



UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE  
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS  
CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN



PERIODO	:	PREGRADO OCTUBRE 2025 - MARZO 2026
ASIGNATURA	:	Estructura De Datos
TEMA	:	Aplicación de Listas Enlazadas modelo MVC
ESTUDIANTE	:	Diego Montesdeoca y Angel Rodriguez
NIVEL-PARALELO - NRC	:	30748
DOCENTE	:	Margoth Elisa Guaraca Moyota
FECHA DE ENTREGA	:	25/10/2025

SANTO DOMINGO – ECUADOR

## Tabla de contenido

1.	Introducción.....	3
2.	Objetivos.....	3
2.1.	Objetivo General: .....	3
2.2.	Objetivos Específicos: .....	4
3.	Desarrollo / Marco Teórico/ Práctica.....	4
3.1.	Lista Enlazada Simple .....	4
3.2.	Patrón Modelo-Vista-Controlador (MVC) .....	5
3.3.	Estructura del Modelo .....	7
3.4.	Implementación del Controlador .....	13
3.5.	Implementación de la Vista .....	18
4.	Conclusiones.....	28
5.	Recomendaciones .....	29
6.	Bibliografía/ Referencias.....	30
7.	Anexos:.....	<b>¡Error! Marcador no definido.</b>

## Tabla de contenido

Figura 3.1	Diagrama UML .....	6
Figura 3.2	Definición de la clase Movie.....	7
Figura 3.3	Metodos Getter and setter de movie.java.....	8
Figura 3.4	Método toString de movie.java.....	8
Figura 3.5	Implementación de la clase Node .....	9
Figura 3.6	Estructura basica de la clase LinkedList .....	10
Figura 3.7	Métodos de inserción en la lista enlazada .....	11
Figura 3.8	Método de eliminación por título.....	12
Figura 3.9	Método de búsqueda y obtención de películas.....	13
Figura 3.10	Constructor y configuración del MovieController .....	14
Figura 3.11	Manejo de eventos ActionListener .....	14
Figura 3.12	Listeners de selección y documento.....	15
Figura 3.13	Movimiento de película al inicio .....	16
Figura 3.14	Movimiento de película al final .....	16
Figura 3.15	Validación de datos en saveMovie() .....	17
Figura 3.16	Actualización e inserción de películas .....	17
Figura 3.17	Método de eliminación de películas.....	18
Figura 3.18	Ejecucion inicial.....	19
Figura 3.19	Ventana emergente al guardar .....	20
Figura 3.20	Selección en tabla se completan los datos.....	20

Figura 3.21 Movimiento de dato al inicio .....	21
Figura 3.22 Movimiento de dato al final .....	22
Figura 3.23 Elimina un dato de la tabla.....	26
Figura 3.24 Validación de datos y ventanas emergentes .....	27
Figura 3.25 Validacion fecha 1888-present .....	28

## **1. Introducción**

El presente informe detalla el desarrollo de una aplicación de software destinada a la gestión dinámica de una colección de objetos de tipo "Película". El proyecto se concibió como un ejercicio práctico en el campo de las estructuras de datos, centrándose en la implementación de una Lista Enlazada Simple para el almacenamiento y manipulación de la información.

Para asegurar una arquitectura de software robusta, escalable y modular, se adoptó el patrón de diseño Modelo-Vista-Controlador (MVC). El Modelo encapsula la estructura de datos (Lista Enlazada Simple) y la lógica de negocio; la Vista se encarga de la interfaz gráfica de usuario (GUI); y el Controlador actúa como el puente que gestiona las interacciones del usuario y coordina las actualizaciones entre el Modelo y la Vista.

La aplicación final permite a un usuario realizar operaciones fundamentales sobre la colección de películas, tales como la inserción, eliminación y búsqueda de elementos, demostrando la operatividad de la lista enlazada simple en un entorno de desarrollo Java con interfaz gráfica.

## **2. Objetivos**

### **2.1. Objetivo General:**

Desarrollar una aplicación de gestión de información en el lenguaje de programación Java, implementando una Lista Enlazada Simple, con el patrón MVC para la organización.

## **2.2. Objetivos Específicos:**

- Implementar una lista enlazada simple en Java con todas sus operaciones básicas (inserción al inicio/final, eliminación, búsqueda y recorrido) para gestionar un catálogo de películas.
- Desarrollar una interfaz gráfica utilizando el patrón MVC que permita visualizar y manipular los datos de las películas, incluyendo funcionalidades CRUD, búsqueda en tiempo real y reorganización de elementos.
- Integrar mecanismos de validación de datos en el formulario de entrada para garantizar la integridad de la información, verificando campos obligatorios, formato del año y evitando duplicados.

## **3. Desarrollo / Marco Teórico/ Práctica**

El proyecto se sustenta en dos pilares teóricos principales: la Lista Enlazada Simple y el patrón Modelo-Vista-Controlador (MVC).

### **3.1. Lista Enlazada Simple**

Una lista enlazada es una colección de elementos, denominados nodos, donde el orden no está dado por su ubicación física contigua en la memoria (como en los arreglos), sino por referencias explícitas. En una Lista Enlazada Simple, cada nodo posee dos campos: uno para almacenar la información relevante (el dato, en este caso, un objeto Movie) y otro campo, denominado enlace o next, que contiene la dirección de memoria del nodo siguiente en la

secuencia. El último nodo de la lista tiene su campo de enlace apuntando a un valor null, lo que señala el final de la estructura. La lista se gestiona mediante una referencia principal conocida como cabeza (head), que apunta al primer nodo.

Las Listas Enlazadas Simples ofrecen una ventaja significativa sobre las estructuras estáticas como los arrays al permitir una expansión dinámica sin requerir memoria extra para reacomodar elementos, además de facilitar la inserción y eliminación de nodos de forma más eficiente en términos de movimiento de datos.

### **3.2. Patrón Modelo-Vista-Controlador (MVC)**

El patrón MVC es una arquitectura de diseño que separa la representación de la información de la interacción del usuario con dicha información, promoviendo la reutilización del código y la organización:

Modelo (Model): Contiene la lógica del negocio y la estructura de datos (LinkedList.java, Node.java, Movie.java). Es la capa responsable de gestionar el estado de la aplicación.

Vista (View): Presenta los datos al usuario y captura sus entradas (MainWindow.java, FormPanel.java, ListPanel.java). Es la capa de la interfaz gráfica.

Controlador (Controller): Responde a los eventos del usuario (acciones en la Vista) y determina qué lógica del Modelo debe ejecutarse, actualizando la Vista con los resultados (MovieController.java).

