TEMA 4: OPTIMIZACIÓN Y DOCUMENTACIÓN

1.- Introducción

Por documentación se entiende el texto escrito que acompaña a los proyectos de cualquier tipo, en nuestro caso de software. La documentación es un requisito importante en un proyecto comercial, el cliente siempre va a solicitar que se documenten las distintas fases del proyecto.

Podemos distinguir los siguientes tipos de documentación:

- **Documentación de las especificaciones**: este documento tiene por objeto asegurar que tanto el desarrollador como el cliente tienen la misma idea sobre las funcionalidades del sistema. En ingeniería del software está la norma IEEE 830 que recoge recomendaciones para la documentación de requerimientos de software, se indica que las especificaciones deben describir la siguiente documentación:
  - **Introducción**. En ella se definen los fines y los objetivos del software.
  - **Descripción de la información**. Se realiza una descripción detallada del problema, incluyendo el hardware y el software necesario.
  - **Descripción funcional**. Descripción de cada función requerida en el sistema, incluyendo diagramas.
  - **Descripción del comportamiento**. Comportamiento del software ante sucesos externos y controles internos.
  - **Criterios de validación**. Documentación sobre límites de rendimiento, clases de pruebas, respuesta esperada del software , consideraciones especiales.

- **Documentación del diseño**: en la fase de diseño se decide la estructura de datos a utilizar, la forma en la que se van a implementar las distintas estructuras , el contenido de las clases , sus métodos y sus atributos, los objetos a utilizar. Tambien se definen las funciones, sus datos de entrada y salida, que tarea realizan, etc.
- **Documentación del código fuente**: durante la fase de implementación, cuando se está programando, es necesario comentar convenientemente cada una de las partes que tiene el programa. Estos comentarios se incluyen en el código fuente con el objeto de clarificar y explicar cada elemento del programa, se deben comentar las clases , los métodos , las variables y todo elemento que se considere importante.
- **Documentación de usuario final**: es la documentación que se entrega al usuario tanto especializado como no especializado. En ella se describirá cómo utilizar las aplicaciones del proyecto. Este tipo de documento puede ser redactado por cualquier persona , no tiene que ser alguien involucrado en el proyecto.

2.- Documentación del código fuente

Es necesario documentar el código para explicar lo que hace el programa, para que de esta manera todo el equipo de desarrollo sepa lo que está haciendo y por qué.
En un programa bien documentado es mucho más fácil reparar errores y añadirle nuevas funcionalidades para adaptarlo a nuevos escenarios que si carece de documentación.
Hay dos reglas que debemos de tener en cuenta:

- Todos los programas tienen errores y descubrirlos solo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente.
- Todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito.

Normalmente los programas que tienen éxito serán modificados en el futuro, bien por el autor del programa , o por otro programador del equipo. Por esta razón es necesario tener bien documentado el

código, para poder hacer modificaciones de forma sencilla. Esa documentación va a permitir que otro programador ajeno localice enseguida los cambios a realizar.

A la hora de docuementar interesa que se explique lo que hace una clase o un método y por qué y para qué se hace. Así se indicará , de qué se encarga una clase, un paquete , un método o una variable.Cuál es el uso esperado de esa variable o de ese método. Qué algoritmo se utiliza para resolver algún problema.

Para documentar proyectos, normalmente cada lenguaje dispone de su propia herramienta como PHPDoc, Javadoc, etc.

3.- Javadoc en Eclipse

Es la utilidad de Java para extraer y generar documentación directamente del código en formato HTML. La documentación y el código se van a incluir dentro del mismo fichero.

Las recomendaciones sobre los comentarios y la documentación del código fuente son:

3.1.- Los tipos de comentarios para generar la documentación son:

- Comentarios de línea: comienzan con los caracteres "//" y terminan con la línea.
- Comentarios de tipo C: comienzan con los caracteres "/*" , y terminan con los caracteres "*/". Pueden agrupar a varias líneas.
- Comentarios de documentación Javadoc: estos comentarios se colocan entre los delimitadores /**. . . . . */, agrupan varias líneas , y cada línea irá precedida por un *, estos comentarios deben colocarse antes de la declaración de una clase , un campo, un método o un constructor. Dentro de estos delimitadores se podrá escribir etiquetas HTML . Los comentarios Javadoc están formados por dos partes una descripción seguida de un bloque tags.

