Universidad de La Laguna

Proyecto Final. Arquitectura de software (Modelo-Vista-Controlador).

Fabrizzio Daniell Perilli Martín
alu0101138589@ull.edu.es

Diseño Arquitectónico y Patrones

Universidad de La Laguna

5 de enero de 2024

Universidad de La Laguna

Índice

1. Introducción

El documento en cuestión proporciona una descripción detallada sobre el desarrollo de un programa en Java. La aplicación obtiene los datos de las criptomonedas a través de una API y se encarga de notificar a los usuarios suscritos acerca de los precios y otros datos relevantes que se actualizan periódicamente.

2. Objetivo

Adquirir conocimientos y habilidades para aplicar patrones de diseño que faciliten un diseño correcto del software. En esta instancia, se examinan el patrón de diseño *Observador* y la arquitectura de software *Modelo-Vista-Controlador*, ya que forman parte de la solución, permitiendo mejorar la escalabilidad y desacoplamiento del código.

3. Desarrollo

Partiendo de la continuidad de la práctica previamente desarrollada, se llevó a cabo una refactorización del código con el fin de adaptarlo a la arquitectura de software Modelo-Vista-Controlador (MVC). Este proceso se concentró en la importancia de dividir las responsabilidades de cada componente para mejorar la calidad y el diseño del sistema.

Se implementó un cuadro de mando para la vista de escritorio después de completar la refactorización. Este cuadro de mando utilizó la API para obtener datos y crear gráficas de visualización. Posteriormente, se creó una vista de CLI para disponer de dos versiones de la aplicación. Es importante tener en cuenta que esta versión carece de un cuadro de mando.

Finalmente, se añadió la funcionalidad de seleccionar las criptomonedas disponibles en la aplicación mediante un archivo .csv. Este archivo contiene los nombres y las imágenes de las criptomonedas que serán mostradas en la aplicación.

3.1. Diseño

A través del diseño se proporciona una estructura clara y organizada del sistema, facilitando la comprensión del software por parte de los desarrolladores y otros interesados.

3.1.1. Estructura

Principalmente el proyecto está dividido en diferentes paquetes que separan las diferentes funcionalidades del proyecto *(Ver Figura 1)*.

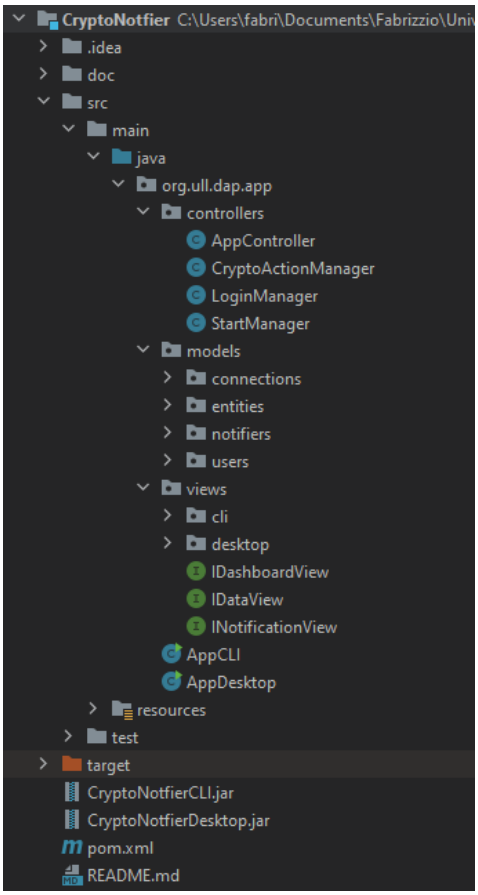

Figura 1: *Estructura del proyecto.*

Existen tres principales paquetes relacionados con la arquitectura:

- **Paquete controllers**: Contiene toda la lógica de la aplicación, incluyendo la

gestión de eventos de las vistas.

