Received 20 Febrer, 2025; revised ; Date of publication 12 Març, 2025; date of current version 12 Març, 2025.

Implementació del Model Vista Controlador en una Aplicació Pràctica

Josep Ferriol, Daniel García, Khaoula Ikkene, Biel Perelló

[1]Universitat de les Illes Balears, Departament d'Enginyeria Informàtica

Autor de contacte: Daniel García (email: daniel.garcia19@estudiant.uib.es)

ABSTRACT Aquest document presenta la implementació del Model Vista Controlador (MVC) en una aplicació pràctica. Es detallen els conceptes teòrics de l'arquitectura MVC, el patró per esdeveniments i el càlcul del cost asimptòtic de la instrucció crítica. A més, es descriu l'entorn de programació utilitzat, incloent les llibreries de Swing per a la interfície gràfica. La pràctica consisteix en desenvolupar una aplicació que implementi els algorismes de suma i producte de matrius. Es presenten gràfiques dels costos computacionals assimptòtics d'aquests algorismes, així com una anàlisi del temps d'execució i la constant multiplicativa. L'aplicació permet iniciar i arrancar els processos de suma i producte de matrius i mostra els resultats en una interfície gràfica desenvolupada amb Java Swing.
Finalment, s'analitzen els resultats obtinguts i es presenten les conclusions del projecte.

INDEX TERMS Arquitectura de Programari, Cost Asimptòtic, Java Swing, Model-Vista-Controlador, Multiplicació de Matrius, Programació Basada en Esdeveniments, Suma de Matrius

I. Introducció

El Model Vista Controlador (MVC) és un patró d'arquitectura de programari àmpliament utilitzat per separar la lògica de presentació, la lògica de negoci i la gestió de dades. Aquesta separació permet millorar la modularitat i mantenibilitat dels sistemes, facilitant així el desenvolupament i manteniment de les aplicacions.

Aquest projecte té com a objectiu implementar el patró MVC en una aplicació pràctica que realitzi operacions de suma i producte de matrius. A més, es pretén analitzar els beneficis d'aquest patró en termes de modularitat i eficiència.

En aquesta memòria, es presenten els conceptes teòrics de l'arquitectura MVC, incloent la seva estructura i els avantatges que ofereix. També s'explica el patró per esdeveniments, que permet una comunicació eficient entre els components del sistema mitjançant la gestió d'esdeveniments i notificacions.

A continuació, es descriu l'entorn de programació utilitzat per al desenvolupament de l'aplicació, incloent les llibreries de Swing per a la interfície gràfica i les eines emprades per a la gestió del codi i el control de versions.

La part central de la pràctica consisteix en desenvolupar una aplicació que implementi els algorismes de suma i producte de matrius, amb complexitats $O(n^2)$ i $O(n^3)$ respectivament. Es presenten gràfiques dels costos computacionals asimptòtics d'aquests algorismes, així com una anàlisi detallada del temps d'execució i la constant multiplicativa. L'aplicació permet iniciar i arrancar els processos de suma i producte de matrius, junts o per separat, i mostra els resultats en una interfície gràfica desenvolupada amb Java Swing.

Finalment, s'analitzen els resultats obtinguts, destacant els beneficis de l'ús del patró MVC en termes de modularitat i mantenibilitat del codi. També es presenten les conclusions del projecte i es proposen possibles millores futures per optimitzar el sistema.

II. Conceptes Teòrics

A. Model Vista Controlador (MVC)

El Model Vista Controlador (MVC) és un patró d'arquitectura de programari que separa l'aplicació en tres components principals: el Model, la Vista i el Controlador. Aquesta separació permet una millor modularitat i mantenibilitat del codi.

- **Model**: Gestiona les dades i la lògica de negoci de l'aplicació. És responsable de l'accés a la base de dades, càlculs i qualsevol altra lògica de negoci necessària. La lògica de negoci, també coneguda com a lògica de l'aplicació, és un conjunt de regles que es segueixen en el programari per reaccionar davant diferents situacions. A més de marcar un comportament quan ocorren coses dins del programari, també té normes sobre el que es pot fer i el que no es pot fer, conegudes com a regles del negoci.
- **Vista**: S'encarrega de la presentació de les dades a l'usuari. És responsable de la interfície gràfica i de mostrar la informació de manera comprensible. La

Vista actua com el frontend o interfície gràfica d'usuari (GUI).

- **Controlador**: Actua com a intermediari entre el Model i la Vista. Gestiona les interaccions de l'usuari, actualitza el Model i refresca la Vista. El Controlador és el cervell de l'aplicació que controla com es mostren les dades.

