



Entornos de desarrollo (ED)

Sergio Badal Raúl Palao

Extraído de los apuntes de: Cristina Álvarez Villanueva; Fco. Javier Valero Garzón; M.ª Carmen Safont



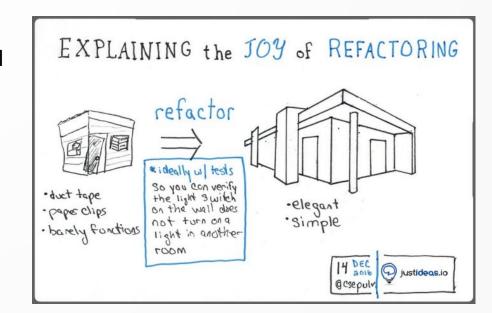
4 REFACTORIZACIÓN

- 4.2 CUÁNDO REFACTORIZAR
- 4.3 TIPOS DE REFACTORIZACIÓN
- **4.4 CONSEJOS Y PATRONES**
- 4.5 COHESIÓN Y ACOPLAMIENTO
- 4.6 IMPLEMENTACIÓN



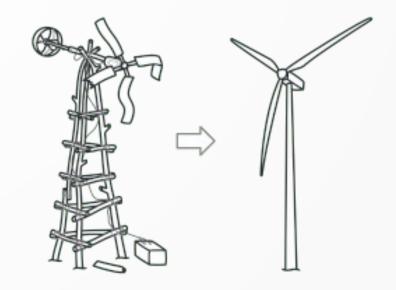


- Programar código no únicamente es buscar funcionalidad, sino también limpieza y elegancia de escritura.
- Así, aunque debería irse programando ya con una estructura inicial y buscando la limpieza de código, una vez terminado siempre se pasa a la fase de Refactorización.
- En ella le damos "chapa y pintura" a nuestro código.
 - Es decir, cambiamos la estructura interna de nuestro software para hacerlo más fácil de comprender, de modificar, más limpio y sin cambiar su comportamiento observable.
- Por tanto, refactorizar es:
 - Mejorar el código fuente sin cambiar el resultado del programa.



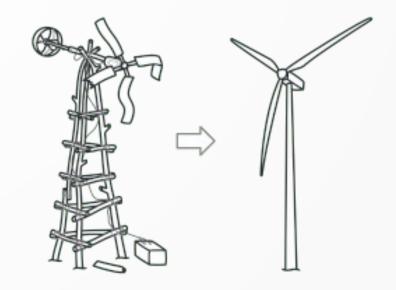


- Refactorizamos por varios motivos:
 - Para eliminar código duplicado
 - Con una visión global del código.
 - Para hacerlo más legible/fácil de entender
 - Renombrando identificadores, reorganizando parámetros, métodos, clases, paso de mensajes y dependencias.
 - Para encontrar errores al reorganizar un programa (*)





- Refactorizamos por varios motivos:
 - Para eliminar código duplicado
 - Con una visión global del código.
 - Para hacerlo más legible/fácil de entender
 - Renombrando identificadores, reorganizando parámetros, métodos, clases, paso de mensajes y dependencias.
 - Para encontrar errores al reorganizar un programa (*)



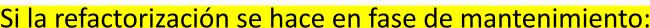


- ¿Qué se consigue?
 - mejorar el diseño del código
 - mejorar su legibilidad
 - mejora la productividad de los programadores
- En otras palabras:
 - 1) Calidad. Código sencillo y bien estructurado, legible y entendible sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses.
 - 2) Eficiencia. Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo invertido en evitar la duplicación de código y en simplificar el diseño se ve compensado a la hora de las modificaciones
 - 3) Evitar la reescritura de código. Refactorizar es mejor que reescribir.





- ¿Qué se consigue?
 - mejorar el diseño del código
 - mejorar su legibilidad
 - mejora la productividad de los programadores
- En otras palabras:
 - 1) Calidad. Código sencillo y bien estructurado, legible y entendible sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses.
 - 2) Eficiencia. Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo invertido en evitar la duplicación de código y en simplificar el diseño se ve compensado a la hora de las modificaciones
 - Evitar la reescritura de código. Refactorizar es mejor que reescribir.



Si la refactorización se hace en fase de mantenimiento:





Le llamamos MANTENIMIENTO PERFECTIVO o REFACTORIZACIÓN A POSTERIORI

4 REFACTORIZACIÓN

4.1 ¿QUÉ ES REFACTORIZAR?

