

Watch: 🤔 **Muerte por deuda técnica: Cuándo refactorizar (Caso Evernote)**

14:25

## 🤔 Muerte por deuda técnica: Cuándo refactorizar

Los problemas derivados de acumular demasiada **deuda técnica** en nuestra aplicación no son algo que debieramos menospreciar, tal y como vemos en el ejemplo de la [Evernote](#), que se vio en la necesidad de parar durante 18 meses todos sus desarrollos sólo para poder rehacer la aplicación 🔄

El caso de Evernote se encuadra en un escenario bastante frecuente de entrega continua de valor, si en este ritmo de trabajo vamos empujando los problemas técnicos hacia un rincón oscuro en lugar de irlos

abordando, acabamos generando esa deuda técnica. Una de las soluciones a este problema es la que aquí adoptaron: Frenar los desarrollos de nuevas features y rehacer toda esa maraña desde o

Por supuesto, si se hubieran ido detectando a tiempo todos esos ‘code smells’ que conforman la maraña, los refactors hubieran sido tremendamente más sencillos y habría permitido establecer unas bases sólidas de clean code sobre las que trabajar 🧨

Pero ¿Cuándo refactorizar? ¿Es necesario un parón total para ponernos a refactorizar código? Por supuesto habrá fases tempranas en las que probablemente No sea necesario: Imaginad que aún se está validando simplemente la idea del producto a vender.

Por otro lado, aunque la aplicación ya sea un producto sólido y que genere beneficios, debemos seguir teniendo en consideración cuándo hacerlo 📅 y a qué alcance: El riesgo para este tipo de acciones en una empresa de venta online será mucho mayor en el periodo de Navidad que a finales de verano. Por supuesto, cuanto más deuda técnica acumulemos, más riesgo y mayor impacto tendrá cuando nos decidamos a pagarla





## La regla del boy scout

Al igual que los buenos scouts procuran dejar el campo un poco mejor de como lo encontraron, una buena práctica para evitar precisamente esta ‘muerte por deuda técnica’ es aprovechar cuando debemos modificar algún código para mejorarlo si detectamos pequeños code smells por el camino 👍

Con esto entramos en una toma de decisión bastante importante ¿mejoramos el código primero o lo hacemos después de hacer nuestros desarrollos? Optar por una u otra alternativa va a depender de distintos factores como:

- Mi conocimiento del dominio ¿Conozco bien el código afectado o espero a terminar mi desarrollo?

- Afectación sobre mi desarrollo ¿Facilitará mi tarea el hecho de refactorizar primero?




Evidentemente, es algo más que recomendado contar con una mínima cobertura de tests que nos brinden la tranquilidad de saber que el refactor aplicado no está rompiendo nada  y apostar si es posible por un rollout progresivo en lugar de tratar de moverlo todo de una sola vez 

Como apunte final 👁👁 queremos incidir en que rehusar de un código 'legacy' sin pararnos a identificar los problemas que presenta y las cosas que lo hacen difícil de mantener, muy probablemente nos llevará justamente a repetir los mismos problemas aunque nos lancemos a rehacer todo desde 0, eso es lo que hace tan importante y necesario aprender a identificar todos esos code smells y cómo solucionarlos

En las siguientes lecciones profundizaremos en cómo identificar y refactorizar todos esos code smells, ¡Vamos allá!

- [We broke up with the Monolith, and started dating Event Sourcing:](#)  
Slides de la charla que comentamos en el video

## ¿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 la siguiente lección:  Long Method Code Smell!