Antes de refactorizar el codigo se realizan casos de prueba que validen el correcto funcionamiento del sistema, esto provee información sobre el comportamiento actual del sistema.

Los casos de prueba deben cumplir con los siguientes requisitos:

1) Deben ser automáticos de manera tal que se puedan ejecutar todos a la vez con simplemente “hacer clic un botón”

2) Deben ser auto verificables de forma que no tenga que gastar tiempo en la verificación de los resultados. Estos errores deben ser reportados por cada caso de prueba.

3) Deben ejecutarse de manera independiente uno del otro, de forma que los resultados de uno no afecten los resultados del resto.

Los casos de prueba permiten verificar repetida e incrementalmente si los cambios introducidos han alterado el comportamiento observable del programa.

Después de refactorizar, Los resultados obtenidos con los casos de prueba deben ser los mismos para verificar que el comportamiento no se ha visto afectado.

**Refactorizar:**

**Extraer método**: se usa cuando se tiene un fragmento de código que es posible agrupar, donde vamos a transformar el fragmento a un método nuevo donde el nombre va a explicar su propósito

Ej:

Antes de refactorizar:

Private void Calcular()

{

Int a = calculo();

System.out.println(“------calculando ------”);

System.out.println(“el resultado es : ”+ a);

}

Después de refactorizar:

Private void Calcular()

{

Int a = calculo();

Mostrar();

}

Private void Mostrar()

{

System.out.println(“------calculando ------”);

System.out.println(“el resultado es : ”+ a);

}

**Variable Temporal en línea**: se realiza cuando: la variable temporal dificulta el extraer el método, la variable temporal fue asignada una vez con una simple expresión, vamos a reemplazar todas las referencias con la expresión

Ej: antes de refactorizar

Int resultado = Calculo();

return (resultado);

después de refactorizar

return(Calculo());

**Reemplazar temporal con la consulta:** es una de las refactorizaciones vitales antes de Extraer método, la variable temporal esta guardando un resultado de una expresión. Extraer la expresión en un método. Remplazar todas las referencias de la variable con el método.

Ej: antes de refactorizar

Double division = numerado / denominador;

If(división > 10)

{

System.out.prinln(“resultado mayor a 10”);

}

Else

{

System.out.prinln(“resultado menor a 10”);

}

Despues de la refactorizacion:

Private double Division ()

{

Return (numerado / denominador);

}

If(División() > 10)

{

System.out.prinln(“resultado mayor a 10”);

}

Else

{

System.out.prinln(“resultado menor a 10”);

}

**Método con “Método-Objeto”**: si es un método largo pero es difícil aplicar Extraer método por el modo en que se utilizan variables locales, el método se transforma en un objeto de forma que todas las variables locales sean campos del mismo, el constructor recibe objeto y parámetros originales, se copia el método original con nombre calcular() y se procede a refactorizar, lo que facilita aplicar Extraer método

Ej: antes de refactorizar

class Pedido

{

double precio(int numeroItems)

{

double precioBasePrimario;

double precioBaseSecundario;

int valorX = numeroItems \* delta();

//computo largo...

}

}

Después de refactorizar

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

**Extraer Clase**: se usacuando una clase está haciendo trabajo de dos y se debe crear una nueva clase y separar las responsabilidades

**Alinear Clase**: ocurre cuando una clase no está haciendo mucho y se realiza lo opuesto a Extraer Clase.

**Reemplazar numero mágico con constante simbolica:** Un literal tiene significado especial, Si en un momento hay que cambiar el numero, el esfuerzo necesario puede ser enorme

Ej antes de refactorizar:

double energiaPotencial(double masa, double altura)

{

return masa \* 9.81 \* altura;

}

Después de refactorizar:

static const double INTENSIDAD\_DE\_GRAVEDAD = 9.81;

double energiaPotencial(double masa, double altura)

{

return masa \* INTENSIDAD\_DE\_GRAVEDAD \* altura;

