



2.601 / 2.603

Programación II

Prof. Dr. Diego Corsi / Prof. Matías Ávalos

Unidad 1: El Modelo de Objetos	1
Unidad 2: Estado de los objetos.....	9
Unidad 3: Comportamiento de los objetos	17
Unidad 4: Herencia	21
Unidad 5: Encapsulamiento.....	29
Unidad 6: Polimorfismo.....	31
Repaso: Ejemplo práctico de Ingeniería de Software.....	71
Unidad 7: Excepciones.....	145
Unidad 8: Archivos	149
Unidad 9: Interfaces Gráficas de Usuario	165
Unidad 10: Acceso a Bases de Datos.....	191
Unidad 11: Aplicaciones Web	203
Unidad 12: Frameworks	219
Bibliografía	235



Unidad 1

El Modelo de Objetos

En vez de un procesador de bits consumiendo estructuras de datos, tenemos un universo de objetos bien comportados, cada uno de ellos pidiéndole a otro que cortésmente le realice sus variados deseos.¹

La frase anterior resume muy claramente lo que es un sistema orientado a objetos: un universo de objetos que interactúan. Veamos ahora algunas definiciones.

Objeto: Es una unidad que tiene estado, comportamiento e identidad. Un objeto puede caracterizar una entidad física (un teléfono, un cliente) o una entidad abstracta (un número, una fecha). Su **estado** está dado por los valores de sus “atributos” (“variables de instancia”), su **comportamiento** por las funciones (“métodos”) que ejecuta cuando recibe solicitudes (“mensajes”) y su **identidad** por sus nombres.

Clase: Es la declaración de los *miembros* (atributos y métodos) que un conjunto de objetos similares tienen en común. A partir de una clase, se crean (“instancian”) tantos objetos de la misma como sean necesarios al desarrollar determinado programa.

Por ejemplo, si en un universo hay dos perros - un dogo argentino y un pastor inglés - cuyo atributo característico es su ladrido, y deseamos hacer que ambos ladren (primero el dogo argentino y luego el pastor inglés), deberíamos enviarles el mensaje `ladrar()` a los objetos `dogoArgentino` y `pastorIngles`, previamente instanciados:

```
Universo.java
package perros;

public class Universo {

    private static Perro dogoArgentino;
    private static Perro pastorIngles;

    public static void main(String[] args) {
        dogoArgentino = new Perro("Guau guau!");
        pastorIngles = new Perro("Woof woof!");

        dogoArgentino.ladear();
        pastorIngles.ladear();
    }
}
```

```
Perro.java
package perros;

public class Perro {

    private String ladrido;

    public Perro(String s) {
        ladrido = s;
    }

    public void ladear() {
        System.out.println(ladrido);
    }
}
```

Obsérvese que la clase `Universo` tiene dos atributos (`dogoArgentino` y `pastorIngles`) y un método (`main`) dentro del cual se instancian los objetos (con el operador `new`) y se les envían mensajes. La ejecución de un programa en Java empieza siempre por el método `main`, que debe estar en alguna de las clases del proyecto.

¹ *"Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires."*

Ingalls, Daniel H. H. “Design Principles Behind Smalltalk”. En: *BYTE Magazine* (Agosto, 1981). McGraw-Hill, Nueva York, p. 290



La clase **Perro** tiene un atributo (**ladrido**, que es un objeto de la clase **String**) y un método (**ladrar**, que no recibe argumentos ni devuelve ningún valor).

Cada vez que se instancia un objeto **Perro**, se ejecuta el *constructor* de la clase, el cual se encarga de asignarle al atributo **ladrido** el valor recibido en el parámetro **s**. Es importante destacar que los constructores no son métodos (no tienen valor de retorno, ni siquiera **void**).

Cada vez que se le solicita a una instancia de **Perro** que ladre (invocando su método **ladrar**, al cual se accede colocando el nombre del objeto seguido del operador **.** y el nombre del método), se muestra por pantalla el valor del atributo **ladrido** de ese objeto, mediante el método **println** del objeto **System.out** (la salida estándar)¹.

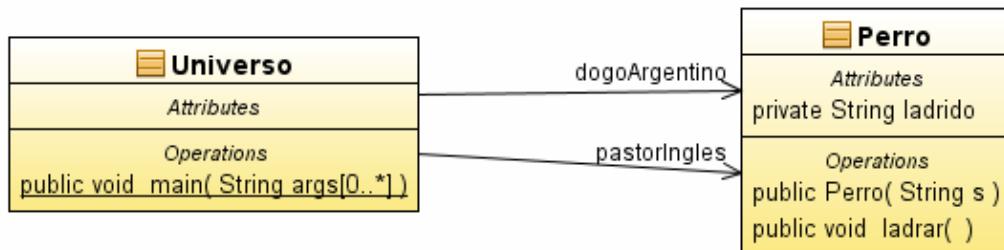
Por una cuestión de orden, las clases se agrupan en paquetes. En el ejemplo anterior, ambas clases pertenecen al paquete **perros**. El paquete al que pertenece una clase se indica en la cláusula **package**.

El nombre del archivo donde se guarda una clase *pública* (aquella a la que puede accederse desde clases alojadas en otros paquetes) debe coincidir con el nombre de la clase. Por ejemplo, la clase **Perro** debe guardarse en el archivo **Perro.java**.

Por convención, en Java los nombres de las clases comienzan con mayúscula.

Como veremos más adelante, es conveniente calificar los atributos como **private** y los constructores y la mayoría de los métodos como **public**.

El lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad es el UML (*Unified Modeling Language* o Lenguaje Unificado de Modelado), un lenguaje gráfico para especificar, construir y documentar sistemas. UML cuenta con varios tipos de diagramas. Por ejemplo, un *diagrama de clases* es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases. El diagrama de clases correspondiente al ejemplo anterior es el siguiente:



En la clase **Universo**, el modificador **static** del método **main** indica que se trata de un método estático (que puede ser invocado a pesar de no haber sido instanciado un objeto de la clase a la que pertenece). Para poder ser utilizado dentro de un método estático, un atributo o un método también debe ser estático, como ocurre con **dogoArgentino** y **pastorIngles**. En cambio, en la clase **Perro**, dentro del método **ladrar** (no estático) es posible usar el atributo **ladrido** (que tampoco es estático).

¹ Observe que se cumple el principio “*Tell, don’t ask*”. En vez de pedirle al perro que devuelva el ladrido y mostrarlo en otra clase, se le dice directamente que ladre. Esto es un correcto diseño OO.



Mis anotaciones: Cómo implementar un proyecto en Java usando NetBeans

Pasos para crear el proyecto y probarlo:

.....
.....
.....

Pasos para formatear el código y dejarlo presentable:

.....
.....
.....

Pasos para generar la documentación con javadoc:

.....
.....
.....

Pasos para generar los diagramas de clases UML:

.....
.....
.....

Pasos para realizar el despliegue (*deployment*) del proyecto:

.....
.....
.....

Pasos para correr el sistema terminado:

.....
.....
.....



En el siguiente ejemplo puede observarse cómo desde el método `main` de la clase `Calculos` se utilizan métodos y atributos estáticos de la clase `Aritmetica` sin instanciar ningún objeto de esa clase:

```
Calculos.java
package calculos;

public class Calculos {
    public static void main(String[] args) {
        System.out.println(
            Aritmetica.sumar(4, 5));
        System.out.println(
            Aritmetica.restar(7, 3));
        System.out.println(
            Aritmetica.cantidad);
    }
}
```

```
Aritmetica.java
package calculos;

public class Aritmetica {
    public static int cantidad = 0;
    public static int sumar(int a, int b) {
        cantidad++;
        return a + b;
    }
    public static int restar(int a, int b) {
        cantidad++;
        return a - b;
    }
}
```

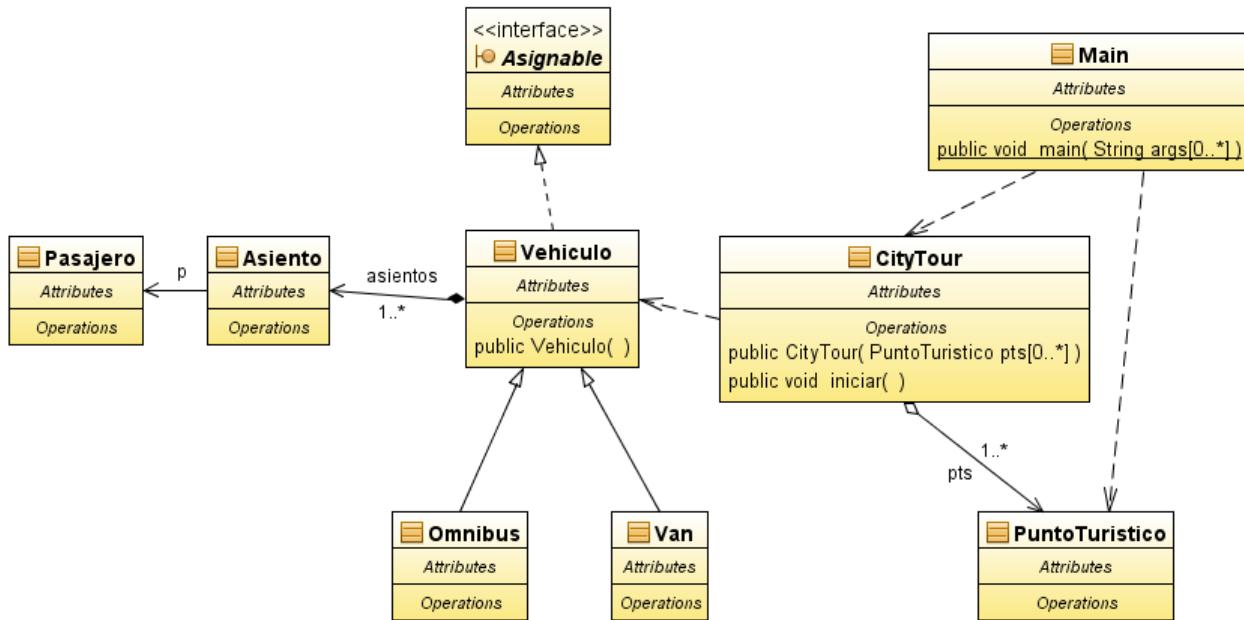
En la clase `Aritmetica`, la variable `cantidad` es un *atributo de clase* y las funciones `sumar` y `restar` son *métodos de clase*. Obsérvese que en un desarrollo estrictamente orientado a objetos, la utilización de estos recursos es poco frecuente.

El desarrollo del software orientado a objetos normalmente comienza con la definición de un *modelo*. Un **modelo** es una representación de un sistema del mundo real que resulta de llevar a cabo el proceso de *abstracción*. La **abstracción** es una simplificación que incluye sólo aquellos detalles relevantes para determinado propósito y descarta los demás. Las diferentes clases definidas durante el proceso de abstracción guardan ciertas *relaciones* entre sí, que pueden ser de diversos tipos:

- **Generalización:** Una clase derivada es un (*is-a*) tipo especial de otra clase más general. Los objetos de la clase derivada “heredan” los miembros no privados de la clase base o *superclase* y pueden ser usados en contextos que requieran objetos de esta última. En Java, la clase derivada extiende (*extends*) la clase base.
- **Realización:** Una clase cumple con un contrato especificado en una interfaz. En Java, la clase implementa (*implements*) la interfaz.
- **Asociación:** Una clase tiene un atributo que es una instancia de otra clase. La relación puede tener cualquier significado: “tiene” (*has-a*), “viaja en”, “visita a”, etc.
- **Agregación:** Una clase (“el todo”) está compuesta de atributos (“las partes”) que son instancias de otras clases independientes. En Java, los objetos componentes se reciben ya instanciados como argumentos en el constructor de la clase compuesta.
- **Composición:** Una clase (“el todo”) está compuesta de atributos (“las partes”) que son instancias de otras clases que existen exclusivamente para ser sus componentes. El objeto compuesto y sus componentes tienen el mismo ciclo de vida. En Java, los objetos componentes se instancian dentro de la clase compuesta.
- **Dependencia:** Una clase usa (*uses*) otra clase y puede verse afectada si ésta cambia. En Java, la clase dependiente usa variables locales que son instancias de la otra clase, o tiene métodos que reciben argumentos o retornan instancias de ésta.



