



## Parte IV – Refactoring y diseño simple

Tema 6 - Refactoring

1



### La regla de tres (Don Roberts)

- La primera vez que uno hace algo lo resuelve directamente
- La segunda vez que uno hace algo similar, intenta no duplicar el trabajo, pero al final acaba repitiendo el trabajo de todos modos.
- La tercera vez que uno hace algo similar, reorganiza

2

2

## ¿Qué es Refactoring?

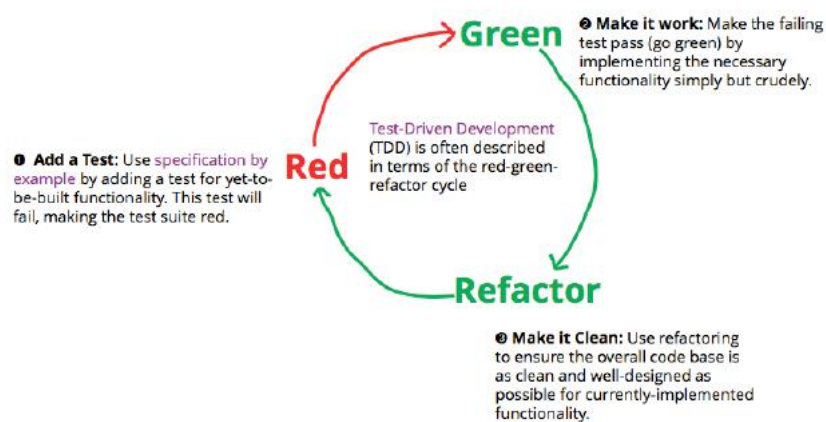
- Refactoring es una técnica para cambiar la apariencia interna y la organización del código sin alterar el comportamiento externo del mismo
- La reorganización persigue:
  - Mantener el código modular
  - Mantener el código fácil de modificar, haciéndolo legible
  - Eliminar bloques de código duplicados
  - Optimizar el rendimiento del código
  - Mantener el código organizado en regiones
  - Mantener actualizados los comentarios del código

Principios de Desarrollo de Software

3

3

## Proceso de Refactoring



4

## Solamente se puede llevar un sombrero en cada momento

- El sombrero de Refactoring es un sombrero de ala
- El sombrero de añadir nueva funcionalidad es un casco de trabajo para protegerse



Refactoring

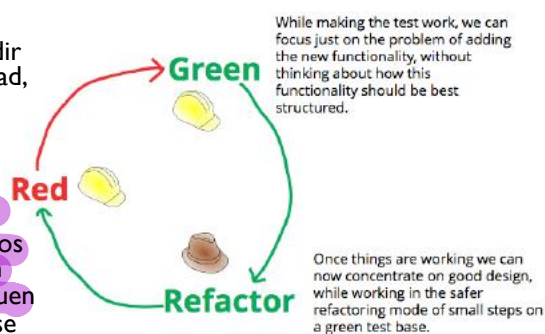


Adding Function

5

## Separar los sombreros te ayuda a mantenerte enfocado


- Mientras se hace el trabajo de prueba podemos focalizarnos en el problema de añadir nueva funcionalidad, sin pensar sobre cómo debería estructurarse la misma
- Una vez las cosas funcionan podemos concentrarnos en disponer de un buen diseño, mientras se trabaja en el modo de refactorización más seguro



Flujo de trabajo del refactoring en TDD

6

Regla del Boy Scout: Deja el lugar  
más limpio de lo que te encontraste




**Litter-Pickup  
Refactoring**

7

**Refactoring Comprensivo**

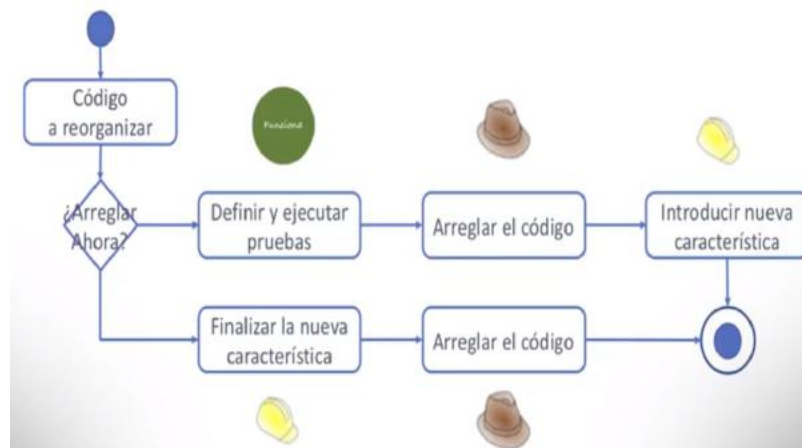
Tengo que introducir una nueva función...