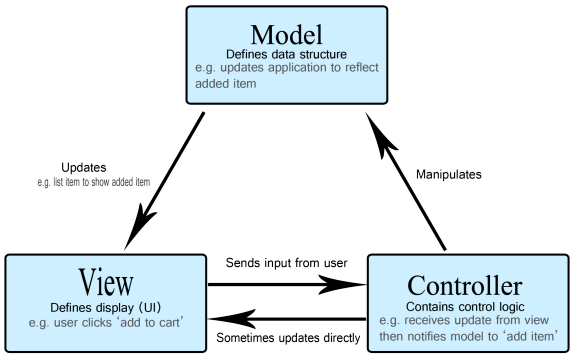

FIGURE 1. **Esquema del Model-Vista-Controlador.**

El concepte de MVC va ser introduït per primera vegada per Trygve Reenskaug , qui el va proposar com una forma de desenvolupar el GUI d'aplicacions d'escriptori. Avui en dia, el patró MVC s'utilitza per a aplicacions web modernes perquè permet que l'aplicació sigui escalable, mantenible i fàcil d'expandir.

Les principals utilitats del patró MVC inclouen:

- **Modularitat**: Facilita la separació de responsabilitats, permetent que cada component es desenvolupi i es mantingui de manera independent. Això respecta el principi de la responsabilitat única, on una part del codi no ha de saber què fa tota l'aplicació, només ha de tenir una responsabilitat específica.
- **Reutilització de codi**: Els components poden ser reutilitzats en diferents parts de l'aplicació o en altres projectes.
- **Facilitat de manteniment**: La separació de responsabilitats facilita la detecció i correcció d'errors, així com la implementació de noves funcionalitats.
- **Separació de preocupacions (SoC)**: El patró MVC ajuda a dividir el codi frontend i backend en components separats, fent que sigui molt més fàcil gestionar i fer canvis a qualsevol dels costats sense que interfereixin entre si.

El patró MVC és especialment útil quan diversos desenvolupadors necessiten actualitzar, modificar o depurar una aplicació completada simultàniament. Aquesta separació de preocupacions permet que els desenvolupadors treballin en diferents parts de l'aplicació sense afectar altres parts del codi.

En resum, el patró MVC és una eina poderosa per a la construcció d'aplicacions escalables i mantenibles, permetent una clara separació de responsabilitats i facilitant la gestió del codi.

B. Patró per Esdeveniments

El patró per esdeveniments és un mecanisme de comunicació entre components que permet gestionar esdeveniments i notificacions de manera eficient. En aquest patró, els components poden generar esdeveniments i subscriure's a esdeveniments generats per altres components.

- **Generació d'esdeveniments**: Els components poden generar esdeveniments quan es produeixen canvis en el seu estat o quan es completen determinades accions.
- **Subscripció a esdeveniments**: Els components poden subscriure's a esdeveniments generats per altres components per rebre notificacions i actuar en conseqüència.
- **Desacoblament**: El patró per esdeveniments permet desacoblar els components, ja que no necessiten conèixer els detalls dels altres components amb els quals interactuen.

C. Càlcul del Cost Asimptòtic

El càlcul del cost asimptòtic és una tècnica utilitzada per analitzar l'eficiència dels algorismes en termes de temps d'execució i ús de memòria. Es basa en la notació Big-O, que descriu el comportament asimptòtic d'un algorisme quan la mida de les dades d'entrada tendeix a l'infinit.

La notació Big-O és una forma de mesurar com escala un programa o un algorisme i el temps que tardarà a executar-se. Aquesta mesura és útil per comparar l'eficiència de dos algorismes, per exemple, d'ordenació. Big-O no proporciona una mesura exacta del temps d'execució, sinó una avaluació de com de ràpid s'incrementa el temps d'execució en funció de les dades d'entrada. En termes matemàtics, descriu el límit d'una funció quan els arguments d'entrada tendeixen a l'infinit o a un valor en particular.

- **Notació Big-O**: Descriu el pitjor cas del temps d'execució d'un algorisme en funció de la mida de les dades d'entrada. Per exemple, $O(n^2)$ indica que el temps d'execució creix quadràticament amb la mida de les dades.
- **Cost asimptòtic de la instrucció crítica**: En aquest projecte, s'analitza el cost asimptòtic de les operacions de suma i producte de matrius, amb complexitats $O(n^2)$ i $O(n^3)$ respectivament.

Els diferents tipus de temps de Big-O inclouen:

- **Temps Constant O(1)**: El temps d'execució es manté constant independentment de la mida del conjunt de dades d'entrada.
- **Temps Logarítmic O(log n)**: El temps d'execució depèn del logaritme de la mida del conjunt de dades d'entrada. Per exemple, els algorismes que divideixen el

conjunt de dades d'entrada en meitats segueixen aquesta notació.

