

INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

APRENDIZAJE ACTIVO BASADO EN CASOS



Nivel 4

Mecanismos de Reutilización y Desacoplamiento

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Explicar la importancia de desacoplar las clases que hacen parte de un programa y utilizar interfaces para independizar los contratos funcionales de las implementaciones concretas. Esto con el fin de hacer más flexible y fácil de cambiar el programa que se construya.
- Utilizar las interfaces `Collection`, `List` e `Iterator` de Java que permiten la manipulación abstracta de estructuras contenedoras.
- Explicar la importancia de la herencia como mecanismo de reutilización, con el cual es posible construir nuevas clases a partir de clases ya existentes, las cuales han sido diseñadas con el propósito de facilitar la implementación de una familia de entidades que comparten elementos en común.
- Utilizar la herencia como mecanismo de construcción de aplicaciones en Java y entender el papel que juega la clase `Object` en dicho lenguaje.
- Entender el uso que le hemos dado a la herencia en niveles anteriores, para construir interfaces de usuario y tipos de excepciones.
- Construir interfaces de usuario que incluyan menús de opciones y gráficas simples en 2 dimensiones.

2. Motivación

Es un hecho que los programas de computador que se construyen deben evolucionar, a medida que el problema y el contexto en el que éste ocurre cambian. Piense por ejemplo en el caso de la aerolínea del nivel anterior. ¿Qué pasa si en lugar de un solo tipo de avión debemos manejar varios? ¿Qué tan difícil es generar un nuevo reporte de los vuelos si las autoridades aeroportuarias así lo exigen? Estas modificaciones en el programa corresponden tanto a cambios en los requerimientos funcionales como en el mundo del problema, y difícilmente pueden predecirse en la etapa de análisis. Un programa de buena calidad es aquél que además de resolver el problema actual que se plantea, tiene una estructura que le permite evolucionar a costo razonable. No nos interesa construir un programa que tengamos que rehacer cada vez que algún aspecto del problema cambie o un nuevo requerimiento aparezca.

En este nivel introduciremos el concepto de **interfaz**, como un mecanismo que nos va a permitir construir programas en donde el acoplamiento entre las clases (nivel de dependencia entre ellas) sea bajo, de manera que podamos

cambiar una de ellas con un bajo impacto sobre todas las demás. Esto nos va a facilitar la evolución de los programas, puesto que las clases van a poder disfrutar de un cierto nivel de aislamiento que les va a permitir adaptarse a los cambios sin impactar las demás clases presentes o, por lo menos, con un impacto menor. Una interfaz la podemos ver como un contrato funcional expresado en términos de un conjunto de firmas, el cual representa los compromisos que debe asumir cualquier clase que quiera jugar un papel particular dentro de un programa. Consideremos de nuevo el caso de la aerolínea. ¿Qué pasa si en lugar de la clase `Vuelo`, definimos la lista de métodos que cualquier clase que quiera implementar esa parte del modelo del mundo deba incluir? En ese caso definiríamos la interfaz `IVuelo`, una de cuyas implementaciones posibles sería la clase `Vuelo` que ya hicimos.

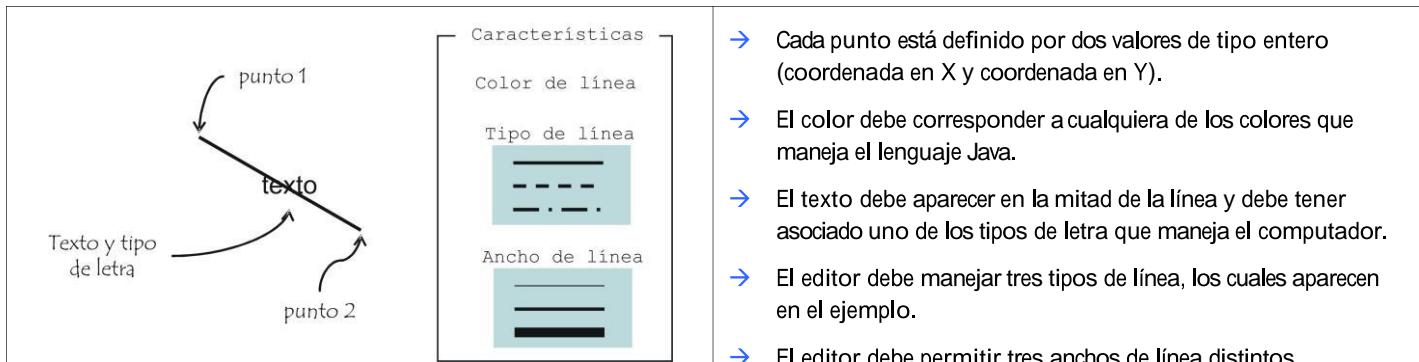
No se debe confundir el término interfaz que utilizaremos en este nivel (un contrato funcional), con la interfaz de usuario de un programa (la parte de la aplicación encargada de la comunicación con el usuario).

Además de hacer evolucionar los programas a bajo costo, nos enfrentamos siempre al reto de hacer las implementaciones de la manera lo más eficiente posible (en tiempo e inversión). Para esto, lo ideal sería poder aprovechar otros desarrollos previos que nos ayuden en la construcción de nuestro programa. Esa es la idea del mecanismo de reutilización de clases denominado **herencia**, tema de este nivel. Imagínese lo elevados que serían los costos si cada vez que nos lanzamos a construir un nuevo programa debemos partir desde cero. ¿Qué tal tener que implementar cada vez las clases contenedoras? ¿Qué tal si en lugar de utilizar la clase `JFrame`, como hemos hecho hasta ahora, tuviéramos que implementarla? Es mucho más simple basarnos en la implementación existente en Java de una ventana y contentarnos con extenderla, precisando en el constructor el contenido que queremos que tenga, pero aprovechando su comportamiento básico y los métodos que ya fueron implementados (recuerde que usamos los métodos `add()`, `setTitle()` y `setLayout()` en la construcción de nuestra ventana como si nosotros los hubiéramos implementado). Para eso utilizamos la palabra `extends` en la declaración de nuestra clase, que es la manera en Java de expresar que una clase hereda (especializa o extiende) de otra clase. Este mecanismo lo hemos usado desde niveles anteriores, pero sólo hasta ahora lo vamos a estudiar a fondo y a ver cómo construir ese tipo de clases.

El último tema que vamos a tratar en este nivel tiene que ver con el manejo de dibujos en dos dimensiones (líneas, círculos, cuadrados, etc.), y con el manejo de menús de opciones. Con estos dos nuevos elementos cubrimos los programas que requieren de este tipo de elementos gráficos para interactuar con el usuario, como editores de planos de edificaciones, editores de dibujos, editores de lenguajes gráficos, etc.

3. Caso de Estudio #1: Un Editor de Dibujos

En este caso queremos construir un editor (que llamaremos `Paint`), que nos permita crear dibujos que contengan líneas, rectángulos y figuras ovaladas. Las líneas deben tener asociados un color, un punto inicial, un punto final, un texto, un tipo de letra (*font*), al igual que un ancho y un tipo de línea (punteada, continua, etc.). Los rectángulos deben estar definidos por dos puntos, por un texto con su tipo, por un color de fondo y por el tipo de línea que debe dibujarse sobre el perímetro. Las figuras ovaladas están definidas por dos puntos, los cuales describen el rectángulo en el cual está incluida la figura, por un texto con su tipo, por un color de fondo y por el tipo de línea del borde del óvalo. En la figura 1 se resumen las características con las que se puede describir cada una de las figuras.



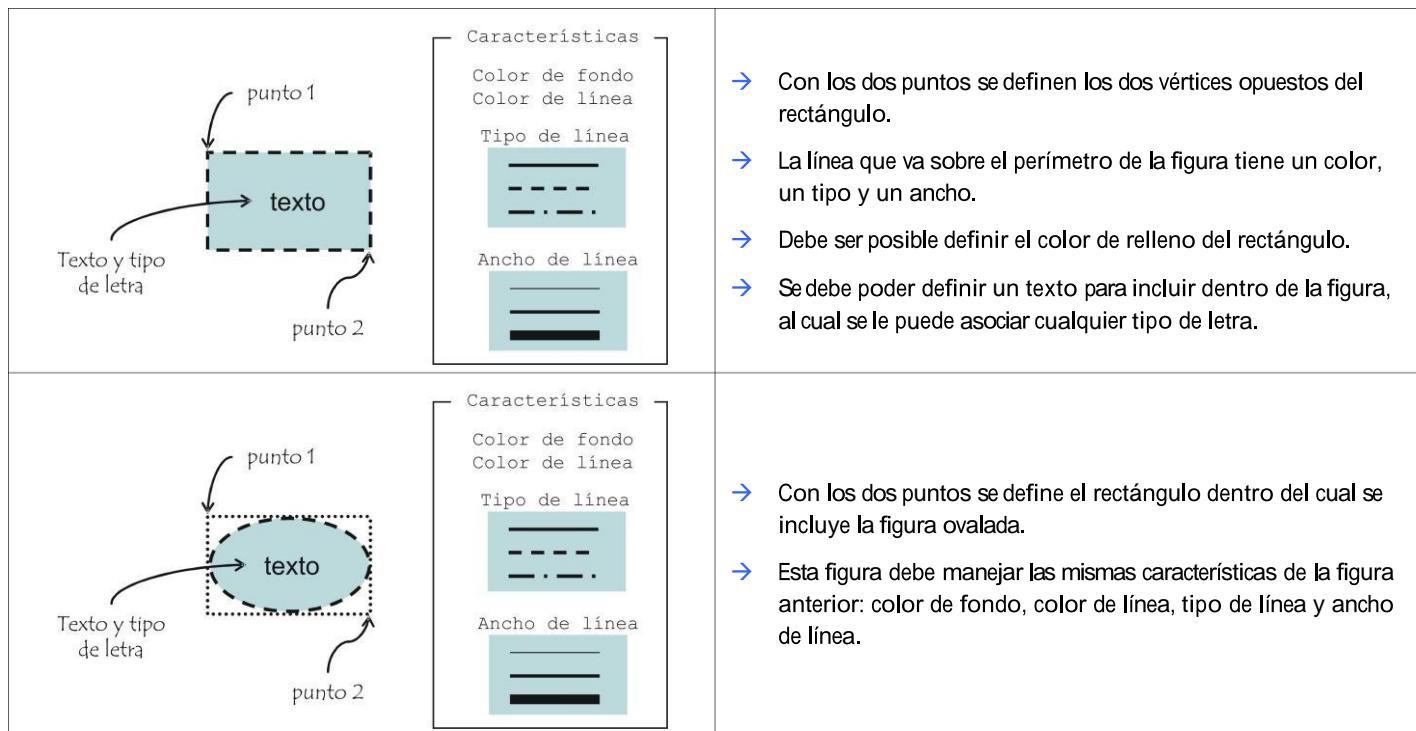
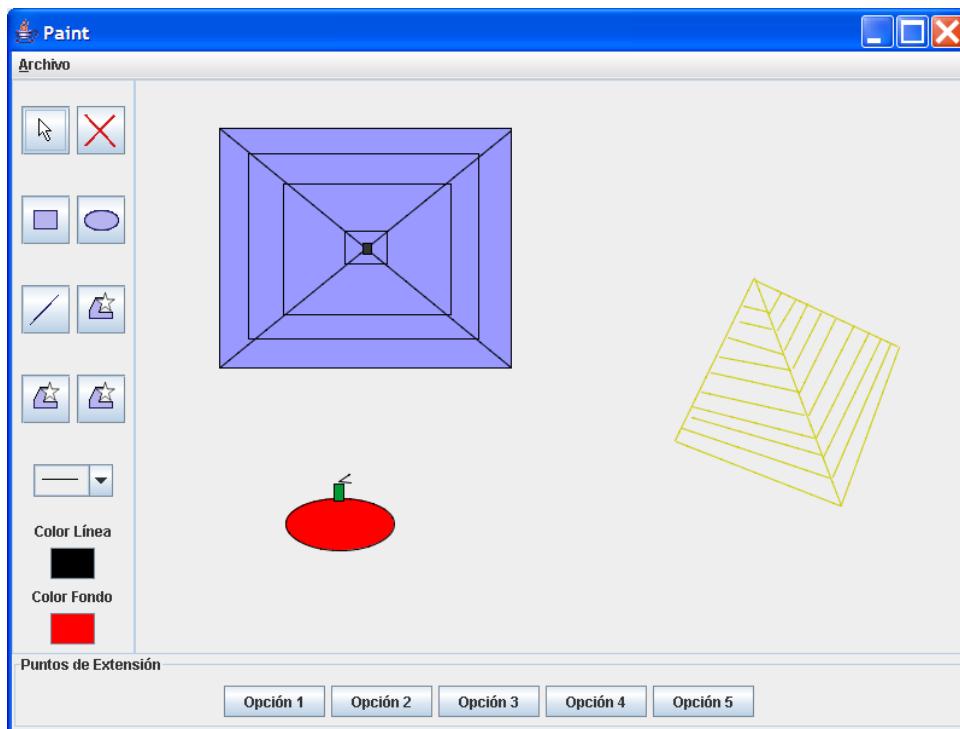


Fig. 1 – Características de las figuras del editor.

Las opciones que debe ofrecer el programa son: (1) agregar una nueva figura al dibujo sin ningún texto asociado, dando la información necesaria para crearla (los puntos, los colores, etc.), (2) seleccionar una de las figuras que hacen parte del dibujo, (3) eliminar la figura seleccionada del dibujo, (4) cambiar el texto asociado con la figura seleccionada, (5) salvar el dibujo en un archivo y (6) cargar un dibujo de un archivo. En la figura 2 aparece la interfaz de usuario que debe tener el programa.



- Arriba a la izquierda aparece el punto de inicio del menú plegable "Archivo" que permite abrir un archivo con un dibujo o salvar el dibujo que en ese momento se esté editando.
- Con los botones de la izquierda, que tienen la imagen de la respectiva figura, se agregan nuevos elementos al dibujo. Para hacerlo se debe seleccionar el respectivo botón, escoger el color de la línea y el color del fondo (si no es una línea). Luego, se deben indicar con el ratón los dos puntos en la zona de edición. Al dar el segundo punto, aparece la figura con las características pedidas.
- Si se siguen dando puntos, se siguen creando figuras del mismo tipo. Para terminar el modo inserción se debe seleccionar el botón con la flecha.

Fig. 2 – Interfaz de usuario del editor de dibujos.

Para seleccionar una figura del editor se debe hacer clic con el ratón sobre ella (sin estar en modo de inserción). Si hay una imagen seleccionada, ésta debe mostrarse claramente al usuario. Para eliminar la figura seleccionada se utiliza el botón marcado con una “X”, el cual aparece en la parte superior izquierda de la ventana.

Es importante que cada vez que el usuario vaya a salir del programa sin haber salvado su trabajo o cuando vaya a cargar un nuevo dibujo sin haber hecho persistir las modificaciones del actual, el editor le pregunte al usuario si desea salvar antes de continuar.

La persistencia se debe hacer en archivos secuenciales de texto, utilizando para esto cualquier formato decidido por el diseñador, pero permitiendo que los archivos se puedan visualizar y modificar desde un editor de texto. Esto puede facilitar en algunos casos la creación de dibujos desde un programa.

La principal restricción del editor es que debe estar construido de manera que sea fácilmente extensible, con nuevas figuras y nuevos requerimientos funcionales. La interfaz de usuario, por ejemplo, ya tiene listos los botones para tres nuevos tipos de imagen.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> • Proponer un diseño que sea fácilmente extensible, el cual permita la incorporación de nuevas figuras y requerimientos funcionales, a bajo costo. 	<ul style="list-style-type: none"> • Estudiar el mecanismo de herencia, como una manera de construir programas que tienen una estructura que permite que los nuevos elementos sean creados por extensión de elementos ya existentes. • Estudiar las interfaces como un medio para obtener diseños en los cuales las clases tienen un bajo nivel de acoplamiento, lo cual facilita la evolución del programa.
<ul style="list-style-type: none"> • Manejar en archivos secuenciales de texto la persistencia de los dibujos. 	<ul style="list-style-type: none"> • Repasar el manejo de archivos secuenciales de texto y la manera de asignar responsabilidades de persistencia entre las clases.
<ul style="list-style-type: none"> • Utilizar menús plegables de opciones, para facilitar la interacción con el usuario. 	<ul style="list-style-type: none"> • Estudiar los componentes <code>JMenuBar</code>, <code>JMenu</code> y <code>JMenuItem</code> del framework gráfico de Java y la manera de incorporarlos al resto de la interfaz de usuario.
<ul style="list-style-type: none"> • Dibujar en la zona de edición del programa figuras geométricas en 2 dimensiones y tomar algunos eventos generados por el ratón sobre dicha zona. 	<ul style="list-style-type: none"> • Estudiar la manera de dibujar figuras en un programa. • Estudiar las clases básicas de manejo geométrico en Java (<code>Line2D</code>, <code>Rectangle2D</code> y <code>Ellipse2D</code>). • Estudiar los eventos generados por el ratón que pueden ser escuchados e interpretados por un programa, los cuales hacen parte de la interfaz <code>MouseListener</code>.

3.2. Comprensión de los Requerimientos

Los seis requerimientos funcionales definidos en el enunciado están dirigidos a un solo actor que vamos a denominar “Usuario”. En la siguiente tarea pedimos al lector que especifique cada uno de dichos requerimientos, haciendo explícitas las entradas que el usuario debe suministrar y el resultado obtenido.

 TAREA #1	<p><u>Objetivo:</u> Entender el problema del caso de estudio del editor de dibujos.</p> <p>(1) Lea detenidamente el enunciado del caso de estudio del editor de dibujos, (2) identifique y complete la documentación de los requerimientos funcionales.</p>			
Haga el diagrama de casos de uso:				
Requerimiento funcional 1	Nombre	R1 – Agregar una nueva figura.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Seleccionar una figura.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Eliminar la figura seleccionada.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			
Requerimiento funcional 4	Nombre	R4 – Cambiar el texto de la figura seleccionada.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			

Requerimiento funcional 5	Nombre	R5 – Salvar el dibujo en un archivo.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			
Requerimiento funcional 6	Nombre	R6 – Cargar un dibujo de un archivo.	Actor	Usuario
	Resumen			
	Entradas			
	Resultado			

Hay tres requerimientos no funcionales que fueron expresados en el enunciado y que deben hacerse explícitos en los documentos producidos en la etapa de análisis. Estos son:

Requerimiento no funcional 1	Tipo: Persistencia
	<p>Descripción:</p> <ul style="list-style-type: none"> • La información del modelo del mundo debe poder ser persistente. • Se deben usar archivos secuenciales de texto para almacenar la información y estos archivos se deben poder visualizar y modificar con cualquier editor de texto.
Requerimiento no funcional 2	Tipo: Extensibilidad
	<p>Descripción:</p> <ul style="list-style-type: none"> • El programa debe estar diseñado de tal manera que su evolución sea sencilla y a bajo costo, permitiendo así integrar nuevos tipos de figuras y nuevos requerimientos funcionales.
Requerimiento no funcional 3	Tipo: Visualización e interacción
	<p>Descripción:</p> <ul style="list-style-type: none"> • Para la creación de las figuras, el usuario debe poder utilizar el ratón directamente sobre la zona de edición, para indicar sus coordenadas. • El usuario debe poder seleccionar una figura del dibujo haciendo clic sobre ella.

3.3. Interfaces: Compromisos Funcionales

Al comenzar a estudiar el modelo del mundo del problema nos encontramos con dos elementos principales, relacionados tal como aparece en la figura 3: la entidad “dibujo” y la entidad “figura”. En el diagrama de clases se puede ver que hay una relación con cardinalidad indefinida, para indicar que un dibujo tiene asociado un grupo de figuras. Por ahora no vamos a entrar en los detalles de cada una de estas dos clases, sino que vamos a tratar de buscar una manera de desacoplarlas para que puedan evolucionar de la manera lo más independiente posible y satisfacer así uno de los requerimientos no funcionales del enunciado.

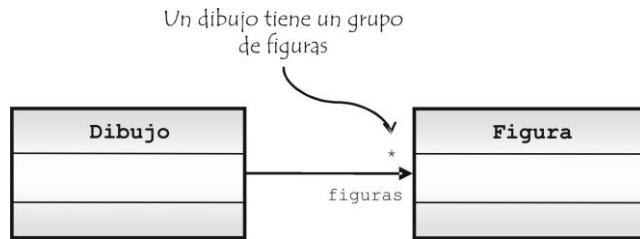


Fig. 3 – Diagrama de análisis del editor de dibujos.

Para empezar debemos reconocer que existe una tercera entidad, que sólo aparece en la etapa de diseño, y que representa la contenedora de figuras. Con el fin de hacerla explícita, transformamos el diagrama de clases del análisis en un primer borrador de diseño, el cual se muestra en la figura 4. Por ahora vamos a llamar **Contenedora** a esa entidad, pero eso cambiará más adelante cuando hayamos terminado de refinar nuestro diseño.

