Práctica 5.
Aplicación de Notas: Diseño de Objetos.

Grado en Ingeniería Informática - Ingeniería del Software
Dpto. de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Introducción y objetivos

Para realizar la presente práctica es necesario:

- Haber desarrollado el proyecto Android de la aplicación de gestión de *Notas* en su versión número 3 tal como se explicó en la práctica 4. Este proyecto se utilizará como base para realizar el trabajo de la asignatura.
- Tener un conocimiento básico de Modelio. La herramienta Modelio se presentó en la práctica 2 de la asignatura.

Los objetivos de la práctica son los siguientes:

Objetivo 1. Aplicar el conocimiento adquirido en diseño de objetos para añadir una nueva funcionalidad a la aplicación: el *envío de notas*, tanto por correo electrónico como por SMS. Esta nueva funcionalidad se desarrollará con el patrón de diseño *Bridge*.

Objetivo 2. Utilizar Modelio [2] para añadir detalles del diseño de objetos en las clases del modelo de la aplicación de notas. Se detallarán las características de las clases e interfaces que intervienen en el patrón *Bridge*.

Objetivo 3. Generar la documentación de diseño detallada a partir de la documentación introducida en el propio código fuente, utilizando la herramienta *Javadoc*.

2. Actividades a realizar en la práctica

2.1. Actualización de los diagramas de la práctica 4 para reflejar la nueva funcionalidad de la aplicación

En esta primera parte de la práctica vais a añadir una nueva funcionalidad a la aplicación: el *envío de notas por correo*.

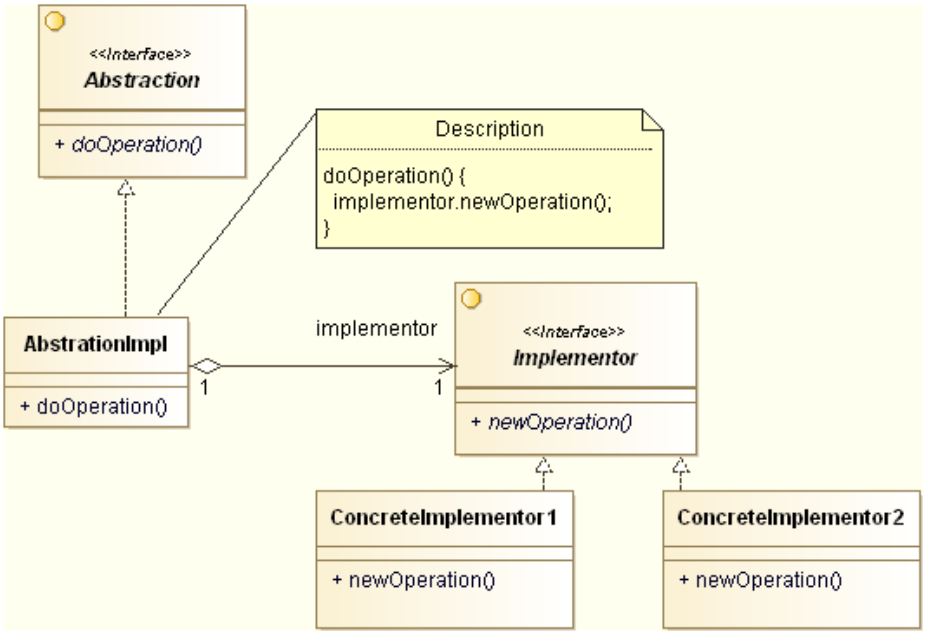

Figura 1: Estructura del patrón *Bridge*.

Para desarrollar esta nueva funcionalidad vais a aplicar el patrón de diseño *Bridge* [1]. Un patrón de diseño proporciona una solución, en forma de plantilla, a un problema que ocurre repetidamente en el desarrollo de software.

En particular, el patrón *Bridge* permite desacoplar una abstracción de su implementación para que las dos puedan variar independientemente. En la figura 1 se muestra la estructura del patrón:

- La interfaz *Abstraction* define la interfaz de la abstracción.

- La clase *AbstractionImpl* implementa la interfaz de la abstracción utilizando (delegando en) una referencia a un objeto de tipo *Implementor*.
- La interfaz *Implementor* define la interfaz para las clases de la implementación. La interfaz no tiene por qué corresponder directamente con la interfaz de la abstracción.
- Las clases *ConcreteImplementor1*, *ConcreteImplementor2*, etc., implementan la interfaz *Implementor*.