- **Paquete models**: Este paquete está dividido en cuatro subpaquetes, cada uno de los cuales tiene una función diferente. El primero gestiona las conexiones con la API y el archivo CSV, el segundo almacena los datos de las consultas, y los dos últimos, el paquete *notifiers* y *users* son componentes del patrón observador

- **Paquete views**: Engloba la parte del cliente, es decir, las interfaces de usuario. En este caso, se dividen en dos tipos: *desktop* y *cli*

Es importante destacar que el proyecto posee dos funciones principales: una para iniciar el proyecto a través de la línea de comandos y otra para iniciar el proyecto mediante ventanas gráficas.

La estructura del proyecto sigue un arquetipo, plantilla utilizada para crear nuevos proyectos en Maven. Estos arquetipos ayudan a estandarizar la estructura de los proyectos, facilitando su creación.

3.1.2. Patrones de diseño

Los patrones de diseño representan soluciones típicas a problemas frecuentes en el diseño de software. Cada patrón actúa como una plantilla que puede ser personalizada para abordar un problema de diseño específico, proporcionando una descripción clara y sencilla de la solución para dicho problema.

Como se había mencionado anteriormente en el proyecto se hace uso del patrón arquitectonico Modelo-Vista-Controlador.

Se encarga de separar los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador. Por un lado define componentes para la representación de la información y, por otro lado, para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento *(Ver Figura 2)* .

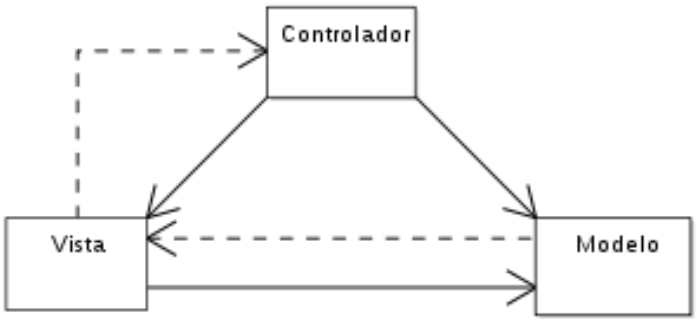

Figura 2: *Diagrama de Clases. Modelo-Vista-Controlador.*

Un diagrama sencillo que muestra la relación entre el modelo, la vista y el controlador. Nota: las líneas sólidas indican una asociación directa, y principalmente las punteadas una indirecta

También es importante destacar el patrón de diseño *Observador*. Define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Se clasifica como un patrón de diseño comportamiento, por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a clases y objetos.

Los componentes del patrón son los siguientes *(Ver Figura 3)*:

- **Sujeto**: El sujeto proporciona una interfaz para agregar y eliminar observadores. El Sujeto conoce a todos sus observadores.
- **Observador**: Define el método que usa el sujeto para notificar cambios en su estado.
- **Sujeto concreto**: Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- **Observador concreto**: Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

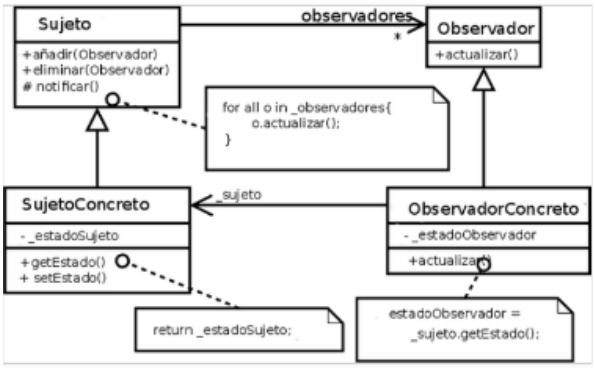

Figura 3: *Diagrama de Clases. Patrón Observador.*

En relación con el proyecto, los elementos del patrón corresponden a las siguientes clases:

- **Sujeto**: *Observable*.
- **Observador**: *IObserver*.
- **Sujeto concreto**: *CryptoNotifier*.
- **Observador concreto**: *User*.