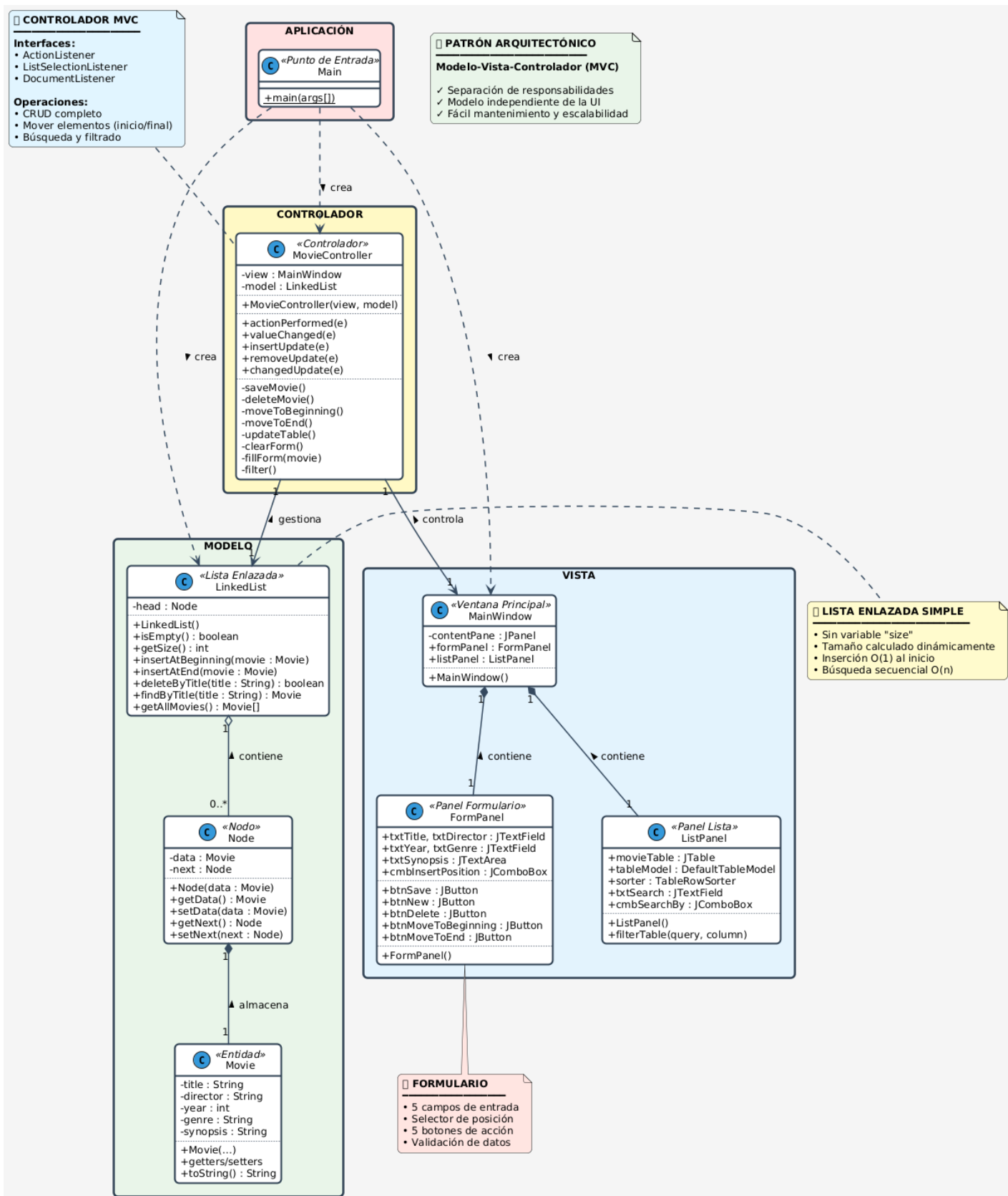


Figura 3.1 Diagrama UML

## Práctica (Análisis de la Implementación)

La implementación se realizó en Eclipse Java y se estructuró en los paquetes Model, View, Controller y App, conforme a los requisitos de la práctica.

### 3.3. Estructura del Modelo

**Movie.java:** Se definió con atributos para almacenar el título, director, año, género y sinopsis de la película, junto con sus métodos getter y setter.

```
1 package Model;
2 public class Movie {
3
4     private String title;
5     private String director;
6     private int year;
7     private String genre;
8     private String synopsis;
9
10    public Movie(String title, String director, int year, String genre, String synopsis) {
11        this.title = title;
12        this.director = director;
13        this.year = year;
14        this.genre = genre;
15        this.synopsis = synopsis;
16    }
```

Figura 3.2 Definición de la clase Movie

```

17 // Getters and Setters
18 public String getTitle() {
19     return title;
20 }
21 public void setTitle(String title) {
22     this.title = title;
23 }
24 public String getDirector() {
25     return director;
26 }
27 public void setDirector(String director) {
28     this.director = director;
29 }
30 public int getYear() {
31     return year;
32 }
33 public void setYear(int year) {
34     this.year = year;
35 }
36 public String getGenre() {
37     return genre;
38 }
39 public void setGenre(String genre) {
40     this.genre = genre;
41 }
42 public String getSynopsis() {
43     return synopsis;
44 }
45 public void setSynopsis(String synopsis) {
46     this.synopsis = synopsis;
47 }

```

Figura 3.3 Metodos Getter and setter de movie.java

```

49 @Override
50 public String toString() {
51     return this.title + " (" + this.year + ")";
52 }
53 }

```

Figura 3.4 Método toString de movie.java

**Node.java:** Cada nodo almacena un objeto Movie y una referencia next de tipo Node, confirmando la implementación de una Lista Enlazada Simple.



```

1 package Model;
2 public class Node {
3
4     private Movie data;
5     private Node next;
6
7     // Constructor
8     public Node(Movie data) {
9         this.data = data;
10        this.next = null;
11    }
12
13    // Getters and Setters
14    public Movie getData() {
15        return data;
16    }
17
18    public void setData(Movie data) {
19        this.data = data;
20    }
21
22    public Node getNext() {
23        return next;
24    }
25
26    public void setNext(Node next) {
27        this.next = next;
28    }
29 }

```

Figura 3.5 Implementación de la clase Node

**LinkedList.java:** Implementó las operaciones clave:

**getSize():** Calcula el tamaño de la lista de forma dinámica mediante un recorrido completo desde head hasta null, contando cada nodo. Esta implementación evita el uso de una variable auxiliar size, privilegiando el cálculo bajo demanda.

```

1 package Model;
2 public class LinkedList {
3     private Node head;
4     public LinkedList() {
5         this.head = null;
6     }
7     public boolean isEmpty() {
8         return head == null;
9     }
10    public int getSize() {
11        int count = 0;
12        Node current = head;
13        while (current != null) {
14            count++;
15            current = current.getNext();
16        }
17        return count;
18    }

```

Figura 3.6 Estructura básica de la clase LinkedList

**insertAtBeginning(Movie movie):** La inserción requiere crear un nuevo nodo y reasignar la referencia head para que apunte al nuevo nodo, y el nuevo nodo apunte al nodo previamente en la cabeza.

**insertAtEnd(Movie movie):** Requiere recorrer la lista desde head hasta que el campo next del nodo actual sea null, y luego asignar el nuevo nodo a ese campo.

```

19 // Insert at the beginning of the list
20 public void insertAtBeginning(Movie movie) {
21     Node newNode = new Node(movie);
22     newNode.setNext(this.head);
23     this.head = newNode;
24 }
25 // Insert at the end of the list
26 public void insertAtEnd(Movie movie) {
27     Node newNode = new Node(movie);
28     if (isEmpty()) {
29         this.head = newNode;
30     } else {
31         Node current = this.head;
32         while (current.getNext() != null) {
33             current = current.getNext();
34         }
35         current.setNext(newNode);
36     }
37 }

```

Figura 3.7 Métodos de inserción en la lista enlazada

**deleteByTitle(String title):** Se debe localizar el nodo a eliminar y el nodo anterior.

Una vez localizado, se redefine el puntero next del nodo anterior para que apunte al sucesor del nodo a eliminar, excluyéndolo de la lista. Se maneja un caso especial si el nodo a eliminar es la head.

```

38 // Delete an element by its title
39 public boolean deleteByTitle(String title) {
40     if (isEmpty()) {
41         return false;
42     }
43     if (head.getData().getTitle().equalsIgnoreCase(title)) {
44         head = head.getNext();
45         return true;
46     }
47
48     Node previous = head;
49     Node current = head.getNext();
50     while (current != null && !current.getData().getTitle().equalsIgnoreCase(title)) {
51         previous = current;
52         current = current.getNext();
53     }
54     if (current != null) {
55         previous.setNext(current.getNext());
56         return true;
57     }
58     return false;
59 }

```

Figura 3.8 Método de eliminación por título

**findByTitle(String title):** Implica un recorrido secuencial de la lista, comparando el título del objeto Movie en cada nodo, retornando el objeto Movie si se encuentra, o null en caso contrario.