Ejemplo de documentación:

```
/**
*       <h2> Ejemplo de documentación </h2>
*       Se pueden añadir etiquetas HTML , para <b> mejorar </b> la presentación.
*       Añadimos unas viñetas
*       <ul>
*       <li> Inserción de registros </li>
*       <li> Modificación de listas</li>
*       </ul>
*
*       @autor : Alumno
*       @versión v1.2
*
*/
class prueba {
        . . . . . .
```

3.1.1.- Uso de etiquetas de documentación

Se pueden usar tags para documentar ciertos aspectos concretos como la versión de la clase , el autor , los parámetros utilizados, o los valores devueltos. Las etiquetas de Javadoc van precedidas por @, las más usadas son:

ETIQUETA	DESCRIPCIÓN
@author	Autor de la clase. Solo para las clases
@version	Versión de la clase. Solo para clases
@see	Referencia a otra clase , ya sea del API , del mismo proyecto o de otro. Por ejemplo: @see cadena @see paquete.clase#miembro @see enlace.
@ param	Descripción de parámetro. Una etiqueta por cada parámetro
@return	Descripción de lo que devuelve. Solo si no es void. Podrá describir valores de retorno especiales según las condiciones que se den, dependiendo del tipo de dato.
@throws	Descripción de la excepción que puede propagar. Habrá una etiqueta throws por cada tipo de excepción.
@deprecated	Marca el método como obsoleto. Solo se mantiene por compatibilidad.
@since	Indica el n.º de versión desde la que existe el método.

3.1.2.- Ejemplo de documentación de una clase usando etiquetas:

```
/**
*       <h2> Clase Empleado, se utiliza para crear y leer empleados de una Base de Datos </h2>
*
*       @versión v1.2
*       @author: Alumno
*
*/
public class Empleado{
        /**
        * Atributo Nombre del empleado
        */

        private String nombre;

        /**
        * Atributo apellido del empleado
        */

        private String apellido;

        /**
        * Salario del empleado
        */

        private double salario;
```

```
	/**
	* Constructor con 3 parámetros
	* Crea objetos Empleado, con nombre, apellido y salario
	*@param nombre Nombre del empleado
	* @param apellido Apellido del empleado
	* @param salario Salario del empleado
	*/

	public Empleado(String nombre, String apellido, double salario)
	{
		this.nombre=nombre;
		this.apellido=apellido;
		this.salario=salario;
	}

	// Métodos públicos

	/**
	* Sube el salario al empleado
	*@see Empleado
	* @param subida
	*
	*/

	public void subidasalario (double subida)
	{
		salario=salario+subida;
	}

	//Métodos privados

	/**
	* Comprueba que el nombre no este vacio
	*@return <ul>
	*		<li> true: el nombre es una cadena vacía </li>
	*		<li> false: el nombre no es una cadena vacía </li>
	*	</ul>
	*/

	private boolean comprobar()
	{
		if(nombre.equals(“”))
		{
			return false;
		}
		return true;
	}
}
```

OBSERVACIÓN: Los comentarios de documentación Javadoc deben de colocarse ANTES de las declaraciones de las clases, de los métodos, o de los atributos.

4.- Generar la documentación

La mayor parte de los entornos de desarrollo incluyen un enlace para configurar y ejecutar Javadoc. Para hacerlo desde Eclipse , se abre el menú Project y se elige Generate Javadoc.

Configuración

- En Javadoc **command** se tiene que indicar dónde se encuentra el archivo ejecutable Javadoc , el javadoc.exe . Se pulsa el botón Configure para buscarlo dentro de la carpeta donde se encuentra instalado el JDK . Dentro de esa carpeta se elige la carpeta bin.
- En los dos cuadros inferiores se elegirá el proyecto y las clases a documentar.
- Se selecciona la visibilidad de los elementos que se van a documentar . Con Private se documentarán todos los miembros públicos , privados y protegidos.
- Por último, se indica la carpeta de destino donde se almacenará el código HTML (se recomienda poner un nombre diferente al que viene por defecto)

Se hace clic en next , y en la siguiente ventana se indica el título del documento html que se genera , y se eligen las opciones para la generación de las páginas HTML. Como mínimo se selecciona la barra de navegación y el índice.

EJERCICIO 1

- **Realiza pruebas de documentación en los proyectos de las actividades anteriores.**
- **Añade autor y versión a las clases.**
- **Añade descripción a las clases , los métodos y sus parámetros utilizando etiquetas HTML**
- **Generar la documentación cambiando las distintas opciones.**
- **Realiza las mismas operaciones con el IDE Netbeans**

5.- Refactorización

La refactorización es una técnica que permite la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin que esto suponga alteraciones en su comportamiento externo ; la refactorización persigue mejorar la comprensión del código para facilitar así nuevos desarrollos , la resolución de errores o la adición de alguna funcionalidad al software.

La refactorización tiene como objetivo limpiar el código para que sea más fácil de entender y de modificar , permitiendo una mejor lectura para comprender qué es lo que se está realizando.

¿Qué hace la refactorización?

- Limpiar el código, mejorando la consistencia y la claridad.
- Mantiene el código, no corrige errores ni añade funciones nuevas.
- Facilita la realización de cambios en el código.
- Se obtiene un código limpio y altamente modularizado.