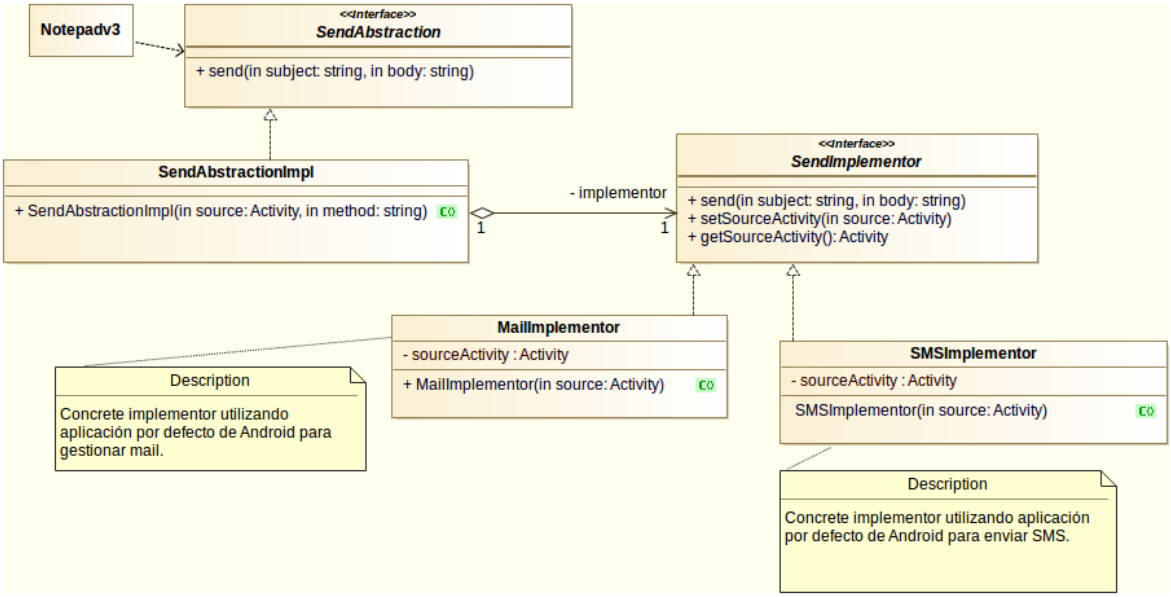

Figura 2: Adaptación del patrón *Bridge* para el envío de notas por correo.

En el caso de la aplicación de notas, vamos a utilizar el patrón *Bridge* para separar la abstracción de enviar notas del mecanismo específico que podríamos utilizar dependiendo del entorno de ejecución. En la figura 2 se muestra la adaptación del patrón *Bridge* al contexto de la aplicación de notas. Observa este diagrama de clases sabiendo que:

- La interfaz *SendAbstraction* define la interfaz de la abstracción del envío de un mensaje (a través del método `send()`).
- La clase *SendAbstractionImpl* implementa la interfaz de la abstracción, delegando el envío de mensajes a un objeto de tipo *SendImplementor*.
- El objetivo de la clase *MailImplementor* es implementar la interfaz *SendImplementor* utilizando la aplicación por defecto de Android para gestionar el correo electrónico.
- El objetivo de la clase *SMSImplementor* es implementar la interfaz *SendImplementor* utilizando la aplicación por defecto de Android para enviar SMS.

Modificad con *Modelio* el modelo de la aplicación de notas construido en la práctica 4 para:

- Reflejar la nueva funcionalidad del sistema en el diagrama de casos de uso.
- Incluir las clases e interfaces que intervienen en el patrón *Bridge*. Cread un nuevo diagrama de clases a semejanza del que aparece en la figura 2, incluyendo detalles de visibilidad, tipos de atributos y signatura de métodos.

2.2. Modificación de la aplicación *Notepad* para enviar notas a través de la aplicación de mail de Android

Vais a modificar la aplicación de *Notas* para enviar notas a través de la aplicación de correo proporcionado en Android por defecto. Es decir, vamos a implementar parte del diagrama de clases mostrado en la figura 2. En esta sección veréis cómo enviar notas a través de la clase *MailImplmementor*.

Vamos a tomar como punto de partida el proyecto Android modificado en la práctica anterior.