El siguiente diagrama de clases muestra cómo se representan en UML las relaciones entre clases:



Dentro del método `main` de la clase **Main** se declara y crea una lista de objetos de la clase **PuntoTuristico** y se la utiliza como argumento del constructor al crear luego un objeto de la clase **CityTour**. Por ello, tanto entre **Main** y **PuntoTuristico** como entre **Main** y **CityTour**, la relación existente es una:

Dado que un objeto de la clase **CityTour** tiene como atributo una lista de instancias de **PuntoTuristico**, pero ésta fue creada fuera de la clase **CityTour** y en consecuencia puede existir independientemente de ella, la relación entre **CityTour** y **PuntoTuristico** es una:

Dentro del método `iniciar` de la clase **CityTour** se declara e instancia un **Vehiculo**. Entonces, la relación entre estas clases es una:

Tanto **Omnibus** como **Van** son subclases de **Vehiculo**. La relación entre **Omnibus** y **Vehiculo** y la relación entre **Van** y **Vehiculo** es una:

La clase **Vehiculo** implementa la interfaz **Asignable**. Por lo tanto, la relación entre **Vehiculo** y **Asignable** es una:

Un vehículo tiene como atributo una lista de instancias de **Asiento**. Como ésta se crea dentro del constructor de **Vehiculo**, sólo puede existir mientras exista la instancia de **Vehiculo**. Por lo tanto, la relación entre **Vehiculo** y **Asiento** es una:

Cada asiento “es ocupado por” un pasajero. Por eso, cada instancia de **Asiento** contiene en su atributo `p` una referencia a un objeto de la clase **Pasajero**. Entonces, la relación entre **Asiento** y **Pasajero** es una:



EJERCICIO N° 1

Modele las siguientes clases utilizando UML:

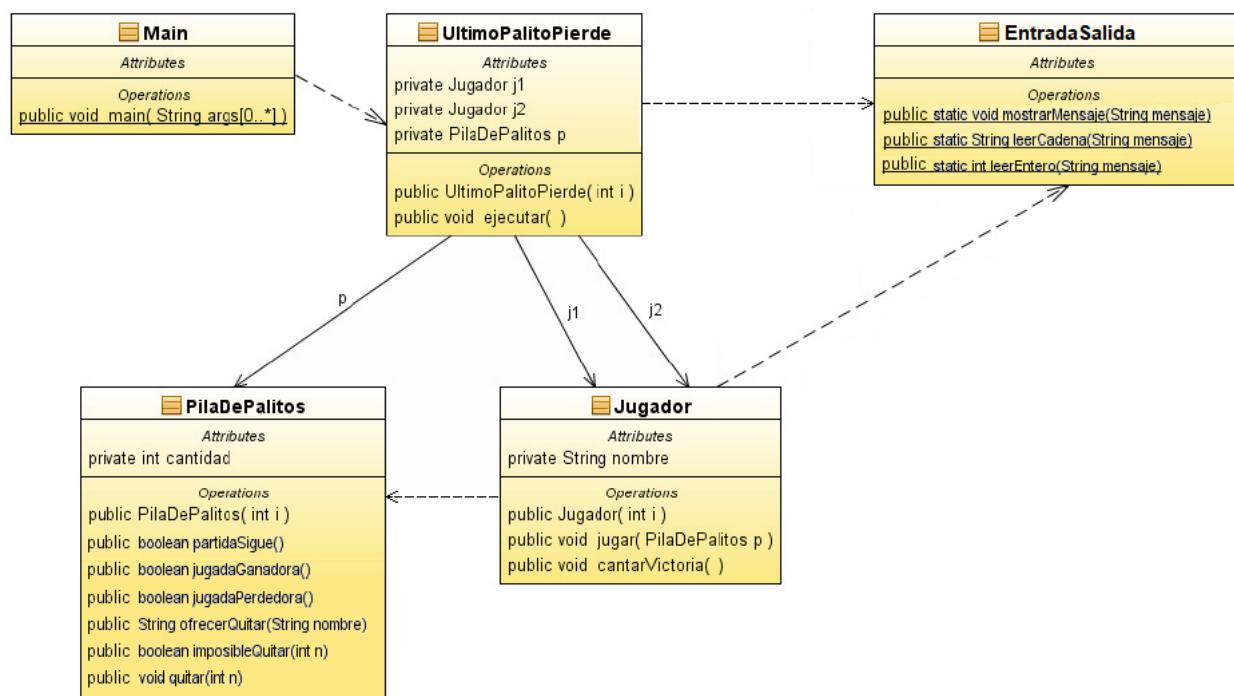
- 1) Punto
- 2) Circulo
- 3) Triangulo
- 4) NumeroFraccionario
- 5) CuentaCorriente
- 6) Ascensor

Prográmelas en Java e implemente nuevos proyectos con ellas.

EJERCICIO RESUELTO

Último Palito Pierde

Hay N palitos apilados. Cada uno de los 2 jugadores sacará 1, 2 o 3 palitos por turno. El jugador que saca el último palito pierde.



Main.java

```
package palitos;

public class Main {

    public static void main(String[] args) {
        UltimoPalitoPierde juego = new UltimoPalitoPierde(50);
        juego.ejecutar();
    }
}
```



EntradaSalida.java

```
package palitos;

import javax.swing.*;

public class EntradaSalida {

    public static void mostrarMensaje(String mensaje) {
        JOptionPane.showMessageDialog(null, mensaje);
    }

    public static String leerCadena(String mensaje) {
        return JOptionPane.showInputDialog(mensaje);
    }

    public static int leerEntero(String mensaje) {
        String strLeida = JOptionPane.showInputDialog(mensaje);
        int iLeido;
        try {
            iLeido = Integer.parseInt(strLeida);
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(null, "El valor ingresado es incorrecto.\n" + ex);
            iLeido = 0;
        }
        return iLeido;
    }
}
```

UltimoPalitoPierde.java

```
package palitos;

import static palitos.EntradaSalida.*;

public class UltimoPalitoPierde {

    private final Jugador j1;
    private final Jugador j2;
    private final PilaDePalitos p;

    public UltimoPalitoPierde(int i) {
        mostrarMensaje("Hay " + i + " palitos apilados."
                + "\nCada uno de los 2 jugadores sacará 1, 2 o 3 palitos por turno."
                + "\nEl jugador que saca el último palito pierde."
                + "\nSuerte!");
        p = new PilaDePalitos(i);
        j1 = new Jugador(1);
        j2 = new Jugador(2);
    }

    public void ejecutar() {
        while (p.partidaSigue()) {
            j1.jugar(p);
            if (p.jugadaGanadora()) {
                j1.cantarVictoria();
            } else if (p.jugadaPerdedora()) {
                j2.cantarVictoria();
            } else {
                j2.jugar(p);
                if (p.jugadaGanadora()) {
                    j2.cantarVictoria();
                } else if (p.jugadaPerdedora()) {
                    j1.cantarVictoria();
                }
            }
        }
    }
}
```



Jugador.java

```
package palitos;
import static palitos.EntradaSalida.*;

public class Jugador {

    private String nombre;

    public Jugador(int i) {
        do {
            nombre = leerCadena("Jugador Nro. " + i + ", cómo te llamas?");
        } while (nombre.equals(""));
    }

    public void jugar(PilaDePalitos p) {
        int n;
        do {
            n = leerEntero(p.ofrecerQuitar(nombre));
        } while (p.imposibleQuitar(n));
        p.quitar(n);
    }

    public void cantarVictoria() {
        mostrarMensaje(nombre + " es el vencedor!");
    }
}
```

PilaDePalitos.java

```
package palitos;
public class PilaDePalitos {

    private int cantidad;

    public PilaDePalitos(int i) {
        cantidad = i;
    }

    public boolean partidaSigue() {
        return cantidad > 1;
    }

    public boolean jugadaGanadora() {
        return cantidad == 1;
    }

    public boolean jugadaPerdedora() {
        return cantidad == 0;
    }

    public String ofrecerQuitar(String nombre) {
        return nombre + ", quedan " + cantidad + " palitos. "
            + "Cuántos retiras (1 a " + (cantidad > 3 ? 3 : cantidad) + "?";
    }

    public boolean imposibleQuitar(int n) {
        return n < 1 || n > 3 || n > cantidad;
    }

    public void quitar(int n) {
        cantidad -= n;
    }
}
```



Unidad 2

Estado de los objetos

Como se mencionó en la unidad anterior, los objetos que interactúan en un sistema orientado a objetos tienen un **estado** que está dado por los valores de sus “atributos” (“variables de instancia”) en determinado momento.

Generalmente, el estado de un objeto evoluciona en el tiempo, ya que desde sus métodos es posible acceder a sus atributos y modificarlos el valor. La excepción son los atributos calificados como **final**, ya que éstos permanecen *constantes*.

Los atributos de un objeto no deberían ser manipulables directamente por el resto de los objetos del sistema, por eso casi siempre se los califica como **private** y se accede a ellos mediante métodos. Este asunto se trata detalladamente en la Unidad 5.

Por convención, en Java los nombres de los atributos empiezan en minúsculas. De nuevo, la excepción son los atributos calificados como **final**, que se escriben totalmente en mayúsculas. Se recomienda *declarar* los atributos (es decir, indicar su tipo y su nombre) antes de los constructores, y utilizar éstos para *inicializarlos* (es decir, asignarles su primer valor). Por ejemplo:

```
package fabrica;  
  
public class Usuario {  
    private int clave;  
    private DocumentoDeIdentidad doc;  
    public Usuario(int clave) {  
        this.clave = clave;  
        doc = new DocumentoDeIdentidad();  
    }  
    public void ingresar() {  
        .  
        .  
    }  
}
```

En el caso anterior, los atributos **clave** y **doc** se inicializan dentro del constructor de **Usuario**. Como el valor para inicializar el atributo **clave** se recibe en el constructor mediante un parámetro que también tiene el nombre **clave**, debe utilizarse **this.clave** para referirse al atributo. En el caso del atributo **doc**, éste se inicializa dentro del constructor con una instancia de **DocumentoDeIdentidad** creada en ese mismo lugar.

Recordemos que dentro del método **ingresar** es posible acceder a **clave** y a **doc**.

PREGUNTA DE REPASO

¿Qué relación hay entre la clase **Usuario** y la clase **DocumentoDeIdentidad**? ¿Por qué?

.....

.....



En Java, cualquier variable declarada usando un tipo primitivo contendrá directamente un dato. Tal es el caso de la variable `clave` en el ejemplo anterior. En cambio, si el tipo usado para declarar una variable no es primitivo, ésta contendrá una referencia a un objeto, es decir, la dirección de memoria donde el objeto está almacenado. En el ejemplo anterior, esto es lo que ocurría con la variable `doc`.

Los 8 tipos primitivos en Java son los siguientes:

Tipo	Rango	Valor por defecto
<code>byte</code>	-128 .. 127	0
<code>short</code>	-32768 .. 32767	0
<code>int</code>	-2147483648 .. 2147483647	0
<code>long</code>	-9223372036854775808 .. 9223372036854775807	0L
<code>float</code>	-3.4E38 .. -1.18E-38 .. 0 .. 1.18E-38 .. 3.4E38	0.0f
<code>double</code>	-1.8E308 .. -2.23E-308 .. 0 .. 2.23E-308 .. 1.8E308	0.0d
<code>char</code>	'\u0000' .. '\uffff'	'\u0000'
<code>boolean</code>	false .. true	false

El valor por defecto se refiere al valor que toma una *variable de instancia* (es decir, un atributo) cuando se la usa sin inicialización previa. En el caso de las *variables locales* de los constructores y de los métodos, su uso sin inicialización previa no es posible (el compilador lo marca como un error).

Un atributo que represente el texto característico de un objeto (por ejemplo, el apellido de una persona), puede implementarse en Java como una instancia de la clase `String`. Esta clase tiene 13 constructores diferentes que permiten construir cadenas de caracteres a partir de otros elementos. Por ejemplo, si escribiéramos lo siguiente: `new String (new char[]{'h', 'o', 'l', 'a'})` estaríamos creando una instancia de `String` a partir de un arreglo *anónimo* de 4 caracteres (cada uno de los cuales aparece entre apóstrofos).

Por una cuestión de practicidad, en Java es posible crear un objeto de la clase `String` simplemente colocando caracteres entre comillas. Por ello, al atributo `apellido` (previamente declarado como `String`) se le puede asignar directamente la referencia al objeto "`Corsi`", creado en forma automática sin necesidad de utilizar el operador `new`, como puede observarse en el siguiente ejemplo:

```
private String apellido;
.
.
.
apellido = "Corsi";
```

El valor por defecto de los atributos que son instancias de la clase `String` (o de cualquier otra clase) es `null`. Cuando una variable vale `null`, ésta no se refiere a ningún objeto, por ello no es posible enviarle mensajes. La cadena vacía "" sí es un objeto. Por lo tanto, una variable `String` inicializada con la cadena vacía puede responder a los mensajes que le envíemos.



Los mensajes que puede responder una instancia de la clase `String` son los métodos que implementa esta clase. Los más comúnmente utilizados son los siguientes: `length`, `equals`, `charAt`, `substring`, `concat`, `contains`, `indexOf`, `isEmpty`, `trim`, `startsWith`, `endsWith`, `toLowerCase`, `toUpperCase`, `replace`, `replaceAll`.

Mis anotaciones: Qué hacen y cómo se usan los principales métodos de String

`length`

lee la cantidad de chars que tiene un string

`equals`

booleano que verifica si los objetos son iguales a

`charAt`

devuelve el carácter en la posición pedida

`substring`

separa una cadena en 2 cadenas

`concat` (aunque existe este método, lo más usual es usar el operador)
enlaza 2 cadenas

`contains`

booleano que confirma que

`indexOf`

devuelve un entero buscando algo dentro de otro

`isEmpty`

booleano que confirma si la cadena está vacía

`trim`

`startsWith` / `endsWith`

`toLowerCase` / `toUpperCase`

`replace` / `replaceAll`



Un atributo que represente un grupo de elementos del mismo tipo primitivo (por ejemplo, los números premiados en un sorteo) o un grupo de elementos de la misma clase (por ejemplo, los alumnos inscriptos en un curso) puede implementarse en Java como un arreglo.

