Técnico en

DESARROLLO DE SOFTWARE

Diseño e Implementación del Software

Instituto Von Neumann

Unidad IV

Diseño e Implementación del Software

<Deuda técnica>

Todo el mundo hace todo lo posible para escribir un código excelente desde cero. Probablemente no haya un programador que intencionalmente escriba código sucio en detrimento del proyecto. Pero ¿en qué momento el código limpio se vuelve impuro?

Ward Cunningham sugirió originalmente la metáfora de la "deuda técnica" con respecto al código sucio.

Si obtiene un préstamo de un banco, esto le permite realizar compras más rápido. Usted paga más por acelerar el proceso: no solo paga el capital, sino también el interés adicional del préstamo. No hace falta decir que incluso

puede acumular tantos intereses que la cantidad de intereses supere sus ingresos totales, lo que imposibilita el pago total.

Lo mismo puede pasar con el código. Puede acelerar temporalmente sin escribir pruebas para nuevas funciones, pero esto ralentizará gradualmente su progreso todos los días hasta que finalmente pague la deuda escribiendo pruebas.

Causas de la deuda técnica

Presión empresarial

En ocasiones la presión empresarial crea circunstancias comerciales que pueden obligar a implementar funcionalidades sin que estén completamente terminadas. En este caso, se realizarán correcciones o chapuzas en el código para ocultar las partes inacabadas del proyecto.

Falta de comprensión de las consecuencias de la deuda técnica

En ocasiones, es posible que su empleador no entienda que la deuda técnica tiene "interés" en la medida en que ralentiza el ritmo de desarrollo a medida que se acumula la deuda. Esto puede hacer que sea demasiado difícil dedicar el tiempo del equipo a la refactorización porque la gerencia no ve el valor de ello. No combatir la estricta coherencia de los componentes.

Aquí es cuando el proyecto se asemeja a un monolito en lugar del producto de módulos individuales. En este caso, cualquier cambio en una parte del proyecto afectará a otras. El desarrollo del equipo se hace más difícil porque es difícil aislar el trabajo de los miembros individuales.

Falta de pruebas

La falta de comentarios inmediatos fomenta soluciones o chapucerías rápidas pero arriesgadas. En el peor de los casos, estos cambios se implementan y

se implementan directamente en la producción sin ninguna prueba previa. Las consecuencias pueden ser catastróficas. Por ejemplo, una revisión de apariencia inocente podría enviar un correo electrónico de prueba extraño a miles de clientes o, lo que es peor, vaciar o corromper una base de datos completa.

Falta de documentación

Esto ralentiza la incorporación de nuevas personas al proyecto y puede detener el desarrollo si las personas clave abandonan el proyecto.

Falta de interacción entre los miembros del equipo

Si la base de conocimientos no se distribuye por toda la empresa, las personas terminarán trabajando con una comprensión obsoleta de los procesos y la información sobre el proyecto. Esta situación puede verse exacerbada cuando sus mentores capacitan incorrectamente a los desarrolladores junior.

Desarrollo simultáneo a largo plazo en varias ramas

Esto puede conducir a la acumulación de deuda técnica, que luego aumenta cuando se fusionan los cambios. Cuantos más cambios se hagan de forma aislada, mayor será la deuda técnica total.

Refactorización retrasada

Los requisitos del proyecto cambian constantemente y en algún momento puede que las funcionalidades ya diseñadas y desarrolladas no se adapten a estos nuevos retos y exista la posibilidad de volver a escribir dicho código y volver a diseñarlo.

Hay que tomar en cuenta que cuanto más se retrase la refactorización, más código dependiente tendrá que volver a trabajarse en el futuro.

Falta de seguimiento del cumplimiento

Esto se origina prácticamente cuando cada uno escribe el código como quiere sin seguir ningún lineamiento o simplemente copia fragmentos de código de proyectos anteriores.

Incompetencia

Esto es simplemente cuando el desarrollador simplemente no cuenta con las competencias necesarias para escribir un código decente.

Refactorización

El desarrollar software es muy parecido al escribir un libro, el libro tendrá un éxito enorme si el autor logra plasmar todas las ideas que quiere representar de la forma más clara posible o también podrá obtener un fracaso rotundo si no llega a darse a entender en las páginas del libro, que hace el autor en tal caso que no se entienda su contenido, será que escribe un nuevo libro creo que no, el autor realiza las correcciones dentro del mismo texto para hacerlo más entendible.

El ejemplo anterior aplicado al software en que la refactorización pueden ser esas mejoras necesarias para hacer más comprensible el código sin tener que escribir de nuevo el sistema ya desarrollado.