- 4.3 TIPOS DE REFACTORIZACIÓN
- **4.4 CONSEJOS Y PATRONES**
- 4.5 COHESIÓN Y ACOPLAMIENTO
- 4.6 IMPLEMENTACIÓN





- ¿Recuerdas los tipos de mantenimiento?
 - Repasemos la UD1 ...
 - ... eran 3 ...
 - ... y ...
 - ... jacabamos de añadir uno más!
 - •
 - •
 - •
 - NUEVO: j





- ¿Recuerdas los tipos de mantenimiento?
 - Repasemos la UD1 ...
 - ... eran 3 y ...
 - ... jacabamos de añadir uno más!
 - EVOLUTIVO
 - ADAPTATIVO
 - CORRECTIVO
 - NUEVO: ¡PERFECTIVO!





- ¿Recuerdas los tipos de mantenimiento?
 - Repasemos la UD1 ...
 - ... eran 3 y ...
 - ... jacabamos de añadir uno más!
 - EVOLUTIVO => mejoras
 - ADAPTATIVO => cambios legales/coyunturales
 - CORRECTIVO => solución de errores
 - NUEVO: ¡PERFECTIVO! => refactorización a posteriori





- ¿Recuerdas los tipos de mantenimiento?
 - Repasemos la UD1 ...
 - ... eran 3 ...
 - ... y ...

... jacabamos de añadir uno más!

- EVOLUTIVO => mejoras
- ADAPTATIVO => cambios legales/coyunturales
- CORRECTIVO => solución de errores
- NUEVO: ¡PERFECTIVO! => refactorización a posteriori





El mantenimiento perfectivo es el único que suele realizar el autor de la fase de codificación. ¿Por qué?







- ¿Recuerdas los tipos de mantenimiento?
 - Repasemos la UD1 ...
 - ... eran 3 ...

... y ...

... jacabamos de añadir uno más!

- EVOLUTIVO => mejoras
- ADAPTATIVO => cambios legales/coyunturales
- CORRECTIVO => solución de errores
- NUEVO: ¡PERFECTIVO! => refactorización a posteriori





El mantenimiento perfectivo es el único que suele realizar el autor de la fase de codificación. ¿Por qué?

- El propio autor del código es quien mejor lo conoce.
- El autor del código es quien realmente refactoriza. El resto, puede que refactorice con éxito pero gran parte del tiempo reescribirá.



- Pero...
 - ¡No siempre esperamos a la fase de mantenimiento (perfectivo) para refactorizar!
- ¿Cuándo debemos refactorizar?
 - Si vemos que el diseño se vuelve complicado y difícil de entender, y que cada paso hacia adelante empieza a ser muy costoso, lo mejor es parar de desarrollar/codificar y analizar si el rumbo tomado es correcto.
- Los síntomas que indican que algún código de software tiene problemas se conocen como "Bad Smells" y suelen ser ...
 - Código duplicado, métodos largos, clases largas, cláusulas switch innecesarias, comentarios, etc.
 - Se debe aplicar una refactorización que permita corregir ese problema





- Pero...
 - ¡No siempre esperamos a la fase de mantenimiento (perfectivo) para refactorizar!
- ¿Cuándo debemos refactorizar?
 - Si vemos que el diseño se vuelve complicado y difícil de entender, y que cada paso hacia adelante empieza a ser muy costoso, lo mejor es parar de desarrollar/codificar y analizar si el rumbo tomado es correcto.
- Los síntomas que indican que algún código de software tiene problemas se conocen como
 "Bad Smells" y suelen ser ...
 - Código duplicado, métodos largos, clases largas, cláusulas switch innecesarias, comentarios, etc.
 - Se debe aplicar una refactorización que permita corregir ese problema

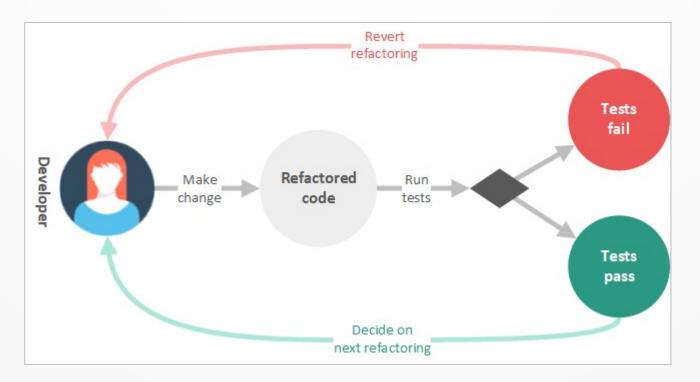


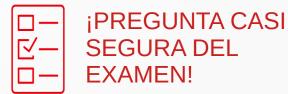
Si la refactorización se hace ANTES de la fase de mantenimiento:

Le llamamos REFACTORIZACIÓN CONTINUA



- Es imprescindible que el proyecto tenga pruebas automáticas, que nos permitan saber si el desarrollo sigue cumpliendo los requisitos que implementaba.
 - Sin ellas, refactorizar conlleva un alto riesgo.