Un arreglo es un objeto contenedor cuyo tamaño se establece al momento de su instanciación. El acceso a los elementos contenidos en el arreglo se realiza a través de un índice entero encerrado entre corchetes. El índice del primer elemento del arreglo es el cero.

Las variables que hacen referencia a arreglos se declaran colocando un par de corchetes vacíos antes o después de su nombre. Por convención, la primera forma es la variante preferida. Por ejemplo:

```
private int[] arrDNI;      // Declaración correcta. Forma recomendada  
  
private int arrOtrosDNI[]; // Declaración correcta. Forma no recomendada
```

Una vez declarada una variable que hará referencia a un arreglo, es necesario instanciar este último antes de comenzar a usarlo:

```
arrDNI = new int[100]; // El arreglo contendrá 100 elementos enteros
```

Para acceder a cierta posición del arreglo, se especifica su índice entre los corchetes:

```
arrDNI[3] = 20853678; // Guarda un entero en la posición 3 del arreglo  
  
System.out.println(arrDNI[3]); // Obtiene y muestra el entero guardado  
// en la posición 3
```

Cuando se trata de atributos de una clase, es conveniente instanciar los arreglos en los constructores. En otros casos (por ejemplo, cuando los arreglos son variables locales dentro de métodos), pueden declararse, instanciarse y cargarse los arreglos mediante una única sentencia:

```
int[] mesesLargos = {1, 3, 5, 7, 8, 10, 12};
```

En el caso anterior, para realizar la instanciación, el tamaño del arreglo se calcula automáticamente considerando la cantidad de datos que aparecen entre las llaves.

La propiedad `length` de un arreglo siempre contiene el tamaño del mismo, como puede observarse en el siguiente ejemplo:

```
for (int i = 0; i < mesesLargos.length; i++){  
    System.out.println(mesesLargos[i]);  
}
```



Los arreglos multidimensionales se declaran utilizando un par de corchetes por cada dimensión. Por ejemplo, un arreglo bidimensional (*matriz*) de enteros, se declarará así:

```
int[][][] imagen2D;
```

Para instanciar el arreglo, *puede* indicarse el tamaño de todas las dimensiones:

```
imagen2D = new int[10][10]; // El arreglo tendrá 10 filas y 10 columnas
```

Sin embargo, como los arreglos multidimensionales, en realidad, son arreglos de arreglos, la instanciación puede realizarse por partes. En el caso de un arreglo bidimensional, por ejemplo, podría instanciarse primero el arreglo vertical y luego, a medida que se lo va recorriendo, asignarle a cada posición un nuevo arreglo horizontal. Aunque lo más común es que todas las filas tengan igual cantidad de columnas, esto no es un requerimiento. Por ejemplo, el siguiente arreglo tiene una columna en la primera fila (la fila 0), dos columnas en la segunda fila, tres columnas en la tercera fila, etc.:

```
imagen2D = new int[10][]; // El arreglo tendrá 10 filas

for (int i = 0; i < imagen2D.length; i++) {
    imagen2D[i] = new int[i+1];
}
```

A veces, es necesario guardar una copia del contenido de un arreglo. Por ejemplo, si necesitamos tener en un arreglo ciertas letras en minúsculas y en otro arreglo las mismas letras en mayúsculas, podríamos generar una copia del primer arreglo y luego pasar a mayúsculas el contenido copiado. El siguiente código muestra un intento fallido para realizar esta tarea:

```
char[] minusculas = {'h', 'o', 'l', 'a'};

char[] mayusculas;

mayusculas = minusculas; // Si se tratara de variables primitivas,
                        // estaríamos copiando minusculas en mayusculas

for (int i = 0; i < mayusculas.length; i++) {
    mayusculas[i] = Character.toUpperCase(minusculas[i]);
}

System.out.println(new String(minusculas));
System.out.println(new String(mayusculas));
```

OBSERVACIÓN

System.out.println(new String(minusculas)); muestra:

System.out.println(new String(mayusculas)); muestra:



Lo que ocurre en el ejemplo anterior es que, como los arreglos son objetos, la asignación `mayusculas = minusculas` no copia los caracteres contenidos en el arreglo `minusculas` en el arreglo `mayusculas`. De hecho, lo que copia la asignación es la dirección de memoria donde está guardado el objeto a que hace referencia la variable de la derecha. Por ello, ambas variables terminan refiriéndose al mismo objeto, y los cambios hechos a éste cuando es referenciado mediante la variable `mayusculas` también se ven cuando más tarde se lo referencia mediante la variable `minusculas`.

En Java, la clase `java.util.Arrays` proporciona el método `copyOfRange`, el cual permite generar copias completas o parciales de arreglos.

```
char[] minusculas = {'h', 'o', 'l', 'a'};  
char[] mayusculas;  
  
mayusculas = java.util.Arrays.copyOfRange(minusculas, 0, 4);  
  
for (int i = 0; i < mayusculas.length; i++) {  
    mayusculas[i] = Character.toUpperCase(mayusculas[i]);  
}  
  
System.out.println(new String(minusculas));  
System.out.println(new String(mayusculas));
```

El segundo parámetro de `copyOfRange` corresponde al índice del primer elemento a copiar, y el tercer parámetro es el índice a partir del cual ya no se copia.

OBSERVACIÓN

`mayusculas = java.util.Arrays.copyOfRange(minusculas, 1, 3);` hace que el arreglo `mayusculas` contenga los siguientes elementos:

Otros métodos útiles proporcionados por la clase `java.util.Arrays` son: `equals`, `fill`, `binarySearch` y `sort`.

Mis anotaciones: Qué hacen y cómo se usan estos métodos de `java.util.Arrays`

`equals`

.....
.....

`fill`

.....
.....

`binarySearch`

.....
.....

`sort`

.....
.....



Aunque pueden utilizarse para codificar atributos que representen un grupo de elementos, los arreglos tienen la desventaja de poseer un tamaño fijo y, además, requieren que el programador implemente en los métodos muchas operaciones que son básicas como, por ejemplo, `add`, `remove`, `contains` y `size`. Por eso, como alternativa a los arreglos, Java proporciona una biblioteca denominada *Collections*, compuesta por varias interfaces y sus respectivas implementaciones. Entre las implementaciones más utilizadas se encuentran las clases `ArrayList`, `LinkedList` y `TreeSet`.

Una instancia de la clase `ArrayList` se comporta como un arreglo instanciado sin especificarle el tamaño. Permite guardar una cantidad de **objetos** no conocida de antemano, ya que su tamaño se irá ajustando automáticamente. El siguiente fragmento de código muestra los métodos de `ArrayList` usados más frecuentemente:

```
ArrayList<String> apellidos = new ArrayList<>();
System.out.println("apellidos esta vacio: " + apellidos.isEmpty());
apellidos.add("Lopez");
apellidos.add("Martinez");
apellidos.add(0, "Alvarez");
apellidos.add("Garcia");
System.out.println("Apellidos: " + apellidos);

System.out.print("Recorrido con for tradicional: ");
for (int i = 0; i < apellidos.size(); i++) {
    System.out.print(apellidos.get(i) + " ");
}
System.out.println();

System.out.print("Recorrido con for mejorado: ");
for (String str : apellidos) {
    System.out.print(str + " ");
}
System.out.println();

System.out.print("Recorrido con iterador: ");
Iterator<String> it = apellidos.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + " ");
}
System.out.println();

System.out.println("Martinez está en: " + apellidos.indexOf("Martinez"));
System.out.println("Dominguez está: " + apellidos.contains("Dominguez"));
System.out.println("En la posicion 1 se encuentra: " + apellidos.get(1));
apellidos.set(1, "Rodriguez");
System.out.println("Lopez fue reemplazado por Rodriguez: " + apellidos);
apellidos.remove(1);
System.out.println("Despues de borrar a Rodriguez: " + apellidos);
apellidos.remove("Alvarez");
System.out.println("Despues de borrar a Alvarez: " + apellidos);
```



Mis anotaciones: Qué hacen los principales métodos de java.util.ArrayList

isEmpty
size
add
set
get
remove
contains
indexOf
iterator

Para investigar

¿Qué es un iterador? ¿Cuáles son sus principales métodos?

.....
.....

¿Qué métodos de la clase `LinkedList` la distinguen de `ArrayList`? ¿Por qué?

.....
.....

¿Qué caracteriza a los objetos agregados en una instancia de `TreeSet`? (ver abajo)

.....
.....

```
TreeSet<Integer> tr = new TreeSet<>();  
tr.add(12); tr.add(63); tr.add(34); tr.add(45);  
System.out.println(tr)
```

En el ejemplo anterior, se declaró que `tr` contendría objetos de la clase `Integer`. Sin embargo, luego se agregaron 12, 63, 34 y 45, que son valores del tipo primitivo `int`. Esto es válido, ya que Java tiene un mecanismo denominado *autoboxing* que convierte automáticamente datos primitivos en objetos para que puedan ser guardados en instancias de `ArrayList`, `LinkedList`, `TreeSet`, etc. Los objetos resultantes del *autoboxing* son siempre instancias de alguna de las ocho *wrapper classes*: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` y `Character`. Estas clases, además, ofrecen **métodos estáticos** muy útiles. Por ejemplo, en la pág. 14 se usó: El mecanismo opuesto al *autoboxing*, denominado *auto-unboxing*, también existe.

EJERCICIO N° 2

Realice una posible implementación de las clases mostradas en el diagrama de la pág. 5.



Unidad 3

Comportamiento de los objetos

Como se mencionó en la Unidad 1, todo objeto que interactúa en un sistema orientado a objetos tiene un **comportamiento** que está dado por las funciones (“métodos”) que ejecuta cuando recibe solicitudes (“mensajes”) de otros objetos.

También durante la instanciación de los objetos se ejecutan ciertas instrucciones, por ejemplo, para inicializar sus atributos. Este comportamiento está codificado en *constructores*, los cuales (a diferencia de los métodos) no tienen valor de retorno (ni siquiera `void`) y deben tener el mismo nombre que su clase.

En el siguiente ejemplo, que ya se mostró en la Unidad 1, la clase `Perro` tiene un constructor para inicializar el atributo `ladrido` (lo cual ocurre cada vez que desde la clase `Universo` se instancia un objeto de la clase `Perro`) y un método denominado `ladrar` que no recibe argumentos ni retorna un valor (dado que se usó `void` para indicar el tipo del valor de retorno) y que al ser invocado desde la clase `Universo` muestra por pantalla el valor del atributo `ladrido` del objeto correspondiente.

```
Universo.java
package perros;

public class Universo {

    private static Perro dogoArgentino;
    private static Perro pastorIngles;

    public static void main(String[] args) {
        dogoArgentino = new Perro("Guau guau!");
        pastorIngles = new Perro("Woof woof!");

        dogoArgentino.ladrar();
        pastorIngles.ladrar();
    }
}
```

```
Perro.java
package perros;

public class Perro {

    private String ladrido;

    public Perro(String s) {
        ladrido = s;
    }

    public void ladrar() {
        System.out.println(ladrido);
    }
}
```

Cada método debe limitarse a realizar **una única tarea bien definida**, y su nombre debe expresar esa tarea con efectividad. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.

Desde el resto de los objetos de un sistema se debería poder solicitarle a un objeto la ejecución de sus métodos, por eso a éstos casi siempre se los califica como `public`. Una excepción son aquellos métodos auxiliares que sólo se invocarán desde dentro del propio objeto, en cuyo caso se los debe hacer invisibles para los demás objetos, calificándolos como `private`. Este asunto se trata en detalle en la Unidad 5.

Recordemos que hay tres formas de invocar un método:

1. Pasándole un mensaje a un objeto, es decir, usando una variable que se refiera al objeto, seguida de punto (`.`) y el nombre del método. Ej:
2. Utilizando el nombre de la clase, seguido de punto (`.`) y el nombre de un método de la clase, si éste último es estático (`static`). Ej:
3. Utilizando sólo el nombre del método, si éste es parte de la misma clase.



En el siguiente ejemplo, que es una modificación de la clase Perro vista antes, se invoca el método que es parte de la misma clase. Como no se espera que este método auxiliar sea utilizado desde otras clases, se lo calificó como

```
package perros;  
  
public class Perro {  
  
    private String ladrido;  
  
    public Perro(String s) {  
        ladrido = s;  
    }  
  
    public void ladrar() {  
        System.out.println(ponerMayusculas(ladrido));  
    }  
  
    private String ponerMayusculas(String m) {  
        .  
        .  
    }  
}
```

Existen tres formas de finalizar un método y regresar el control al código que lo invocó. Si el método no devuelve un resultado, el control regresa cuando:

1. el flujo del programa llega a la llave de cierre del método,
2. se ejecuta la sentencia `return;`

Si el método devuelve un resultado, el control regresa cuando:

3. se ejecuta la sentencia `return expresión;` la cual evalúa la expresión y después devuelve el resultado al código que hizo la invocación.