3.1.3. Diagrama de clases

Para obtener representación gráfica que ilustra la estructura y relaciones entre las clases en un sistema de software se presenta el diagrama de clases del sistema. El diagrama será separado por cada componente de la arquitectura para una mejor visualización *(Ver Figura 4, 5 y 6)*

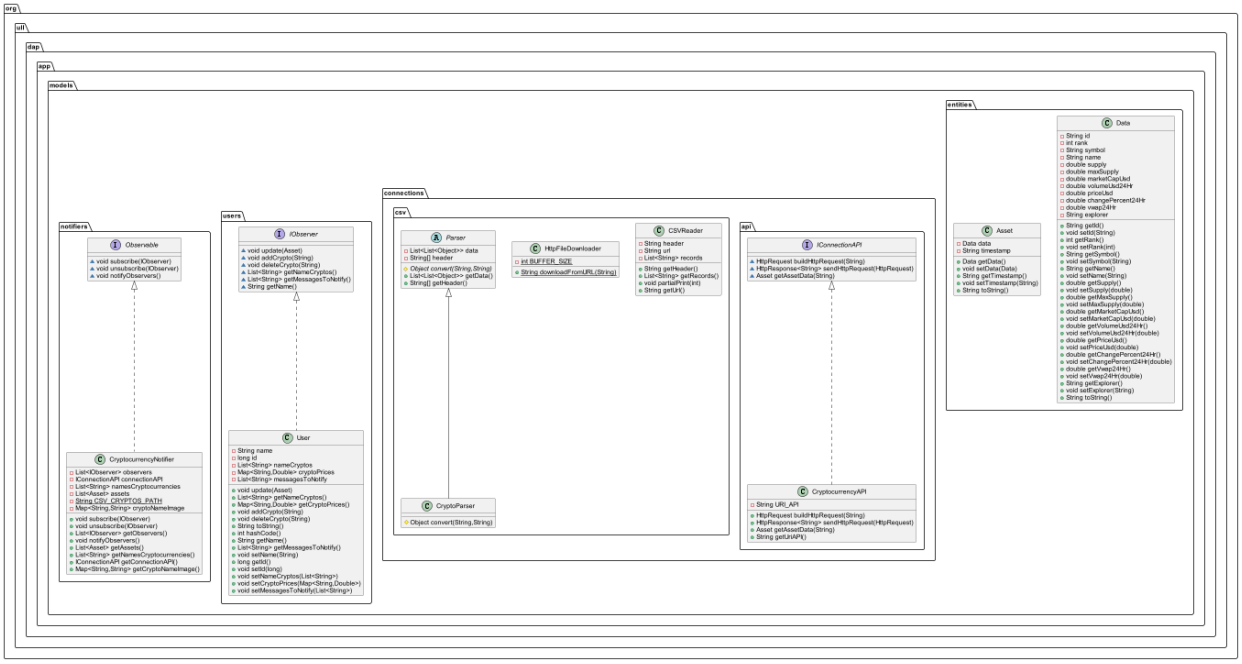

Figura 4: *Diagrama de clases del proyecto. Modelos.*

Se puede observar la presencia del patrón observador en los paquetes *users* y *notifiers*, la clase sujeto del patrón es la que interatúa directamente con el controlador. En el paquete *connections* se encuentran todas las conecciones que necesita establecer el sujeto del patrón para obtener los datos y finalmente el paquete *entities* contienes las clases que almacenarán los datos de las criptomonedas.

