CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

ENTORNOS DE DESARROLLO.

TEMA 10. DOCUMENTACIÓN Y FACTORIZACIÓN

CENTRO:	*CENTRO ESPECÍFICO DE EDUCACIÓN A DISTANCIA DE LA COMUNIDAD VALENCIANA (CEEDCV)*
WEB:	*http://www.ceedcv.es*
AÑO ACADÉMICO:	*2011/2012*

PROFESOR:	*FRANCISCO ALDARIAS RAYA*
EMAIL:	*paco.aldarias@ceedcv.es*

MODULO:	*ENTORNOS DE DESARROLLO.*
CICLO FORMATIVO:	*CFGS: DESARROLLO DE APLICACIONES WEB*
DEPARTAMENTO:	*INFORMÁTICA*
CURSO:	*1º*

Versión 1. 2011-2012

Changeset: 36:4b20a666dcdb
Ultima actualización 2 de septiembre de 2012

FRANCISCO ALDARIAS RAYA
paco.aldarias@ceedcv.es
Departamento de Informática.
Centro Específico de Educación a Distancia de la Comunidad Valenciana. (CEED)
http://www.ceedcv.es
En Valencia (España).
Documento realizado con software libre: linux debian, LaTeX.

Reconocimientos a Javier Martín Juan y a Angel Vidal Estibalez por su colaboración con estos apuntes.

Índice

1. DOCUMENTACIÓN

1.1. Definición

Se entiende por documentación todo lo concerniente a la documentación del propio desarrollo del software y de la gestión del proyecto, pasando por modelaciones (UML),diagramas de casos de uso, pruebas, manuales de usuario, manuales técnicos, etc; todo con el propósito de eventuales correcciones, usabilidad, mantenimiento futuro y ampliaciones al sistema.

1.2. Tipos de documentación

La documentación de un programa puede ser :

1.- **Interna**. La que lleva el código fuente.

2.- **Externa**. La que está fuera del código fuente.

2. DOCUMENTACIÓN EXTERNA

La documentación asociada a un programa, pero no contenida en él se denomina externa. Una buena documentación externa debe incluir al menos:

1.- Análisis de requisitos.

2.- Diseño del programa.

3.- Diseño de la Base de Datos Relacional. Diagramas Entidad Relación.

4.- Diagramas UML.

5.- Manual de instalación.

6.- Manual del usuario

7.- Historia del desarrollo del programa

8.- Modificaciones posteriores.

3. DOCUMENTAR UN PROYECTO

3.1. Guía básica de documentación

La guía para documentar un proyecto de desarrollo de software orientado a objetos sería:

1.- Alcance del Sistema

1.- Planteamiento del problema
2.- Justificación
3.- Objetivos generales y específicos
4.- Desarrollo con proceso unificado

2.- Análisis y especificación de requisitos

1.- identificación y descripción de pasos
2.- especificación de requisitos
    1.- objetivos del sistema
    2.- requisitos de información
    3.- restricciones del sistema
3.- requisitos funcionales
    1.- **diagramas de casos de uso**
    2.- definición de actores
    3.- documentación de los casos de uso
4.- requisitos no funcionales

3.- Diseño del sistema XXX

1.- **diagrama de clases**
2.- modelo entidad relación
3.- modelo relacional
4.- diccionario de datos
5.- **diagrama de secuencias**
6.- **diagrama de actividades**

4.- Implementación

1.- Arquitectura del sistema
2.- Implementación con estándares
3.- Arquitectura de desarrollo
4.- estándar de codificación
5.- sistema de control de versiones
6.- **diagrama de despliegue**

5.- Pruebas

1.- Planificación
2.- desarrollo de las pruebas

6.- Resultados

1.- conclusiones
2.- trabajos futuros
3.- anexos
    1.- manual de usuario
    2.- manual de instalación

3.2. Cuanto documentar

La buena documentación es esencial para un proyecto software. Sin ella un equipo se perderá en un mar de código. Por otro parte, demasiada documentación del tipo equivoca es lo peor; porque entonces distrae e induce error.

La documentación debe ser creada, pero prudentemente. Un complejo protocolo de comunicación debe ser documentado. Un complejo esquema relacional necesita ser documentado. Un complejo framework reusable necesita ser documentado.

Sin embargo, ninguna de este tipo de cosas necesita cientos de páginas de UML. La documentación de software debe ser corta y concreta. El valor de una documentación software es inversamente proporcional a su tamaño.

Para un proyecto de 12 personas trabajando sobre un proyecto de miles de líneas java, debería tener un total de 25 a 200 páginas de documentación, con un preferencia a ser pequeña, un diagrama ER, una página o dos de como construir el sistema, instrucciones de prueba, etc.

Se puede poner esta documentación en un wiki, o alguna herramienta de colaboración para que cualquiera del equipo pueda acceder para poner sus pantalla y buscarlas. y cualquiera pueda cambiarla cuando se a necesario.

Lleva mucho trabajo hacer un pequeño documento, pero ese trabajo vale la pena. Las personas leerán un pequeño documento, pero no 1000 páginas de un tomo.

Javadoc es una excelente herramienta de java. Debe de usarse. Pero es poco y focalizado. Permite describir funciones que otro usarán. Se debería escribir con cuidado y debería contener suficiente información para ayudar al usuaria a entenderlo.

4. MÉTRICA

Metodología para documentar grandes proyectos de la Administración del Estado Español.

Métrica es una metodología de planificación, desarrollo y mantenimiento de sistemas de información. Promovida por el Ministerio de Administraciones Públicas del Gobierno de España para la sistematización de actividades del ciclo de vida de los proyectos software en el ámbito de las administraciones públicas.

Métrica está orientada al proceso y, en su versión 3, estos procesos son:

- Planificación de Sistemas de Información (PSI).
- Desarrollo de Sistemas de Información (DSI). Debido a su complejidad, está a su vez dividido en cinco procesos:
  - ⋆ Estudio de Viabilidad del Sistema (EVS).
  - ⋆ Análisis del Sistema de Información (ASI).
  - ⋆ Diseño del Sistema de Información (DSI).
  - ⋆ Construcción del Sistema de Información (CSI).
  - ⋆ Implantación y Aceptación del Sistema (IAS).
- Mantenimiento de Sistemas de Información (MSI).

5. RUP. PROCESO UNIFICADO DE RATIONAL

Metodología para documentar grandes proyectos usando UML.

El Proceso Unificado de Rational (Rational Unified Process en inglés, habitualmente resumido como RUP) es un proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos.

El RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables al contexto y necesidades de cada organización.

También se conoce por este nombre al software desarrollado por Rational, hoy propiedad de IBM, el cual incluye información entrelazada de diversos artefactos y descripciones de las diversas actividades. Está incluido en el Rational Method Composer (RMC), que permite la personalización de acuerdo con las necesidades.

6. READYSET. DOCUMENTACIÓN CON PLANTILLAS.

Mediante ReadySET podemos utilizar plantillas donde se documentará cada parte del proyecto informático.

`http://readyset.tigris.org/es/index.html`

Las partes de que consta son:

1.- Planeamiento del proyecto.

2.- Requerimientos y Especificaciones.

3.- Arquitectura y Diseño

4.- Implementación y Pruebas

5.- Entrega e Instalación.

6.- Operaciones y Soporte

7.- Continuidad o Final.

7. DOCUMENTACIÓN INTERNA

Una vez que se genera el código fuente, la función de un módulo debe resultar clara sin necesidad de referirse a ninguna especificación del diseño. En otras palabras, el código debe ser comprensible

El buen código se comenta solo. No es necesario comentar todas las líneas de código.

- **Documentación del código.**

  La documentación interna del código comienza con la elección de los nombres (variables y etiquetas), continúa con la localización y composición de los comentarios y termina con la organización visual del programa.

  La elección de nombres de identificadores significativos es crucial para la legibilidad. Los lenguajes que limitan la longitud de los nombres de las variables o de las etiquetas a unos pocos caracteres, implícitamente limitan la comprensión.

  La posibilidad de expresar comentarios en lenguaje natural como parte del listado del código fuente es algo que aparece en todos los lenguajes de propósito general. Los comentarios pueden resultar una clara guía durante la última fase de la ingeniería del software, el mantenimiento.

- **Comentario de prólogo.**

  Al principio de cada módulo debe haber un comentario de prólogo con el siguiente formato:

  1.- Una sentencia que indique la función del módulo .
  2.- Una descripción de la interfaz
  - un ejemplo de ”secuencia de llamada”
  - una descripción de todos los argumentos
  - una lista de los módulos subordinados

  3.- Una explicación de los datos pertinentes, tales como las variables importantes y su uso, restricciones y limitaciones y de otra información importante
  4.- Una historia del desarrollo que incluya
  - el diseñador del módulo (autor)
  - el revisor (auditor) y la fecha
  - fechas de modificación

  Los comentarios descriptivos se encuentran en el cuerpo del código fuente y se usan para describir las funciones de procesamiento.

