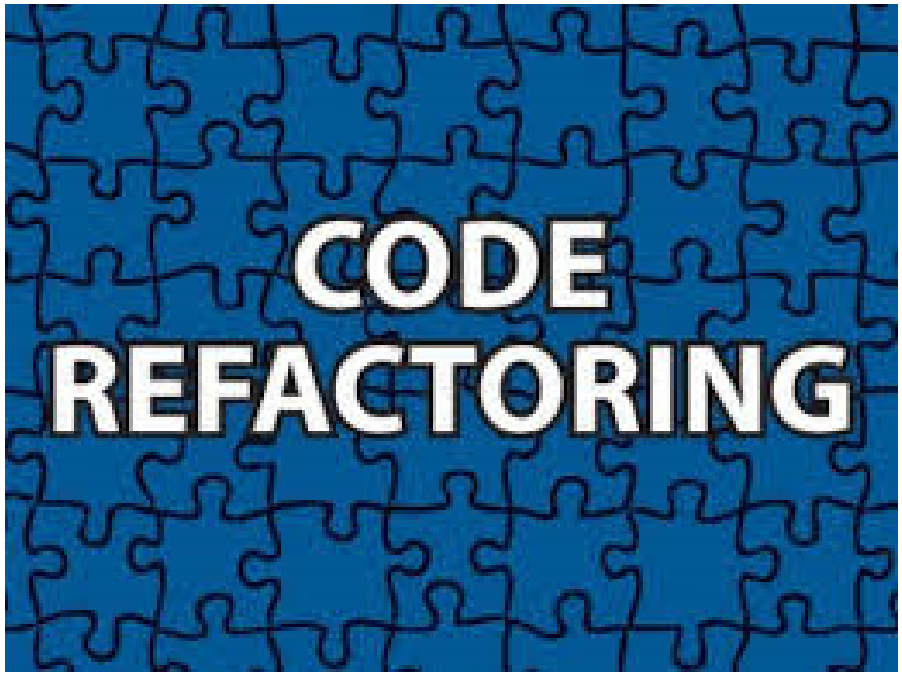
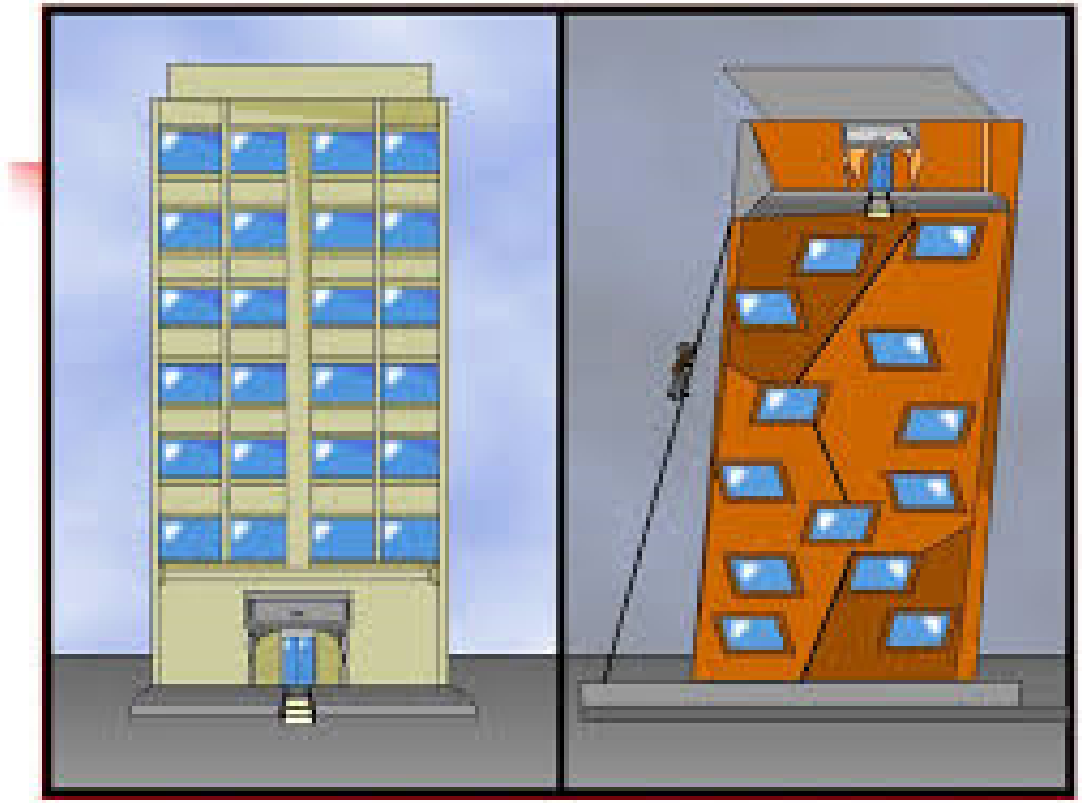
Clase 10