**getAllMovies():** Realiza un recorrido de la lista para obtener todos los objetos Movie en un arreglo, para su posterior visualización en la tabla de la interfaz.

```

60 // Find a node by title
61 public Movie findByTitle(String title) {
62     Node current = head;
63     while (current != null) {
64         if (current.getData().getTitle().equalsIgnoreCase(title)) {
65             return current.getData();
66         }
67         current = current.getNext();
68     }
69     return null;
70 }
71 // Get all movies to display them
72 public Movie[] getAllMovies() {
73     if (isEmpty()) {
74         return new Movie[0];
75     }
76     int totalSize = getSize();
77     Movie[] movies = new Movie[totalSize];
78     Node current = head;
79     int i = 0;
80     while (current != null) {
81         movies[i] = current.getData();
82         current = current.getNext();
83         i++;
84     }
85     return movies;
86 }
87 }

```

Figura 3.9 Método de búsqueda y obtención de películas

### 3.4. Implementación del Controlador

El **MovieController.java** implementa las interfaces **ActionListener**, **ListSelectionListener** y **DocumentListener** para capturar las acciones de los botones de la interfaz, la selección de filas en la tabla y los cambios en el campo de búsqueda. A través de los métodos de la clase **LinkedList**, ejecuta la lógica de negocio. También valida la entrada de datos (p. ej., que el año sea un número válido) antes de interactuar con el Modelo, y se comunica con la Vista para mostrar mensajes de éxito o error al usuario mediante **JOptionPane**.

```

1 package Controller;
2+ import Model.LinkedList;
14
15 public class MovieController implements ActionListener, ListSelectionListener, DocumentListener {
16
17     private MainWindow view;
18     private LinkedList model;
19
20 public MovieController(MainWindow view, LinkedList model) {
21     this.view = view;
22     this.model = model;
23
24     // Register listeners
25     this.view.formPanel.btnSave.addActionListener(this);
26     this.view.formPanel.btnNew.addActionListener(this);
27     this.view.formPanel.btnDelete.addActionListener(this);
28     this.view.formPanel.btnMoveToBeginning.addActionListener(this);
29     this.view.formPanel.btnMoveToEnd.addActionListener(this);
30     this.view.listPanel.movieTable.getSelectionModel().addListSelectionListener(this);
31     this.view.listPanel.txtSearch.getDocument().addDocumentListener(this);
32     this.view.listPanel.cmbSearchBy.addActionListener(this);
33
34     // Update table (will be empty at start)
35     updateTable();
36 }
--

```

Figura 3.10 Constructor y configuración del MovieController

```

38     // --- Listeners ---
39
40     @Override
41 public void actionPerformed(ActionEvent e) {
42     Object source = e.getSource();
43     if (source == view.formPanel.btnSave) {
44         saveMovie();
45     } else if (source == view.formPanel.btnNew) {
46         clearForm();
47     } else if (source == view.formPanel.btnDelete) {
48         deleteMovie();
49     } else if (source == view.formPanel.btnMoveToBeginning) {
50         moveToBeginning();
51     } else if (source == view.formPanel.btnMoveToEnd) {
52         moveToEnd();
53     } else if (source == view.listPanel.cmbSearchBy) {
54         filter();
55     }
56 }

```

Figura 3.11 Manejo de eventos ActionListener

```

58     @Override
59     public void valueChanged(ListSelectionEvent e) {
60         if (!e.getValueIsAdjusting()) {
61             int selectedRow = view.listPanel.movieTable.getSelectedRow();
62             if (selectedRow != -1) {
63                 // Convert view index to model index (in case it's filtered)
64                 int modelRow = view.listPanel.movieTable.convertRowIndexToModel(selectedRow);
65                 String title = (String) view.listPanel.tableModel.getValueAt(modelRow, 0);
66
67                 Movie movie = model.findByTitle(title);
68                 if (movie != null) {
69                     fillForm(movie);
70                 }
71             }
72         }
73     }
74
75     @Override
76     public void insertUpdate(DocumentEvent e) {
77         filter();
78     }
79
80     @Override
81     public void removeUpdate(DocumentEvent e) {
82         filter();
83     }
84
85     @Override
86     public void changedUpdate(DocumentEvent e) {
87         filter();
88     }
89

```

Figura 3.12 Listeners de selección y documento

Adicionalmente, el controlador implementa métodos especializados para la manipulación de la posición de elementos en la lista:

**moveToBeginning():** Permite mover una película existente desde cualquier posición de la lista hacia el inicio. El método elimina el elemento de su posición actual utilizando `deleteByTitle()` y lo reinserta al principio mediante `insertAtBeginning()`, manteniendo la integridad de la estructura de datos.

```

243 private void moveToBeginning() {
244     int selectedRow = view.listPanel.movieTable.getSelectedRow();
245     if (selectedRow == -1) {
246         JOptionPane.showMessageDialog(view, "Please select a movie from the list to move.", "No Movie Selected", JOptionPane.WARNING_MESSAGE);
247         return;
248     }
249
250     // Get the movie title from the selected row
251     int modelRow = view.listPanel.movieTable.convertRowIndexToModel(selectedRow);
252     String title = (String) view.listPanel.tableModel.getValueAt(modelRow, 0);
253
254     // Find the movie in the list
255     Movie movie = model.findByTitle(title);
256     if (movie == null) {
257         JOptionPane.showMessageDialog(view, "Movie not found.", "Error", JOptionPane.ERROR_MESSAGE);
258         return;
259     }
260
261     // Remove from current position and insert at beginning
262     model.deleteByTitle(title);
263     model.insertAtBeginning(movie);
264
265     JOptionPane.showMessageDialog(view, "Movie moved to beginning successfully.", "Move Successful", JOptionPane.INFORMATION_MESSAGE);
266     updateTable();
267     clearForm();
268 }

```

Figura 3.13 Movimiento de película al inicio

**moveToEnd():** Similar al anterior, pero mueve el elemento seleccionado hacia el final de la lista usando insertAtEnd(). Esta funcionalidad demuestra la flexibilidad de las listas enlazadas para reorganizar elementos sin necesidad de desplazar bloques grandes de memoria.

```

270 private void moveToEnd() {
271     int selectedRow = view.listPanel.movieTable.getSelectedRow();
272     if (selectedRow == -1) {
273         JOptionPane.showMessageDialog(view, "Please select a movie from the list to move.", "No Movie Selected", JOptionPane.WARNING_MESSAGE);
274         return;
275     }
276
277     // Get the movie title from the selected row
278     int modelRow = view.listPanel.movieTable.convertRowIndexToModel(selectedRow);
279     String title = (String) view.listPanel.tableModel.getValueAt(modelRow, 0);
280
281     // Find the movie in the list
282     Movie movie = model.findByTitle(title);
283     if (movie == null) {
284         JOptionPane.showMessageDialog(view, "Movie not found.", "Error", JOptionPane.ERROR_MESSAGE);
285         return;
286     }
287
288     // Remove from current position and insert at end
289     model.deleteByTitle(title);
290     model.insertAtEnd(movie);
291
292     JOptionPane.showMessageDialog(view, "Movie moved to end successfully.", "Move Successful", JOptionPane.INFORMATION_MESSAGE);
293     updateTable();
294     clearForm();
295 }
296 }

