CLEAN CODE

SE Radio episode 577

15 DE MARZO DE 2024

Sandra García Morán UO283182
Sonia Moro Lauda UO282189
Lucía Villanueva Rodríguez UO283535

Introducción

En primer lugar, nos gustaría explicar porque estamos aquí y porque vamos a hablar del capítulo 577 del podcast Software Engeniering Radio.

Este capítulo se titula **Casey Muratori on Clean Code, Horrible Performance?** Este podcast comienza hablando de un popular video realizado por el entrevistado Casey Muratori, dicho video se llama: Clean Code, Horrible Performance y fue muy controvertido.

Contexto del video:

- El video forma parte de un curso más amplio sobre conciencia del rendimiento en el desarrollo de software.
- Casey presenta una visión general de los factores que afectan el rendimiento y cómo se relacionan con el código limpio.

Importancia del código limpio:

- Casey enfatiza la relevancia de escribir código limpio y fácil de entender.
- El código limpio mejora la mantenibilidad, facilita la colaboración entre desarrolladores y reduce la probabilidad de errores.

Ejemplo de "Clean Code":

- Casey analiza un ejemplo del libro "Clean Code" de Robert Martin.
- El ejemplo se centra en virtual functions (funciones virtuales) y cómo su uso puede afectar el rendimiento.
- Casey demuestra cómo seguir principios de código limpio puede llevar a ineficiencias en el rendimiento.

Equilibrio necesario:

- Aunque el código limpio es crucial, no debe ser a expensas del rendimiento.
- Casey argumenta que a veces es necesario romper los principios del código limpio para optimizar el rendimiento.
- En lugar de adherirse ciegamente a las reglas de código limpio, los desarrolladores deben considerar el contexto específico y encontrar un equilibrio adecuado.

Reacciones encontradas:

- El video generó debates y discusiones en la comunidad de desarrollo de software.
- Algunos estuvieron de acuerdo con los puntos de Casey, mientras que otros discreparon enérgicamente.
- La discusión sobre cómo equilibrar la calidad del código y el rendimiento es un tema apasionante y provocador.

¿Quién es Casey Muratori? Casey Muratori es un programador especializado en investigación y desarrollo de motores de juegos. Actualmente, está trabajando en 1935, un próximo juego narrativo interactivo sobre el crimen organizado en la Nueva York de la década de 1930. Casey escribe todo el código para este proyecto. Además, es el anfitrión de Handmade Hero, una serie educativa que enseña técnicas de programación de juegos a nivel bajo mediante ejemplos. Anteriormente, contribuyó a proyectos como The Granny Animation SDK, Bink 2 y The Witness. Su código ha sido utilizado en miles de juegos, incluyendo franquicias populares como Age of Empires, Ultima, Guild Wars y Gears of War. Casey también tiene intereses en ficción interactiva, computación concurrente y arquitectura de sistemas operativos

En resumen, Casey Muratori es un apasionado programador con una amplia experiencia en el mundo de los motores de juegos y la tecnología de animación.

¿Qué es clean code?

Clean Code se refiere a escribir código de manera que sea **fácil de entender, mantener y modificar**. Aunque no es un conjunto estricto de reglas, sigue una serie de **principios** que ayudan a producir código intuitivo y de alta calidad.

Clean Code es clasificado por algunos como una filosofía utilizada en el desarrollo de software cuyo principal objetivo facilitar la lectura y escritura de código.

Es especialmente útil cuando se trabaja en equipo o grupo, ya que muy probablemente otro miembro del grupo tendrá que modificar o consultar tu código.

Cabe destacar que escribir código limpio puede implicar tanto escribir más código de lo normal como escribir menos código, depende de las circunstancias en las que se dé.

Aquí están algunas características clave:

1. **Nombres significativos**:
   Utiliza nombres descriptivos para variables, funciones y clases. Evita letras genéricas como x, y, a o b y utilizar nombres más especificados.
2. **Evita palabras innecesarias**:
   Elimina palabras redundantes que no aportan información adicional. Por ejemplo, en lugar de accountList, usa simplemente accounts.
3. **Comentarios y claridad**:
   No uses comentarios para explicar por qué se usa una variable. Si un nombre necesita un comentario, es mejor renombrar la variable. Un buen nombre debe revelar su intención sin necesidad de comentarios.
4. **Estructura y consistencia**:
   Sigue convenciones y estándares para que el código sea uniforme. Mantén la estructura coherente y evita duplicación innecesaria.
5. **Legibilidad y simplicidad**:
   Escribe código conciso y expresivo. Prioriza la legibilidad sobre la brevedad.
   En resumen, Clean Code es aquel que es fácil de leer, entender y mantener, al tiempo que sigue siendo robusto y seguro para satisfacer las demandas de rendimiento, es independiente del desarrollador que lo creo.