Refactoring



Mejorando el diseño del código existente

1. Introducción

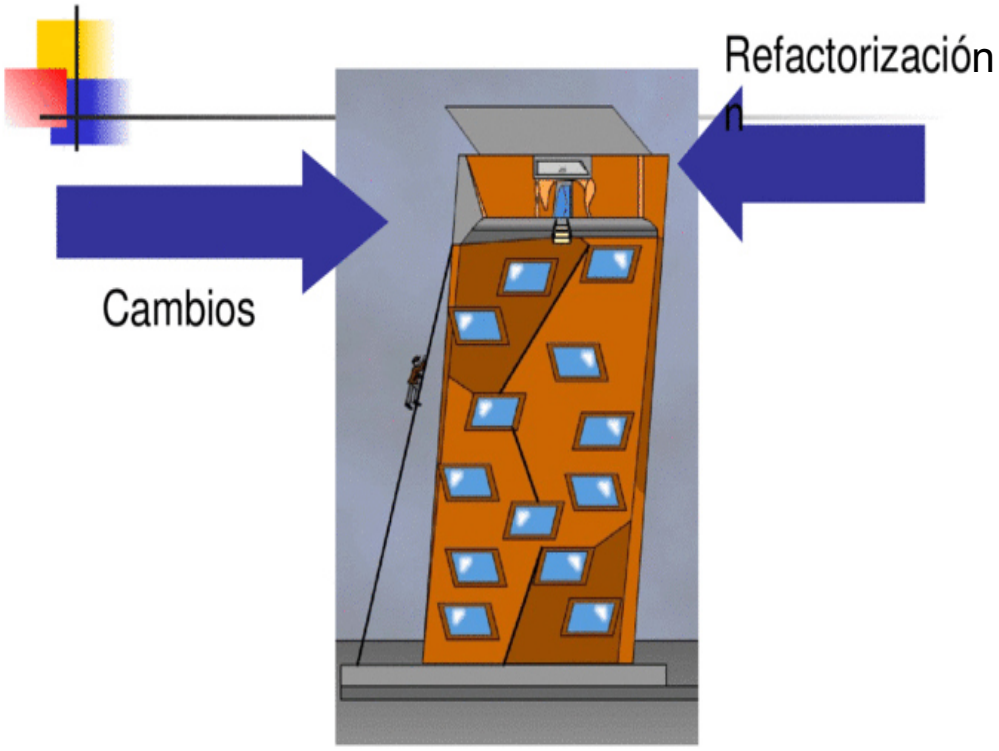


¿Si su software fuera un edificio, se parecería más a uno de la izquierda o de la derecha ?

1. Introducción

Mientras más antiguo sea el código y más grande, el software padece:

- "Código mutante"
- "Diseño roto"



Ese software necesita **cambios**

1.1 ¿Por qué nuestro software sufre degeneración?

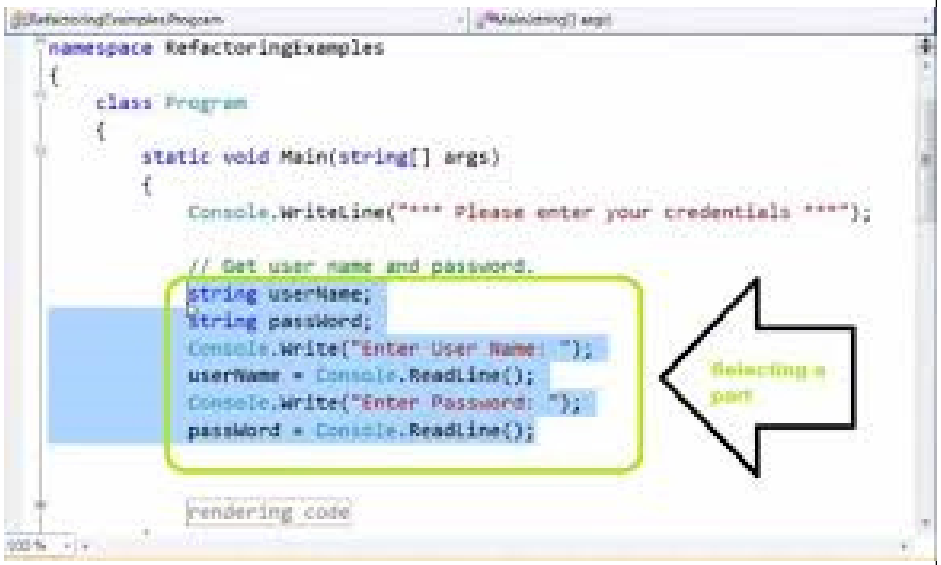
Hay que cumplir con la fecha de entrega comprometida,
es LA PRIORIDAD NUMERO UNO!