- **Temps Lineal O(n)**: El temps d'execució depèn directament de la mida del conjunt de dades d'entrada. Com més elements s'afegeixin al processament, el temps d'execució s'incrementarà proporcionalment.
- **Temps Quadràtic O(n^2)**: El creixement del temps d'execució és exponencial, generalment observat en funcions amb bucles anidats.
- **Temps Exponencial O(2^n)**: El temps d'execució es duplica amb cada element que s'afegeix al conjunt de dades d'entrada.
- **Temps Factorial O(n!)**: El creixement del temps d'execució és pràcticament vertical, incrementant considerablement amb cada element afegit al conjunt de dades d'entrada.

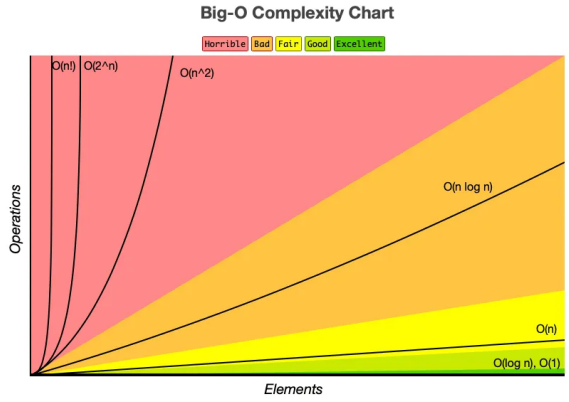

FIGURE 2. **Exemple de diferents complexitats de la notació Big-O.**

La notació Big-O és important perquè permet als desenvolupadors i científics de dades avaluar i comparar l'eficiència de diferents algorismes. Això ajuda a triar l'algorisme més adequat per a una tasca específica, optimitzant el rendiment d'aplicacions i sistemes.

Per calcular la notació Big-O d'un algorisme, s'analitzen els bucles, bucles anidats, recursions i operacions dominants del codi, que són les que més afecten el temps d'execució. A més, un factor important que afecta el rendiment i l'eficiència del programa és el maquinari, el sistema operatiu i la CPU que s'utilitza. No obstant això, això no es té en compte quan s'analitza el rendiment d'un algorisme. En canvi, el que importa és la complexitat temporal i espacial en funció de la mida de l'entrada.

En resum, la notació Big-O és una eina essencial per avaluar l'eficiència dels algorismes i triar els més adequats per a cada situació, optimitzant així el rendiment de les aplicacions.

D. Conceptes Bàsics

En aquest apartat es presenten alguns conceptes fonamentals que són rellevants per a la implementació del patró MVC i l'anàlisi dels algorismes utilitzats en la pràctica.

El llenguatge de programació és un factor important que pot influir en l'eficiència i la facilitat de desenvolupament d'una aplicació. En aquest projecte, s'ha utilitzat Java, un llenguatge orientat a objectes que facilita la modularitat i la reutilització del codi.

La Programació Orientada a Objectes (POO) és una metodologia de programació que organitza el codi en classes i objectes, permetent una millor estructuració i mantenibilitat del sistema. Les interfícies, per la seva banda, defineixen un contracte que les classes han d'implementar, facilitant la interoperabilitat entre diferents components del sistema.

La separació de la visió, el control i les dades és un principi fonamental del patró MVC. Aquesta separació permet que cada component es desenvolupi i es mantingui de manera independent, millorant la modularitat i la mantenibilitat del sistema.

L'eficiència dels algorismes és un aspecte crucial per al rendiment del sistema. En aquest projecte, s'han implementat algorismes de suma i producte de matrius amb complexitats O(n^2) i O(n^3) respectivament. La notació Big-O permet descriure el comportament asimptòtic d'aquests algorismes quan la mida de les dades d'entrada tendeix a l'infinit.

És important tenir en compte les mides asimptòtiques i les regles pràctiques per fer primeres aproximacions sobre l'eficiència dels algorismes. Tot i que alguns algorismes poden tenir una bondat teòrica, és important considerar les constants multiplicatives i la balança entre l'eficiència i el cost de producció, així com entre la memòria i el temps d'execució.

Aquests conceptes bàsics proporcionen una base sòlida per a la implementació del patró MVC i l'anàlisi dels algorismes utilitzats en aquest projecte.