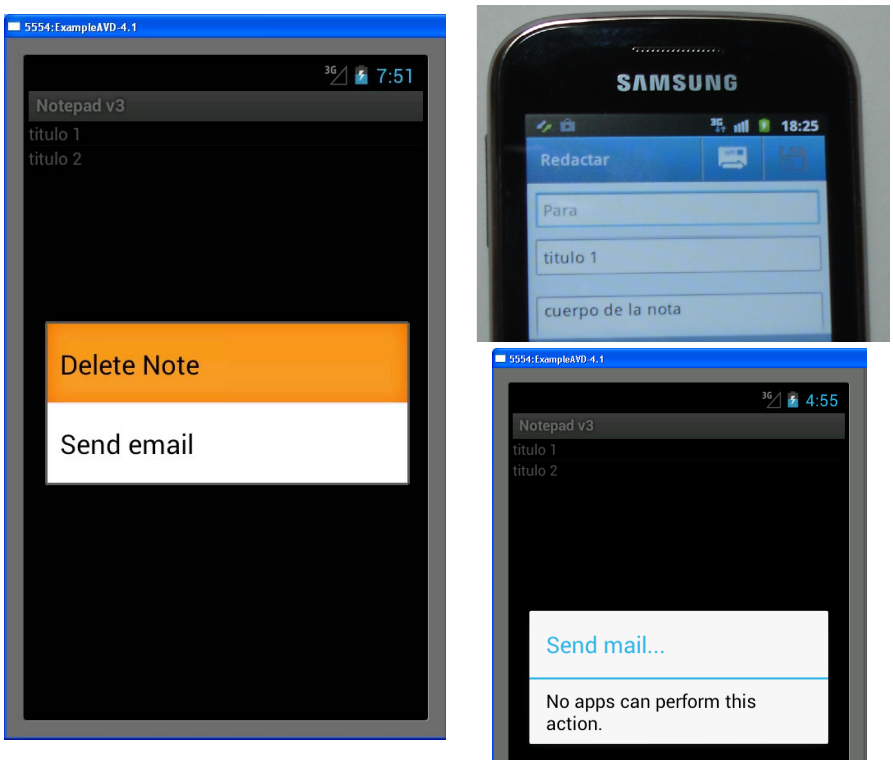

Figura 3: Envío de notas en la aplicación *Notepadv3* utilizando *MailImplementor*.

A continuación, vais a seguir los siguientes pasos:

Paso 1. Descargad de *Moodle* el fichero "softwarePractica5.zip" con el software necesario para el desarrollo de la práctica 5.

Paso 2. Cread el paquete *es.unizar.eina.send* en vuestro proyecto Android, añadiendo las siguientes clases e interfaces: *SendAbstraction*, *SendAbstractionImpl*, *SendImplementor* y *MailImplementor*.

Paso 3. Modificad la clase *Notepadv3* para añadir una nueva opción en el menú contextual que permita enviar por correo una nota previamente seleccionada (véase la parte izquierda de la figura 3) utilizando la interfaz *SendAbstraction* a través de una instancia de *SendAbstractionImpl*. El valor del parámetro *method* es irrelevante para esta primera versión de la aplicación. Tened en cuenta las siguientes consideraciones:

- Para modificar el menú contextual, observad cómo se realiza actualmente la eliminación de notas.
- Para recuperar el título y el cuerpo de la nota, examinad el procedimiento usado actualmente en la clase *NoteEdit* para la modificación de una nota existente.

Paso 4. Implementad el método `send()` de la clase *MailImplementor* con el siguiente código:

```
Intent emailIntent = new Intent(android.content.Intent.ACTION_SEND);
emailIntent.putExtra(android.content.Intent.EXTRA_SUBJECT, subject);
emailIntent.setType("plain/text");
emailIntent.putExtra(android.content.Intent.EXTRA_TEXT, body);
getSourceActivity().startActivity(Intent.createChooser(emailIntent, "Send
    mail..."));
```

Si probáis a ejecutar la aplicación con el móvil, se abrirá la aplicación por correo que tengáis configurada por defecto (como se muestra en la parte superior derecha de la figura 3). Sin embargo, si utilizáis el emulador, comprobaréis que no es posible enviar las notas si no se ha configurado previamente la aplicación de correo (*Mail*) en el emulador (aparecerá un error similar al que se muestra en la parte inferior derecha de la figura 3). Como dejar configurado vuestro correo en el emulador no es demasiado recomendable, es preferible que depuréis la aplicación sobre vuestros propios dispositivos Android[1].

