Resumen Refactoring: Improving the Design of Existing Code

by Martin Fowler

Capitulo 1 y 2: Capitulos mas que nada introductorios, con conceptos interesantes pero ya cosas que entendia de objetos y de lo visto en estas primeras clases ademas de que la mayoria de cosas uno lo puede intuir facilmente. No veo importante resumir o dejar tanto texto de esto, prefiero hacer enfasis en el cap 3. Para un posible final, estaria bueno releerlos.

Capitulo 3:

MALOS OLORES:

Duplicated Code: Este es sencillo, obviamente indica que cada caso es distinto ya que el tener codigo duplicado en un metodo no es lo mismo que tener codigo duplicado en dos metodos de sub clases o mil ejemplos mas, para solucionar cada caso habria que usar x o y tecnica pero eso ya va a ser una cuestion nuestra. Pero hay que tener claro y pensar bien cada caso, entiendo que en practicamente todos tener codigo duplicado es malo aunque puede pasar que el eliminarlo sea mas complejo y haga menos leible el codigo, en este caso lo mejor es dejarlo y ahorrarnos problemas.

Long Method: Es importante mantener metodos cortos ya que esto los hara mas leible y a su vez es muy importante el nombre que se le ponga a cada metodo ya que esto obviamente ayuda a la legibilidad de los metodos. Practicamente siempre para solucionar este olor se debe extraer metodos para asi acortarlo. Un bloque de codigo con un comentario es un indicio a que ese bloque de codigo puede ser un metodo. El tener muchos condicionales tambien es un indicio.

Large Class: Un indicio puede ser demasiadas variables de instancia esto nos puede indicar que una clase quiere hacer demasiadas cosas y el tener demasiadas variables de instancia tambien es un indicio de codigo repetido. Separar las clases es una forma de dividir las variables. Al igual que con las variables, una clase con mucho codigo es indicio de que hay que dividirlo. Ademas de extraer/separar clases, tambien se pueden extraer subclases. Ademas de disminuir su codigo haciendo metodos (los cuales entiendo podrian estar en una clase padre).

Long Parameter List: Hace referencia a que un metodo reciba muchos parametros. Cambiar un parametro por un metodo puede ser una opcion para solucionarlo. Ademas existen mas formas para solucionar esto que serian mas logicas de cada caso.

Divergent Change: Esto ocurre cuando una clase es comun que se modifique por cada vez que se “agranda” el sistema, por ejemplo si hay 3 metodos que se tienen que cambiar cada vez que se realiza una modificacion es probable que sea mejor tener dos objetos en vez de uno para unificar estos metodos por decir de algun modo y solo modificar un objeto por cada tipo de cambio. Para esto se puede usar una clase abstracta.

Shotgun Surgery: Este es similar al divergent change pero siendo lo contrario. Basicamente el Shotgun Surgery es un cambio que altera muchas clases. Dado que cuando los cambios estan dispersos son dificiles de encontrar, para estos casos se pueden mover los metodos a una unica clase para tener todo en un solo lugar por asi decir, esto produce divergent change pero por un “bien mayor”.

Feature Envy: Hace referencia a un metodo que parece mas interesado en una clase distinta que en la que se encuentra. Basicamente cuando usa muchas metodos de otra clase. Es un indicio que el mismo metodo deberia ser de la otra clase. Habria que mover el metodo. Si un metodo utiliza datos de varias clases se manda a la clase que mas utilice basicamente.

Data Clumps: Datos que siempre se encuentran juntos, deberian ser un objeto. Puede que pase que luego de juntarlos y hacerlo un objeto haya que pasarlo como parametro o utilizarlos de otra forma y haya algun dato que se use mas que otro pero esto no seria una preocupacion, ya que entiendo es preferible cambiar varios parametros por un objeto. Y luego de realizar esto podriamos buscar metodos que posean feature envy y que finalmente sean metodos que deban estar en la nueva clase, lo que convierte a la nueva clase en una mas productiva por decir de algun modo.

Primitive Obsession: Entiendo hace foco en transformar elementos primitivos como puede ser una fecha o mejor dicho un periodo el cual se puede realizar con dos fechas en una clase que tenga sus propias funcionalidades, este seria el ejemplo mas claro para este olor la verdad. Entiendo que hay muchos ejemplos mas pero bueno basicamente seria formar una clase con datos primitivos que entiendo formarian algo mas grande y tendrian sus propias funcionalidades. Para esto se pueden usar muchas herramientas dependiendo como se encuentren los datos.