III. Entorn de Programació

Per al desenvolupament d'aquest projecte s'ha emprat un conjunt d'eines i tecnologies que han facilitat la implementació del patró MVC i l'anàlisi dels algorismes de suma i producte de matrius. A continuació, es descriuen les principals eines utilitzades:

- **Llenguatge de programació: Java**
  Java és un llenguatge de programació orientat a objectes que ofereix una gran portabilitat i una àmplia gamma de llibreries i eines per al desenvolupament d'aplicacions. La seva sintaxi clara i la seva robustesa el fan ideal per a projectes que requereixen una alta mantenibilitat i escalabilitat. A més, Java és independent de la plataforma, la qual cosa permet executar l'aplicació en diferents sistemes operatius sense necessitat de modificar el codi.
- **Llibreries utilitzades: Swing**
  Swing és una llibreria de Java per a la creació d'interfícies gràfiques d'usuari (GUI). Proporciona un conjunt complet de components gràfics, com ara botons, camps de text, taules i panells, que permeten construir interfícies d'usuari riques i interactives. Swing és altament personalitzable i permet crear interfícies

gràfiques que s'adapten a les necessitats específiques de l'aplicació.

- **IDE: IntelliJ IDEA**
  IntelliJ IDEA és un entorn de desenvolupament integrat (IDE) per a Java que ofereix una àmplia gamma de funcionalitats per a facilitar el desenvolupament d'aplicacions. Algunes de les seves característiques més destacades inclouen la completació de codi intel·ligent, la navegació ràpida pel codi, les eines de refactorització i la integració amb sistemes de control de versions. Aquestes funcionalitats ajuden a millorar la productivitat i a mantenir un codi net i organitzat.
- **Control de versions: GitHub**
  GitHub és una plataforma de desenvolupament col·laboratiu que utilitza el sistema de control de versions Git. Permet gestionar el codi font del projecte, fer seguiment dels canvis, col·laborar amb altres membres de l'equip i mantenir un historial complet de les modificacions realitzades. GitHub també ofereix eines per a la gestió de projectes, com ara issues i pull requests, que faciliten la coordinació i la revisió del codi.

L'ús d'aquestes eines i tecnologies ha permès desenvolupar una aplicació robusta i mantenible, seguint els principis del patró MVC i assegurant una bona gestió del codi i la col·laboració entre els membres de l'equip.

IV. Desenvolupament

En aquesta secció es descriu el desenvolupament de la pràctica, que consisteix en la implementació d'una aplicació basada en el patró Model Vista Controlador (MVC) i el patró per esdeveniments. L'aplicació permet realitzar operacions de suma i producte de matrius, mostrant els costos computacionals asimptòtics d'aquests algorismes.

A. Arquitectura MVC

L'aplicació està estructurada seguint el patró MVC, que separa la lògica de negoci, la interfície d'usuari i el control de la interacció en tres components principals:

- **Model**: La classe `Dades` gestiona les dades de l'aplicació, incloent els resultats de les operacions de suma i producte de matrius. Aquesta classe també calcula les constants multiplicatives i emmagatzema els resultats en llistes. La classe `Matriu` representa una matriu i proporciona mètodes per sumar i multiplicar matrius.
- **Vista**: Les classes `FinestraMatriu`, `dibuixConstantMult` i `Eixos` s'encarreguen de la presentació de les dades a l'usuari. Utilitzen la llibreria Swing per crear la interfície gràfica i mostrar els resultats de les operacions en taules i gràfics.
- **Controlador**: La classe `Main` actua com a controlador, gestionant les interaccions de l'usuari i coordinant les operacions entre el model i la vista. Aquesta classe és responsable de crear i gestionar els fils d'execució per a les operacions de suma i producte de matrius, així com de mantenir la comunicació entre els diferents components de l'aplicació.

La classe `Main` conté els següents elements principals:

- **Inicialització**: El mètode `main` és el punt d'entrada de l'aplicació. Aquest mètode crea una instància de la classe `Main` i crida al mètode `inicio` per iniciar l'aplicació. El mètode `inicio` inicialitza les dades, els processos i l'`ExecutorService`, i després crida al mètode `crearInterficie` per crear la interfície gràfica en un fil separat.
- **Creació de la Interfície Gràfica**: El mètode `crearInterficie` crea la finestra principal de l'aplicació utilitzant la classe `FinestraMatriu`. També crea una segona finestra per mostrar els resultats de les operacions de suma i producte de matrius utilitzant la classe `dibuixConstantMult`. Aquestes finestres es mostren a l'usuari i permeten la interacció amb l'aplicació.
- **Gestió de Processos**: El mètode `preparar` s'encarrega de buidar les dades i aturar qualsevol procés en execució abans de començar una nova operació. Això assegura que les dades anteriors no interfereixin amb les noves operacions.
- **Comunicació**: La classe `Main` implementa la interfície `Comunicar`, que defineix el mètode `comunicar`. Aquest mètode s'utilitza per rebre missatges de la interfície gràfica i dels fils d'execució, i per enviar missatges als components corresponents. Per exemple, quan l'usuari fa clic en un botó per iniciar una operació, es crida al mètode `comunicar` amb el missatge corresponent, que després es processa per iniciar els fils d'execució adequats.
- **Gestió de Fils d'Execució**: La classe `Main` utilitza un `ExecutorService` per gestionar els fils d'execució de les operacions de suma i producte de matrius. Quan es rep un missatge per iniciar una operació, es creen instàncies de les classes `SumaM` i `MultM` i s'envien al `ExecutorService` per ser executades en fils separats. Això permet executar les operacions de manera concurrent i millorar l'eficiència de l'aplicació.
- **Actualització de la Interfície Gràfica**: Quan es completen les operacions de suma i producte de matrius, els resultats es mostren a la interfície gràfica. La classe `Main` s'encarrega d'enviar missatges a les classes `FinestraMatriu` i `dibuixConstantMult` per actualitzar les taules i els gràfics amb els nous resultats. Això permet a l'usuari visualitzar els costos computacionals asimptòtics de les operacions.

En resum, la classe `Main` actua com a controlador central de l'aplicació, gestionant les interaccions de l'usuari, coordinant les operacions entre el model i la vista, i assegurant que les operacions es realitzin de manera eficient i concurrent. Aquesta arquitectura permet una clara separació de responsabilitats, millorant la modularitat i mantenibilitat del codi.

B. Implementació del Patró per Esdeveniments

L'aplicació utilitza el patró per esdeveniments per gestionar la comunicació entre els components de manera eficient. Aquest patró permet que els components es comuniquin entre ells mitjançant l'enviament i la recepció de missatges, sense necessitat de conèixer els detalls interns dels altres components.

1) Interfície `Comunicar`

La interfície `Comunicar` defineix un mètode `comunicar` que permet enviar missatges entre els components. Aquesta interfície és implementada per diverses classes de l'aplicació, permetent una comunicació uniforme i consistent. La definició de la interfície és la següent:

```
public interface Comunicar {
    public void comunicar(String s);
}
```

2) Classe `Main`

La classe `Main` actua com a controlador central de l'aplicació i és responsable de gestionar la comunicació entre els components, com s'ha explicat anteriorment. Quan l'usuari interactua amb la interfície gràfica, la classe `Main` rep els missatges corresponents i els processa per iniciar, aturar o actualitzar les operacions de suma i producte de matrius. El mètode `comunicar` de la classe `Main` processa els missatges i crida als mètodes adequats per gestionar les operacions.

3) Classes `SumaM` i `MultM`

Les classes `SumaM` i `MultM` implementen la interfície `Comunicar` per rebre missatges del controlador i aturar els fils d'execució quan sigui necessari. Quan es rep un missatge per aturar l'operació, el mètode `comunicar` d'aquestes classes estableix una variable booleana `stop` a `true`, la qual cosa fa que el bucle principal del mètode `run` s'aturi.

4) Classe `FinestraMatriu`

La classe `FinestraMatriu` també implementa la interfície `Comunicar` per rebre missatges del controlador i actualitzar la interfície gràfica en conseqüència. Per exemple, quan es rep un missatge per iniciar una operació, la classe `FinestraMatriu` activa les barres de progrés corresponents per indicar que l'operació està en marxa. Quan es rep un missatge per aturar l'operació, les barres de progrés es desactiven.

5) Flux de Comunicació

El flux de comunicació entre els components es pot resumir en els següents passos:

1) L'usuari interactua amb la interfície gràfica, per exemple, fent clic en un botó per iniciar una operació.
2) La classe `FinestraMatriu` envia un missatge al controlador (classe `Main`) mitjançant el mètode `comunicar`.
3) El controlador processa el missatge i, si és necessari, crea instàncies de les classes `SumaM` o `MultM` per executar les operacions en fils separats.
4) Les classes `SumaM` i `MultM` realitzen les operacions de suma i producte de matrius respectivament. Durant l'execució, poden enviar missatges al controlador per actualitzar la interfície gràfica.
5) Quan es completa una operació o es rep un missatge per aturar-la, les classes `SumaM` i `MultM` envien missatges al controlador per desactivar les barres de progrés i actualitzar els resultats.
6) El controlador envia missatges a la classe `FinestraMatriu` per actualitzar la interfície gràfica amb els nous resultats.