En contra:

- Refactorizar no está considerado como avance del proyecto para los clientes y no suele haber tiempo para ello
 - ¿A qué etapa del ciclo de vida te recuerda esto?

A favor:

- Refactorizar es una forma de mejorar la calidad.
- Bien realizada, rápida y segura, genera satisfacción también en el cliente ¿por qué?







En contra:

- Refactorizar no está considerado como avance del proyecto para los clientes y no suele haber tiempo para ello
 - ¿A qué etapa del ciclo de vida te recuerda esto? documentación

A favor:

- Refactorizar es una forma de mejorar la calidad.
- Bien realizada, rápida y segura, genera satisfacción también en el cliente ¿por qué?
 - Mejora el rendimiento, reduce tiempos





Peligro:

- Es un arma de doble filo, ya que puede ser eterna (siempre hay un "código mejor") => espiral refactorizadora
- Es importante mantener la refactorización bajo control
- Definir claramente los objetivos antes de comenzar a refactorizar y estimar su duración ANTES DE COMENZAR.
- Si se excede el tiempo planificado es necesario un replanteamiento.
- Refactorizar demasiado implica:
 - Pérdida de tiempo
 - Incremento de complejidad de diseño de tanto refactorizar
 - Sensación de no estar avanzando. Sensaciones anímicas negativas.





4 REFACTORIZACIÓN

4.1 ¿QUÉ ES REFACTORIZAR?

4.2 CUÁNDO REFACTORIZAR

4.3 TIPOS DE REFACTORIZACIÓN

4.4 CONSEJOS Y PATRONES

4.5 COHESIÓN Y ACOPLAMIENTO

4.6 IMPLEMENTACIÓN





4.3 TIPOS DE REFACTORIZACIÓN

| • | Refactorización continua [Fase/s]: | |
|---|---|--|
| | | |
| | | |
| | | |
| • | Refactorización a posteriori [Fase/s]: | |
| | | |
| | | |
| | | |





4.3 TIPOS DE REFACTORIZACIÓN

- Refactorización continua [Fase/s diseño/codificación/pruebas/documentación]:
 - No siempre esperamos a la fase de mantenimiento para refactorizar.
 - Si vemos que el diseño se vuelve complicado y difícil de entender, y que cada paso hacia adelante empieza a ser muy costoso, lo mejor es parar de desarrollar/codificar y analizar si el rumbo tomado es correcto.
- Refactorización a posteriori [Fase mantenimiento (perfectivo)]:
 - Refactorizar no está considerado como avance del proyecto para los clientes y no suele haber tiempo para ello
 - Este mantenimiento es el único que suele realizar el autor de la fase de codificación.







4 REFACTORIZACIÓN

- 4.1 ¿QUÉ ES REFACTORIZAR?
- 4.2 CUÁNDO REFACTORIZAR
- 4.3 TIPOS DE REFACTORIZACIÓN
- **4.4 CONSEJOS Y PATRONES**
- 4.5 COHESIÓN Y ACOPLAMIENTO
- 4.6 IMPLEMENTACIÓN





4.4 CONSEJOS Y PATRONES

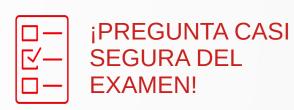
- Tips/consejos:
 - Escribir pruebas unitarias y funcionales (de integración)
 - Si no se hace así, es demasiado costoso e implica mucho riesgo.
 - Usar herramientas especializadas (i.e. IDEs)
 - Buscar patrones de refactorización y de diseño.
 - Permite detectar "Bad Smells" de los que no es consciente y que harán que su código se degrade progresivamente.
 - Comenzar a refactorizar el código tras añadir cada nueva funcionalidad en grupo
 - Realizar discusiones en grupos sobre la conveniencia de realizar alguna refactorización suele ser muy productivo.
 - Añadir la **refactorización continua** como una fase más del ciclo de vida
 - Incorpora la refactorización como una etapa/fase más





4.4 CONSEJOS Y PATRONES

- Los patrones más habituales son (no los memorices, entiéndelos):
 - Duplicated code (el mismo código o programa en muchas partes)
 - Si se detecta el mismo código en más de un lugar... ¡refactorizar!
 - ¿qué hacer?: decidir dónde va ese programa y reutilizarlo
 - 2) Long method (en líneas de código del método)
 - Si un método tiene "muchas" líneas de código... ¡refactorizar!
 - ¿qué hacer?: dividir el método en submétodos y/o redistribuir sus funcionalidades
 - 3) Large class (en líneas de código de la clase)
 - Si una clase intenta resolver muchos problemas... ¡refactorizar!
 - ¿qué hacer?: dividir la clase en subclases y/o redistribuir sus funcionalidades
 - 4) Long parameter list (parámetros de un método)
 - Si un método tiene demasiados parámetros... ¡refactorizar!
 - ¿qué hacer?: estudiar si realmente el método está cohesionado (realiza una única función)

