**TRABAJO PRÁCTICO N°3\_c - ING. SOFTWARE II**

**ALUMNO**: FERNANDO PÉREZ

**DOCENTES**: Dr. Pedro E. Colla y Hernán Sanchez

**UADER-FCYT**

**1)** Para utilizar el patrón cadena de responsabilidad creamos una clase abstracta llamada Handler que define una interfaz para las solicitudes que se quieren hacer. Esta clase tiene un método abstracto llamado con el mismo nombre que lo que hace es recibir un número como argumento y devolver True si la solictud fue manejada con éxito o False si no lo fue.

Luego se crea la clase concreta que se llama PrimoHandler que hereda de Handler su método con la modificación de que este método verifique si el número como argumento es primo o no, en caso de serlo lo imprima en pantalla y devuelva True, caso contrario no haga nada y devuelva False.

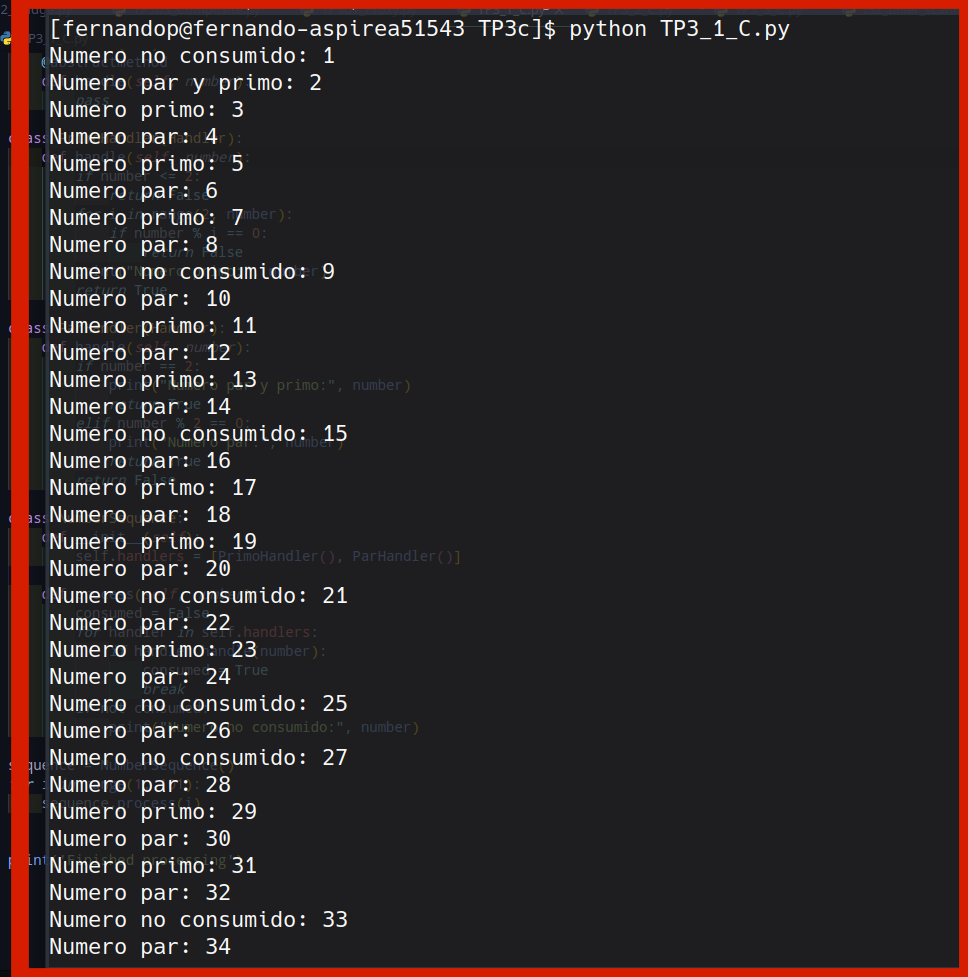
También se crea otra clase concreta llamada ParHandler que hereda de Handler su método con la diferencia que este verifica si los numeros son pares o no, en caso de serlo lo imprime por pantalla y devuelve True, y caso contrario, no hace nada y devuelve False.