5.1.- Cuándo refactorizar

La refactorización se debe ir haciendo mientras se va desarrollando la aplicación. Debemos de refactorizar cuando observemos las siguientes causas:

- **Código duplicado:** Si detectamos el mismo código en más de un lugar, se debe de buscar la forma de extraerlo y unificarlo.
- **Métodos muy largos.** Un método muy largo indica que está realizando tareas que deberían ser realizada por otros métodos. Se debe de identificar y descomponer el método en otros más pequeños. En la programación orientada a objeto, cuanto más corto es un método más fácil es reutilizarlo.
- **Clases muy grandes**. Si una clase intenta resolver muchos problemas , tendremos una clase con muchos métodos, atributos o incluso instancias. Hay que intentar hacer clases más pequeñas , de forma que cada una trate con un conjunto pequeño de responsabilidades bien definidas.
- **Lista de parámetros extensa** . En la programación orientada a objeto no se suelen pasar muchos parámetros a los métodos, sino solo aquellos necesarios para que el objeto involucrado consiga lo necesario. Tener muchos parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros, y pasar ese objeto como argumento en lugar de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.
- **Cambio divergente:** una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre si, a lo mejor conviene eliminar la clase .
- **Cirugía o tiro pistola:** este síntoma se presenta cuando después de un cambio en una determinada clase, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- **Envidia de funcionalidad:** se observa este síntoma cuando tenemos un método que utiliza más cantidad de elementos de otra clase que de la suya propia . Se suele solucionar el problema pasando el método a la clase cuyos elementos utiliza más.
- **Clase de solo datos.** Clases que solo tienen atributos y métodos de acceso a ellos (“get” y “set”). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.
- **Legado rechazado:** este síntoma lo encontramos en subclase que utilizan solo unas 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 jerarquia de clases no es correcto. La delegación suele ser la solución a este tipo de inconvenientes.

Ventajas de la refactorización:

.- Mantenimiento del diseño del sistema.
.- Incremento de facilidad de lectura y comprensión del código fuente.
.- Detección temprana de fallos.
.- Aumento en la velocidad en la que se programa.