1.1 ¿Por qué nuestro software sufre degeneración?

- Es difícil hacer una estimación confiable.
- Es difícil cumplir con lo planeado.
- Aparecen los bugs.
- Resulta difícil solucionar los bugs.
- Aparecen Expertos o Dueños de código.

1.2 ¿Por qué pasa esto?

- El software es complejo.
- Hay diferentes modos de manejar la complejidad, a través de un proceso de:
  - encapsulación
  - frameworks
  - reutilización,etc.
- Hay que empezar a manejar la complejidad en el nivel de código.



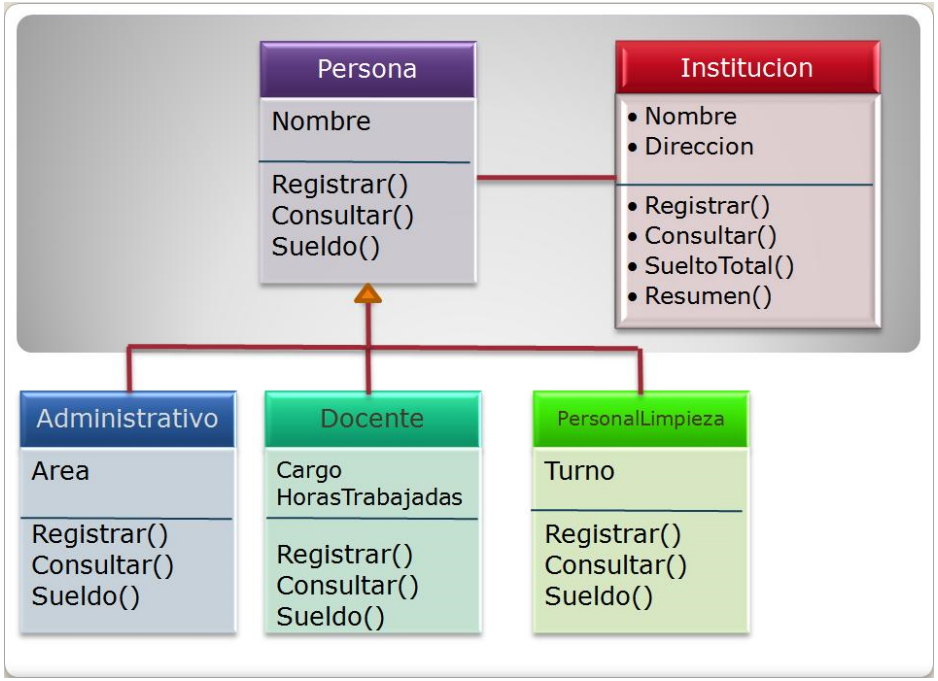
1.2 ¿Por qué pasa esto?

- Antes, nuestras prioridades eran tener un código rápido, que ocupe poca memoria, optimizado utilizando los algoritmos más eficaces etc...

- Hoy en día el enfoque es un código simple.

1.3 ¿Cómo es un código simple?

- Funciona bien.
- Comunica bien lo que esta haciendo.
- No tiene duplicación.
- Tiene el menor número posible de clases y métodos.



1.4 ¿Cuáles son los beneficios?