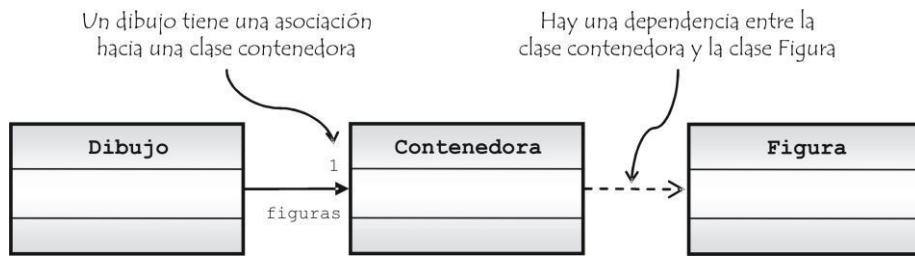


Fig. 4 – Primer borrador de diseño del editor de dibujos.

Vamos a concentrarnos ahora en la entidad **Figura**. El reto es tratar de encontrar una manera de implementar la clase **Dibujo** sin depender directamente de las clases **Línea**, **Rectángulo** y **Ovalo**, que van a representar las figuras que menciona el enunciado. Si logramos hacer esto, va a ser más fácil añadir nuevas figuras más tarde durante la evolución del programa. La pregunta que nos debemos hacer es, ¿existe algo que tengan en común todas las figuras que queremos incluir en el dibujo? La respuesta es simple: todas tienen en común las responsabilidades que deben asumir sus métodos, porque aunque cada figura los puede implementar de una manera distinta, todos deberían tener los mismos. La idea con una interfaz es agrupar bajo un nombre un conjunto de signaturas de métodos para representar los compromisos funcionales de una o varias clases de las cuales no nos queremos preocupar por ahora de su implementación.

Considere la siguiente definición en Java de una interfaz, que representa la responsabilidad funcional de cualquier figura que quiera hacer parte de un dibujo en nuestro editor:

<pre> package uniandes.cupi2.paint.mundo; public interface IFigura { } </pre>	<ul style="list-style-type: none"> → Una interfaz en Java se declara de manera parecida a como se declara una clase, utilizando la palabra “interface”. → Una interfaz se guarda en un archivo con el mismo nombre de la interfaz (IFigura.java), respetando también la estructura de paquetes.
<pre> public void dibujar(Graphics2D g, boolean seleccionada); </pre>	<ul style="list-style-type: none"> → Dibuja la figura sobre la superficie que se recibe como parámetro. Si la figura está seleccionada (lo indica el segundo parámetro), la debe pintar de manera diferente.
<pre> public boolean estaDentro(int x, int y); </pre>	<ul style="list-style-type: none"> → Sirve para saber si un punto está dentro de una figura o no. Cada figura establece qué quiere decir que un punto dado esté en su interior. Sirve para seleccionar una figura.

<code>public String darTexto();</code>	→ Retorna el texto asociado con la figura.
<code>public void cambiarTexto(String txt);</code>	→ Cambia el texto asociado con la figura, por la cadena que se recibe como parámetro.
<code>public Font darTipoLetra();</code>	→ Retorna el tipo de letra actual del texto. La clase Font es una clase de Java que se encuentra en el paquete java.awt.
<code>public void cambiarTipoLetra(Font fuenteTexto);</code>	→ Cambia el tipo de letra actual del texto.
<code>public String darTipoFigura();</code>	→ Retorna la cadena de caracteres que va a identificar este tipo de figuras en la persistencia.
<code>public void guardar(PrintWriter out);</code> }	→ Guarda la figura en un archivo que recibe como parámetro.

Si aceptamos que esos ocho métodos constituyen el contrato funcional que debe respetar cualquier figura de nuestro editor, podemos escribir la clase `Dibujo` completamente en abstracto y más tarde construir las clases que implementan las figuras inicialmente pedidas en el enunciado. Introduciendo esta interfaz en el diseño, obtenemos el diagrama de clases de la figura 5, en donde se ilustra la sintaxis para representar interfaces en UML.

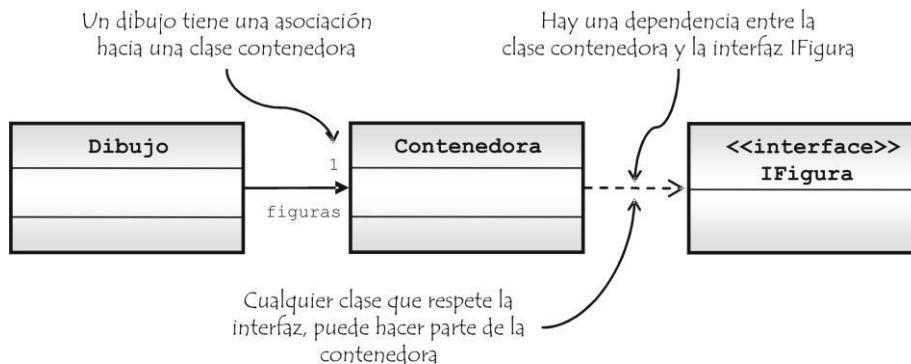


Fig. 5 – Segundo borrador de diseño del editor de dibujos.



Es usual que los nombres de las interfaces comiencen por la letra “I”, en mayúsculas, para así hacer explícito que no se trata de una clase concreta sino de un contrato funcional.



Una interfaz está constituida por un conjunto de signaturas de métodos, a los cuales se les asocia un nombre para representar un contrato funcional. Una interfaz se declara en Java en un archivo aparte que tiene el mismo nombre de la interfaz, respetando la misma jerarquía de paquetes de las clases.

Y, ¿cómo diseñar los métodos que se deben incluir en una interfaz? ¿Cómo saber si hace falta alguno o si los parámetros de los métodos están completos? Ese tema lo abordaremos en un nivel posterior. Por ahora nos concentraremos en la manera de utilizar interfaces diseñadas por alguien más.

3.4. Referencias de Tipo Interfaz

Una vez definida una interfaz podemos declarar atributos, parámetros o variables de ese tipo y manipularlos como si fueran referencias a objetos normales, con la única restricción de que los únicos métodos que podemos invocar sobre ellos, son los contenidos en la interfaz. A los atributos, parámetros y variables de un tipo definido por una interfaz,

únicamente les podemos asignar objetos de una clase que implemente esa misma interfaz. Esto se ilustra en el ejemplo 1.

EJEMPLO #1 	<p><u>Objetivo:</u> Ilustrar el uso de referencias cuyo tipo es una interfaz.</p> <p>En este ejemplo presentamos algunos métodos que podríamos implementar en la clase <code>Dibujo</code>, usando la declaración anterior de la interfaz <code>IFigura</code>. Estos métodos sólo se presentan como ilustración, ya que más adelante mostraremos una mejor implementación de la clase.</p>
<pre>public class Dibujo { private IFigura[] figuras;</pre>	<ul style="list-style-type: none"> → Declaramos un arreglo de objetos cuyo tipo es la interfaz <code>IFigura</code>. En él podremos almacenar objetos de cualquiera de las clases que cumplan el contrato.
<pre>public void inicializacion(IFigura fig) { figura[0] = fig; figura[1] = new Linea(...); figura[2] = new Rectangulo(...); figura[3] = new Ovalo(...); }</pre>	<ul style="list-style-type: none"> → Este método ilustra algunas posibles inicializaciones para el arreglo de figuras. Mientras no tengamos los constructores de las clases, no podemos completar este método (por ahora usamos puntos suspensivos para indicar que no están completas las llamadas). Hay que evitar este tipo de métodos en los cuales hay una dependencia hacia clases concretas, puesto que va a comprometer la evolución del programa. Aquí sólo lo usamos para ilustrar el uso de las interfaces. → Fíjese que las referencias a <code>IFigura</code> se pueden pasar como parámetro y asignar.
<pre>public void dibujar(Graphics2D g) { for(int i = 0; i < figuras.length; i++) { figuras[i].dibujar(g, false); } }</pre>	<ul style="list-style-type: none"> → Este método permite dibujar en la pantalla todas las figuras presentes. La clase <code>Graphics2D</code> representa la zona de dibujo y será estudiada más adelante. → Nos contentamos con pedirle a cada objeto del arreglo que invoque su método de dibujo. Cada uno utilizará su propia implementación para hacerlo. → Fíjese que este método es completamente independiente de las figuras que maneja el editor y no requiere ningún cambio si aparece un nuevo tipo de figura.
<pre>public boolean selecciono(int x, int y) { for(int i = 0; i < figuras.length; i++) { if(figuras[i].estaDentro(x, y)) return true; } return false; }</pre>	<ul style="list-style-type: none"> → Este método recibe como parámetro las coordenadas de la ventana en donde el usuario hizo clic y retorna verdadero si seleccionó una de las figuras presentes, o falso en caso contrario. → Este método le pregunta a cada una de las figuras que contiene el dibujo si las coordenadas del clic están en su interior. → Cada figura utiliza su propia implementación del método <code>estaDentro()</code> cuando recibe la llamada. → Este método también es completamente independiente del conjunto de figuras que soporte el editor.

3.5. Construcción de una Clase que Implementa una Interfaz

Cuando queramos construir una clase que implemente una interfaz, debemos hacer dos cosas. Por un lado, declarar en el encabezado de la clase que vamos a respetar esa interfaz, y, por otro, incluir un método en la clase por cada método de la interfaz, usando la misma signatura. La sintaxis se ilustra en la siguiente tabla. El contenido exacto de esas clases (en el contexto del caso de estudio) es tema de una sección posterior:

<pre>public class Linea implements IFigura { ... }</pre>	<ul style="list-style-type: none"> → Con la palabra “implements” informamos al compilador nuestra intención de respetar la interfaz IFigura. → En la clase Linea debemos implementar los ocho métodos de la interfaz IFigura. Si no lo hacemos, el compilador genera un mensaje de error.
<pre>public class Rectangulo implements IFigura { ... }</pre>	<ul style="list-style-type: none"> → La clase Rectangulo, al igual que todas las clases con figuras del editor, deben cumplir con el contrato funcional definido en la interfaz IFigura.
<pre>public class Ovalo implements IFigura { ... }</pre>	<ul style="list-style-type: none"> → No es suficiente con implementar los métodos de la interfaz, sino que es indispensable que en el encabezado se declare el compromiso de hacerlo.



Si en una clase que dice que va a implementar una interfaz no se incluyen todos los métodos, se obtiene por ejemplo el siguiente error del compilador:

The type Linea must implement the inherited abstract method IFigura.estaDentro(int, int)

En UML las interfaces se muestran con el estereotipo <<interface>> y se usa una asociación especial para indicar que una clase implementa una interfaz dada, tal como se muestra en la figura 6, para el caso del editor de figuras.

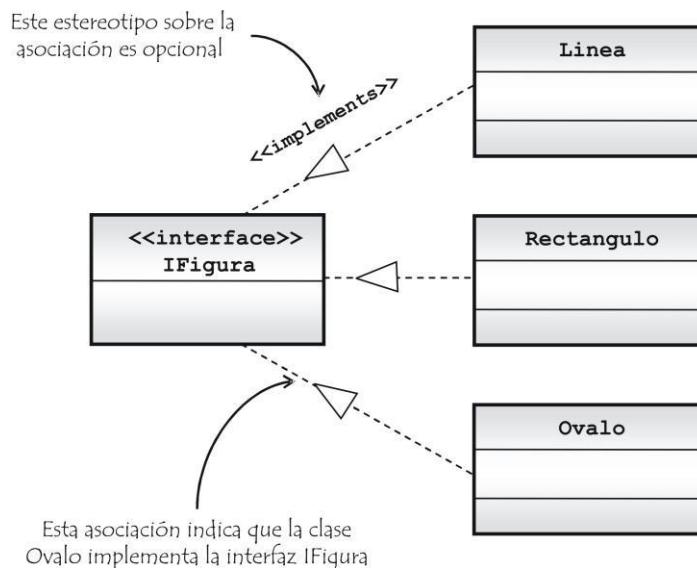


Fig. 6 – Sintaxis para expresar que una clase implementa una interfaz.

Una sintaxis usada con frecuencia para representar la conexión entre dos clases a través de una interfaz es la que se muestra en la figura 7. Allí se hace explícito que la clase C1 tiene una asociación hacia cualquier implementación de la interfaz I1, y que la clase C2 satisface el contrato funcional exigido por dicha interfaz. Esta es la sintaxis usada en los modelos de componentes de software, tema de un curso posterior, los cuales siempre se encuentran aislados por medio de una interfaz.

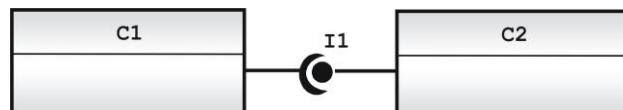


Fig. 7 – Otra sintaxis para mostrar la conexión entre clases pasando por una interfaz.



Con la definición de la interfaz `IFigura` ya logramos un primer nivel de desacoplamiento entre las clases del programa, lo que nos va a permitir más adelante agregar nuevas figuras al editor con modificaciones mínimas.

3.6. Interfaces para Contenedoras

Si avanzamos en la solución del caso de estudio, la siguiente etapa que nos encontramos es el diseño de la estructura contenedora de figuras. Por ahora contamos con tres grupos de opciones que ya hemos estudiado en niveles anteriores: arreglos, vectores o estructuras enlazadas. Cualquiera de esas opciones sería válida, pero, ¿no sería mejor volver a aplicar la misma idea de las interfaces para obtener un nuevo punto de desacoplamiento? ¿Qué nos impide definir en términos de un contrato el funcionamiento de los grupos de objetos, y escribir nuestra clase `Dibujo` usando los métodos de dicha interfaz? Eso tendría la ventaja de que no dependeríamos de una implementación concreta de la contenedora, lo cual facilitaría la evolución del programa.

En Java ya existen dos interfaces llamadas `Collection` y `List`, que permiten a un programador manipular grupos de objetos sin necesidad de restringirse a una implementación concreta. La clase `ArrayList`, que hemos usado hasta este momento para manejar grupos de instancias, implementa estas dos interfaces, tal como se muestra en la figura 8.

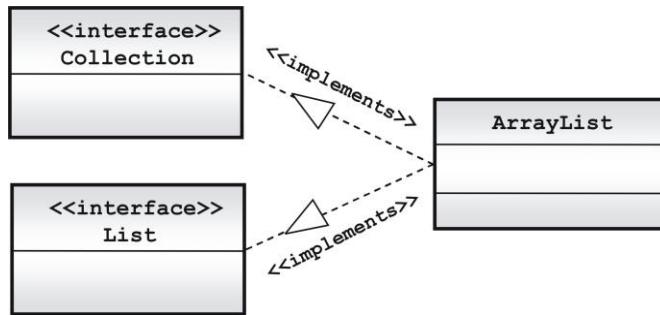


Fig. 8 – Relación entre la clase `ArrayList` y las interfaces `Collection` y `List`.

Lo interesante es que Java dispone de otras clases que satisfacen los mismos contratos funcionales y que pueden remplazar en cualquier momento un `ArrayList`, a condición de escribir los métodos y las declaraciones en términos de las respectivas interfaces. En la figura 9 mostramos otras dos clases de Java (`Vector` y `LinkedList`) que implementan las mismas interfaces.

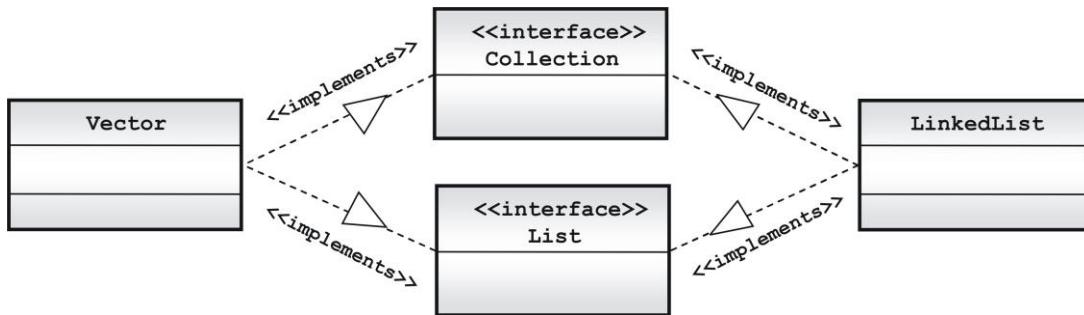


Fig. 9 – Contratos funcionales de las clases `Vector` y `LinkedList`.

Esto quiere decir que si, por ejemplo, existe un método que espera como parámetro una referencia a un objeto que cumpla la interfaz `Collection`, podemos pasarle una instancia de la clase `ArrayList`, una instancia de la clase `Vector` o una instancia de la clase `LinkedList`, y dicho método funcionará correctamente.

Veamos entonces los principales métodos incluidos en estas dos interfaces. En la interfaz `Collection`, se encuentran los métodos que permiten manejar grupos de elementos sin ningún orden o posición específica: básicamente podemos agregar un elemento, eliminarlo y preguntar si el elemento se encuentra en la colección.

<code>public interface Collection</code>	
{	
<code>public boolean add(Object o);</code>	→ Agrega a la colección el objeto que recibe como parámetro. Retorna verdadero si la operación fue exitosa.
<code>public boolean addAll(Collection c);</code>	→ Agrega a la colección todos los objetos de la colección que recibe como parámetro. Retorna verdadero si la operación fue exitosa.
<code>public void clear();</code>	→ Elimina todos los elementos de la colección, dejándola vacía.
<code>public boolean contains(Object o);</code>	→ Retorna verdadero si la colección tiene al menos un elemento “e” que cumple que <code>e.equals(o)</code> es verdadero.
<code>public boolean isEmpty();</code>	→ Retorna verdadero si la colección no tiene ningún elemento.
<code>public Iterator iterator();</code>	→ Retorna un iterador sobre la colección. Esto es tema de la siguiente sección.
<code>public boolean remove(Object o);</code>	→ Elimina la referencia que la colección tiene hacia un elemento “e”, que cumple que <code>e.equals(o)</code> es verdadero. El método retorna verdadero si dicho elemento fue localizado y eliminado.
<code>public int size();</code>	→ Retorna el número de elementos presentes en la colección.
}	

La interfaz `List` contiene los métodos necesarios para manejar secuencias de valores, en los cuales cada elemento tiene una posición. Es más general que la interfaz `Collection` e incluye todos los métodos de esta primera interfaz, concretando en algunos casos su comportamiento. Por ejemplo, para el método `add()` que agrega un elemento a la colección, esta interfaz especifica que lo debe añadir como último elemento de la lista. Veamos los métodos de la interfaz `List` que no son compartidos con la interfaz `Collection`.

<code>public interface List</code>	
{	
<code>public void add(int i, Object o);</code>	→ Inserta el objeto que recibe como parámetro en la posición “i” de la lista, desplazando los elementos ya presentes.
<code>public Object get(int i);</code>	→ Retorna la referencia al objeto que se encuentra en la posición “i” de la lista.
<code>public int indexOf(Object o);</code>	→ Retorna el índice del primer elemento “e” de la lista, que cumple que <code>e.equals(o)</code> es verdadero. Si no encuentra ninguno retorna el valor -1.
<code>public Object remove(int i);</code>	→ Elimina la referencia al objeto que se encuentra en la posición “i” de la lista y retorna la referencia que allí se encontraba.
<code>public Object set(int i, Object o);</code>	→ Reemplaza el elemento que se encontraba en la posición “i” por el objeto que llega como parámetro. Retorna el elemento que antes se encontraba en esa posición.
}	

Regresando a nuestro editor de dibujos, para mejorar el nivel de desacoplamiento vamos a utilizar en el diseño cualquier contenedora que cumpla con el contrato funcional definido por la interfaz `List`. Esto nos lleva al diagrama de clases que aparece en la figura 10. Allí incluimos también tres elementos importantes en el modelado de la entidad `Dibujo`: una asociación hacia la interfaz `IFigura` para señalar la figura seleccionada, un atributo de tipo lógico para indicar si el dibujo ha sido modificado y un atributo de tipo cadena de caracteres para almacenar el nombre del archivo del cual fue leído el dibujo. Ya estamos entonces listos para implementar la clase `Dibujo`, sin haber generado dependencias directas hacia otras clases del diseño.

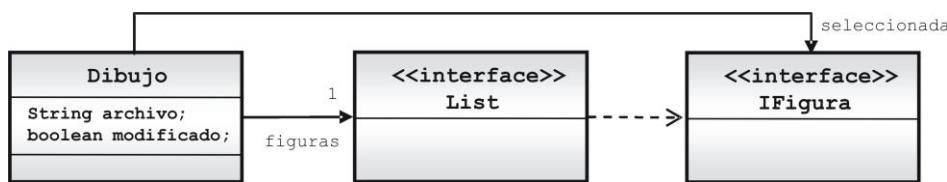


Fig. 10 – Diagrama de clases de diseño, para la clase Dibujo.

En el ejemplo 2 aparecen las declaraciones, el constructor y el invariante de la clase `Dibujo`. La implementación de los demás métodos los presentaremos en la siguiente sección.

EJEMPLO #2

Objetivo: Presentar la declaración de una clase, definida en términos de interfaces.