Switch Statements: hace referencia a los problemas que producen las sentencias switch (codigo duplicado) y que hay que tener en cuenta el polimorfismo para solucionar estos casos. Si uno de los condicionales es nulo se podria usar un null. Basicamente hace referencia a los momentos que haces IF x hace esto, y hace esto, z hace esto. Esto se soluciona haciendo clases a x, y, z que serian subclases de otras y usando polimorfismo.

Parallel Inheritance Hierarchies: hace referencia a los casos en que cada vez que se hace una subclase de una clase en una jerarquia hay que hacer otra subclase en otra jerarquia. La verda que no se me ocurren ejemplos para esto. Para identificar esto, las instancias de una hacen referencia a la otra. Podemos mover metodos y datos para solucionar y eliminar la jerarquia referencia entiendo. Osea juntarlo todo en una.

Lazy Class: hace referencia a clases sin comportamiento o con muy poco comportamiento, podriamos decir que cuesta mantener cada clase y entonces si cada clase no hace los meritos suficiente para mantenerse hay que eliminarla. Esto se puede dar si al realizar refactorizacion una clase queda muy chica. Tambien puede pasar con una jerarquia que tiene subclases que ya no hacen lo suficiente y por ello se pueden juntar en una clase.

Speculative Generality: hace referencia a clases abstractas que no hacen lo suficiente. Si esta tiene metodos que no se utilizan se eliminan y si tiene metodos abstractos se pueden pasar a las clases concretas. Este olor puede detectarse cuando los unicos usuarios de un metodo o clase son los casos de prueba.

Temporary Field: hace referencia a variables de instancia que se establecen solo en determinadas circunstancias. Basicamente se espera que un objeto necesite de todas sus variables de instancia. Estas variables deberian ser transladadas a una nueva clase en donde siempre sean referenciadas, esta nueva clase recibe todo el codigo de estas variables y en la primera clase se pueden usar null en los casos donde se esperaba que la variable no fuera instanciada. Esto se puede dar cuando una clase necesita varias variables para un solo metodo especifico en este caso todo esto se puede transladar a una nueva clase.

Mesaage Chains: esto se da cuando se produce que un objeto pida a otro objeto y este a otro y asi. Puede verse como una larga lista de getThis. Una solucion para esto es ver la parte de codigo que se usa del objeto que tanto se esta buscando y extraerla para asi ahorrarnos esta cadena. Y ademas se podria usar un nuevo metodo para ahorrarnos “el camino” si es que mas metodos lo usan. Osea entiendo es mejor una cadena de metodos que de mensajes.

Middle Man: hace referencia a los casos en los cuales una clase tiene muchos metodos que son delegando a otra clase por eso es un “hombre de medio”. Esto se puede solucionar eliminando esta clase y yendo directamente a la clase que esta haciendo las funciones. Si esta clase de en medio tiene comportamiento adicionales se puede definir como una subclase. Lo cual permite ampliar el comportamiento sin funcionar como un delegado.

Inappropiate Intimacy: Esta no me quedo clara. Capaz hace referencia a clases que no tienen relacion con otras y hay que modificarlas para que se relacionen.

Alternative classes with differente interfaces: hace referencia a renombrar metodos de distintas clases que hagan lo mismo y tengan distinto nombre. Si sucede que queda raro se pueden mover los metodos entre clases.

Incomplete library class: darle otra leida.

Data class: entiendo que se refiere a clases sin comportamiento. Habria que buscarselo.

Refused Bequest: hace referencia a jerarquias que estan mal porque una clase recibe demasiados metodos que no utiliza para esto habria que definir de mejor forma la jerarquias para que esto no suceda. Aunque se indica como un olor leve por lo cual no seria tan grave si sucede que no usen todo porque es lo comun, se deberia hacer lo de organizar la jerarquia si es muy alevoso que utiliza pocos metodos.

Comments: hace referencia a que a menudo se usan los comentarios para explicar el codigo, lo cual obviamente esta mal ya que se supone el codigo deberia ser legible e interpretable por si solo. Ya que para eso los metodos tienen nombre y las clases y las variables y todo. Como consejo se indica que si sentimos la necesidad de escribir un codigo, pensemos en refactorizar antes.