El patrón de diseño Modelo-Vista-Controlador (MVC) y su implementación en Java Swing.

Stefanny Nikoll Hidalgo Urrea

Tecnólogo en Análisis y desarrollo de Software

SENA - Servicio Nacional de Aprendizaje

Neiva – Huila

Noviembre, 2024

Palabras claves: MVC (Modelo-Vista-Controlador), java, programación, orientada a objetos, patrones, diseño.

TABLA DE CONTENIDO.

INTRODUCCIÓN.

En el proceso de enseñanza de la programación, uno de los principales desafíos que enfrentan los docentes es ayudar a los estudiantes a superar un hábito común: el de escribir código extenso, desordenado y difícil de entender. Es frecuente que los estudiantes intenten resolver problemas mediante grandes bloques de código que mezclan diversas funcionalidades, como algoritmos de procesamiento de datos, manejo de colores, mensajes al usuario, gráficos, acceso a bases de datos, y más. Este tipo de código, aunque puede funcionar, se vuelve difícil de leer, comprender, mantener y modificar con el tiempo, lo que plantea un problema significativo a medida que los programas crecen en complejidad.

El *Algoritmo 1* ilustrado en el artículo es un ejemplo claro de esta mala práctica, donde se combinan en un único bloque de código tareas muy distintas, como la entrada de datos, el procesamiento, la visualización de resultados, la gestión de errores y la presentación de gráficos. Aunque puede parecer un caso extremo, este enfoque desorganizado no es raro, especialmente en muchas aplicaciones hechas a medida y en aplicaciones web desarrolladas localmente. A menudo, las herramientas de desarrollo rápido de aplicaciones (RAD), como Delphi, Visual Basic, y los lenguajes de scripting para desarrollo web (como PHP, ASP, o JSP), contribuyen a la creación de aplicaciones con este tipo de problemas. Estas herramientas permiten a los programadores implementar rápidamente funciones complejas, pero a menudo sin tener en cuenta las buenas prácticas de organización del código, lo que resulta en aplicaciones fáciles de desarrollar, pero difíciles de comprender, modificar o mantener a largo plazo.

Este artículo propone un enfoque diferente para resolver estos problemas, a través de un patrón de diseño que sugiere una separación clara entre los módulos de entrada, procesamiento y salida de datos dentro de una aplicación. La idea es organizar el código de forma que cada módulo tenga una responsabilidad específica, lo que facilita su comprensión, mantenimiento y extensión. Además, se discutirá cómo implementar este patrón de diseño en la librería Java Swing, una de las más utilizadas para la creación de interfaces gráficas en aplicaciones Java. El artículo también abordará las ventajas y desventajas de aplicar este patrón, proporcionando una perspectiva crítica sobre cómo puede mejorar la estructura del código y la calidad del software a largo plazo.

RESUMEN.

Durante la década de 1970, lenguajes como SmallTalk y Simula I, fueron fundamentales para el desarrollo del paradigma de programación orientada a objetos, introduciendo conceptos clave como objetos, clases, encapsulación, herencia y polimorfismo. Aunque hoy en día estos lenguajes ya no se usan para aplicaciones comerciales, los principios que establecieron siguen siendo la base de lenguajes modernos como C++, Java y C#. Además, SmallTalk innovó al ser el primer lenguaje en permitir el diseño de interfaces gráficas con múltiples "ventanas" en una misma pantalla, un concepto que posteriormente fue adoptado por sistemas como GEMS, Macintosh, X11 y Windows.

En cuanto al diseño de interfaces de usuario, SmallTalk implementó el patrón de diseño **MVC (Modelo-Vista-Controlador)**, propuesto por el profesor Trygve Reenskaug. Este patrón organiza una aplicación dividiéndola en tres componentes claramente definidos: el **Modelo**, las **Vistas** y el **Controlador**.

El **Modelo** es un conjunto de clases que representan los datos o la información del mundo real que la aplicación debe procesar. Este modelo no se preocupa por cómo se muestran los datos ni por los mecanismos que gestionan esos datos. En la práctica, el modelo se divide en dos subcomponentes: el **modelo del dominio**, que refleja los conceptos centrales del problema a resolver (como cliente, factura, temperatura, etc.), y el **modelo de la aplicación**, que coordina la interacción con las vistas y se encarga de notificar cualquier cambio en el modelo del dominio.