Los métodos pueden devolver como máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga varios valores. El tipo del valor devuelto se indica antes del nombre del método, al declararse este último. Como ya se explicó anteriormente, la palabra reservada `void` se utiliza para indicar que un método no devuelve ningún valor.

Los métodos (y también los constructores) pueden tener *variables locales*. Recordemos que éstas no tienen valor por defecto, por lo que es obligatorio inicializarlas antes de utilizarlas, de lo contrario es imposible llevar a cabo la compilación. Solamente es posible acceder a las variables locales dentro del ámbito en que están declaradas, y al finalizar la ejecución del método o constructor al que pertenecen, el valor de las mismas no se mantiene.

Los métodos (pero no los constructores) pueden ser *recursivos*.

Un método (o un constructor) puede ser invocado con cero o más **argumentos**, si hay disponible una declaración que tenga una firma “compatible”. La firma (*signature*) de un método o constructor está compuesta por su nombre y los tipos de sus **parámetros**, por lo tanto es posible que haya múltiples versiones de un método o constructor, siempre y cuando sus firmas difieran. Esto se denomina *sobrecarga*.



Definiciones

Parámetro: Dato declarado entre paréntesis junto al nombre de un método.

Argumento: Valor que se le proporciona a un parámetro al invocar un método.

En el siguiente ejemplo, desde el método `probarSobrecarga` se invoca el método `cuadrado` una vez con un argumento `int`, lo cual es válido ya que hay una declaración del método con una firma compatible (la primera declaración de `cuadrado`), y otra vez con un argumento `double`, lo cual también es válido, ya que la segunda declaración del método tiene esa firma.

SobrecargaMetodos.java

```
public class SobrecargaMetodos {  
  
    public void probarSobrecarga(){  
        System.out.printf("El cuadrado del entero 7 es %d\n", cuadrado(7));  
        System.out.printf("El cuadrado del double 7.5 es %f\n", cuadrado(7.5));  
    }  
  
    private int cuadrado(int valorInt){  
        System.out.printf("Se ha invocado cuadrado con argumento int: %d\n", valorInt);  
        return valorInt * valorInt;  
    }  
  
    private double cuadrado(double valorDouble){  
        System.out.printf("Se ha invocado cuadrado con argumento double: %f\n", valorDouble);  
        return valorDouble * valorDouble;  
    }  
}
```

PruebaSobrecargaMetodos.java

```
public class PruebaSobrecargaMetodos {  
  
    public static void main( String args[]){  
        SobrecargaMetodos sm = new SobrecargaMetodos();  
        sm.probarSobrecarga();  
    }  
}
```

Otra característica importante de las invocaciones de los métodos es la **promoción de argumentos**: la conversión implícita del valor de un argumento al tipo que el método espera recibir en su correspondiente parámetro. Por ejemplo, una aplicación puede llamar al método estático `sqrt` de la clase `Math` con un argumento entero, a pesar de que el método espera recibir un argumento `double`. La sentencia `System.out.println(Math.sqrt(4));` evalúa sin problemas `Math.sqrt(4)` e imprime el valor `2.0`. La lista de parámetros de la declaración del método hace que se convierta el valor `int 4` en el valor `double 4.0` antes de pasarle ese valor al método `sqrt`.

Tratar de realizar estas conversiones puede ocasionar errores de compilación, si no se satisfacen las reglas de promoción que especifican qué conversiones son permitidas y pueden realizarse sin perder datos. En el ejemplo anterior de `sqrt`, un `int` se convierte en `double` sin modificar su valor. No obstante, la conversión de un `double` a un `int` trunca la parte fraccionaria del valor `double`; en consecuencia, se pierde parte del valor. La conversión de tipos de enteros largos a tipos de enteros pequeños (por ejemplo, de `long` a `int`) también puede producir valores modificados.



Las reglas de promoción se aplican a los valores de tipos primitivos que se les pasan como argumentos a los métodos. La tabla siguiente lista los tipos a los cuales se puede promover cada uno de los tipos primitivos. Observe que las promociones válidas para un tipo dado siempre se realizan a un tipo que aparece más arriba en la tabla. Por ejemplo, un `int` puede promoverse a los tipos más altos `long`, `float` y `double`.

Tipo	Promociones válidas
<code>double</code>	Ninguna
<code>float</code>	<code>double</code>
<code>long</code>	<code>float</code> o <code>double</code>
<code>int</code>	<code>long</code> , <code>float</code> o <code>double</code>
<code>char</code>	<code>int</code> , <code>long</code> , <code>float</code> o <code>double</code>
<code>short</code>	<code>int</code> , <code>long</code> , <code>float</code> o <code>double</code> (pero no <code>char</code>)
<code>byte</code>	<code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> o <code>double</code> (pero no <code>char</code>)
<code>boolean</code>	Ninguna (los valores <code>boolean</code> no se consideran números en Java)

Al convertir valores a tipos inferiores en la tabla, se producirán distintos valores si el tipo inferior no puede representar el valor del tipo superior (por ejemplo, el valor `int 2000000` no puede representarse como un `short`, y cualquier número de punto flotante con dígitos después de su punto decimal no puede representarse en un tipo entero como `long`, `int` o `short`). Por lo tanto, en casos en los que la información puede perderse debido a la conversión, el compilador requerirá que utilicemos un “casteo” para forzar explícitamente la conversión; en caso contrario, ocurre un error de compilación. En esencia decimos: “Sé que esta conversión podría ocasionar pérdida de información, pero aquí eso está bien”.

Suponga que el método `cuadrado` calcula y retorna el cuadrado de un entero y por ende requiere un argumento `int`. Para invocar a `cuadrado` con un argumento `double` llamado `valorDouble`, tendríamos que escribir la llamada al método de la siguiente forma: `cuadrado((int) valorDouble)`. Por ende, si el valor de `valorDouble` es `4.5`, el método recibe el valor y retorna, no

En Java, el pasaje de argumentos a los métodos es **por valor**. Lo que se le pasa al método invocado es una copia del valor del argumento. El método trabaja exclusivamente con la copia. Si el argumento utilizado en la invocación está almacenado en una variable, las modificaciones a la copia no afectan el valor de la variable original. Por ejemplo: `borrar(x)` no cambia el valor de la variable `x`.

Sin embargo, si el parámetro no es de un tipo primitivo, el método que recibe el argumento puede - a través de la copia - acceder a los miembros públicos del objeto a que hace referencia el parámetro e interactuar con él, incluso modificándole el estado. Por ejemplo, con el método que se muestra abajo, si `x` es una referencia a una instancia de `StringBuilder`, `borrar(x)` no modificará el valor de `x` (`x` seguirá refiriéndose al mismo objeto), pero luego de la invocación, la cadena almacenada en el objeto estará vacía:

```
public void borrar(StringBuilder s) {  
    s.setLength(0);  
}
```



Unidad 4

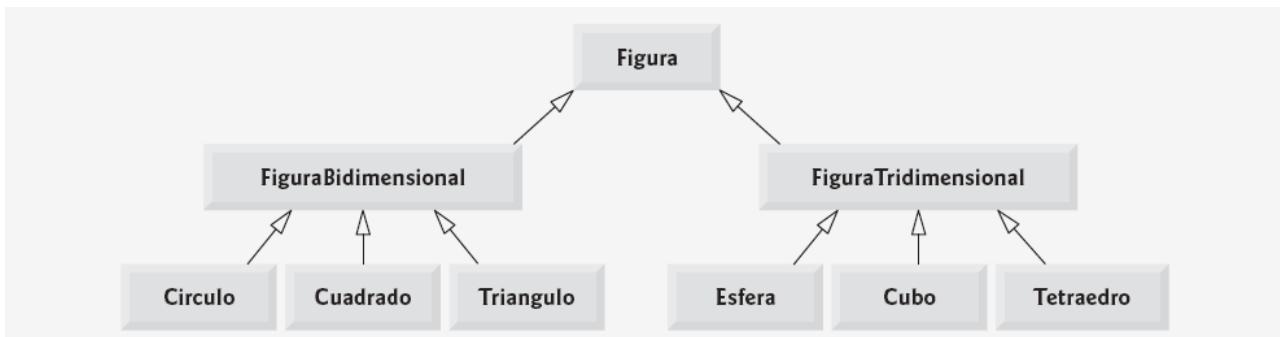
Herencia

Una de las características principales de la programación orientada a objetos es la **herencia**, que es una forma de reutilización de software en la que se crea una nueva clase aprovechando los miembros de *una* clase existente (*herencia simple*, como en Java) o de varias (*herencia múltiple*, como en C++). Con la herencia, los programadores ahorran tiempo durante el desarrollo, al reutilizar software probado y depurado de alta calidad. Esto también aumenta la probabilidad de que un sistema se implemente con efectividad.

A la clase previamente existente¹ se la conoce como **superclase** (o *clase base*), y a las nuevas clases se las conoce como **subclases** (o *clases derivadas*). Cada subclase puede convertirse en la superclase de futuras subclases.

Una subclase generalmente agrega sus propios atributos y métodos. Por lo tanto, una subclase es más específica que su superclase y representa a un grupo más especializado de objetos. Generalmente, la subclase exhibe los comportamientos de su superclase junto con comportamientos adicionales específicos de esta subclase. Es por ello que a la herencia se la conoce algunas veces como **especialización**.

La *superclase directa* es la clase de la cual la subclase hereda en forma explícita. Una *superclase indirecta* es cualquier clase que se encuentre arriba de la superclase directa en la **jerarquía de clases**, que es la jerarquía que define las relaciones de herencia entre las clases. Por ejemplo, en la siguiente jerarquía de clases, **FiguraBidimensional** es una superclase directa de **Circulo**, **Cuadrado** y **Triangulo**, mientras que **Figura** es una superclase indirecta de **Circulo**, **Cuadrado**, **Triangulo**, **Esfera**, **Cubo** y **Tetraedro**.



Cada flecha en la jerarquía resulta de realizar una **generalización** y representa una relación “es un”. Por ejemplo, al seguir las flechas en esta jerarquía de clases, podemos decir que un **Triangulo** es una **FiguraBidimensional** y es una **Figura**, así como una **Esfera** es una **FiguraTridimensional** y también es una **Figura**.

¹ En adelante, emplearemos exclusivamente el singular, ya que en este curso sólo vamos a usar la herencia simple. Cabe destacar que en Java es posible utilizar las *interfaces* para obtener muchos de los beneficios de la herencia múltiple, evitando al mismo tiempo los problemas asociados. Recordemos que la relación entre una clase y una interfaz se denomina **realización**.



En Java, la jerarquía de clases empieza con la clase `Object` (en el paquete `java.lang`), de la cual heredan *todas* las clases en Java, ya sea en forma directa o indirecta. Por ejemplo, todas las clases heredan de `Object` el método `finalize`, el cual se ejecuta cuando un objeto quedó desreferenciado (ninguna variable hace referencia a él) y su memoria será liberada por el *Garbage Collector*, que es un proceso que está permanentemente ejecutándose en segundo plano para “limpiar” la memoria.

El siguiente ejemplo muestra cómo la clase `ObjetoEducado`, por ser (de manera implícita) una subclase de `Object`, hereda el método `finalize`, el cual, en lugar de ser conservado tal como está definido en `Object`, es redefinido para mostrar un mensaje de despedida.

ObjetoEducado.java

```
package ejemploFinalize;

public class ObjetoEducado {

    public ObjetoEducado() {
        System.out.println("Soy un objeto educado: Digo HOLA al aparecer!");
    }

    public void finalize() {
        System.out.println("Soy un objeto educado: Digo CHAU antes de desaparecer!");
    }
}
```

Main.java

```
package ejemploFinalize;

public class Main {

    public static void main(String[] args) {
        ObjetoEducado obj = new ObjetoEducado();
        System.out.println("El objeto existe y es referenciado por la variable obj.");
        obj = null;
        System.out.println("El objeto aun existe pero ya no es referenciado.");
        System.gc();
    }
}
```

Cabe destacar entonces que, a menudo, un método de la superclase puede no ser apropiado para una subclase, ya que la subclase requiere una versión personalizada del método. En dichos casos, la subclase puede **sobrescribir** (redefinir) el método de la superclase con una implementación apropiada (*method overriding*).