Aquest patró per esdeveniments permet una comunicació eficient entre els components de l'aplicació, millorant la modularitat i mantenibilitat del codi. Cada component pot centrar-se en la seva responsabilitat específica, mentre que la comunicació es gestiona de manera uniforme i consistent mitjançant la interfície `Comunicar`.

C. Operacions de Suma i Producte de Matrius

Les classes `SumaM` i `MultM` implementen les operacions de suma i producte de matrius respectivament. Aquestes classes executen les operacions en fils separats per millorar l'eficiència i permetre l'execució concurrent. A continuació es descriuen els passos principals de cada operació:

- **Suma de Matrius**: La classe `SumaM` genera dues matrius aleatòries de mida incremental i calcula la seva suma. El temps d'execució es mesura i s'emmagatzema en la classe `Dades`. També es calcula la constant multiplicativa per a la suma de matrius.
- **Producte de Matrius**: La classe `MultM` genera dues matrius aleatòries de mida incremental i calcula el seu producte. El temps d'execució es mesura i s'emmagatzema en la classe `Dades`. També es calcula la constant multiplicativa per al producte de matrius.

D. Interfície Gràfica

L'aplicació utilitza la llibreria Swing per crear una interfície gràfica que permet a l'usuari introduir la mida de les matrius i iniciar les operacions de suma i producte. La interfície també mostra els resultats en gràfics, permetent a l'usuari visualitzar els costos computacionals asimptòtics de les operacions.

1) Classe `FinestraMatriu`

La classe `FinestraMatriu` és responsable de crear la interfície gràfica principal de l'aplicació. Aquesta classe utilitza diversos components de Swing per permetre a l'usuari introduir la mida de les matrius, iniciar les operacions i visualitzar els resultats. Els elements principals de la interfície inclouen:

- **Camps de text i botons**: La interfície inclou un camp de text per introduir la mida de les matrius (`nField`) i diversos botons per iniciar les operacions i aturar-les. També hi ha un botó per netejar els resultats.
- **Barres de progrés**: Les barres de progrés (`sumarBar` i `multiplicarBar`) indiquen l'estat de les operacions de suma i producte de matrius.
- **Àrea de dibuix**: La classe `Eixos` s'utilitza per dibuixar els gràfics dels resultats. Aquesta àrea de dibuix es troba dins del panell principal (`mainPanel`).

2) Classe `dibuixConstantMult`

La classe `dibuixConstantMult` és responsable de mostrar els resultats de les operacions en taules. Aquesta classe crea dues taules (`taulaSuma` i `taulaMult`) per mostrar els temps reals i esperats de les operacions de suma i producte de matrius. Les taules es troben dins de panells amb títols (`panelSuma` i `panelMult`) per facilitar la seva identificació.

- **Model de taula**: Les taules utilitzen models de taula (`modelSuma` i `modelMult`) per gestionar les dades que es mostren. Aquests models defineixen les columnes de les taules i permeten afegir noves files amb els resultats de les operacions.
- **Actualització de taules**: La classe `dibuixConstantMult` implementa mètodes per afegir noves files a les taules (`afegirFilaSuma` i `afegirFilaMutl`) i per netejar les taules (`cleanTables`).

3) Classe `Eixos`

La classe `Eixos` és responsable de dibuixar els gràfics dels resultats de les operacions de suma i producte de matrius. Aquesta classe extén `JPanel` i sobreescriu el mètode `paintComponent` per dibuixar els eixos i els punts dels gràfics.

- **Dibuix dels eixos**: El mètode `paintComponent` dibuixa els eixos X i Y, així com les línies guia per facilitar la lectura dels gràfics.
- **Dibuix dels punts**: Els punts dels gràfics es dibuixen en funció dels resultats emmagatzemats a la classe `Dades`. Els punts de les operacions de suma es dibuixen en verd, mentre que els punts de les operacions de producte es dibuixen en vermell.
- **Actualització del gràfic**: La classe `Eixos` inclou un mètode `pintar` que es pot cridar per actualitzar el gràfic amb els nous resultats.

4) Finestres de l'Aplicació

L'aplicació crea dues finestres principals per a la interfície gràfica:

- **Finestra Principal**: La finestra principal es crea utilitzant la classe `FinestraMatriu` i mostra els camps de text, botons, barres de progrés i l'àrea de dibuix per als gràfics.