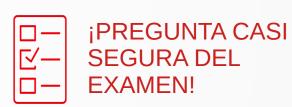






4.4 CONSEJOS Y PATRONES

- 5) Divergent change (cambio divergente)
- Cuando una clase es frecuentemente modificada por diversos motivos ... ¡refactorizar!
- ¿qué hacer?: algo pasa en la clase, ¿reescribir?
- 6) Shotgun surgery (cirugía de escopeta)
- Si tras un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio ... ¡refactorizar!
- ¿qué hacer? reducir el acoplamiento (interdependencia entre módulos)
- 7) Feature envy (envidia de funcionalidad)
- Un método que utiliza más cantidad de elementos de otra clase que de la propia...¡refactorizar!
- ¿qué hacer? pasar el método a la clase cuyos componentes son más requeridos para usar







4 REFACTORIZACIÓN

- 4.1 ¿QUÉ ES REFACTORIZAR?
- 4.2 CUÁNDO REFACTORIZAR
- 4.3 TIPOS DE REFACTORIZACIÓN
- 4.4 CONSEJOS Y PATRONES

4.5 COHESIÓN Y ACOPLAMIENTO

4.6 IMPLEMENTACIÓN



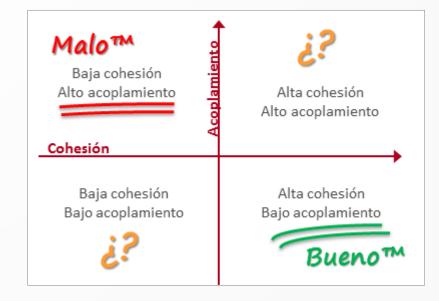


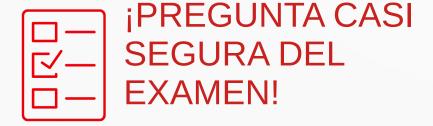
4.5 COHESIÓN Y ACOPLAMIENTO

Acoplamiento entre módulos / métodos / clases (hay que reducirlo)

- El acoplamiento es la manera que se relacionan varios componentes entre ellos.
- Si existen muchas relaciones entre los componentes, con muchas dependencias entre ellos, tendremos un grado de acoplamiento alto.
- Si los componentes son independientes unos de otros, el acoplamiento será bajo.
- Al igual que con la cohesión, podemos ver el acoplamiento a distintos niveles y existe acoplamiento entre los métodos de una misma clase (o las funciones de un módulo), entre distintas clases o entre distintos paquetes.

MÁS INFO: https://blog.koalite.com/2015/02/cohesion-y-acoplamiento/





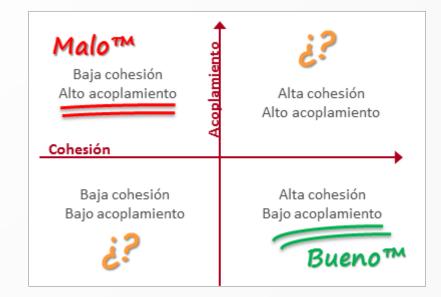


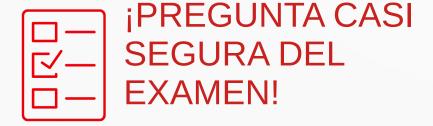
4.5 COHESIÓN Y ACOPLAMIENTO

Cohesión de un módulo / método / clase (hay que aumentarla)

- Podríamos definir la cohesión como lo estrecha que es la relación entre los componentes de algo.
- Si hablamos de clases, una clase tendrá una cohesión alta si sus métodos están relacionados entre sí, tienen una "temática" común, trabajan con tipos similares, etc.
- Si pasamos a componentes de mayor tamaño, como paquetes o librerías, tendríamos una cohesión alta cuando las clases que lo forman están muy relacionadas entre sí, con un objetivo claro y focalizado.

MÁS INFO: https://blog.koalite.com/2015/02/cohesion-y-acoplamiento/







4 REFACTORIZACIÓN

- 4.1 ¿QUÉ ES REFACTORIZAR?
- 4.2 CUÁNDO REFACTORIZAR
- 4.3 TIPOS DE REFACTORIZACIÓN
- **4.4 CONSEJOS Y PATRONES**
- 4.5 COHESIÓN Y ACOPLAMIENTO

4.6 IMPLEMENTACIÓN





4.6 IMPLEMENTACIÓN

Existen muchas formas de refactorizar de manera automática.