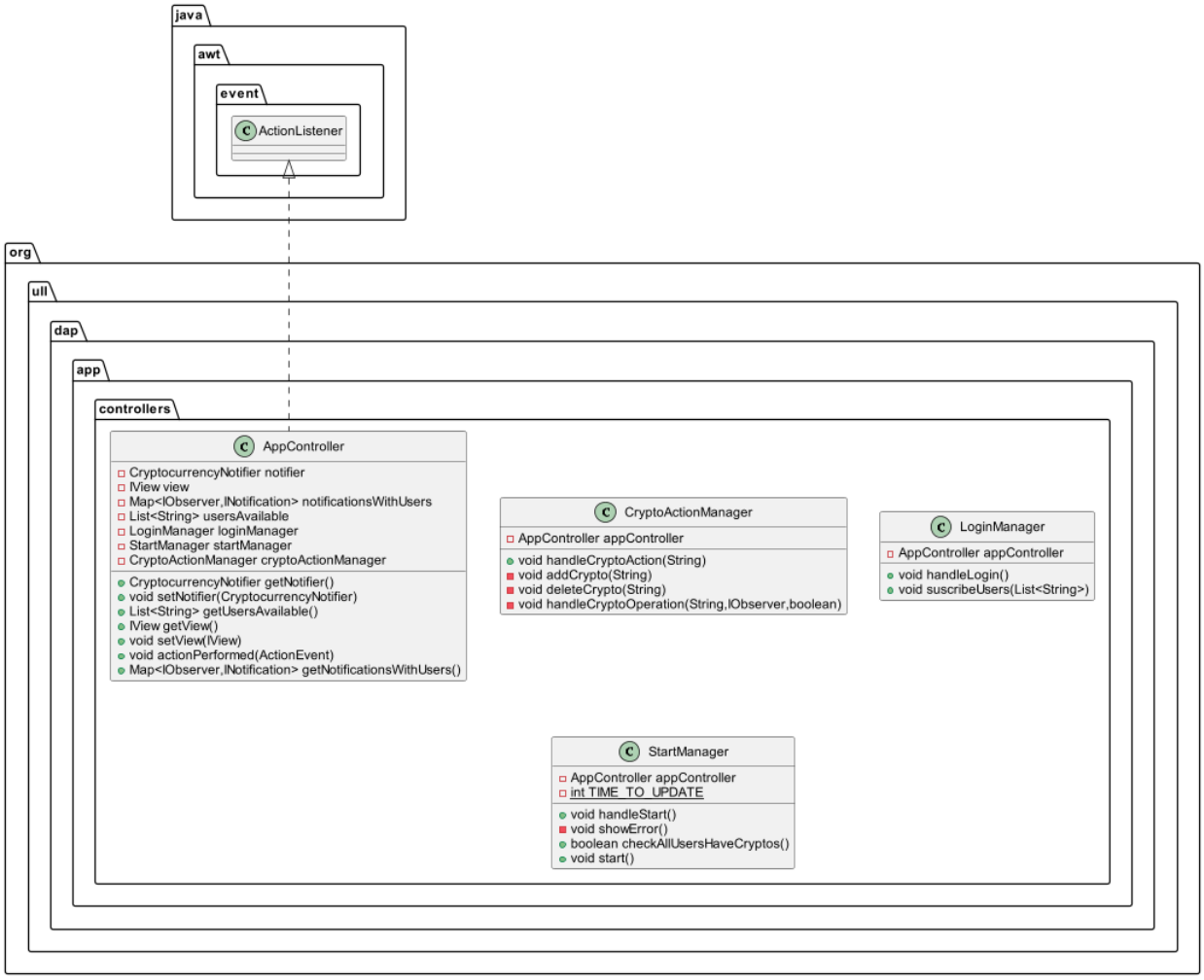

Figura 5: *Diagrama de clases del proyecto. Controladores.*

En el diagrama se puede apreciar cuatro clases. La clase *AppController* actúa como controlador principal, encargándose de gestionar los eventos y delegar las responsabilidades a las otras clases. Por ejemplo, la clase *LoginManager* se encarga de almacenar a los usuarios seleccionados y suscribirlos al notificardor. La clase *CryptoActionManager* se encarga de gestionar las criptomonedas para cada usuario, mientras que la clase *StarManager* se encarga de verificar que todo los datos sean correctos para iniciar la ejecución del programa.