Principios Clean Code

A continuación, voy a hablar sobre los principios del Clean Code, los cuales es fundamental tener en cuenta durante el desarrollo software.

-KISS (Keep It Simple, Stupid) (Hazlo simple, estúpido):

Este principio se refiere a que el código de un programa que debe ser lo más simple posible desde el principio, de manera que ayudará a reducir la complejidad. Por tanto, cuando se quiere resolver un problema, dado a que en la programación hay muchas maneras de abordarlo, se intentará buscar la más sencilla. El código simple tiene menos errores y es más fácil de modificar.

Los beneficios que proporciona son: Mayor calidad del código, código más simple de mantener, más flexible y fácil de ampliar, modificar y mejorar.

-DRY (Don't Repeat Yourself) (No te repitas a ti mismo):

Consiste principalmente en evitar la duplicación de código. Se quiere evitar que si tienes que hacer un cambio en un sitio, no haya que hacerlo en varios.

Este principio sería el contrario a WET, el cual es código redundante que incluye duplicaciones innecesarias.

Las ventajas que ofrece son: El código será mantenible, de manera que será más sencillo hacer cambios, añadir cosas nuevas, y realizar pruebas y fácil de entender.

La solución a este principio sería usando la refactorización del código, patrones de diseño para conseguir abstracción y variables compartidas, y también la reutilización de código.

-YAGNI (You Ain't Gonna Need It) (No lo vas a necesitar):

Este principio surge a partir de la Programación Extrema (XP) que consiste en la programación ágil, en adaptarse a los cambios de forma rápida. YAGNI consiste en que solo se debe implementar las funcionalidades que el software necesite. La implementación se realiza cuando se necesita, no hay que pensar en el futuro.

Los **beneficios** que da son:

- Ahorro de tiempo de implementación de funcionalidad que no va a ser útil.
- No se complica el código de manera innecesaria.
- Evitar el exceso de diseño (Over-Engineering): YAGNI se opone a crear soluciones complejas y detalladas para problemas que pueden no llegar a existir. En lugar de anticipar todas las posibles necesidades futuras, los desarrolladores se centran en implementar solo lo necesario en el momento presente.
- Facilita la adaptabilidad: el código tiende a ser más flexible y fácil de adaptar a cambios en los requisitos.
- Menos mantenimiento: Reducir la cantidad de código innecesario también significa que hay menos código para mantener.

Otros principios fundamentales que se deben de tener en cuenta a la hora de conseguir un Clean Code:

Nombres descriptivos:

El nombre de las variables, funciones, clases… deben tener un nombre que exprese su propósito. Esto ayudará a la legibilidad del código y será más fácil de entenderlo y describirá mejor su intención.

La longitud de los nombres de las variables será proporcional a su ámbito. Si es un ámbito pequeño será un nombre más corto, si es grande será más largo.

Pero las funciones son al contrario, si su ámbito es pequeño, es decir, será una función que haga algo muy específico, deberán tener un nombre largo, y si es grande deberá tener un nombre corto (una palabra) ya que hace algo general. Las clases usan la misma regla que las funciones.

Funciones:

Deben ser pequeñas. Se recomienda que ocupen entre 10 y 20 líneas. De manera que será fácil de entender y expresará una intención clara.

Deben cumplir el principio de responsabilidad única: Una función realiza una sola tarea. En caso de que realice varias cosas, habría que extraer en varios métodos.

Se recomienda usar pocos argumentos (no más de 3) para que sea fácil de entender, y que cada argumento de los métodos sean de una sola palabra, que tenga un nombre significativo.

Se deberían evitar los efectos secundarios para evitar cambios inesperados en el sistema.

Por último, esta son otras técnicas y consejos que podemos llevar a cabo para tener un Clean Code:

Regla Boyscout: Esta regla dice “Deja el campamento más limpio de lo que lo encontró”, y en la programación sería dejar el código mejor/más limpio de cómo te lo encontraste, si encuentras algo que podría estar mejor, lo cambias. Se basa en hacer pequeñas mejoras continuas del código, en vez de hacer refactorizaciones grandes de una vez.

Comentarios y documentación para explicar el código, añadir aclaraciones, ayudar a entenderlo mejor, pero tampoco ser redundante.