Por último, creamos una clase llamada NumberSequence que tiene una lista de objetos Handler con un método llamado process que toma un número como argumento y lo pasa a través de la cadena de responsabilidad. En caso de que ningún objeto handler pueda manejar la solicitud el número se marca como no consumido y se muestra por pantalla.

Como se muestra a continuación:



Y acá un ejemplo de uso utilizando su implementación:

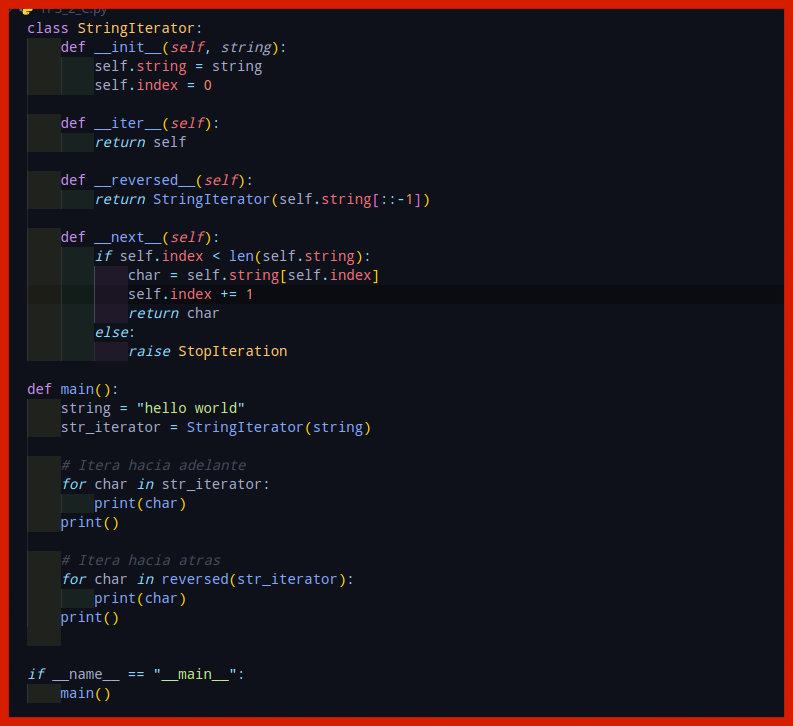


**2)**  Para implementar una clase bajo el patrón Iterator que almacene una cadena de carácteres y permita recorrerla en sentido directo y reverso creamos una clase StringIterator que implementa métodos privados como \_\_iter\_\_(), \_\_reversed\_\_(), y \_\_next\_\_() que nos permiten recorrer el recorrido de la cadena en ambos sentidos, el motivo por el cual se utilizan los guiones antes y después del método definido es para evitar conflictos con palabras claves utilizadas en el lenguaje.