¿Cuándo refactorizar?

Regla de tres

1. Cuando estés haciendo algo por primera vez, simplemente hazlo.
2. Cuando esté haciendo algo similar por segunda vez, sienta vergüenza de tener que repetirlo, pero haga lo mismo de todos modos.
3. Cuando esté haciendo algo por tercera vez, comience a refactorizar.

Al agregar una característica

La refactorización lo ayuda a comprender el código de otras personas. Si tiene que lidiar con el código sucio de otra persona, intente refactorizarlo primero. El código limpio es mucho más fácil de comprender. Lo mejorará no solo para usted sino también para quienes lo usen después de usted.

La refactorización facilita la adición de nuevas funciones. Es mucho más fácil hacer cambios en código limpio.

Al corregir un error

Los errores en el código se comportan como los de la vida real: viven en los lugares más oscuros y sucios del código. Limpie su código y los errores prácticamente se descubrirán solos.

Los gerentes aprecian la refactorización proactiva, ya que elimina la necesidad de realizar tareas de refactorización especiales más adelante. ¡Jefes felices hacen programadores felices!.

Durante una revisión de código

La revisión del código puede ser la última oportunidad de arreglar el código antes de que esté disponible para el público.

Es mejor realizar tales revisiones en pareja con un autor. De esta manera, podría solucionar problemas simples rápidamente y medir el tiempo para solucionar los más difíciles.

¿Cómo refactorizar?

Lista de verificación de la refactorización realizada correctamente

El código debería volverse más limpio, si el código sigue igual de sucio después de la refactorización... bueno, lo siento, pero acaba de perder una hora de su vida. Trate de averiguar por qué sucedió esto.

Ocurre con frecuencia cuando te alejas de la refactorización con pequeños cambios y mezclas un montón de refactorizaciones en un gran cambio. Así que es muy fácil perder la cabeza, especialmente si tienes un límite de tiempo.

Pero también puede suceder cuando se trabaja con un código extremadamente descuidado. Independientemente de lo que mejore, el código en su conjunto sigue siendo un desastre.

En este caso, vale la pena pensar en reescribir completamente partes del código. Pero antes de eso, deberías tener pruebas escritas y reservar una buena cantidad de tiempo. De lo contrario, terminará con los tipos de resultados de los que hablamos en el primer párrafo.

La nueva funcionalidad no debe crearse durante la refactorización

No mezcle la refactorización y el desarrollo directo de nuevas funciones. Intente separar estos procesos al menos dentro de los límites de las confirmaciones individuales.

Todas las pruebas existentes deben pasar después de la refactorización

Hay dos casos en los que las pruebas pueden fallar después de la refactorización:

- Cometiste un error durante la refactorización. Esta es una obviedad: siga adelante y corrija el error.
- Tus pruebas fueron de muy bajo nivel. Por ejemplo, estaba probando métodos privados de clases.

En este caso, las pruebas tienen la culpa. Puede refactorizar las pruebas en sí o escribir un conjunto completamente nuevo de pruebas de nivel superior. Una excelente manera de evitar este tipo de situación es escribir pruebas de estilo BDD.

Técnicas de refactorización

Métodos de composición

Gran parte de la refactorización se dedica a componer correctamente los métodos. En la mayoría de los casos, los métodos excesivamente largos son la raíz de todos los males. Los caprichos del código dentro de estos métodos ocultan la lógica de ejecución y hacen que el método sea extremadamente difícil de entender e incluso más difícil de cambiar.

Las técnicas de refactorización de este grupo agilizan los métodos, eliminan la duplicación de código y allanan el camino para futuras mejoras.

- Método de extracción.
- Método en línea.
- Extraer variable.
- Temperatura en línea.
- Reemplazar temperatura con consulta.
- Dividir variable temporal.
- Eliminar asignaciones a parámetros.
- Reemplazar método con objeto de método.
- Algoritmo de sustitución.

Mover funciones entre objetos

Incluso si ha distribuido la funcionalidad entre diferentes clases de una manera menos que perfecta, todavía hay esperanza.

Estas técnicas de refactorización muestran cómo mover de manera segura la funcionalidad entre clases, crear nuevas clases y ocultar los detalles de implementación del acceso público.

- Método de movimiento
- Mover campo
- Extraer clase
- Clase en línea
- Ocultar delegado
- Eliminar intermediario
- Introducir método extranjero
- Introducir Extensión Local

Organización de datos

Estas técnicas de refactorización ayudan con el manejo de datos, reemplazando primitivas con funcionalidad de clase rica. Otro resultado importante es el desenredado de las asociaciones de clases, lo que hace que las clases sean más portátiles y reutilizables.