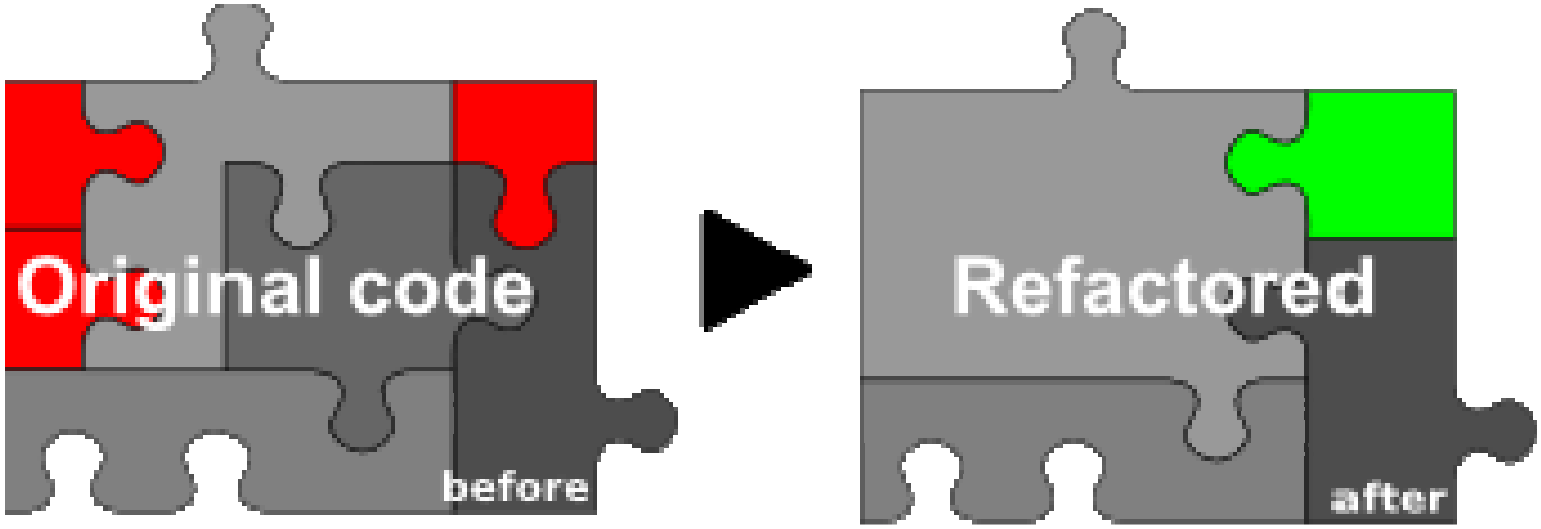
- El código es más fácil de cambiar, evolucionar o arreglar.
- El código es más fácil de leer y entender.
- Es más fácil hacerlo bien desde la primera vez, así estamos programando más rápido.

1.5 ¿Qué significa Refactorizar?

Refactorizar significa cambiar el código internamente sin alterar su funcionalidad externa. En general, con motivos de mejorar el diseño y obtener un código más simple.

1.6 ¿Cómo nos puede apoyar la Refactorización?

La **Refactorización** enseña técnicas para descubrir el código de mala calidad y técnicas para cambiarlo.



Refactorización

- "Modificación hecha en la estructura interna del software para mejorar su comprensión y abaratar el costo de mantenimiento sin alterar su comportamento externo". (Martín Fowler)

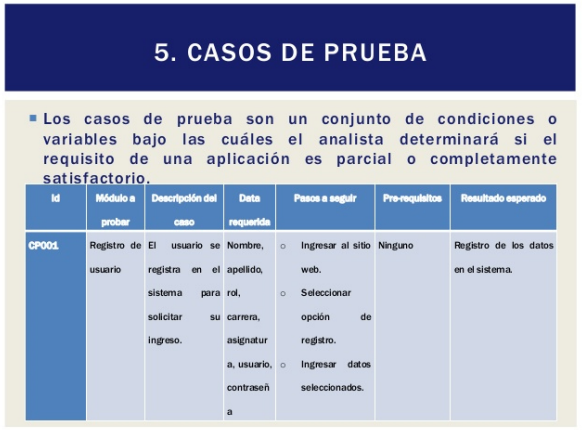
- "Es el arte de evolucionar el diseño del código". (William C. Wake)

2. Proceso de refactorización

Es importante contar con un buen lote de casos de prueba que validen el correcto funcionamiento del sistema.

Los pasos a seguir son los siguientes:

- ***Ejecutar las pruebas, para obtener información sobre el comportamiento actual del sistema.***
- ***Analizar los cambios a realizar.***
- ***Aplicar los cambios.***
- ***Ejecutar las pruebas y corroborar que los resultados antes y luego de efectuada la refactorización son iguales.***



5. CASOS DE PRUEBA

- Los casos de prueba son un conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio.

Id	Módulo a probar	Descripción del caso	Data requerida	Pasos a seguir	Pre-requisitos	Resultado esperado
CP001	Registro de usuario	El usuario se registra en el sistema para solicitar su ingreso.	Nombre, apellido, rol, carrera, asignatura, usuario, contraseña	○ Ingresar al sitio web. ○ Seleccionar opción de registro. ○ Ingresar datos seleccionados.	Ninguno	Registro de los datos en el sistema.

2. Proceso de refactorización

Si los resultados de las pruebas son iguales se verifica que no se ha modificado el comportamiento observable del sistema y, por consiguiente, no se han introducido fallas.

Optimizar no es Refactorizar.

- En ambos casos se modifica el código fuente sin alterar el comportamiento observable del software.
- En la optimización se le suele agregar complejidad al código.

2. Proceso de refactorización

Aspectos favorables

- Favorece el mantenimiento del diseño del sistema.
- Facilita la lectura y comprensión del código fuente.
- Facilita la detección temprana de errores.
- Permite programar mas rápido, lo que eleva la productividad de los desarrolladores.