Las **Vistas** son las clases responsables de mostrar la información contenida en el modelo al usuario. Cada vista está asociada a un modelo, y pueden existir múltiples vistas para un mismo modelo, mostrando la misma información de diferentes maneras (por ejemplo, un reloj analógico y uno digital mostrando la hora del sistema). Las vistas se actualizan automáticamente cada vez que el modelo cambia, gracias a las notificaciones generadas por el modelo de la aplicación.

El **Controlador** actúa como el intermediario que gestiona la interacción del usuario con la aplicación, respondiendo a eventos como datos introducidos o opciones seleccionadas desde el menú. El controlador tiene acceso tanto al modelo como a las vistas y es responsable de modificar el modelo o abrir y cerrar vistas según sea necesario. Sin embargo, el controlador, el modelo y las vistas están desacoplados, lo que significa que el modelo y las vistas no conocen la existencia del controlador.

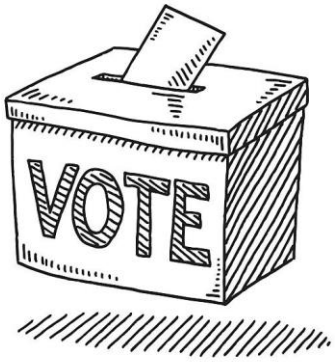
Este enfoque modular facilita una mayor organización y separación de responsabilidades dentro del código, lo que a su vez mejora la mantenibilidad y escalabilidad de las aplicaciones.

EJEMPLO.

Imaginemos una aplicación diseñada para almacenar y procesar los datos de las elecciones municipales. En este caso, el **modelo del dominio** sería simple: consistiría en tres conjuntos principales de datos: los **votos**, las **mesas de votación** y los **departamentos**. Cada voto almacenaría la selección hecha por el votante y la mesa donde se emitió el voto. Cada mesa contendría información sobre el lugar de votación y el departamento al que pertenece.

Las **vistas** del modelo también serían variadas pero simples. Por ejemplo, se podrían mostrar gráficos estadísticos de los votos por departamento en forma de barras o de torta, una tabla con el total de votos, o los votos totales representados en barras o torta. Aunque las vistas presenten la misma información de manera diferente, todas estarían basadas en el mismo modelo de datos.

El **controlador** sería responsable de gestionar la interacción del usuario. Cuando el usuario quiera ver una vista específica, el controlador crearía esa vista, la cual obtendría la información necesaria del modelo y la mostraría. Si el usuario añadiera nuevos votos, el controlador actualizaría el modelo del dominio. Luego, al detectar que hubo cambios, el modelo de la aplicación notificaría a todas las vistas asociadas para que se actualicen automáticamente, garantizando que las vistas siempre muestren la información más actualizada del modelo.



VENTAJAS

Desarrollar una aplicación con el patrón de diseño **MVC** ofrece varias ventajas clave:

- La aplicación es **modular**, lo que facilita su mantenimiento y comprensión.
- Las **vistas** siempre muestran información actualizada automáticamente, sin que el programador tenga que gestionar este proceso.
- Las **modificaciones en el modelo** solo requieren cambios en el modelo y sus interfaces con las vistas, sin afectar el resto del sistema.
- Cambiar las **vistas** no impacta a los otros módulos de la aplicación.
- MVC es ampliamente utilizado en marcos como **Java Swing, Apache Struts y ASP.NET**, entre otros.
- Las aplicaciones que usan MVC son más **extensibles** y **mantenibles** que aquellas basadas en otros patrones.

En resumen, MVC facilita la gestión y evolución de aplicaciones grandes, mejorando su organización y capacidad de mantenimiento.

DESVENTAJAS

Implementar el patrón de diseño **MVC** puede aumentar el **tiempo de desarrollo** inicial, ya que requiere crear más clases y estructuras que un enfoque tradicional. Esto hace que la fase de desarrollo sea más lenta, especialmente al principio. Sin embargo, esta desventaja se ve compensada en el **mantenimiento** de la aplicación, ya que las aplicaciones basadas en MVC son mucho más **mantenibles**, **extensibles** y **modificables** a largo plazo.

MVC también exige una **arquitectura inicial** más compleja, que debe incluir clases e interfaces para gestionar la comunicación entre los módulos de la aplicación. Esta arquitectura básica debe contar con un mecanismo de eventos para las notificaciones del modelo, y al menos tres clases fundamentales: **Modelo**, **Vista** y **Controlador**, que se encargan de la comunicación y actualización automática.