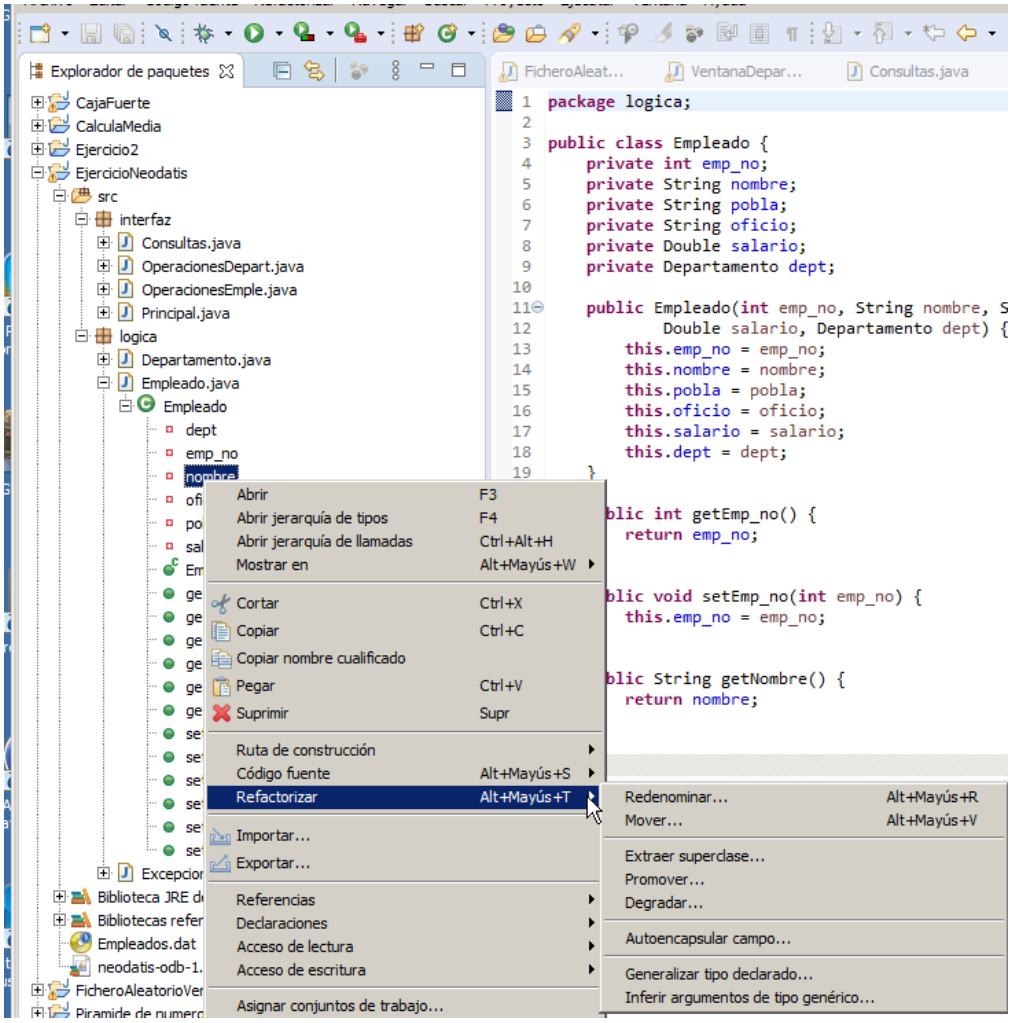
Inconvenientes de la refactorización

.- Las bases de datos. Un cambio de base de datos es muy costoso pues los sistemas están muy acoplados a las bases de datos, y sería necesario una migración tanto de estructura como de datos.
.- Los cambios de interfaces.

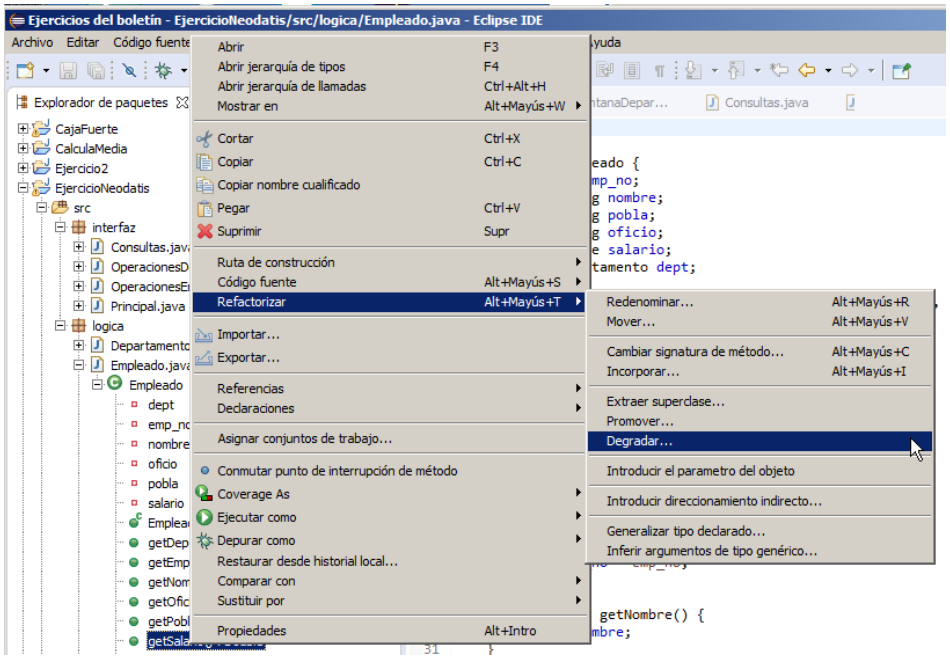
5.2.- Refactorización en Eclipse

Eclipse tiene diversos métodos de refactorización . Dependiendo de dónde invoquemos a la refactorización tendremos un menú contextual u otro con sus diferentes opciones de refactorización.

