Refactorización

Definición

Refactorización (n)

> Cambio realizado a la estructura interna del software
> para hacerlo… más fácil de comprender
> y más fácil de modificar
> sin cambiar su comportamiento observable.

Refactorizar (v)

> Reestructurar el software
> aplicando una secuencia de refactorizaciones.

¿Por qué se "refactoriza" el software?

1. Para mejorar su diseño
   a. Conforme se modifica, el software pierde su estructura.
   b. Eliminar código duplicado simplificar su mantenimiento.
2. Para hacerlo más fácil de entender
   p.ej. La legibilidad del código facilita su mantenimiento
3. Para encontrar errores
   p.ej. Al reorganizar un programa, se pueden apreciar con mayor facilidad las suposiciones que hayamos podido hacer.
4. Para programar más rápido

Al mejorar el diseño del código, mejorar su legibilidad y reducir los errores que se cometen al programar, se mejora la productividad de los programadores.

Ejemplo

Generación de números primos

© Robert C. Martin, "The Craftsman" column,
Software Development magazine, julio-septiembre 2002

Para conseguir un trabajo nos plantean el siguiente problema…

Implementar una clase que sirva para calcular todos los números primos de 1 a N utilizando la criba de Eratóstenes

Una primera solución

Necesitamos crear un método que reciba como parámetro un valor máximo y devuelva como resultado un vector con los números primos.

Para intentar lucirnos, escribimos…

```
/**
 * Clase para generar todos los números primos de 1 hasta
 * un número máximo especificado por el usuario. Como
 * algoritmo se utiliza la criba de Eratóstenes.
 * <p>
 * Eratóstenes de Cirene (276 a.C., Cirene, Libia – 194
 * a.C., Alejandría, Egipto) fue el primer hombre que
 * calculó la circunferencia de la Tierra. También
 * se le conoce por su trabajo con calendarios que ya
 * incluían años bisiestos y por dirigir la mítica
 * biblioteca de Alejandría.
 * <p>
 * El algoritmo es bastante simple: Dado un vector de
 * enteros empezando en 2, se tachan todos los múltiplos
 * de 2. A continuación, se encuentra el siguiente
 * entero no tachado y se tachan todos sus múltiplos. El
 * proceso se repite hasta que se pasa de la raíz cuadrada
 * del valor máximo. Todos los números que queden sin
 * tachar son números primos.
 *
 * @author Fernando Berzal
 * @version 1.0 Enero'2005 (FB)
 */
```

El código lo escribimos todo en un único (y extenso) método, que procuramos comentar correctamente:

```
public class Criba
{
  /**
   * Generar números primos de 1 a max
   * @param max es el valor máximo
   * @return Vector de números primos
   */
   public static int[] generarPrimos (int max)
   {
     int i,j;

     if (max >= 2) {

        // Declaraciones
        int dim = max + 1; // Tamaño del array
        boolean[] esPrimo = new boolean[dim];

        // Inicializar el array
        for (i=0; i<dim; i++)
            esPrimo[i] = true;

        // Eliminar el 0 y el 1, que no son primos
        esPrimo[0] = esPrimo[1] = false;

        // Criba
        for (i=2; i<Math.sqrt(dim)+1; i++) {
            if (esPrimo[i]) {
               // Eliminar los múltiplos de i
               for (j=2*i; j<dim; j+=i)
                   esPrimo[j] = false;
            }
        }

        // ¿Cuántos primos hay?
        int cuenta = 0;

        for (i=0; i<dim; i++) {
            if (esPrimo[i])
               cuenta++;
        }
```

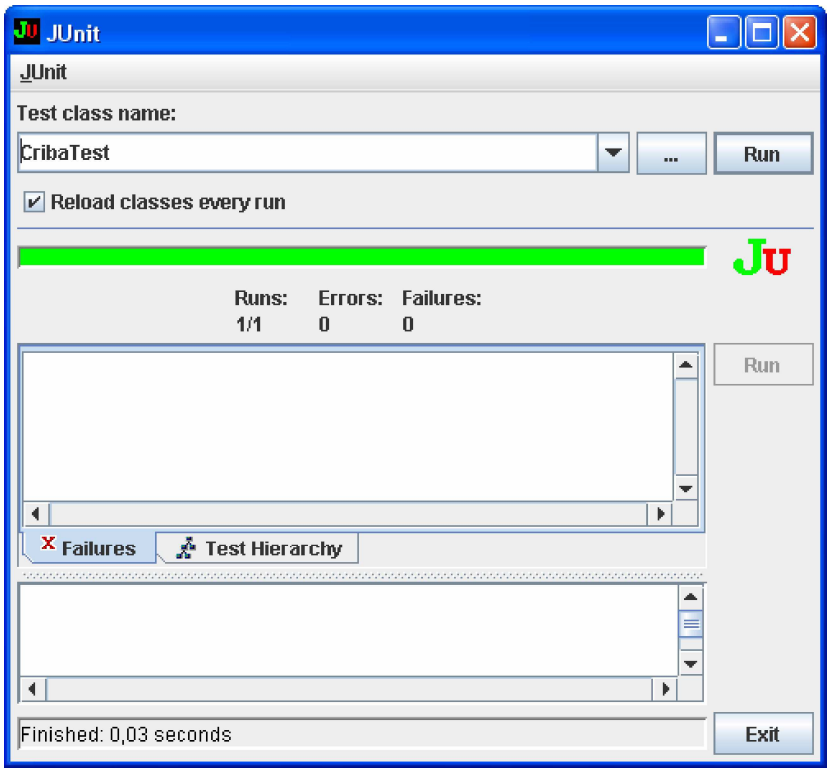
```
            // Rellenar el vector de números primos
            int[] primos = new int[cuenta];

            for (i=0, j=0; i<dim; i++) {
                if (esPrimo[i])
                   primos[j++] = i;
            }

            return primos;

        } else { // max < 2

            return new int[0];  // Vector vacío
        }
    }
}
```