}

**Descomponer un condicional**

ejemplo: antes de refactorizar

If(numero > 1 || numero < 10)

{

System.out.prinln(“el numero es: ”+numero);

}

y su refactorización:

If(Numero(numero))

{

System.out.prinln(“el numero es: ”+numero);

}

Observe que aquí hemos cambiado un condicional "complejo" por una llamada a función, invertido la lógica para eliminar la negación (la negación siempre cuesta más entenderla que usar el antónimo), y dos expresiones por dos llamadas a funciones. Es decir, la claridad viene a un costo aparente de tres llamadas a procedimiento --que el compilador debe eliminar *(in-lining).*

**Consolidar una expresión condicional**

Muy parecido al anterior, se une en una sola función booleana una expresión "desparramada" por varios if anidados.

Ej: antes de refactorizar

Private int verificar(int numero)

{

If(numero == 5)

Return 0;

If(numero == 0)

Return 0;

If(numero == 1)

Return 0;

}

Despues de refactorizar

Private int verificar(int numero)

{

If(Revisar(numero))

return 0;

}

**Reemplazar un condicional por polimorfismo**

Esta refactorización me parece muy característica del estilo OO.

Considere el siguiente trozo de código y modifíquelo:

double getSpeed(){   
    switch (\_type) {   
        case CAR:   
            return getMaxSpeed();   
        case TRUCK:   
            return getMaxSpeed() - getLoadFactor()\* \_load;   
        case BUS:   
            return getMaxSpeed() - getSafetyFactor()\*\_risk;   
    }

La solución es definir una clase MotorVehicle, con un método abstracto  getSpeed(). Luego defina la subclases Car, Truck, Bus y redefina ("override") el método getSpeed(0 para cada uno de ellos. El switch desaparece.

**Refactorización: Reemplazar código de tipo por subclases**

(Fowler p. 218)   
Suponga una clase que tiene un atributo con un rango de valores, p. ej.

class Person   
{ ...   
    final private int O;   
    final private int A;   
    final private int B;   
    final private int AB;   
    private int bloodGroup;   
    ...   
}

Es preferible crear una clase BloodGroup y asociarla a Person:

 class BloodGroup   
{...   
    final private int O;   
    final private int A;   
    final private int B;   
    final private int AB;   
    private int bloodGroup;   
    ...   
}

Esto permite más seguridad en el manejo del "tipo" BloodGroup.

**Refactorización: Introducir objetos nulos**

(Fowler p. 260)   
Suponga que repetidas veces revisa a ver si hay un valor nulo, p. ej.

if (customer == null) plan = BillingPlan.basic();   
else plan = customer.getPlan();

Puede ser más claro definir un objeto NullCustomer y aprovechar polimorfismo.

class NullCustomer extends Customer   
{...   
public Plan getPlan() {return BillingPlan.basic()}   
...   
}

Entonces el código original queda como:

plan = customer.getPlan();

¿Es más legible? ¿Es más seguro? Es otro estilo...

**Patrón: Unitario (Singleton)**

Note que conceptualmente no debe haber más de un objeto NullCustomer en un sistema. Este patrón permite garantizar que sólo hay una instancia de una clase, evitando así la proliferación innecesaria de objetos.

El patrón Unitario propone una solución al problema. En un pseudo-Java:

public pseudo-interface Singleton   
{   
    /\* Ojo, en Java, las interfaces no pueden definir atributos, ni cuerpos de métodos \*/   
    private  static int \_instances = 0;   
    private Singleton()  // el(los) constructor(es) de la clase es(son) privado(s).   
    public static getInstance()   
        {   
            if (\_instances == 0) {   
                \_instances++;   
                   return Singleton() //esto tampoco es Java!!   
        }   
            else return this ; //Esto tampoco es Java!!

class NullCustomer extends Customer implements Singleton   
{...

public static getInstance()   
    {super //esto casi es Java...   
    }   
...   
public Plan getPlan() {return BillingPlan.basic()}   
...   
}

En general, el patrón requiere una forma de asegurar que una clase tenga a lo sumo una instancia. Para ello, Gamma et al proponen el juego del "atributo" que cuenta el número de instancias con el cuerpo del método de creación.