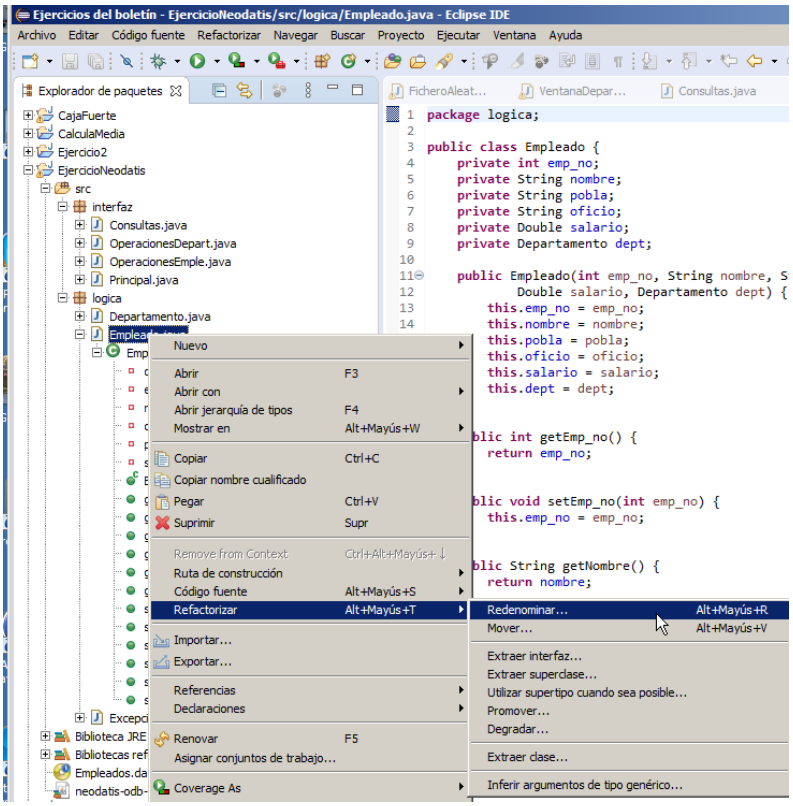
Se invoca desde un atributos



Se invoca desde un método



Se invoca desde una clase



5.3.- Métodos de refactorización

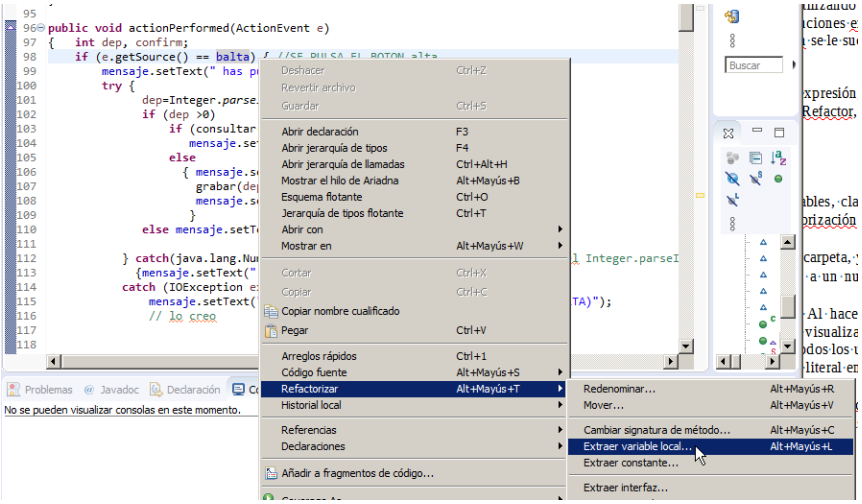
Los métodos de refactorización son las practicas para refactorizar el código, utilizando las herramientas podremos plantear casos para refactorizar y se mostrarán las posibles soluciones enlas que podremos ver el antes y el después de refactorizar. A los métodos de refactorización se le suelen llamar “patrones de refactorización”.

Para refactorizar se selecciona el elemento (puede ser una clase , una variable, una expresión, un bloque de instrucciones, un método, etc.) → botón derecho del ratón , se selecciona Refactor, y a continuación seleccionamos el método de refactorización.

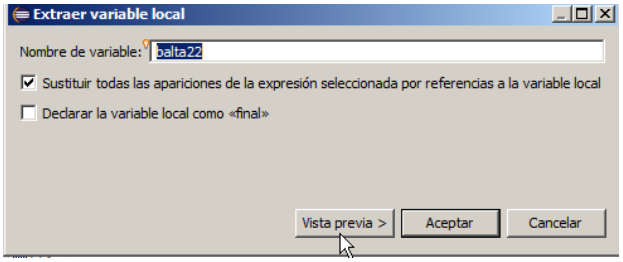
Algunos de los métodos más comunes son:

- **Rename.** Es una de las opciones más utilizadas. Cambia el nombre de variables, clases, métodos, paquetes, directorios y casi cualquier identificador Java. Tras la refactorización , se modifican las referencias a ese identificador.
- **Move.** Mueve una clase de un paquete a otro, se mueve el archivo .java a la carpeta, y se cambian todas las referencias. Tambien se puede arrastrar y soltar una clase a un nuevo paquete, se realiza una refactorización automática.
- **Extract Constant**. Convierte un número o cadena literal en una constante. Al hacer la refactorización se mostrará dónde se van a producir los cambios, y se puede visualizar el estado antes de refactorizar y después de refactorizar. Tras la refactorización , todos los usos del literal se sustituyen por esa constante. El objetivo es modificar el valor del literal en un único lugar.