**Comprehension  
Refactoring**

8

## Refactoring Oportunista



9

## Refactoring Preparatorio

- A menudo, comienzas a trabajar en la adición de nuevas funcionalidades y te das cuenta de que las estructuras existentes no son totalmente útiles con lo que vas a hacer.
- Al hacer este cambio con su sombrero de refactorización, puede hacer que la nueva funcionalidad sea mucho más fácil de introducir.

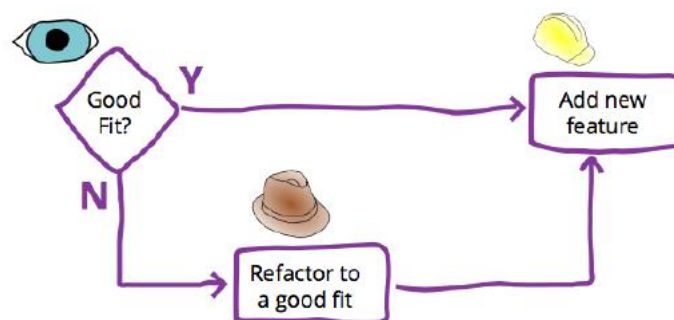
We should have done it this way

Preparatory Refactoring

10

## Refactoring Preparatorio

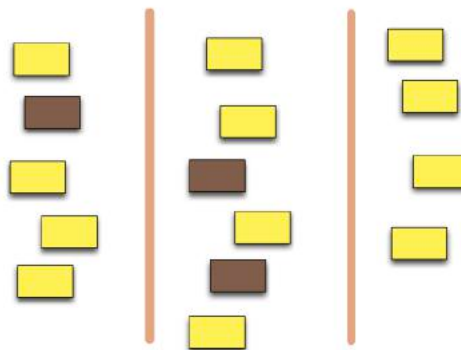
- Se puede hacer justo antes de añadir una nueva característica



11

## Refactoring Planificado

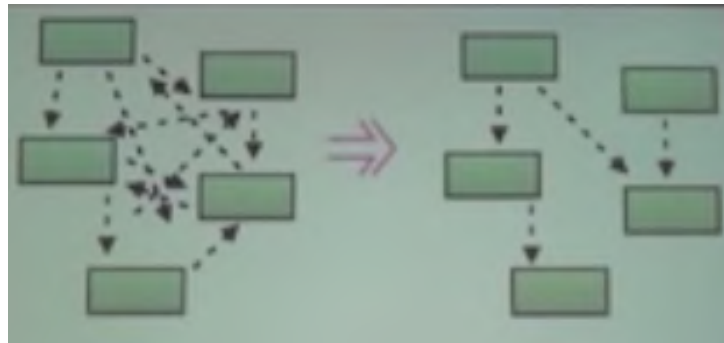
- Lo que no encaja hay que quitarlo, depurarlo...



12

## Refactoring a largo plazo

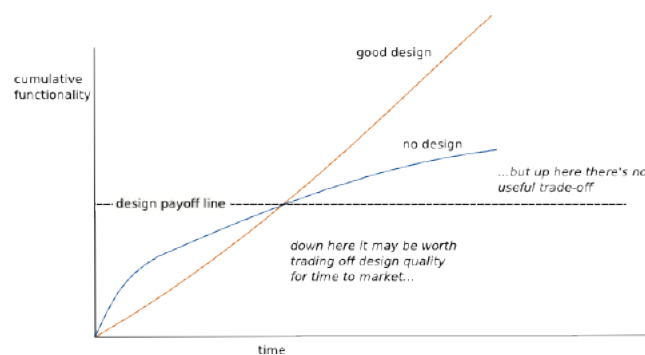
- Realmente necesitamos resolver estas dependencias, pero quizá lleve un par de meses



13

## ¿Es la refactorización un despilfarro inútil?

- ¿Dónde está el equilibrio?
  - Sin un buen diseño al principio avanzas rápido pero después el trabajo se ralentiza
  - Con un buen diseño la relación entre el tiempo empleado y la funcionalidad acumulada se mantiene constante