- **Declaración de datos**

  El orden en la declaración de los datos se debe estandarizar.

El orden hace que los datos sean fáciles de descubrir, comprobar y mantener. Cuando se declaran múltiples nombres de variables en una sentencia, merece la pena ponerlos en orden alfabético.

Si el diseño describe una estructura de datos compleja, se deben usar comentarios para explicar las particularidades inherentes a la implementación en el lenguaje de programación.

- **Construcción de sentencias**

  La construcción del flujo lógico del software se establece durante el diseño. La construcción de sentencias individuales es parte de la codificación. La construcción de sentencias se debe basar en una regla general: cada sentencia debe ser simple y directa, el código no debe ser retorcido.

  Muchos lenguajes de programación permiten disponer múltiples sentencias en una misma línea. El ahorro de espacio que esto implica no está justificado por la pobre legibilidad del resultado.

  Las sentencias de código fuente se pueden simplificar al:

  1.- Evitar el uso de complicadas comparaciones condicionales
  2.- Eliminar las comparaciones con condiciones negativas
  3.- Evitar un gran anidamiento de bucles o de condiciones
  4.- Usar paréntesis para clarificar las expresiones lógicas o aritméticas
  5.- Usar espacios y/o símbolos claros para aumentar la legibilidad del contenido de la sentencia
  6.- Pensar "¿Podría yo entender esto si no fuera la persona que lo codificó?"

- **Entrada/salida**

  La forma en que se implementa la E/S puede ser una característica determinante de la aceptación del sistema por una comunidad de usuarios. El estilo de la entrada salida variará con el grado de integración humana. Se deben considerar una serie de principios para el diseño y la codificación de la E/S

  1.- Validar todos los datos de entrada
  2.- Comprobar las importantes combinaciones de elementos de entrada
  3.- Mantener el formato de entrada simple
  4.- Etiquetar las peticiones interactivas de entrada, especificando las opciones posibles o los valores límite.
  5.- Etiquetar todas las salidas y diseñar todos los informes.

7.1. ¿A quienes interesa el código fuente?

- Autores del propio código
- Otros desarrolladores del proyecto
- Clientes de la API del proyecto

7.2. ¿Por qué documentarlo?

- Mantenimiento
- Reutilización

7.3. ¿Qué documentar?

- Obligatorio:
  - ⋆ Clases y paquetes
  - ⋆ Constructores, métodos y atributos
- Conveniente
  - ⋆ Fragmentos no evidentes
  - ⋆ Bucles, algoritmos . . .

8. JAVADOC

8.1. Definición

Javadoc es una herramienta para el lenguaje java, que analiza las declaraciones y comentarios de la documentación interna del código de los ficheros fuente java y genera la correspondiente pagina web como documentación externa.

Javadoc permite generar la documentación del API (Application Programming Interface) de los programas desarrollados.

Javadoc es una herramienta analiza (parse) las declaraciones y los comentarios de la documentación de un conjunto de ficheros fuente y produce páginas html describiendo clases, clases internas (subclases), interfaces, constructores y atributos de la clase (también llamados fields o campos).

Javadoc permite tener documentación interna dentro del código, y documentación externa como documentación técnica en el que se detalla la aplicación java.

8.2. Comentarios Javadoc

Los comentarios en Java pueden ser:

- Un línea
- Múltilínea
- Javadoc

Los comentarios javadoc son como los comentarios multilínea de java pero empiezan con dos asteriscos.

Ejemplo:

```
\\ Un linea

\*
* Comentario Multinea
*\

\**
* Comentario Javadoc
*\
```

Un detalle importante a tener en cuenta es que SIEMPRE que se quiera comentar algo, una clase, un método, una variable, etc..., dicho comentario se debe poner inmediatamente antes del item a comentar. En caso contrario la herramienta de generación automática no lo reconocerá.

Los comentarios javadoc tienen dos partes:

1.- La parte de descripción

2.- La parte de tags o etiquetas.

Ejemplo:

```
/**
*
* Descripción principal (Texto/HTML)
*
* Tags (Texto/HTML)
*
*/
```

Ver la figura 1 de la página 13.

8.3. Las etiquetas

Aparte de los comentarios propios, la utilidad javadoc nos proporciona una serie de etiquetas o tags, para completar la información que queremos dar de una determinada clase o método. Son las siguientes:

1.- **@author** nombre

Indica el autor de la clase en el argumento nombre. Un comentario de este tipo puede tener más de un autor en cuyo caso podemos usar tantas etiquetas de este tipo como autores hayan colaborado en la creación del

```
/**
 * Returns the index of the first occurrence of the specified element in
 * this vector, searching forwards from <code>index</code>, or returns -1 if
 * the element is not found.
 *
 * @param o element to search for
 * @param index index to start searching from
 * @return the index of the first occurrence of the element in
 *         this vector at position <code>index</code> or later in the vector;
 *         <code>-1</code> if the element is not found
 * @throws IndexOutOfBoundsException if the specified index is negative
 * @see     Object#equals(Object)
 */
public int indexOf(Object o, int index) ...
```

indexOf

```
public int indexOf(Object o,
                   int index)
```

Returns the index of the first occurrence of the specified element in this vector, searching forwards from `index`, or returns -1 if the element is not found.

Parameters:
`o` - element to search for
`index` - index to start searching from

Returns:
the index of the first occurrence of the element in this vector at position `index` or later in the vector; `-1` if the element is not found.

Throws:
`IndexOutOfBoundsException` - if the specified index is negative

See Also:
`Object.equals(Object)`

Figura 1: Comentarios Javadoc

código fuente o bien podemos ponerlos a todos en una sola etiqueta. En éste último caso, Javadoc inserta una (,) y un espacio entre los diferentes nombres.

2.- **@deprecated** comentario

Añade un comentario indicando que este API no debería volver a usarse, aunque aun siga perteneciendo a la distribución del SDK que estemos utilizando, por estar desautorizado o ”desfasado”. No obstante, ésto es sólo una advertencia que nosotros damos a los usuarios de nuestras API’s, al igual que las distribuciones de Sun hacen con nosotros. Realmente, lo que estamos haciendo al decir que una determinada API está desfasada es prevenir de que en un futuro podrán surgir incompatibilidades si seguimos usándolas ya que éste es el paso a previo a la desaparición del API en concreto.

En la primera frase del comentario, que es la que la documentación nos la muestra en la sección del resumen de nuestra clase, deberíamos como mínimo poner desde que versión de nuestra API está desautorizada y por quién se debería sustituir. A partir de Java 1.2 podemos usar @link para referenciar por quién debemos hacer la sustitución.

3.- **@exception**nombre-clase descripción

Esta etiqueta actúa exactamente igual que @throws.

4.- **@link** nombre etiqueta

Inserta un enlace autocontenido que apunta a nombre. Esta etiqueta acepta exactamente la misma sintáxis que la etiqueta @see, que se describe más abajo, pero genera un enlace autocontenido en vez de colocar el enlace en la sección "See Also". Dado que esta etiqueta usa los carácteres para separarla del resto del texto in-line, si necesitas emplear el caracter ''}'' dentro de la etiqueta debes usar la notación HTML `&#125;`

No existe ninguna limitación en cuanto al número de etiquetas de este tipo permitidas. Puedes usarlas tanto en la parte de la descripción como en cualquier porción de texto de cualquier otra etiqueta de las que nos proporciona JavaDoc.

En el siguiente ejemplo, vemos como crear dentro de nuestra documentación un enlace in-line al método getComponentAt(int, int).

5.- **@deprecated** Usar el método

`{@link #getComponentAt(int, int) getComponentAt}`

A partir de esta descripción, la herramienta de generación automática de código generará el siguiente código HTML (supone que se está refiriendo a una clase del mismo paquete): Usar el método `<a href="Component.html#getComponentAt(int, int)">getComponentAt</a>` , el cual aparecerá en la página HTML como:

Usar el método getComponentAt

6.- **@param** parámetro descripción

Añade un parámetro y su descripción a la sección "Parameters" de la documentación HTML que generará. Por tanto, para cada método emplearemos tantas etiquetas de este estilo como parámetros de entrada tenga dicho método.

Se aplica a parámetros de constructores y métodos. Describe los parámetros del constructor/método.

Ejemplo:

```
/**
* Borra de una lista de items todos sus elementos
*
* @param formIndex indice del primer elemento a borrar
* @param toIndex indice del último elemento a borrar
*/
protected void removeRange( int formIndex, int toIndex){
  ...
}
```

7.- **@return** descripción

Añade a la sección "Returns" de la documentación HTML que va a generar la descripción del tipo que devuelve el método.