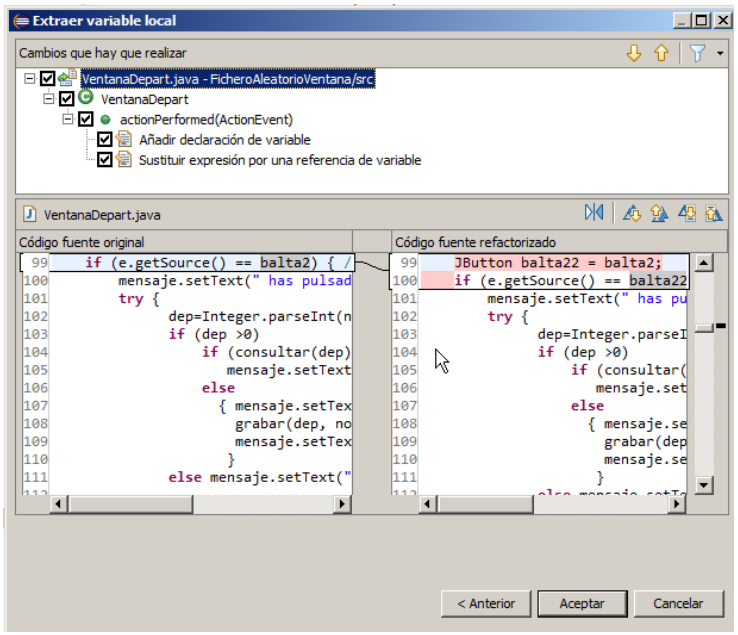
- **Extract Local** Variable. Asignar una expresión a variable local. Tras la refactorización , cualquier referencia a la expresión en el ámbito local se sustituye por la variable. La misma expresión en otro método no se modifica.



En la siguiente imagen podemos ver los cambios que se producirán al extraer una variable local , se muestran dónde se realizan los cambios , y el detalle antes y después.



Hacemos clic en el botón vista previa



- **Convert Local Variable to Field**. Convierte una variable local en un atributo privado de la clase. Tras la refactorización , todos los usos de la variable local se sustituye por ese atributo.

****

EJERCICIO 2

- Abre el proyecto FicheroAleatorioVentana y realiza los siguientes cambios en la clase VentanaDepart.java

- Extrae una variable local llamada existedepart de la cadena : "DEPARTAMENTO EXISTE"
- Extrae una constante local llamada NOEXISTEDEPART de la cadena: "DEPARTAMENTO NO EXISTE".
- Convierte la variable local creada en un atributo de la clase.
- Extrae un variable local llamada depar_error de la cadena: "DEPARTAMENTO ERRÓNEO" , y conviértela en un atributo de la clase.

****

- **Extract Method** . Nos permite seleccionar un bloque de código y convertirlo en un método. El bloque de código no debe dejar llaves abiertas. Eclipse ajustará automáticamente los parámetros y el retorno de la función . Esto es muy útil para utilizarlo cuando se crean métodos muy largos, que se podrán dividir en varios métodos. También es muy útil extraer un método cuando se tiene un grupo de instrucciones que se repiten varias veces.

Al extraer el método hay que indicar el modificador de acceso: público, protegido , privado o sin modificador.