```

Figura 3.14 Movimiento de película al final

**saveMovie():** Este método se encarga de crear nuevas películas y permite al usuario seleccionar la posición de inserción mediante el ComboBox cmbInsertPosition, que ofrece las opciones 'Insert at Beginning' e 'Insert at End'. Dependiendo de la selección del usuario, invoca insertAtBeginning() o insertAtEnd() del modelo.



```

90 // --- Business Logic Methods ---
91
92 private void saveMovie() {
93     String title = view.formPanel.txtTitle.getText().trim();
94     String director = view.formPanel.txtDirector.getText().trim();
95     String yearStr = view.formPanel.txtYear.getText().trim();
96     String genre = view.formPanel.txtGenre.getText().trim();
97     String synopsis = view.formPanel.txtSynopsis.getText();
98
99     if (title.isEmpty() || director.isEmpty() || yearStr.isEmpty() || genre.isEmpty()) {
100         JOptionPane.showMessageDialog(view, "Please complete all fields (Title, Director, Year, Genre).", "Incomplete Fields", JOptionPane.WARNING_MESSAGE);
101         return;
102     }
103
104     int year;
105     try {
106         if (!Pattern.matches("\\d{4}", yearStr)) {
107             JOptionPane.showMessageDialog(view, "Year must be a 4-digit number.", "Invalid Year Format", JOptionPane.ERROR_MESSAGE);
108             return;
109         }
110         year = Integer.parseInt(yearStr);
111         if (year < 1888 || year > java.time.Year.now().getValue() + 1) {
112             JOptionPane.showMessageDialog(view, "Please enter a valid year (between 1888 and present).", "Invalid Year", JOptionPane.ERROR_MESSAGE);
113             return;
114         }
115     } catch (NumberFormatException ex) {
116         JOptionPane.showMessageDialog(view, "Year must be a valid number.", "Invalid Year Format", JOptionPane.ERROR_MESSAGE);
117         return;
118     }
119
120     Movie existingMovie = model.findByTitle(title);
121     if (existingMovie != null && view.formPanel.txtTitle.isEditable()) {
122         JOptionPane.showMessageDialog(view, "A movie with that title already exists.", "Duplicate Title", JOptionPane.ERROR_MESSAGE);
123         return;
124     }
125
126

```

Figura 3.15 Validación de datos en saveMovie()

```

125
126     if (existingMovie != null && !view.formPanel.txtTitle.isEditable()) {
127         // Update existing movie
128         existingMovie.setDirector(director);
129         existingMovie.setYear(year);
130         existingMovie.setGenre(genre);
131         existingMovie.setSynopsis(synopsis);
132         JOptionPane.showMessageDialog(view, "Movie updated successfully.", "Update Successful", JOptionPane.INFORMATION_MESSAGE);
133         updateTable();
134     } else {
135         // Create new movie and insert at selected position
136         Movie newMovie = new Movie(title, director, year, genre, synopsis);
137
138         // Check the selected insert position from ComboBox
139         String insertPosition = (String) view.formPanel.cmbInsertPosition.getSelectedItem();
140         if ("Insert at Beginning".equals(insertPosition)) {
141             model.insertAtBeginning(newMovie);
142         } else {
143             model.insertAtEnd(newMovie);
144         }
145
146         JOptionPane.showMessageDialog(view, "Movie saved successfully at " + insertPosition.toLowerCase() + ".", "Save Successful", JOptionPane.INFORMATION_MESSAGE);
147         updateTable();
148     }
149
150     clearForm();
151 }
152

```

Figura 3.16 Actualización e inserción de películas

**deleteMovie():** gestiona la eliminación de películas de la lista mediante una interfaz de confirmación modal. Primero verifica que el usuario haya seleccionado una película de la tabla, luego obtiene el título de la fila seleccionada y muestra un cuadro de diálogo de confirmación. Si el usuario confirma la acción, invoca al modelo para eliminar la película por título, actualiza la tabla para reflejar los cambios y limpia el formulario. En caso de error durante la eliminación, notifica al usuario mediante un mensaje emergente.

```

153 private void deleteMovie() {
154     int selectedRow = view.listPanel.movieTable.getSelectedRow();
155     if (selectedRow == -1) {
156         JOptionPane.showMessageDialog(view, "Please select a movie from the list to delete.", "No Movie Selected", JOptionPane.WARNING_MESSAGE);
157         return;
158     }
159
160     int modelRow = view.listPanel.movieTable.convertRowIndexToModel(selectedRow);
161     String title = (String) view.listPanel.tableModel.getValueAt(modelRow, 0);
162     int response = JOptionPane.showConfirmDialog(view, "Are you sure you want to delete the movie '" + title + "'", "Confirm Deletion", JOptionPane.YES_NO_OPTION);
163
164     if (response == JOptionPane.YES_OPTION) {
165         boolean deleted = model.deleteByTitle(title);
166         if (deleted) {
167             JOptionPane.showMessageDialog(view, "Movie deleted successfully.", "Deletion Successful", JOptionPane.INFORMATION_MESSAGE);
168             updateTable();
169             clearForm();
170         } else {
171             JOptionPane.showMessageDialog(view, "Could not delete the movie.", "Error", JOptionPane.ERROR_MESSAGE);
172         }
173     }
174 }

```

Figura 3.17 Método de eliminación de películas

### 3.5. Implementación de la Vista

La Vista se compone de tres clases principales:

**MainWindow.java:** Ventana principal (JFrame) que contiene los paneles FormPanel y ListPanel en un diseño GridLayout.

**FormPanel.java:** Panel que contiene los campos de texto para ingresar los datos de las películas (título, director, año, género, sinopsis), un ComboBox para seleccionar la posición de inserción (al inicio o al final), y los botones de acción: Guardar, Nuevo, Eliminar, Mover al Inicio y Mover al Final.

**ListPanel.java:** Panel que contiene la tabla JTable para mostrar el catálogo de películas, con funcionalidad de búsqueda mediante un campo de texto y un ComboBox para seleccionar el criterio de búsqueda (Título, Género, Año).

### 3.6. Funcionalidades Avanzadas de Manipulación de Lista

La aplicación implementa funcionalidades que demuestran las ventajas de las listas enlazadas para la reorganización dinámica de elementos:

**Selección de Posición de Inserción:** A través del componente cmbInsertPosition (JComboBox), el usuario puede elegir si desea insertar una nueva película al inicio o al final

de la lista. Esta funcionalidad expone directamente al usuario las dos operaciones fundamentales de inserción de la lista enlazada.

