# Cadena de responsabilidades (Aegon)

Para la solución de Aegon se utilizan cadenas de responsabilidades para la ejecución ordenada de una serie de acciones que se tienen que ir ejecutando una a una y parar toda la cadena en el caso de que una parte falle.

Estas cadenas se componen de una serie de “Handlers” y cada uno de ellos se encarga de una parte concreta sin importar que “handlers” se han ejecutado anteriormente y sin conocer que “handlers” se tienen que ejecutar posteriormente.

Handler

Handler

Handler

Handler

Durante la cadena, se comparte entre cada “handler” un contexto con datos que nos sirve para conocer el estado y mover valores a lo largo de la cadena.

En resumen, se sigue el patrón de diseño de comportamiento “Chain of Responsibility Pattern”.

Como añadido al patrón base, utilizamos unos handlers extras para el control de excepciones que veremos posteriormente.

## Handlers

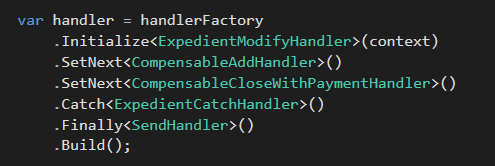
Cada handler debe implementar la clase AbstractHandler que nos aporta una serie de implementaciones, control de errores, escritura de logs y diversas ayudas para simplificar al máximo cada implementación específica de cada funcionalidad y unificar el comportamiento.

## Creación de la cadena

Para la creación de una nueva cadena de responsabilidades se utiliza la fábrica IHandlerFactory, la cual nos ofrece únicamente un método para inicializar la cadena indicándole el primer handler que se ejecutará de la cadena y pasándole el contexto. Adicionalmente, en la implementación de esta fábrica al inicializar la cadena se rellena el contexto con los datos que le falten.

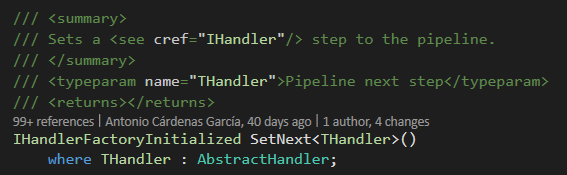
A partir de aquí se procede a componer la cadena utilizando los métodos SetNext, Catch, Finally o Build que veremos más adelante.

Por facilitar la lectura y simplificar la lectura del código, nos ofrece un estilo “Fluent interface”.



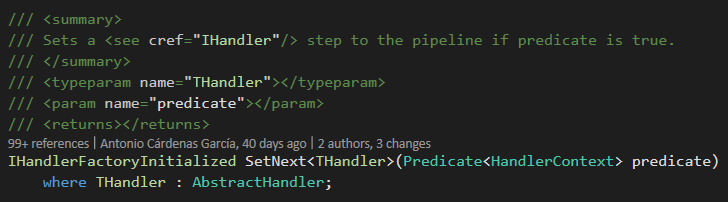
### SetNext

El método SetNext sirve para encadenar handler’s de forma ordenada, es el método principal que usaremos ya que lo utilizaremos tantas veces como handler’s queramos encadenar.

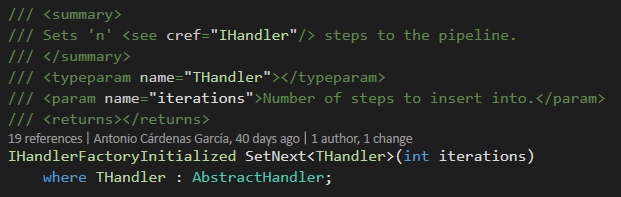


Además, tenemos 5 sobrecargas para este método que nos ayudan a adaptarnos a las diversas casuísticas que podríamos encontrarnos:

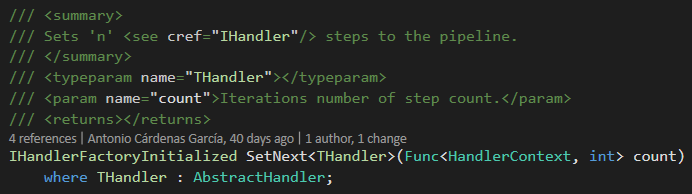
1. Pasándole un predicado.



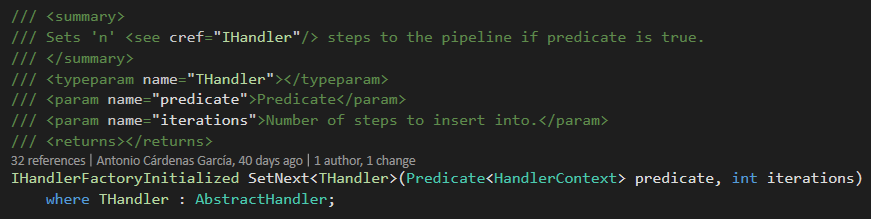
1. Pasándole un numero de iteraciones



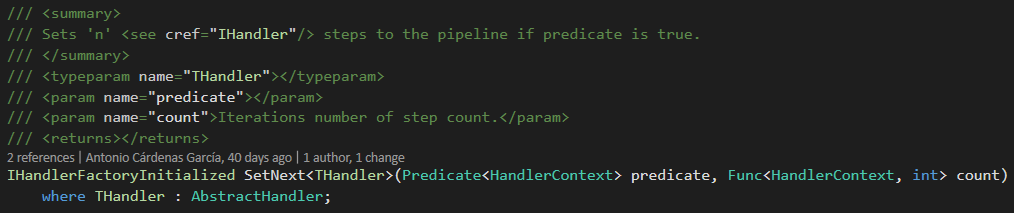
1. Pasándole una Func que definirá un número de iteraciones.