Se aplica a métodos. Describe el valor de retorno del método. Se incluye la descripción del valor de retorno.

Ejemplo:

```
/*
* ....
* @param s1 Texto que ocupa la
* @param s2 Texto que ocupa el
* @param s3 Texto que ocupa la
* @return La cadena conformada
*/
public String concatena(String s1, String s2, String s3) {
  ...
}
```

Ejemplo:

```
/*
* Comprueba si el vector no tiene componentes
* @return <code>true<code> si no tiene componentes
* <code>false</code> en otro caso.
*/
public boolena concatena() {
  return elementCount==0;
}
```

8.- **@see** referencia

Añade un cabecero ”See Also” con un enlace o texto que apunta a una referencia. El comentario de la documentación puede contener cualquier número de etiquetas de este tipo y todas, al generar la documentación, se agruparán bajo el mismo cabecero. Esta etiqueta se puede conformar de tres maneras diferentes, siendo la última forma la más utilizada. `@see "string"`

En este caso, no se genera ningún enlace. La string es un libro o cualquier otra referencia a una información que no está disponible via URL. JavaDoc distingue este caso buscando en el primer carácter de la cadena las comillas dobles (”). Por ejemplo:

@see ”The Java Programming Language”

que genera el siguiente texto HTML:

See Also:

”The Java Programming Language”

`@see <a href="URL#value">label</a>`

Añade un enlace definido por `URL#value`. Esta dirección puede ser relativa o absoluta. JavaDoc distingue este caso buscando en el primer carácter el símbolo `"<"`. Por ejemplo:

`@see <a href="spec.html#section">Java Spec</a>` que genera el siguiente enlace: See Also: Java Spec `@see package.class#member texto`

Añade un enlace, con el texto visible texto, que apunta en la documentación al nombre especificado en el lenguaje Java al cual hace referencia. Aquí por nombre se debe entender un paquete, una clase, un método o un campo. El argumento texto es opcional, si se omite, JavaDoc nos dará la información

mínima, ésto es, el nombre de, por ejemplo, la pareja `paquete#método`. Usa este campo cuando quieras especificar algo más, cuando quieras representarlo por un nombre más corto o simplemente si quieres que aparezca con otro nombre. Veámos a continuación varios ejemplos usando la etiqueta @see.

Nota: El comentario a la derecha muestra cómo aparecerá la etiqueta en las especificaciones HTML si la referencia a la que apunta estuviera en otro paquete distinto al que estamos comentando.

```
@see java.lang.String // String
@see java.lang.String The String class // The String class
@see String // String
@see String#equals(Object) // String.equals(Object)
@see String#equals // String.equals(java.lang.Object)
@see java.lang.Object#wait(long) // java.lang.Object.wait(long)
@see Character#MAX_RADIX // Character.MAX\_RADIX
@see <a href="spec.html">Java Spec</a> // Java Spec
@see ''The Java Programming Language'' // Language ''The Java Programming''
```

9.- **@since** texto

Indica con texto desde cuándo se creó este paquete, clase o método. Normalmente se pone la versión de nuestra API en que se incluyó, así en posteriores versiones sabremos a qué revisión pertenece o en qué revisión se añadió. Por ejemplo:

```
@since JDK1.1
```

10.- **@serial** field-description

Su uso está destinado a señalar un campo serializable. Por defecto, todos los campos (variables) son susceptibles de ser serializados lo cual no quiere decir que nuestra aplicación lo tenga que hacer. Por tanto, se usa sólo cuando se tenga una variable serializable.

11.- **@serialField** field-name field-typefield-description

Documenta un componente ObjectStreamField de un miembro serialPersistentFields de una clase Serializable. Esta etiqueta se debería usar para cada componente ObjectStreamField.

12.- **@serialData** data-description

Se emplea para describir los datos escritos por el método writeObject y todos los datos escritos por el método Externalizable.writeExternal. Esta etiqueta puede ser usada en aquellas clases o métodos que intervengan los métodos writeObject, readObject, writeExternal, and readExternal.

13.- **@throws** nombre-clase descripción

Como ya se apuntó, esta etiqueta es la gemela de @exception. En ambos casos, se añade una cabecera ”Throws” a la documentación generada con el nombre de la excepción que puede ser lanzada por el método (nombre-clase) y una descripción de por qué se lanza.

Se aplica a constructores y métodos. Describe posibles excepciones. Un @throws por cada posible excepción

Ejemplo:

```
/**
* Analiza el argumento string  como un numero decimal
* <code>long</code>
*
* @param  s  un <code>String</code> contiene el <code>long<code>
*
* @return  el <code>long</code> modificado
*
* @exception NumberformaException  si el string no contien un
*                   <code>long</code>analizable.
*/
public static long parseLong(String s)
          throws NumberformaException {
            ...
          }
```

14.- **@version** version

Añade un cabecero a la documentación generada con la versión de esta clase. Por versión, normalmente nos referimos a la versión del software que contiene esta clase o miembro.

8.4. Uso de tags

8.4.1. Comentario de la página Overview.

Las etiquetas que se pueden emplear en esta sección son: @see, @link, @since. Observa que ésto no se corresponde con ninguna parte de un código Java. La página "Overview", que reside en un fichero fuente al que normalmente se le da el nombre de overview.html es la que nos da una idea global de las clases y paquetes que conforman nuestra API (ver especificaciones html de Sun).

8.4.2. Comentario de paquetes.

Este tipo de comentarios tampoco lo hacemos en nuestro código fuente sino en una página html que reside en cada uno de los directorios de nuestros paquetes y que siempre se le da el nombre de package.html.

Las etiquetas que podemos utilizar en este caso son: @see, @link, @since, @deprecated

8.4.3. Comentario de clases e interfaces.

Disponemos de las siguientes etiquetas: @see, @link, @since, @deprecated, @author, @version.

Ejemplo:

```
/**
* A class representing a window on the screen.
* For example:
```

```
* <pre>
* Window win = new Window(parent);
* win.show();
* </pre>
*
* @author Sami Shaio
* @version %I%, %G%
* @see java.awt.BaseWindow
* @see java.awt.Button
*/
class Window extends BaseWindow {
...
}
```

8.4.4. Comentario de variables.

Las etiquetas que podemos utilizar en esta ocasión son: @see, @link, @since, @deprecated, @serial, @serialField.

Ejemplo:

```
/**
* The X-coordinate of the component.
* @see #getLocation()
*/
int x = 1263732;
```

8.4.5. Comentario de métodos y constructores.

En este caso disponemos de: @see, @link, @since, @deprecated, @param, @return, @throws (@exception), @serialData.

Ejemplo:

```
/**
* Returns the character at the specified index. An index
* ranges from <code>0</code> to <code>length() - 1</code>.
* @param index the index of the desired character.
* @return the desired character.
* @exception StringIndexOutOfRangeException if the index is not in the
* range <code>0</code>  to <code>length()-1</code>.
* @see java.lang.Character#charValue()
*/
public char charAt(int index) {
...
}
```

8.5. Uso de javadoc

La utilidad de documentación javadoc es un programa que se suministra dentro de la distribución de J2SE.

Modo de uso:

javadoc [opciones] [paquetes] [archivosfuente] [@ficheros]

Con [opciones] modificamos el funcionamiento de javadoc. Se pueden consultar con:

```
javadoc --help
```

Ejemplo: La orden de terminal que crea el javadoc html en una carpeta incluye autor y versión del fichero Miclase.java es:

```
javadoc -d Micarpeta -version -author -private MiClase.java
```

Ejemplo: La orden de terminal que crea el javadoc html en una carpeta incluye autor y versión de todos los ficheros .java es:

```
javadoc -d Micarpeta -version -author -private *.java
```

9. JAVADOC EN NETBEANS

Netbeans permite crear la documentación interna, de forma sencilla, para ello veremos como añadir comentarios y la documentación html con javadoc.

9.1. Visualizar el API de las librerias standard

Seleccionar el elemento. y Pulsar CTRL+SHIFT+Espacio o Menú Fuente - Mostrar Documentación.

El Javadoc para este elemento es mostrado mediante una ventana emergente.

Ver la figura 2 de la página 19.