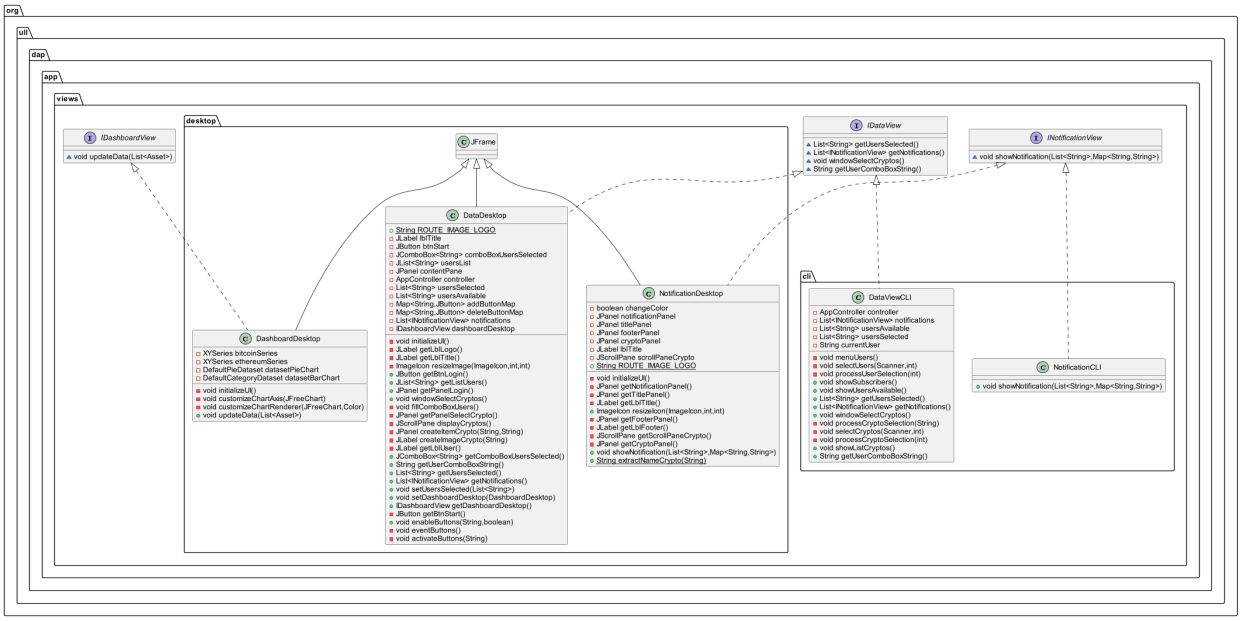

Figura 6: *Diagrama de clases del proyecto. Vistas.*

Existen tres tipos de vistas:

- **IDataView**: Se encarga de obtener los datos introducidos por el usuario y enviarlos al controlador, tiene una implementación por línea de comandos y otra por ventanas gráficas.

- **INotificationView**: Contiene el display de visualización de las notificaciones que recibe cada uno de los usuarios, igualmente tiene dos implementaciones, por *CLI* y *GUI*.

- **IDashboardView**: Visualiza las gráficas creadas a partir de los datos obtenidos de cada consulta que se van actualizando, a diferencias de las otras vistas sólo tiene la implementación por ventanas gráficas.