1. Pasándole un predicado y un número de iteraciones.

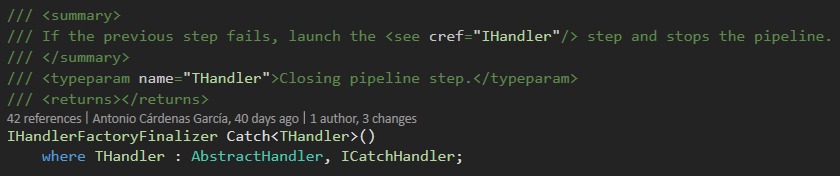


1. Pasándole un predicado y una Func que definirá un numero de iteraciones.



### Catch

El método catch se utiliza para que, en caso de producirse una excepción durante la ejecución de cualquier handler de la cadena, además de parar la ejecución de la misma, se ejecuta la implementación del handler indicado en el catch.



En el caso en el que no se produzca ninguna excepción durante la cadena, nunca se invocará al handler del catch.

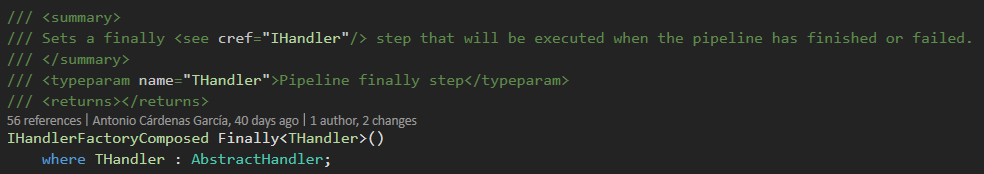
Este método es opcional para la creación de la cadena, podemos añadirlo o no en función de nuestras preferencias a la hora de crear la cadena.

El catch es global de la cadena y para asegurar la integridad solo se puede invocar el método catch una única vez por cadena y debe ser de los últimos pasos debido a que una vez encadenemos el catch no nos dejará encadenar más pasos utilizando el SetNext.

Después de llamar al método Catch solo podremos llamar a los métodos Finally o Build que veremos a continuación.

### Finally

El método finally encadena un handler que será el último en ejecutarse en cualquier situación. Tanto si la cadena se ejecuta correctamente como si hay alguna excepción que hace saltar el catch, el finally se ejecutará como último paso de la cadena.



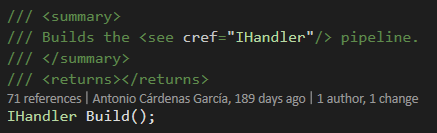
El finally al igual que el catch es global de la cadena y una vez invocado no se permite añadir ningún otro paso a la cadena ni tampoco añadir un catch a la misma.

Este método es opcional para la creación de la cadena, podemos añadirlo o no en función de nuestras preferencias a la hora de crear la cadena.

Una vez añadido el finally, solo podremos llamar al método Build que veremos a continuación.

### Build

El método build sirve para construir la cadena y que podamos lanzarla en el momento que queramos.



Una vez invocado el método build, ya no se puede modificar de ninguna manera la cadena.

Adicionalmente, la solución dispone de un PipelineManager que nos informa de los estados de las cadenas de responsabilidades y en el momento en el que se hace el build es cuando se registra la cadena en el PipelineManager para posteriormente conocer el estado de la misma.

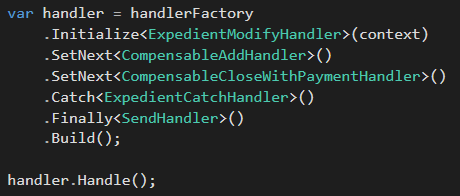
## Ejecución de la cadena

Para ejecutar la cadena, una vez la hemos creado añadiendo los SetNext con los pasos y opcionalmente agregando un Catch y Finally o no, después de invocar al método Build éste nos devolverá un IHandler que nos dará la posibilidad de llamar al método Handle que será el encargado de lanzar la cadena ejecutando cada uno de los pasos configurados previamente.

Ejemplo:

Se inicializa la cadena, se añaden dos handlers, se añade un catch, se añade un finally y se construye la cadena. Asignando la cadena construida a la variable “handler”.

Una vez construida, ejecutamos la cadena con el método Handle().



## PipelineManager

Para conocer el estado de las cadenas de responsabilidades que se han creado y las que se han lanzado tenemos una clase estática PipelineManager.