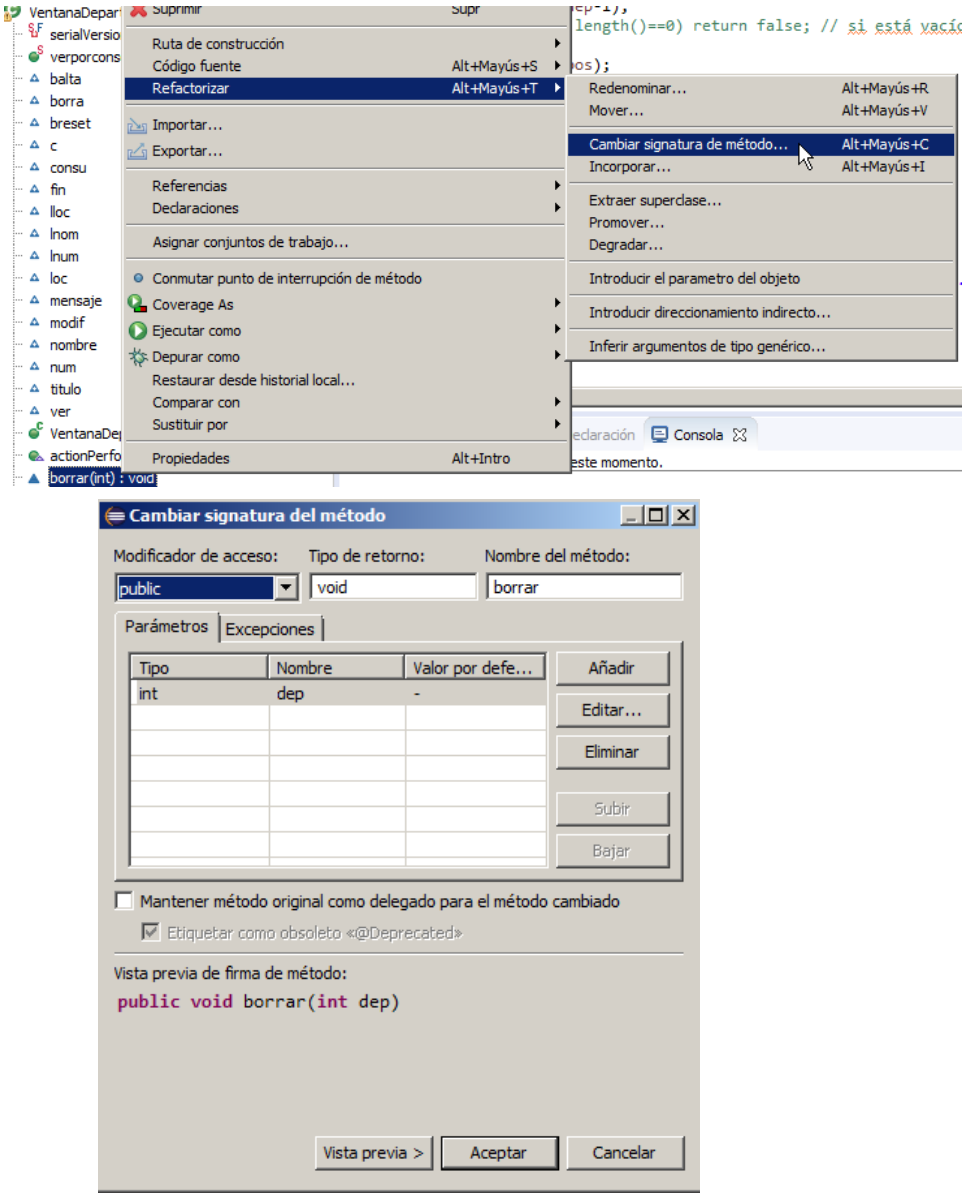
EJERCICIO 3

Extraer métodos dentro de la clase VentanaDepart.java

- Dentro del método **public void actionPerformed(ActionEvent e)** , extrae varios métodos, uno para cada una de estas operaciones ; insertar departamento, consultar, borrar y modificar.
- Extraer los métodos de las instrucciones que van dentro de los distintos ***if (e.getSource())***, que preguntan por balta, consu, borra y modif, llamarlos altadepart, consuldepart, borradepart y modifdepart.

****

- Change Method Signature. Este método permite cambiar la firma de un método; es decir, ***el nombre del método y los parámetros que tiene.*** De forma automática se actualizarán todas las dependencias y llamadas al método dentro del proyecto.



En la ventana para cambiar la firma de un método , se indicará el nuevo nombre del método, el tipo de dato que devuelve, los nuevos parámetros, se pueden editar los parámetros y cambiarlos, o también asignar un valor por defecto.

OBSERVACIÓN: ***Si al refactorizar cambiamos el tipo de datos que devuelve el método , aparecerán errores de compilación, por lo que debemos modificarlo manualmente.***

****

EJERCICIO 4

- **Prueba a cambiar la firma de los métodos creados anteriormente.**
- **Añade un parámetro de tipo String, valor por defecto "PRUEBA"**
- **Cambia el tipo de dato devuelto en los métodos, haz que retornen un entero.**
- **Comprueba y corrige los errores de retorno.**

****

- **Inline** . Nos permite ajustar una referencia a una variable o método con la línea en la que se utiliza y conseguir así una única línea de código. Cuando se utiliza , se sustituye la referencia a la variable o método con el valor asignado a la variable o la aplicación del método, respectivamente.

Ejemplo: dentro de la clase FicheroAleatorioVentana nos encontramos con la siguiente declaración.

```
File fichero = new File("AleatorioDep.dat");
RandomAccessFile file;
file= new RandomAccessFile(fichero, "rw");
```

Posicionamos el cursor en la referencia al método o variable, en este caso la variable "fichero" . Seleccionamos la opción "Inline" y el resultado es:

```
RandomAccessFile file;
file= new RandomAccessFile (new File("AleatorioDep.dat"), "rw");
```

- **Member Type to Top Level.** Convierte una clase anidada en una clase de nivel superior con su propio archivo de java. Si la clase es estática , la refactorización es inmediata. Si no es estática nos pide un nombre para declarar el nombre de la clase que mantendrá la referencia con la clase inicial.

****

EJERCICIO 5

- Crea esta clase anidada dentro de la clase FicheroAleatorioVentana

```
class claseAnidada{
        void entrada(){
                System.out.println( "Método entrada. " );
        }

        String salida (int d) {
                System.out.println( "Salida . " );
                return "Salida el " + d;
        }
} // fin de clase anidada
```

- Crea un objeto de esta clase y llama a los métodos dentro del método verporconsola. Añade este código en el método verporconsola.

```
FicheroAleatorioVentana fa=new FicheroAleatorioVentana();
claseAnidada ej = new claseAnidada();
```

```
ej.entrada();
System.out.println( " Llamo a Salida: "+ej.salida(10));
```

- Prueba la ejecución (pulsa el botón "Ver por consola" de la ventana de ejecución.
- Convertir la clase anidada en una de nivel superior con su archivo asociado (Move Type to new file). Observa que se ha creado una nueva clase con su fichero java.
- Prueba la ejecución.

