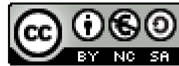
Práctica 3
Gráficos Vectoriales con SVG
(versión 11.10.27)

Programación 3
Curso 2011-2012

David Rizo Valero
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante



1. Introducción

En esta tercera práctica implementaremos una pequeña aplicación que, a partir de un fichero de texto de entrada que contendrá sencillas órdenes de dibujo, generará un nuevo fichero de texto con la especificación del gráfico a generar, utilizando para ello un subconjunto del lenguaje de gráficos vectoriales *Scalable Vector Graphics Tiny 1.2*[1], al cual nos referiremos de ahora en adelante como SVG. Para ello nos apoyaremos en la librería desarrollada en la anterior práctica, la cual, enriqueceremos con nuevas propiedades y el tratamiento de errores mediante excepciones.

2. El diagrama de clases

Las nuevas propiedades son el color y grosor de la línea de su perímetro y el color de relleno de la figura. Para modelar el color introducimos el enumerado *Color* que encapsula los colores que podemos usar en SVG.

Un cambio significativo de esta práctica a la anterior es que en ésta tenemos que implementar una aplicación, no sólo una librería de clases. Para ello hemos creado la clase *DibujoSVG* que representa esa aplicación. Es una clase *singleton*,

[1] `http://www.w3.org/TR/SVGTiny12/`

lo cual significa que puede existir como máximo una instancia de esa clase en nuestra aplicación. El patrón de diseño[2] *singleton* obliga a que sólo exista una instancia como máximo de una clase dada, al tiempo que hace esta instancia accesible desde cualquier lugar de la aplicación. La clase *DibujoSVG* encapsula además los flujos de entrada y salida, así como la representación en memoria del dibujo (es decir, un *Lienzo*).

Esto nos lleva a un diseño de clases para nuestra pequeña aplicación como el de la Figura 1.

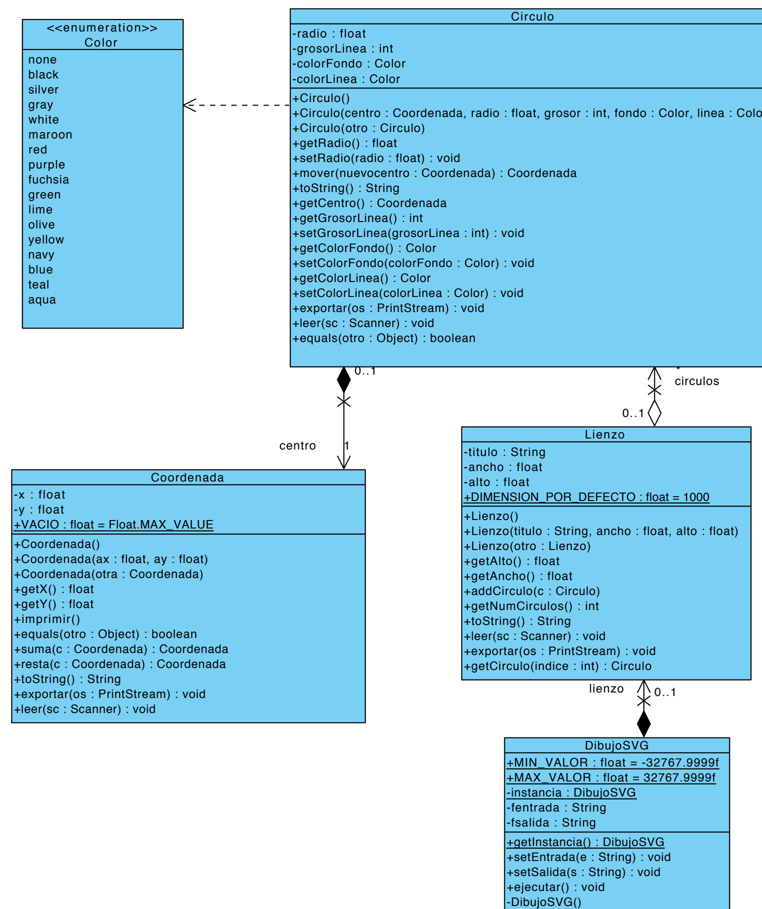


Figura 1: Diagrama de clases UML.

