***Memoria Grupal Práctica IS – Grupo 6***

***¿Cómo hemos usado Scrum?***

*Roles*

En cuanto a los roles, todos fuimos Desarrolladores, y además Marcos fue el Scrum Master. Los Desarrolladores se ocuparon de decidir de puntuar y elegir las tareas adecuadas antes de cada Sprint, además de realizarlas durante el transcurso del mismo.

*Aplicación de Scrum y resolución de problemas*

La aplicación del framework Scrum a la práctica fue bastante simple. Para ello, se emplearon Sprints semanales donde se realizaban una fracción de todas las historias del Product Backlog. Cada viernes lectivo se llevaba a cabo una reunión en la que se evaluaban los resultados del Sprint anterior, y se reflexionaba sobre el trabajo durante el mismo. A continuación, se puntuaban y asignaban las historias del próximo Sprint, y durante esa semana, los desarrolladores las completaban.

Siempre que el equipo se topó con algún tipo de duda, problema o conflicto, lo intentó resolver consultando a las personas adecuadas. Por ejemplo, si alguien tenía una pregunta sobre el funcionamiento de una parte del código que había elaborado otra persona, le preguntaba a ella. Por otro lado, hubo una ocasión en la que tuvimos que consultar a nuestro profesor sobre problemas técnicos con GitHub.

En cuanto a los artefactos de Scrum, el Product Backlog fue elaborado por el Product Owner, que también proporcionó el Objetivo del Producto. Esto fue clave a la hora de entender las historias y tareas. El Sprint Backlog fue elaborado por los desarrolladores a escala semanal y permitió al equipo avanzar con el proyecto a un ritmo más realista y consistente gracias al Objetivo del Sprint. Por último, los incrementos fueron cada uno de los avances concretos que acercaban el proyecto al Objetivo del Producto, y que fueron testeados individualmente para asegurar su funcionamiento.

***Repositorio (***[***enlace***](https://github.com/PabloHernandezUDC/practica-is)***)***

*Uso*

Durante la realización del proyecto se utilizó un repositorio de GitHub para sincronizar el código entre desarrolladores y archivar los cambios de manera eficiente. Cada desarrollador, a su vez, usaba su repositorio local para mantener los cambios durante el proceso de edición.

*Ramas y conflictos de merge*

La política de creación de ramas fue más bien sencilla. Además de *main*, cada desarrollador creó una rama a su nombre, y él/ella debía asegurarse de trabajar única y exclusivamente en su rama hasta que el código estuviese 100% preparado para integrarse en la rama principal *main*. Resolver los conflictos de *merge* en *main* era responsabilidad del desarrollador que estuviese incorporando sus cambios, y por eso era importante trabajar sobre código que estuviese al día.

*Nombrado de commits*

En cuanto al nombrado de commits, no se disponía de una política concreta. La única regla era intentar describir brevemente el contenido del commit en el mensaje, pero nada más.