3.2. Visualización

La ejecución del programa se realiza mediante ventanas gráficas o por línea de comandos *(Ver Figura 7, 8, 9, 10, 11, 12 y 13)*:

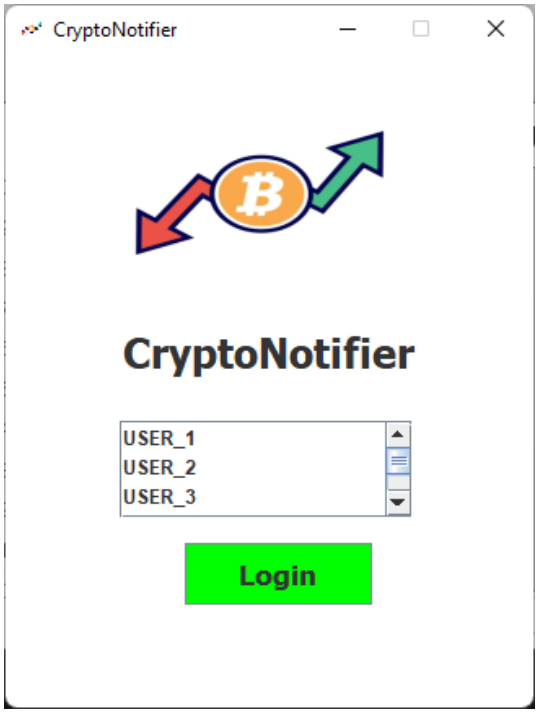

Figura 7: *Ventana de login. Vista GUI.*

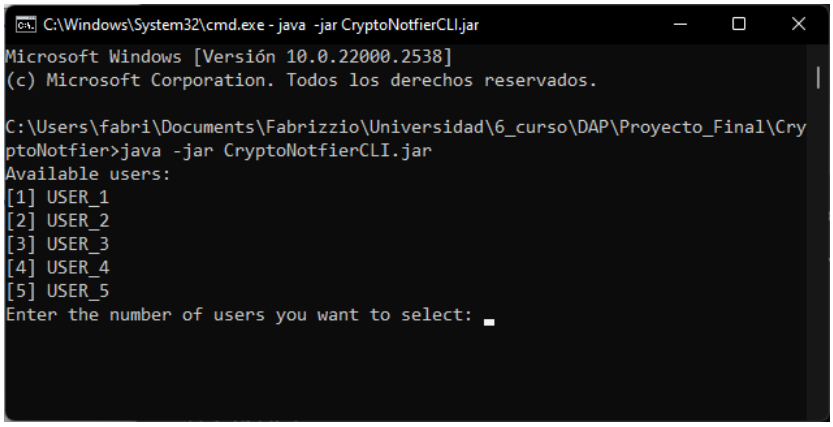

Figura 8: *Ventana de login. Vista CLI.*

Las figuras anteriores representan las dos vistas que se encargan de la selección de usuarios que serán notificados cuando el precio de las criptomonedas cambie en un periodo de tiempo.