Se pide implementar este diagrama de clases con el comportamiento de la aplicación que se especifica a continuación.

[2]Un *patrón de diseño* es un modo formal de describir soluciones simples y elegantes a problemas específicos en el diseño orientado a objetos.

3. Formato de entrada

El objetivo de nuestra aplicación es traducir un fichero de entrada en un dibujo vectorial SVG. En esta versión de la aplicación, el formato de entrada será el de la Figura 2, donde <título> y <tipo objeto> son cadenas; el título puede estar vacío. <ancho lienzo>, <altura lienzo> y <valor> son números enteros o reales con la misma sintaxis que en Java [3]. Nótese que la última línea acaba con un <EOL>.

```
<ancho lienzo>␣<altura lienzo>␣<título><EOL>
{<tipo objeto>␣<valores>*<EOL>|HACER␣<operacion>␣<id
fig>␣<valores>*<EOL>}*
<EOF>
```

Figura 2: Formato de entrada general.

Como se puede ver, en la primera línea se define el lienzo y en cada una de las líneas restantes se define una figura o se indica una operación a realizar (HACER). Cada línea contiene:

<tipo objeto>␣$< x_0 >$␣$< y_0 >$␣<grosor>␣<color fondo>␣<color línea>␣<valores específicos de la figura>

donde <tipo objeto> será siempre Circulo [4] en esta práctica, $< x_0 >$ e $< y_0 >$ son las coordenadas del origen, <grosor> es un valor numérico entero para el grosor de la línea de perímetro de una figura, <color fondo> es el nombre de color de relleno de la figura y <color línea> es el nombre de color para la línea. Los colores vienen especificados en el UML [5]. En ese listado de colores, 'none' no es en realidad un color en SVG. En nuestra aplicación representa la ausencia de color. Estos campos son obligatorios para todas las figuras.

Finalmente, <valores específicos de la figura>, en el caso del tipo Circulo, corresponderá a un valor real que define el radio del círculo.

En la evaluación de la práctica sólo se usarán ficheros de entrada con un **formato** o sintaxis válida. Sin embargo, se debe tener en cuenta que los **valores** proporcionados en estos ficheros pueden no ser válidos, extremo que se deberá controlar mediante el uso de excepciones que describiremos más adelante (página 5).

[3] En la figura, <EOL> indica fin de línea, <EOF> indica fin de fichero y ␣ indica uno o más espacios en blanco; un asterisco '*' tras un elemento significa *cero o más* elementos y las llaves '{','}' agrupan campos.

[4] En siguientes versiones se ampliará la variedad de figuras

[5] http://www.w3.org/TR/SVGMobile12/painting.html#Color

3.1. Operaciones con figuras

En esta versión de la aplicación, la única operación que se puede realizar con las figuras es moverlas. Esto se especifica en el fichero de entrada de la siguiente forma:

```
HACER mover␣<id fig>␣<x destino>␣<y destino>
```

donde <id fig> es un número entero que indica la figura a mover. La primera figura definida en el fichero tiene identificador 1, la siguiente 2 y así sucesivamente. Los otros valores son los de la coordenada a la que hay que mover la figura (es decir, el nuevo origen de la figura).