En este ejemplo mostramos la declaración de la clase `Dibujo`, al igual que el constructor y el método que verifica el invariante.

```

package uniandes.cupi2.paint.mundo;

import java.util.List;

public class Dibujo
{
    // -----
    // Atributos
    // -----

    private List figuras;
    private IFigura seleccionada;
    private String archivo;
    private boolean modificado;

    public Dibujo( )
    {
        figuras = new ArrayList( );
        seleccionada = null;
        archivo = null;
        modificado = false;

        verificarInvarianto( );
    }

    private void verificarInvarianto( )
    {
        assert ( figuras != null ) : "Lista nula";
        if( seleccionada != null )
        {
            for( int i = 0; i < figuras.size( ); i++ )
            {
                IFigura f = ( IFigura )figuras.get( i );
                if( seleccionada == f )
                    return;
            }
            assert ( true ) : "Selección inválida ";
        }
    }
}
  
```

- La interfaz `List` (al igual que la interfaz `Collection`) se encuentran en el paquete `java.util`.
- El primer atributo es una lista con las figuras que hacen parte del dibujo, respetando el orden en el que fueron agregadas en el editor. Todas las figuras implementan la interfaz `IFigura`.
- El segundo atributo contiene la figura que está seleccionada actualmente en el dibujo (si la hay).
- El tercer atributo es el nombre del archivo donde se está guardando actualmente el dibujo.
- El último atributo indica si el dibujo ha sido modificado.
- En el constructor de la clase creamos una instancia de la clase `ArrayList` para manejar la lista de figuras. Si decidimos cambiar de estructura contenedora, basta con modificar una línea de toda la clase.
- Luego, inicializamos los demás atributos de la clase y verificamos que se cumpla el invariante.
- El invariante debe verificar que la lista se encuentre inicializada y que la eventual figura seleccionada haga parte de las figuras del dibujo.
- Para la segunda condición hacemos un ciclo buscando la figura seleccionada y retornando tan pronto la encontramos. Si llega al final del ciclo quiere decir que la figura no hace parte de las que se encuentran incluidas en el dibujo.
- Para el ciclo utilizamos los métodos `size()` y `get()` de la interfaz `List`.



Con este diseño obtenemos un segundo punto de desacoplamiento: la clase `Dibujo` ahora no depende ni de la implementación de la estructura contenedora, ni de las implementaciones de los distintos tipos de figuras que definamos. Eso nos permitirá hacer evolucionar el editor de manera mucho más sencilla, incorporando nuevas figuras y requerimientos funcionales. En caso de necesidad, podríamos incluso hacer nuestra propia implementación de la interfaz `List`, para cumplir, por ejemplo, con algún requerimiento no funcional como persistencia o seguridad.

3.7. Iteradores para Recorrer Secuencias

En algunos casos, y haciendo abstracción de la estructura exacta que estemos manejando, nuestro interés se concentra en hacer un recorrido sobre los elementos contenidos en una estructura. Esto es, pasar una vez sobre cada uno de los objetos presentes. Para hacer esto existe una interfaz en Java, llamada `Iterator`, la cual cuenta con los siguientes métodos:

<code>public interface Iterator</code>	
{ <code>public boolean hasNext();</code>	→ Retorna verdadero si en el iterador quedan todavía elementos por recorrer y falso en caso contrario.
 <code>public Object next();</code>	→ Retorna el siguiente objeto contenido en el iterador.

Tanto la interfaz `Collection` como la interfaz `List` tienen un método que les permite retornar un iterador sobre la estructura contenedora. En los siguientes ejemplos mostramos la manera de utilizar los iteradores para hacer recorridos sobre la lista de figuras de nuestro editor.

EJEMPLO #3 	<p><u>Objetivo:</u> Mostrar la manera de utilizar un iterador para recorrer los elementos de una estructura contenedora.</p> <p>En este ejemplo mostramos un método de la clase <code>Dibujo</code>, cuya solución implica un recorrido por la lista de figuras.</p> <pre>public class Dibujo { private List figuras; private IFigura seleccionada; private String archivo; private boolean modificado;</pre>
<pre>public void dibujar(Graphics2D g) { Iterator iter = figuras.iterator(); while(iter.hasNext()) { IFigura f = (IFigura)iter.next(); f.dibujar(g, f == seleccionada); } }</pre>	<ul style="list-style-type: none"> → En la lista, las figuras se van agregando al final de la estructura a medida que van llegando. Por esta razón se deben dibujar en ese orden, para que las últimas figuras den la impresión de estar sobre las primeras figuras. → El método que dibuja las figuras comienza pidiendo un iterador a la lista de figuras y asignándolo a la variable "iter". → El ciclo se repite mientras haya elementos en el iterador que no hayan sido recorridos. → Con el método <code>next()</code> obtenemos el siguiente objeto del iterador e indicamos que debe implementar la interfaz <code>IFigura</code>. → Finalmente llamamos el método de dibujo de la interfaz <code>IFigura</code>, cuyo primer parámetro es la superficie gráfica de dibujo y cuyo segundo parámetro es un valor lógico que indica si la figura está seleccionada. → Este método no depende de la estructura contenedora que utilicemos, ni de las figuras que se incluyan dentro del editor.

Antes de implementar el método que escribe el dibujo en memoria secundaria (el cual utiliza también un iterador), vamos a definir la estructura, contenido y convenciones que va a seguir este archivo. Sabemos que debe ser un archivo secuencial de texto, que pueda ser editado fácilmente. Aunque todavía no sabemos la información que debe almacenar cada una de las figuras, sabemos que todas ellas deben implementar el método `guardar(PrintWriter)`. También tenemos en esa misma interfaz el método `darTipoFigura()` que retorna una cadena de caracteres con el nombre de la figura. En la figura 11 aparece el esqueleto de la estructura del archivo que vamos a utilizar.

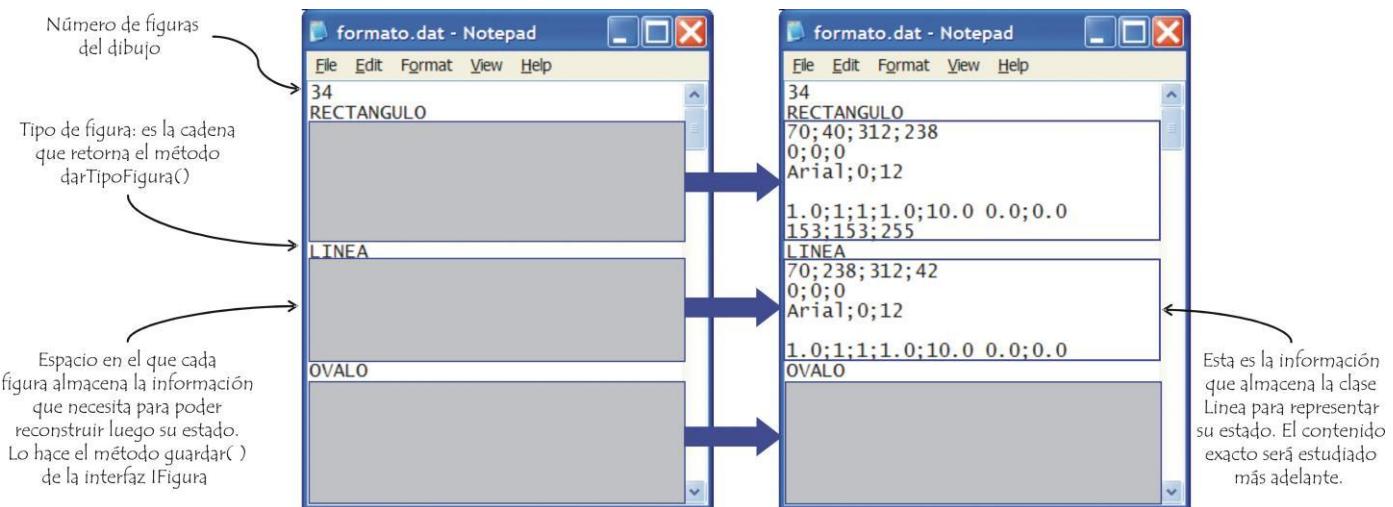


Fig. 11 – Estructura del archivo de persistencia.

En el ejemplo 4 aparece la implementación del método que salva el contenido de un dibujo en un archivo, utilizando la estructura anterior.

EJEMPLO #4 <p><u>Objetivo:</u> Mostrar la manera de utilizar un iterador para recorrer los elementos de una estructura contenedora.</p> <p>En este ejemplo mostramos el método de la clase <code>Dibujo</code> que utiliza un iterador para salvar su estado en un archivo.</p>	<pre>public class Dibujo { private List figuras; private String archivo; private boolean modificado; public void salvar() throws IOException { PrintWriter out = new PrintWriter(archivo); out.println(figuras.size()); Iterator iter = figuras.iterator(); while(iter.hasNext()) { IFigura f = (IFigura)iter.next(); out.println(f.darTipoFigura()); f.guardar(out); } out.close(); modificado = false; } }</pre>	<ul style="list-style-type: none"> → El método lanza la excepción <code>IOException</code> si hay un error en el disco duro en el momento de escribir el archivo. → Lo primero que hace el método es abrir un flujo de escritura, asociado con el archivo en el cual se quiere salvar la información. → Luego, escribe el número de figuras presentes en el dibujo, siguiendo la convención antes definida. → Pide después un iterador sobre la lista de figuras y, a cada una de ellas, le pide su tipo para escribirlo en el archivo y luego le delega la responsabilidad de escribir en el archivo la información que necesite para recuperar después su estado. Fíjese que el flujo pasa como parámetro a cada figura para que escriba allí su información. → Finalmente, el método cierra el flujo de escritura e indica que el dibujo actual no ha sido modificado.
---	---	--

3.8. Implementación del Editor de Dibujos

En esta sección vamos a plantear como tarea la construcción de los distintos métodos que necesita la clase `Dibujo`, dejando pendiente únicamente el método que lee un dibujo de un archivo, puesto que depende de los métodos constructores de las clases que implementan las figuras, los cuales no hemos definido todavía.

 <p>TAREA #2</p>	<p><u>Objetivo:</u> Construir algunos métodos de la clase <code>Dibujo</code>, usando un diseño basado en interfaces.</p> <p>Desarrolle los métodos que se plantean a continuación. Siga las indicaciones que se dan en cada uno de los casos.</p>
<pre>public void agregarFigura(IFigura f) { }</pre>	<ul style="list-style-type: none"> → Agrega una nueva figura al dibujo, al final de la lista de figuras.
<pre>public void reiniciar() {}</pre>	<ul style="list-style-type: none"> → Reinicia el dibujo, eliminando todas las figuras
<pre>public String darNombreArchivo() {}</pre>	<ul style="list-style-type: none"> → Retorna el nombre del archivo donde se está guardando la información de este dibujo.
<pre>public void hacerClick(int x, int y) {}</pre>	<ul style="list-style-type: none"> → Selecciona la última figura agregada al dibujo que se encuentra en el punto (x, y). Si no hay ninguna figura en esas coordenadas, no deja ninguna figura como seleccionada. → Este método debe hacer un recorrido de atrás hacia adelante de la lista de figuras.
<pre>public IFigura darSeleccionada() {}</pre>	<ul style="list-style-type: none"> → Retorna la figura que se encuentra seleccionada o null si no hay ninguna.

<pre>public void eliminarFigura() { }</pre>	<ul style="list-style-type: none"> → Elimina la figura seleccionada. Si no hay ninguna figura seleccionada, este método no hace nada. → Utiliza el método remove(o) de la interfaz List.
<pre>public boolean haSidoModificado() { }</pre>	<ul style="list-style-type: none"> → Indica si el dibujo ha sido modificado.
<pre>public void traerAdelante() { }</pre>	<ul style="list-style-type: none"> → Los siguientes métodos no corresponden a requerimientos funcionales, sino a extensiones del editor. → Deja la figura seleccionada como la última de la lista de figuras, de manera que se vea sobre las demás figuras del dibujo
<pre>public int contarLineas() { }</pre>	<ul style="list-style-type: none"> → Cuenta y retorna el número de líneas que hay en el dibujo. Utilice el método darTipoFigura() de la interfaz IFigura.
<pre>public void seleccionarSiguiente() { }</pre>	<ul style="list-style-type: none"> → Selecciona la siguiente figura del dibujo, de acuerdo con el orden de inserción. Si no hay ninguna figura seleccionada, este método no hace nada.
<pre>public int contarTextos() { }</pre>	<ul style="list-style-type: none"> → Cuenta y retorna el número de figuras del dibujo que tienen un texto asociado.

3.9. Otras Interfaces Usadas Anteriormente

Hay tres interfaces que hemos utilizado anteriormente sin hacerlo explícito. En esta sección, antes de continuar con la solución de nuestro caso de estudio, vamos a hacer un breve recorrido por ellas, para mostrar sus métodos y su uso.

- La interfaz `ActionListener`:

Paquete: <code>java.awt.event</code> Métodos: <code>void actionPerformed(ActionEvent e)</code>	Uso: <ul style="list-style-type: none"> → Los paneles activos de la interfaz de usuario deben implementar esta interfaz, cuando contienen botones u otros componentes gráficos que generan este tipo de eventos. → El método <code>actionPerformed()</code> será invocado cada vez que un evento ocurra, recibiendo como parámetro un objeto de la clase <code>ActionEvent</code> con la información necesaria para que el panel pueda decidir la acción que debe seguir.
--	--

- La interfaz `ListSelectionListener`:

Paquete: <code>javax.swing.event</code> Métodos: <code>void valueChanged(ListSelectionEvent e)</code>	Uso: <ul style="list-style-type: none"> → Los paneles activos de la interfaz de usuario deben implementar esta interfaz cuando tienen un componente gráfico de la clase <code>JList</code> y quieren reaccionar a un cambio en la selección del usuario. → El método <code>valueChanged()</code> será invocado cada vez que cambie el elemento seleccionado por el usuario sobre la lista, recibiendo como parámetro un objeto de la clase <code>ListSelectionEvent</code> con la información necesaria para que el panel pueda reaccionar.
---	--

- La interfaz `Serializable`:

Paquete: <code>java.io</code> Métodos: <code>ninguno</code>	Uso: <ul style="list-style-type: none"> → Esta es una interfaz muy particular de Java, puesto que no exige que se desarrolle ningún método en particular en la clase que la implementa. → La declaración la toma el compilador como una “autorización” para que las instancias de la clase puedan ser serializadas. → Esto se hace por seguridad, puesto que siempre hay un riesgo de manipulación del estado del objeto cuando éste se encuentra en un archivo, o ha sido enviado a otro computador.
---	---

3.10. La Herencia como Mecanismo de Reutilización

Vamos a concentrarnos ahora en la implementación de las distintas figuras que hacen parte del editor y a crear una estructura de base que nos permita crear otras nuevas a bajo costo. Lo primero que debemos resaltar es que todas las figuras, además de tener que implementar la interfaz `IFigura`, tienen una estructura común, la cual hacemos explícita en la figura 12. Allí se puede apreciar que todas las figuras comparten ciertas características, y que nos podemos imaginar una especie de jerarquía o taxonomía de figuras, en donde se clasifican las figuras por los elementos que comparten.

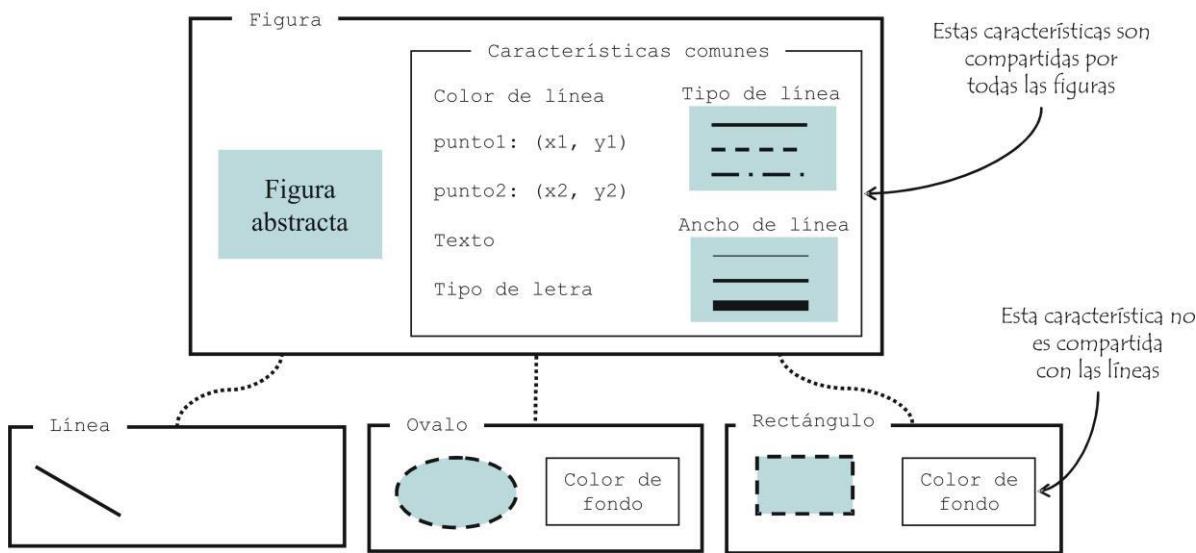


Fig. 12 – Características comunes de las figuras.

Intentemos implementar la clase `Figura` que se sugiere en la jerarquía anterior, de manera que incluya todos los elementos (atributos y métodos) que comparten las figuras del editor. El sólo hecho de implementar (y probar) una sola vez todo lo que tienen en común las tres figuras de nuestro editor ya representa una ganancia para nosotros. Aprovechamos también para identificar los elementos que no comparten, ya sea porque requieren más atributos (como es el caso del color del fondo para los óvalos y los rectángulos), o porque la implementación de los métodos debe ser diferente (el método de dibujar, por ejemplo, no puede ser compartido).

EJEMPLO #5 <p>Objetivo: Implementar una parte de la clase <code>Figura</code>, de manera que contenga los elementos compartidos por las tres figuras de nuestro editor.</p> <p>En este ejemplo mostramos y explicamos las declaraciones de los atributos comunes de las figuras, e implementamos algunos de los métodos que comparten. Hay métodos que no podemos desarrollar hasta haber estudiado las clases de Java que permiten dibujar figuras geométricas, de manera que los veremos en secciones posteriores.</p> <pre> public abstract class Figura implements IFigura { // ----- // Atributos // ----- private int x1; private int y1; private int x2; private int y2; private String texto; private Font tipoLetra; private Color colorLinea; private BasicStroke tipoLinea; } </pre>	<ul style="list-style-type: none"> → En el encabezado debemos indicar que es sólo una implementación parcial y para eso usamos la palabra “<code>abstract</code>”. → También indicamos que nos comprometemos con el contrato funcional descrito por <code>IFigura</code>. → Este grupo de atributos representa los dos puntos que definen la figura: (x_1, y_1) y (x_2, y_2). Debe ser claro que cada una de las figuras los puede interpretar de un modo diferente. → Estos dos atributos representan el texto asociado con la figura y el tipo de letra en el cual éste se debe presentar en el dibujo. La clase <code>Font</code> la provee Java para manejar esto (la estudiaremos más adelante en este nivel). → Este atributo representa el color de la línea. En el caso del rectángulo y el óvalo, es el color de la línea que marca el perímetro. → Este atributo describe el ancho y el tipo de la línea. La clase <code>BasicStroke</code> hace parte de Java y la veremos en una sección posterior.
---	---

```

public Figura( int x1f, int y1f, int x2f, int y2f,
              Color colorLineaF,
              BasicStroke tipoLineaF )
{
    x1 = x1f;
    x2 = x2f;
    y1 = y1f;
    y2 = y2f;

    colorLinea = colorLineaF;
    tipoLinea = tipoLineaF;

    texto = "";
    tipoLinea =
        new Font("Arial",Font.PLAIN, 12);

    verificarInvarianto();
}

```

- Este es el primer constructor de la clase. Permite construir una figura si le dan un valor para cada uno de los atributos, con excepción del texto, que comienza en vacío por defecto y con un tipo de letra estándar ("Arial 12").
- Al finalizar el constructor, verificamos como de costumbre que se cumpla el invariante de la clase.

```

public Figura( BufferedReader br )
              throws IOException,
                     FormatoInvalidoException
{
    ...
}

```

- Este es el segundo constructor de la clase. Permite construir una figura a partir de un flujo de entrada conectado a un archivo. El contenido exacto de este método lo veremos más adelante.

```

public abstract void dibujar( Graphics2D g,
                               boolean seleccionada );

public abstract boolean estaDentro( int x, int y );

public abstract String darTipoFigura();

```

- Estos tres métodos de la interfaz IFigura no se pueden implementar aquí, puesto que no tienen sentido a menos que se concrete la figura que se quiere representar. Por esta razón los métodos se declaran como abstractos y no se les define un cuerpo.
- Es obligatorio poner esta declaración, para asegurarle al compilador que sí vamos a cumplir el contrato funcional descrito por la interfaz, aunque no sea directamente en esta clase.

```

public String darTexto( )
{
    return texto;
}

```

- Retorna el texto asociado con la figura.

```

public void cambiarTexto( String txt )
{
    texto = txt;
}

```

- Cambia el texto asociado con la figura.

```

public Font darTipoLetra( )
{
    return tipoLetra;
}

```

- Retorna el tipo de letra asociado con el texto de la figura.

```

public void cambiarTipoLetra( Font fuenteTexto )
{
    tipoLetra = fuenteTexto;
}

```

- Cambia el tipo de letra asociado con el texto de la figura.

```

private void verificarInvariantes() {
    assert colorLinea != null : "Atributo inválido";
    assert tipoLinea != null : " Atributo inválido";
    assert tipoLetra != null : " Atributo inválido";

    assert x1 >= 0 : "Coordenada x1 inválida";
    assert x2 >= 0 : "Coordenada x2 inválida";
    assert y1 >= 0 : "Coordenada y1 inválida";
    assert y2 >= 0 : "Coordenada y2 inválida";
}
}

```

- Este método verifica que el invariante de la clase se cumpla.
- Lo primero que valida es que el color de la línea, el tipo de línea y el tipo de letra tengan algún valor distinto de nulo (revisa que los respectivos objetos hayan sido creados).
- Despues verifica que las coordenadas de los dos puntos sean valores superiores o iguales a cero, puesto que vamos a trabajar únicamente sobre ese espacio de coordenadas.