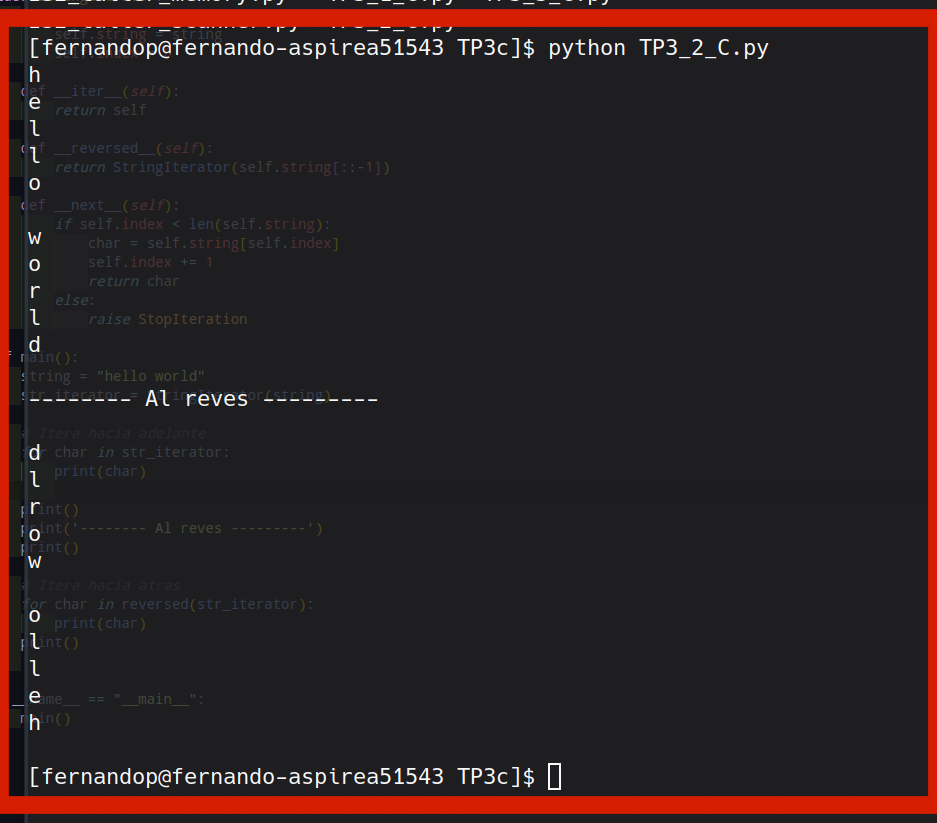
Primero el método \_\_iter\_\_ lo que hace es recorrer la cadena hacia adelante, el método \_\_next\_\_ devuelve el siguiente carácter de la cadena, y el método \_\_reversed\_\_ devuelve el carácter anterior de la cadena.

Luego creamos en el método main una variable string que contenga un mensaje y también creamos un objeto de tipo StringIterator que almacene dicho string. Y por último definimos dos ciclos, uno que recorra el string desde atrás hacia adelante y otro que haga lo contrario, esto con los métodos respectivos del StringIterator.

Se muestra imagen del código a continuación:



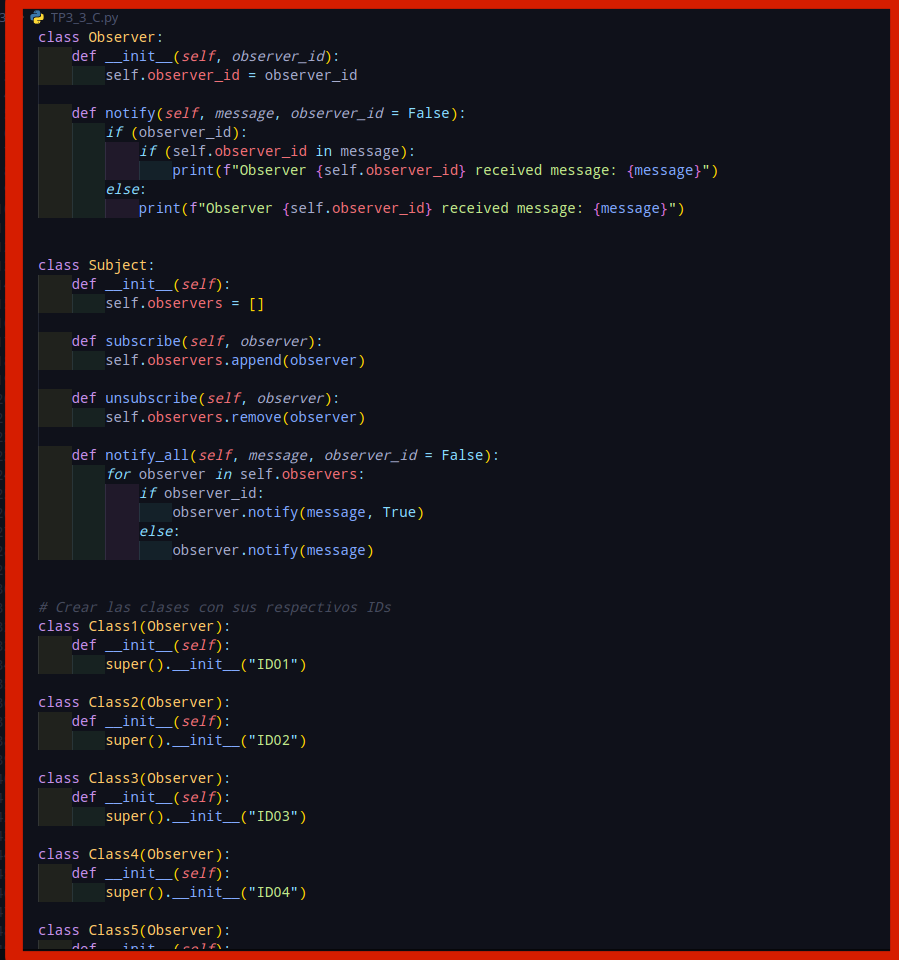
Y acá de su funcionamiento:



**3)** Para implementar una clase bajo el patrón Observer donde una serie de clases estén subscriptas, se crea una clase Observer con un método notify que verifique si el ID del observador coincida con el ID del mensaje emitido. Luego se crea otra clase Subject con métodos subscribir y cancelar subscripción, además del método notify\_all que notifica a todos los usuarios subscriptos, hay dos formas una es pasando un parámetro junto con el mensaje booleano True que indica que solo se le notifique a las clases que se encuentren en el mensaje y la segunda es no pasando ningún parámetro booleano junto con el mensaje que está en false por default, el cuál notifica el mensaje a todas las clases suscritas. El primer ejemplo es con el parámetro True y el segundo con el parámetro False.

Luego se crean 5 clases con sus propios IDs para diferenciarse, después se crea un objeto Subject para cada clase y se subscriben.

Imagen del código:



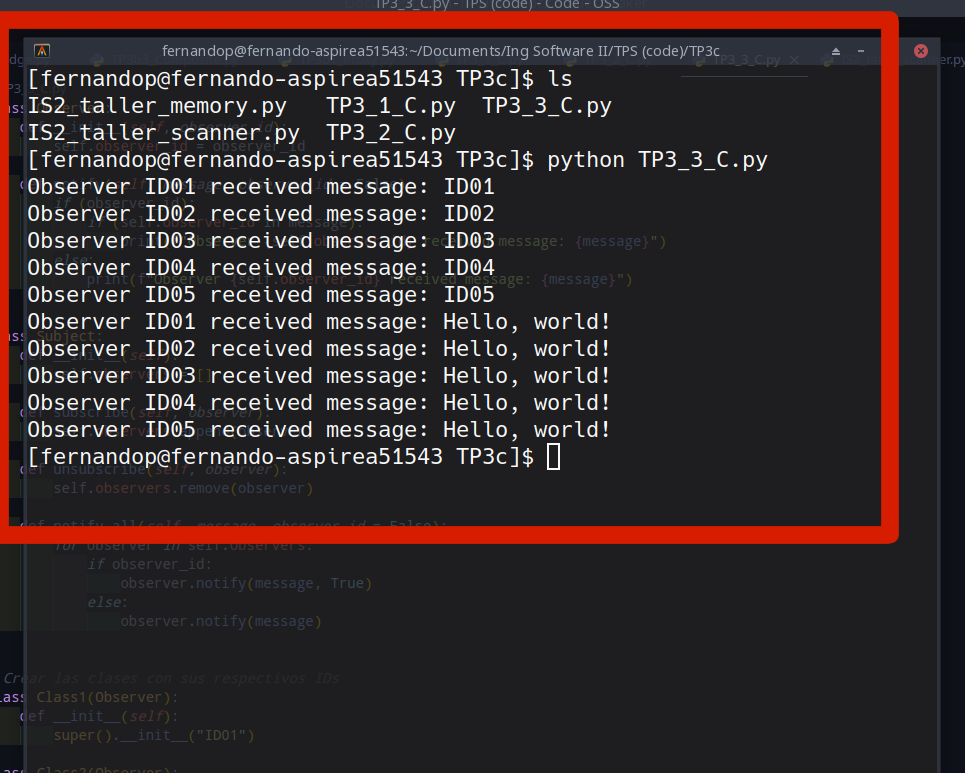
Primer ejemplo:

Se emiten 9 IDs y se notifica a los observadores subscriptos solo el mensaje que les corresponda. Con lo cual, al menos 5 IDs emitidos coinciden con los IDs de las clases implementadas debido a eso se mostraría el contenido a esas respectivas clases, en este caso el mensaje que se les muestra es las mismas IDs emitidas.

Segundo ejemplo:

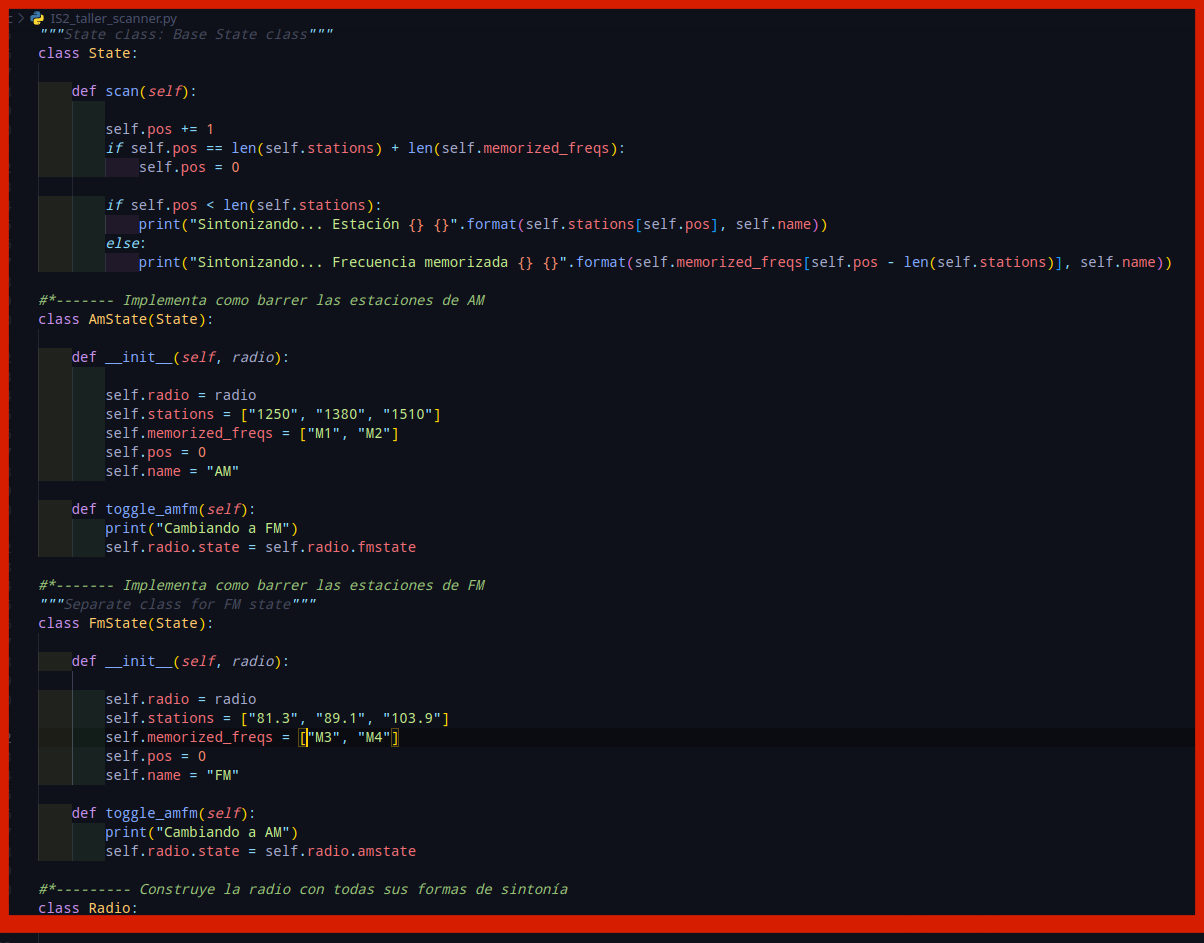
Se emite un mensaje “Hello, world!” para todas las clases suscriptas a diferencia del ejemplo anterior que solo el mensaje es para la ID que se encuentre en dicho mensaje, en éste caso los mensajes emitidos corresponde a todas las clases, esto es por medio del parámetro booleano mencionado anteriormente que en este caso está en False.