En la figura 3 se puede ver un ejemplo de un fichero de entrada completamente válido, donde se define un círculo de radio 10, ubicado en la coordenada 500, 250, con grosor de línea 17, color de fondo 'red' y color de línea 'blue'. A continuación se mueve esa misma figura a la coordenada 100, 300. En las tres siguientes líneas se crea un nuevo círculo y se mueve dos veces seguidas.

```
1000␣500␣Dos␣mas␣dos␣son␣cuatro
Circulo␣500␣250␣17␣red␣blue␣10
HACER␣mover␣1␣100␣300
Circulo␣100␣150␣17␣green none␣20
HACER␣mover␣2␣100␣200
HACER␣mover␣2␣200␣250
```

Figura 3: Ejemplo de entrada

4. Formato de salida

Una vez construidos los objetos en memoria, el objetivo de la aplicación es escribir un fichero de salida en formato SVG con cada una de las figuras a dibujar. Por ejemplo, para la entrada de la Figura 3, se debe producir la siguiente salida (los espacios en blanco y saltos de línea son libres)[6]:

```
<?xml version="1.0"?>
<svg width="1000.0" height="500.0" version="1.2" baseProfile="tiny" xmlns="http://www.w3.org/2000/svg">
<desc>Dos mas dos son cuatro</desc>
<circle cx="100.0" cy="300.0" fill="red" stroke="blue" stroke-width="17" r="10.0" />
<circle cx="200.0" cy="250.0" fill="green" stroke-width="17" r="20.0" />
</svg>
```

[6]Ver la documentación de SVG para más información

El orden en el que aparecen en el fichero de salida las figuras a dibujar debe ser el mismo que el orden en el que aparecen en el fichero de entrada. Las figuras deben aparecer a continuación de la etiqueta *desc*. En la siguiente sección se explican más detalles del formato de salida. El orden en los parámetros (width, height, etc...) de cada elemento del XML no es importante.

El número de decimales exportado será el que saque por defecto el método `print(float)`.

4.1. Manejo de errores

Para manejar las condiciones de error que se pueden dar en la aplicación, se incorpora el tratamiento de errores mediante excepciones. Los errores que se han de tratar mediante excepciones son la creación de coordenadas fuera de rango (bien como resultado de leer el fichero de entrada o de realizar una operación con coordenadas), así como la especificación de figuras u operaciones desconocidas o valores no válidos.

Como resultado de la gestión de errores, no se creará ningún objeto con valores no válidos, ya que los constructores lanzarán excepciones en esos casos.

Esto implica que no vamos a necesitar más la constante `Coordenada.COORDENADA_ERROR` y por tanto la debemos eliminar de Coordenada.

Con el objetivo de poder tener información específica del error que se ha producido, vamos a crear el conjunto de clases de la figura 4 que ubicaremos en el paquete `modelo.excepciones`.

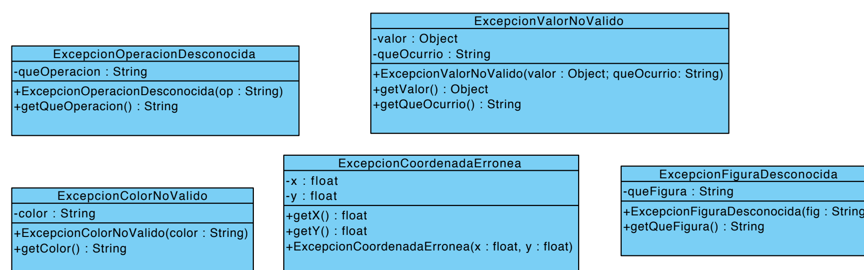


Figura 4: Excepciones.

Recuerda que la implementación de una excepción se realiza extendiendo la clase `Exception`.

La especificación de qué métodos lanzan qué excepciones se detalla a continuación, en la descripción de los métodos de las clases.

5. Interfaz de las clases

Se describe a continuación los métodos de las clases. No se describen los atributos ni relaciones dado que ya aparecen en el UML ni los que no cambian respecto a la práctica anterior. Tampoco se detallan más los métodos *set* o *get*, que, apareciendo en el UML, no hacen nada distinto a devolver o asignar un valor a la propiedad asociada, o en el caso de los *set*, además, comprobar si los valores a asignar son correctos.

5.1. Coordenada

El constructor por defecto y de copia son los mismos que los de la práctica anterior.