Una **clase abstracta** es una implementación parcial, que sólo puede ser utilizada como base para construir otras clases de manera más eficiente. Esto quiere decir que no se pueden crear instancias de una clase abstracta, así tenga definido un método constructor. En Java se utiliza la palabra `abstract` para declarar este tipo de clases.



Un **método abstracto** es un método definido en una clase abstracta, para el cual no existe una implementación. Más adelante, las clases que se construyan con base en esta clase abstracta, serán responsables de implementar estos métodos. En Java estos métodos no tienen cuerpo y se utiliza la palabra `abstract` como parte de su signatura.

3.10.1. Superclases y Subclases

Ahora que ya tenemos una parte de la implementación de la clase `Figura`, ¿hay alguna manera de construir sobre ella las clases que necesitamos para nuestro editor? La respuesta es sí y el mecanismo que lo permite se denomina **herencia**.

La clase sobre la cual nos queremos basar para construir otra clase se denomina la **superclase** y la nueva clase se denomina la **subclase**. En Java se utiliza la sintaxis que se muestra en los siguientes fragmentos de código:

<pre> public class Linea extends Figura { ... } </pre>	<ul style="list-style-type: none"> → Para indicar que la clase <code>Linea</code> hereda (especializa o extiende) de la clase <code>Figura</code>, utilizamos la palabra “<code>extends</code>”. → El compilador localiza el archivo en el que está declarada la superclase y considera todas las declaraciones que hagamos en la subclase como extensiones de la primera.
<pre> public abstract class FiguraRellena extends Figura { // ----- // Atributos // ----- private Color colorFondo; ... } </pre>	<ul style="list-style-type: none"> → Definimos una segunda clase abstracta, que hereda de la clase <code>Figura</code>, con las características comunes que tienen las figuras ovaladas y los rectángulos. → Sigue siendo abstracta, porque hay métodos que no tienen una implementación. → Declaramos como atributo la característica adicional que tiene este tipo de figuras.
<pre> public class Ovalo extends FiguraRellena { ... } </pre>	<ul style="list-style-type: none"> → La clase <code>Ovalo</code> es una subclase de la clase <code>FiguraRellena</code>. Allí sólo implementaremos lo que no esté ya definido en las superclases.
<pre> public class Rectangulo extends FiguraRellena { ... } </pre>	<ul style="list-style-type: none"> → La clase <code>Rectangulo</code> también es una subclase de la clase <code>FiguraRellena</code>.

En el diagrama de clases de UML se utiliza la sintaxis mostrada en la figura 13 para indicar que una clase hereda de otra. Allí se puede apreciar también que se utiliza el estereotipo <>abstract>> para indicar que una clase es abstracta.

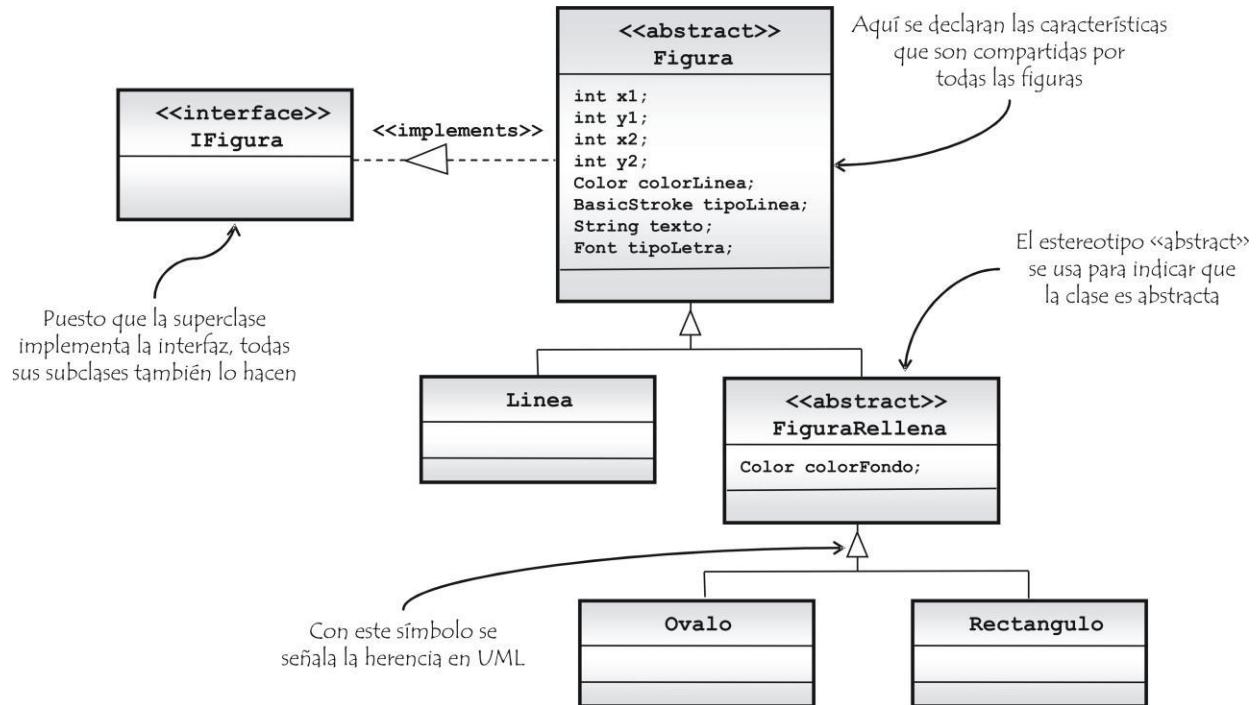


Fig. 13 – Representación de la herencia en el diagrama de clases de UML.

Puesto que las subclases (Linea, Ovalo y Rectangulo) cumplen también los contratos funcionales (**IFigura**) a los que se comprometió la superclase (**Figura**), en nuestro editor podemos manipular las figuras como referencias a esa interfaz sin ningún problema. Con este diseño vamos a poder incluir nuevas figuras en el editor con un esfuerzo mínimo, puesto que cada nueva figura va a **reutilizar** la estructura de base que ya le construimos.



Cuando una clase extiende de otra, hereda todos los atributos y métodos de la superclase. De esta forma, la subclase sólo tiene que incluir los nuevos atributos, los nuevos métodos, una implementación para los métodos abstractos de la superclase y los métodos de la superclase para los cuales quiera cambiar la implementación. Para esto último, basta con utilizar la misma firma del método heredado para que el compilador entienda que la nueva clase quiere modificar la implementación definida en la superclase.

Desde el punto de vista de la clase que utiliza una subclase, no hay ninguna diferencia entre los métodos que ésta hereda y los métodos que implementa. Hacia afuera una subclase ofrece como suyos todos los métodos propios junto con los métodos heredados. Todo lo anterior se ilustra en el ejemplo 6.

EJEMPLO #6



Objetivo: Ilustrar el mecanismo de herencia.

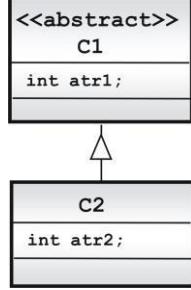
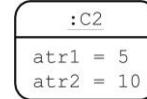
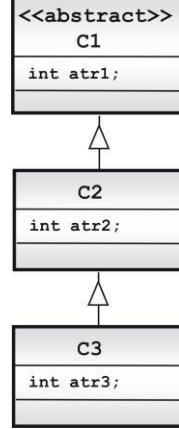
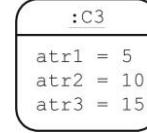
En este ejemplo ilustramos en un contexto simplificado el funcionamiento del mecanismo de herencia. Estudie las clases que se presentan a continuación y el comportamiento que se describe.

```

public abstract class C1
{
    private int atr1;

    public C1()
    {
        atr1 = 5;
    }
}
  
```

- La clase C1 es abstracta. Tiene un único atributo (atr1) y un constructor sin parámetros que inicializa el atributo en el valor 5.
- Por ser una clase abstracta, no es posible crear instancias de ella (a pesar de tener un método constructor implementado).
- Esta clase únicamente puede ser utilizada para construir subclases a partir de ella.

<pre>public abstract int m1(); public void cambiar(int valor) { atr1 = valor; }</pre>	<p>→ Cuenta también con dos métodos: uno abstracto (m1), sin parámetros y otro implementado (cambiar) que permite modificar el valor del atributo.</p>
<pre>public class C2 extends C1 { private int atr2; public C2() { atr2 = 10; } public int m1() { return atr2; } }</pre>	<p>→ La clase C2 es una subclase de la clase C1. No es una clase abstracta, porque implementa los métodos abstractos heredados. Es posible entonces crear instancias de esta clase.</p>  <p>→ El siguiente es el diagrama de objetos de una instancia de C2, tan pronto ha sido creada. Fíjese que el objeto tiene 2 atributos y que para su construcción debió ser llamado primero el constructor de la superclase (de manera implícita) y, luego, el constructor de la subclase:</p>  <p>→ Note que al crearse la instancia, lo que se hace es reunir en un solo objeto los atributos de la superclase (atr1) y los de ella misma (atr2).</p>
<pre>public class C3 extends C2 { private int atr3; public C3() { atr3 = 15; } public void cambiar(int valor) { atr3 = valor * 3; } public String darCadena() { return "valor=" + atr3; } }</pre>	<p>→ La clase C3 hereda de la clase C2, luego tenemos los siguientes diagramas de clases y de objetos:</p>   <p>→ Cada instancia de la clase C3 tiene 3 atributos: dos heredados y uno propio.</p> <p>→ La clase C3 redefine el método cambiar(), puesto que tiene la misma firma del método con el mismo nombre de la clase C1. Eso quiere decir que las instancias de C3 contestarán a las invocaciones de este método de manera distinta a como lo van a hacer las instancias de C1 o de C2.</p> <p>→ El método m1() es heredado de la clase C2. El método darCadena() es añadido por esta clase.</p>

```

public class C4
{
    private C2 obj2;
    private C3 obj3;

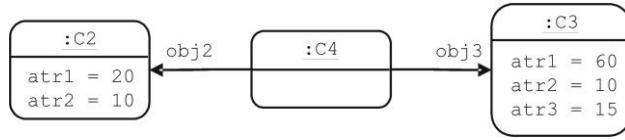
    public C4()
    {
        obj2 = new C2();
        obj3 = new C3();

        obj2.cambiar( 20 );
        obj3.cambiar( 20 );

        String st = obj3.darCadena();
    }
}

```

- La clase C4 tiene un atributo de la clase C3 y un atributo de la clase C2. Puede invocar sobre ambas instancias los métodos m1() y cambiar(), pero con comportamientos distintos en algunos casos. Apenas ha sido creada, una instancia de C4 tendrá los siguientes valores:



- Note que la instancia de C2 utiliza el método cambiar() implementado en la clase C1, mientras la instancia de C3 utiliza su propia implementación de este método.
- Sólo sobre el objeto obj3 se puede invocar el método darCadena(), puesto que fue agregado por la clase C3.



En el momento de redefinir un método, se debe respetar exactamente la misma firma. Si se cambia el tipo de los parámetros, el compilador va a decidir que son dos métodos distintos y no redefine el de la superclase. Si la diferencia es sólo en el tipo de retorno, el compilador presenta un mensaje de error del siguiente estilo:

The return type is incompatible with C1.cambiar(int)

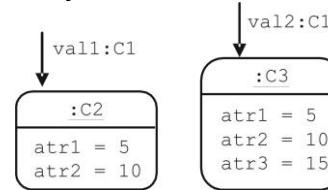
Pasemos ahora a ver cómo se comportan las referencias a objetos, cuando se maneja una jerarquía de herencia. Esto se ilustra con la siguiente secuencia de instrucciones, cuyo significado se explica mediante diagramas de objetos. Vamos a utilizar las clases y la jerarquía de herencia definidas por el ejemplo anterior.

```
C1 val1 = null;
C1 val2 = null;
```

- Es posible definir referencias a una clase aunque ésta sea abstracta. A estas referencias se les va a poder asignar un objeto de cualquiera de las subclases. Esta propiedad se denomina polimorfismo.

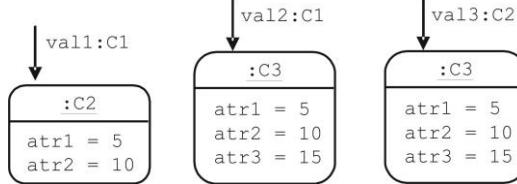
```
val1 = new C2();
val2 = new C3();
```

- Extendemos la sintaxis gráfica utilizada hasta ahora, para hacer explícito el tipo de la referencia que apunta a un objeto:



```
C2 val3 = new C3();
```

- Las variables que hacen referencia a instancias de la clase C2 pueden señalar objetos de la clase C3, puesto que ésta es una de sus subclases:



```

val1.cambiar( 99 );
int aux = val2.m1();

```

- A través de una referencia a una clase, sólo se pueden invocar los métodos que dicha clase tiene, aunque sean abstractos. Puesto que en la clase C1 se encuentran definidos los métodos m1() y cambiar(), es válido hacer estas dos llamadas.
- Durante la ejecución, para determinar el método exacto que debe invocar, Java utiliza la clase del objeto y no el tipo de la referencia a través de la cual se hace el llamado. Por esa razón se puede invocar el método m1() que es abstracto en C1. Esta propiedad se denomina asociación dinámica.

	<ul style="list-style-type: none"> → Para la primera llamada, va a utilizar el método cambiar() de la clase C2. Como allí no está redefinido, utiliza la implementación de la clase C1. → Para la segunda invocación, va a utilizar el método m1() de la clase C3. Como en C3 no está redefinido, utiliza la implementación de este método que encuentra en C2. → Siempre comienza a buscar el método sobre la clase real del objeto y no sobre la clase a la cual pertenece la referencia.
// Inválido String s = val2.darCadena();	<ul style="list-style-type: none"> → Esta llamada es inválida. Aunque val2 esté haciendo referencia a un objeto de la clase C3, que sí tiene el método darCadena() implementado, el compilador restringe las llamadas a los métodos de la clase a la cual pertenece la referencia. Como val2 es una referencia de la clase C1 y el método darCadena() no está allí definido, no se acepta que se haga esta invocación.
val3.cambiar(12);	<ul style="list-style-type: none"> → Esta llamada es válida. val3 es una referencia de la clase C2, pero se encuentra señalando a un objeto de la clase C3. Puesto que la clase C2 tiene ese método definido, la llamada se acepta, pero en ejecución se va a utilizar la implementación que de ese método tiene la clase C3, obteniendo el siguiente estado:
C1 val4 = val3;	<ul style="list-style-type: none"> → Es válido asignar a una variable de una superclase el valor que tiene una variable de una subclase. Al final, ambas referencias quedan apuntando al mismo objeto. En este caso asignamos a la variable val4 (clase C1) la referencia al objeto que está siendo señalado por la variable val3 (clase C2):
C3 val5 = (C3)val3; String s = val5.darCadena();	<ul style="list-style-type: none"> → Para asignar una referencia a una variable de una subclase, es indispensable utilizar el operador de conversión (casting). En nuestro caso, esto es indispensable para poder invocar los métodos de la clase C3 que no están definidos en la clase C2.

Si al momento de aplicar el operador de conversión, el computador se da cuenta de que el tipo del objeto no es compatible con el tipo de la referencia, lanza la siguiente excepción:

java.lang.ClassCastException



Una **subclase** es una versión especializada de una **superclase**, la cual hereda todos los atributos y métodos definidos en la superclase y añade los suyos propios.



El **polimorfismo** es una propiedad de los lenguajes orientados a objetos, que permite a una variable tomar como valor un objeto de cualquiera de sus subclases. Es un mecanismo que facilita la reutilización de código. La **asociación dinámica** es otra propiedad muy importante de algunos lenguajes orientados a objetos (como Java), que garantiza que la decisión de cuál método invocar se toma siempre en ejecución (no durante la compilación), permitiendo así encontrar la implementación más adecuada para cada caso.

3.10.2. Elementos Privados y Elementos Protegidos

Antes de continuar desarrollando nuestro caso de estudio, vamos a detenernos un poco en el tipo de **visibilidad** que deben tener los atributos y métodos de una clase. Hasta ahora hemos definido por defecto que todos los atributos son privados y que todos los métodos son públicos, pero por razones de eficiencia, en algunos casos podría ser conveniente que los métodos de una subclase tuvieran acceso directo a los atributos de la superclase, sin llegar al extremo de dejarlos públicos para todo el mundo. En esta sección presentamos un nuevo tipo de visibilidad llamada protegida (`protected`), en la cual logramos el punto intermedio mencionado anteriormente. Esto se ilustra en el ejemplo 7.

EJEMPLO #7



Objetivo: Mostrar la visibilidad protegida para atributos y métodos de una superclase.

En este ejemplo mostramos, en un contexto simplificado, la manera como se manejan los atributos y métodos con visibilidad protegida.

```
public class C5
{
    private int atr1;
    protected int atr2;

    public C5( )
    {
        atr1 = 0;
        atr2 = 5;
    }

    public int darValor1( )
    {
        return atr1;
    }

    protected void servicio( )
    {
        atr2 = atr1 + 20;
    }
}
```

→ La clase C5 definida en este ejemplo tiene dos atributos. El primero tiene visibilidad privada, de manera que sólo puede ser consultado y modificado por los métodos de esta misma clase. El segundo tiene visibilidad protegida, lo cual indica que puede ser consultado y modificado tanto por los métodos de esta clase, como por los métodos de cualquier de sus subclases. Para las demás clases, este segundo atributo se considera privado.

→ Si cualquier clase, incluidas las subclases, quiere consultar el valor del atributo “atr1”, debe hacer una llamada al método `darValor1()`, ya que este atributo está declarado como privado.

→ Puesto que el método `servicio()` es protegido, únicamente puede ser invocado desde las subclases de la clase C5. Este tipo de métodos se construyen así, de manera que faciliten el desarrollo de los métodos de las subclases, y no porque correspondan a una responsabilidad directa de la clase en la que están.

```
public class C6 extends C5
{
    public int m1( )
    {
        servicio( );
        return atr2 + darValor1( );
    }
}
```

→ La clase C6 es una subclase de C5 que no tiene atributos adicionales.

→ En su único método invoca el método `servicio()` de la superclase, y retorna un valor utilizando directamente el atributo `atr2` de la clase C5 y pasando por el método `darValor1()` para obtener el valor del otro atributo.

→ Aquí no es posible hacer referencia de manera directa al atributo `atr1` de la superclase.



Dependiendo de la visibilidad de un elemento (atributo o método) es posible restringir o permitir el acceso a ellos de la siguiente manera: (1) si es privado, el acceso está restringido a los métodos de la clase, (2) si es protegido, tienen acceso los métodos de la clase y los métodos de las subclases o (3) si es público, todos tienen acceso al elemento.

Teniendo en cuenta lo estudiado anteriormente, vamos a modificar la visibilidad de algunos de los atributos y métodos de las clases `Figura` y `FiguraRellena`, tal como se presenta a continuación. Esto nos va a permitir el acceso directo desde las subclases a algunos atributos y al método que verifica el invariante.

<pre>public abstract class Figura implements IFigura { // ----- // Atributos // ----- protected int x1; protected int y1; protected int x2; protected int y2; protected Color colorLinea; protected BasicStroke tipoLinea; private String texto; private Font tipoLetra; ... protected void verificarInvariante() { ... } protected void dibujarTexto(Graphics2D g) { ... } }</pre>	<ul style="list-style-type: none"> → Cambiamos la visibilidad de todos los atributos geométricos y gráficos de la clase, para facilitar la implementación en las subclases de los métodos de dibujo. → Dejamos privados los atributos que manejan el texto asociado con la figura y aprovechamos los métodos de acceso y modificación que ya habíamos planteado anteriormente. → Cambiamos la visibilidad del método que verifica el invariante, para que pueda ser llamado desde los métodos que verifican el invariante en las subclases. → Agregamos un método de servicio (<code>dibujarTexto</code>), que será utilizado desde las subclases, el cual presenta en la pantalla el texto asociado con la figura en un punto intermedio entre los dos puntos que la definen. Es importante anotar que este método no debe ser público, puesto que no es una de las responsabilidades directas de la clase <code>Figura</code> (no resuelve completamente un requerimiento funcional, sino que puede ayudar a los métodos de las subclases resolviendo una parte del problema que se les plantea). Sólo pretendemos seguir simplificando el desarrollo de nuevas clases, construyendo métodos de apoyo. Esto no implica ninguna obligación de parte de los métodos de las subclases de utilizar estos métodos de servicio.
<pre>public abstract class FiguraRellena extends Figura { // ----- // Atributos // ----- protected Color colorFondo; ... protected void verificarInvariante() { ... } }</pre>	<ul style="list-style-type: none"> → Cambiamos la visibilidad del atributo que almacena el color de fondo de la figura rellena, con el mismo argumento que usamos en la clase <code>Figura</code>. → Dejamos el método que calcula el invariante como protegido (usualmente lo definimos como privado), de manera que pueda ser invocado desde las subclases.