El ejemplo anterior mostró un caso de herencia implícita, ya que todas las clases heredan de `Object`. La herencia, en Java, se hace explícita mediante la palabra reservada `extends`. Por lo tanto, aunque resulte redundante, podría haberse declarado la clase `ObjetoEducado` de la siguiente manera:

```
public class ObjetoEducado extends Object {
```



Las clases nuevas pueden heredar de las clases disponibles en **bibliotecas de clases**. Muchas organizaciones desarrollan sus propias bibliotecas de clases y también pueden aprovechar otras de terceros. Es probable que algún día, la mayoría del software nuevo se construya a partir de componentes reutilizables estándar, como sucede actualmente con la mayoría de los automóviles y del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

En el siguiente ejemplo, la clase **MiFecha** extiende la clase **Date**, que es una clase de la biblioteca de clases **java.util**. Por eso, las instancias de **MiFecha** (..... y) disponen de los mismos miembros públicos que las instancias de **Date** (..... y). En el caso del método **getTime**, la instancia de utiliza el método que heredó de **Date**. En cambio, el método fue sobrescrito, por lo tanto los comportamientos de **miHoy** y **miAyer** no serán iguales a los de **hoy** y **ayer**, cuando se invoque ese método.

MiFecha.java

```
package ejemploFechas;

import java.text.SimpleDateFormat;
import java.util.Date;

public class MiFecha extends Date {

    public MiFecha (){
    }

    public MiFecha (long ms){
        super(ms);
    }

    public String toString(){
        String s = new SimpleDateFormat("EEEE,' dd 'de' MMMM 'de' yyyy" +
                                         " 'a las' kk:mm:ss 'hs.'").format(this);
        return s.substring(0, 1).toUpperCase() + s.substring(1);
    }
}
```

Main.java

```
package ejemploFechas;

import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Date hoy = new Date();
        System.out.println(hoy.toString());

        Date ayer = new Date(hoy.getTime() - 86400000);
        System.out.println(ayer.toString());

        MiFecha miHoy = new MiFecha();
        System.out.println(miHoy.toString());

        MiFecha miAyer = new MiFecha(miHoy.getTime() - 86400000);
        System.out.println(miAyer.toString());
    }
}
```



Los constructores no se heredan. En Java, todas las clases tienen un **constructor por defecto**: el constructor sin parámetros. Como ocurre con todos los constructores, su primera tarea es llamar al constructor de su superclase directa, para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. Esto es así, porque siempre que se crea un objeto de una subclase se empieza una **cadena de llamadas** a los constructores, donde el constructor de la subclase, antes de realizar sus propias tareas, invoca al constructor de la superclase, ya sea en forma explícita (por medio de `super`) o implícita (llamando al constructor por defecto de la superclase). De igual forma, si la superclase deriva de otra clase, el constructor de la superclase invoca al constructor de la siguiente clase más arriba en la jerarquía, y así sucesivamente.

Debe destacarse que si se declara un constructor con parámetros, el constructor por defecto deja de estar disponible y, para poder instanciar objetos sin pasar argumentos, es necesario declarar un nuevo constructor sin parámetros.

En el ejemplo anterior, puede observarse que `Date` posee dos constructores: uno sin parámetros y otro que recibe los milisegundos correspondientes a una fecha. La clase `MiFecha`, por defecto, sólo tiene el constructor sin parámetros. Entonces, para poder instanciar `miAyer` usando como argumento los milisegundos correspondientes al día de ayer, fue necesario escribir un constructor de `MiFecha` que recibiera un `long` e invocara *explícitamente* (mediante) al constructor correcto de `Date`. Como `miHoy` se instancia sin pasar argumentos, también fue necesario declarar un nuevo constructor sin parámetros para `MiFecha`, el cual invoca *implícitamente* al constructor sin parámetros de `Date`.

EJERCICIO N° 3

Dibuje un diagrama UML que muestre las relaciones entre las siguientes clases. Indique, además, qué aparece por la salida del programa.

<pre>package cadenadellamadas; public class Main { public static void main(String[] args){ Torcaza t = new Torcaza("torcaza"); } }</pre>	<pre>package cadenadellamadas; public class Torcaza extends Paloma { public Torcaza(String s) { System.out.println("Soy una "+s+"!"); } }</pre>
<pre>package cadenadellamadas; public class Animal { public Animal() { System.out.println("Soy un animal!"); } }</pre>	<pre>package cadenadellamadas; public class Ave extends Animal { public Ave() { System.out.println("Sin uso!"); } public Ave(int n) { System.out.println("Soy " + (n==1? "un" : "") + " ave!"); } }</pre>
<pre>package cadenadellamadas; public class Paloma extends Ave { public Paloma() { super(1); System.out.println("Soy una paloma!"); } }</pre>	



Cuando un método de una subclase sobrescribe un método de una superclase, se puede acceder al método de la superclase desde la subclase, si se antepone al nombre del método la palabra clave **super** y un separador punto (.), como se ve a continuación:

```
Empleado.java
package empresa;
public class Empleado {
    private double sueldoBasico;
    public Empleado() {
        sueldoBasico = 10000;
    }
    public double obtenerSueldo() {
        return sueldoBasico;
    }
}
```

```
Gerente.java
package empresa;
public class Gerente extends Empleado{
    private double adicional;
    public Gerente() {
        adicional = 5000;
    }
    public double obtenerSueldo() {
        return super.obtenerSueldo() + adicional;
    }
}
```

```
Main.java
package empresa;
public class Main {
    public static void main(String[] args) {
        Empleado cadete = new Empleado();
        System.out.println("Cadete: " + cadete.obtenerSueldo ());
        Gerente gerente = new Gerente();
        System.out.println("Gerente: " + gerente.obtenerSueldo ());
    }
}
```

A veces, todo o parte del comportamiento de ciertos objetos es demasiado específico como para implementarlo en una superclase. En tales casos, el método se debe declarar anteponiéndole la palabra reservada **abstract** en la superclase e implementarse en cada subclase. Así, por contener al menos un método abstracto, también la superclase debe declararse anteponiéndole la palabra reservada **abstract**. En consecuencia, ya no será posible crear objetos de esa clase, porque las clases abstractas no son instanciables.

El siguiente ejemplo muestra cómo en la clase abstracta **PolígonoRegular** se declara el método abstracto **calcularArea**, el cual se implementa luego en las clases **TrianguloEquilátero** y **Cuadrado**, dos subclases de **PolígonoRegular**.

```
PolígonoRegular.java
package poligonosRegulares;
public abstract class PolígonoRegular {
    private double lado;
    public PolígonoRegular(double lado) {
        this.lado = lado;
    }
    public double getLado() {
        return lado;
    }
    public abstract double calcularArea();
}
```



TrianguloEquilatero.java

```
package poligonosRegulares;

public class TrianguloEquilatero extends PoligonoRegular {

    public TrianguloEquilatero(double lado) {
        super(lado);
    }

    public double calcularArea() {
        return getLado() * getLado() * Math.sqrt(3) / 4;
    }
}
```

Cuadrado.java

```
package poligonosRegulares;

public class Cuadrado extends PoligonoRegular {

    public Cuadrado(double lado) {
        super(lado);
    }

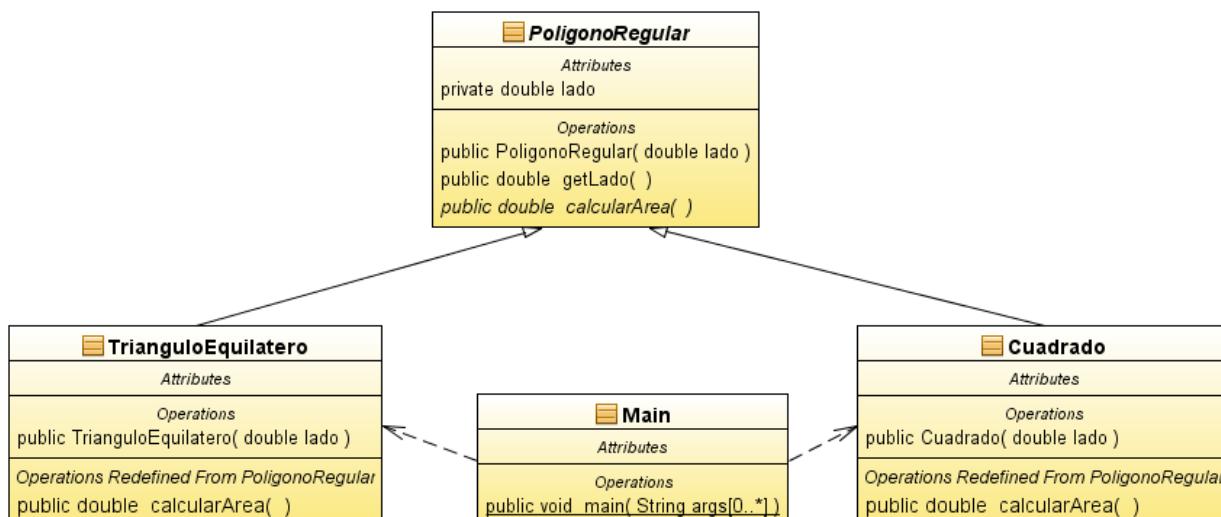
    public double calcularArea() {
        return getLado() * getLado();
    }
}
```

Main.java

```
package poligonosRegulares;

public class Main {
    public static void main(String[] args) {
        TrianguloEquilatero tri = new TrianguloEquilatero(5);
        Cuadrado cua = new Cuadrado(5);
        System.out.println ("Triangulo: " + tri.calcularArea());
        System.out.println ("Cuadrado : " + cua.calcularArea());
    }
}
```

El siguiente diagrama UML corresponde al código anterior. Observe cómo los nombres de las **clases y métodos abstractos** aparecen escritos en



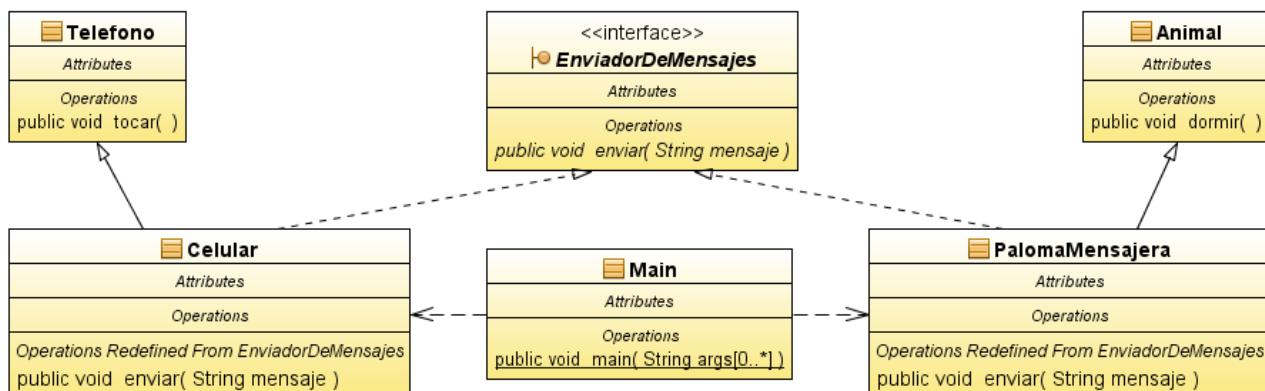


La herencia simple soluciona de raíz el problema básico de la herencia múltiple: el *problema del diamante* (llamado así por la forma que tiene el diagrama de clases donde ocurre este problema). Supongamos que la clase **Vehículo** define un método **avanzar**. Las clases **Auto** y **Lancha** extienden **Vehículo**: un auto es un vehículo y una lancha también es un vehículo, por lo tanto ambas clases heredan el método **avanzar**. Ahora bien, dado que un vehículo anfibio es un auto y también es una lancha, podría declararse mediante herencia múltiple que **VehiculoAnfibio** extienda **Auto** y **Lancha**. ¿Qué comportamiento debería tener entonces un vehículo anfibio: el método **avanzar** que hereda de **Auto** o el que hereda de **Lancha**?

Sin embargo, la inexistencia de la herencia múltiple podría resultar limitante para ciertos diseños. Por ejemplo, ¿qué tienen en común un celular y una paloma mensajera? La respuesta es que ambos permiten enviar mensajes, por eso podrían modelarse mediante **Celular** y **PalomaMensajera**, dos subclases de la clase abstracta **EnviadorDeMensajes** que deberían implementar el método abstracto **enviar** declarado en esa clase. Sin embargo, este diseño es muy limitado, ya que la herencia simple impide que **Celular** herede también de **Telefono** o que **PalomaMensajera** herede de **Animal**, dos clases llenas de funcionalidad.

La solución consiste en hacer que **EnviadorDeMensajes** sea una interfaz (*interface*) implementada por **Celular** (que extiende **Telefono**) y por **PalomaMensajera** (que extiende **Animal**). Como los métodos declarados en las interfaces (por definición) son abstractos, no ocurre el *problema del diamante* porque, en las **realizaciones**, no se heredan implementaciones de métodos, sino la responsabilidad de implementarlos.

El siguiente diagrama UML muestra la interfaz **EnviadorDeMensajes** y su relación con las clases **Celular** y **PalomaMensajera**.