M. Grand propone la siguiente elegante implementación, para Java:

public class NullClient implements Client{   
    private static NullClient instance = new NullClient();

    /\* Se necesita definir este constructor vacío para evitar que el compilador de Java genere un constructor público por defecto   
    \*/   
private NullClient(){}

/\*Retornar una referencia al único objeto de esta clase\*/   
public static NullClient getInstance(){   
    return instance} //getInstance

/\* Métodos reescritos de Client \*/   
public Plan getPlan() {return BillingPlan.basic()}   
...   
}//NullClient

Hay varios errores si intentamos implementar la solución propuesta en Java. Uno de los más sutiles tiene que ver con que si se deja de referenciar un objeto de la clase, la clase puede ser recolectada como basura --esto puede traer fallas muy curiosas.

M. Grand propone la siguiente solución para evitar una recolección errada:

public class ObjectPreserver implements Runnable{   
    /\*This keeps this class and everything it references from being garbage collected   
    \*/   
private static ObjectPreserver lifeline = new ObjectPreserver();   
/\* Since this class won be garbage collected, neither will this HashSet or the object it references   
\*/   
private static HashSet protectedSet = new HashSet();

private ObjectPreserver(){   
    new Thread(this).start();   
} //constructor

public void run() {   
    try {   
        wait();   
    } catch (InterruptedException e){   
    } // try   
    } //run

/\*\*   
    \* Garbage collection of objects passed to this method will be prevented until they are passed to the unpreservedObject method   
\*/   
public static void preserveObject(Object o){   
    protectedSet.add(o);   
} //preserveObject

/\* Objects passed to this method lose the protection that the preserveObject method gave them from garbage collection   
\*/   
public static void unpreserveObject(Object o){   
    protectedSet.remove(o);   
} //class ObjectPreserver

Nótese también que el patrón puede verse como un caso especial de *Reemplazar un condicional por polimorfismo.*

**Patrón: Estado**

[\*\*Falta desarrollar]

**Refactorización: Reemplazar código de tipo con Estado/Estrategia**

(Fowler, p. 227).

 Este patrón de refactorización es similar a *Reemplazar código de tipo por subclases*. Este caso es un poco más delicado, pues en el otro patrón un objeto no puede cambiar de tipo durante su ciclo de vida: en este caso sí. El punto es que cómo el ente cambia su *comportamiento* (por ej. una persona deja de portarse como Ingeniero y empieza a comportarse como Gerente), es conveniente separar los aspectos fijos del ente (por ejemplo nombre, ci) de los que dependan del estado.

El reemplazo utilizando cambio en la estrategia suele ser más conveniente si lo que se cambia tiene un fuerte caracter algorítmico.

**Refactorización: Encapsular la colección**

(Fowler p. 208)   
No siempre es conveniente que un método regrese una colección, pues le permite al cliente alterar el contenido de la colección, lo que puede representar una falla de seguridad o encapsulamiento.

En este patrón de r

Otros

* Duplicar datos obervados” (MVC)
* “Encapsular colección”
* “Reemplazar código de tipo con enumeración”
* “Reemplazar código de tipo con subclase”
* “Reemplazar código de tipo con Estado/ Estrategia”
* “Descomponer condicional”
* “Reemplazar condicional con polimorfismo”
* “Introducir Objeto Nulo”
* “Reemplazar método constructor con la factoría”
* “Reemplazar código de error con la Excepción”
* “Subir Método”
* “Subir Campo”
* Se hace un cambio, se ejecutan todos los tests y, si
* todo sigue funcionando, se hace otro pequeño cambio.