

Watch:  **Large Class: Detección, problemas y estado ideal**

06:47

## Large Class: Detección, problemas y estado ideal

Aunque seguimos con el mismo ejemplo de los videos anteriores, vamos a introducir dos diferencias: Por un lado vamos a pasar verlo con Javascript (tenéis el acceso al ejemplo justo [aquí](#)), y por otro pasaremos a enfocarnos al code smell de **Large Class** 📄

Como indica su nombre, este code smell hace referencia a aquellas clases que son excesivamente largas. Si bien podríamos pensar que este está estrechamente derivado del Long Method, no siempre será así



Pongamos como ejemplo el típico modelo de User, el cual tiene una serie de campos 'base', puede ser que de repente necesitemos que para un determinado contexto (Bounded Contexts en DDD) este User además de esos campos tenga otros tantos más, por los que se los añadimos al modelo

Al final, con esto conseguiremos tener una clase User enorme por todo ese conjunto de campos que sólo necesitamos en un determinado contexto 🧑

Por otra parte, podríamos considerar llevar la gestión de estas diferencias a la capa de servicio y que fueran unos UserMoocManager y UserBackofficeManager se ocuparan de toda esa información particular de cada contexto, pero ni estaríamos hablando de una buena práctica (como veremos a continuación), ni estaríamos evitando que finalmente en BD siguiéramos teniendo una tabla User con un montón de columnas

Siguiendo el enfoque de DDD, si los campos de User necesarios en el contexto de Mooc son diferentes a los necesarios para el Backoffice, tiene bastante sentido separarlo en dos clases User diferentes con los datos que realmente requieren para su contexto

Esta separación nos ayuda no sólo a que este modelo User no sea una clase inmensa, sino también a que, a nivel de infraestructura, en BD contemos con tablas con menor número de columnas y que cualquier posible bloqueo sobre una de las tablas de usuarios no implique el bloqueo en el resto de contexto

Todo esto nos lleva a considerar detalles de diseño de la aplicación para definir a dónde debemos llevar la lógica. Tal como plantea el principio de **Tell, Don't Ask** o **Ley de Demeter**, deberíamos empujar toda la lógica de negocio lo más profundo posible, siempre que no dependa de ningún servicio externo. Con esto nos acercamos justamente a dos aspectos clave de la Programación Orientada a Objetos: Alta Cohesión  y Bajo Acoplamiento 

Así, si en la **capa de servicio** podemos evitar tener una clase larga a través de la **Responsabilidad Única** y acotada (Composición sobre

Herencia), en nuestros **modelos de dominio** tendremos que hacer uso de la **Segregación** en base al contexto

## ¿Alguna Duda?

Si tienes cualquier duda sobre el contenido del video o te apetece compartir cualquier sugerencia recuerda que puedes en abrir una nueva discusión justo aquí abajo 🙌🙌🙌

¡Nos vemos en el siguiente video: 🙌 Qué Refactoring NO hacer: Un gran poder conlleva una gran responsabilidad (única)!