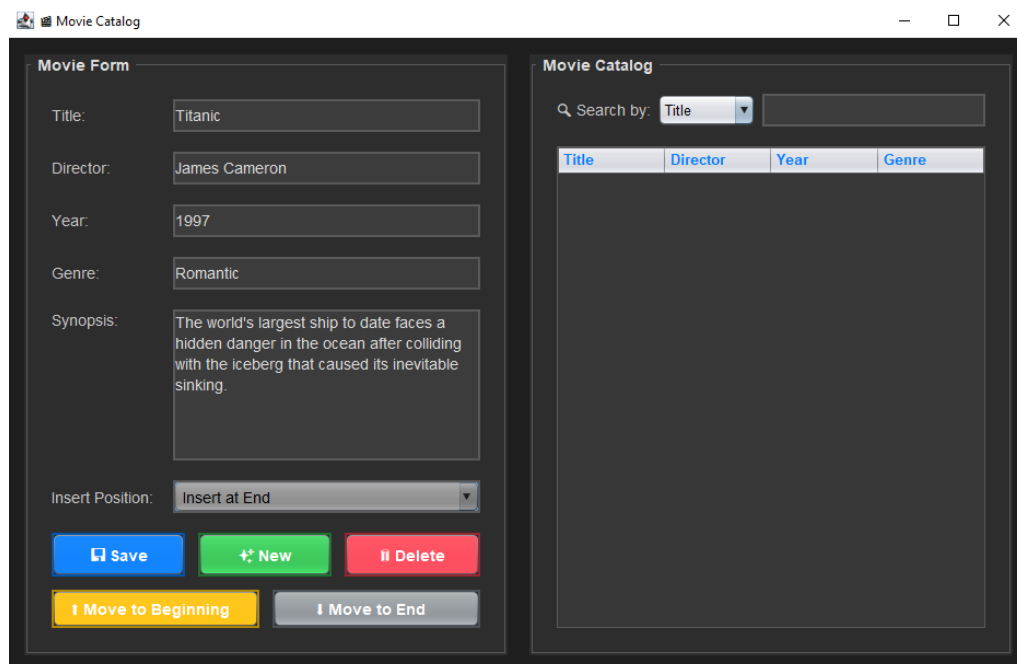
**Reorganización de Elementos Existentes:** Los botones "Move to Beginning" y "Move to End" permiten al usuario reorganizar el catálogo moviendo películas existentes a diferentes posiciones. Internamente, esto se logra mediante:

1. Búsqueda del elemento mediante `findByTitle()`
2. Eliminación de la posición actual mediante `deleteByTitle()`
3. Reinserción en la nueva posición (`insertAtBeginning()` o `insertAtEnd()`)
4. Actualización inmediata de la tabla para reflejar el cambio visual

Esta funcionalidad demuestra que las listas enlazadas son especialmente eficientes para reorganizar elementos, ya que solo requiere modificar referencias (punteros), sin necesidad de copiar o mover bloques de memoria como ocurriría con arrays.

## Ejecución del código

Interfaz Ejecutada (con datos ingresados)



The screenshot displays a desktop application window titled "Movie Catalog". It is divided into two main panels. The left panel, titled "Movie Form", contains input fields for "Title" (filled with "Titanic"), "Director" (filled with "James Cameron"), "Year" (filled with "1997"), and "Genre" (filled with "Romantic"). There is also a text area for "Synopsis" containing the text: "The world's largest ship to date faces a hidden danger in the ocean after colliding with the iceberg that caused its inevitable sinking." Below these fields is a dropdown menu for "Insert Position" set to "Insert at End". At the bottom of the form are five buttons: "Save" (blue), "New" (green), "Delete" (red), "Move to Beginning" (yellow), and "Move to End" (grey). The right panel, titled "Movie Catalog", features a search bar with a dropdown menu set to "Title" and an empty input field. Below the search bar is a table with four columns: "Title", "Director", "Year", and "Genre". The table is currently empty.

Figura 3.18 Ejecucion inicial

## Interfaz con datos en la tabla de registro de las películas

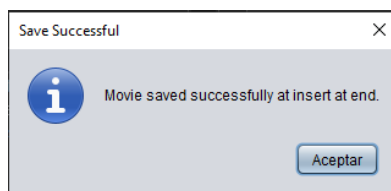


Figura 3.19 Ventana emergente al guardar

Una vez guardado al dar click en la fila de cualquier película, le hacen los datos automáticamente a la izquierda.

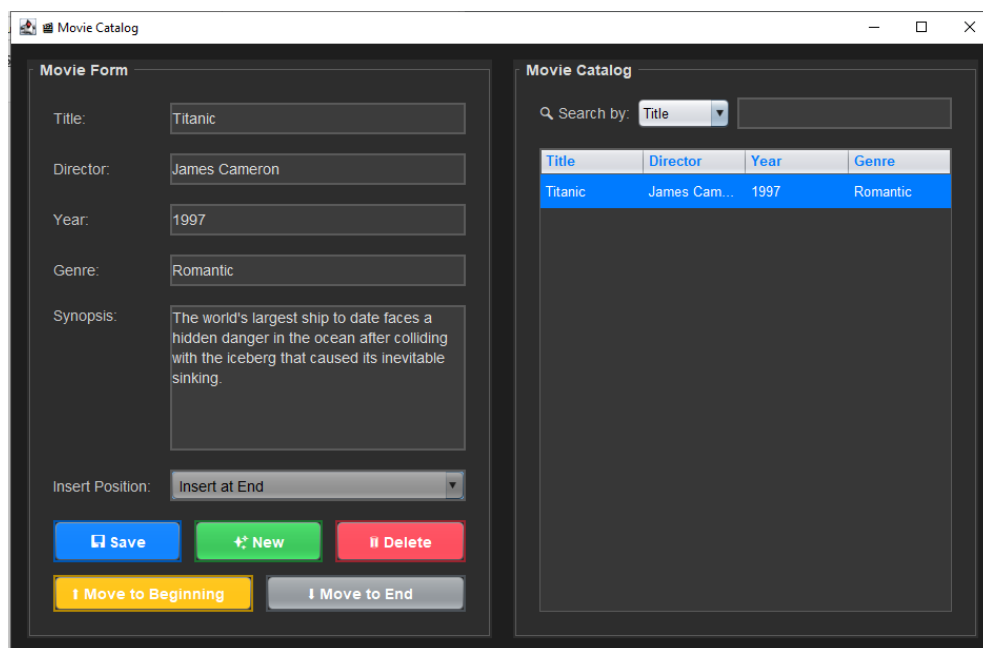
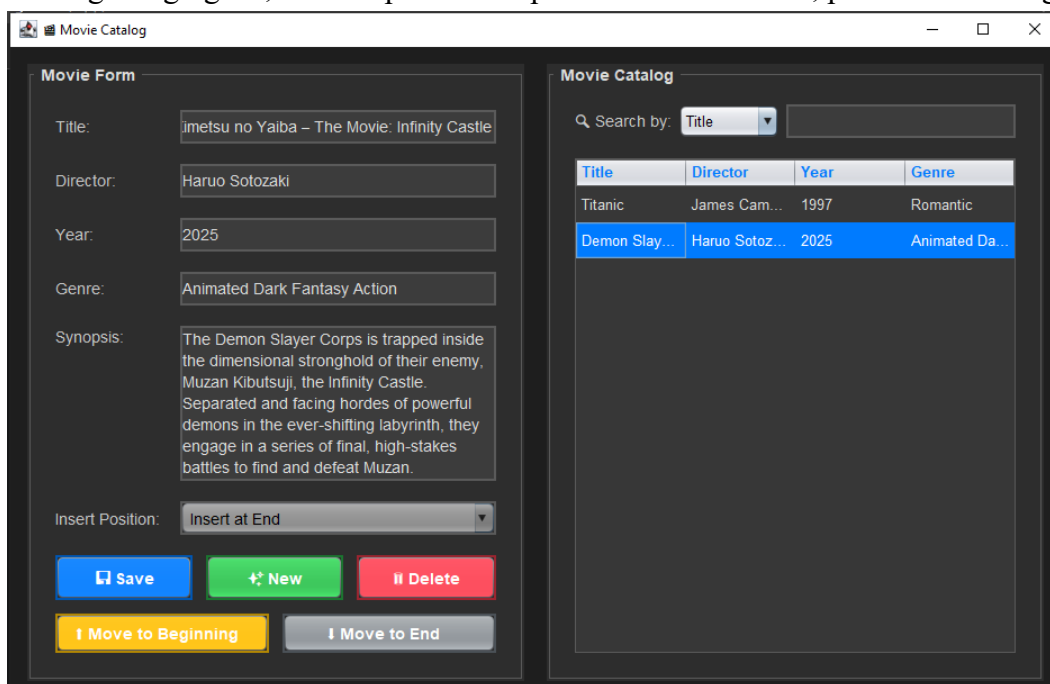


Figura 3.20 Selección en tabla se completan los datos

## Opción New (Insert at End)

No tiene lógica agregada, solo limpia los campos con un solo botón, para un nuevo ingreso.