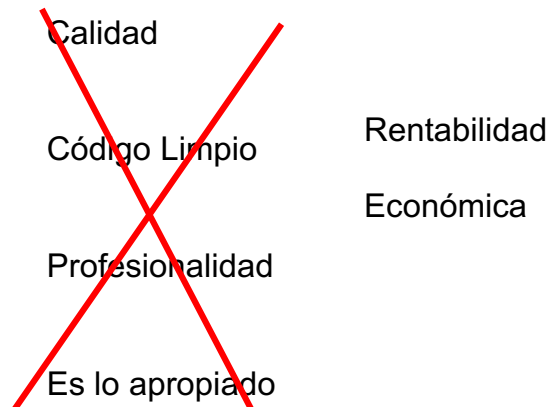


Trade-off Stamina Hypothesis

14



## ¿Por qué aplicar refactoring?



15

## Pasos para aplicar Refactoring

- Identificar el lugar en donde el refactoring tiene que ser aplicado.
  - Código difícil de entender
  - Al incorporar nuevas necesidades en el sistema
  - Mejorar la eficiencia del sistema
  - Situaciones predefinidas de reorganización de código
- Si existe una prueba para el código que se está considerando, es necesario ejecutarlas
- Si no, escribir todas las pruebas unitarias necesarias y conseguir que funcionen
- Localice el tipo de reorganización de código que deba ser aplicado, mediante el análisis del código o buscando en el catálogo.
- Ejecutar las pruebas entre cada paso para asegurarse de que el comportamiento no ha cambiado
- En caso de necesidad adapte el código de prueba a los interfaces que han cambiado
- Cuando la reorganización de código finaliza con éxito, ejecute las pruebas otra vez, integre y ejecute las pruebas de unidad completas y las pruebas funcionales o de aceptación.

16



## ¿Cuándo es necesario reorganizar el código?

- Código duplicado
  - Principal razón para reorganizar
- Métodos muy largos
  - Heredado de la programación procedimental
- Clases muy grandes
  - Clases con muchas responsabilidades
- Lista de parámetros muy grandes
  - No son necesarias cuando se trabaja con objetos
- Cambios divergentes
  - Una clase se cambia comúnmente de diversas maneras por diversas razones
- Cirugía de la escopeta (matar moscas a cañonazos)
  - Los cambios influyen en muchas otras clases y métodos

Principios de Desarrollo de Software

17

17

## ¿Cuándo es necesario reorganizar el código?

- Data Clumps
  - Los datos que se utilizan conjuntamente en diversas partes del código deberían formar una clase
- Obsesión por los tipos de datos básicos
  - utilice las clases además, o en vez de tipos de datos básicos
- Sentencias de tipo CASE
  - la orientación del objeto tiene otras maneras de ocuparse de acciones dependiendo de tipos
- Jerarquías de herencia paralelas
  - a veces útil pero a menudo innecesario
- Clases perezosas
  - Una clase que no se ejecuta a menudo debe ser eliminada
- Generalidad especulativa
  - no invierta mucho en la flexibilidad para el futuro
- Cadenas de mensajes

Principios de Desarrollo de Software

18

18

## ¿Cuándo es necesario reorganizar el código?

- El intermediario
  - Eliminarlo si la delegación es todo lo que realiza
- Intimidad inapropiada
  - restricción del conocimiento de los internals de otras clases
- Librerías de clases incompletas
  - Normalmente se deben extender para agregar la funcionalidad deseada
- Clases de datos
  - Deben adoptar tareas adicionales para manejar su información
- Peticiones rechazadas
  - si las subclases utilizan solamente muy poco de la proporcionada por sus padres
- Comentarios
  - un comentario es un buen lugar para decir porqué usted hizo o no una función o sección de código

Principios de Desarrollo de Software

19

19

## Beneficios de Refactoring

- Los programas que son difíciles de leer son difíciles de modificar
- Los programas que tienen lógica duplicada son difíciles de modificar
- Los programas que requieren cambiar el código cuando se necesita un comportamiento adicional son difíciles de modificar
- Los programas con lógica condicional compleja son difíciles de modificar

Principios de Desarrollo de Software

20

20

## El Refactoring **no siempre es beneficioso**

- Cuando uno aprende una nueva técnica que mejora en gran medida su productividad, es difícil entender porqué no se aplica. Normalmente se aprende en un contexto, incluso en el ámbito de un único proyecto, sin embargo no se aprecia que en otras circunstancias la técnica no es provechosa
- **No tener tiempo suficiente** es generalmente una **muestra que se necesita** hacer cierto **Refactoring**.

Principios de Desarrollo de Software

21

21

## Conclusión

*Refactoring →  
Clean Code →  
Faster Delivery*

22