DISEÑO

* Tras analizar el problema presentando, directamente se evidencia que encaja exactamente con un problema donde podría ser posible aplicar el principio de “Chain of Responsability” para manejar las validaciones.
* Tras definir esto, cada una de las validaciones será un módulo o clase independiente, que únicamente tenga la responsabilidad de su validación
* También vemos que, con tantas validaciones secuenciales y reutilizables para otros clientes, sería recomendable aplicar el principio de “Open/Closed” para componer la cadena de validaciones en tiempo de ejecución
* Claramente vemos la clase “Client” como la principal y encargada de hacer las peticiones (request) al sistema y de recibir las respuestas (response). Además, esta sería la encargada de armar las cadenas de validaciones a utilizar
* Esta cadena de validaciones, que a futuro puede llegar a escalarse con validaciones nuevas, sería una clase abstracta “BaseHandler” que implementa la lógica común y que suministre un método que las validaciones implementen
* Inicialmente contaríamos con las siguientes validaciones:
  + Usuarios autenticados
  + Usuarios con permisos administrativos
  + Sanear datos
  + Direcciones IP
  + Manejo de caché

Tras definir todo esto, y revisando los parámetros que se necesitarían para algunas validaciones, se construye el siguiente diagrama UML:



En resumen, el cliente utiliza el manejador de validaciones “Handler” mediante las peticiones de “Request” y “Response”. El manejador tiene cada una de las validaciones existentes y futuras

IMPLEMENTACIÓN

Link repo: [JuanesGalvis/actividad\_problema\_diseno\_UdeM](https://github.com/JuanesGalvis/actividad_problema_diseno_UdeM)

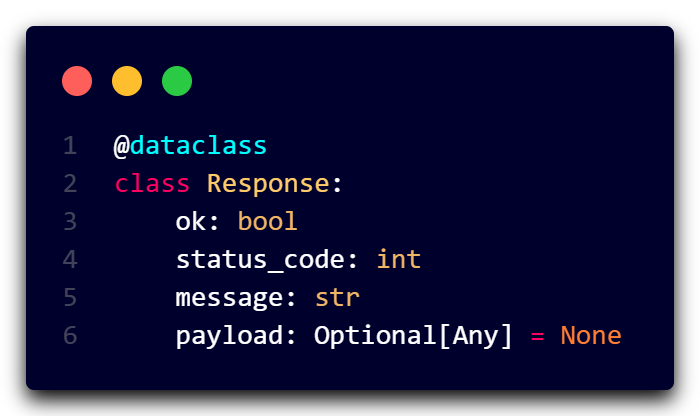
Tras hacer el diagrama UML, se procede a crear cada una de las clases:

Request:



* “ip” como parámetro para hacer la validación mencionada
* “data” como la información que consume el servicio
* “auth\_token” como token para validar que el usuario se encuentre autenticado

Response:



* “ok” para mandar como verdadero o falso si fue exitosa la petición
* “status\_code” código del resultado de la petición – estándar
* “message” mensaje o descripción del resultado
* “payload” información del resultado que arroja la petición

Handler:



Manejador de validaciones, definiendo los métodos de set\_next para la cadena de responsabildiades y handle como la validación

BaseHandler:



Base que utiliza lo definido en “Handler”, esto define el método de set\_next para que continue con la cadena e implementando la validación, en caso de ser exitosa continua con la cadena y en caso de que no devuelva el resultado para finalizar la cadena

VALIDACIONES:

AuthenticationHandler:



Ejemplo simple de una validación de autenticación donde primero revisa el token y el usuario almacenado en dicho token. En caso de que alguna no funcione devuelve un 401 y un “ok”: False que terminará la cadena de validaciones. En caso contraría guarda el usuario autenticado y devuelve como 200 y “ok”: True para que continue la cadena

ValidationHandler:



Validación y sanear la información, en caso de que a la información le falten campos se devolverá un 400 – “ok”: False. Pero si todo está correcto se devuelve 200 – “ok”: True para que continue con la cadena

RateLimitHandler:



Validación para la cantidad de peticiones por dirección IP, en este caso se realiza la validación de que no haya realizado más de 10 peticiones en 1 minuto (60 segundos). En caso de que sea así devolverá 429 – “ok”:False. Si no se almacena la petición para tenerla en cuenta en el conteo y se devuelve un 200 – “ok”: True y continuará la cadena

CacheHandler:



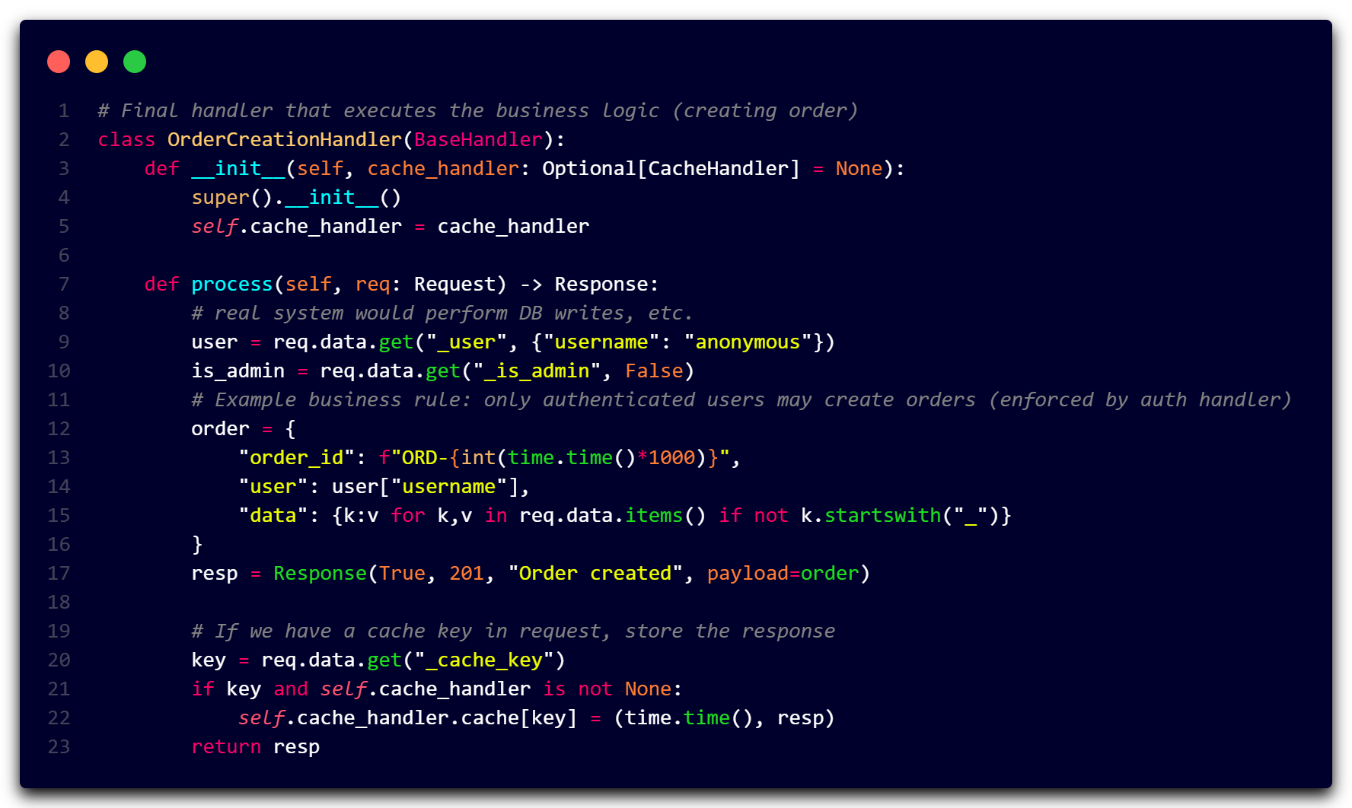
Este es un ejemplo corto y demostrativo de la validación de Caché, dado que simplemente utiliza “dict” en Python y una llave hash para identificarlo. Al pasar por la validación se revisa si ya existe o si se continua con la siguiente validación y se guarda en caché. En este caso no se responde de forma no exitosa, el “ok” siempre será “True” lo que cambia es el mensaje “Cached response” o “No cache - continue”

AuthorizationHandler:



Validación encarga de ver el permiso, está utiliza un parámetro “\_user” en el cuerpo de la petición para identificar si es un usuario autenticado. Posteriormente valida si es administrador, en caso de que si devuelve un 200 – True y en caso contrario devuelve un 403 – False

OrderCreationHandler:



Clase final donde si pasa todas las validaciones anteriores finalmente se aplica/almacena la petición como tal. En un sistema real se aplicaría la lógica de negocio, venta, almacenamiento en base de datos, etc. Devolviendo un 201 – True.

Client:



Cliente encargado de enviar las peticiones

PATRONES DE DISEÑO

**Patrones de diseño:**

* **Chain of Responsibility**: cada verificación es un handler. Permite encadenar verificaciones y detener el flujo si una falla. Facilita añadir/retirar handlers sin cambiar el cliente.
* **Builder / Fluent setup** (ligero): la set\_next(...) permite componer la cadena de forma fluida.
* **Strategy (parcial)**: la idea de intercambiar implementaciones concretas (por ejemplo distintos RateLimitHandler o distintas CacheHandler) es propia del patrón Strategy — los handlers actúan como estrategias de validación.
* **Template Method (implícito)**: BaseHandler.handle() define el flujo general (call process() y pasar al siguiente), y cada handler implementa su process() (variación del template method).

**Principios SOLID y buenas prácticas:**

* **Single Responsibility Principle (SRP)**: cada handler hace una sola verificación.
* **Open/Closed Principle (OCP)**: añadimos nuevos handlers sin modificar los existentes.
* **Dependency Inversion**: OrderCreationHandler recibe un CacheHandler (dependencia inyectada) — podríamos inyectar interfaces concretas.
* **Separation of Concerns**: validación, autenticación, caching y negocio están separados.
* **Testabilidad**: cada handler se puede probar de forma aislada.