Coordenada(float x, float y) throws ExcepcionCoordenadaErronea. Este constructor lanza una excepción cuando la coordenada está fuera del rango definido por *DibujoSVG.MIN_VALOR* y *DibujoSVG.MAX_VALOR*.

Coordenada suma(Coordenada c) throws ExcepcionCoordenadaErronea. Cuando esta operación genera una coordenada no válida se lanza esta excepción [7].

Coordenada resta(Coordenada c) throws ExcepcionCoordenadaErronea. Análogo a `suma`.

boolean equals(Object obj) En esta práctica cambiamos el tipo de dato del parámetro para evitar problemas con los tests unitarios JUnit.

void exportar(PrintStream os) Escribe en el flujo de salida pasado por parámetro una coordenada conforme la especificación SVG:

```
cx="<valor x>" cy="<valor y>"
```

donde `<valor x>` y `<valor y>` se sustituyen por lo correspondientes valores de la coordenada.

[7] Puedes apoyarte en el constructor para no tener que duplicar el código de comprobación de rango

Para escribir valores en el flujo de salida puedes utilizar los métodos print y println de la clase PrintStream [8].

void leer(Scanner is) throws ExcepcionCoordenadaErronea Leerá dos reales de la entrada (tratada por la clase Scanner [9]), y asigna esos dos reales a x e y en ese orden. Se realizará la misma comprobación de rango que en el constructor, lanzando la excepción ExcepcionCoordenadaErronea si no se respeta ese rango.

Para leer float puedes usar el método nextFloat de Scanner.

5.2. Circulo

void setRadio(float pradio) throws ExcepcionValorNoValido Este método lanza una excepción cuando el radio es negativo con el código:

```
throw new ExcepcionValorNoValido(pradio,"Radio");
```

void setGrosorLinea(int gl) throws ExcepcionValorNoValido Asigna el grosor de línea lanzando una ExcepcionValorNoValido con la cadena "Grosor" cuando el grosor es negativo.

Circulo(Coordenada centro, float radio, int grosor, Color fondo, Color linea) throws ExcepcionValorNoValido Asigna valores a las propiedades a partir de los parámetros comprobando la validez de éstos. Es aconsejable usar los métodos *set* para ello dado que éstos ya realizan esa comprobación.

void exportar(PrintStream os) Emite la cadena

```
<circle cx="<centro x>" cy="<centro y>" fill="<color fondo>" stroke="<color línea>" stroke-width="<grosor línea>" r="<radio>" />
```

donde cx="<centro x>" cy="<centro y>" se puede generar usando el método exportar de Coordenada y los colores se emiten simplemente haciendo os.print(colorFondo) o (colorLinea). En el caso de que un color sea 'none', no se imprime ni el parámetro ni el valor (véase el segundo círculo del ejemplo de salida en la página 4).

[8] véase documentación en la API de Java http://download.oracle.com/javase/6/docs/api/java/io/PrintStream.html

[9] http://download.oracle.com/javase/6/docs/api/java/util/Scanner.html

void leer(Scanner sc) throws ExcepcionValorNoValido, ExcepcionCoordenadaErronea, ExcepcionColorNoValido La lectura del círculo a partir del fichero de entrada debe controlar la corrección de los valores leídos. Para ello puede aprovechar el método `leer` de la propiedad `centro` que lanza una excepción si la coordenada no es correcta. También para aprovechar el código ya escrito, podemos usar los métodos `setGrosorLinea` y `setRadio`. Para la lectura de los valores enumerados, se puede usar el siguiente código de ejemplo que captura la excepción lanzada por Java cuando la cadena leída no corresponde a ningún identificador del enumerado de colores, y lanzamos nuestra propia `ExcepcionColorNoValido`:

```
String s="";
try {
    s = sc.next(); // lee la siguiente línea
    this.colorFondo = Color.valueOf(s);
} catch (IllegalArgumentException e) {
    throw new ExcepcionColorNoValido(s);
}
```

boolean equals(Object obj) Devolveremos cierto cuando todos los campos de ambos objetos sean iguales, pudiendo usar el generador de código de Eclipse, y adaptándolo si es necesario para que el argumento sea del tipo `Object`.