2. Proceso de refactorización

Aspectos desfavorables

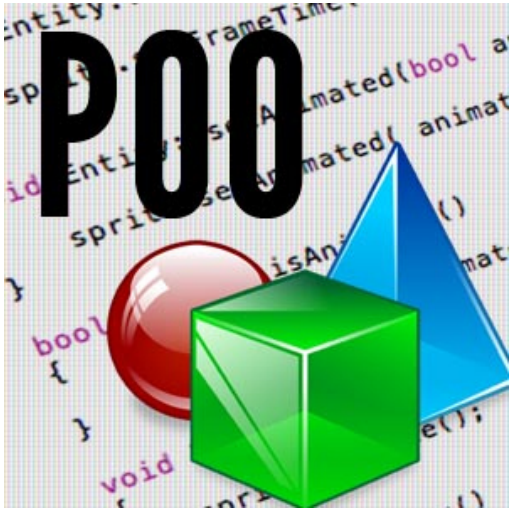
- En las **bases de datos**, se deben aplicar los cambios necesarios y luego migrar los datos existentes en la base de datos, lo que es muy costoso.

2. Proceso de refactorización

Aspectos desfavorables

- Los **cambios de interface**. Uno de los beneficios de la POO es el hecho de poder cambiar la implementación sin alterar la interface expuesta.

El **cambio de interface**:
-es **simple** si se tiene acceso al código fuente de todos los clientes de la interface a refactorizar.
-es **complejo** si no se dispone del código fuente modificable de los clientes de la misma.



2. Proceso de refactorización

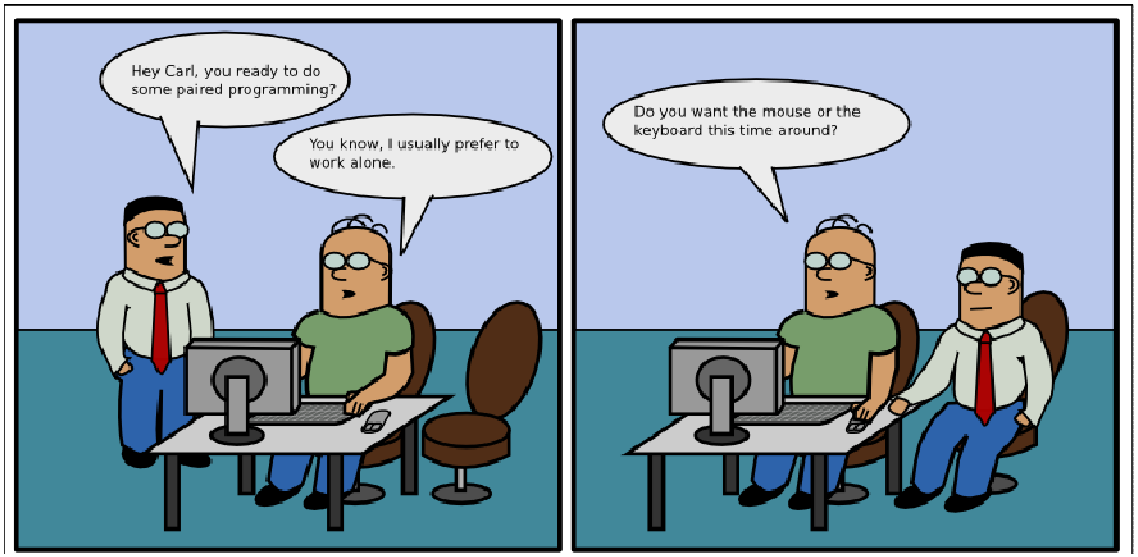
¿Cómo y cuándo refactorizar?

Hay que **refactorizar** cuando:

- Estamos agregando nueva funcionalidad al código.
- Estamos solucionando una falla.
- Estamos haciendo revisión de código. (Distribución del conocimiento dentro del equipo de desarrollo).

2. Proceso de refactorización

¿Cómo y cuándo refactorizar?



La **técnica *pair programming*** *de* **extreme programming** implica una constante revisión de código y refactorizaciones a lo largo del desarrollo. *Esta técnica involucra* dos desarrolladores por computadora.

2. Proceso de refactorización

¿Cuándo no deberíamos refactorizar?

- Es más fácil hacerlo de nuevo.
- El código no funciona.
- Demasiado cerca de la fecha de entrega comprometida.

2. Proceso de refactorización

Identificar puntos débiles de código

- Es difícil definir si el código es malo o bueno, o cuándo deberíamos cambiarlo.
- Es difícil imponer las métricas.