Hacer tests: Pruebas que sean rápidas e independientes, que se puedan ejecutar en cualquier entorno, que devuelvan respuestas booleanas. Utilizar el Desarrollo Guiado por Pruebas (TDD).

Usar un **estilo de indentación** para ayudar a la legibilidad.

Manejo de errores: Lanzando excepciones, utilizar try-catch.

Rendimiento

En el podcast se aborda el rendimiento, que es crucial ya que incide directamente en la eficiencia y experiencia del usuario final. En este sentido, se discute el impacto de las decisiones de diseño en el rendimiento, la importancia del conocimiento sobre rendimiento, el monitoreo y la optimización de este, así como el papel fundamental que juega la arquitectura en este aspecto.

1.Impacto de las decisiones de diseño en el rendimiento:

En sus análisis, Casey Muratori resalta cómo las decisiones de diseño pueden influir de manera directa en el rendimiento del software. Por ejemplo, en el podcast se menciona cómo la separación artificial de funcionalidades, como en el caso de los microservicios, puede ralentizar el rendimiento si las funciones se superponen o si se agregan capas innecesarias de abstracción.

Muratori argumenta que las decisiones de diseño que priorizan la legibilidad y claridad pueden tener consecuencias negativas en el rendimiento, especialmente si no se consideran adecuadamente las implicaciones de estas en términos de rendimiento. Esto está alineado con su enfoque en la optimización del código y su convicción de que el rendimiento es un aspecto fundamental del desarrollo de software que no debe descuidarse.

2.Importancia del conocimiento sobre rendimiento:

Casey Muratori enfatiza la importancia de comprender las implicaciones de rendimiento al elegir lenguajes de programación y diseñar la arquitectura de software. Destaca que la falta de conocimiento sobre los costos de rendimiento puede conducir a decisiones poco informadas y a problemas de rendimiento en el futuro. Además, subraya que tener conciencia sobre el impacto en el rendimiento de ciertas decisiones permite a los desarrolladores tomar medidas proactivas para optimizar el código y evitar problemas de rendimiento antes de que se conviertan en obstáculos significativos para el funcionamiento del software.

3.Monitoreo y optimización del rendimiento:

En su enfoque de desarrollo, Casey Muratori hace hincapié en el monitoreo constante del rendimiento del código y en el mantenimiento de métricas razonables para identificar posibles cuellos de botella y áreas de mejora. Este enfoque se alinea con su práctica de revisar regularmente el tamaño de las líneas de código y mantener métricas de rendimiento para facilitar futuras optimizaciones. También destaca la importancia de centrarse en áreas críticas del código donde se identifiquen problemas de rendimiento y en la optimización proactiva de algoritmos ineficientes y consultas de bases de datos, reflejando así su enfoque pragmático de la optimización del rendimiento para garantizar un funcionamiento eficiente del software.

4.Rol de la arquitectura en el rendimiento:

Casey Muratori discute cómo la arquitectura del software juega un papel crucial en el rendimiento del sistema en su conjunto. Por ejemplo, en su análisis sobre los microservicios, señala cómo una arquitectura mal diseñada puede afectar negativamente el rendimiento al introducir complejidad adicional en la comunicación entre componentes. Aboga por un

enfoque de diseño de arquitectura que fomente naturalmente la optimización en áreas críticas del código, lo cual implica una comprensión profunda de los requisitos de rendimiento del sistema y la planificación para escenarios de carga máxima y escalabilidad futura, siendo esto fundamental para su enfoque de arquitectura consciente del rendimiento.

Sin embargo, ¿qué tiene que ver todo esto con el clean code y por qué se menciona en el podcast?

El video que se comenta al inicio es un video donde Casey Muratori analiza un ejemplo del libro "Clean Code" e ilustra cómo las decisiones de diseño enfocadas en la legibilidad y claridad pueden tener repercusiones negativas en el rendimiento del software. Muratori muestra cómo la aplicación de principios de "código limpio" puede llevar a la introducción de abstracciones innecesarias o a la separación artificial de funcionalidades, lo que puede resultar en un código más lento y menos eficiente.

La premisa del video es controvertida porque desafía la noción convencional de que el código limpio siempre conduce a un mejor rendimiento. Muratori argumenta que si bien la legibilidad y la claridad son importantes, también es crucial considerar las implicaciones de rendimiento al tomar decisiones de diseño. En este sentido, sostiene que las decisiones de diseño deben equilibrar la legibilidad con el rendimiento, y que priorizar exclusivamente uno sobre el otro puede llevar a compromisos no deseados.