5.3. Lienzo

Lienzo(String ptitulo, float pancho, float palto) throws ExcepcionValorNoValido En esta práctica debemos controlar que el ancho y el alto no sean negativos (pueden ser cero), lanzando una excepción `ExcepcionValorNoValido` con el parámetro `queOcurrio` con valores "Ancho" y "Alto" respectivamente cuando alguno de los dos sea inválido. La comprobación se realizará en ese orden.

Dado que la comprobación de ancho y alto la realizaremos también en el siguiente método `leer` puede ser interesante crear métodos `set` para estas propiedades que lancen la excepción indicada en las situaciones erróneas, y así poderlo aprovechar en este constructor y en `leer`.

Circulo getCirculo(int indice) throws ExcepcionValorNoValido Devuelve el círculo colocado en la posición determinada por `indice`, teniendo en cuenta que nuestros índices comienzan desde 1. En caso de que el índice no corresponda a ninguna figura se emitirá una `ExcepcionValorNoValido` indicando el índice (conforme se ha leído del fichero) y el valor "Indice" para el parámetro `queOcurrio`.

void leer(Scanner sc) throws ExcepcionValorNoValido, ExcepcionCoordenadaErronea, ExcepcionOperacionDesconocida, ExcepcionFiguraDesconocida, ExcepcionColorNoValido Siguiendo el formato descrito en la sección 3 de la página 3, la lectura de lienzo comprobará la validez de todos los valores leídos del fichero. En el caso de `titulo`, se debe leer la línea entera que se encuentra tras el ancho y el alto. Para ello usaremos el método `nextLine` de `Scanner`. Además, para evitar que se introduzcan espacios en blanco por delante o detrás de la cadena leída, podemos usar el método `trim` de `String`.

Además de la comprobación de alto y ancho, comprobaremos que la primera palabra de cada línea es "`HACER`", o el nombre de una figura (en esta práctica sólo tenemos "`Circulo`"). En el caso de que no sea así, lanzaremos una excepción `ExcepcionFiguraDesconocida` pasando como cadena esa primera palabra de la línea.

Para leer círculos, es conveniente emplear el método `leer` de `Circulo`. Los círculos leídos se añadirán al vector de círculos del lienzo por detrás.

Cuando leamos una operación, cadena leída tras la palabra "`HACER`", deberemos comprobar que es "`mover`' (en siguientes prácticas habrán más operaciones), lanzando una excepción `ExcepcionOperacionDesconocida` en caso de que no lo sea.

Debemos comprobar también que el índice leído que selecciona la figura es válido. Podemos aprovechar que tenemos un método `getCirculo` que ya realiza esa comprobación.

La comprobación de que la coordenada leída sea válida la realizará el propio `leer` de `Coordenada`.

void exportar(PrintStream ps) La generación del código SVG se realizará imprimiendo en el `PrintStream` la cabecera del XML con los datos del lienzo (ancho, titulo, alto) conforme se describe en la sección 4, para luego exportar cada círculo con su método `exportar`. Se puede separar cada círculo con un fin de línea usando un `println();` del `PrintStream`.

Finalmente, hay que cerrar el XML con un "`</svg>`".

5.4. DibujoSVG

Fíjate que en esta clase el constructor es privado. La única forma de obtener una instancia de la clase es llamar al método *getInstancia()*.