En estos casos se habla de Malos Olores en el código (Bad Smells - Kent Beck)

3. Síntomas que indican la necesidad de refactorizar

A continuación se describen brevemente:

1. Código duplicado
2. Código muerto
3. Métodos largos
4. Clases largas
5. Lista de parámetros larga
6. Cambios divergentes
7. Cirugía de la escopeta
8. Envidia de funcionalidades
9. Los switch
10. Obsesión con los primitivos

3. Síntomas que indican la necesidad de refactorizar

11. Comentarios
12. Grupos de datos
13. Clase perezosa
14. Generalización especulativa
15. Campo temporal
16. Mensajes encadenados
17. Intermediario
18. Intimidad inapropiada
19. Clases alternativas con diferentes interfaces
20. Legado rechazado

Código duplicado

- Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- Si un código con un bug aparece repetido 5 veces, es como tener 5 bugs en el software.

Posible Solución:

- Sacar el código duplicado a un método o clase nueva y utilizarlo en todos aquellos lugares que se necesita.

Código muerto

- Aparición de código que no se utiliza, probablemente procedente de versiones anteriores, prototipos o pruebas.
- También incluye aquél código comentado que se dejó por si algún día…

Posible Solución:
- Revisar realmente si se debe usar, sino borrarlo.

Métodos largos

- Dificultan mucho su comprensión.
- Seguramente realizan más de una responsabilidad.
- En la POO cuando más corto es un método más fácil su reutilización.
- Programas con métodos mas cortos, tienen vida mas larga.

Posible Solución:

- Detectar las diferentes responsabilidades y sacarlas a métodos o clases nuevas.

Clases largas

- Ocurre lo mismo que con los métodos largos.
- Seguramente se encargan de más de una responsabilidad.
- Un síntoma es cuando la clase tiene demasiados atributos.

Posible Solución:
- Detectar las diferentes responsabilidades y sacarlas a clases nuevas colaboradoras.

Lista de parámetros larga

- Una lista larga de parámetros es difícil de comprender.
- No siempre hay que pasar toda la información al método, a veces podemos preguntar a otro objeto.
- En la POO no se suelen pasar muchos parámetros a los métodos, sólo los necesarios para que el objeto involucrado consiga lo necesario.

Posible Solución:

- Encapsular los parámetros relacionados en objetos tipados o reemplazar parámetro por un método.
- No usar datos globales sólo para traspasar datos.

Cambios divergentes

- Una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí.

Posible Solución:
- Extraer las diferentes responsabilidades a distintas clases.

Cirugía de la escopeta

- Opuesto al Cambio divergente, es un tipo de cambio que requiere muchas pequeñas modificaciones a clases diversas.

Posible Solución:

- Trata de corregir ese problema reuniendo todas las partes del código que tienen que modificarse en una única clase cohesiva.

Envidia de las funcionalidades

- Métodos de una clase más interesados en datos de otra clase que en los datos suyos.
- La envidia de la Clase A por recursos de la Clase B es una indicación del acoplamiento fuerte de la Clase A con la Clase B.

Posible Solución:

- Mover la funcionalidad del método de Clase A a la Clase B, que ya está más cerca de la mayoría de datos implicados en la tarea.
- Extraer a un método el código envidioso y mover sólo ese método a la Clase B.

Los Switch

- Lo típico de un software Orientado a Objetos es el uso mínimo de los switch (o case, o switch escondido).
- El problema es la duplicación, el mismo switch en lugares diferentes.

Posible Solución:
- Remplazar condicional con polimorfismo.

Obsesión con los primitivos

- Muchas personas tienen la obsesión de negarse a utilizar objetos pequeños para modelar algunos comportamientos.
- Por ejemplo: es mejor utilizar un objeto para modelar un número de teléfono, que representarlo mediante una cadena de caracteres.

Posible Solución:
- Crear una nueva clase con los primitivos.

Comentarios

- Los Comentarios pueden mentir, el código no.
- No pueden reemplazar falta de código mal escrito.
- Después de una refactorización meticulosa, lo más probable es que los comentarios sean innecesarios.

Posible Solución:

- Extraer el trozo de código comentado a un método y darle un nombre.
- Usar comentarios en zonas en las que el código no sea seguro o se quiera recordar algo para futuras modificaciones.

Grupos de datos

- Uso de un mismo conjunto de datos o propiedades en diferentes lugares en vez de crear una clase apropiada para almacenar los datos, lo que a su vez provoca el incremento de parámetros en métodos y clases.

Posible Solución:
- Aglutinar en uno o varios objetos lógicos y pasarlos como nuevos parámetros.

Clase perezosa

- Toda clase debe tener definida una responsabilidad, una razón para existir.
- Cada clase que se crea cuesta dinero para mantenerla y comprenderla.

Posible Solución:
- Una clase sin responsabilidad hay que dotarla de sentido o bien eliminarla.

Generalización especulativa

- Siempre comienza con las palabras “Algún día necesitaré…”, o “…por si algún día lo necesito”.
- Si se va a armar una infraestructura que va a hacer cosas complejas, puede ser también compleja de entender y eso tiene un costo adicional.