Para comprobar que nuestro código funciona bien, decidimos probarlo con un programa que incluye distintos casos de prueba (para lo que usaremos la herramienta **JUnit**, disponible en http://www.junit.org/):

```
import junit.framework.*;  // JUnit
import java.util.*;

// Clase con casos de prueba para Criba

public class CribaTest extends TestCase
{
    // Programa principal (usa un componente de JUnit)

    public static void main(String args[])
    {
      junit.swingui.TestRunner.main (
        new String[] {"CribaTest"});
    }

    // Constructor

    public CribaTest (String nombre)
    {
      super(nombre);
    }
```

```
    // Casos de prueba

    public void testPrimos()
    {
      int[] nullArray = Criba.generarPrimos(0);
      assertEquals(nullArray.length, 0);

      int[] minArray = Criba.generarPrimos(2);
      assertEquals(minArray.length, 1);
      assertEquals(minArray[0], 2);

      int[] threeArray = Criba.generarPrimos(3);
      assertEquals(threeArray.length, 2);
      assertEquals(threeArray[0], 2);
      assertEquals(threeArray[1], 3);

      int[] centArray = Criba.generarPrimos(100);
      assertEquals(centArray.length, 25);
      assertEquals(centArray[24], 97);
    }
}
```

Al ejecutar los casos de prueba, conseguimos tener ciertas garantías de que el programa funciona correctamente:



Después de eso, vamos orgullosos a enseñar nuestro programa y… un programador no dice que, si queremos el trabajo, mejor no le enseñemos eso al jefe de proyecto (que no tiene demasiada paciencia)

Veamos qué cosas hemos de mejorar…

Primeras mejoras

Parece evidente que nuestro método `generarPrimos` realiza tres funciones diferentes, por lo que de `generarPrimos` extraemos tres métodos diferentes. Además, buscamos un nombre más adecuado para la clase y eliminamos todos los comentarios innecesarios.

```
/**
 * Esta clase genera todos los números primos de 1 hasta un
 * número máximo especificado por el usuario utilizando la
 * criba de Eratóstenes
 * <p>
 * Dado un vector de enteros empezando en 2, se tachan todos
 * los múltiplos de 2. A continuación, se encuentra el
 * siguiente entero no tachado y se tachan sus múltiplos.
 * Cuando se llega a la raíz cuadrada del valor máximo, los
 * números que queden sin tachar son los números primos
 *
 * @author Fernando Berzal
 * @version 2.0 Enero'2005 (FB)
 */

public class GeneradorDePrimos
{
  private static int dim;
  private static boolean esPrimo[];
  private static int primos[];

  public static int[] generarPrimos (int max)
  {
    if (max < 2) {
       return new int[0];  // Vector vacío
    } else {
       inicializarCriba(max);
       cribar();
       rellenarPrimos();
       return primos;
    }
  }
```

```
  private static void inicializarCriba (int max)
  {
    int i;

    dim = max + 1;
    esPrimo = new boolean[dim];

    for (i=0; i<dim; i++)
        esPrimo[i] = true;

    esPrimo[0] = esPrimo[1] = false;
  }

  private static void cribar ()
  {
    int i,j;

    for (i=2; i<Math.sqrt(dim)+1; i++) {
        if (esPrimo[i]) {
           // Eliminar los múltiplos de i
           for (j=2*i; j<dim; j+=i)
               esPrimo[j] = false;
        }
    }
  }

  private static void rellenarPrimos ()
  {
    int i, j, cuenta;

    // Contar primos
    cuenta = 0;

    for (i=0; i<dim; i++)
        if (esPrimo[i])
           cuenta++;

    // Rellenar el vector de números primos
    primos = new int[cuenta];

    for (i=0, j=0; i<dim; i++)
        if (esPrimo[i])
           primos[j++] = i;
  }
}
```

Los mismos casos de prueba de antes nos permiten comprobar que, tras la refactorización, el programa sigue funcionando correctamente.