El código de la interfaz **EnviadorDeMensajes** es el siguiente:

```
EnviadorDeMensajes.java
package comunicaciones;

public interface EnviadorDeMensajes {
    public void enviar(String mensaje);
}
```



Las demás clases del proyecto se muestran a continuación:

Celular.java

```
package comunicaciones;

public class Celular extends Telefono implements EnviadorDeMensajes {

    public void enviar(String mensaje) {
        System.out.println("RING RING: " + mensaje);
    }
}
```

Telefono.java

```
package comunicaciones;

public class Telefono {

    public void tocar() {
        System.out.println("RING RING RING");
    }
}
```

PalomaMensajera.java

```
package comunicaciones;

public class PalomaMensajera extends Animal implements EnviadorDeMensajes {

    public void enviar(String mensaje) {
        System.out.println("Llega volando: " + mensaje);
    }
}
```

Animal.java

```
package comunicaciones;

public class Animal {

    public void dormir() {
        System.out.println("ZZZ...");
    }
}
```

Main.java

```
package comunicaciones;

public class Main {

    public static void main(String[] args) {
        Celular c = new Celular();
        c.tocar();
        c.enviar("Hola");
        PalomaMensajera p = new PalomaMensajera();
        p.enviar("Hola");
        p.dormir();
    }
}
```

Como se ve en Main, el objeto c tiene comportamiento de Telefono y de EnviadorDeMensajes, ya que responde a los mensajes tocar y enviar. De la misma manera, p es EnviadorDeMensajes y Animal, por eso responde a enviar y a dormir.



Unidad 5

Encapsulamiento

El término *encapsulamiento* suele utilizarse para referirse a cosas diferentes:

1. la agrupación de estado y comportamiento en una unidad conceptual (la clase);
2. la aplicación de mecanismos de restricción de acceso a los atributos y métodos;
3. el ocultamiento de información como principio de diseño.

De acuerdo al primer significado, el **encapsulamiento** refleja el hecho de que los objetos son entidades que agrupan los atributos y los métodos que operan sobre ellos. Literalmente, un objeto es una cápsula cuyo interior (su implementación) sólo se debería poder utilizar a través de puntos de contacto con el exterior (su interfaz) bien definidos.

El segundo significado se refiere al control de la **accesibilidad** o visibilidad de los métodos y atributos desde el exterior, mediante palabras claves (frecuentemente denominadas *especificadores de acceso*), que definen qué miembros forman parte de la interfaz y cuáles son parte de la implementación.

Finalmente, el **ocultamiento de información** es un principio que sugiere, como condición necesaria para obtener un buen diseño, que todos los detalles de la implementación deben ser invisibles desde el exterior.

Cuando se menciona el término "encapsulamiento" sin más, generalmente se está haciendo referencia a las tres definiciones como un todo.

En Java, la declaración de una clase de objetos garantiza la primera acepción de encapsulamiento. Sin embargo, esto no garantiza de ningún modo el ocultamiento de información. El ocultamiento de información se consigue haciendo que los atributos sean privados y que el acceso a los mismos se lleve a cabo mediante métodos públicos (*getters* y *setters*).

Java ofrece cuatro niveles de accesibilidad para los miembros de una clase: visibilidad de paquete (por defecto) y la que se obtiene al anteponerle a cada declaración alguno de los especificadores de acceso **public**, **private** o **protected**.

¿Puede accederse desde ↓ a un miembro →?	public	protected	sin especif.	private
Misma clase				
Otra clase del mismo paquete				
Una subclase del mismo paquete				
Otra clase de otro paquete				
Una subclase de otro paquete				

Complete la tabla anterior experimentando con las clases siguientes:



```
package accesos;  
public class MiClase {  
    int atributoMiClase;  
    public MiClase() {  
        atributoMiClase = 10;  
    }  
    public void metodoMiClase() {  
        atributoMiClase++;  
    }  
    public void mostrarMiAtributo() {  
        System.out.println(atributoMiClase);  
    }  
}
```

```
package accesos;  
public class UnaSubclase extends MiClase {  
    public void metodoSubclase(MiClase o) {  
        o.atributoMiClase++;  
    }  
    public void mostrarMiAtributo() {  
        System.out.println(atributoMiClase);  
    }  
}
```

```
package accesos;  
import otroPaquete.ClaseImportada;  
import otroPaquete.SubclaseImportada;  
public class Main {  
    public static void main(String[] args) {  
        MiClase objMiClase = new MiClase();  
        OtraClase objOtraClase = new OtraClase();  
        UnaSubclase objUnaSubclase = new UnaSubclase();  
        ClaseImportada objClaseImportada = new ClaseImportada();  
        SubclaseImportada objSubclaseImportada = new SubclaseImportada();  
        System.out.print("Valor original del atributo: ");  
        objMiClase.mostrarMiAtributo();  
        objMiClase.metodoMiClase();  
        System.out.print("Valor del atributo modificado por el propio objeto: ");  
        objMiClase.mostrarMiAtributo();  
        objOtraClase.metodoOtraClase(objMiClase);  
        System.out.print("Valor del atributo modificado por un objeto de otra clase: ");  
        objMiClase.mostrarMiAtributo();  
        objUnaSubclase.metodoSubclase(objMiClase);  
        System.out.print("Valor del atributo modificado por un objeto de una subclase: ");  
        objMiClase.mostrarMiAtributo();  
        System.out.print("** Valor del atributo en el objeto de esa subclase: ");  
        objUnaSubclase.mostrarMiAtributo();  
        objClaseImportada.metodoClaseImportada(objMiClase);  
        System.out.print("Valor de atributo modificado por objeto de una clase importada: ");  
        objMiClase.mostrarMiAtributo();  
        objSubclaseImportada.metodoSubclaseImportada(objMiClase);  
        System.out.print("Valor de atributo modificado por objeto de subclase importada:");  
        objMiClase.mostrarMiAtributo();  
        System.out.print("** Valor del atributo en el objeto de esa subclase importada: ");  
        objSubclaseImportada.mostrarMiAtributo();  
    }  
}
```

```
package otroPaquete;  
import accesos.MiClase;  
public class ClaseImportada {  
    public void metodoClaseImportada(MiClase o) {  
        o.atributoMiClase++;  
    }  
}  
  
package otroPaquete;  
import accesos.MiClase;  
public class SubclaseImportada extends MiClase {  
    public void metodoSubclaseImportada(MiClase o){  
        o.atributoMiClase++;  
    }  
    public void mostrarMiAtributo() {  
        System.out.println(atributoMiClase);  
    }  
}  
  
package accesos;  
public class OtraClase {  
    public void metodoOtraClase(MiClase o) {  
        o.atributoMiClase++;  
    }  
}
```

-30-



Unidad 6

Polimorfismo

El cuarto y último de los pilares de la programación orientada a objetos, junto con la *abstracción* (pág. 4), el *encapsulamiento* (pág. 29) y la *herencia* (pág. 21), es el **polimorfismo**. En griego, πολλούς μορφή significa “muchas formas”. Es por ello que, en programación, se denomina así a la capacidad de contener valores de distintos tipos (en el caso de las *variables polimórficas*) o a la capacidad de recibir argumentos de diferentes tipos (en el caso de los *métodos polimórficos*).

Según cómo sea el conjunto de los tipos posibles, Cardelli y Wegner¹ clasifican el polimorfismo en dos categorías: **polimorfismo ad-hoc**² (funciona con un número limitado y conocido de tipos, no necesariamente relacionados entre sí) y **polimorfismo universal** (funciona con un número prácticamente infinito de tipos relacionados entre sí). En cada categoría, a su vez, es posible encontrar dos variedades de polimorfismo: como variantes del polimorfismo ad-hoc pueden clasificarse el *polimorfismo por sobrecarga* y el *polimorfismo por coerción*, mientras que el *polimorfismo paramétrico* y el *polimorfismo por inclusión* son variantes del polimorfismo universal.

Polimorfismo por sobrecarga

Dentro de una misma clase, es posible escribir dos o más métodos con el mismo nombre pero diferente firma (*signature*), como se explicó en la pág. 18. El resultado es un polimorfismo *aparente*, ya que no se trata de un método que puede recibir muchos tipos de argumentos, sino que hay varios métodos con el mismo nombre, y durante el proceso de *compilación* se elige cuál usar según el o los argumento(s) pasado(s). En el siguiente ejemplo, el método `unir` está sobrecargado para recibir dos o dos

```
package uniones;
public class SobrecargaTest {
    public static void main(String[] args) {
        Unidor u = new Unidor();
        System.out.println(u.unir("123", "45"));
        System.out.println(u.unir(123, 45));
    }
}
```

```
package uniones;
public class Unidor {
    public String unir(String a, String b) {
        return a.concat(b);
    }
    public int unir(int a, int b) {
        return (int) (a * Math.pow(10, Math.ceil(Math.log10(b))) + b);
    }
}
```

¹ Cardelli, L. & Wegner, P. “On Understanding Types, Data Abstraction, and Polymorphism”. En: *Computing Surveys* (Diciembre, 1985). Vol. 17, n. 4, p. 471

² *Ad hoc* es una locución latina que significa literalmente “para esto”. Se usa para referirse a algo específico que es adecuado sólo para un determinado fin o en una determinada situación.



Polimorfismo por coerción

Aplicar *coerción* sobre alguien significa forzar su voluntad o su conducta. En programación, significa forzar a que un dato de un tipo sea tratado como si fuera de otro tipo. Por ejemplo, *coerción* es la operación realizada para convertir (de manera implícita) un argumento al tipo esperado por un método, es decir, al tipo con que fue declarado el parámetro correspondiente. En este caso, también se trata de un polimorfismo *aparente*, ya que la conversión que ocurre no cambia el hecho de que el método en realidad sólo trabaja con un único tipo de dato. En el siguiente ejemplo, el método **duplicar** recibe primero un y luego un

```
package coercion;
public class CoercionTest {
    public static void main(String[] args) {
        double x = 12.5;
        int n = 12;
        Duplicador d = new Duplicador();
        System.out.println(d.duplicar(x));
        System.out.println(d.duplicar(n));
    }
}
```

```
package coercion;
public class Duplicador {
    public double duplicar(double x) {
        return 2 * x;
    }
}
```

Polimorfismo paramétrico

Cuando se escribe *código genérico*, es decir, sin mencionar ningún tipo de datos específico, para que pueda ser usado con datos que serán tratados de manera idéntica independientemente de su tipo, se está aprovechando el *polimorfismo paramétrico*. En Java, a esta variante de polimorfismo se la conoce como *Generics*. En el siguiente ejemplo, la clase **PilaGenerica** puede usarse para instanciar pilas específicas para cualquier tipo de objeto. Para mostrar esto, dentro del método **main** se instancian **pInt** (una pila de) y **pStr** (una pila de).

```
package generics;
public class GenericsTest {
    public static void main(String[] args) {
        PilaGenerica<Integer> pInt = new PilaGenerica<>();
        pInt.push(10); pInt.push(20); pInt.push(30);
        System.out.println(pInt.pop());
        System.out.println(pInt.pop());
        System.out.println(pInt.pop());
        PilaGenerica<String> pStr = new PilaGenerica<>();
        pStr.push("Arbol"); pStr.push("Casa"); pStr.push("Auto");
        System.out.println(pStr.pop());
        System.out.println(pStr.pop());
        System.out.println(pStr.pop());
    }
}
```



```
package generics;
public class PilaGenerica<T> {
    private T elementoTope;
    private PilaGenerica<T> restoPila;
    public PilaGenerica() {
        elementoTope = null;
        restoPila = null;
    }
    public void push(T elem) {
        PilaGenerica<T> aux = new PilaGenerica<>();
        aux.elementoTope = elementoTope;
        aux.restoPila = restoPila;
        restoPila = aux;
        elementoTope = elem;
    }
    public T pop() {
        T tope = elementoTope;
        if (restoPila != null) {
            elementoTope = restoPila.elementoTope;
            restoPila = restoPila.restoPila;
        }
        return tope;
    }
}
```

Polimorfismo por inclusión

El *polimorfismo por inclusión*, también conocido como *polimorfismo por herencia* o *polimorfismo de subclases* (o *de subtipos*), es la variante más común de polimorfismo encontrada en la Programación Orientada a Objetos y, por tal motivo, la mayoría de las veces se lo denomina directamente *polimorfismo* a secas. Se basa en el hecho de que las superclases *incluyen* a todas las instancias de sus subclases o, dicho de otra forma, todos los objetos de una subclase son también objetos de las superclases de ésta. Por ello, aunque se declare un atributo, una variable local, un parámetro o el contenido de una colección como siendo de una superclase, en realidad puede referirse a cualquier instancia de esa clase o de alguna sus subclases. Los objetos no pierden sus características (estado y comportamiento) al ser referenciados de esta manera, por lo que es posible invocar los métodos que éstos hayan sobrescrito (*method overriding*), y su comportamiento será el esperado. Veamos el siguiente ejemplo:

```
package poligonosRegulares;
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<PoligonoRegular> c = new ArrayList<>();
        c.add(new TrianguloEquilátero(5));
        c.add(new Cuadrado(5));
        c.add(new PentágonoRegular(5));
        for (int i = 0; i < c.size(); i++) {
            PoligonoRegular p = c.get(i);
            System.out.println(p.getNombre() + ": " + Math.floor(p.calcularArea()*100)/100);
        }
    }
}
```