Posible Solución:
- Escribir la funcionalidad que haga falta para las necesidades actuales, y no para las futuras.

Campo temporal

- No cumple el principio de encapsulamiento y ocultamiento de variables haciendo que éstas pertenezcan a la clase cuando su ámbito debería ser exclusivamente el método que las usa.

Posible Solución:

- Si una propiedad sólo se usa para un par de métodos, se debería convertir en un parámetro de esos métodos.
- Otra alternativa es sacar a una clase nueva los campos y métodos que los usan.

Mensajes encadenados

- Se produce cuando se realiza una cadena de llamadas a métodos de clases distintas utilizando como parámetros el retorno de las llamadas anteriores, como A.getB().getC().getD() que dejan ver un acoplamiento excesivo de la primera clase con la última.

Posible Solución:

- Mover los métodos a niveles anteriores o esconder el uso del delegado para minimizar la cadena de llamadas.

Intermediario

- Este es quien se ocupa de delegar las llamadas, y simplemente hace eso.
- Al explorar la clase se observa que solamente está llamando a un método y delegando la llamada.
- Cuestiona la necesidad de tener clases cuyo único objetivo es actuar de intermediaria entre otras dos clases.

Posible Solución:

- Saltear al intermediario y usar la clase que implementa la acción directamente.

Intimidad inapropiada

- Se produce cuando dos clases comienzan a conocer demasiados detalles una de otra.

Posible Solución:

- Mover métodos y propiedades de una clase a otra para eliminar la intimidad.
- Cambiar la comunicación bidireccional por unidireccional.
- Con clases con intereses comunes, extraer dichos intereses comunes a una clase externa.

Clases alternativas con diferentes interfaces

- Indica la ausencia de interfaces comunes entre clases similares.
- También aplicable a métodos.
- Por ejemplo: CalculaFactura() y FacturaCalcula(). Es probable que hagan lo mismo, cómo elegimos uno u otro?

Posible Solución:

- Está muy claro que si el código es duplicado, debe ser eliminado y sino podemos renombrar métodos o utilizar sobrecarga u otros mecanismos que nos ayuden a entenderlo mejor.

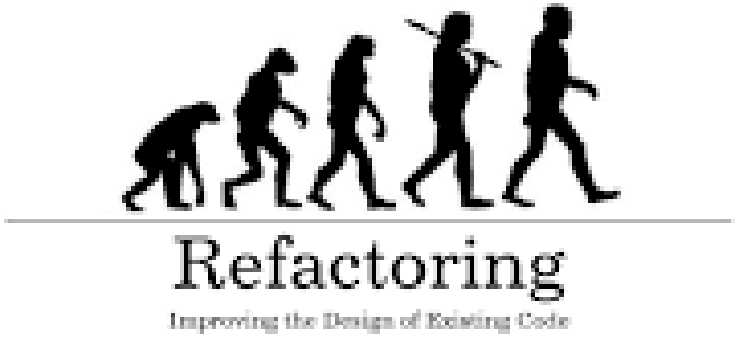
Legado rechazado

- Subclases que usan sólo pocas características de sus superclases.
- Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto.

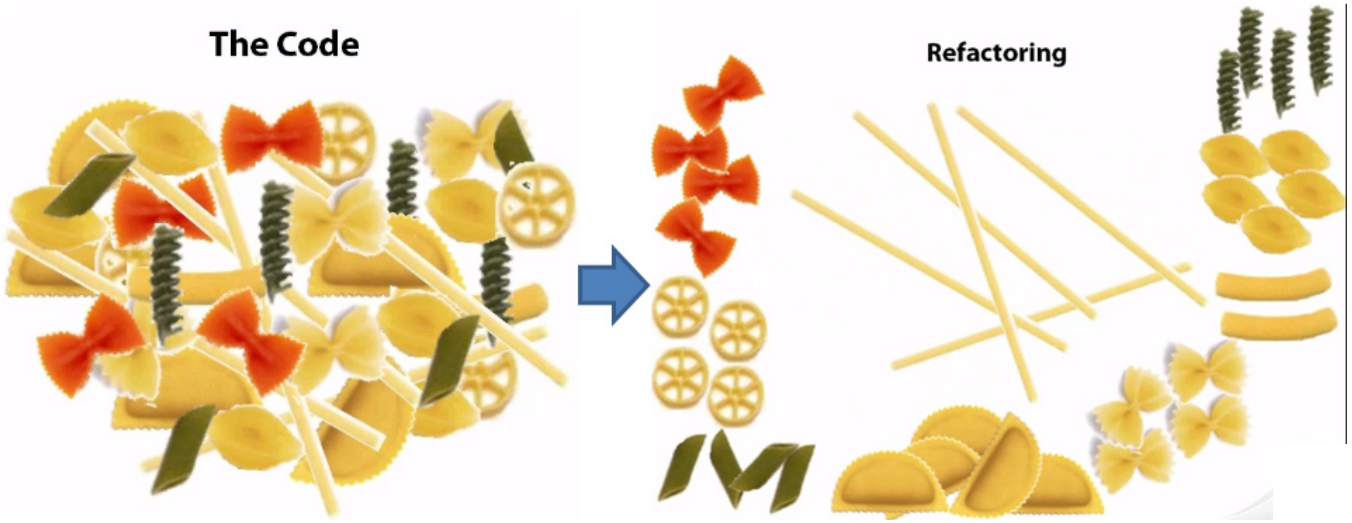
Posible Solución:
- La delegación suele ser la solución a éste tipo de inconvenientes.

