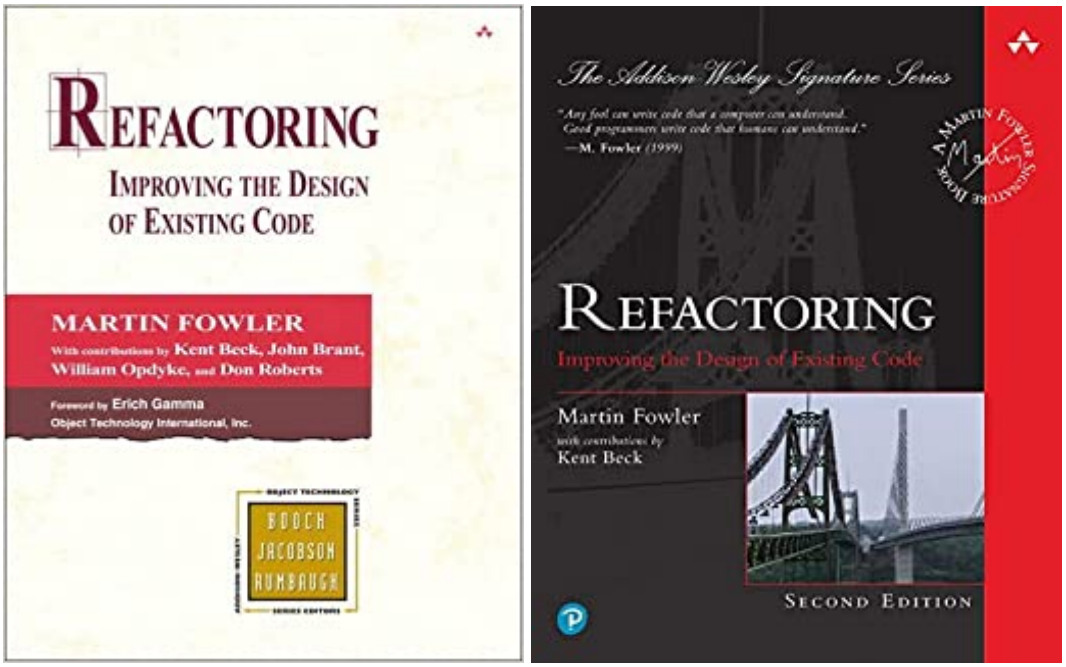
Refactoring



1er y 2da edición de Refactoring de Martin Fowler.

¿Qué es un Refactor?

Es un cambio realizado a la estructura interna del software con el objetivo de hacer que el mismo sea más sencillo de entender y más barato de modificar sin cambiar su comportamiento observable.

¿Qué es Refactoring?

Refactoring es el proceso de reestructuración del software mediante la aplicación de una serie de refactors sucesivos sin modificar su comportamiento observable (lo que el usuario percibe).

REFACTORING no agrega funcionalidad y NO modifica el comportamiento del sistema.

“The two hats”

The two hats es la metáfora elegida en el libro de Refactoring para dejar en claro la separación de etapas dentro de nuestro proceso de desarrollo. Cuando vamos a agregar funcionalidad “nos ponemos un sombrero” y cuando vamos a hacer refactoring nos lo quitamos y nos ponemos un sombrero diferente. Lo importante de esto es entender que agregar funcionalidad y hacer refactoring son etapas **mutuamente exclusivas**.

Cuando hacemos refactoring no podemos, mejor dicho podemos pero no deberíamos, agregar funcionalidad y/ó modificar los tests. Modificar los tests implicaría que perderíamos la capacidad de verificar si nuestro refactor “rompió” algo o no. Los cambios a los tests solamente son permitidos en los siguientes escenarios

- Para agregar un caso de prueba que olvidamos
- Para ajustar los tests a la nueva interfaz si la misma cambió (quizás cambiamos la firma de algún método)

Cuando agrego funcionalidad agrego el código que sea necesario para implementar la misma y los tests que verifican que esta funciona, pero no hago refactoring a la vez.

Esta separación de etapas nos ayuda a mantener un enfoque disciplinado. No modificar los tests durante la etapa de refactoring nos garantiza que no estamos introduciendo ningún defecto en el código.

Sugerencias de cuándo aplicar Refactoring

Algunas reflexiones personales

- No hay que pedir permiso para hacer Refactoring. Si ven algo en el código en el que están trabajando que no está del todo bien lo mejor es arreglarlo lo antes posible (obviamente dependiendo del tamaño del refactor).
- Lo mejor es incorporar como hábito hacer Refactoring a diario. No tienen porqué ser Refactors enormes, con que mejoremos el código un 1% o un poquito todos los días a largo plazo esta acumulación de mejoras termina siendo tremendamente valiosa.