Un segundo intento

El código ha mejorado pero aún es algo más enrevesado de la cuenta:

eliminamos la variable `dim` (nos vale `esPrimo.length`),
elegimos identificadores más adecuados para los métodos y
reorganizamos el interior del método `inicializarCandidatos` (el antiguo `inicializarCriba`).

```
public class GeneradorDePrimos
{
  private static boolean esPrimo[];
  private static int primos[];

  public static int[] generarPrimos (int max)
  {
    if (max < 2) {
       return new int[0];
    } else {
       inicializarCandidatos(max);
       eliminarMultiplos();
       obtenerCandidatosNoEliminados();
       return primos;
    }
  }

  private static void inicializarCandidatos (int max)
  {
    int i;

    esPrimo = new boolean[max+1];

    esPrimo[0] = esPrimo[1] = false;

    for (i=2; i<esPrimo.length; i++)
        esPrimo[i] = true;
  }

  private static void eliminarMultiplos ()
  … // Código del antiguo método cribar()

  private static void obtenerCandidatosNoEliminados ()
  … // Código del antiguo método rellenarPrimos()
}
```

El código resulta más fácil de leer tras la refactorización.

Mejoras adicionales

El bucle anidado de `eliminarMultiplos` podía eliminarse si usamos un método auxiliar para eliminar los múltiplos de un número concreto.

Por otro lado, la raíz cuadrada que aparece en `eliminarMultiplos` no queda muy claro de dónde proviene (en realidad, es el valor máximo que puede tener el menor factor de un número no primo menor o igual que N). Además, el `+1` resulta innecesario.

```
private static void eliminarMultiplos ()
{
  int i;

  for (i=2; i<maxFactor(); i++)
      if (esPrimo[i])
         eliminarMultiplosDe(i);
}

private static int maxFactor ()
{
  return (int) Math.sqrt(esPrimo.length) + 1;
}

private static void eliminarMultiplosDe (int i)
{
  int multiplo;

  for ( multiplo=2*i;
        multiplo<esPrimo.length;
        multiplo+=i)
      esPrimo[multiplo] = false;
}
```

De forma análoga, el método `obtenerCandidatosNoEliminados` tiene dos partes bien definidas, por lo que podemos extraer un método que se limite a contar el número de primos obtenidos…

Hemos ido realizando cambios que mejoran la implementación sin modificar su comportamiento externo (su interfaz), algo que verificamos tras cada refactorización volviendo a ejecutar los casos de prueba.

¿Cuándo hay que refactorizar?

Cuando se está escribiendo nuevo código

Al añadir nueva funcionalidad a un programa (o modificar su funcionalidad existente), puede resultar conveniente refactorizar:

- para que éste resulte más fácil de entender, o
- para simplificar la implementación de las nuevas funciones cuando el diseño no estaba inicialmente pensado para lo que ahora tenemos que hacer.

Cuando se intenta corregir un error

La mayor dificultad de la depuración de programas radica en que hemos de entender exactamente cómo funciona el programa para encontrar el error. Cualquier refactorización que mejore la calidad del código tendrá efectos positivos en la búsqueda del error.

> De hecho, si el error "se coló" en el código es porque no era lo suficientemente claro cuando lo escribimos.

Cuando se revisa el código

Una de las actividades más productivas desde el punto de vista de la calidad del software es la realización de revisiones del código (recorridos e inspecciones). Llevada a su extremo, la programación siempre se realiza por parejas (*pair programming*, una de las técnicas de la programación extrema, XP [eXtreme Programming]).

¿Por qué es importante la refactorización?

Cuando se corrige un error o se añade una nueva función, el valor actual de un programa aumenta. Sin embargo, para que un programa siga teniendo valor, debe ajustarse a nuevas necesidades (mantenerse), que puede que no sepamos prever con antelación. La refactorización, precisamente, facilita la adaptación del código a nuevas necesidades.

¿Qué síntomas indican que debería refactorizar?

El código es más difícil de entender (y, por tanto, de cambiar) cuando:

- usa identificadores mal escogidos,
- incluye fragmentos de código duplicados,
- incluye lógica condicional compleja,
- los métodos usan un número elevado de parámetros,
- incluye fragmentos de código secuencial muy extensos,
- está dividido en módulos enormes (estructura monolítica),
- los módulos en los que se divide no resultan razonables desde el punto de vista lógico (cohesión baja),
- los distintos módulos de un sistema están relacionados con otros muchos módulos de un sistema (acoplamiento fuerte),
- un método accede continuamente a los datos de un objeto de una clase diferente a la clase en la que está definida (posiblemente, el método debería pertenecer a la otra clase),
- una clase incluye variables de instancia que deberían ser variables locales de alguno(s) de sus métodos,
- métodos que realizan funciones análogas se usan de forma diferente (tienen nombres distintos y/o reciben los parámetros en distinto orden),
- un comentario se hace imprescindible para poder entender un fragmento de código (deberíamos re-escribir el código de forma que podamos entender su significado).

  NOTA: Esto no quiere decir que dejemos de comentar el código.