Finalmente, dado que **MVC** es un patrón orientado a objetos, su implementación resulta más **costosa y difícil** en lenguajes que no siguen este paradigma, lo que limita su uso en algunos entornos.

MVC EN JAVA SWING.

Java es un lenguaje de programación orientado a objetos creado por Sun Microsystems. Sus características principales incluyen la portabilidad, ya que es compatible con múltiples plataformas, la simplicidad y un amplio conjunto de bibliotecas. Java no solo es un lenguaje, sino también una plataforma en sí misma.

Dentro del kit de desarrollo de Java (SDK) se encuentra **Java Swing**, una librería para crear interfaces gráficas de usuario (GUI), que destaca por:

- Ofrecer numerosos componentes visuales como botones, campos de texto, tablas, menús y árboles.
- Ser independiente de la plataforma, lo que garantiza que las aplicaciones desarrolladas con Swing funcionen de manera consistente en diferentes sistemas operativos. A diferencia de **AWT**, que usa componentes nativos del sistema operativo, Swing utiliza componentes propios, lo que mejora la portabilidad.
- Permitir cambiar los estilos visuales de la interfaz (conocidos como "look and feels") en tiempo de ejecución, lo que permite que la misma aplicación se vea como si fuera nativa de Windows, Motif u otros entornos, según se desee.
- Cumplir con el estándar **JavaBeans**, lo que facilita el uso de sus componentes en entornos de desarrollo integrados (IDEs) como JBuilder, Sun Forte for Java o Eclipse.
- Estar basado en el patrón de diseño **Modelo-Vista-Controlador (MVC)**, lo que proporciona un alto grado de extensibilidad y personalización de sus componentes.

En resumen, Java y su librería Swing ofrecen una plataforma robusta y flexible para el desarrollo de aplicaciones gráficas multiplataforma.

CONCLUSIONES.

El artículo analiza los desafíos comunes que enfrentan los docentes al enseñar programación, específicamente el hábito de los estudiantes de escribir código desorganizado y difícil de mantener. Este enfoque no solo complica la comprensión del código, sino que también dificulta su mantenimiento y evolución, especialmente en aplicaciones más complejas. Para abordar este problema, se propone el uso del patrón de diseño Modelo-Vista-Controlador (MVC), que promueve la separación de responsabilidades al dividir una aplicación en tres componentes principales: Modelo, Vista y Controlador.

El uso de MVC facilita el desarrollo de aplicaciones más modulares y organizadas, lo que mejora la mantenibilidad y escalabilidad del software. Java Swing, una librería gráfica para Java, es presentada como un ejemplo práctico de implementación de este patrón, destacando su capacidad para crear interfaces gráficas multiplataforma. Swing no solo es independiente de la plataforma, sino que también permite cambiar estilos visuales ("look and feels") y se integra bien en entornos de desarrollo como Eclipse y JBuilder.

El enfoque modular de MVC ofrece múltiples ventajas, como la actualización automática de vistas cuando los datos cambian y la posibilidad de modificar componentes sin afectar al resto de la aplicación. Sin embargo, también presenta algunas desventajas, como un mayor tiempo de desarrollo inicial y una arquitectura más compleja que puede resultar costosa en lenguajes que no son orientados a objetos.

En conclusión, aunque la implementación del patrón MVC puede requerir un esfuerzo adicional al principio, sus beneficios en términos de organización, mantenibilidad y extensibilidad hacen que sea una elección valiosa para el desarrollo de aplicaciones robustas a largo plazo, especialmente en entornos donde la escalabilidad y el mantenimiento son críticos.

BIBLIOGRAFÍA.

http://www.scielo.org.bo/scielo.php?script=sci_arttext&pid=S1683-07892004000100005

Java Swing architecture. http://www.cs.unc.edu/Courses/wwwp-s99/members/walkera/swing/presentSwingII.htm. [Links]

Joseph Bergin. Building Graphical User Interfaces with the MVC pattern. http://csis.pace.edu/bergin/mvc/mvcgui.html. [Links]

Steve Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html. [Links]

John Deacon. Model-View-Controller (MVC) Architecture. http://www.jdl.co.uk/briefings/mvc.pdf. [Links]

Amy Fowler. A Swing architecture overview. http://java.sun.com/products/jfc/tsc/articles/architecture/. [Links]

Sun Microsystems. JavaTM 2 Platform, Standard Edition, v 1.4.2, API Specification. http://java.sun.com/products/jdk1.4/doc.