En una clase abstracta vamos a encontrar tres tipos de métodos: (1) los métodos abstractos, para los cuales no es imaginable una implementación, (2) los métodos finales, que ya son la solución que necesitan todas las subclases para asumir una cierta responsabilidad y (3) los métodos de servicio (declarados como protegidos), que son un medio para facilitar el desarrollo de algunas partes de las subclases. En la etapa de diseño se deben identificar los métodos de estos tres grupos.

3.10.3. Acceso a Métodos de la Superclase

Un problema que se nos presenta algunas veces es que desde un método `m1()` de una subclase, no podemos invocar el método `m1()` de la superclase que estamos redefiniendo. Suponga que en la clase `FiguraRellena` queremos redefinir el método que verifica el invariante, el cual ya fue implementado en la superclase `Figura`. Lo queremos redefinir, porque ahora hay que incluir una verificación sobre el nuevo atributo que incluimos. Lo ideal sería poder invocar el método de la superclase que estamos redefiniendo y luego sí verificar la condición adicional. Para poder hacer esto Java nos provee una variable llamada `super`, que siempre hace referencia a la superclase. En el ejemplo 8 mostramos la manera como se usa.

EJEMPLO #8


Objetivo: Mostrar la manera como se utiliza la variable `super`, para tener acceso a los métodos de la superclase.

En este ejemplo avanzamos en el desarrollo de la clase `Línea`, mostrando la manera de implementar los métodos constructores y el método que retorna el tipo de figura para la persistencia.

```
public class Línea extends Figura
{
    // -----
    // Constantes
    //

    public final static String TIPO = "LÍNEA";

    // -----
    // Constructores
    //

    public Línea( int x1f, int y1f, int x2f, int y2f,
                  Color colorLineaF, BasicStroke tipoLineaF )
    {
        super( x1f, y1f, x2f, y2f, colorLineaF, tipoLineaF );
    }

    public Línea( BufferedReader br ) throws IOException,
                                              FormatoInvalidoException
    {
        super( br );
    }

    // -----
    // Métodos
    //

    public String darTipoFigura( )
    {
        return TIPO;
    }
}
```

- La constante que definimos en esta clase la vamos a usar para establecer la cadena de caracteres que va a identificar este tipo de figuras en el momento de persistir en un archivo.
- El método `darTipoFigura()` hace parte del contrato definido por la interfaz `IFigura`. En este caso, retorna la cadena "LÍNEA".
- En la superclase `Figura` tenemos dos constructores. El primero, pide como parámetro un valor para cada uno de los atributos. El segundo, recibe un flujo de entrada conectado a un archivo, por el cual recibe la misma información.
- En esta clase también tendremos dos constructores. Como no hay información adicional que debamos manejar, nos contentamos con invocar el respectivo constructor de la superclase. Para eso utilizamos la variable "super". Como no indicamos un método en particular, se invoca el constructor. En el primer caso, le pasamos la misma información que recibimos como parámetro. En el segundo caso pasamos el flujo de lectura que recibimos.
- El método que verifica el invariante en la clase `Línea` es igual al que implementamos en la clase `Figura`, de manera que no tenemos que implementar nada aquí: sencillamente ese método se hereda.

EJEMPLO #9

Objetivo: Mostrar la manera como se utiliza la variable `super`, para tener acceso a los métodos de la superclase.

En este ejemplo avanzamos en el desarrollo de la clase `FiguraRellena`, mostrando la manera de implementar los métodos constructores, el método que salva la información en un archivo y el método que verifica el invariante.

```

public abstract class FiguraRellena extends Figura
{
    // -----
    // Atributos
    // -----
    protected Color colorFondo;

    // -----
    // Constructores
    // -----

    public FiguraRellena( int x1f, int y1f, int x2f, int y2f,
                          Color colorLineaF, Color colorFondoF,
                          BasicStroke tipoLineaF )
    {
        super( x1f, y1f, x2f, y2f, colorLineaF, tipoLineaF );
        colorFondo = colorFondoF;
    }

    public FiguraRellena( BufferedReader br ) throws
                               IOException, FormatoInvalidoException
    {
        super( br );
        String lineaFondo = br.readLine( );
        String[] strValoresFondo = lineaFondo.split( ";" );
        if( strValoresFondo.length != 3 )
            throw new FormatoInvalidoException( lineaFondo );
        try
        {
            int r2 = Integer.parseInt( strValoresFondo[ 0 ] );
            int g2 = Integer.parseInt( strValoresFondo[ 1 ] );
            int b2 = Integer.parseInt( strValoresFondo[ 2 ] );
            colorFondo = new Color( r2, g2, b2 );
        }
        catch( NumberFormatException nfe )
        {
            throw new FormatoInvalidoException( lineaFondo );
        }
    }

    // -----
    // Métodos
    // -----

    public void guardar( PrintWriter out )
    {
        super.guardar( out );
        out.println( colorFondo.getRed( ) + ";" +
                    colorFondo.getGreen( ) + ";" +
                    colorFondo.getBlue( ) );
    }

    protected void verificarInvariante( )
    {
        super.verificarInvariante( );
        assert colorFondo != null : "Color inválido";
    }
}

```

- Esta clase define un nuevo atributo (`colorFondo`), de manera que nos toca incluir esta nueva información en la verificación del invariante y en la persistencia, al igual que extender los métodos constructores.
- Para el primer constructor recibimos toda la información que necesita la figura, más el color de fondo. Invocamos el constructor de la superclase pasándole la información que recibimos y, luego, almacenamos el color de fondo en el atributo definido con este fin.
- Para el segundo constructor, después de invocar el respectivo constructor de la superclase, leemos del archivo la línea con la información del color que salvamos (conocemos el formato, porque fue el que definimos en el método de guardar). Esa línea la partimos para recuperar los valores de los componentes rojo, verde y azul del color. Con estos valores numéricos reconstruimos el objeto con el color de fondo y lo guardamos en el atributo que declaramos. En caso de cualquier problema, lanzamos una excepción.
- El método que almacena la información en un flujo de salida (`guardar`), invoca por medio de la variable “`super`” el mismo método que fue implementado en la superclase. Luego, agrega al archivo la información del color de fondo, de manera que después pueda recuperarlo. Para esto almacenamos el valor de los componentes rojo, verde y azul del color.
- Finalmente tenemos el método que verifica el invariante de la clase. Debemos como primera medida invocar el respectivo método de la clase `Figura` (el cual está declarado como protegido). Eso lo hacemos utilizando la variable “`super`”. Luego, validamos que el nuevo atributo tenga un valor distinto de `null`.



La variable `super` nos permite tener acceso a los métodos de la superclase, cuando: (1) queremos invocar su método constructor o (2) queremos invocar el método que estamos redefiniendo. En ningún otro caso es necesario usarla.

3.10.4. La Clase Object

En Java existe una clase llamada `Object` de la cual heredan todas las demás clases, así esto no lo hagamos explícito en la declaración con la palabra “`extends`”. Esta clase se puede ver como la raíz de toda la jerarquía de herencia, tal como se sugiere en la figura 14. En esa imagen podemos apreciar también que las clases `Vector` y `ArrayList` comparten buena parte de sus implementaciones mediante el mecanismo de herencia y la construcción de clases abstractas. Una práctica muy común en el diseño orientado a objetos.

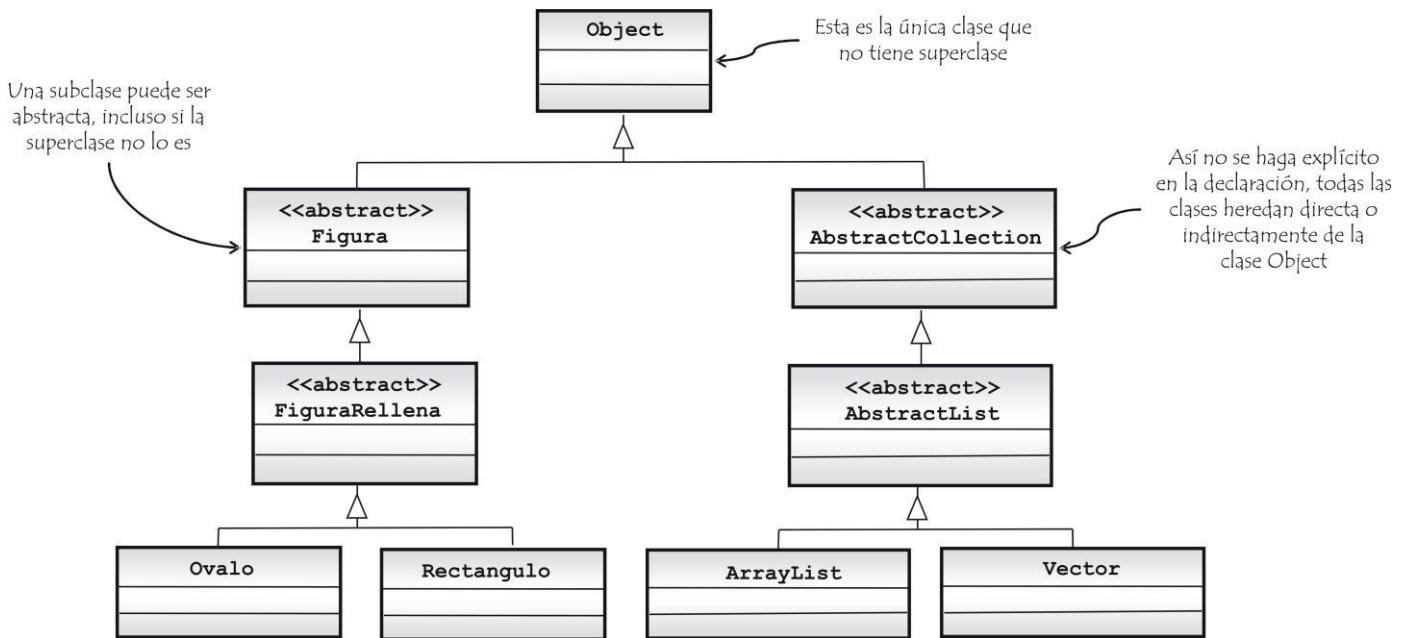


Fig. 14 – Jerarquía de herencia en Java comenzando en la clase `Object`.

La clase `Object` no es abstracta (se pueden crear instancias de esta clase) y tiene tres métodos en los cuales nos vamos a interesar en esta parte, los cuales ya hemos utilizado en niveles anteriores:

- `public boolean equals(Object o)`: Este método recibe como parámetro otro objeto y retorna verdadero si dicho objeto y el objeto que recibe el llamado son el mismo. En nuestras clases, cada vez que queramos modificar el concepto de igualdad entre objetos, debemos redefinir este método. El hecho de que haga parte de la clase `Object` implica que todos los objetos de todas las clases se pueden comparar con este método.
- `public String toString()`: Este método retorna una representación como cadena de caracteres del objeto. Por defecto corresponde a una cadena que contiene el nombre de la clase, seguido del carácter „@“ y luego un valor numérico. Algunos componentes gráficos como el `JList`, invocan este método sobre cada elemento que van a presentar, de manera que si este método no está redefinido, utilizan la implementación por defecto que viene con la clase `Object`. Siempre es válido invocar este método sobre cualquier objeto en Java.
- `protected Object clone()`: Este método protegido permite crear una copia de un objeto a partir de otro. Al igual que con los métodos anteriores, si queremos asociar un significado especial al concepto de clonación, debemos redefinir este método en nuestra clase.

En el ejemplo 10 presentamos la manera de redefinir estos métodos en una clase.

EJEMPLO #10

Objetivo: Mostrar la manera de redefinir los métodos básicos de la clase `Object`.

En este ejemplo creamos una clase simple, en la cual redefinimos los tres métodos de la clase `Object` presentados anteriormente.

```
public class Persona
{
    private String nombre;
    private int codigo;

    public Persona( String nom, int cod )
    {
        nombre = nom;
        codigo = cod;
    }

    public boolean equals( Object o )
    {
        Persona p = ( Persona )o;
        return codigo == p.codigo;
    }

    public String toString()
    {
        return codigo + ": " + nombre;
    }

    public Object clone()
    {
        return new Persona( nombre, codigo );
    }
}
```

```
Persona p1 = new Persona( "Paola", 2929 );

Persona p2 = ( Persona )p1.clone();

System.out.println( p1 );
System.out.println( p2 );

if( p1.equals( p2 ) )
    System.out.println( "IGUALES" );
else
    System.out.println( "DISTINTOS" );
```

- Definimos en este ejemplo la clase `Persona`, que por defecto va a heredar de la clase `Object`. Si quisieramos, podríamos hacerlo explícito en el encabezado, agregando “`extends Object`”.
- La clase tiene dos atributos: el nombre de la persona y un código que suponemos único.
- Para redefinir el método `equals()` debemos respetar la firma exacta del método que se encuentra definido en la clase `Object`. Por esa razón el parámetro que recibimos es de ese tipo. Lo primero que hacemos es la conversión del parámetro a una referencia de la clase `Persona`. Luego, puesto que estamos suponiendo que el código es único, si la persona que llega como parámetro tiene el mismo código, retornamos verdadero.
- Para redefinir el método `toString()` simplemente retornamos una cadena de caracteres con la información que consideramos importante en la clase. Si este objeto llega a un componente gráfico, esta será la cadena de caracteres que se despliegue.
- Para redefinir el método de clonación, retornamos un objeto de la clase `Persona`, con la información de la persona actual. Podemos hacer este retorno sin necesidad de hacer la conversión explícita a la clase `Object`, aprovechando el polimorfismo que existe en Java. También habríamos podido utilizar el método implementado en la clase `Object`, contentándonos con llamar `super.clone()`.

- Suponga que ejecutamos estas instrucciones desde algún método de otra clase. Con los métodos redefinidos, obtenemos la siguiente salida en consola:

```
2929: Paola
2929: Paola
IGUALES
```

- Si las ejecutamos sin redefinir el método `equals()` obtenemos:

```
2929: Paola
2929: Paola
DISTINTOS
```

- Si las ejecutamos sin redefinir el método `toString()` da:

```
uniandes.cupi2.herencia.Persona@10b62c9
uniandes.cupi2.herencia.Persona@82ba41
IGUALES
```

- Si las ejecutamos sin redefinir el método `clone()` no compila, porque la implementación por defecto de este método en la clase `Object` lanza la excepción `CloneNotSupportedException` que no estamos manejando en el código. Si resolvemos ese problema, aparece en ejecución dicha excepción, puesto que la implementación por defecto exige que la clase en la que se use implemente la interfaz `Cloneable`. Si corregimos esto, el código funciona correctamente.

Pero, ¿cuál es realmente la ganancia de tener la clase `Object` en el lenguaje? Hay dos respuestas a esta pregunta. La primera es que sirve para garantizar que todos los objetos tengan un comportamiento común mínimo, que permita construir clases adaptables, que aprovechen el polimorfismo y la asociación dinámica. La segunda respuesta es que son la base para construir las estructuras polimorfas reutilizables (como por ejemplo las clases `ArrayList` y `Vector`). En dichas estructuras almacenamos elementos de la clase `Object`, lo que por el polimorfismo del lenguaje, nos permite guardar allí objetos de cualquier clase que queramos. Eso se ilustra en la figura 15. Allí se puede apreciar que dentro de un `ArrayList` hay en realidad un arreglo de referencias de la clase `Object` y un atributo de tipo entero que indica cuántos elementos están presentes. En cada casilla del arreglo pueden haber una instancia de absolutamente cualquier clase. Esto explica por qué cuando utilizamos el método `get()` para recuperar un objeto de un `ArrayList` debemos utilizar el operador de conversión.

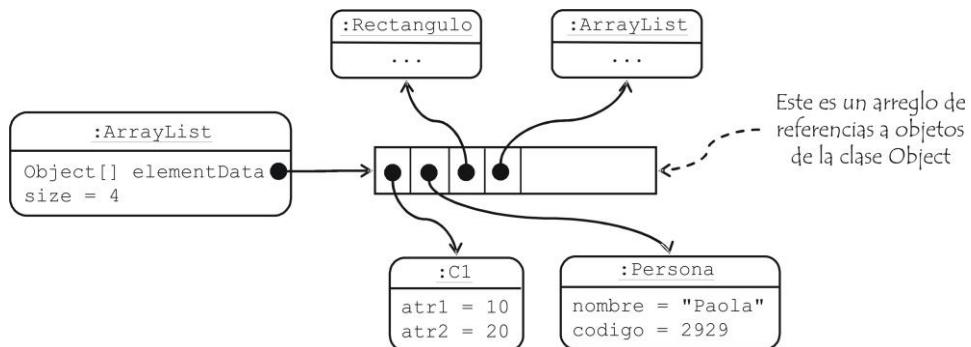


Fig. 15 – Una estructura contenedora polimorfa.

En la siguiente tarea vamos a implementar una contenedora polimorfa simple.



TAREA #3

Objetivo: Implementar una contenedora polimorfa, definiendo una estructura que almacene elementos de la clase `Object`.

Siga las instrucciones que se plantean a continuación:

1. Cree en Eclipse un proyecto llamado `n10_contenedoras`. Vamos a construir las clases en el paquete `uniandes.cupi2.contenedoras`.

2. Declare una interfaz (archivo `IPila.java`) llamada `IPila`, que incluya las siguientes signatures de métodos:

```
public interface IPila
{
    public void agregar( Object elem );
    public Object darPrimero( );
    public void eliminar( );
    public int darLongitud( );
}
```

→ Una pila es una estructura en la cual los elementos se insertan por el tope y se retiran por el mismo punto. El método `agregar()` inserta un nuevo elemento en el tope de la pila. El método `darPrimero()` retorna el elemento que se encuentra en el tope. El método `eliminar()` suprime el elemento del tope de la pila. El método `darLongitud()` retorna el número de elementos en la pila.

3. Construya una clase llamada `PilaArreglo`, que implemente la interfaz `IPila` antes descrita y un método que verifique el invariante de la clase. En el constructor debe definir una capacidad máxima de la pila de 20 elementos. Utilice los atributos que se plantean a continuación:

```
public class PilaArreglo implements IPila
{
    private Object[] elems;
    private int numElems;
    ...
}
```

→ En esta implementación la pila está representada por un arreglo de tipo `Object`, en el cual vamos a almacenar los elementos. El elemento del tope de la pila se encuentra en la posición “`numElems`” del arreglo. Si la pila está vacía, el atributo “`numElems`” tiene el valor -1.

4. Defina los escenarios de prueba y construya la clase `PilaArregloTest` que valida el correcto funcionamiento de la clase `PilaArreglo`. Ejecute las pruebas utilizando JUnit.



En este punto, el lector es capaz de construir una estructura contenedora polimorfa simple, que cumpla un contrato funcional definido por una interfaz. Podría eventualmente construir una contenedora que implementara la interfaz `List` que necesitamos en el editor de dibujos que estamos desarrollando.

3.10.5. Extensión de Interfaces

La herencia también se puede aplicar a las interfaces. En ese caso la herencia se interpreta como la suma de los contratos funcionales definidos por las interfaces de las cuales hereda. En este contexto es preferible utilizar el término “extensión”, puesto que no se aplican exactamente las mismas reglas que se utilizan en el mecanismo de herencia entre clases. En la figura 16 se puede apreciar la relación entre algunas interfaces definidas en Java, en donde no existe una interfaz que sea la raíz de toda la jerarquía.