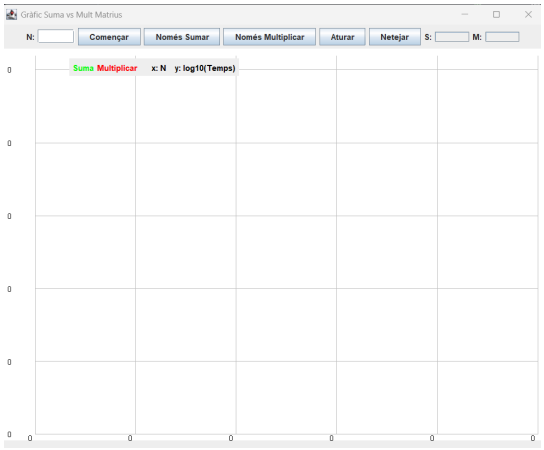

FIGURE 3. **Exemple de la interfície principal implementada.**

- **Finestra de Resultats**: La finestra de resultats es crea utilitzant la classe `dibuixConstantMult` i mostra les taules amb els temps reals i esperats de les operacions de suma i producte de matrius.

FIGURE 4. **Exemple de la interfície de resultats implementada.**

Aquestes finestres permeten a l'usuari interactuar amb l'aplicació, iniciar les operacions i visualitzar els resultats de manera clara i organitzada.

E. Gestió de Fils d'Execució

La classe `Main` utilitza un `ExecutorService` per gestionar els fils d'execució de les operacions de suma i producte de matrius. Això permet executar les operacions de manera concurrent i millorar l'eficiència de l'aplicació. Els fils es poden iniciar i aturar mitjançant missatges enviats a través de la interfície `Comunicar`.

1) ExecutorService

L'ExecutorService' és una interfície que proporciona mecanismes per gestionar un conjunt de fils d'execució. En aquest projecte, s'utilitza per executar les classes `SumaM` i `MultM` en fils separats. La classe `Main` crea un 'ExecutorService' amb un grup de fils fixos mitjançant el mètode `Executors.newFixedThreadPool(3)`:

```
executorService =
Executors.newFixedThreadPool(3);
```

Això permet tenir fins a tres fils d'execució simultanis, assegurant que les operacions de suma i producte de matrius es puguin executar de manera concurrent.

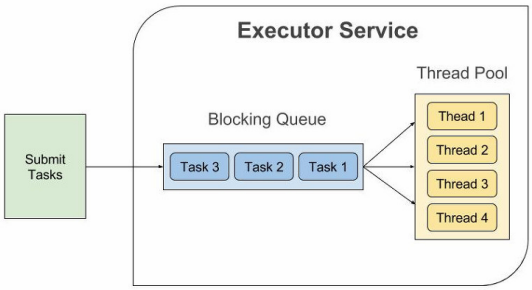

FIGURE 5. **Esquema de funcionament de l'Executor Service a Java.**

2) Inici i Aturada de Fils

Quan l'usuari interactua amb la interfície gràfica per iniciar una operació, la classe `Main` rep un missatge mitjançant el mètode `comunicar`. Aquest mètode processa el missatge i, si és necessari, crea instàncies de les classes `SumaM` o `MultM` per executar les operacions en fils separats. Per exemple, quan es rep un missatge per iniciar les operacions de suma i producte de matrius, es creen i s'executen les instàncies de `SumaM` i `MultM`:

```
SumaM sumaTask = new SumaM(this);
MultM multTask = new MultM(this);

procesos.add(sumaTask);
procesos.add(multTask);

executorService.execute(sumaTask);
executorService.execute(multTask);
```

Això permet que les operacions de suma i producte es puguin executar de manera concurrent, millorant l'eficiència de l'aplicació.

3) Casos de Concurrència

L'aplicació pot gestionar diferents casos de concurrència, depenent de les operacions que l'usuari decideixi executar:

- **Execució Concurrent de Suma i Producte**: Quan l'usuari inicia les operacions de suma i producte de matrius simultàniament, les instàncies de `SumaM` i `MultM` s'executen en fils separats. Això permet que les dues operacions es realitzin al mateix temps, aprofitant els avantatges de l'execució concurrent.
- **Execució de Suma o Producte per Separat**: Quan l'usuari decideix executar només una de les operacions (suma o producte), només es crea i s'executa la instància corresponent (`SumaM` o `MultM`). Per exemple, si es rep un missatge per iniciar només l'operació de suma, es crida al metòde executaClass amb els següents parametres: SumM.class i n. De manera que aquest metode crea i executa només la instància de `SumaM`:

```
Comunicar proces = (Comunicar) classe
    .getConstructor(Main.class)
    .newInstance(this);
procesos.add(proces);
executorService.submit((Runnable) proces);
```

4) Aturada de Fils

Quan es rep un missatge per aturar una operació, el mètode `comunicar` de la classe `Main` envia un missatge a les instàncies de `SumaM` i `MultM` per aturar els fils d'execució. Això es fa establint la variable booleana `stop` a `true`, la qual cosa fa que el bucle principal del mètode `run` s'aturi:

```
for (Comunicar enmarxa : procesos) {
    enmarxa.comunicar("aturar");
}
```

Això assegura que les operacions es puguin aturar de manera controlada i que els recursos es puguin alliberar correctament.