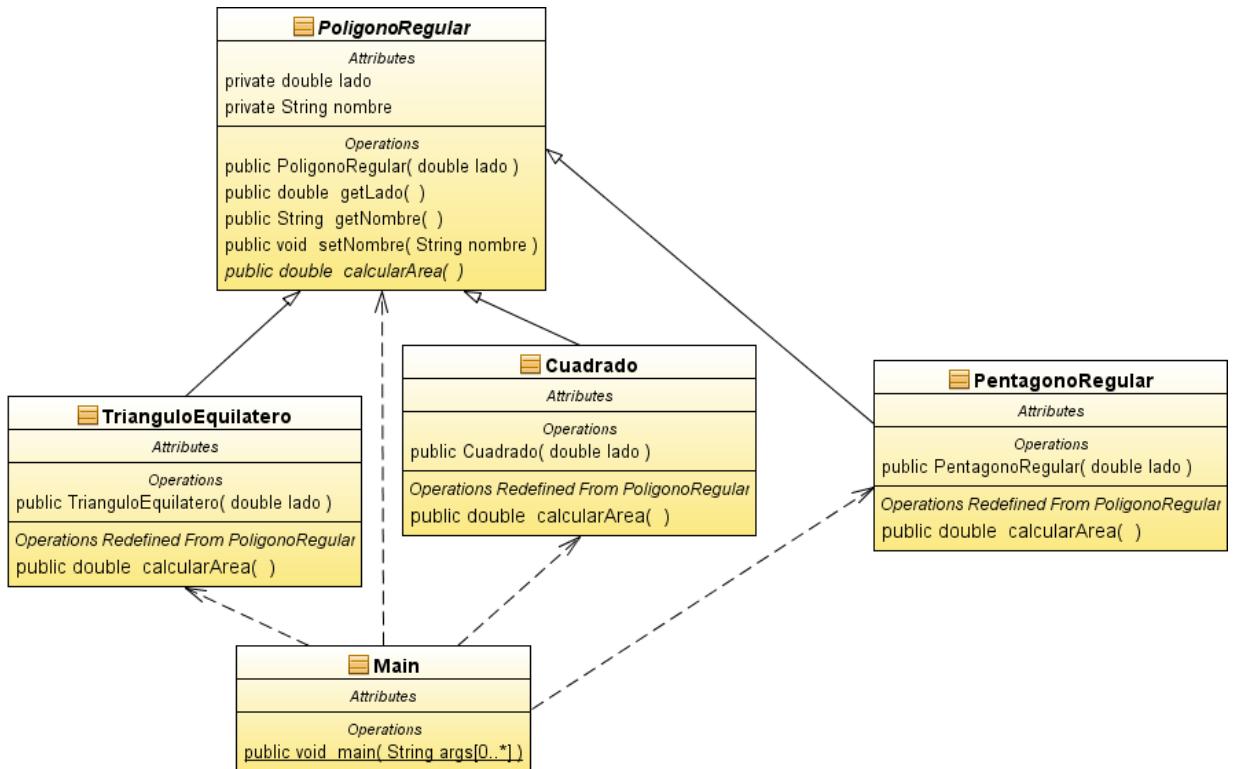
```
package poligonosRegulares;
public abstract class PoligonoRegular {
    private double lado;
    private String nombre;
    public PoligonoRegular(double lado) {
        this.lado = lado;
    }
    public double getLado() {
        return lado;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public abstract double calcularArea();
}
```

```
package poligonosRegulares;
public class TrianguloEquilatero extends PoligonoRegular {
    public TrianguloEquilatero(double lado) {
        super(lado);
        setNombre("Triangulo equilatero");
    }
    public double calcularArea() {
        return getLado() * getLado() * Math.sqrt(3) / 4;
    }
}
```

```
package poligonosRegulares;
public class Cuadrado extends PoligonoRegular {
    public Cuadrado(double lado) {
        super(lado);
        setNombre("Cuadrado");
    }
    public double calcularArea() {
        return getLado() * getLado();
    }
}
```

```
package poligonosRegulares;
public class PentagonoRegular extends PoligonoRegular {
    public PentagonoRegular(double lado) {
        super(lado);
        setNombre("Pentagono regular");
    }
    public double calcularArea() {
        return 5 * getLado() * getLado() / (4 * Math.tan(Math.PI / 5.0));
    }
}
```

El diagrama UML correspondiente a las clases del ejemplo anterior es el siguiente:

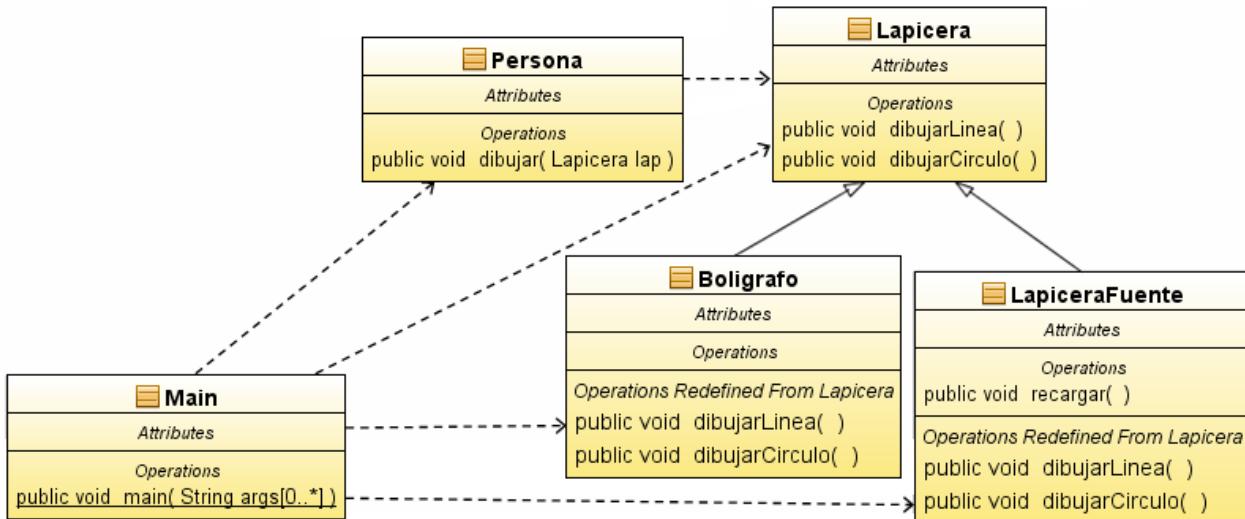


En la clase abstracta **PoligonoRegular** se declaran los atributos privados `lado` (un `double`) y `nombre` (un `String`) que caracterizan a cualquier polígono regular. El valor del `lado` se asigna en el constructor, y el valor del nombre se asigna mediante el método `setNombre`. Para obtener el valor de estos atributos privados, se proporcionan los métodos públicos `getLado` y `getNombre`. El método `calcularArea` se declara **abstract** en **PoligonoRegular**, y se redefine con fórmulas específicas en las subclases concretas **TrianguloEquilatero**, **Cuadrado** y **PentagonoRegular** mediante el mecanismo de sobrescritura de métodos (*method overriding*). En los constructores de estas subclases se le pasa el valor del `lado` al constructor de la superclase abstracta (con `super`) y se invoca el método `setNombre` (que se heredó) con el nombre del polígono que se está construyendo.

En el ejemplo anterior, el polimorfismo se utiliza varias veces en la clase **Main**. Primero, aparece un caso de *polimorfismo paramétrico* al declarar `c` como un `ArrayList<PoligonoRegular>`. Las clases de la biblioteca *Collections* (pág. 15) pueden hacer uso de esta variedad de polimorfismo porque son genéricas. Despues, se invoca tres veces el método `add` del objeto `c`, cada vez con un argumento que es una instancia de una clase diferente. Como `add` espera instancias de **PoligonoRegular** y recibe objetos que son instancias de sus subclases, este caso es de *polimorfismo por inclusión*. Otra ocurrencia de esta variedad de polimorfismo se da cuando se le asigna a la variable `p` un objeto de una subclase de **PoligonoRegular** proveniente de la colección `c` y, luego, se invocan los métodos `getNombre` y `calcularArea` del objeto refiriéndose a éste mediante la variable `p`.



Tal como habíamos anticipado en la pág. 27, modelar una jerarquía de clases basándose exclusivamente en las relaciones de herencia produce diseños poco flexibles. Observemos el siguiente ejemplo:



```
package polimorfismo;  
public class Main {  
    public static void main(String[] args) {  
        Lamicera lap = new Lamicera();  
        Boligrafo bol = new Boligrafo();  
        LamiceraFuente lapFuente = new LamiceraFuente();  
  
        Persona p = new Persona();  
  
        p.dibujar(lap);  
        p.dibujar(bol);  
        p.dibujar(lapFuente);  
    }  
}
```

```
package polimorfismo;  
public class Persona {  
    public void dibujar(Lamicera lap) {  
        lap.dibujarLinea();  
        lap.dibujarCirculo();  
    }  
}
```

```
package polimorfismo;  
public class Lamicera {  
    public void dibujarLinea() {  
        System.out.println("Se ha dibujado una linea con una LAPICERA");  
    }  
    public void dibujarCirculo() {  
        System.out.println("Se ha dibujado un circulo con una LAPICERA");  
    }  
}
```



```
package polimorfismo;

public class Boligrafo extends Lapicera {

    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con un BOLIGRAFO");
    }

    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con un BOLIGRAFO");
    }
}
```

```
package polimorfismo;

public class LapiceraFuente extends Lapicera {

    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con una LAPICERA FUENTE");
    }

    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con una LAPICERA FUENTE");
    }

    public void recargar() {
        System.out.println("Se ha recargado una LAPICERA FUENTE");
    }
}
```

Así como una persona es capaz, en el mundo real, de dibujar tanto con una lapicera en general como con una lapicera fuente o un bolígrafo en particular, en la clase `Main` el objeto `p` reacciona correctamente cuando le pasamos el mensaje `dibujar` con cualquier argumento que sea una `Lapicera` (`lap` es obviamente una instancia de `Lapicera`, y también `lapFuente`, que se refiere a una `LapiceraFuente`, y `bol`, que se refiere a un `Boligrafo`, pasan el test “*es-un(a)*” `Lapicera`).

OBSERVACIÓN

Al correr el ejemplo anterior, la salida por pantalla es:

.....
.....
.....
.....
.....
.....

Sabiendo que las lapiceras fuente son recargables (de hecho, la clase `LapiceraFuente` contiene un método denominado `recargar`), se podría requerir que, una vez que el objeto `p` haya terminado de dibujar usando la lapicera fuente a que se refiere `lapFuente`, ésta fuera recargada. El código a agregar en `main` sería el siguiente:

```
lapFuente.recargar();
```



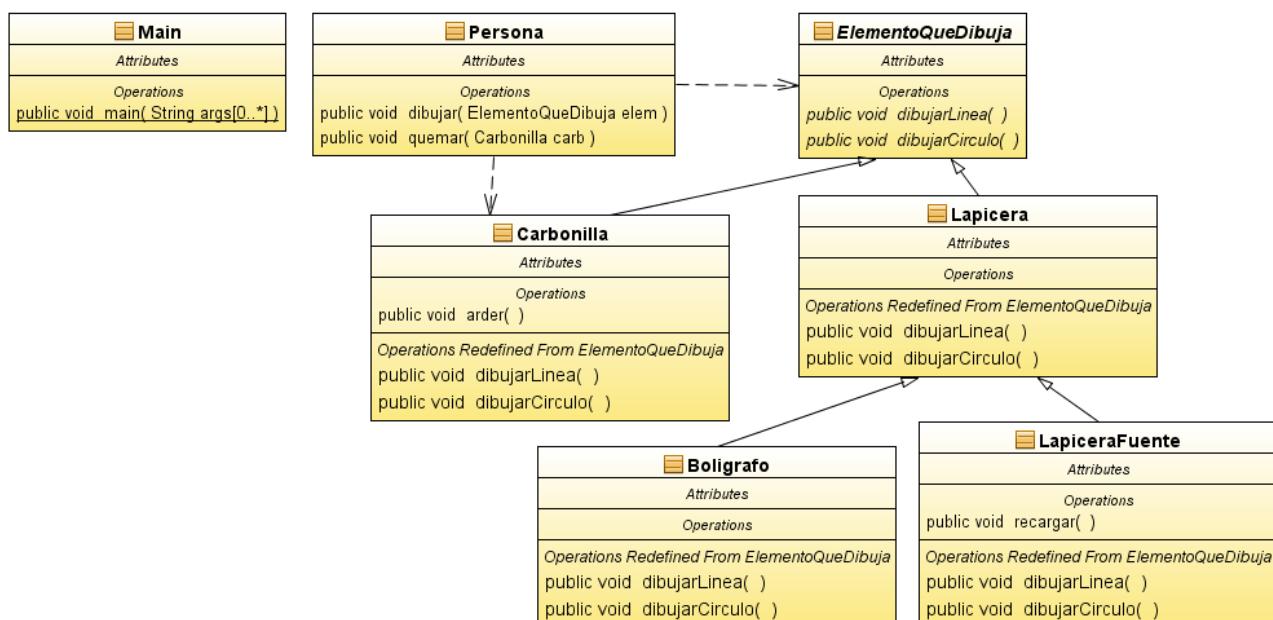
Sin embargo, el método `recargar` no se encuentra, y no es posible compilar el código anterior. Esto ocurre porque la variable `lapFuente` se refiere a una lapisera fuente, pero fue declarada como `Lapicera`, y la clase `Lapicera` no contiene un método `recargar`. La solución consiste en utilizar una variable `lapF` declarada como `LapiceraFuente` y hacer que se refiera al mismo objeto a que se refería `lapFuente`, para así poder invocar el método `recargar`. Para ello debe hacerse la siguiente conversión explícita (*cast*):

```
LapiceraFuente lapF = (LapiceraFuente) lapFuente;  
lapF.recargar();
```

