TIDY FIRST!, KENT BECK

Enol Rodríguez Hevia – UO287935
Alfredo Jirout Cid – UO288443
Carlos Cabrera Moral - UO288595
Grupo ES0103

Tabla de contenido

Introducción

En el episodio 615 de Software Engineering Radio, publicado el 10 de mayo de 2024, el anfitrión Giovanni Asproni conversa con Kent Beck, científico jefe en Mechanical Orchard e inventor de metodologías como Extreme Programming y Test-Driven Development. La discusión se centra en el concepto de "Tidy First?", título del libro más reciente de Beck. A lo largo del episodio, se exploran temas como la importancia de la limpieza del código antes de realizar cambios funcionales y cómo equilibrar decisiones de diseño y calidad del código con aspectos como costes y valor. Además, se abordan consideraciones sobre el impacto de la inteligencia artificial en el trabajo de los desarrolladores de software.

Entrevista

Crítica a la rigidez normativa

Kent critica ciertas recomendaciones de diseño de software que no están respaldadas por hechos reales y pueden llevar a conclusiones erróneas. Un ejemplo de esto es la regla que dice que las funciones deben tener entre 5 y 9 líneas de código. Según él, esta idea no refleja lo que realmente sucede en los proyectos de software.

En lugar de seguir reglas estrictas sin cuestionarlas, Kent sugiere que entendamos cómo evoluciona el código en la práctica. Explica que el desarrollo de software sigue ciertos patrones naturales, como las leyes de potencia, lo que significa que características como la cantidad de parámetros o el tamaño de las funciones tienden a crecer a medida que el sistema se hace más grande.

Cohesión y Acoplamiento

Kent explica que el acoplamiento es una relación que se da cuando un cambio en un elemento del código obliga a modificar otro. Por otro lado, la cohesión indica qué tan interdependientes son los elementos dentro de un mismo módulo, si al cambiar un elemento es necesario modificar otros dentro de la misma unidad, entonces esa unidad es altamente cohesionada. Mejorar la cohesión ayuda a contener el impacto del acoplamiento y a reducir la propagación de cambios en el código.

¿Qué es un Tidying?

Un tidying es una serie de pequeños ajustes en la estructura del código que, aunque individualmente puedan parecer insignificantes, en conjunto pueden generar un gran impacto. Es como un copo de nieve que, por sí solo, no cambia nada, pero que, sumado a muchos otros, puede desencadenar una avalancha de mejoras. Esto puede ir desde reorganizar funciones y mejorar nombres hasta reducir el acoplamiento. La idea central que nos presenta Kent es que antes de hacer un cambio en el comportamiento del código, se debe considerar si es necesario "ordenarlo" primero (Tidy First?), aunque la decisión no siempre es obvia. Algunos programadores lo hacen de forma automática, mientras que otros nunca lo hacen; Kent sugiere que la clave es encontrar un equilibrio.

Código limpio

La idea clave sobre un código limpio es que un código el cual estás modificando a menudo suele estar limpio por el hecho que se está refactorizando constantemente. Sin embargo, un código que lleva mucho tiempo sin tocarse tiene altas probabilidades de no estar limpio ya que hace tiempo que no es revisado.

Kent Beck insiste en la metáfora de que un software es más un jardín que una escultura perfecta, ningún software es perfecto. Para explicar esto Beck hace referencia al principio de Pareto.

Principio de Pareto

El principio de Pareto, también conocido como la regla 80/20, describe un fenómeno que establece que aproximadamente el 80 % de los resultados provienen del 20 % de las acciones. Esto aplicado a desarrollo software haría referencia que el 80% de los cambios que tengas que hacer los vas a tener que hacer sobre 20% del código.

Grandes refactorizaciones vs Refactorizaciones incrementales

El problema de las grandes refactorizaciones es la desconfianza que genera en las relaciones personales de los trabajadores ya que no saben los cambios realizados hasta que terminas la refactorización y puede que piensen que no has cambiado nada. Beck asegura que esto es un aspecto clave. Sin embargo, si se van realizando pequeños cambios de forma incremental los cambios son visibles cada poco sobre el código y los demás trabajadores pueden ganar confianza en que estás haciendo tus tareas asignadas.

Diferencias entre cambios estructurales y cambios de comportamiento

Los cambios estructurales suelen ser reversibles, como reorganizar funciones, mientras que los cambios de comportamiento pueden tener efectos irreversibles en el mundo real, como errores en pagos o reportes. A pesar de esto, también puede haber algunos cambios estructurales, como renombrar APIs o modificar URLs, que pueden ser difíciles de revertir. Beck sugiere minimizar el impacto con transiciones seguras, como mantener versiones paralelas antes de eliminar funciones antiguas.

El impacto de los pequeños cambios en la arquitectura y el diseño

Beck argumenta que el diseño es fractal, que significa que es la aplicación de los mismos principios de diseño en diferentes niveles de un sistema, desde las expresiones individuales en una función hasta la arquitectura general del sistema. Si se pueden realizar cambios estructurales con facilidad, la necesidad de diseño previo disminuye, ya que tomar decisiones con poca información puede no ser lo más conveniente.

¿Cuándo diseñar?

Diseñar primero, tiene una gran resistencia dado que el tiempo que se está diseñando es tiempo en el que no se está escribiendo ni una línea de código.

Muchas veces es más importante tener una aplicación lista lo antes posible que tener un buen diseño

El dinero temprano “es como tener más dinero”, con esa premisa nos damos cuenta de que lo más importante es tener una financiación antes que un buen diseño. Hoy en día los tipos de interés son elevados, “mucho más que hace 10 años” por lo que tener dinero en un principio nos genera un valor alto y diseñando antes y “perdiendo tiempo de programar” nos retrasaría la salida al mercado de la aplicación e igual no ganaríamos tanto dinero como si la sacásemos antes debido a estos tipos de interés.

Resalta la importancia del dinero, ya que si no hay dinero al principio es posible que la aplicación fracase.

¿Ordenar el código?
Debe ser lo primero, un código desordenado lleva más tiempo para ser cambiado además de que “hay que descifrarlo”. Al ordenarlo, mejora los cambios a futuro

El coste del software = coste de cambiar el software, el coste inicial del software es muy pequeño, los grandes cambios son los que se llevan la mayor parte del dinero.

Nuestro mayor enemigo es el acoplamiento, si tenemos un alto acoplamiento cualquier cambio que hagamos en la aplicación nos constara mucho dinero, quitarlo nos genera un coste de acoplamiento más el coste del desacoplamiento

¿Cambio del código?
A la hora de revisar el código, hay que diferenciar los cambios estructurales (fáciles de revisar y los cambios de comportamiento (aportar test que reflejen que funciona, deben de cubrir casos de uso especiales y deben dar suficiente confianza)

Es importante no caer en un “bucle de” cambio lógico, cambio de comportamiento, cambio lógico, de comportamiento ...

TDD
Explica que no incluye ordenamiento porque TDD es un flujo de trabajo específico y muchas críticas a TDD se basan en malentendidos o en la confusión con otros enfoques. Beck enfatiza que su intención no es imponer TDD, sino describir su experiencia y dejar que otros lo adopten si les resulta útil.

BIBLIOGRAFÍA

- https://se-radio.net/2024/05/se-radio-615-kent-beck-on-tidy-first/
- https://asana.com/es/resources/pareto-principle-80-20-rule