En resum, l'aplicació implementa el patró MVC i el patró per esdeveniments per gestionar les operacions de suma i producte de matrius, mostrant els resultats en una interfície gràfica desenvolupada amb Java Swing. Aquesta arquitectura permet una clara separació de responsabilitats, millorant la modularitat i mantenibilitat del codi.

V. Resultats i Anàlisi

S'exposen els resultats obtinguts, avaluant la modularitat del sistema i l'eficiència en la gestió d'esdeveniments. Es presenta una comparació de temps d'execució i costos computacionals.

Les següents gràfiques mostren els resultats d'executar el programa amb $N = 1000$.

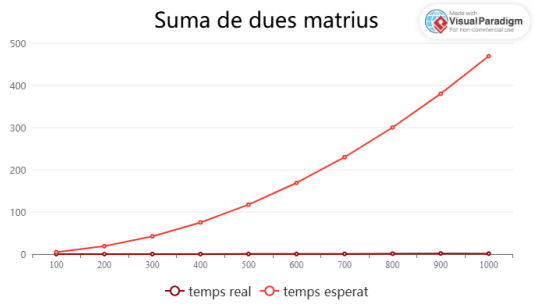

FIGURE 6. Gràfica amb els resultats de sumar dues matrius.

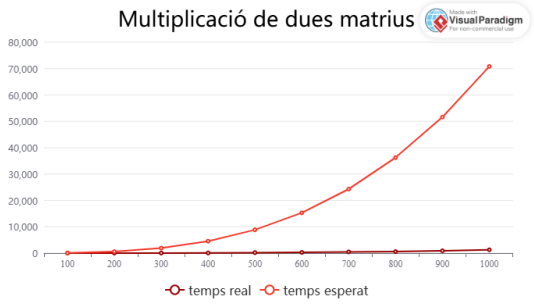

FIGURE 7. Gràfica amb els resultats de sumar dues matrius.

Com es pot veure, el temps esperat de la **Figura 6** creix de manera quadràtica, seguint una funció n^2. Tot i ser ràpid, aquest creixement és més lent que el de la multiplicació de matrius, que té un creixement cúbic n^3. Aquest darrer implica que el temps d'execució es dispara quan augmenta la mida de la matriu, fent inviable l'algorisme `multiplicar` per dimensions molt grans.

A més, hi ha una diferència clara entre el temps real i l'esperat en ambdues gràfiques. El temps esperat es determina mitjançant una constant multiplicativa calculada prèviament, que aproxima el comportament asimptòtic de l'algorisme. Aquesta discrepància existeix perquè el temps esperat marca una cota superior del temps d'execució en el pitjor cas.

Cal considerar que el càlcul està simplificat i no inclou factors com la gestió de memòria, l'eficiència de la caché o les optimitzacions del compilador, que poden influir en el temps real d'execució. Aquí només analitzem el temps pur de les operacions de suma i multiplicació de la classe `Matriu`, sense tenir en compte la càrrega inicial de dades ni altres processos interns.

En resum, el càlcul computacional és útil per estimar la durada d'un algorisme i comparar estratègies, però sempre cal validar els resultats amb proves experimentals per tenir una visió més precisa del rendiment real.

VI. Conclusions

Es resumeixen els avantatges de l'ús del patró MVC i es proposen possibles millores futures per optimitzar el sistema.

Com s'ha mencionat prèviament i en diferents seccions de la memòria, el patró MVC permet separar millor els mòduls d'un programa, facilitant una programació més modular, escalable i fàcil de mantenir. També afavoreix la realització de proves unitàries i l'adaptació a nous requisits, ja que cada component pot evolucionar independentment.

REFERENCES

[1] Oracle. (2014). *ExecutorService Interface (Java Platform SE 8)*. Recuperat de: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

[2] GeeksforGeeks. (2021). *MVC Design Pattern*. Recuperat de: https://www.geeksforgeeks.org/mvc-design-pattern/

[3] GeeksforGeeks. (2021). *Benefit of Using MVC*. Recuperat de: https://www.geeksforgeeks.org/benefit-of-using-mvc/

[4] GeeksforGeeks. (2021). *Analysis of Algorithms — Big-O Analysis*. Recuperat de: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/