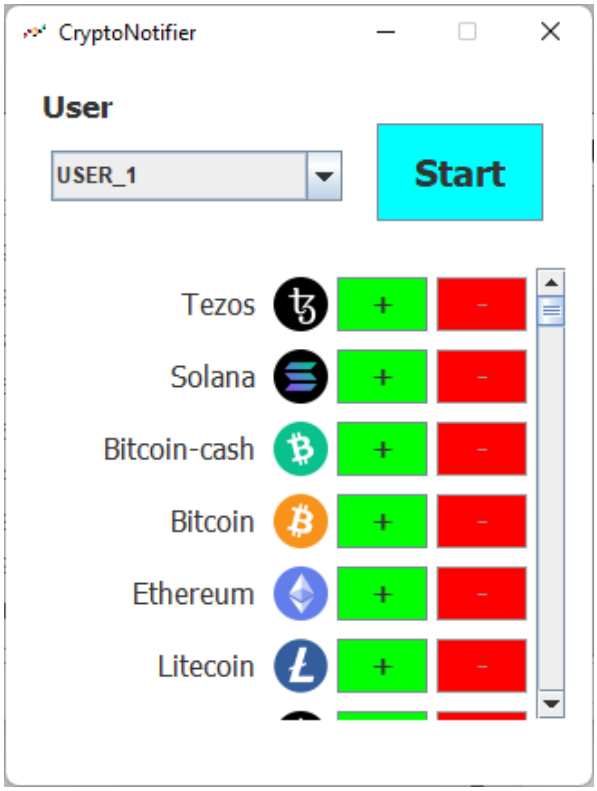

Figura 9: *Ventana de selección de criptomonedas. Vista GUI.*

Figura 10: *Ventana de selección de criptomonedas. Vista CLI.*

Cada uno de los usuarios selecciona las criptomonedas que desea recibir información, en las figuras anteriores se muestran las dos visualizaciones disponibles.

Figura 11: *Ventana notificación. Vista GUI.*

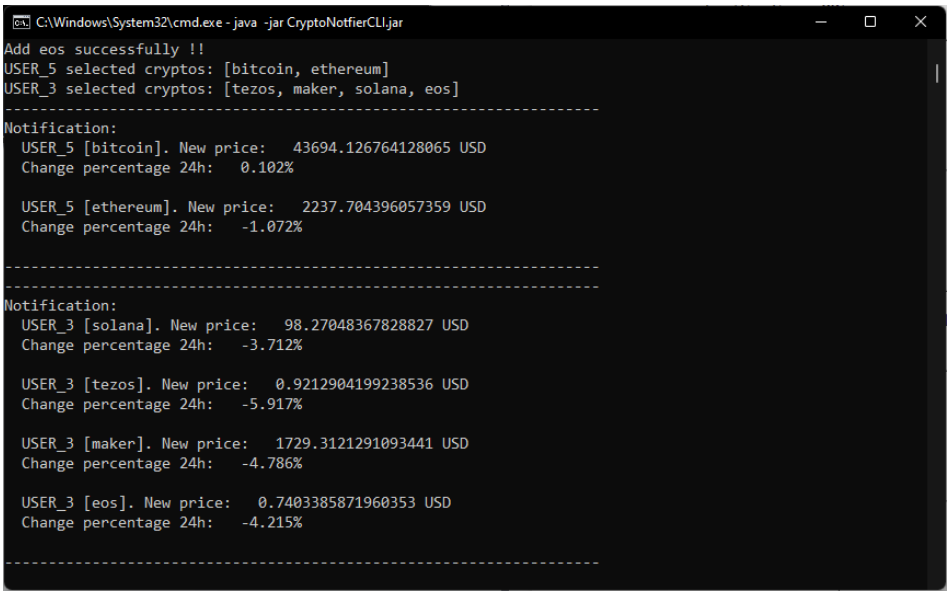

Figura 12: *Ventana de notificación. Vista CLI.*

Cada usuario tiene un display con las notificaciones que recibe, estás notificaciones llegan a la una vista que se puede representar por línea de comandos o por ventanas gráficas.