- Cambiar valor a referencia
- Cambiar referencia a valor
- Datos observados duplicados
- Campo autoencapsulado
- Reemplazar valor de datos con objeto
- Reemplazar matriz con objeto
- Cambiar asociación unidireccional a bidireccional
- Cambiar asociación bidireccional a unidireccional
- Encapsular campo
- Colección encapsulada
- Reemplazar número mágico con constante simbólica
- Reemplazar código de tipo con clase
- Reemplazar código de tipo con subclases
- Reemplazar código de tipo con estado/estrategia
- Reemplazar subclase con campos

Simplificar expresiones condicionales

Los condicionales tienden a volverse cada vez más complicados en su lógica con el tiempo, y también existen más técnicas para combatir esto.

- Consolidar expresión condicional
- Consolidar fragmentos condicionales duplicados
- Descomponer condicional
- Reemplazar condicional con polimorfismo
- Quitar indicador de control
- Reemplazar condicional anidado con cláusulas de protección
- Introducir objeto nulo
- Introducir aserción

Simplificación de las llamadas a métodos

Estas técnicas hacen que las llamadas a métodos sean más simples y fáciles de entender. Esto, a su vez, simplifica las interfaces para la interacción entre clases.

- Agregar parámetro
- Eliminar parámetro
- Método de cambio de nombre
- Consulta separada del modificador
- Método de parametrización
- Introducir objeto de parámetro
- Conservar todo el objeto
- Eliminar método de configuración
- Reemplazar parámetro con métodos explícitos
- Reemplazar parámetro con llamada de método
- Ocultar método
- Reemplazar constructor con método de fábrica

- Reemplazar código de error con excepción
- Reemplazar excepción con prueba

Lidiando con la Generalización

La abstracción tiene su propio grupo de técnicas de refactorización, principalmente asociadas con el movimiento de la funcionalidad a lo largo de la jerarquía de herencia de clases, la creación de nuevas clases e interfaces y la sustitución de herencia por delegación y viceversa.

- Tire hacia arriba del campo
- Método de extracción
- Cuerpo del constructor de dominadas
- Empuje hacia abajo el campo
- Método de empujar hacia abajo
- Extraer subclase
- Extraer superclase
- Extraer interfaz
- Contraer jerarquía
- Método de plantilla de formulario
- Reemplazar herencia con delegación
- Reemplazar delegación con herencia

Code Smells

Algunos de los code smells que podemos encontrar son:

Bloaters

Los bloaters son código, métodos y clases que han aumentado a proporciones tan gigantescas que es difícil trabajar con ellos. Por lo general, estos no surgen de inmediato, sino que se acumulan con el tiempo a medida

que el programa evoluciona (y especialmente cuando nadie hace un esfuerzo por erradicarlos).

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Abusar de la programación orientada a objetos

Todos estos olores son aplicaciones incompletas o incorrectas de los principios de programación orientada a objetos.

- Clases alternativas con diferentes interfaces
- Legado rechazado
- Cambiar declaraciones
- Campo Temporal

Prevenir cambios

Estos olores significan que, si necesita cambiar algo en un lugar de su código, también debe realizar muchos cambios en otros lugares. Como resultado, el desarrollo del programa se vuelve mucho más complicado y costoso.

- Cambio divergente
- Jerarquías de herencia paralelas
- Cirugía de escopeta

Prescindibles

Un prescindible es algo sin sentido e innecesario cuya ausencia haría que el código fuera más limpio, más eficiente y fácil de entender.

- Comentarios.
- Código duplicado.
- Clase de datos.
- Código muerto.
- Clase perezosa.
- Generalidad especulativa.

Acopladores

Todos los olores de este grupo contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación excesiva.

- Característica Envidia.
- Intimidad inapropiada.
- Clase de biblioteca incompleta.
- Cadenas de mensajes.
- Hombre en el medio.

Referencias de contenido

- ***https://www.atlassian.com/git/tutorials/why-git***
- ***https://www.atlassian.com/git/tutorials/what-is-git***
- ***http://www.lnds.net/blog/2010/07/control-de-versiones-distribuido.html***
- ***https://git-scm.com/book/es/v1/Empezando-Acerca-del-control-de-versiones***
- ***https://azure.microsoft.com/es-es/overview/what-is-a-virtual-machine/***
- ***https://www.1and1.es/digitalguide/servidores/know-how/desarrollo-web-con-stacks-de-software/***
- ***http://www.pmoinformatica.com/2012/09/ambientes-de-desarrollo-de-software.html***