Opción New (Insert at Beginning)

Movie Form

Title:

Inception

Director:

Christopher Nolan

Year:

2010

Genre:

Science Fiction / Action / Thriller

Synopsis:

A skilled professional thief who steals corporate secrets by infiltrating the subconscious minds of his targets is offered a chance to have his criminal history erased if he can successfully plant an idea into a target's mind—a process known as "inception."

Insert Position:

Insert at Beginning

Save

New

Delete

Move to Beginning

Move to End

Movie Catalog

Search by:

Title

Title	Director	Year	Genre
Titanic	James Cam...	1997	Romantic
Inception	Christopher ...	2010	Science Fict...
Demon Slay...	Haruo Sotoz...	2025	Animated Da...

Voy a mover Titanic al inicio

Movie Form

Title:

Titanic

Director:

James Cameron

Year:

1997

Genre:

Romantic

Synopsis:

The world's largest ship to date faces a hidden danger in the ocean after colliding with the iceberg that caused its inevitable sinking.

Insert Position:

Insert at End

Save

New

Delete

Move to Beginning

Move to End

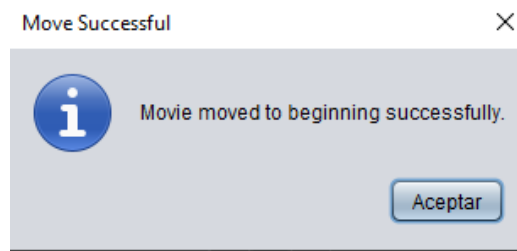
Movie Catalog

Search by:

Title

Title	Director	Year	Genre
Demon Slay...	Haruo Sotoz...	2025	Animated Da...
Inception	Christopher ...	2010	Science Fict...
Interstellar	Christopher ...	2014	Science Fict...
Titanic	James Cam...	1997	Romantic

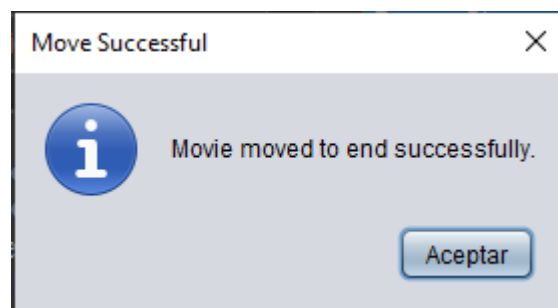
Figura 3.21 Movimiento de dato al inicio



**Y así mismo se puede mover al final**

Title ▲	Director	Year	Genre
Demon Slay...	Haruo Sotoz...	2025	Animated Da...
Inception	Christopher ...	2010	Science Fict...
Interstellar	Christopher ...	2014	Science Fict...
Titanic	James Cam...	1997	Romantic

*Figura 3.22 Movimiento de dato al final*



Title ▼	Director	Year	Genre
Titanic	James Cam...	1997	Romantic
Interstellar	Christopher ...	2014	Science Fict...
Inception	Christopher ...	2010	Science Fict...
Demon Slay...	Haruo Sotoz...	2025	Animated Da...

## Búsqueda por tres diferentes campos

### (Title, Genre, Year)

La búsqueda se realiza de forma que se va comparando las letras que ubica, para ver si están en el título de la película, aunque no lo escriba por completo.

**Movie Catalog**

Search by: Title in

Title	Director	Year	Genre
Interstellar	Christopher ...	2014	Science Fict...
Inception	Christopher ...	2010	Science Fict...
Demon Slay...	Haruo Sotoz...	2025	Animated Da...

**Movie Catalog**

Search by: Title ince

Title	Director	Year	Genre
Inception	Christopher ...	2010	Science Fict...

### Genre

**Movie Catalog**

Search by: Genre in

Title	Director	Year	Genre
-------	----------	------	-------

**Movie Catalog**

Search by: Genre science

Title	Director	Year	Genre
Interstellar	Christopher ...	2014	Science Fict...
Inception	Christopher ...	2010	Science Fict...

**Movie Catalog**

Search by: Genre roman

Title	Director	Year	Genre
Titanic	James Cam...	1997	Romantic

**Movie Catalog**

Search by: Genre animated

Title	Director	Year	Genre
Demon Slay...	Haruo Sotoz...	2025	Animated Da...

Year

**Movie Catalog**

Search by: Year ads

Title	Director	Year	Genre
-------	----------	------	-------



**Movie Catalog**

Search by: Year 10

Title	Director	Year	Genre
Inception	Christopher ...	2010	Science Fict...

**Movie Catalog**

Search by: Year 20

Title	Director	Year	Genre
Interstellar	Christopher ...	2014	Science Fict...
Inception	Christopher ...	2010	Science Fict...
Demon Slay...	Haruo Sotoz...	2025	Animated Da...

**Movie Catalog**

Search by: Year 19

Title	Director	Year	Genre
Titanic	James Cam...	1997	Romantic

**Movie Catalog**

Search by: Year 2014

Title	Director	Year	Genre
Interstellar	Christopher ...	2014	Science Fict...