****

- **Extract Interface.** Este método de refactorización nos permite escoger los métodos de una clase para crear una Interface. Una Interface es una especie de plantilla que define los métodos acerca de lo que puede o no hacer una clase. La Interface define los métodos pero no los desarrolla. Serán las clases que implementen las Interfaces quien desarrolle los métodos.

Ejemplo: se define la interfaz Animal , y los métodos comer y respirar porque todos los animales comen y respiran. Sin embargo cada animal va a comer y respirar de una forma diferente, por eso en cada animal se deben desarrollar esos métodos.

Clase Interface	Clase que implementa la Interface

interface Animal
{
void comer();
int respirar();
}
	class Perro implements Animal{
	public void comer(){
	//se define como come el perro
	}
	public int respirar(){
	//se define como respira el perro
	}
	}

****

EJERCICIO 6

- Crea la interface InterfaceVentanaDepart que contenga los métodos creados en el ejercicio 3. Al crear la interface solo se pueden añadir los métodos públicos. Realiza los cambios necesarios que se necesitan para crearla.

- Observa en el explorador de paquete cómo se visualiza la clase creada, observa también el cambio producido en la declaración de la clase VentanaDepart.

****

- **Extract Superclass.** Este método permite extraer una superclase . Si la clase ya utilizaba una superclase, la recién creada pasará a ser su superclase. Se pueden seleccionar los métodos y atributos que forman parte de la superclase. En la superclase, los métodos están actualmente allí, así que si hay referencias a campos de clase original, habrá fallos de compilación.

****

EJERCICIO 7

- A partir de la clase VentanaDepart crea la superclase "SuperclaseDepart", que incluya solo los métodos de grabar y visualizar.
- Observa la clase generada y los constructores generados. Observa que la declaración de la clase *VentanaDepart* ha cambiado, ahora es **extends** de *SuperclaseDepart*.
- Observa los errores generados, dos se producen por hacer referencias a campos de la clase original (nombre y loc), el tercero por definir un objeto de la clase **claseAnidada** con el parámetro "**this**" de la clase actual.
- Soluciona los errores moviendo las declaraciones de los campos a la superclase, y **casteando** el argumento del objeto de la clase *claseAnidada* a *InterfaceVentanaDepart*

****

- **Conver Anonymous Class to Nested.** Este métdo de refactorización permite convertir una clase anónima a una clase anidada de la clase que la contiene. *Una clase anónima es una clase sin nombre de la que solo se crea un único objeto*, de esta clase no se pueden definir constructores . Se utilizan con frecuencia cuando se crean ventanas, para gestionar los eventos de los distintos componenetes de la interfaces gráfica.

Una clase anónima se puede definir de las siguientes formas:

1. Se utiliza la palabra new seguida de la definición de la clase anónima entre llaves {……}
2. Palabra new seguida del nombre de la clase de la que hereda (sin extends) y la definición de la clase entre llaves {…….}
3. Palabra new seguida del nombre de la interface (sin implements) y la definición de la clase anónima entre llaves {…….}

Ejemplo de clase anónima que implementa el método actionPerformed, que es el único método de la interface ActionListener (esta clase detecta los eventos de acción, es decir , cuando se pulsa un botón , o cuando se pulsa INTRO, o cuando se cambia el elemento de una lista desplegable o se elige un elemento de menú).

```
JButton btnDepart = new Jbutton("Operaciones Departamentos");
btnDepart.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent arg0)
        {
                OperacionesDepart opDepart=new OperacionesDepart();
                opDepart.setVisible(true);
        }
}
```

Se convierte la clase anónima **new ActionListener** a clase anidada llamada **OperacionesDep** dentro de la clase donde está. Nos posicionamos sobre la clase anónima , botón derecho → Refactor → *Convert Anonymous Class to Nested Class*. Se elige el modificador de acceso y en *Type name* se escribe el nombre. Si se selecciona final se indica que no se podrán crear clases derivadas de esta clase.

Después de la refactorización la nueva clase quedará de la siguiente forma:

```
private final class OperacionesDep implements ActionListener
{
        public void actionPerformed(ActionEvent arg0)
        {
                OperacionesDepart opDepart=new OperacionesDepart();
                opDepart.setVisible(true);
        }
}
```

La ejecución del evento quedará de la siguiente forma:

```
JButton btnDepart = new JButton ("Operaciones Departamentos");
btnDepart.addActionListener (new OperacionesDep());
```

OBSERVACIÓN: @Override: informa al compilador que el elemento está pensado para sobrescribir a un elemento declarado en una superclase.

REALIZA LOS EJERCICIOS DE REFACTORIZACIÓN CON NETBEANS