Paso 5. Por último, deberéis **crear un nuevo diagrama de secuencia** que muestre la interacción entre los objetos del sistema de acuerdo al mecanismo de envío de notas que acabáis de implementar.

[1] En `http://developer.android.com/training/basics/firstapp/running-app.html` se explica cómo depurar una aplicación sobre dispositivos reales conectados con un cable USB.

2.3. Modificación de la aplicación *Notepad* para enviar notas por SMS

Vais a modificar vuestro proyecto Android para implementar la clase *SMSImplementor* descrita en la figura 2. Es decir, en el método *send* de la clase *SMSImplementor* enviaréis la nota por SMS lanzando la aplicación por defecto de Android con las siguientes líneas de código:

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.putExtra("sms_body", subject+": "+body);
intent.setType("vnd.android-dir/mms-sms");
getSourceActivity().startActivity(intent);
```

Para que funcione correctamente, debéis autorizar el envío de SMS en el fichero de manifiesto («`AndroidManifest.xml`») de vuestra aplicación:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Se debe modificar también el constructor de la clase *SendAbstractionImpl* para instanciar el *SendImplementor* adecuado dependiendo del método elegido:

```
        if (method.equalsIgnoreCase("SMS"))
                implementor = new SMSImplementor(sourceActivity);
        else
                implementor = new MailImplementor(sourceActivity);
```

Por último, será necesario modificar la instanciación de *SendAbstractionImpl* con el método deseado (valor del parámetro *method*) desde la clase *Notepadv3*. Si se considera conveniente, también se puede ajustar el método a utilizar en función del tamaño de la nota seleccionada: por ejemplo, un SMS para notas cortas (menor de 100 caracteres) y un correo electrónico para notas más largas.

Modificad el diagrama de secuencia construido en la actividad anterior (ver sección 2.2) para reflejar las 2 alternativas de mecanismos de envío utilizados.

2.4. Generación de documentación de diseño detallada

Vais a generar la documentación de diseño detallada a partir de la documentación introducida en el propio código fuente utilizando la herramienta automática *Javadoc*.

En el apéndice A se incluye una guía de estilo de codificación en Java con recomendaciones que son aconsejables seguir para facilitar la legibilidad y reusabilidad del código fuente, tanto dentro de un equipo de desarrollo como para facilitar la transferencia del código fuente a otros equipos de desarrollo.

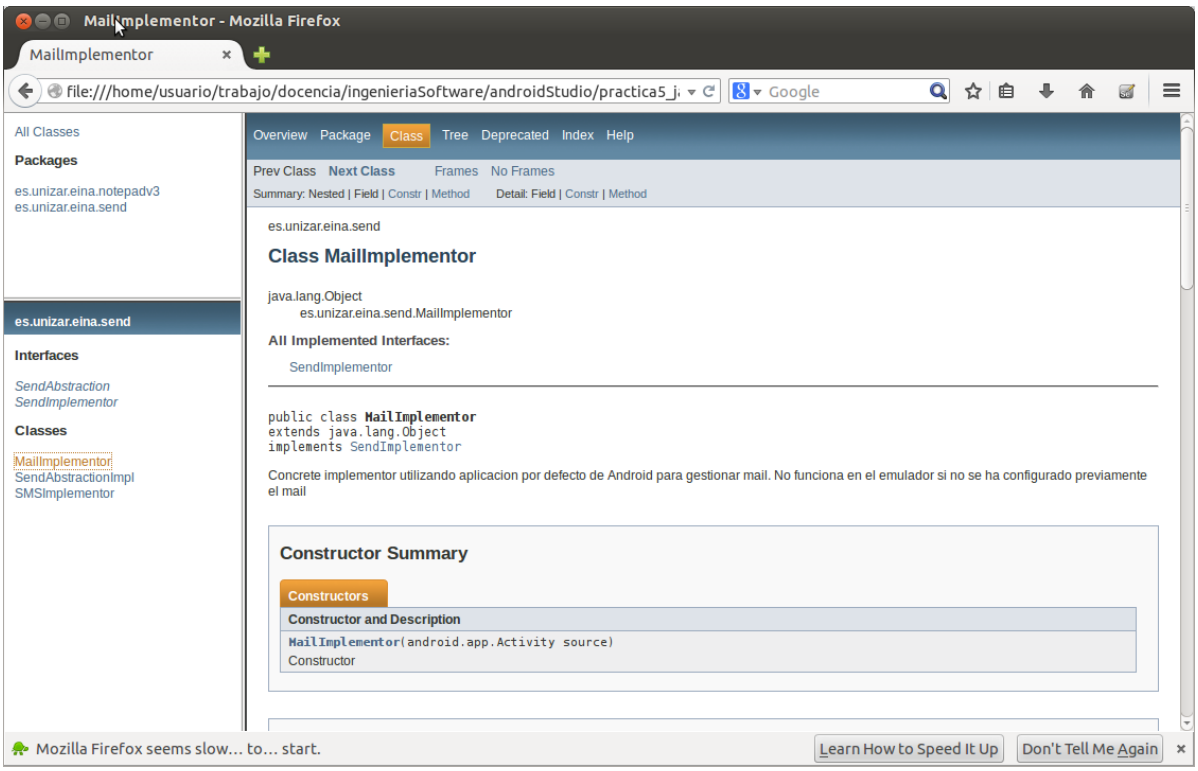

Figura 4: Ejemplo de la documentación generada para el proyecto *Notepad*.