Imagen de ambos mensajes de los ejemplos anteriores:

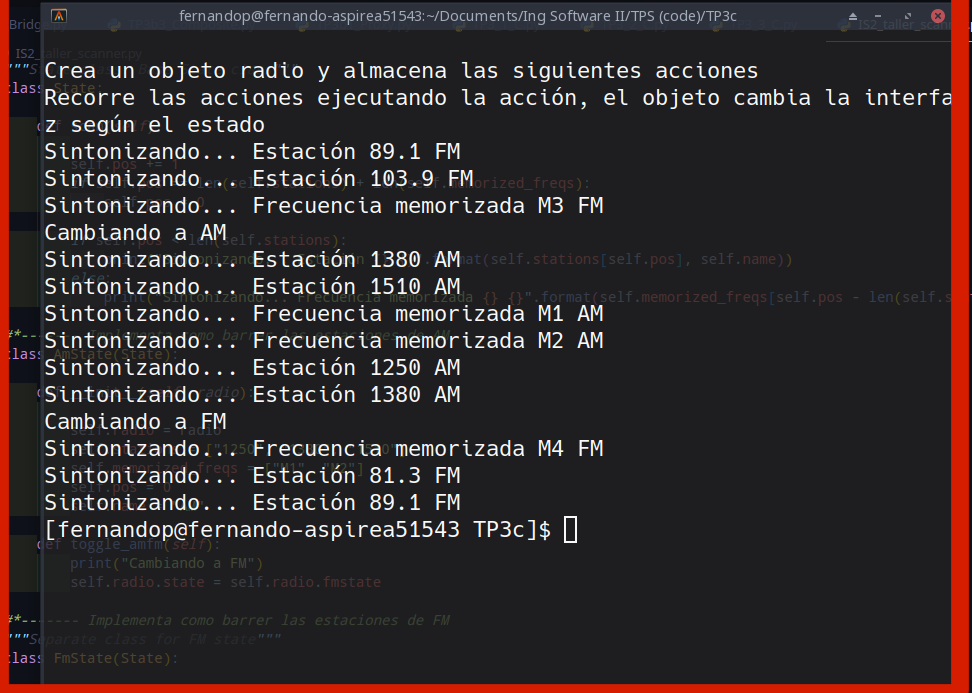


Como se ve, al principio los mensajes van solo a las IDS correspondientes y en el último el mensaje es general para todas las clases subscriptas.