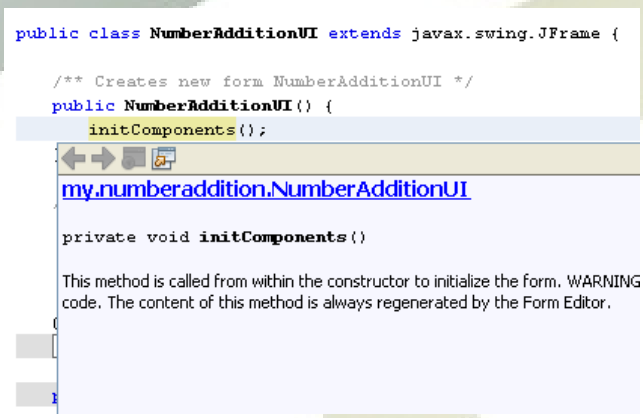

Figura 2: API de las librerias standard

9.2. Para comentar un método.

Podemos hacer salgan los argumentos para comentarlos nos podremos delante del método y escribiremos `/**` y pulsaremos INTRO.

Ver la figura 3 de la página 20.

```
/**
 *
 * @param args the command line arguments
 */

public static void main(String[] args) {
        BufferedReader dataIn = new BufferedReader(new
                InputStreamReader(System.in));

        String name = "";
        System.out.println("Please, enter your name");
```

Figura 3: Documentar Método

9.3. Añadir tags a un método.

Pondremos el símbolo arroba y nos mostrará los tags disponibles.

Ver la figura 4 de la página 20.

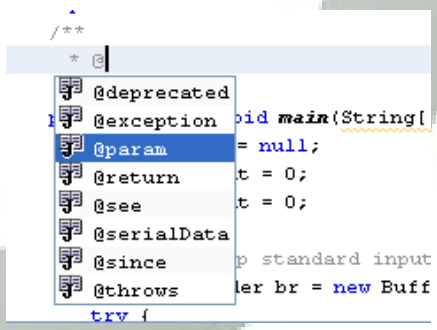

Figura 4: Tags disponibles

9.4. Generar JavaDoc

Podremos generar una página web con la documentación, a través delante Menú Ejecutar - Generar Javadoc.

Ver la figura 5 de la página 21.

9.5. Personalizar el formato de javadoc

Para personalizar el formato de javadoc, seleccionamos el nombre del proyecto, y en el Menú Archivo - Proyecto Propiedades (Javadoc). Entre otras cosas permite poner el titulo a la ventana que genera Javadoc

Ver la figura 6 de la página 21.

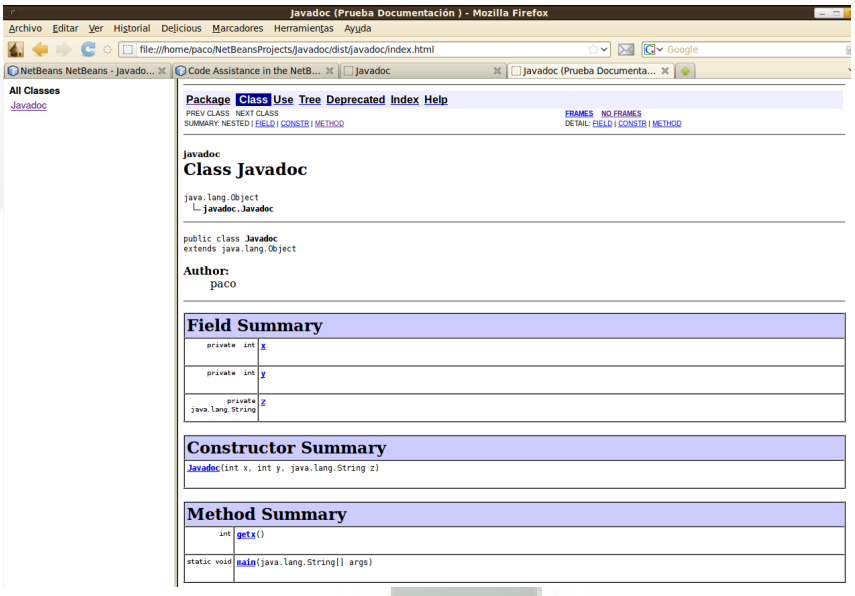

Figura 5: Salida de generar javadoc

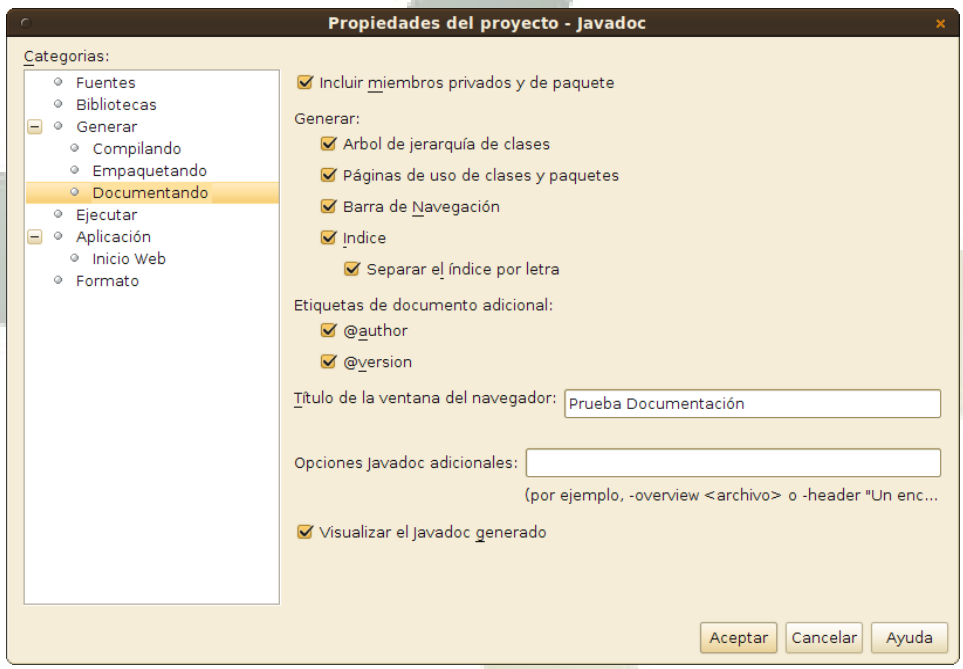

Figura 6: Personalizar el generar javadoc

9.6. Buscar métodos sin documentación.

Ir al Menú Herramientas - Analizar JavaDoc. Esto permitirá ver que métodos no están comentados. Y marcando reparar nos genera los comentarios. Seguidamente lo tacha para indicarnos que ya tiene sus comentarios.

Ver la figura 7 de la página 22.

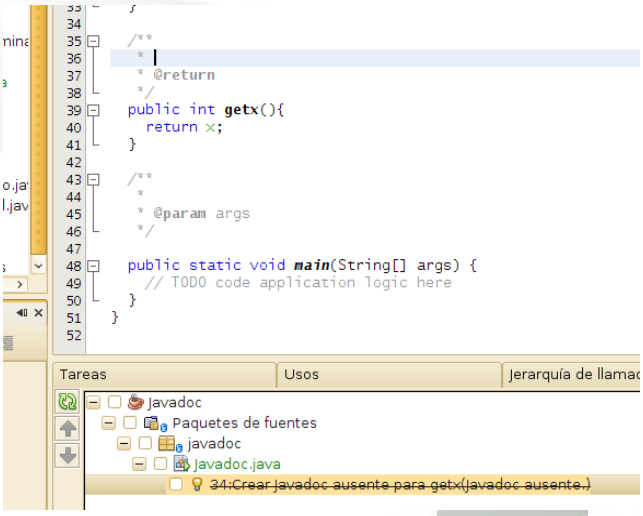

Figura 7: Analizar javadoc

10. REFACTORIZACIÓN

10.1. Definición

Refactorización:
Es mejorar el código fuente sin cambiar el resultado del programa.

10.2. Características

- Cambio realizado a la estructura interna del software para hacerlo más fácil de comprender y más fácil de modificar sin cambiar su comportamiento observable.
- Reestructurar el software aplicando una secuencia de refactorizaciones.
- Optimizar el código

10.3. ¿Por qué se refactoriza el software?

1.- Para mejorar su diseño

- Conforme se modifica, el software pierde su estructura.
- 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.

11. TIPOS DE REFACTORIZACIÓN

11.1. Cambiar de nombre

Se debe refactorizar cuando el nombre no revela que realiza. También llamado rename method. Se puede realizar sobre variables, métodos, comentarios, etc.