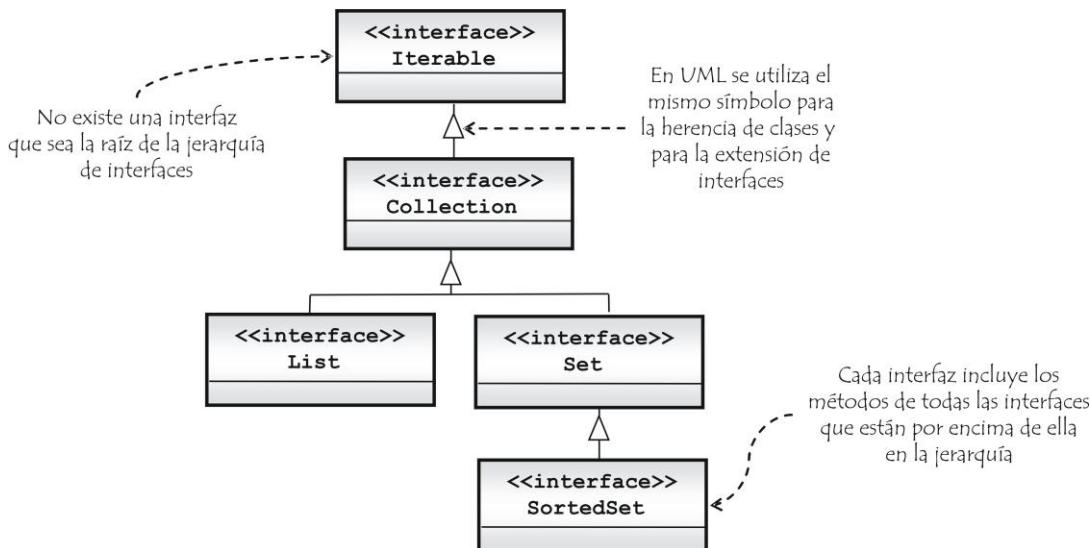


Fig. 16 – Ejemplo de extensión de interfaces en Java.

En el caso de las interfaces la extensión puede ser múltiple, en el sentido de que una interfaz puede extender a la vez de varias interfaces. El mecanismo general de extensión se ilustra en el ejemplo 11.

EJEMPLO #11



Objetivo: Mostrar el mecanismo de extensión de interfaces en Java.

En este ejemplo se presentan algunas interfaces simples que utilizan el mecanismo de extensión para establecer su contrato funcional.

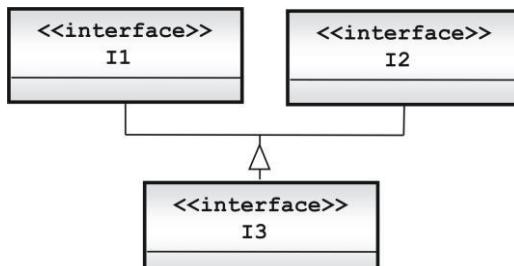
```

public interface I1
{
    public void m1( int a );
}

public interface I2
{
    public void m2( );
}

public interface I3 extends I1, I2
{
    public void m3( int b );
}
  
```

- Las interfaces I1 e I2 incluyen cada una la firma de un método. Cada interfaz se define en su propio archivo (en este caso I1.java e I2.java).
- La interfaz I3 está construida por extensión de las dos interfaces anteriores. Cualquier clase que quiera implementar I3 debe incluir la definición de los tres métodos (m1, m2 y m3).



3.11. Uso del Mecanismo de Herencia en Niveles Anteriores

En los niveles anteriores construimos clases utilizando el mecanismo de herencia sin hacerla explícita, en al menos tres casos: para hacer ventanas (heredando de la clase `JFrame`), para hacer paneles (heredando de la clase `JPanel`) y para definir tipos de excepción (heredando de la clase `Exception`). En el ejemplo 12 volvemos a mirar esas tres situaciones sobre partes del caso de estudio, y aprovechamos para explicar la solución usando la terminología introducida en este nivel

EJEMPLO #12


Objetivo: Mostrar el uso que le hemos dado a la herencia en niveles anteriores y explicar las soluciones utilizando la terminología correcta.

En este ejemplo se presentan algunas clases del caso de estudio, que se deben construir utilizando el mecanismo de herencia.

```
public class FormatoInvalidoException extends Exception
{
    public FormatoInvalidoException( String linea )
    {
        super( "Formato inválido:" + linea );
    }
}
```

- Para crear un nuevo tipo de excepción debemos construir una subclase de la clase `Exception`. En nuestro caso definimos un constructor que recibe como parámetro la línea del archivo donde se encontró el problema. Llamamos al constructor de la superclase (usando la variable “super”), con un mensaje explicando el error.

```
public class InterfazPaint extends JFrame
{
    private Dibujo dibujo;

    public InterfazPaint( )
    {
        dibujo = new Dibujo( );
        ...
        panelBotones = new PanelBotones( this );
        add( panelBotones, BorderLayout.WEST );

        setSize( 800, 600 );
        setTitle( "Paint" );

        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setLocationRelativeTo( null );
        ...
    }
}
```

- Para crear la ventana de la interfaz de usuario debemos construir una subclase de la clase `JFrame`. Dicha clase ya tiene definidos unos atributos y métodos con el comportamiento básicos de las ventanas.
- En el constructor invitamos los métodos heredados (`add`, `setSize`, `setTitle`, etc.) como si hubieran sido desarrollados en nuestra clase.
- Hay varios atributos que agregamos, en particular uno (`dibujo`) para manejar la referencia al modelo del mundo.
- Cuando creamos una instancia de esta clase, va a tener espacio para todos los atributos heredados y para los atributos aquí definidos.

```
public class PanelBotones extends JPanel
                    implements ActionListener
{
    public PanelBotones( InterfazPaint ip )
    {
        ...
        setBorder( new TitledBorder( "" ) );

        ...
        botonColorFondo.addActionListener( this );
    }

    public void actionPerformed( ActionEvent evento )
    {
        ...
    }
}
```

- Para crear un panel dentro de la ventana del programa, debemos construir una subclase de la clase `JPanel`.
- Como es un panel activo (tiene botones a los cuales les debe atender sus eventos), debe implementar la interfaz `ActionListener`.
- Como parte se esta interfaz, implementa el método `actionPerformed()`.
- De nuevo, en esta clase utilizamos todos los métodos heredados, como si fueran propios.

3.12. Los Componentes de Manejo de Menús

Para incluir menús de opciones en un programa, debemos utilizar tres clases que nos provee Java en su paquete `javax.swing`: la clase `JMenuBar` (representa la barra de menú), la clase `JMenu` (representa un menú colgante) y la clase `JMenuItem` (representa una opción de un menú colgante). En la figura 17 mostramos la relación que existe entre el menú que queremos construir para el caso de estudio y las clases antes mencionadas. Allí podemos ver que necesitamos una instancia de la clase `JMenuBar`, la cual tiene asociado un objeto de la clase `JMenu` (que representa el menú colgante llamado “Archivo”) y éste tiene en su interior cinco objetos de la clase `JMenuItem`, cada uno representando una opción del programa.

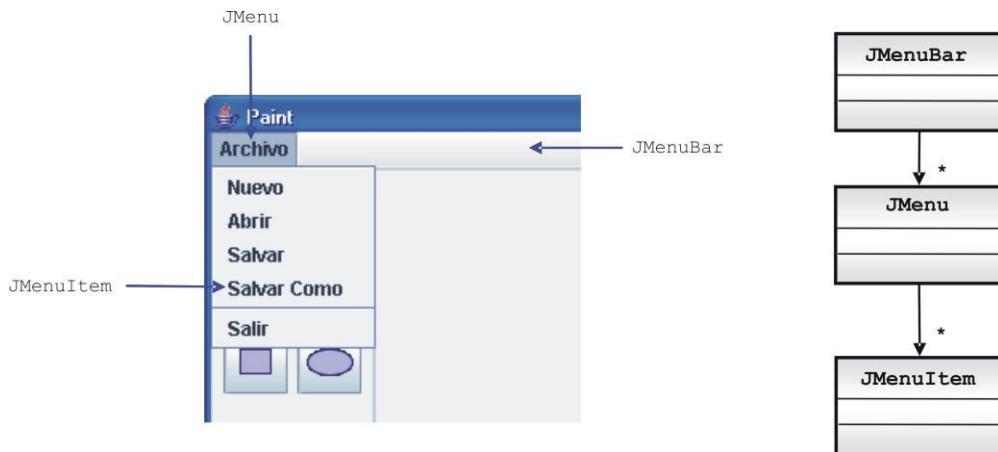


Fig. 17 – Relación entre el menú del programa y las clases de Java que lo implementan.

Los principales métodos de las clases que permiten implementar los menús son los siguientes:

Clase `JMenuBar`:

- `JMenuBar ()`: Este es el método constructor de la clase y permite crear una barra de menú sin ningún elemento.
- `add (menú)`: Con este método se agrega al final de la barra un nuevo menú colgante (instancia de la clase `JMenu`).

Clase `JMenu`:

- `JMenu (título)`: Este es el método constructor de la clase y permite crear un menú colgante cuyo título es el recibido como parámetro.
- `add (opción)`: Con este método se agrega al final del menú colgante una nueva opción (instancia de la clase `JMenuItem`).
- `addSeparator ()`: Este método permite incluir dentro del menú colgante una línea de separación, como aquélla que se muestra en la figura 17 entre la opción “Salvar Como” y “Salir”. Estas líneas de separación son muy útiles para agrupar las opciones del menú por el tipo de funcionalidad que proveen.

Clase `JMenuItem`:

- `JMenuItem (nombre)`: Permite crear una opción con un nombre asociado (por ejemplo, “Salvar Como”).
- `setActionCommand (comando)`: Los eventos de las opciones de los menús van a ser manejados de manera similar a como se manejan los eventos de los botones. Con este método podemos asociar con la opción del menú una cadena de caracteres que va a permitir identificar la orden del usuario en el momento de atender el evento. Esta es la cadena que va a retornar el método `getActionCommand ()` cuando se invoca desde el método `actionPerformed ()`.

- `addActionListener(responsable)`: Con este método definimos cuál es el componente que va a atender el evento que se produce cuando el usuario selecciona esta opción del menú. El parámetro “responsable” debe ser una instancia de una clase que implemente la interfaz `ActionListener`.

Utilizando las clases anteriores, obtenemos el diagrama con el diseño parcial que se muestra en la figura 18. Allí podemos ver que la barra de menús es la encargada de responder a los eventos generados por las opciones de sus menús (implementa la interfaz `ActionListener`) y que lo hace delegando esta responsabilidad en la ventana principal hacia la cual tiene una asociación.

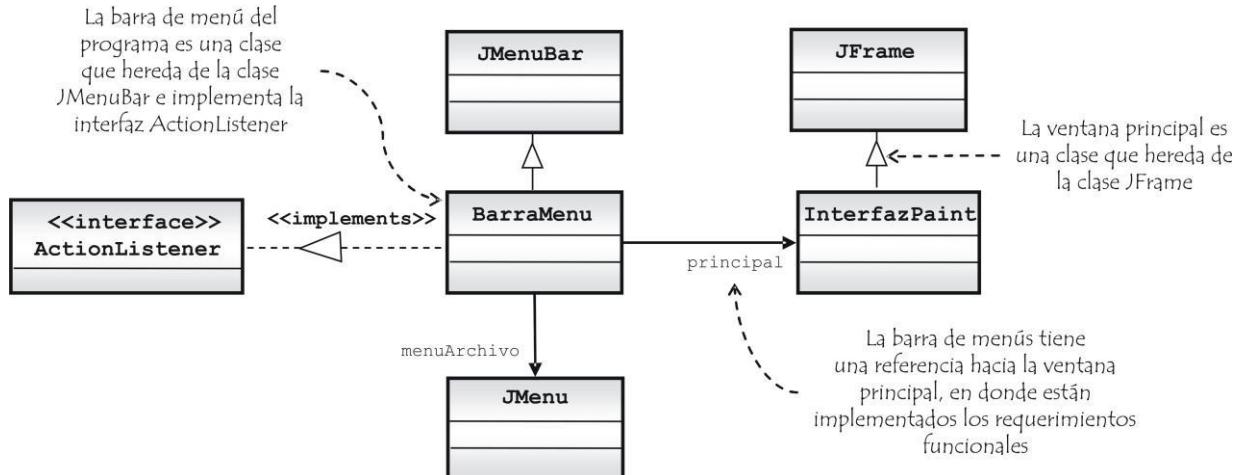


Fig. 18 – Diagrama parcial de clases de la interfaz de usuario para el editor de dibujos.

En el ejemplo 13 mostramos la implementación de la parte de la interfaz de usuario que tiene que ver con el manejo del menú de opciones.

EJEMPLO #13  <p><u>Objetivo:</u> Mostrar la manera de implementar un menú de opciones en un programa.</p> <p>En este ejemplo mostramos la implementación de la clase <code>BarraMenu</code> encargada de implementar el menú de opciones en el editor de dibujos.</p>	<pre> public class BarraMenu extends JMenuBar implements ActionListener { // ----- // Constantes // ----- private static final String NUEVO = "Nuevo"; private static final String ABRIR = "Abrir"; private static final String SALVAR = "Salvar"; private static final String SALVAR_COMO = "SalvarComo"; private static final String SALIR = "Salir"; // ----- // Atributos // ----- private InterfazPaint principal; private JMenu menuArchivo; private JMenuItem itemNuevo; private JMenuItem itemAbrir; private JMenuItem itemSalvar; private JMenuItem itemSalvarComo; private JMenuItem itemSalir; } </pre>	<p>→ Esta clase debe heredar de la clase <code>JMenuBar</code> e implementar la interfaz <code>ActionListener</code>, para poder hacer el manejo de los eventos generados cuando el usuario selecciona una opción.</p> <p>→ Declaramos 5 constantes de tipo cadena de caracteres, para identificar las opciones del menú.</p> <p>→ Como atributos de la clase definimos: una referencia a la ventana principal, una instancia de la clase <code>JMenu</code> y cinco atributos, uno por cada una de las opciones.</p>
--	--	---

```

public BarraMenu( InterfazPaint ip )
{
    principal = ip;

    menuArchivo = new JMenu( "Archivo" );
    add( menuArchivo );

    itemNuevo = new JMenuItem( "Nuevo" );
    itemNuevo.setActionCommand( NUEVO );
    itemNuevo.addActionListener( this );
    menuArchivo.add( itemNuevo );

    itemAbrir = new JMenuItem( "Abrir" );
    itemAbrir.setActionCommand( ABRIR );
    itemAbrir.addActionListener( this );
    menuArchivo.add( itemAbrir );

    itemSalvar = new JMenuItem( "Salvar" );
    itemSalvar.setActionCommand( SALVAR );
    itemSalvar.addActionListener( this );
    menuArchivo.add( itemSalvar );

    itemSalvarComo = new JMenuItem( "Salvar Como" );
    itemSalvarComo.setActionCommand( SALVAR_COMO );
    itemSalvarComo.addActionListener( this );
    menuArchivo.add( itemSalvarComo );

    menuArchivo.addSeparator( );

    itemSalir = new JMenuItem( "Salir" );
    itemSalir.setActionCommand( SALIR );
    itemSalir.addActionListener( this );
    menuArchivo.add( itemSalir );
}

```

```

public void actionPerformed( ActionEvent evento )
{
    String comando = evento.getActionCommand( );

    if( NUEVO.equals( comando ) )
    {
        principal.reiniciar( );
    }
    else if( ABRIR.equals( comando ) )
    {
        principal.abrir( );
    }
    else if( SALVAR.equals( comando ) )
    {
        principal.salvar( );
    }
    else if( SALVAR_COMO.equals( comando ) )
    {
        principal.salvarComo( );
    }
    else if( SALIR.equals( comando ) )
    {
        principal.dispose( );
    }
}

```

- El método constructor recibe como parámetro una referencia a la ventana principal, en donde se encuentran implementados los requerimientos funcionales del programa.
- Lo primero que hacemos es almacenar dicha referencia en el respectivo atributo.
- Luego, creamos el menú llamado “Archivo” y lo agregamos a la barra.
- Despues pasamos a crear cada una de las opciones del menú. Para esto seguimos 4 pasos: (1) crear una instancia de la clase JMenuItem, (2) asignar un nombre al evento que se asocia con el objeto anterior, (3) indicar que el evento será atendido por la clase que estamos construyendo y (4) agregar la nueva opción al menú.
- Utilizamos el método addSeparator() de la clase JMenu para agregar una línea de separación.

- Este método hace parte de la interfaz ActionListener y será invocado cada vez que el usuario seleccione una opción de algún menú de la barra.
- El método recibe como parámetro una instancia de la clase ActionEvent, en la cual se incluye la información del evento generado por el usuario.
- Lo primero que hacemos es tomar el nombre del evento. Aquí recuperamos la cadena de caracteres que asociamos con cada una de las opciones en el constructor.
- Luego, establecemos cuál de las cinco opciones disponibles en el menú fue la que el usuario seleccionó e invocamos el método de la ventana principal que implementa dicho requerimiento funcional.

A continuación planteamos al lector dos tareas con extensiones a la barra de menús del programa.

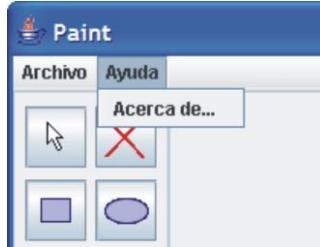


Objetivo: Extender el menú de opciones del editor de dibujos.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Agregue un nuevo menú llamado “Ayuda” con una opción “Acerca de...”, tal como aparece en la siguiente figura:



3. Cree una clase llamada `AcercaDe` que herede de la clase `JDialog` y que presente en la pantalla un diálogo modal con la información del autor del programa y la fecha de creación.

4. Asocie con el evento de la opción “Acerca de...” la presentación del diálogo anterior.

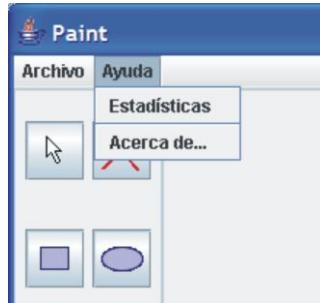


Objetivo: Extender el menú de opciones del editor de dibujos.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Agregue una nueva opción llamada “Estadísticas” al menú de “Ayuda” creado en la tarea anterior:



3. Cree una clase llamada `Estadisticas` que herede de la clase `JDialog` y que presente en la pantalla un diálogo modal con la siguiente información estadística: (1) número total de figuras en el editor, (2) tipo de la figura seleccionada (si hay alguna), (3) número de rectángulos en el dibujo, (4) número de líneas en el dibujo y (5) número de óvalos en el dibujo.

4. Asocie con el evento de la opción “Estadísticas” la presentación del diálogo anterior.

3.13. Manejo de Eventos del Ratón

El siguiente problema al que nos enfrentamos es lograr tomar los eventos generados por el usuario cuando éste hace clic con el ratón sobre la zona de dibujo del editor. El manejo de los eventos en Java siempre se hace utilizando dos conceptos (escuchador y evento) que se integran con la máquina virtual de Java como se muestra en la figura 19. Allí se muestra el caso general del modelo de escucha de eventos y se ilustra con los eventos generados por un botón.

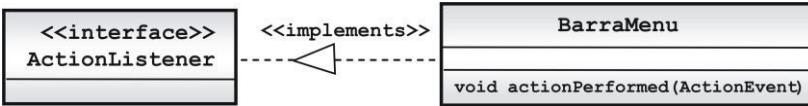
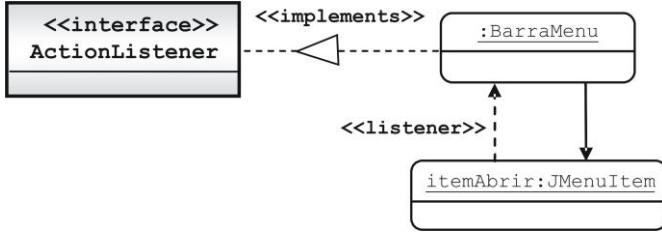
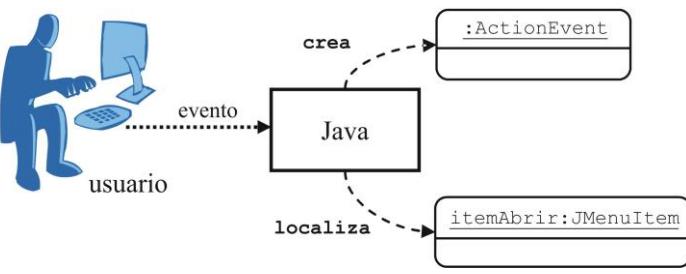
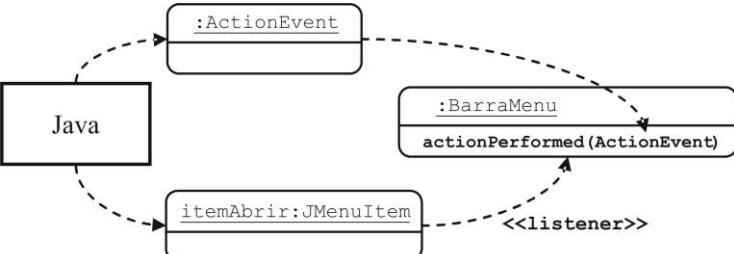
 <pre> classDiagram interface ActionListener { <<interface>> } class BarraMenu { <<implements>> void actionPerformed(ActionEvent) } ActionListener --> BarraMenu </pre>	<ul style="list-style-type: none"> → Una clase implementa la interfaz de un tipo de escuchador (listener). En el caso de los botones, dicha interfaz se llama ActionListener. → Las instancias de dicha clase van a poder escuchar ese tipo de eventos.
 <pre> classDiagram interface ActionListener { <<interface>> } class BarraMenu { <<implements>> void actionPerformed(ActionEvent) } class JMenuItem { <<listener>> BarraMenu } ActionListener --> BarraMenu JMenuItem --> BarraMenu </pre>	<ul style="list-style-type: none"> → Un objeto que quiere generar eventos, indica quién va a ser su escuchador. En el caso de los botones eso se hace con el método addActionListener(). → En la figura se muestra que el escuchador de la opción “Abrir” del menú va a ser el objeto de la clase BarraMenu que lo contiene.
 <p>Diagram illustrating the event creation process:</p> <ul style="list-style-type: none"> A user (usuario) interacts with a computer, generating an event. The event is received by Java. Java creates an ActionEvent object (:ActionEvent). Java localizes the JMenuItem object (itemAbrir: JMenuItem) associated with the event. 	<ul style="list-style-type: none"> → Cuando el usuario realiza alguna acción sobre la interfaz, Java se encarga de crear un objeto que representa el evento, el cual incluye toda la información necesaria para interpretarlo. → El tipo del objeto que se crea depende del tipo de evento producido. En el caso de los botones, se crea una instancia de la clase ActionEvent. → También se identifica el objeto de la interfaz sobre el cual se generó el evento.
 <p>Diagram illustrating Java's role in identifying the listener and calling the actionPerformed method:</p> <ul style="list-style-type: none"> Java identifies the ActionEvent object (:ActionEvent). Java identifies the BarraMenu object (:BarraMenu) which implements the ActionListener interface. Java calls the actionPerformed(ActionEvent) method on the BarraMenu object, passing the ActionEvent object as a parameter. The BarraMenu object has a JMenuItem object (itemAbrir: JMenuItem) registered as a listener, indicated by the <<listener>> association. 	<ul style="list-style-type: none"> → Al objeto sobre el cual se generó el evento (itemAbrir) se le pide que identifique su escuchador (:BarraMenu) y a dicho objeto Java le invoca el método que atiende ese tipo de eventos (actionPerformed), pasándole como parámetro el objeto que representa el evento producido (:ActionEvent).