## Eliminar

Se elimina seleccionando la fila de la película

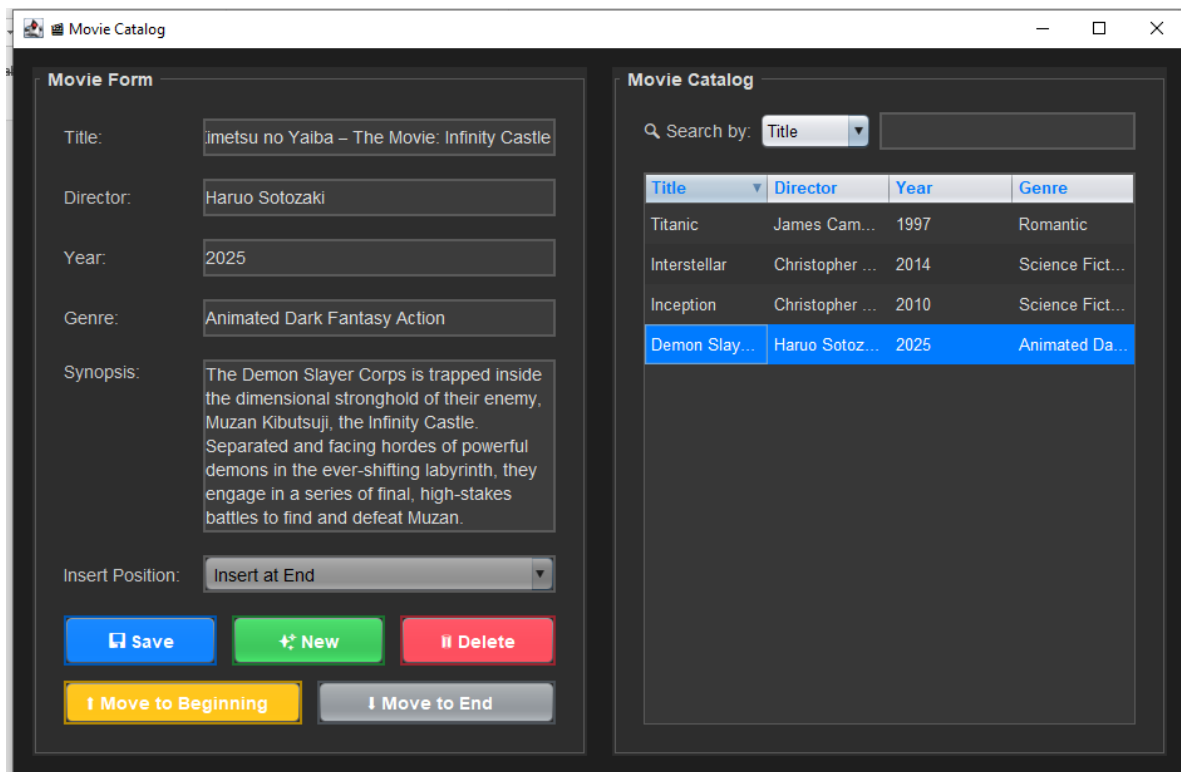
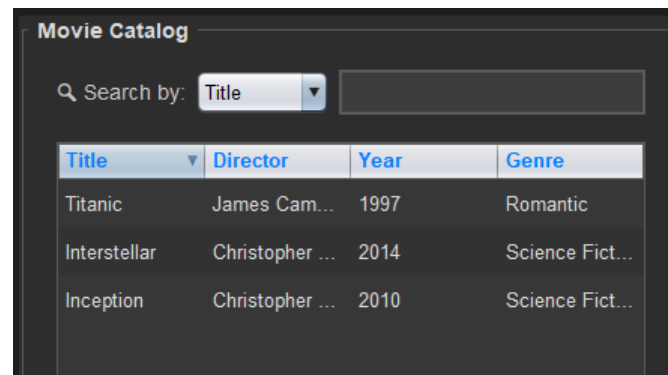
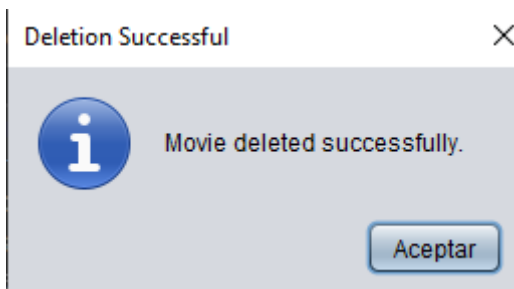
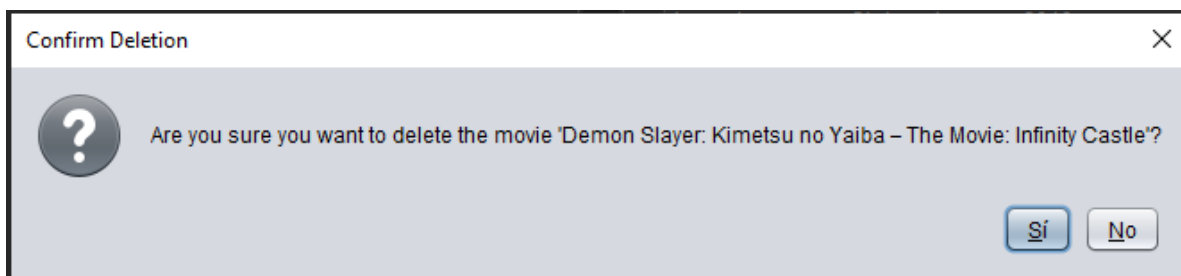


Figura 3.23 Elimina un dato de la tabla



## Validaciones

### Ingresos Incorrectos (Alertas)

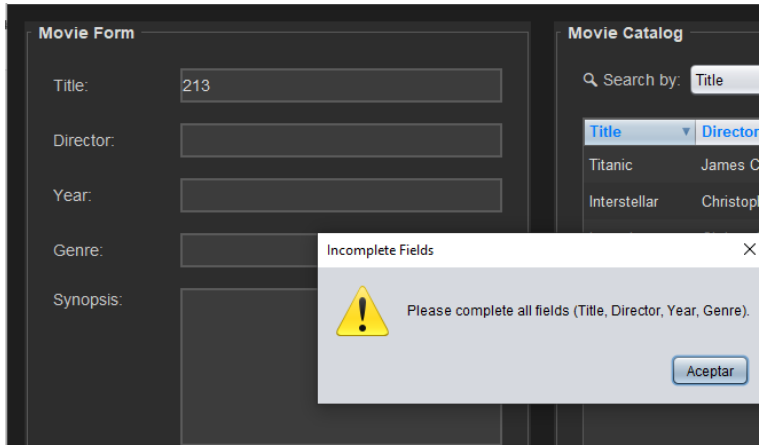
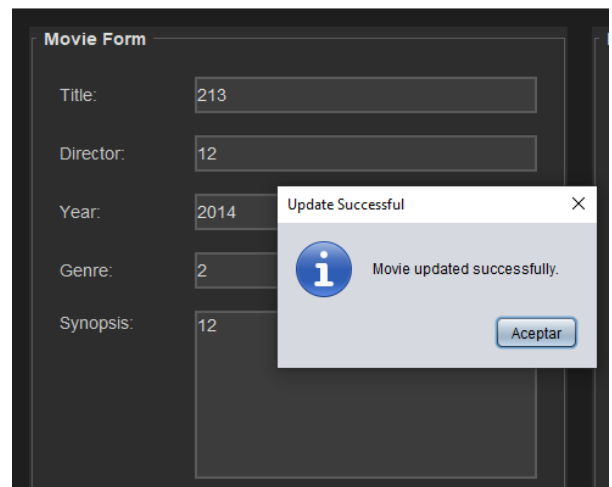
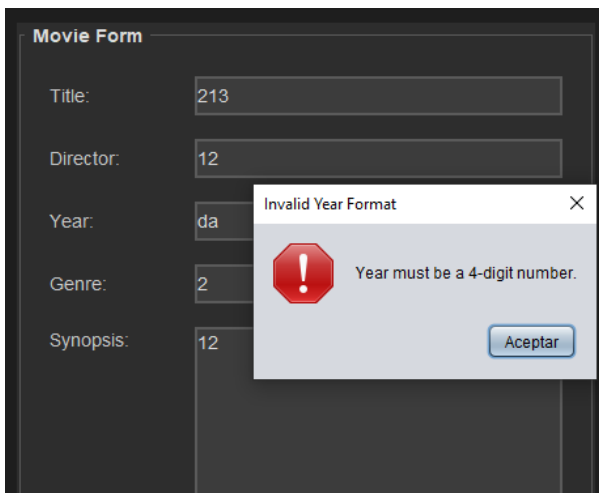
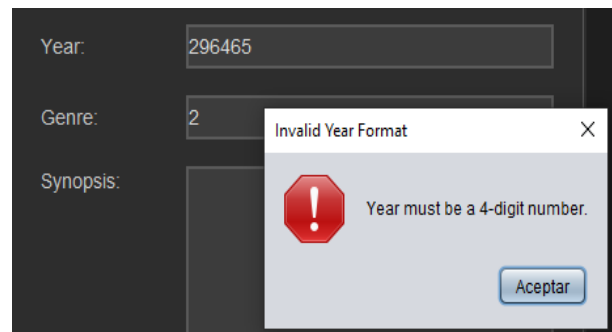
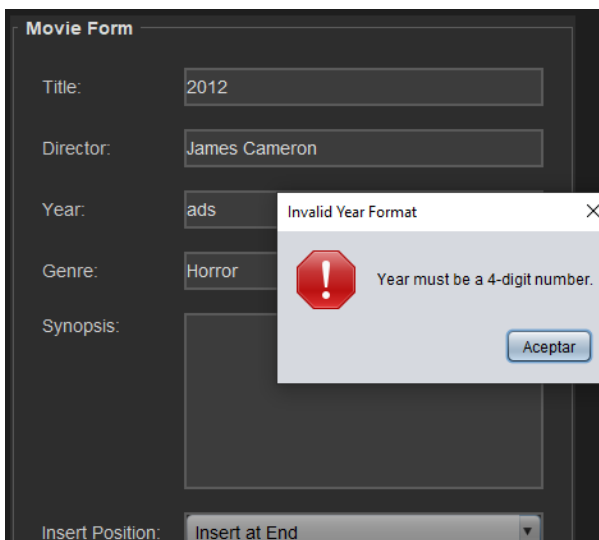