***Explicación del código***

*Patrones/aspectos de diseño*

Realizamos una búsqueda en todo el código que implementamos para identificar patrones de diseño, a partir de la cual obtuvimos la siguiente tabla:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Creacionales | Estructurales | De comportamiento |
| *classModel.py* | No presenta patrones creacionales. | Facade no se implementa explícitamente, pero la clase proporciona métodos simples para acceder y modificar atributos, lo que se asemeja al objetivo del patrón. | No presenta patrones de comportamiento. |
| *dataOp.py* | No presenta patrones creacionales. | Las librerías pandas y tkinter proporcionan funcionalidades que coinciden con el patrón Facade. | En el bucle for dentro de la función getColumns() se recorren las columnas de un DataFrame, lo que refleja ciertos aspectos del patrón Iterator. En la función createColumns() hay una observación de eventos en la interfaz gráfica que desencadena una acción al hacer click en un botón, lo que se asemeja al patrón Observer. |
| *guiOp.py* | Factory Method no está directamente presente, pero la función readFile() y loadModelFromPickleObject() de los módulos dataOp y modelOp respectivamente podrían ser considerados métodos de fábrica si internamente crean y devuelven objetos de manera específica según ciertos parámetros. | Existe un uso potencial del patrón Facade en el módulo customTkinter. La creación de widgets personalizados como CTkButton proporciona una interfaz simplificada para trabajar con los elementos de la interfaz gráfica. | No presenta patrones de comportamiento. |
| *main.py* | No presenta patrones creacionales. | No presenta patrones estructurales. | No presenta patrones de comportamiento. |
| *modelOp.py* | La función makeModel() podría considerarse similar al patrón Factory Method. | La librería customTkinter se asemeja al patrón Facade al proporcionar una interfaz simple para un subsistema más grande y complejo (tkinter en este caso). La función loadModelFromPickledObject() podría considerarse como una forma del patrón Proxy ya que actúa como un intermediario entre la solicitud de cargar el modelo y la presentación de la información correspondiente. | La función makeAndShowGraph() podría interpretarse como un patrón Observer, donde el gráfico es el observador que se actualiza cuando el modelo cambia. |
| *prediction.py* | Factory Method no se utiliza explícitamente, pero la función makePrediction() muestra similitudes. | Se usa el patrón Facade con la función makePrediction() para encapsular la complejidad de la interfaz de usuario para realizar una predicción. | No presenta patrones de comportamiento. |
| *readDbOp.py* | Factory Method no se muestra explícitamente, pero la función readSQL() se asemeja. | La función readSQL() puede considerarse como una implementación del patrón Facade. | Las funciones readRows() y readOrdered() muestran un comportamiento similar al patrón Iterator. Además, la estructura de las funciones sigue un patrón con un método común, lo cual tiene similitudes con el patrón Template Method. |
| *regression.py* | No presenta patrones creacionales. | No presenta patrones estructurales. | No presenta patrones de comportamiento. |

Un ejemplo de patrón de diseño sencillo de ubicar fue el patrón Iterator, el cual podemos encontrar en varias funciones que utilizan un bucle for, como por ejemplo las funciones clearFrame(), getColumns() o createColumns() en el módulo dataOp.py.

A parte de esto, también implementamos el patrón Template Method, lo que nos permitió crear una plantilla con las funciones comunes para realizar una regresión lineal, y una subclase para la regresión lineal simple. Esto nos da la opción de crear otra subclase para poder realizar también regresión lineal múltiple. El patrón está implementado en el módulo regression.py y se llama a alguna de sus funciones en el módulo modelOp.py.

*Pruebas y planes de pruebas*

marcos hazlo tú no tengo ni idea

*Guías o normas de estilo utilizadas*

La guía de estilo que se utilizó en la totalidad del código fue PEP-8. Para el nombramiento de módulos tan solo se escribió en camelCase un nombre breve y que resumiese las funciones que contiene. Para el nombramiento de variables y funciones se disponía de la siguiente política:

1. En inglés
2. En camelCase ("oneTwoThree")
   1. Salvo constantes, entonces SNAKE\_CASE\_CAPS
3. El nombre ha de ser descriptivo (mejor "columnCounter" que "n" o "columns")

*Equipo de desarrolladores*

Marcelo Ferreiro Sánchez: [marcelo.fsanchez@udc.es](mailto:marcelo.fsanchez@udc.es)

Claudia García López: [claudia.garcia.lopez@udc.es](mailto:claudia.garcia.lopez@udc.es)

Marcos Grobas Martínez: [marcos.grobas@udc.es](mailto:marcos.grobas@udc.es)

Pablo Hernández Martínez: [pablo.hernandez.martinez@udc.es](mailto:pablo.hernandez.martinez@udc.es)

Andrea Varela Fernández: [andrea.varela.fernandez@udc.es](mailto:andrea.varela.fernandez@udc.es)