Ver la figura 8 de la página 23

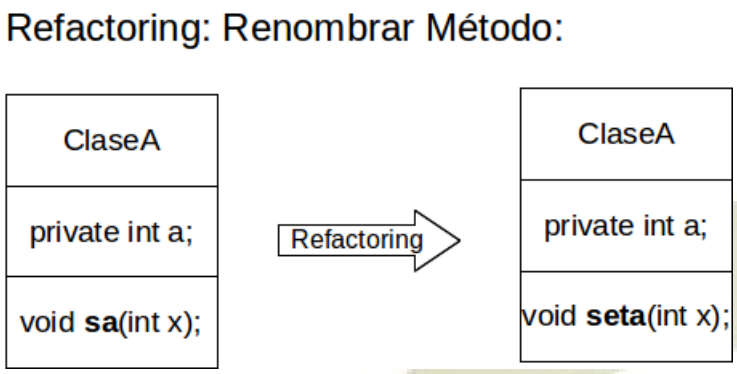

Figura 8: Renombrar Método

11.2. Mover

Un método de una clase será movido a otra otra clase que lo usa más. También llamado Move Method.

Ver la figura 9 de la página 24

Ejemplo:

- Sin Refactoring

```
class Project {
  Person[] participants;
}

class Person {
```

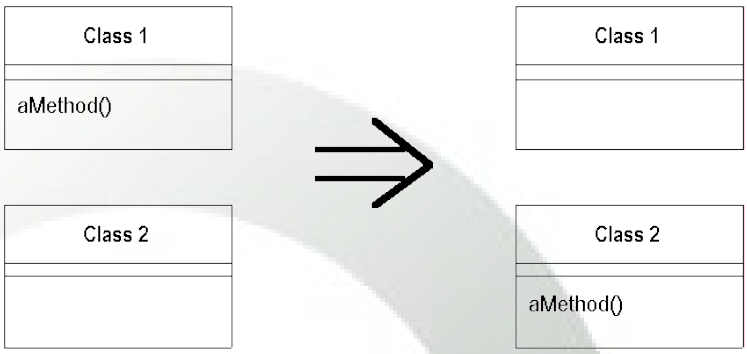

Figura 9: Mover

```
  int id;
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
          if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```

- Con Refactoring

```
class Project {
  Person[] participants;
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
          if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
}

... if (p.participate(x)) ...
```

11.3. Introducir método.

Permite introducir un bloque de código en un método. También llamado Extract Method.

Ejemplo:

- Sin refactorizar

```
void printOwing() {
    printBanner();
    //print details
    System.out.println ("name:      " + _name);
    System.out.println ("amount     " + getOutstanding());
```

```
}
```

- Factorizado.

```
void printOwing() {
   printBanner();
   printDetails(getOutstanding());
}

void printDetails (double outstanding) {
   System.out.println ("name:      " + _name);
   System.out.println ("amount     " + outstanding);
}
```

Existen muchos más tipos que se puede ver en la página web Refactorings in Alphabetical Order que pertenece al libro [FBB99] de Martin Fowler:

`http://www.refactoring.com/catalog/index.html`

11.4. Caso práctico de refactoring

Tengo una aplicación bancaria que ya fue correctamente implementada, esta aplicación usa extensivamente una clase llamada Transacción, la cual posee una operación cuya firma es:

public void asignarCtaCorriente(CuentaCorriente cc);

Esta operación es invocada una gran cantidad de veces por distintas operaciones en nuestra aplicación. El programa es que se debe cambiar la firma de la operación para homogenizar el estándar de nomenclatura establecido por la empresa que desarrolló la aplicación.

La nueva firma debería ser:

public void asignarCuentaCorriente(CuentaCorriente cc);

¿Cuál es la solución?

Debemos iniciar el proceso de refactoring en nuestra aplicación para reflejar este cambio. Modificaremos la firma en la clase Transacción y luego modificar en todo el proyecto cada una de las invocaciones. Si la aplicación es realmente grande, implicará un gran esfuerzo llevar a cabo este cambio manualmente.

Probablemente se introduzcan errores en la aplicación.

A medida que se desarrolla una aplicación con NetBeans, de forma transparente para el programador se genera un repositorio de metadata del código que el programador genera. La metadata se podría definir brevemente como ”datos que describen otros datos”, por lo cual NetBeans puede contar con esta información para realizar cambios sobre los fuentes automáticamente. Gracias a este repositorio de metadata que NetBeans mantiene es posible automatizar muchas tareas de refactoring, por ejemplo, en el caso que describimos anteriormente, NetBeans se encarga de cambiar la firma de la operación de la clase Cuenta, buscar todas las invocaciones y modificarlas. Adicionalmente NetBeans provee un informe del impacto que tendrá el proceso de refactoring, para permitir al programador tomar

decisiones respecto a este proceso. Luego de desplegado dicho informe se puede iniciar el proceso de refactoring de acuerdo a las decisiones tomadas por el programador.

12. REFACTORIZAR CON NETBEANS

12.1. Donde se encuentra

Netbeans al refactoring le llama **reestructurar**.

Se puede obtener :

- Menú Reestructurar
- Menú Contextual (botón derecho del ratón sobre el texto marcado) - Reestructurar.

Ver la figura 10 de la página 26.

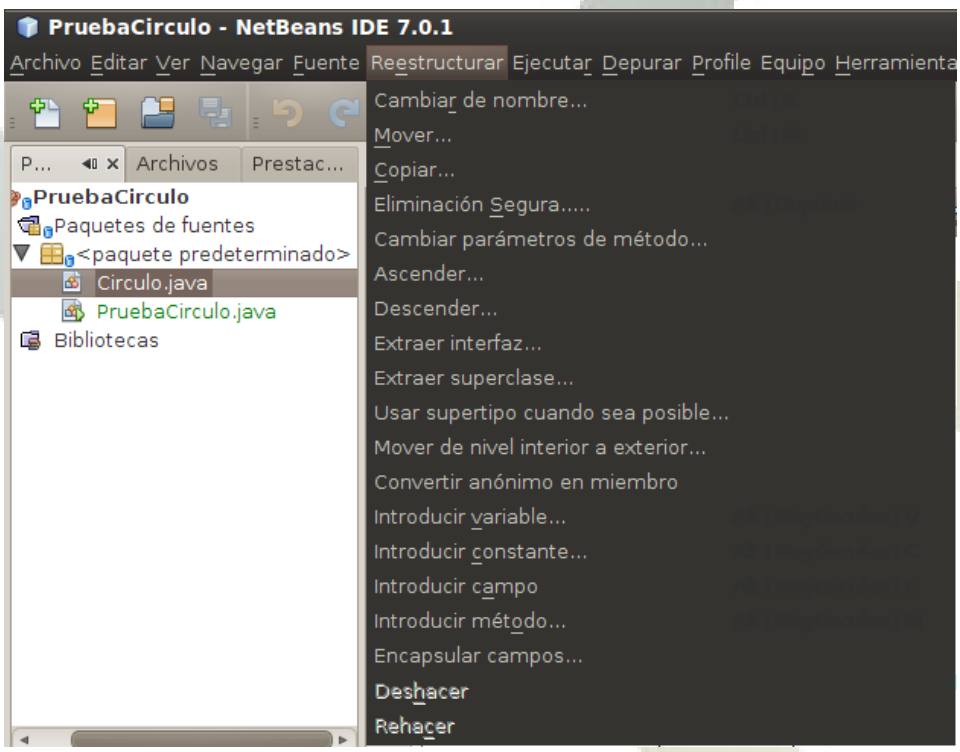

Figura 10: Menú Reestructurar

13. EJERCICIO DE REFACTORIZACIÓN CON NETBEANS

Disponemos de las siguientes clases:

`../edcodigo/edcod10/Refactoring/src/Circulo.java`

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author paco
 */
public class Circulo {

  private int x;
  private int y;
  private double radio;

  public Circulo() {
  }  // constructor

  public Circulo(int valorX, int valorY, double valorRadio) {
    x = valorX;
    y = valorY;
    establecerRadio(valorRadio);
  }

  public void establecerX(int valorX) {
    x = valorX;
  }

  public int obtenerX() {
    return x;
  }

  public void establecerY(int valorY) {
    y = valorY;
  }

  public int obtenerY() {
    return y;
  }

  public void establecerRadio(double valorRadio) {
    radio = (valorRadio < 0.0 ? 0.0 : valorRadio);
  }

  public double obtenerRadio() {
    return radio;
  }

  public double obtenerDiametro() {
    return 2 * radio;
  }

  public double obtenerCircunferencia() {
    return Math.PI * obtenerDiametro();
  }

  public double obtenerArea() {
    return Math.PI * radio * radio;
  }

  @Override
  public String toString() {
    return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
  }
}
```