Figura 3.24 Validación de datos y ventanas emergentes



### Año Incorrecto



## Límite de Fecha

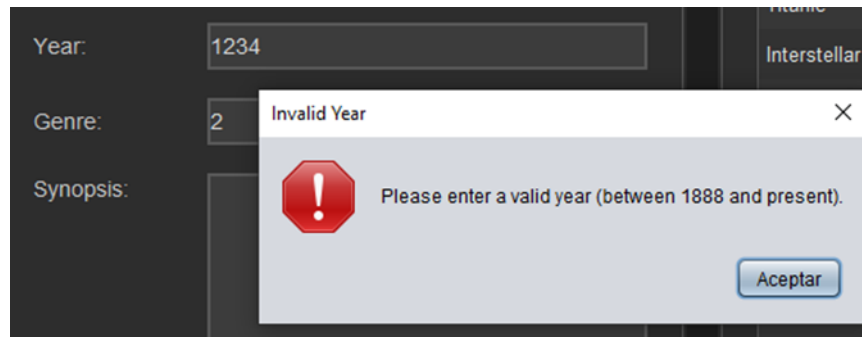
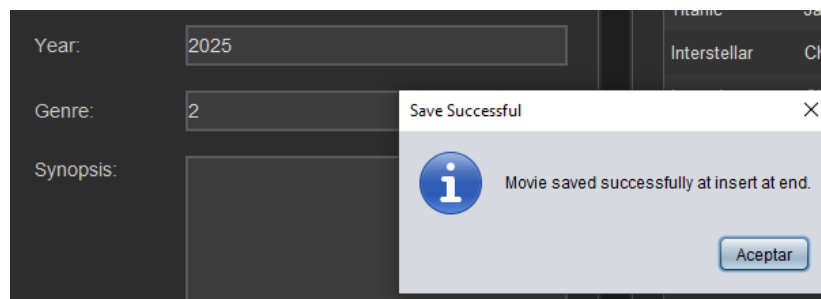


Figura 3.25 Validacion fecha 1888-present

## Año Correcto



## 4. Conclusiones

- La implementación de la aplicación de gestión de películas fue un éxito rotundo, validando la aplicación práctica de la Lista Enlazada Simple como una estructura de datos flexible y dinámica.
- Se cumplieron los objetivos específicos al implementar correctamente las clases del Modelo (Movie, Node, LinkedList) y todas las operaciones fundamentales de la lista enlazada (inserción al inicio/final, eliminación, búsqueda y recorrido), demostrando un conocimiento profundo de la gestión de punteros y referencias en Java.
- La adopción del patrón MVC facilitó la división de responsabilidades. La lógica de la estructura de datos permaneció independiente de la interfaz gráfica, lo cual mejoró la

legibilidad, mantenibilidad y escalabilidad del código, permitiendo futuras modificaciones a la interfaz sin afectar el Modelo.

- Se demostró que la Lista Enlazada Simple es ideal para escenarios donde las inserciones y eliminaciones son frecuentes y se realizan en los extremos o mediante un acceso secuencial, ya que estas operaciones no requieren el reordenamiento de grandes bloques de memoria, a diferencia de un array.
- La implementación de funcionalidades de reorganización (mover al inicio/final) demuestra la flexibilidad de las listas enlazadas para modificar el orden de los elementos de forma eficiente, requiriendo únicamente la manipulación de referencias en lugar del desplazamiento físico de datos en memoria. Esta característica es particularmente valiosa en aplicaciones donde el orden de los elementos es dinámico y puede cambiar frecuentemente según las necesidades del usuario.

## 5. Recomendaciones

Para futuras iteraciones y mejoras del sistema, se plantean las siguientes recomendaciones:

**Implementación de Listas Doblemente Enlazadas:** Migrar la estructura de datos a una Lista Doblemente Enlazada. Esto permitiría la navegación bidireccional, simplificando la eliminación de un nodo arbitrario (ya que no se necesitaría mantener una referencia al nodo anterior) y ofreciendo mayor flexibilidad en las operaciones de recorrido.

**Abstracción con Interfaces:** Utilizar una interfaz `ListADT` (Abstract Data Type) que sea implementada por la clase `List`. Esto aumentaría la modularidad y permitiría cambiar

fácilmente la implementación de la estructura de datos (por ejemplo, a una lista circular) sin modificar el Controlador.

**Funcionalidad de Ordenamiento:** Implementar un método para ordenar los elementos de la lista, por ejemplo, por título o año de estreno, lo cual requeriría la adición de lógica de ordenamiento (como *Bubble Sort* o *Merge Sort* adaptado a listas enlazadas) en la capa del Modelo.

**Persistencia de Datos:** Introducir una capa de persistencia (utilizando archivos de texto, CSV, o una base de datos simple como SQLite) para que los datos ingresados se mantengan disponibles después de cerrar y reabrir la aplicación.

## 6. Bibliografía/ Referencias

UPIICSA. (s.f.). *Estructuras de Datos con Java*. Recuperado de [http://www.sites.upiicsa.ipn.mx/estudiantes/academia\\_de\\_informatica/estructura\\_y\\_rd/docs/u2/RECURSOS/notasEstructuras.pdf](http://www.sites.upiicsa.ipn.mx/estudiantes/academia_de_informatica/estructura_y_rd/docs/u2/RECURSOS/notasEstructuras.pdf).

Universidad Nacional de San Agustín. (2020). *LABORATORIO 04: LINKED LIST*. (Documento EDA\_Lab 04\_2020). Recuperado de <https://es.scribd.com/document/472656129/EDA-Lab-04-2020>.

FCEIA. (s.f.). *Estructura de Datos : Lista Enlazada Simple*. Recuperado de <https://www.fceia.unr.edu.ar/estruc/2006/listensi.htm>.

Molina, D. (s.f.). *LISTAS ENLAZADAS Y EJEMPLOS*. Recuperado de <https://dmmolina.wordpress.com/listas-enlazadas-y-ejemplos/>.

Universidad Michoacana de San Nicolás de Hidalgo. (s.f.). *Listas enlazadas*. (Capítulo 16 de Estructuras de datos en Java). Recuperado de [https://lc.fie.umich.mx/calderon/estructuras/libros/Estructuras\\_de\\_datos\\_en\\_Java/c16.pdf](https://lc.fie.umich.mx/calderon/estructuras/libros/Estructuras_de_datos_en_Java/c16.pdf).