Fig. 19 – Modelo de escucha de eventos en Java.

Para reaccionar a los eventos generados por el ratón, el escuchador (*listener*) debe implementar la interfaz MouseListener (del paquete `java.awt.event`) y el evento que se produce es una instancia de la clase `MouseEvent` que se encuentra en ese mismo paquete. Los métodos con los que cuentan estos elementos se describen a continuación:

Interfaz MouseListener:

- `mouseClicked(Evento)`: Este método se invoca cuando se ha hecho clic con el ratón sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mousePressed(Evento)`: Este método se invoca cuando un botón del ratón ha sido presionado mientras se encontraba sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mouseReleased(Evento)`: Este método se invoca cuando un botón del ratón ha sido soltado mientras se encontraba sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.

- `mouseEntered(evento)`: Este método se invoca cuando el ratón ha entrado a la zona gráfica de un componente. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mouseExited(evento)`: Este método se invoca cuando el ratón ha salido de la zona gráfica de un componente. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.

Clase `MouseEvent`:

- `getX()`: Retorna un valor entero con la coordenada horizontal del punto en el cual sucedió el evento, con respecto al sistema de referencia del componente.
- `getY()`: Retorna un valor entero con la coordenada vertical del punto en el cual sucedió el evento, con respecto al sistema de referencia del componente.
- `getButton()`: Retorna un valor entero que indica cuál de los botones del ratón (izquierdo o derecho) produjo el evento. Los valores posibles de retorno son las constantes `BUTTON1` o `BUTTON2`.
- `getClickCount()`: Retorna el número de veces que el usuario hizo clic con el ratón como parte del evento. Si este método retorna el valor 2, por ejemplo, se trata de un doble clic.

Un componente gráfico declara que quiere generar eventos del ratón utilizando el método `addMouseListener()` tal como se muestra en el ejemplo 14.

EJEMPLO #14



Objetivo: Mostrar la manera de incluir en un programa el manejo de los eventos del ratón.

En este ejemplo presentamos una parte de la clase `PanelEditor`, la cual se encarga de hacer el manejo de los eventos del ratón generados por el usuario en el editor de dibujos.

```
public class PanelEditor extends JPanel implements MouseListener
{
    public PanelEditor( )
    {
        ...
        addMouseListener( this );
    }

    public void mouseClicked( MouseEvent evento )
    {
        if( evento.getButton( ) == MouseEvent.BUTTON1 )
        {
            ...
        }
    }

    public void mousePressed( MouseEvent evento ) { }

    public void mouseReleased( MouseEvent evento ) { }

    public void mouseEntered( MouseEvent evento ) { }

    public void mouseExited( MouseEvent evento ) { }
}
```

- Esta clase va a ser su propio escuchador de eventos del ratón. Por esta razón implementa la interfaz `MouseListener`.
- Dentro del constructor se declara con el método `addMouseListener()` que es la misma clase quien va a escuchar los eventos del ratón, puesto que pasamos como parámetro la variable “this”.
- Únicamente vamos a escuchar uno de los cinco eventos posibles: el evento de clic del ratón. Por esta razón implementamos el método `mouseClicked()`.
- Dentro de dicho método lo primero que hacemos es verificar que el evento se generó con el botón izquierdo del ratón. Luego vendría el código con la reacción del programa a dicho evento, dependiendo de las coordenadas y del estado en el que se encuentre.
- Debemos implementar los cinco métodos de la interfaz, así sólo vayamos a manejar uno de los eventos.

A continuación proponemos al lector una serie de tareas que implican manejar los distintos eventos que pueden ser generados por el ratón.

TAREA #6 	<p><u>Objetivo:</u> Extender el editor de dibujos, manejando los eventos del ratón. Modifique el programa siguiendo las etapas que se plantean a continuación.</p>
	<ol style="list-style-type: none">1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo <code>n10_paint.zip</code>, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.2. Modifique el programa de manera que si el usuario (en modo de selección) hace clic sobre un punto del editor en donde no hay ninguna figura, presente una ventana de diálogo informándole el error.
TAREA #7 	<p><u>Objetivo:</u> Extender el editor de dibujos, manejando los eventos del ratón. Modifique el programa siguiendo las etapas que se plantean a continuación.</p>
	<ol style="list-style-type: none">1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo <code>n10_paint.zip</code>, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.2. Modifique el panel que contiene los botones de extensión para que haya dos zonas de texto que tengan asociadas las etiquetas “X” y “Y”.3. Modifique el programa para que cada vez que el usuario haga clic en cualquier punto de la zona de dibujo, aparezca en las zonas de texto agregadas en el punto anterior las coordenadas del evento.4. Modifique el programa para que aparezca en la zona de extensión la etiqueta “DENTRO” si el ratón se encuentra dentro de la zona de dibujo, o “FUERA” en caso contrario.
TAREA #8 	<p><u>Objetivo:</u> Extender el editor de dibujos, manejando los eventos del ratón. Modifique el programa siguiendo las etapas que se plantean a continuación.</p>
	<ol style="list-style-type: none">1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo <code>n10_paint.zip</code>, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.2. Modifique el programa para que cada vez que el usuario haga doble clic sobre una figura del editor, se abra una ventana con toda la información que se tiene sobre la figura.3. Modifique el programa para que si el usuario hace clic con el botón derecho del ratón sobre la figura seleccionada, ésta deje de estarlo (no queda ninguna figura seleccionada en el editor).

3.14. Dibujo Básico en Java

En esta sección vamos a estudiar los elementos gráficos necesarios para hacer dibujos simples en 2 dimensiones, incluyendo el manejo de textos.

3.14.1. Modelo Gráfico de Java

Para comenzar, debemos entender tres aspectos básicos del funcionamiento del modelo gráfico de Java:

- Todo componente gráfico (en nuestro caso vamos a dibujar sobre un panel) tiene una superficie de dibujo, la cual nos provee los métodos necesarios para hacerlo. El problema aquí se reduce a obtener dicha superficie a partir del componente en el que queremos dibujar y a invocar los métodos correspondientes. En Java la superficie de dibujo está implementada por las clases `Graphics` y `Graphics2D`, tema de la siguiente sección.
- Una de las responsabilidades de Java, en cuanto tiene que ver con el manejo de los componentes gráficos de la interfaz, es decidir en qué momento debe pintarlos en la pantalla. Si por ejemplo el usuario minimiza la ventana y luego la vuelve a maximizar, o si el usuario coloca una ventana de otro programa por delante y luego la cierra, es Java quien decide que debe volver a pintar sus elementos o parte de ellos. Allí nosotros no participamos de manera directa. Nuestra participación en ese caso es indirecta, puesto que queremos que cuando pida a los componentes gráficos que se muestren en la pantalla nos interesa que invoque nuestros métodos de dibujo. Si no lo hacemos así, todo lo que pongamos en la superficie de dibujo de un panel va a desaparecer la primera vez que el panel se tenga que volver a dibujar. La solución es entonces redefinir el método llamado `paintComponent()` que tienen todos los componentes gráficos, para incluir allí nuestras instrucciones de dibujo, tal como se muestra en el siguiente fragmento de código:

```
public class PanelEditor extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        // Aquí deben ir nuestras instrucciones
        // de dibujo
        ...
    }
}
```

- El método `paintComponent()` está definido en la superclase, de manera que Java lo puede invocar cada vez que se necesite volver a dibujar su contenido en la pantalla.
- Nosotros redefinimos este método en nuestra clase. Para eso invocamos como primera medida el método de la superclase encargado de dibujar el componente y luego agregamos nuestras instrucciones de dibujo. En el caso del editor de dibujos, allí pediremos a cada figura que se dibuje sobre la superficie del panel.

- Si queremos que Java cambie lo que hemos dibujado sobre una superficie (por ejemplo, si el usuario pidió eliminar una figura) es nuestra responsabilidad invocar los métodos necesarios para que esto ocurra. Dentro del modelo gráfico de Java hay un método llamado `repaint()` (implementado por todos los componentes) que se encarga de hacer todo el trabajo por nosotros, el cual invoca en algún momento el método `paintComponent()` que antes redefinimos. Eso quiere decir que si queremos proveer en el panel de dibujo un método de refresco, basta con utilizar el siguiente fragmento de código:

```
public class PanelEditor extends JPanel
{
    public void refrescar( )
    {
        repaint( );
    }
}
```

- El método `repaint()` está implementado en todos los componentes gráficos. Al invocarlo generamos una reacción en la cual el componente se vuelve a dibujar y a invocar su método `paintComponent()`.
- Puesto que ese método lo redefinimos e incluimos nuestras instrucciones de dibujo con el estado actual del editor, obtenemos como resultado que se actualiza nuestra visualización.



Para dibujar en un panel debemos redefinir su método `paintComponent()` e incluir allí las instrucciones que pintan sobre su superficie. La superficie de dibujo está implementada con las clases `Graphics` y `Graphics2D`. El método `repaint()` obliga al componente a recalcular su visualización.

3.14.2. Manejo de la Superficie de Dibujo

Todos los componentes gráficos en el *framework swing* de Java tienen una superficie de dibujo, que utiliza el sistema de coordenadas en píxeles que se muestra en la figura 20.

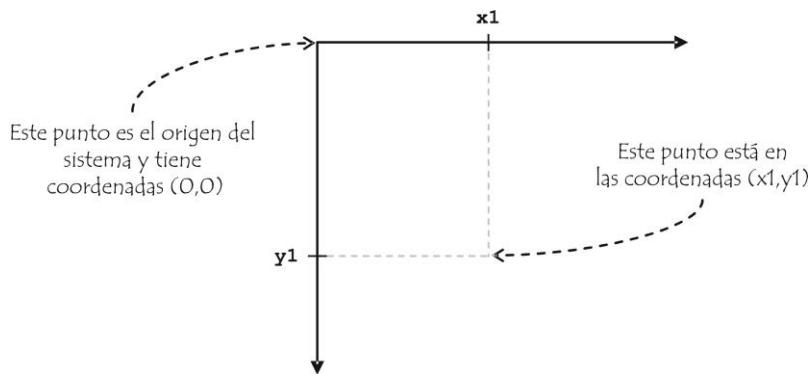


Fig. 20 – Sistema de coordenadas en una superficie de dibujo.

La funcionalidad de dicha superficie se encuentra implementada por la clase `Graphics2D`, la cual hereda de la clase `Graphics`, que es una clase abstracta con algunos métodos básicos. La superficie de dibujo de un componente se recibe como parámetro del método `paintComponent()` mencionado en la sección anterior, de manera que para iniciar el proceso de dibujo únicamente debemos tomar dicho parámetro y convertirlo en un elemento de la subclase `Graphics2D`, tal como se muestra en el siguiente fragmento de código:

```
public class PanelEditor extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        Graphics2D g2 = ( Graphics2D )g;
        // Aquí deben ir nuestras instrucciones
        // de dibujo sobre la superficie g2
        ...
    }
}
```

- El primer paso para dibujar es tomar el objeto que representa la superficie de dibujo del componente gráfico, el cual llega como un parámetro de la clase `Graphics`.
- Lo convertimos de una vez a una referencia de la clase `Graphics2D`, puesto que sabemos que el objeto que llega como parámetro pertenece a dicha subclase.

Comencemos estudiando dos métodos básicos de dibujo que hacen parte de la clase `Graphics2D`, de los cuales mostramos su uso en el ejemplo 15.

- `setColor(color)`: Este método permite definir el color con el que todas las operaciones de dibujo que siguen van a trabajar. Este valor se mantiene hasta que este método sea invocado nuevamente con un valor distinto. Recibe como parámetro un objeto de la clase `Color`.
- `drawLine(x1, y1, x2, y2)`: Con este método dibujamos una línea en la superficie de dibujo (con el color definido por el método anterior) entre los puntos (x_1, y_1) y (x_2, y_2) .

EJEMPLO #15



Objetivo: Mostrar la manera de utilizar los métodos básicos de dibujo.

En este ejemplo mostramos la implementación de un panel con una malla dibujada sobre él, como el que aparece a continuación. Si el tamaño de la ventana cambia, el panel debe completar el dibujo de la malla.



```

public class PanelMalla extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );

        Graphics2D g2 = ( Graphics2D )g;
        g2.setColor( Color.GRAY );

        int alto = getHeight();
        int ancho = getWidth();
        for( int i = 0; i < ancho; i +=25 )
        {
            g2.drawLine( i, 0, i, alto );
        }

        for( int j = 0; j < alto; j +=25 )
        {
            g2.drawLine( 0, j, ancho, j );
        }
    }
}

```

- Redefinimos el método de dibujo del componente y convertimos como primera medida el parámetro en una referencia de la clase Graphics2D.
- Con el método setColor() definimos que todo lo que dibujemos será de color gris (constante Color.GRAY).
- Con los métodos getHeight() y getWidth() de la clase JPanel establecemos el tamaño del mismo.
- Dibujamos líneas verticales cada 25 píxeles desde 0 hasta el ancho del panel. Cada línea va desde la posición 0 hasta el alto del panel.
- Luego dibujamos las líneas horizontales siguiendo un esquema equivalente al anterior.

Para dibujar cadenas de caracteres contamos con los siguientes métodos, para los cuales se ilustra su uso en el ejemplo 16.

- `setFont(tipo)`: Este método recibe como parámetro un objeto de la clase `Font`, el cual define el tipo de letra que debe usarse en todas las operaciones de dibujo de cadenas de caracteres que se hagan a continuación. Este valor se mantiene hasta que este método sea invocado nuevamente con un valor distinto. Para crear un objeto de la clase `Font` se deben definir tres características: el nombre (por ejemplo “Arial”, “Tahoma” o “Courier New”), el estilo (por ejemplo `Font.BOLD`, `Font.PLAIN` o `Font.ITALIC`) y el tamaño (por ejemplo 10, 12 ó 24).
- `drawString(cadena, x, y)`: Este método permite escribir una cadena de caracteres sobre la superficie de dibujo, comenzando en el punto `(x, y)` definido en los parámetros. Utiliza para la operación el tipo de letra definido con el método anterior.

EJEMPLO #16



Objetivo: Mostrar la manera de utilizar los métodos de dibujo de cadenas de caracteres.

En este ejemplo mostramos una parte de la implementación de un panel en el que se dibuja la cadena “Hola” con distintos tipos de letra.



```

public class PanelTipoLetra extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        Graphics2D g2 = ( Graphics2D )g;

        g2.setFont( new Font( "Arial", Font.BOLD, 10 ) );
        g2.drawString( "Hola", 10, 10 );
        g2.setFont( new Font( "Courier New", Font.ITALIC, 18 ) );
        g2.drawString( "Hola", 75, 75 );
    }
}

```

- Antes de escribir la cadena definimos el tipo de letra que queremos utilizar utilizando para esto el método `setFont()`.
- Con el método `drawString()` escribimos la cadena que pasamos como parámetro, comenzando en las coordenadas dadas.

3.14.3. Manejo de Formas Geométricas

Además de las líneas y los textos, es posible dibujar distintos tipos de figuras geométricas, para las cuales existen clases especiales en Java, todas ellas implementando la interfaz `Shape`. Estas figuras se dibujan sobre un objeto de la clase `Graphics2D` utilizando los siguientes métodos:

- `draw(figura)`: Este método recibe como parámetro un objeto de una clase que implementa la interfaz `Shape` y dibuja su borde en la superficie.
- `fill(figura)`: Este método recibe como parámetro un objeto de una clase que implementa la interfaz `Shape` y dibuja la figura rellena en la superficie.

La interfaz `Shape` se encuentra en el paquete `java.awt` y tiene dos métodos que nos interesan:

- `contains(x, y)`: Indica si el punto (x, y) se encuentra en el interior de la figura geométrica.
- `getPathIterator()`: Este método retorna la información geométrica necesaria para dibujar la figura. Su contenido exacto no nos interesa en este momento, puesto que es la superficie la que debe ser capaz de interpretar dicha información para poder hacer el dibujo.

Veamos ahora las clases que implementan las figuras geométricas que necesitamos para nuestro caso de estudio, las cuales se encuentran en el paquete `java.awt.geom`. En la figura 21 aparece una parte de la jerarquía de clases.

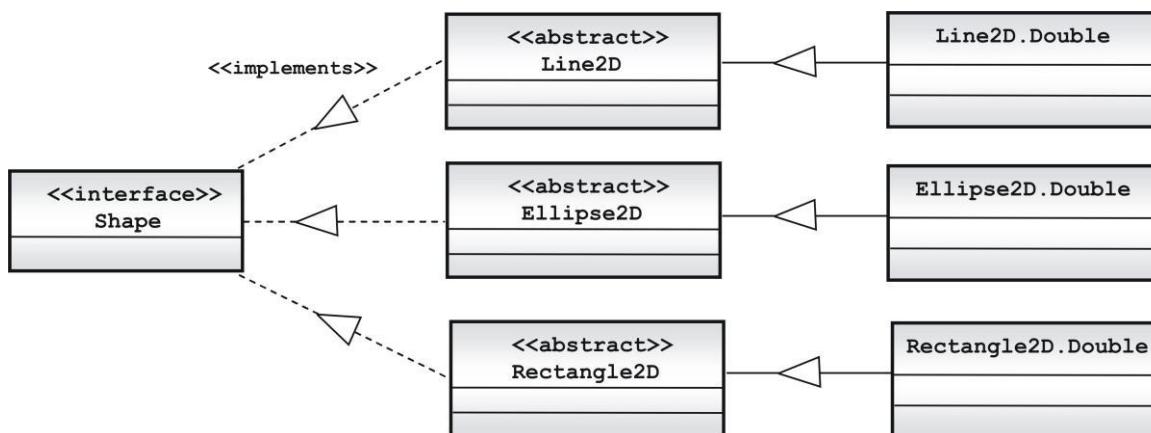


Fig. 21 – Jerarquía parcial de clases geométricas en Java.

En la siguiente tabla se resume la manera de crear instancias de las clases que implementan las figuras geométricas y el resultado. Puesto que las clases `Line2D`, `Ellipse2D` y `Rectangle2D` son clases abstractas, debemos crear instancias de las clases `Line2D.Double`, `Ellipse2D.Double` y `Rectangle2D.Double`:

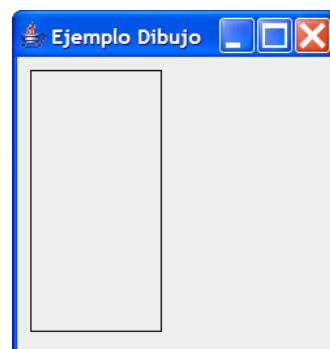
```

int ancho = 100;
int alto = 200;

// Primer parámetro: coordenada X
// Segundo parámetro: coordenada Y
// Tercer parámetro: ancho de la figura en píxeles
// Cuarto parámetro: alto de la figura en píxeles

Rectangle2D.Double rect1 = new Rectangle2D.Double( 10, 10,
                                                 ancho,
                                                 alto );

// Con este método pedimos a la superficie que dibuje el
// borde de la figura
g2.draw( rect1 );
  
```



```

int ancho = 100;
int alto = 200;

// Primer parámetro: coordenada X
// Segundo parámetro: coordenada Y
// Tercer parámetro: ancho del rectángulo que lo contiene
// Cuarto parámetro: alto del rectángulo que lo contiene

Ellipse2D.Double oval1 = new Ellipse2D.Double( 10, 10,
                                              ancho,
                                              alto );

// Con este método pedimos a la superficie que dibuje la
// figura rellena

g2.fill( oval1 );

```



```

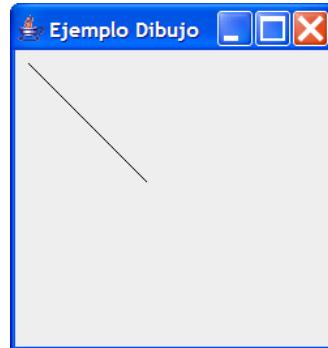
// Primer parámetro: coordenada X del origen
// Segundo parámetro: coordenada Y del origen
// Tercer parámetro: coordenada X del destino
// Cuarto parámetro: coordenada Y del destino

Line2D.Double lin1 = new Line2D.Double( 10, 10, 100, 100 );

// Con este método pedimos a la superficie que dibuje la
// línea

g2.draw( lin1 );

```



En el CD que acompaña el libro puede encontrar una herramienta que le permite utilizar de manera interactiva los métodos de dibujo de Java.