../edcodigo/edcod10/Refactoring/src/PruebaCirculo.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author paco
 */

import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class PruebaCirculo {

  public static void main(String[] args) {

  Circulo circulo = new Circulo( 37, 43, 2.5 );
  String salida = "La coordenada X es " + circulo.obtenerX()
          + "\nLa coordenada Y es " + circulo.obtenerY()
          + "\nEl radio es " + circulo.obtenerRadio();

  circulo.establecerX (35 );
  circulo.establecerY (20 );
  circulo.establecerRadio (4.25 );
  salida += "\n\nLa nueva ubicacion y el radio del circulo son\n"
          + circulo.toString() ;
  DecimalFormat dosDigitos = new DecimalFormat("0.00");
  salida += "\nEl diametro es "
          + dosDigitos.format(circulo.obtenerDiametro()) ;
  salida += "\nLa circunferencia es "
          + dosDigitos.format(circulo.obtenerCircunferencia()) ;
  salida += "\nEl Area es " + dosDigitos.format(circulo.obtenerArea()) ;

  JOptionPane.showMessageDialog (null, salida );
  System.exit (0);

  }
}
```

Si nos situamos en cualquiera de nuestras clases y botón derecho, obtenemos el siguiente menú, donde podemos observar que tenemos la opción **Reestructurar** (Refactorizar):

Ver la figura 11 de la página 29.

A continuación, se encuentra una breve descripción de las diferentes opciones del menú:

13.1. Renombrar

Cambia el nombre de una clase, variable o método. Adicionalmente modifica todo el código del proyecto para referenciar al nuevo nombre.

Cambiamos el nombre Circulo por Circulito y observamos los cambios en el código que donde ponía Circulo ahora pone Circulito.

Ver la figura 12 de la página 29.

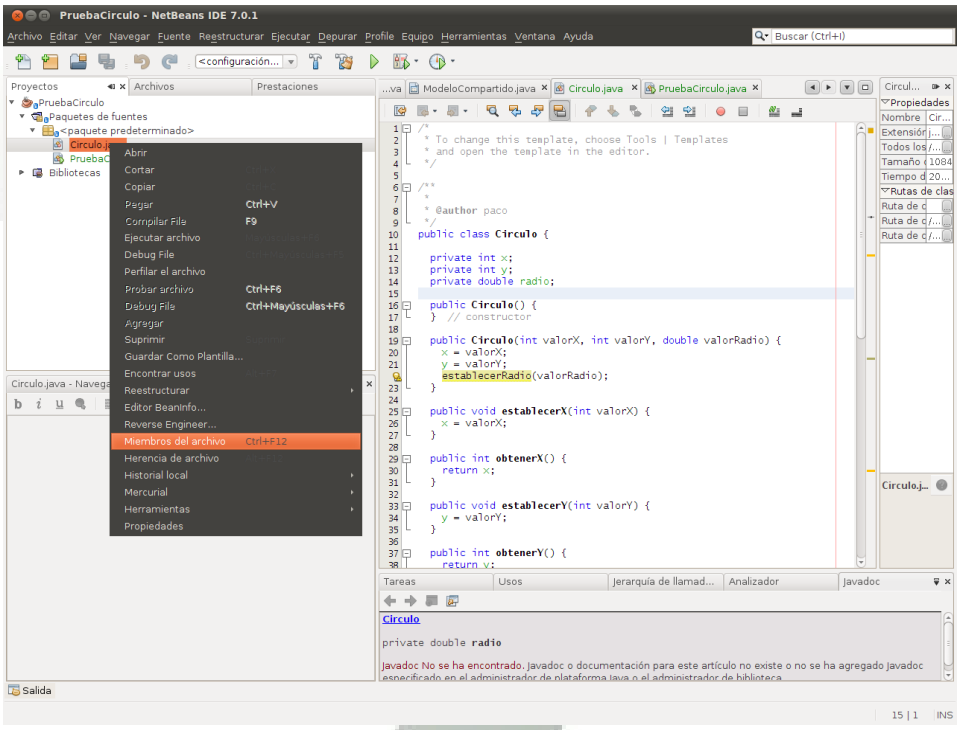

Figura 11: Reestructurar clase

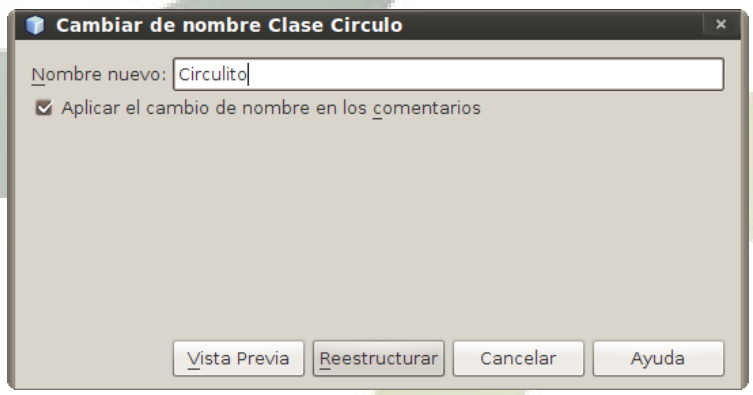

Figura 12: Renombrar clase

13.2. Eliminación Segura

Cuando eliminamos un método o clase, debemos garantizarnos de que nadie más lo utilice en el proyecto. Esta operación verifica y notifica las referencias encontradas, proveyendo de mecanismos para que fácilmente el programador pueda eliminar una a una las referencias, para finalmente llevar a cabo la operación de borrado de modo seguro. Además permite ver como quedaría después de borrar.

Borrar el constructor sin parámetros public Circulito de la clase Circulito.

Ver la figura 13 de la página 30.

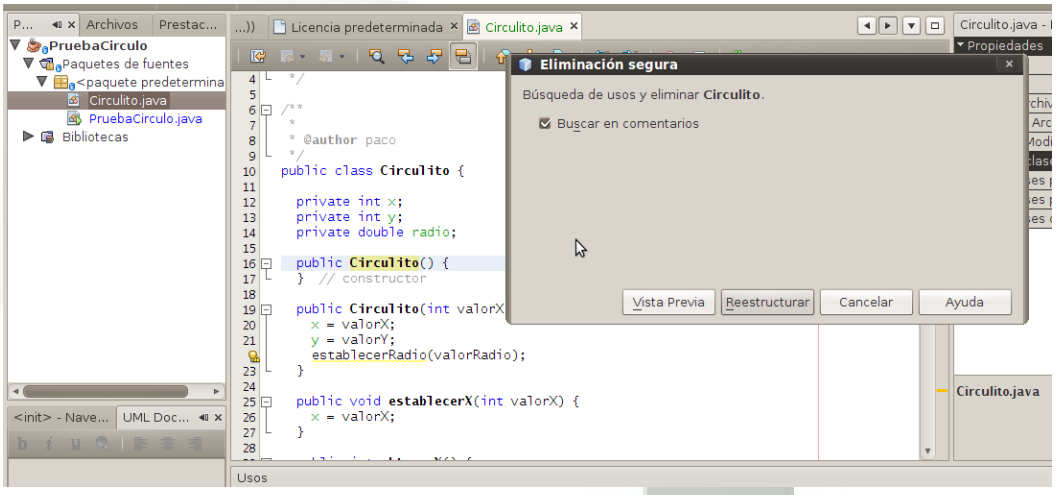

Figura 13: Eliminación segura

13.3. Cambiar parámetros de un método

Permite agregar, eliminar, modificar o cambiar el orden de los parámetros de un método, al igual que su modificador de acceso (private o public).

Cambiar de orden los parámetros del método de forma que en primer lugar este el valor del radio, de la clase Circulito.

Ver la figura 14 de la página 30.

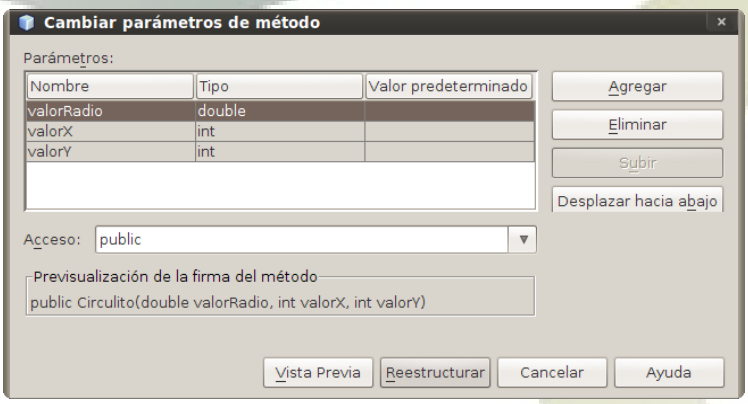

Figura 14: Cambiar parámetros de un método

13.4. Encapsular campos

Es muy común tener que acceder a los campos de una clase por medio de operaciones del tipo:

```
public set<nombre del campo>(...)
public get<nombre del campo>()
```

Es una tarea muy tediosa, por lo cual esta operación permite que el programador solo deba implementar los campos, delegando a NetBeans la tarea de ”encapsularlos. También es posible que todo código del proyecto que accede directamente al campo, pase automáticamente a utilizar el ”setter o el ”getter determinado. Debemos seleccionar el campo a encapsular para que aparezca la opción de encapsulación.

Añadir el atributo: private int z de la clase Circulito; y realizar la encapsulación.

Ver la figura 15 de la página 31.

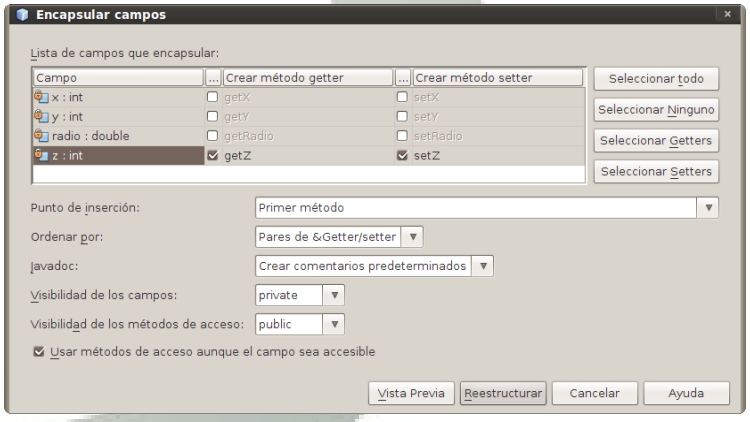

Figura 15: Encapsular campos

13.5. Ascender métodos o campos

Permite subir un método o campo a otra clase de la cual hereda la clase que contiene al método o campo que deseamos subir.

13.6. Descender clases anidadas, métodos o campos

Permite bajar una clase anidada, método o campo a otra clase la cual hereda de la clase que contiene a la clase anidada, método o campo que deseamos bajar.

13.7. Mover una clase

Mueve una clase a otro package o dentro de otra clase. Adicionalmente modifica todo el código del proyecto para referenciar al nuevo lugar donde se movió la clase.

13.8. Convertir una clase anónima anidada a una clase anidada

Crea una nueva clase anidada, la cual tendrá un nombre y un constructor. La clase anónima anidada será sustituida por esta nueva clase anidada.

13.9. Extraer una interfase

Permite seleccionar cuales métodos públicos no estáticos de una clase o interfase, irán a parar a una nueva interfase. La clase de la cual fue extraída la interfase implementará la nueva interfase creada. La interfase de la cual fue extraída la interfase extenderá la nueva interfase.

13.10. Extraer superclase

Despliega al programador los métodos y campos que se pueden mover a una superclase. El programador selecciona cuales desea mover y NetBeans creará una nueva clase abstracta que contendrá dichos campos y métodos, también hará que la clase refactorizada la extienda.

Extraer la Superclase Figura, la cual tendrá como métodos abstractos obtenerArea y toString de la clase Circulito.

Ver la figura 16 de la página 32.

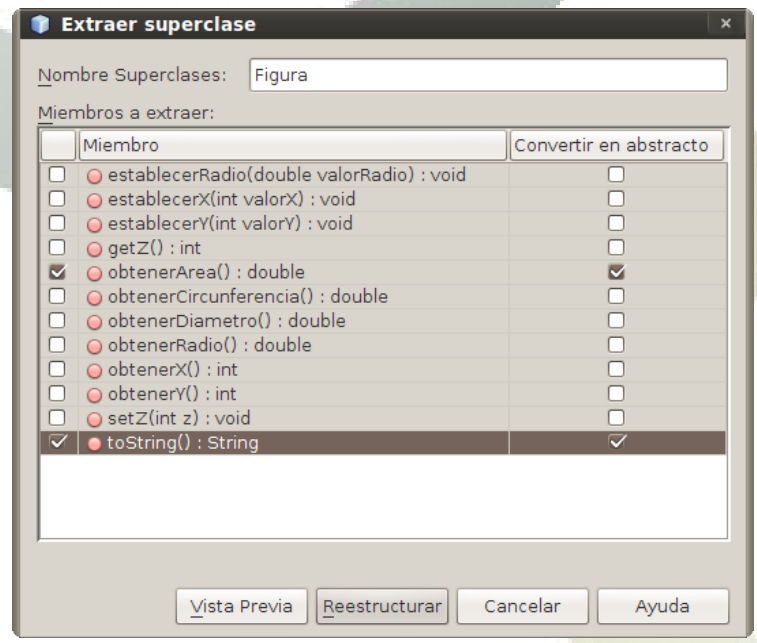

Figura 16: Extraer superclase

13.11. Usar supertipo cuando sea posible

Despliega al programador todas las clases que extiende la clase actual. El programador seleccionará una, y NetBeans buscará en todo el proyecto referencias a la clase que se quiere refactorizar, si encuentra referencias, determinará si es posible utilizar la superclase seleccionada.

13.12. Introducir constantes, variables, campos o métodos

El programador selecciona un fragmento de código, y luego presiona las teclas <alt>+<Enter>. NetBeans desplegará varias opciones útiles, como encapsular ese fragmento en un método y referenciar al método, anidarse en while, if, etc.

Si el bloque de código aparece más veces también lo reemplaza por el nombre del método que pongamos.

Para introducir en método un conjunto de instrucciones debemos seleccionarlas previamente para luego seleccionar el Menú Estructurar - Introducir ....

En la clase PruebaCírculo introducir en un método llamado void pestablecer() las instrucciones :

```
circulo.establecerX (35 );
circulo.establecerY (20 );
circulo.establecerRadio (4.25 );
```

Ver la figura 17 de la página 33.

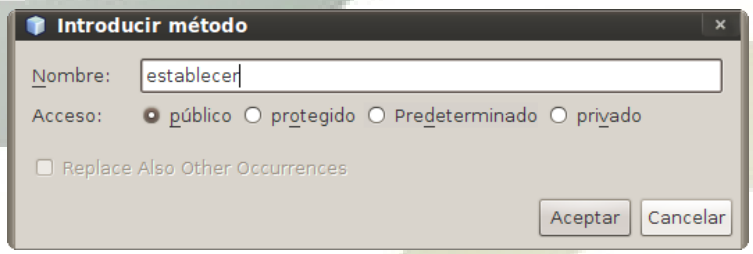

Figura 17: Introducir método

Como resultado será cambiar las instrucciones por una llamada a una función y añadirá al final el método establecer:

```
  public static void establecer(Circulito circulo) {
    circulo.establecerX (35 );
    circulo.establecerY (20 );
    circulo.establecerRadio (4.25 );
  }
```

13.13. Cuando usar refactoring con netbeans

Siempre que sea posible, utilizar las herramientas de refactoring que NetBeans provee, de este modo no solo será mucho más sencillo este tipo de procedimiento, sino que además ejecutaremos un proceso seguro mediante el cual no introduciremos errores humanos.

13.14. Clases resultantes

Las clases resultantes después de las refactorizaciones son:

1.- ../edcodigo/edcod10s/Refactoring/src/Figura.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author pacoaldarias <paco.aldarias@ceedcv.es>
 */
public abstract class Figura {

  public Figura() {}

  public abstract double obtenerArea();

  public abstract String toString();

}
```

2.- ../edcodigo/edcod10s/Refactoring/src/Circulito.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author paco
 */
public class Circulito extends Figura {

  private int x;
  private int y;
  private double radio;
  private int z;

  /**
   * @return the z
   */
  public int getZ() {
    return z;
  }

  /**
   * @param z the z to set
   */
  public void setZ(int z) {
```

```
    this.z = z;
  }

  public Circulito(double valorRadio, int valorX, int valorY) {
    x = valorX;
    y = valorY;
    establecerRadio(valorRadio);
  }

  public void establecerX(int valorX) {
    x = valorX;
  }

  public int obtenerX() {
    return x;
  }

  public void establecerY(int valorY) {
    y = valorY;
  }

  public int obtenerY() {
    return y;
  }

  public void establecerRadio(double valorRadio) {
    radio = (valorRadio < 0.0 ? 0.0 : valorRadio);
  }

  public double obtenerRadio() {
    return radio;
  }

  public double obtenerDiametro() {
    return 2 * radio;
  }

  public double obtenerCircunferencia() {
    return Math.PI * obtenerDiametro();
  }

  public double obtenerArea() {
    return Math.PI * radio * radio;
  }

  public String toString() {
    return "Centro = [" + x + ", " + y + "]; Radio = " + radio;
  }
}
```

3.- ../edcodigo/edcod10s/Refactoring/src/PruebaCirculo.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author paco
 */

import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class PruebaCirculo {
```

```
  public static void main(String[] args) {

  Circulito circulo = new Circulito( 2.5, 37, 43);
  String salida = "La coordenada X es " + circulo.obtenerX()
          + "\nLa coordenada Y es " + circulo.obtenerY()
          + "\nEl radio es " + circulo.obtenerRadio();
    establecer (circulo);

  salida += "\n\nLa nueva ubicacion y el radio del ciï¿½rculo son\n"
          + circulo.toString() ;
  DecimalFormat dosDigitos = new DecimalFormat("0.00");
  salida += "\nEl diametro es "
          + dosDigitos.format(circulo.obtenerDiametro()) ;
  salida += "\nLa circunferencia es "
          + dosDigitos.format(circulo.obtenerCircunferencia()) ;
  salida += "\nEl Area es " + dosDigitos.format(circulo.obtenerArea()) ;

  JOptionPane.showMessageDialog (null, salida );
  System.exit (0);

  }

  public static void establecer(Circulito circulo) {
    circulo.establecerX (35 );
    circulo.establecerY (20 );
    circulo.establecerRadio (4.25 );
  }
}
```