Dentro de esta guía de estilo, la sección A.5.2 describe brevemente cómo documentar el código fuente para que se pueda generar la documentación de forma automática con la herramienta *Javadoc*. La figura 4 muestra un ejemplo de la documentación generada para el proyecto *Notepad*.

2.5. Modificación del diagrama de despliegue

Modificad el diagrama de despliegue construido en la práctica anterior (véase la figura 5) para reflejar que con esta nueva funcionalidad de envío de notas nos estamos conectado con dos nodos externos: en el caso de envío de notas mediante correo electrónico, el nodo correspondiente al servidor de correo saliente que hemos configurado en la aplicación de correo de nuestro dispositivo; o en el caso de envío de notas mediante SMS, el nodo correspondiente a la central de nuestro proveedor de servicios de telefonía.

3. Entrega de la práctica

A través de una tarea accesible en Moodle, subiréis dos documentos:

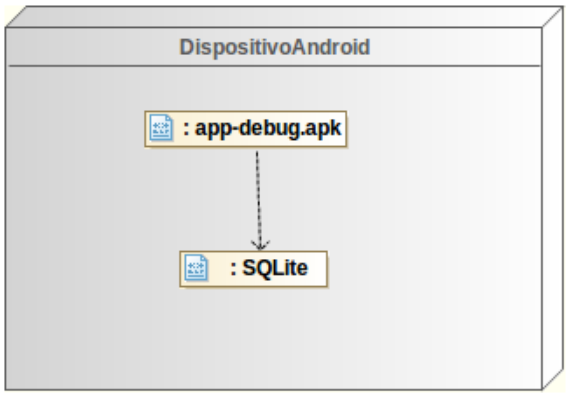

Figura 5: Diagrama de despliegue de la aplicación *Notepad*.

- Un documento con formato PDF donde se incluirán las imágenes correspondientes a las capturas de los diagramas de secuencia, clases, paquetes, componentes y despliegue creados a lo largo del desarrollo de esta práctica con la herramienta Modelio. Cada diagrama irá acompañado de un título descriptivo en dicho documento.
- Un fichero comprimido con la documentación generada por la herramienta *Javadoc*.

La fecha límite para subir este documento a Moodle será el **día anterior a la siguiente sesión de prácticas de cada grupo**. Durante la sesión de la práctica 6, presentaréis al profesor de prácticas el resultado de esta práctica. Recordad que para superar la asignatura hay que presentar todas las prácticas.

A. Guía de estilo de codificación en Java

La experiencia demuestra que las normas establecidas en una guía de estilo de codificación son beneficiosas para reducir errores semánticos y demandan cierta consistencia en la elección de nombres y la organización. Adicionalmente, tienen por objeto asegurar la uniformidad de código entre los distintos programadores. Esta guía está basada en [3].

A.1. Formato de líneas

A.1.1. Indentación

La anchura de línea será de 80 caracteres como máximo y la indentación de 4 caracteres. Un número mayor de caracteres no mejora la legibilidad y hace que se desaproveche

mucha pantalla.

En estructuras compuestas la llave de apertura se pone al final de la línea, la de cierre aparecerá aislada en una línea propia y al mismo nivel de indentación que el elemento asociado:

```
class VectorEntero {
  ...
}
```

Cuando se tenga que dividir una línea se seguirán los siguientes principios:

- Romper después de una coma u operador y alinear al mismo nivel sintáctico:

```
void divisionEntera(Integer dividendo, Integer divisor,
                    Integer cociente, Integer resto) {
  ...
}
```

- Si no es viable alinear al mismo nivel sintáctico, se hará una tabulación a 8 espacios:

```
public static void nombreDeMetodoMuyLargo(Integer dividendo,
       Integer divisor, Integer cociente, Integer resto) {
  ...
}
```

A.1.2. Espaciado

Se utilizarán líneas en blanco libremente para separar partes lógicas distintas de una función. Entre dos métodos habrá siempre una línea en blanco y entre dos clases o interfaces, dos líneas en blanco.

Se debe dejar un espacio en blanco a ambos lados de un operador binario, así como a la derecha de una coma, de un punto o de una coma no terminadores:

```
f(a, -b * c);

for (int i = 1; i < 10; ++i) ...
```

Después de `if`, `while`, `switch` y `for` se dejará un espacio en blanco para no confundirlo con una llamada a un método:

```
while (x < 3) {
     ...
}
```

A.2. Definición de clases

Dentro de la clase se ordenarán los elementos así: variables estáticas, atributos, constructores y métodos.

Dentro de cada grupo aparecerán en primer lugar los elementos públicos, luego los protegidos y por último los privados.

Los atributos deben declararse privados (`private`), excepto los que se pretenda que sean accesibles por herencia, que deben ser protegidos (`protected`). **Nunca deben declararse atributos públicos**.

Se utilizarán métodos de acceso para acceder y cambiar el valor de los atributos con el mismo nombre que el atributo, pero anteponiendo *get* (o *is* para el caso de atributos *booleanos*) o *set* y con la primera letra del atributo en mayúsculas:

```
public int getSalario()           // Para acceder al valor
public void setSalario(int s)     // Para establecer el valor

public boolean isVisible()        // Para acceder al valor
public void setVisible(boolean v) // Para establecer el valor
```

Los métodos que tienen que ser accesibles desde el exterior de una clase deben declararse como públicos (`public`) y los de soporte privados (`private`).

Para acceder a un elemento estático se usará el nombre de la clase y no el de un objeto particular:

```
VectorEntero v;
VectorEntero.dimMaxima(); // Sí.  Es común a todos los vectores
v.dimMaxima(); // No
```

A.3. Identificadores

Deben ser descriptivos y no deben usarse abreviaturas, con la excepción de aquellas que tengan un significado claro:

```
int longi;   // No
int longitud; // Sí
int max;      // Máximo, sí
```

Para contadores, puede usarse una letra:

```
for (int i = 1; i < 10; ++i) {
    ...
}
```

Si un identificador consta de más de una palabra, las letras iniciales de la segunda y siguientes se distinguirán del resto intercalando letras mayúsculas. Este tipo de notación se denomina *Camel case*[2]

```
int precioTotal;
```

Los nombres de las clases y de los interfaces empezarán por letra mayúscula. Las variables, los paquetes, los atributos y los métodos empezarán por minúscula:

```
package paquete;

public class Empleado {
    private String nombre;
    public int salario();
}
```

En el caso de nombres de constantes, se recomienda que se utilicen como identificadores letras mayúsculas. Si el identificador consta de más de una palabra, se separan utilizando el carácter '_'.

```
public static final double PRECIO_MAXIMO = 1000.00;
```

A.4. Instrucciones de control

En general es preferible utilizar varias sentencias simples que codificarlas en una sola como a menudo se hace en C/C++:

```
++y;
x = f(y);
++z;
x = f(++y, z++); //Menos claro
```

Es buena política incluir siempre llaves en las instrucciones compuestas aunque haya una única subsentencia. De este modo evitamos el error de añadir una sentencia y olvidarnos de poner las llaves.

Una instrucción condicional se estructurará así:

```
if (condicion) {
   ...
} else {
   ...
}
```

[2]Puedes leer más sobre ella en la Wikipedia `https://es.wikipedia.org/wiki/Camel_case`.

En el caso de `switch` se pondrán siempre `break` al final de cada caso, incluso aunque no haya caso por defecto. De todas formas, es recomendable que toda instrucción `switch` tenga un caso por defecto.