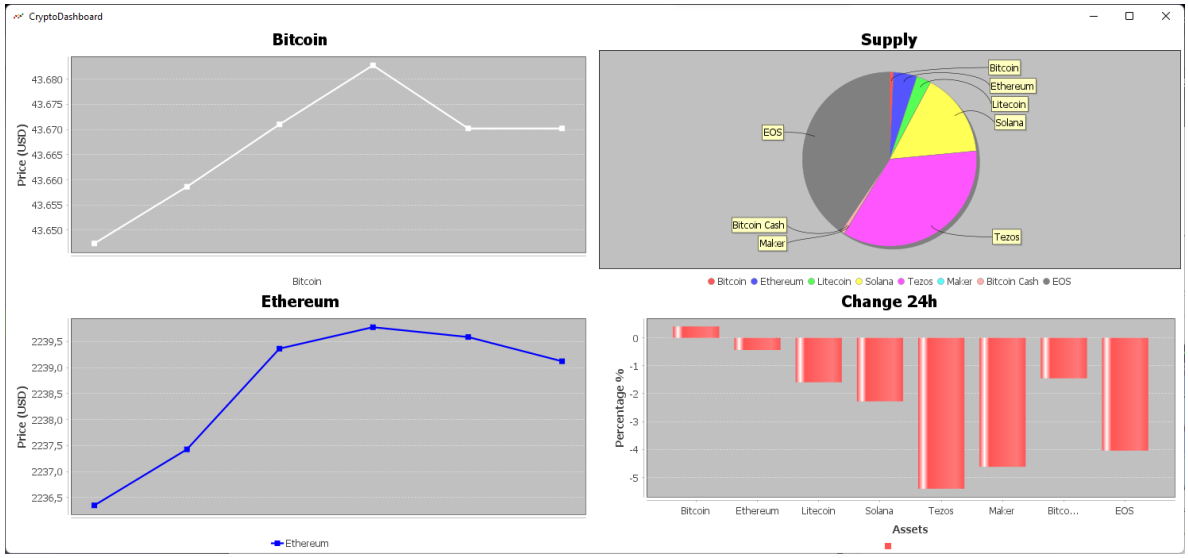

Figura 13: *Ventana del cuadro de mando. Vista GUI.*

La vista de escritorio contiene una ventana que representa un cuadro de mando con las gráficas en tiempo real, en la vista por línea de comandos no se encuentra disponible.

4. Estimación de tiempo

En una fase inicial, se realizó una estimación temporal para cada tarea basándome en la experiencia acumulada en prácticas anteriores, descomponiendo el proyecto final en subtareas más pequeñas y manejables. *(Ver Tabla 1).*

Tarea	**Tiempo estimado**	**Tiempo empleado**
Diseño	1 h	30 min
Refactorización	2 h	4 h
Cuadro de mandos	2 h	4 h
Vista CLI	1 h	1 h
Documentación Doxygen	1 h	30 min
Correción de errores	30 min	2 h
Informe	30 min	1 h
Total	**8 h**	**13 h**

Cuadro 1: *Tabla de estimación de tiempo.*

En relación con las tareas planteadas, el proceso de diseño fue el que requirió menos tiempo, gracias a implementación del patrón de diseño que facilitó su desarrollo. Por otro lado, la corrección de errores, así como la refactorización del código anterior y la creación del cuadro de mandos, representaron las tareas que consumieron más tiempo. Este incremento se atribuyó a la aparición de resultados inesperados durante la ejecución del programa y a la falta de experiencia en la nueva arquitectura implementada, factores que no se tuvieron en cuenta en la planificación inicial.

5. Referencias

1. https://es.wikipedia.org/wiki/Observer_(patrÃşn_de_diseÃśo). (Teoría del Patrón Observador).

2. https://refactoring.guru/es/design-patterns/observer. (Teoría del Patrón Mediador).

3. https://docs.coincap.io. (Documentación API).

4. https://coincap.io. (Página web Coincap).

5. https://acortar.link/pgNHKC. (Datos CSV Criptomonedas).

6. https://acortar.link/WNCDOy. (Teoría cuadro de mando).

7. https://www.jfree.org/jfreechart/javadoc/org/jfree/chart/JFreeChart.html. (Documentación JFreeChart).

8. http://ite4.com/ejemplos-cuadros-de-mando-con-jet-report.html. (Ejemplos de cuadros de mando).

9. https://es.wikipedia.org/wiki/ModeloâĂŞvistaâĂŞcontrolador. (Teoría arquitectura MVC (Modelo-Vista-Controlador)).

10. https://acortar.link/Z24Mh1. (Teoría arquitectura MVC).

11. https://github.com/Fabrizzioperilli/CryptocurrenciesNotifier.git. (Repositorio del proyecto).

12. https://prezi.com/view/DM4vIGZcFXt30uarqOR1/. (Presentación)

13. Transparencias del campus virtual de la asignatura.