14. ORIENTACIONES

14.1. Objetivos

- Conocer la importancia de documentar.
- Conocer los tipos de documentación
- Conocer que se documenta en cada etapa de un proyecto software
- Saber que es documentación interna y externa.
- Conocer el uso de javadoc
- Saber utilizar el ide para la documentación interna/externa.
- Conocer los tipos básicos de refactorización y su aplicación en entornos integrados.

14.2. Requisitos software

Se recomienda usar:

- Sistema Operativo: Lliurex
- IDE: Netbeans.

14.3. Bibliografía

En los apuntes hacen referencia a bibliografía utilizada, la cual se encuentra detallada en la última hoja.

14.4. Recursos en internet

- JavaDoc Tags

  `http://docs.oracle.com/javase/1.4.2/docs/tooldocs/windows/javadoc.html`

- JavaDoc Tags

  `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

- JavaDoc. Tutorial Netbeans

  `http://netbeans.org/kb/docs/java/editor-codereference.html`

- JavaDoc. Tutorial Netbeans

  `http://edu.netbeans.org/quicktour/javadoc.html`

- Ejemplo javadoc

  `http://jungla.dit.upm.es/~pepe/doc/fprg/javadoc.htm`

- Uned.

  `http://www.ia.uned.es/ia/asignaturas/adms/GuiaDidADMS/index.html`

- Plantillas para documentar proyectos.

  `http://readyset.tigris.org/es/index.html`

- Ejemplo de RUP

  `http://users.dsic.upv.es/asignaturas/facultad/lsi/ejemplorup/`

- Refactoring con netbeans.

  `http://cnx.org/content/m17586/latest/`

- Refactoring

  `http://www.refactoring.com/catalog/index.html`

14.5. Temporalización

Esta materia se impartirá a distancia en:

DESCRIPCION	CANTIDAD
Total de horas del tema	8
Total de semanas	2
Total de horas por semana	4
Total de horas de tutorías grupales por semana	1
Total de horas del curso	90

14.6. Cuando se imparte

El tema 10 se imparte en la evaluación 3.

EVALUACIÓN	EVALUACIÓN 1				EVALUACIÓN 2				EVALUACIÓN 3			
TEMA	1	2	3	4	5	6	7	8	9	**10**	11	12

15. EJERCICIOS PROPUESTOS

Realiza las actividades y entrega todos los archivos de las actividades en un archivo comprimido llamado **NombreApellidos-ED-Tema10.zip** y súbelo al aula virtual para su corrección.

Realiza una captura del el escritorio de cada ejercicio que demuestre que lo has realizado tú.

15.1. Javadoc

Realizar los comentarios necesarios al fichero fuente en java, y obtener el fichero html generado por javadoc que se muestra a continuación.

Fuente: `../edcodigo/edcod10/Javadoc/Circulo.java`

```
public class Circulo {
  private double centroX;
  private double centroY;
  private double radio;

  public Circulo(double cx, double cy, double r) {
    centroX = cx;
    centroY = cy;
    radio = r;
  }

  public double getCentroX() {
    return centroX;
  }

  public double getCircunferencia() {
    return 2 * Math.PI * radio;
  }

  public void mueve(double deltaX, double deltaY) {
    centroX = centroX + deltaX;
    centroY = centroY + deltaY;
  }

  public void escala(double s) {
    radio = radio * s;
  }
}
```

Javadoc html:

Class Circulo

```
java.lang.Object
  └Circulo
```

```
public class Circulo
extends java.lang.Object
```

Ejemplo simple de comentarios de javadoc

Version:
23.03.2012
Author:
Paco Aldarias

Constructor Summary

`Circulo(double cx, double cy, double r)`	Constructor.

Method Summary

void	`escala(double s)` Escala el circulo (cambia su radio).
double	`getCentroX()` Getter.
double	`getCircunferencia()` Calcula la longitud de la circunferencia (perimetro del circulo).
void	`mueve(double deltaX, double deltaY)` Desplaza el circulo a otro lugar.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Circulo

```
public Circulo(double cx,
               double cy,
               double r)
```

Constructor.

Parameters:
`cx` - centro: coordenada X.
`cy` - centro: coordenada Y.
`r` - radio.

Method Detail

getCentroX

```
public double getCentroX()
```

Getter.

Returns:
centro: coordenada X.

---

getCircunferencia

```
public double getCircunferencia()
```

Calcula la longitud de la circunferencia (perimetro del circulo).

Returns:
circunferencia.

---

mueve

```
public void mueve(double deltaX,
                  double deltaY)
```

Desplaza el circulo a otro lugar.

Parameters:
`deltaX` - movimiento en el eje X.
`deltaY` - movimiento en el eje Y.

---

escala

```
public void escala(double s)
```

Escala el circulo (cambia su radio).

Parameters:
s - factor de escala.

Entregar el fichero fuente con comentarios, el html y una captura del escritorio.

15.2. Refactoring

Realizar el refactoring con Netbean del ejercicio del apartado 13 de la página 26. Entregar el proyecto resultante de modificar el proyecto PruebaCirculo.

Ver la figura 18 de la página 43.

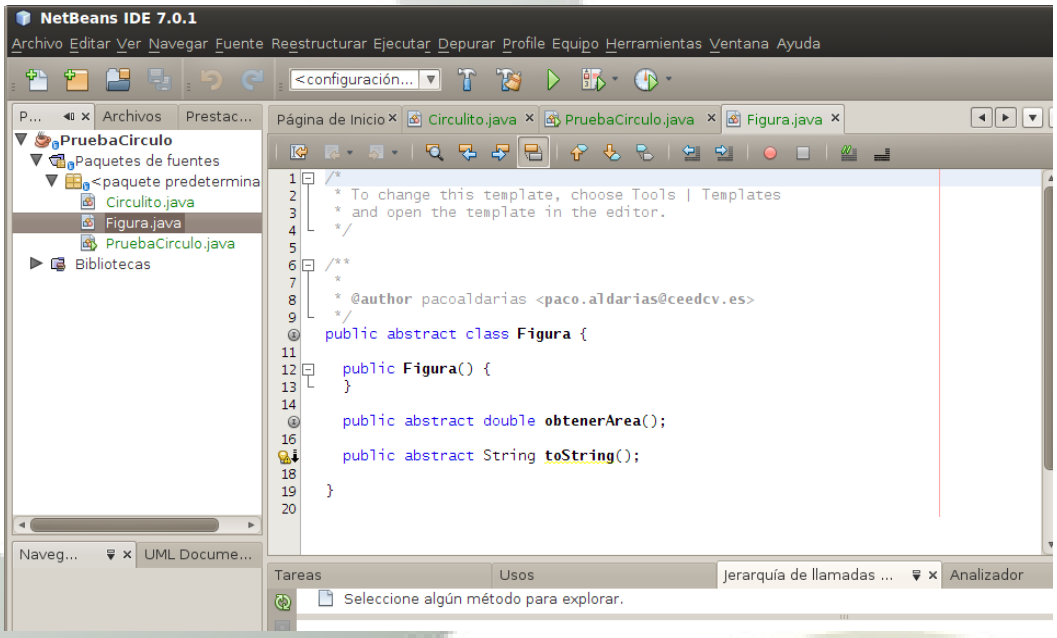

Figura 18: Refactorización con Netbeans

Entregar el proyecto con los cambios y una captura del escritorio.

BIBLIOGRAFÍA

[FBB99] MARTIN FOWLER, KENT BECK, and JOHN BRANT. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1 edition, 1999. ISBN: 0-201-485672.

[PRE02] ROGUER. S. PRESSMAN. *Ingeniería del Software. Un enfoque práctico.* McGrawHill, 5 edition, 2002. ISBN: 8448132149.