```
switch (...) {
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default:
        ...
}
```

Por último, un `try-catch-finally` se indentará de la siguiente forma:

```
try {
    ...
} catch (...) {
    ...
} finally {
    ...
}
```

A.5. Documentación

Hay que distinguir dos tipos de documentación:

- la documentación interna o de implementación para quienes tengan que mantener el código de la clase;
- la documentación externa o de interfaz para los usuarios de la misma.

A.5.1. Documentación interna de implementación

Los aspectos de diseño de detalle que resulten relevantes se documentarán en el propio código fuente.

Cuando el comentario sea de una sola línea se comenzará con "//..." (comentario mono-línea) mientras que si abarca a más de una línea se utilizará la notación de comentario multi-línea:

```
/* ...
   ...
```

```
    ... */
```

A.5.2. Documentación externa de interfaz (utilización de *Javadoc*)

Explica fundamentalmente el interfaz de la clase, para qué sirve la clase, cuáles son sus distintos elementos (métodos, constantes, campos, variables estáticas) y cuál es su función. En el caso de los métodos, se explicará **clara pero brevemente** qué es lo que hacen, independientemente de su implementación.

De cara a usar un generador automático de documentación como *Javadoc* los comentarios tendrán la siguiente estructura:

```
    /** ... */
```

Javadoc genera páginas HTML (y en otros formatos) a partir de los ficheros de código fuente de Java. Además, se pueden añadir etiquetas especiales que especifiquen campos concretos en esa documentación como: `@author`, `@version`, `@param`, `@see`, `@return` o `@exception`.

Por ejemplo, dado el código fuente que se muestra a continuación, se podría invocar a la herramienta *Javadoc* y generar las páginas HTML que se muestran en la figura 6.

```
/**
  * Clase padre de la jerarquía de figuras geométricas
  * @author un autor
  */
public abstract class Figura {

    /** nombre de la figura */
    private String nombre;

    /**
     * Devuelve el área de esta figura
     * @return área de esta figura
     * @see Circulo
     * @see Rectangulo
     */
    abstract public float area();

    /** Dibuja esta figura en la pantalla */
    abstract public void dibuja();

    /** Constructor
     * @param nombreFigura nombre de esta figura
     */
    public Figura(String nombreFigura) {
```

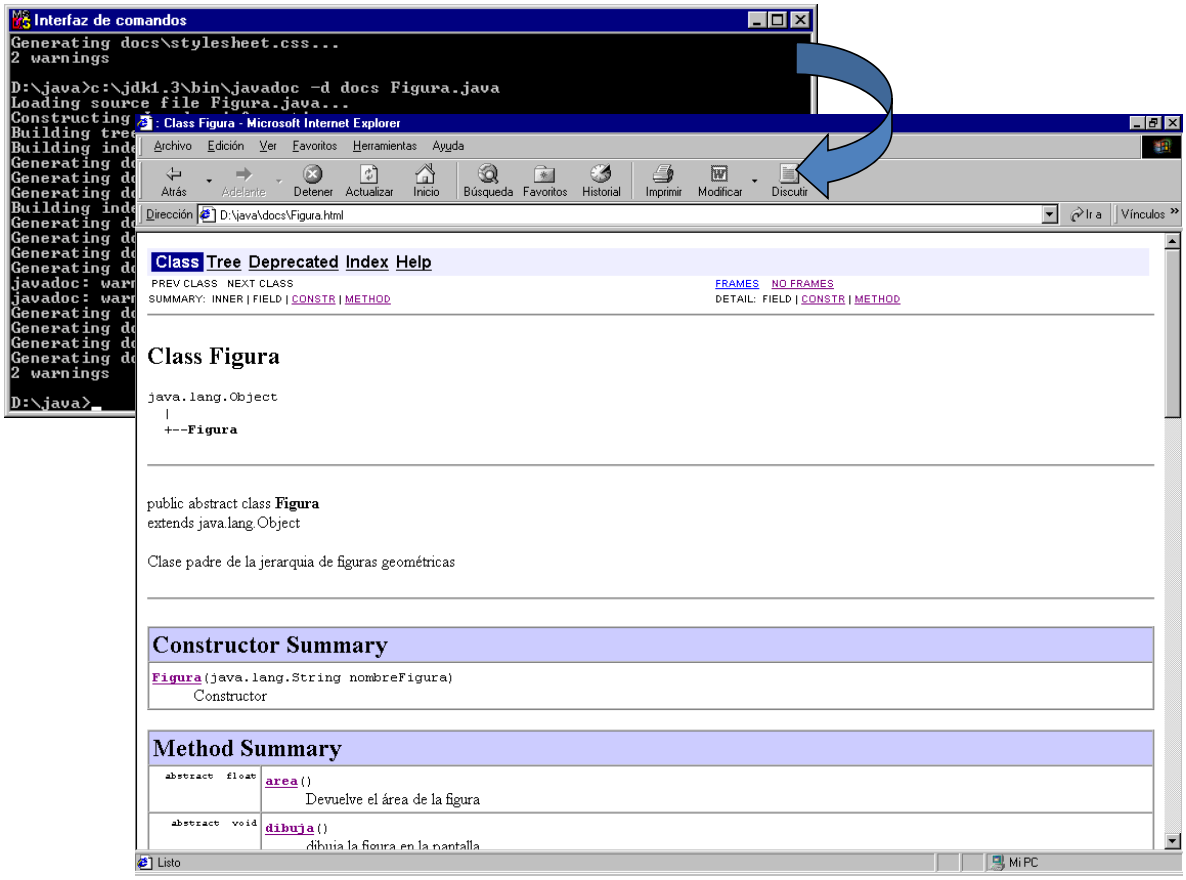

Figura 6: Ejemplo de generación de documentación con la herramienta *Javadoc*.