Las Refactorizaciones

- Ofrecen técnicas detalladas de transformaciones del código.
- Tienen un formato común: Motivación, Mecanismo y Ejemplo.
- Pueden ser a nivel de un objeto, entre dos objetos, entre grupos de objetos y en gran escala.




Técnicas


The Code
Refactoring

Técnicas de refactorización
Extraer método
Variable temporal en línea
Remplazar temporal con la consulta
Remplazar método con Método-Objeto
Remplazar número mágico con constante simbólica
Extraer clase
Alinear clase

Extraer método

- Tenemos un fragmento de código que es posible agrupar.

```
void imprimirDebe() {
        imprimirEncabezado(); //print details
        Console.Out.WriteLine("Nombre:" + nombre);
        Console.Out.WriteLine("Monto:" + debe());}
```

- Vamos a transformar el fragmento a un método nuevo cuyo nombre va a explicar su propósito.

```
void imprimirDebe() {
        imprimirEncabezado();
        imprimirDetalle(debe()); }
void imprimirDetalle(double valor){
         Console.Out.WriteLine("Nombre:" + nombre);
        Console.Out.WriteLine("Monto:" + valor);}
```

Variable Temporal en línea

- La variable temporal fue asignada una vez con una simple expresión.

```
double precioBase = pedido.precioBase();
return (precioBase > 1000);
```

- Vamos a reemplazar todas las referencias con la expresión.

```
return (pedido.precioBase()>1000);
```

Reemplazar Temporal con la consulta

- La variable temporal guarda el resultado de una expresión.

```
double precioBase = cantidad * valorItem;
 if (precioBase > 1000)
                        return precioBase * 0.95;
else
                        return precioBase * 0.98;
```

- Vamos a extraer la expresión en un método. Remplazar todas las referencias de la variable con el método.

```
if (precioBase() > 1000)
                        return precioBase() * 0.95;
else
                        return precioBase() * 0.98;
double precioBase(){ return cantidad * valorItem;}
```

Reemplazar método con Método-Objeto

- El método se transforma en un objeto de tal modo que todas las variables locales sean campos del mismo, el constructor recibe un objeto y parámetros originales, se copia el método original con el nombre calcular() y se procede a refactorizar.

```
class Pedido{
double precio(int numeroItems){
                        double precioBasePrimario;
                        double precioBaseSecundario;
                       int valorX = numeroItems * delta(); /
      //computo largo...
                  } }
```

```
return new CalculaPrecio(this, numeroItems).calcular();
```

Reemplazar Número Mágico con Constante Simbólica

- Si en un momento hay que cambiar el número, el esfuerzo necesario puede ser enorme

```
double energiaPotencial(double masa, double altura){
        return masa * 9.81 * altura;
}
```

```
double energiaPotencial(double masa, double altura){
        return masa * INTENSIDAD_DE_GRAVEDAD * altura;
}
static const double INTENSIDAD_DE_GRAVEDAD = 9.81;
```

Extraer Clase

- Ocurre cuando una clase hace el trabajo de dos.
- Se debe crear una nueva clase y separar las responsabilidades

Alinear Clase

- Ocurre cuando una clase no esta haciendo mucho
- Se debe hacer el proceso inverso al Extraer Clase

Otras Técnicas de refactorización

- Duplicar datos observados (MVC)
- Encapsular colección
- Reemplazar código de tipo con enumeración
- Reemplazar código de tipo con subclase
- Reemplazar código de tipo con Estado/ Estrategia
- Descomponer condicional
- Remplazar condicional con polimorfismo
- Introducir Objeto Nulo
- Reemplazar método constructor con la factoría
- Reemplazar código de error con la Excepción
- Subir Método
- Subir Campo

Otras Técnicas de refactorización

- Bajar método
- Bajar campo
- Extraer subclase
- Extraer interfaz
- Separar dominio de presentación
- Convertir diseño estructurado a objetos
- Extraer jerarquía…