar. (autoEncolar).

Genera automáticamente nodos como el caso de las listas, en este caso siempre se añaden los nodos por el final de la cola.

### 4.7 Validador.

Esta implementado para realizar la validación de datos al ser ingresados en jTextBox usando el evento keyTyped o ya sea utilizando try-catch para pedir números haciendo uso de JOptionPane.

### 4.7.1 Pedir Números. (pedirNumero).

Este método recibe una cadena de texto que se visualizara dentro del JOptionPane.showImputDialog, realiza la petición hasta que se valide correctamente el ingreso de un valor numérico entero, y luego, este es retornado.

### 4.7.2 Actualizar Tabla. (actualizarTabla).

Es el encargado de recibir el nodo del comienzo de la lista, pila ó cola para actualizar los datos que se muestran al usuario, los datos necesarios son: inicio (para el recorrido), tamaño (la cantidad de nodos), JTable, JLabel (muestra el total nodos).

# 5. BIBLIOGRAFIA

* Zahonero, I., y Joyanes Aguilar, L. (1999). Estructura de Datos - Algoritmos, Abstracción y Objetos. España: McGraw-Hill.
* Allen Weiss, M. (2004). Estructuras de datos en Java. España: Addison Wesley - Pearson. 776 pp.
* Guevara, P. Olascoaga, L. (n.d.). Métodos de Ordenamiento. Universidad de Córdoba, Departamento de Ingeniería de Sistemas Telecomunicaciones.