4) Al modificar el programa IS2\_taller\_scanner.py para que incluya una serie de frequencias memorizadas etiquetadas como M1, M2, M3 y M4. Lo que hacemos es crear un nuevo atributo en las estaciones de AM y FM de frequencias memorizadas, en este caso le asignamos M1, M2 a la estación AM y M3, M4 a la estación FM. De esta forma, lo que también hacemos es modificar la clase abstracta State que lo que haga es corroborar que la posición sea menor a la suma de los tamaños de la estación y las frequencias memorizadas, así si es menor recorra todas las frequencias de la estación que corresponda y una vez recorridas todas las frequencias finalice en recorrer las frequencias memorizadas asignadas a esa estación. Como se muestra a continuación el ejemplo del código:



Y a su vez la siguiente imagen de su funcionamiento:

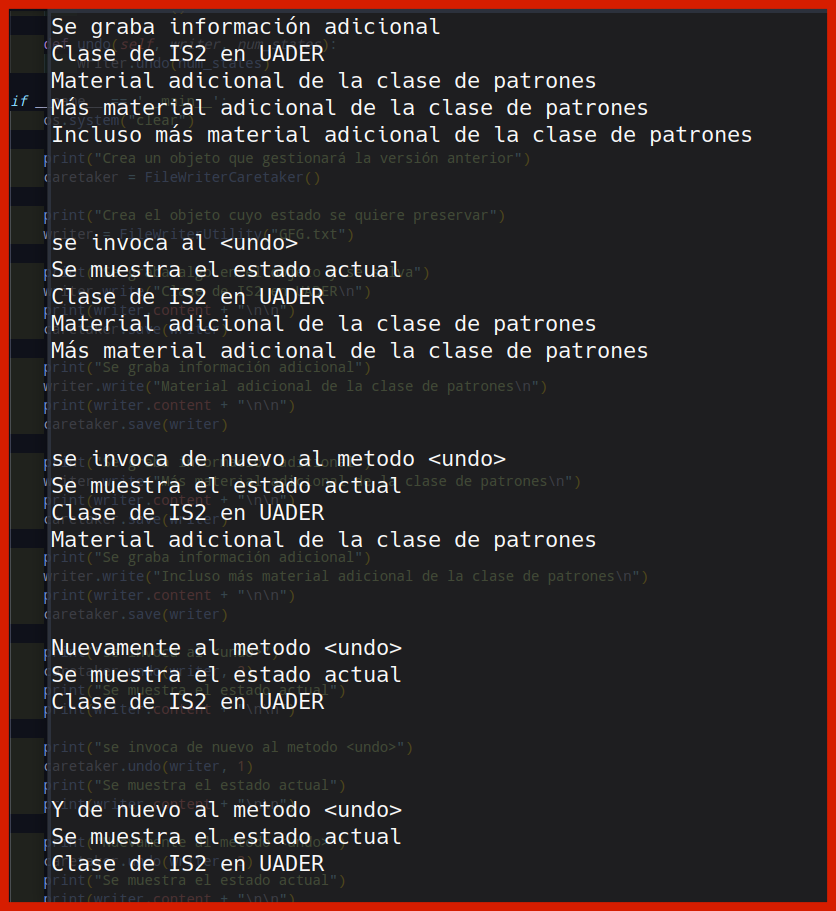


**5)** Se modifica el programa para que la clase tenga la capacidad de almacenar hasta 4 estados en el pasado y poder recuperarlas en cualquier momento pasando como parámetro si se desea recuperar el anterior y los anteriores. Esto se logra mediante la modificación de las clase FileWriterUtility para realizar un seguimiento de los últimos 4 estados utilizando una lista de objetos Memento. Y a la vez modificamos la clase FileWriterCaretaker para que tome un argumento que represente el número de estados que se desea recuperar en el método undo() y además corrobore que no exceda del límite de estados almacenados, de esta forma los estados que se desea recuperar elimina los últimos hasta llegar al objetivo y volver el archivo al estado que se desea. Y el método save() también lo modificamos para que verifique que el número de estados no exceda de 4, y si es así, elimine el estado más antiguo de la lista y agregue el estado actual sin guardar a la lista de estados guardados y así mantener la informarción.

Ejemplo de código a continuación:



Mostramos el ejemplo del código en ejecución:



Como se puede ver en el último estado al no haber nada más que recuperar se mantiene igual.

**GITHUB REPO:** <https://urlis.net/patr-de-comport>