Objetivo: Estudiar la clase `BasicStroke` que permite configurar la línea que bordea las figuras geométricas que dibujamos.

Consulte la documentación (Javadoc) de la clase `BasicStroke` y conteste las preguntas que se plantean a continuación:

1. ¿Qué interfaz implementa la clase?

2. ¿Qué hace el método `setStroke()` de la clase `Graphics2D`?

3. Para cada uno de los parámetros del constructor de la clase, explique su uso y los valores que puede tomar:

Parámetro	Tipo	Valores posibles	Descripción
width			
cap			
join			

miterlimit			
dash			
dash_phase			

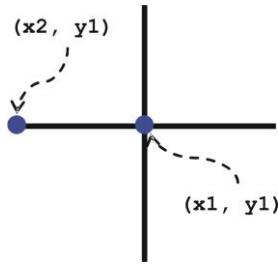
Vamos ahora a recorrer las distintas clases y métodos del editor de dibujos, para estudiar la manera como se utilizaron las clases y métodos antes descritos para construir la solución.

 TAREA #10	<p><u>Objetivo:</u> Recorrer el programa que resuelve el problema planteado en el caso de estudio.</p> <p>Siga los pasos que se dan a continuación y conteste las preguntas que se plantean.</p>
	<p>1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo <code>n10_paint.zip</code>, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.</p>
	<p>2. En la clase <code>Línea</code>. ¿Cómo está implementado el método de dibujo? ¿Qué métodos antes estudiados utiliza?</p>
	<p>3. En la clase <code>Línea</code>. ¿Cómo está implementado el método que indica si un punto se encuentra en su interior? Para qué se utiliza la constante <code>distanciaMinima</code>?</p>
	<p>4. En la clase <code>Rectángulo</code>. ¿Cómo está implementado el método de dibujo? ¿Cómo hace este método para indicar el caso en el cual el rectángulo se encuentra seleccionado?</p>
	<p>5. En la clase <code>Rectángulo</code>. ¿Cómo está implementado el método que indica si un punto se encuentra en su interior?</p>
	<p>6. En la clase <code>Óvalo</code>. ¿Cómo está implementado el método de dibujo? ¿Cómo hace este método para indicar el caso en el cual el óvalo se encuentra seleccionado?</p>

7. Estudiando los métodos encargados de la persistencia en las clases Figura y FiguraRellena, indique el formato con el cual se almacena la información de cada una de las figuras.	Línea	
	Ovalo	
	Rectángulo	
8. ¿Quién utiliza y para qué el método dibujarTexto() de la clase Figura? ¿Por qué se puede decir que es un método de servicio?		

3.15. Evolución del Editor

En esta sección vamos a trabajar en extensiones del editor de dibujos, para validar así la facilidad de evolución del programa. Para cada tarea vamos a seguir cuatro etapas: (1) hacer el análisis de la modificación pedida (entender el problema que se plantea con el cambio), (2) identificar el punto del diseño que se debe modificar y el impacto que tiene sobre el resto del programa (3) diseñar la extensión, construyendo el diagrama de clases y asignando, entre todos los componentes, las responsabilidades que acaban de aparecer e (4) implementar y probar lo diseñado en el punto anterior.

TAREA #11 	<p>Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva figura al editor del caso de estudio.</p> <p>Modifique el programa siguiendo las etapas que se plantean a continuación.</p>
	<ol style="list-style-type: none"> Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo <code>n10_paint.zip</code>, el cual está disponible en el CD que acompaña el libro o en el sitio WEB. Vamos a agregar al editor una nueva figura, que consiste en una cruz como la que aparece a continuación, y cuyas características se definen de la siguiente manera: <div style="display: flex; align-items: center;">  <ul style="list-style-type: none"> → La cruz está compuesta por dos líneas, una vertical y otra horizontal. → Las líneas se cortan exactamente en la mitad. (Todos los lados de la cruz deben tener la misma medida) → El valor $y2$ no es utilizado, y se utiliza $y1$ en los dos casos, para obtener siempre la cruz en una posición horizontal. → Para crear la cruz el usuario debe primero seleccionar el botón con la figura y, luego, hacer clic en el centro de la cruz seguido de un clic sobre el punto en el extremo izquierdo de la línea horizontal. </div>

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar la nueva figura.

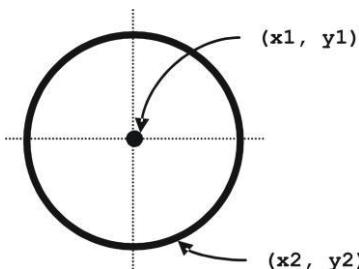
TAREA #12

Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva figura al editor del caso de estudio.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Vamos a agregar al editor una nueva figura, que consiste en una circunferencia como la que aparece a continuación:



- Para definir una circunferencia el usuario debe seleccionar desde la interfaz de usuario el botón con la respectiva figura, hacer clic en el centro (x_1, y_1) y luego hacer clic sobre cualquier punto que se encuentre sobre la circunferencia.
- Para seleccionar la figura, el usuario debe hacer clic sobre la circunferencia (no es una figura rellena).
- Se utilizan las mismas características de las demás figuras para manejar el ancho y el tipo de la línea.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar la nueva figura.

**TAREA #13**

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Asocie con el botón “Opción 1” un nuevo requerimiento funcional, que permita desplazar la figura seleccionada cinco píxeles a la derecha cada vez que se oprima.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

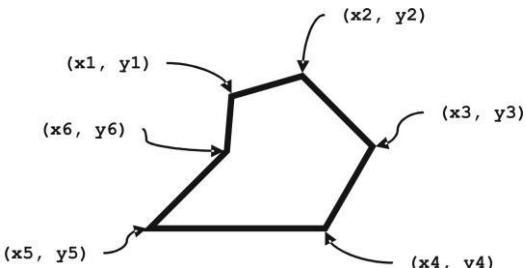
**TAREA #14**

Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva familia de figuras al editor del caso de estudio.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Vamos a agregar al editor una nueva figura, que consiste en un polígono de 6 puntos como el que aparece a continuación:



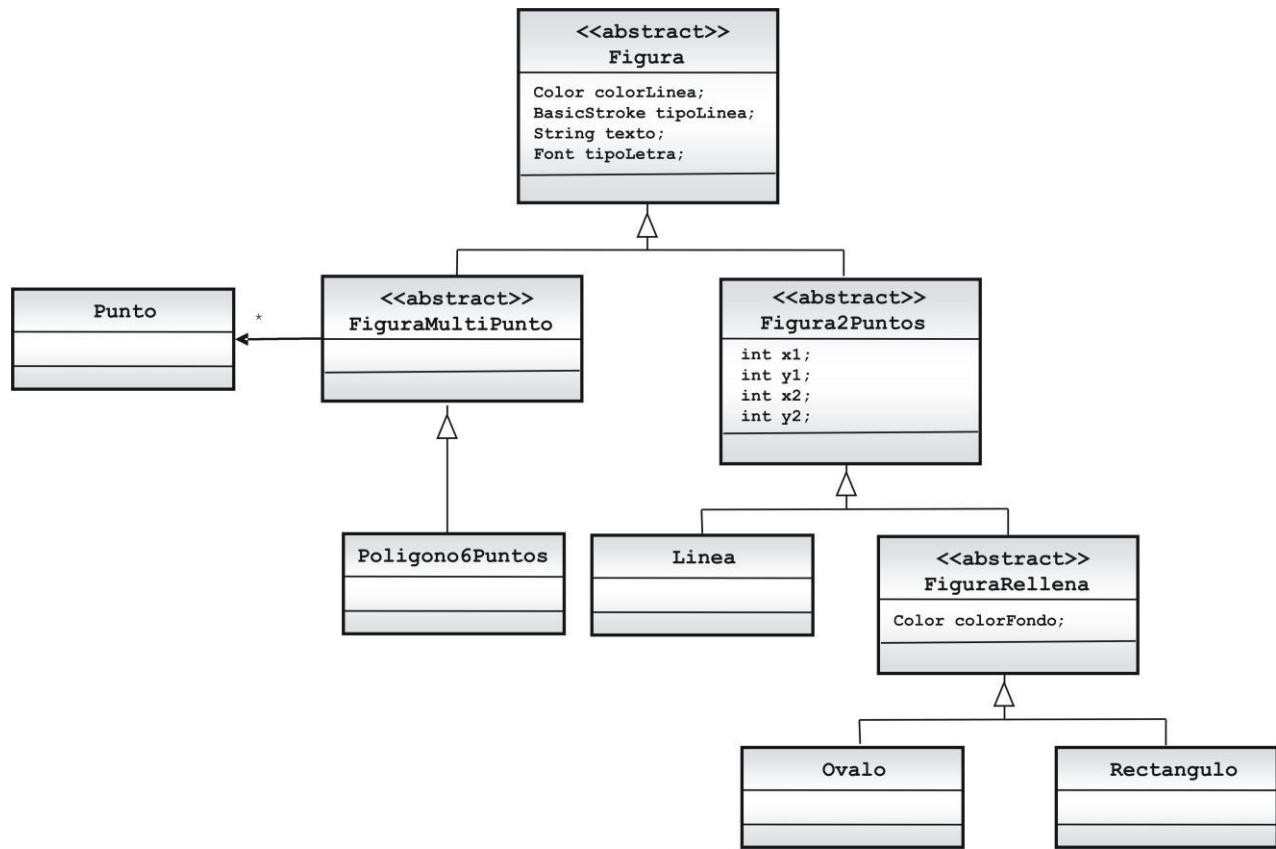
- Para definir un polígono el usuario debe seleccionar desde la interfaz de usuario el botón con la respectiva figura, y luego hacer 6 veces clic señalando los vértices del polígono.
- Para seleccionar la figura, el usuario debe hacer clic sobre el polígono (no es una figura rellena).
- Se utilizan las mismas características de las demás figuras para manejar el ancho y el tipo de la línea.

3. Elimine de la clase `Figura` los dos puntos que definen la geometría de la misma (atributos `x1, y1, x2, y2`). Cree la clase abstracta `Figura2Puntos`, que hereda de la clase `Figura` y que tiene los 4 atributos antes mencionados. Ajuste la herencia de las clases `Línea` y `FiguraRellena` para que ahora hereden de la clase `Figura2Puntos` y que todo siga funcionando correctamente.

4. Cree la clase `Punto`, que tiene como atributos una coordenada `x` y una coordenada `y`. Agréguele a esa clase un constructor, dos métodos para tener acceso al valor de cada coordenada (`darX()` y `darY()`) y dos métodos para modificar dichos valores (`cambiarX()` y `cambiarY()`).

5. Cree la clase `FiguraMultiPunto` que hereda de la clase `Figura`, y que tiene una contenedora de objetos de la clase `Punto`. Agregue los métodos necesarios para que esta clase siga cumpliendo los mismos requerimientos del resto de las figuras (persistencia e invariante, por ejemplo).

6. Cree la figura `Poligono6Puntos` que hereda de la clase `FiguraMultiPunto`. Haga todas las modificaciones necesarias en el programa para que funcione correctamente el editor con este nuevo tipo de figuras. El diagrama de clases que se debe obtener después de todas las modificaciones es el siguiente:

**TAREA #15**

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Agregue un nuevo requerimiento funcional al editor para que, cuando el usuario haga clic de nuevo sobre la figura seleccionada, aparezca una ventana en la que se incluya toda la información de la figura, incluida su área.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

**TAREA #16**

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Asocie con el botón “Opción 2” un nuevo requerimiento funcional, que cambie a mayúsculas los textos de todas las figuras del dibujo.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

**TAREA #17**

Objetivo: Probar la extensibilidad del diseño del programa,

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña el libro o en el sitio WEB.

2. Asocie con el botón “Opción 3” un nuevo requerimiento funcional, que permita cambiar el color de la línea que enmarca la figura. Para esto el programa debe presentar una ventana de selección de color.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta:

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

4. Glosario de Términos

GLOSARIO	Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base del trabajo de los niveles que siguen en el libro.
Interfaz	
Herencia	
Superclase	
Subclase	
Menú	
Desacoplamiento	
Clase Object	
Interfaz List	
Interfaz Collection	
Interfaz Iterator	
Reutilización	
Polimorfismo y asociación dinámica	
Conversión de tipos (<i>casting</i>)	

Atributos protegidos	
Redefinición de un método	
La variable super	
Clase abstracta	
Método abstracto	
Extensibilidad	
Clase Graphics2D	
Interfaz MouseListener	
Interfaz Shape	
Método paintComponent	

5. Hojas de Trabajo



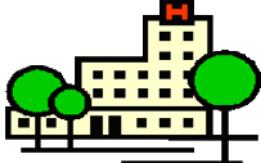
5.1. Hoja de Trabajo #1: Mapa de la Ciudad

Enunciado: Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se desea construir un programa que permita la creación de un mapa sencillo de una ciudad. Dicho mapa está constituido por construcciones que comparten características comunes.

Todas las construcciones tienen unas dimensiones (alto y ancho, medidas en píxeles), un color de fondo, un punto (x, y) que corresponde a la esquina superior izquierda donde se va a ubicar la construcción, y un texto que puede ser modificado y visualizado.

Las construcciones se encuentran divididas en edificaciones y carreteras. Una edificación puede ser una casa, un edificio, un hospital, una estación de policía o una estación de bomberos. Las carreteras pueden ser calles, carreras, glorietas y esquinas. A continuación se muestran esos elementos de la ciudad:

Nombre	Figura	Archivo	Alto	Ancho
Casa		Casa.gif	120	160
Edificio		Edificio.gif	160	200
Hospital		Hospital.gif	160	200
Estación de policía		EstacionPolicia.gif	160	200
Estación de bomberos		EstacionBomberos.gif	160	200
Calle		(se debe dibujar)	40	40
Carrera		(se debe dibujar)	40	40
Glorieta		(se debe dibujar)	40	40
Esquina 1		(se debe dibujar)	40	40
Esquina 2		(se debe dibujar)	40	40
Esquina 3		(se debe dibujar)	40	40
Esquina 4		(se debe dibujar)	40	40

Para dibujar una construcción en el mapa se debe seleccionar el tipo de construcción entre las opciones disponibles y localizar el ratón en la posición del mapa donde ésta se quiere ubicar. No se debe permitir que dos construcciones se sobrepongan. Para tal fin, si la ubicación seleccionada está libre, se debe mostrar la silueta de la construcción. Para crear la construcción basta hacer clic sobre la zona del mapa seleccionada. Para borrar una construcción o modificar el texto asociado, el usuario debe seleccionarla antes, haciendo clic sobre ella.

El programa debe permitir (1) agregar una construcción al mapa, (2) borrar del mapa la construcción seleccionada, (3) cambiar el texto que describe la construcción seleccionada, personalizando sus atributos estilo, tamaño y tipo de fuente. (4) guardar el mapa que se acaba de construir en un archivo y (5) cargar un mapa existente.

La interfaz de usuario que debe tener el programa es la siguiente:



Requerimientos funcionales: Especifique los requerimientos funcionales descritos en el enunciado

Nombre: R1 – Agregar una construcción al mapa.

Resumen:

Entradas:

Resultados:

Nombre:	R2 – Borrar una construcción del mapa.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R3 – Cambiar el texto que describe una construcción.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R4 – Guardar el mapa en un archivo.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R5 – Cargar un mapa.
Resumen:	
Entradas:	
Resultados:	

Modelo conceptual. Construya el diagrama inicial de clases con los cinco elementos que se describen a continuación y las asociaciones que los relacionan: (1) la clase `MapaCiudad` que representa un mapa completo, (2) la interfaz `IConstruccion` que define el contrato funcional de todas las construcciones (por ahora no le defina ningún método), (3) la clase abstracta `Construccion`, que incluye todas las características compartidas de las construcciones, (4) la subclase `Edificacion` (hereda de la clase `Construccion`) que incluye las características propias de este tipo de elementos y (5) la subclase `Carretera` (hereda también de la clase `Construccion`).

Declaración de clases. Escriba en Java la declaración de cada una de las clases del diagrama anterior. El grupo de elementos de tipo `IConstruccion` que tiene la clase `MapaCiudad`, debe ser definido utilizando la interfaz `List`.

```
public class MapaCiudad
{
}
```

```
public abstract class Construccion
{
}
```

Interfaces. Estudie el contrato funcional de la interfaz `IConstrucción` y llene la siguiente tabla con la descripción de los métodos. Para esto cree un proyecto en Eclipse a partir del archivo `n10_mapaCiudad.zip` que se encuentra en el CD que acompaña el libro. Localice el archivo `IConstrucion.java` y analice su contenido. Asegúrese de entender la responsabilidad de cada método.

Signatura	Descripción

Implementación. A continuación se proponen algunas tareas que van a permitir al lector estudiar la solución de esta parte del programa.

1. Si no lo ha hecho antes, cree un proyecto en Eclipse a partir del archivo `n10_mapaCiudad.zip` el cual se encuentra disponible en el CD que acompaña el libro. Localice la clase `InterfazMapaCiudad` y ejecute desde allí el programa. Utilice las distintas opciones disponibles para crear una ciudad.
2. Vamos a comenzar estudiando la clase `MapaCiudad`. Para esto edite el respectivo código desde Eclipse y haga el recorrido que se plantea a continuación por medio de preguntas:

Localice el método llamado `pintarConstrucciones()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro?

Localice el método llamado `eliminarConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro? ¿Cómo localiza la construcción que debe eliminar?

Localice el método llamado `buscarConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro?

Localice el método llamado `sobreponeConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro? ¿Para qué se utiliza la constante `DIFERENCIA`? ¿Qué condiciones verifica?

¿Qué condiciones exige el invariante de la clase?

Edite uno de los archivos en los que se almacenan los mapas (directorio “data”). Estudie la estructura de este archivo y haga un resumen de su contenido.

¿Qué método de la clase es el encargado de leer un mapa de un archivo? ¿Qué responsabilidades asume y cuáles delega en otras clases? ¿Cómo sabe de qué clase debe construir la instancia?	
¿Qué método de la clase es el encargado de escribir un mapa en un archivo? ¿Qué método de la interfaz <code>IConstruccion</code> utiliza?	
3. Pasemos ahora a la clase <code>Construccion</code> :	
¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?	
¿En qué coordenadas dibuja el texto el método <code>pintarTexto()</code> ? ¿Para qué se utiliza la clase <code>FontMetrics</code> ? ¿Quién invoca este método? ¿Por qué hace parte de esta clase?	
¿Por qué el método <code>pintar()</code> está declarado como abstracto?	
¿Por qué el método <code>darTipo()</code> está declarado como abstracto?	
¿Quién invoca los métodos <code>pintarSombreada()</code> y <code>pintarSeleccionada()</code> ? ¿En qué caso se invoca cada uno de ellos?	
¿Qué condiciones se verifican en el invariante de la clase?	
4. Edite la clase <code>Edificacion</code> :	
¿Qué atributos adicionales define esta clase? ¿Qué características modelan?	

¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?	
¿Cómo se implementa el método <code>pintar()</code> en esta clase?	
¿Por qué se declara como abstracta esta clase?	
5. Estudie ahora la clase <code>Carretera</code> :	
¿Qué atributos adicionales define esta clase? ¿Qué sentido tiene entonces definirla?	
¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?	
¿Por qué no hay una implementación del método <code>pintar()</code> en esta clase?	

Diagrama de clases. Dibuje el diagrama de clases con la jerarquía de herencia de la clase `Edificacion`.

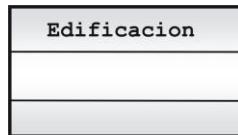
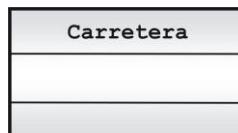


Diagrama de clases. Dibuje el diagrama de clases con la jerarquía de herencia de la clase `Carretera`.



Implementación. A continuación se proponen algunas tareas que van a permitir al lector estudiar la solución de esta parte del programa.

6. Para cada una de las clases que se dan a continuación, describa los métodos que implementa y la forma en que lo hace.

Calle	
Carrera	
Casa	
Edificio	
Esquinal	
Esquina2	
Esquina3	

	Esquina4
	EstacionBomberos
	EstacionPolicia
	Glorieta
	Hospital