La regla de los 3 strikes

- La primera vez que tengo que agregar código, para arreglar un defecto o implementar una funcionalidad, simplemente lo hago.
- La segunda vez si veo que agregué código similar a algo que ya existía la dejo pasar.

- La tercera vez que agrego código similar hago el refactor.

El código duplicado siempre es mucho más barato que la abstracción incorrecta.

The Wrong Abstraction

Refactoring para simplificar una feature

Sumado a la regla de los 3 strikes, siempre es bueno preguntarnos al momento de implementar una funcionalidad qué diseño podría tener, que forma tendría este diseño, que me simplifique la forma de implementar esta funcionalidad.

¿Hay algún fragmento de código que me convendría reutilizar? ¿Lo puedo mover de lugar para reutilizarlo?

¿En lugar de preguntar por el tipo de un objeto puedo implementar una operación polimórfica antes?

Refactoring para aumentar la transparencia

Quizás me tocó trabajar con un fragmento de código que no entendí “de primera”. Puede que me haya llevado a

- Hacerle una pregunta a un compañero
- Leer documentación
- Leer un poco más de código, más allá del fragmento que me tocaba modificar, para entender el contexto

¿Puedo hacer un refactor para que el conocimiento que adquirí ahora del problema se vea reflejado en el código?

Relación de Refactoring con el Diseño

El refactoring va de la mano con la concepción de que el diseño en el software es algo continuo, que nunca termina. El diseño sucede día a día, a la par con el desarrollo del sistema. No es una gran etapa que la hacemos previa al desarrollo y luego nos olvidamos.

Mientras construimos el sistema ganamos conocimiento sobre el dominio y este conocimiento nos permite ir mejorando el diseño. El efecto acumulativo de refactorings

aplicados en serie va mejorando poco a poco la calidad del diseño y previene que el mismo se degrade.

Con la ayuda de Refactoring en lugar de especular en toda la flexibilidad que mi diseño va a necesitar en el futuro, construyo software que solamente resuelve las necesidades actuales.

Pero lo diseño de forma de que esté excelentemente diseñado para las necesidades actuales.

YAGNI

YAGNI, del acrónimo *You Aren't Gonna Need It*, es un principio que proviene de la metodología *Extreme Programming* y establece que no se debe agregar software hasta que sea estrictamente necesario. Nunca debemos programar por necesidades que creemos que podemos llegar a tener.

Programar pensando en lo eventual produce software con complejidad innecesaria, difícil de entender y con código que posiblemente no se utilice.

El BUFD, *Big Up Front Design*, no es realista con la mayoría de los escenarios que ocurren hoy en día en la industria, donde anticipar todos los posibles ejes de cambio en un sistema es muy difícil. El software que contemple todos los cambios que podrían venir sería demasiado complejo de construir.

YAGNI no implica el fin del diseño o de la arquitectura de software, sino que YAGNI implica entender que el diseño es algo que sucede permanentemente. Debemos apuntar a diseños simples que resuelvan el problema que tenemos HOY lo mejor posible.

Comentarios finales

- Hacer refactoring es riesgoso si no tenemos self testing code. No tenemos una forma de validar rápidamente si introducimos algún bug. Lo mejor en esos casos es subir primero el % de coverage y luego refactorear.
- Es difícil que hagamos el código bien "de una". Al comenzar un proyecto no tenemos suficiente conocimiento del dominio como para tomar las mejores decisiones de diseño posible. La clave para mantener el código legible y simple es el refactoring continuo.
- Es conveniente hacer refactoring en pasos pequeños y ejecutar los tests seguido. De esta forma minimizamos el riesgo de romper muchos tests y estar mucho tiempo debuggeando el problema.

- El código perfecto no existe. No podemos estar todo el tiempo haciendo refactoring así como tampoco es bueno no hacer refactoring nunca. Hay que saber balancear. Algunos puntos sobre los que reflexionar pueden ser
    - ¿Este módulo es importante?
    - ¿Cuánto tiempo me va a llevar este refactor?
    - ¿Cuánto gano en productividad con este refactor?

Catálogo de Refactors

https://refactoring.guru/refactoring/techniques

¿Qué es un Smell o Code Smell?

Es un síntoma o indicador, subjetivo, de que puede haber un problema mayor con la calidad del diseño de nuestro sistema. Nos puede indicar puntos dónde quizás sea conveniente aplicar un refactor.

Catálogo de Code Smells

https://refactoring.guru/refactoring/smells

Links útiles

https://martinfowler.com/articles/designDead.html