DibujoSVG getInstancia() Devuelve una referencia al único objeto de la clase. Si éste no existe lo crea. Para comprobar en JUnit que estás haciéndolo bien, puedes usar el método `assertSame` que comprueba que dos objetos son realmente el mismo, la misma dirección de memoria.

void setEntrada(string) Asigna valor a *fentrada*, que es el nombre del fichero de entrada.

void setSalida(string) Asigna valor a *fsalida*, que es el nombre del fichero de salida.

void ejecutar() throws FileNotFoundException, ExcepcionValorNoValido, ExcepcionCoordenadaErronea, ExcepcionOperacionDesconocida, ExcepcionFiguraDesconocida, ExcepcionColorNoValido Este método contiene la funcionalidad básica de la aplicación. Es el encargado de abrir los ficheros de entrada y salida, crear los objetos *Scanner* y *PrintStream* respectivamente que se encargarán de leer el fichero de texto de entrada y escribir los de salida, y enviar los mensajes adecuados a un objeto de tipo Lienzo para que lea el contenido del fichero de entrada y escriba la representación SVG del dibujo allí especificado en el fichero de salida.

Es importante asegurarse que se cierra el flujo de salida encapsulado en `PrintStream` mediante el método `close`.

El siguiente código realiza toda esta funcionalidad:

```
File f = new File(this.fentrada);
if (!f.exists()) {
throw new FileNotFoundException(this.fentrada);
}
lienzo = new Lienzo();
Scanner sc  = new Scanner(f);
PrintStream ps = new PrintStream(this.fsalida);
try {
lienzo.leer(sc);
lienzo.exportar(ps);
} finally {
sc.close();
ps.close();
}
```

6. Programa principal

Se deberá utilizar el siguiente programa principal [10], en la clase `mains.Main3`, que utilice los argumentos de entrada representados por `args`.

```
DibujoSVG d = DibujoSVG.getInstancia();
d.setEntrada(args[0]);
d.setSalida(args[1]);
try {
    d.ejecutar();
} catch (FileNotFoundException e) {
    System.err.println(``No se ha encontrado el fichero especificado: ''+ e.toString());
} catch (ExcepcionValorNoValido e) {
    System.err.println(``El valor ''+ e.getValor() + `` para ''+ e.getQueOcurrio() + `` no es correcto'');
} catch (ExcepcionCoordenadaErronea e) {
    System.err.println(``La coordenada (''+ e.getX()  + ``,''+ e.getY() + ``) no es correcta'');
} catch (ExcepcionOperacionDesconocida e) {
    System.err.println(``Operación ''+ e.getOperacion() + `` desconocida'');
} catch (ExcepcionFiguraDesconocida e) {
    System.err.println(``Figura ''+ e.getFigura() + `` desconocida'');
} catch (ExcepcionColorNoValido e) {
    System.err.println(``El color ''+ e.getColor() + `` no es un color SVG'');
}
```

La labor de este *main* es simplemente recoger parámetros, lanzar la ejecución de la aplicación y recoger las posibles excepciones para presentárselas al usuario adecuadamente.

7. Pruebas unitarias

El alumno debe realizar pruebas unitarias personales con JUnit de cada uno de los métodos.

Debemos crear tests unitarios que comprueben que se lanzan las excepciones esperadas. Para ello, podemos usar el siguiente ejemplo como muestra [11]. En este caso, se comprueba que el código dentro del test lanza `ExcepcionCoordenadaErronea` en algún momento, y si no lo hace falla.

```
@Test(expected=ExcepcionCoordenadaErronea.class)
public final void testCoordenadasErronea1() throws ExcepcionCoordenadaErronea {
    Coordenada e1 = new Coordenada(-50000, 10);
}
```

8. Evaluación

Los criterios y métodos de evaluación son los mismos que los aplicados en la práctica anterior.

---

[10] Ten cuidado al copiar y pegar con las comillas tipográficas

[11] Véase `http://junit.sourceforge.net/doc/faq/faq.htm#tests_7`

9. Entrega

Toda la estructura de directorios debe estar comprimida en un fichero llamado `prog3-3-11-12.tgz` que no supere los 500 KB en la que NO se deben incluir los tests unitarios. El plazo de entrega finaliza el día **8 de Noviembre de 2011** a las **23:59h**.