```
    nombre = nombreFigura;
}

/**
 * Devuelve verdad si y solo si el área de esta figura
 * es menor que la que se pasa como parámetro
 * @param lDer   figura a comparar
 * @return verdad si el área de esta figura
 * es menor que la de <code>lDer</code>
 */
final public boolean menorQue(Figura lDer) {
    return area() < lDer.area();
}

/** Método toString redefinido */
final public String toString() {
    return nombre + " con área " + area();
}
```

```
    public static void main(String[] args) {
        Figura a = new Circulo(2f);
        Figura b = new Rectangulo(2f,3f);
        System.out.println(a);
        System.out.println(b);
    }
}
```

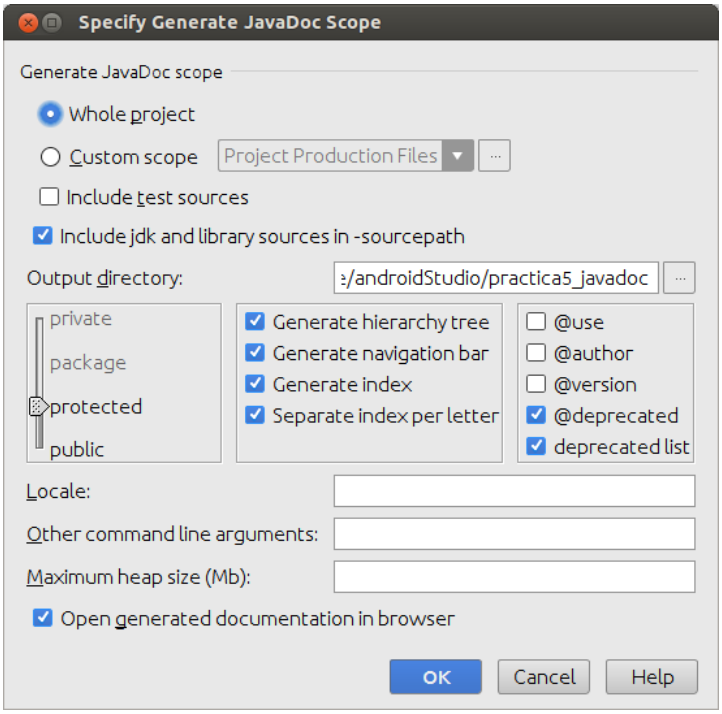

Figura 7: Generación de documentación con la herramienta *Javadoc* desde Android Studio.

Por último, se debe mencionar que desde el entorno de Android Studio se facilita la utilización de la herramienta *Javadoc* a través del cuadro de diálogo que se despliega con la opción de menú Tools » Generate Javadoc (mostrado en la Figura 7):

- Se debe marcar la opción *"Include jdk and library sources ..."*.
- Se debe configurar el directorio de salida de la documentación utilizando la caja de texto *"Output directory"*.
- Se puede configurar el nivel de detalle de la documentación seleccionando los miembros a incluir según sea su visibilidad (*private*, *package*, *protected* o *public*).

Referencias

[1] GeeksforGeeks. Bridge Design Pattern. `https://www.geeksforgeeks.org/bridge-design-pattern/`.

[2] Modelio. Modelio: the open source modeling environment. `http://www.modelio.org/`.

[3] Inc. Sun Microsystems. Code conventions for the java™ programming language, April 1999.