15. Kata, TDD, coding dojo y code smells

kata: actividad en repetición para practicar diferentes soluciones a diferentes escenarios. Se busca aprender en el camino

Code Smells

Algo huele mal, puede haber un problema. Algunos son:

- *Código duplicado*, lógica duplicada
- *long method*: metodo muy largo (SOLID)
- *large class*: clase con muchas responsabilidades
- *long parameter list*: método con muchos parámetros
- *divergent change*: cada vez que se debe cambiar la clase o módulo es por una razón diferente (rompe srp)
- *shotgun surgery*: bastante acomplamiento (un cambio en un lugar implica cambios en varios lugares). si muchas clases tienen dependencias a un tipo, si lo cambiamos lo debemos cambiar en todas las clases por ejemplo.
- *feature envy:* un método de una clase accede constantemennte a datos de otra clase (el método se interesa más en los atributos de la otra clase que de sí misma, probablemente el método es responsabilidad de la otra clase). Probablemente rompe Tell dont ask.
- *data clumps*: se pasan por parámetro puros datos en vez de una encapsulación de estos. Por ejemplo, una clase con tres métodos que reciben los mismo 4 parámetros.
- *primitive Obsession*: se trabajan con tipos primitivos, tambien falta modela un tipo de alto nivel.
- *switch statements*: puede que falte una herencia no modelada, pueden estar en diferentes funciones (una diferente para cada caso).
- *Lazy class*: clase con una sola función (puede que le falten responsabilidades)
- *Message Chains*: suele usarse en el patron builder pero la problemática es que hay varias dependencias encadenadas, si se necesitan un montón de intermediarios sería mejor que no haya una clase que conozca todas las

dependencias (las dependencias siguen existiendo pero la diferencia es que la de más alto nivel no tiene todas las dependencias)
- *Data class*: en vez de un objeto en realidad se usa como si fuese un struct. (dependiendo del caso está bien o no).

refactoring

es: Cambiar la implementación del código (mejorando la calidad) sin cambiar el comportamiento externo (esto lo verificamos con test unitarios).
lo hacemos porque: no es un buen diseño, solucionamos bugs, mejorar prácticas.7

flow

1. todos los test pasan
2. buscamos code smells
3. buscar refactoring
4. aplicar refactoring

TDD

test-driven development: primero programamos los test de lo que queremos hacer, para eso escribimos el código que valida el comportamiento que queríamos agregar.
Trabajamos en un ciclo *R*ed *G*reen *R*efactor.

- *red*: escribimos el código que falla porque aun no está implementado el feature
- *green*: escribimos el código mínimo para que el test pase. Solo nos enfocamos en la funcionalidad en particular que busca testearse. Si se agrega otra funcionalidad, el otro test verá en validar que lo otro suceda.
- *refactor*: modificamos el código, miro panorámicamente el diseño y trato de mejorarlo sin romper el código. La red de contención son los test

previamente escritos

reglas

1. No puedes escribir código productivo a menos que un test falle.
2. Escribimos un test a la vez que sea suficiente como para una falla (de compilació o funcionalidad), esto hace que solo trates de solucionar un problema a la vez.
3. no estas permitido de escribir código que no se enfoque en pasar o fallar un único test unitario.

baby steps

Evitar perder el tiempo en errores grandes: no saltearse test (esto dejaría una cobertura no ideal).

Por ejemplo:
Se desea tener un wallet de cuenta que nos permita determinar cuando disponemos de cada moneda.
Al cual se le pueda agregar (cantidades positivas) y quitar (cantidades existentes positivas) de dicha moneda de la cuenta.
Se desea monitorear el mercado de valores para saber el precio actual y sus variaciones históricas.
Cada cotización es entre dos monedas, indica cuanto se recibe de la primera a cambio de cada unidad de la segunda.
Se desea leer la cotización actual y ser notificado ante un cambio de cotizacion.
Se desea poder reiniciar el historial de una cotización, y saber el mayor y menor valor obtenido por la moneda desde que se reinicio el monitoreo.
Se desea que el usuario pueda definir un conjunto de reglas en base a las cotizaciones que, de cumplirse, realicen ciertas acciones.
Como ejemplo especifican estas:

1. Si el valor actual es menor a 95% del máximo, entonces vender 0.1 unidades de la moneda y reiniciar el monitoreo
2. Si el valor actual es mayor a 105% del mínimo entonces comprar 0.1 unidades de la moneda y reiniciar el monitoreo

3. Si compré 0.1 y el valor actual es mayor o igual al que compre, vendo 0.1
4. Si vendí 0.1 y pasaron mas de 5 minutos y el valor actual es menor o igual al que vendí, compro 0.1