OBSERVACIÓN

Al correr el ejemplo modificado, la nueva línea en la salida por pantalla es:

Recordemos que el objetivo del ejemplo que estamos viendo es mostrar cómo modelar una jerarquía de clases basándose exclusivamente en las relaciones de herencia produce diseños poco flexibles. Para ello, sigamos adelante e incorporemos en la jerarquía de clases la `Carbonilla` que, al igual que una `Lapicera`, sirve para dibujar, pero además puede `arder`, por ejemplo, cuando se la quema para asar comidas a la parrilla. Agreguemos en la clase `Persona` el método `quemar` para poder darle este segundo uso a la carbonilla. Como la carbonilla no es un tipo de lapisera y el método `dibujar` exige un argumento que sea una instancia de `Lapicera`, es necesario agregar a la jerarquía una nueva clase (`ElementoQueDibuja`) como superclase de `Carbonilla` y `Lapicera` y arreglar el método `dibujar` en la clase `Persona` para que acepte instancias de `ElementoQueDibuja`. Las variables polimórficas en `Main` ahora deben ser declaradas como `ElementoQueDibuja` en lugar de `Lapicera`. En consecuencia, debe cambiársele el tipo a la variable `carbo` al pasársela al método `quemar`. El nuevo diagrama de clases es el siguiente (se omitieron las dependencias que tiene `Main`):





Como puede verse, es necesario hacer muchos cambios para agregar una nueva clase en este diseño basado exclusivamente en herencia. El código modificado es el siguiente:

```
package polimorfismo;

public class Main {

    public static void main(String[] args) {
        ElementoQueDibuja lap = new Lapicera();
        ElementoQueDibuja bol = new Boligrafo();
        ElementoQueDibuja lapFuente = new LapiceraFuente();
        ElementoQueDibuja carbo = new Carbonilla();

        Persona p = new Persona();

        p.dibujar(lap);
        p.dibujar(bol);

        p.dibujar(lapFuente);
        LapiceraFuente lapF = (LapiceraFuente) lapFuente;
        lapF.recargar();

        p.dibujar(carbo);
        p.quemar((Carbonilla) carbo);
    }
}
```

```
package polimorfismo;
public class Persona {
    public void dibujar(ElementoQueDibuja elem) {
        elem.dibujarLinea();
        elem.dibujarCirculo();
    }
    public void quemar(Carbonilla carb) {
        carb.arder();
    }
}
```

```
package polimorfismo;
public abstract class ElementoQueDibuja {
    public abstract void dibujarLinea();
    public abstract void dibujarCirculo();
}
```

```
package polimorfismo;
public class Lapicera extends ElementoQueDibuja {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con una LAPICERA");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con una LAPICERA");
    }
}
```



```
package polimorfismo;
public class Boligrafo extends Lapicera {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con un BOLIGRAFO");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con un BOLIGRAFO");
    }
}
```

```
package polimorfismo;
public class LapiceraFuente extends Lapicera {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con una LAPICERA FUENTE");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con una LAPICERA FUENTE");
    }
    public void recargar() {
        System.out.println("Se ha recargado una LAPICERA FUENTE");
    }
}
```

```
package polimorfismo;
public class Carbonilla extends ElementoQueDibuja {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con CARBONILLA");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con CARBONILLA");
    }
    public void arder() {
        System.out.println("Se ha quemado CARBONILLA");
    }
}
```

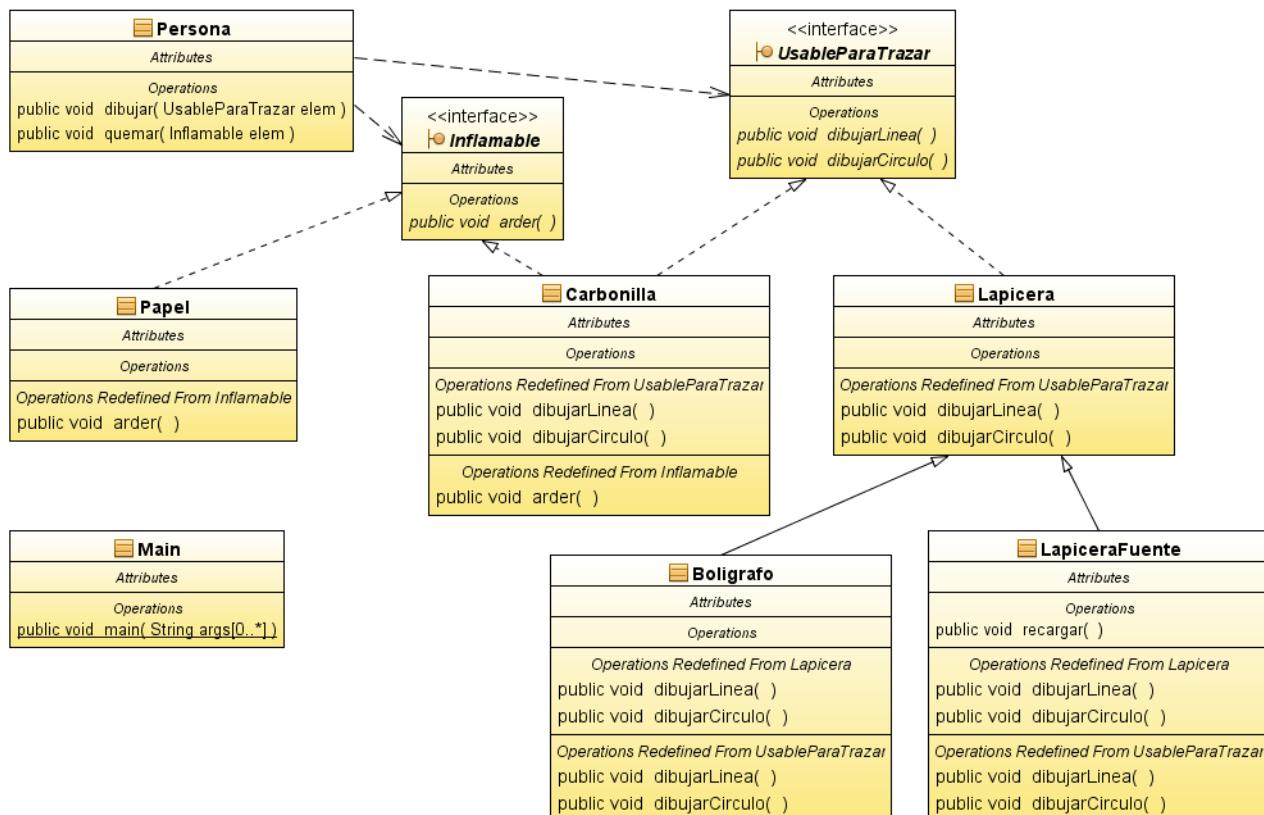
OBSERVACIÓN

Al correr el ejemplo anterior, la salida por pantalla es:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Con el modelo anterior, se consiguió implementar la posibilidad de que un objeto de la clase **Persona** dibuje usando lapiceras y carbonilla, incorporando en la jerarquía de clases una superclase que incluyera (para usar *polimorfismo por inclusión*) a las lapiceras y la carbonilla: **ElementoQueDibuja**. El método **dibujar** de la clase **Persona** debió modificarse para recibir argumentos de la nueva superclase. Supongamos que ahora quisiéramos agregar a la jerarquía de clases el **Papel** y que un objeto de la clase **Persona** tuviera que poder quemarlo. Evidentemente, el papel no es carbonilla, por lo que el método **quemar** de la clase **Persona** deberá modificarse. Una solución sería sobrecargarlo (*polimorfismo por sobrecarga*) para que recibiera un argumento de la clase **Papel** (recordemos que ya está implementado para recibir un argumento de la clase **Carbonilla**). Pero esto sería una solución *ad-hoc*. Agregar una nueva superclase **ElementoQueArde** y hacer que **quemar** reciba argumentos de esta clase serviría para **Papel**, que aún no tiene superclase, pero no para **Carbonilla**, porque ésta ya extiende **ElementoQueDibuja**.

La solución consiste en aprovechar el polimorfismo mediante **interfaces** en lugar de hacerlo mediante herencia. Las interfaces pueden usarse para encapsular un conjunto de métodos sin implementar (los cuales por defecto son **public** y **abstract**) y de valores constantes (los que por defecto son **public**, **static** y **final**). Recordemos que la relación entre la clase y la interfaz implementada se denomina **realización**. El diagrama de clases resultante es el siguiente (nuevamente se omitieron las dependencias que tiene **Main**):



Como puede observarse en el diagrama, tanto **dibujar** como **quemar** son métodos polimórficos de **Persona**, ya que aceptan cualquier argumento que implemente las interfaces **UsableParaTrazar** o **Inflamable**, respectivamente.



El código completo es el siguiente:

```
package polimorfismo;
public class Main {
    public static void main(String[] args) {
        UsableParaTrazar lap = new Lapicera();
        UsableParaTrazar bol = new Boligrafo();
        UsableParaTrazar lapFuente = new LapiceraFuente();
        UsableParaTrazar carbo = new Carbonilla();

        Persona p = new Persona();
        p.dibujar(lap);
        p.dibujar(bol);
        p.dibujar(lapFuente);
        LapiceraFuente lapF = (LapiceraFuente) lapFuente;
        lapF.recargar();
        p.dibujar(carbo);
        p.quemar((Inflamable) carbo);
        Inflamable papel = new Papel();
        p.quemar(papel);
    }
}
```

```
package polimorfismo;
public class Persona {
    public void dibujar(UsableParaTrazar elem) {
        elem.dibujarLinea();
        elem.dibujarCirculo();
    }
    public void quemar(Inflamable elem) {
        elem.arder();
    }
}
```

```
package polimorfismo;
public interface UsableParaTrazar {
    void dibujarLinea();
    void dibujarCirculo();
}
```

```
package polimorfismo;
public interface Inflamable {
    void arder();
}
```

```
package polimorfismo;
public class Papel implements Inflamable {
    @Override
    public void arder() {
        System.out.println("Se ha quemado PAPEL");
    }
}
```



```
package polimorfismo;
public class Lapicera implements UsableParaTrazar {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con una LAPICERA");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con una LAPICERA");
    }
}
```

```
package polimorfismo;
public class Boligrafo extends Lapicera {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con un BOLIGRAFO");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con un BOLIGRAFO");
    }
}
```

```
package polimorfismo;
public class LapiceraFuente extends Lapicera {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con una LAPICERA FUENTE");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con una LAPICERA FUENTE");
    }
    public void recargar() {
        System.out.println("Se ha recargado una LAPICERA FUENTE");
    }
}
```

```
package polimorfismo;
public class Carbonilla implements UsableParaTrazar, Inflamable {
    @Override
    public void dibujarLinea() {
        System.out.println("Se ha dibujado una linea con CARBONILLA");
    }
    @Override
    public void dibujarCirculo() {
        System.out.println("Se ha dibujado un circulo con CARBONILLA");
    }
    @Override
    public void arder() {
        System.out.println("Se ha quemado CARBONILLA");
    }
}
```



OBSERVACIÓN

Al correr el código completo, la nueva línea en la salida por pantalla es:

Como pudo observarse en el ejemplo anterior, el uso de interfaces para aprovechar el polimorfismo es una técnica muy útil. De hecho, el quinto de los **Cinco Principios para el Diseño de Clases** propuestos por Robert C. Martin¹ se refiere a esta cuestión. Aplicados en conjunto, estos principios, conocidos actualmente por sus iniciales en inglés como **Principios S.O.L.I.D.**, hacen más probable que un sistema sea fácil de mantener y ampliar con el tiempo. Son los siguientes²:

1. Principio de responsabilidad única

Cada clase debe tener un único motivo para cambiar. Por ello, en el siguiente ejemplo, la clase `LlamadaHaciaFijoCABA` viola este principio:

```
package solid;

public class LlamadaHaciaFijoCABA {

    public void efectuar(int numero) {
        if (validar(numero)) {
            System.out.println("Llamando al +54 11 " + numero + "...");
        } else {
            System.out.println("Error: " + numero + " no es un numero fijo valido...");
        }
    }

    public boolean validar(int numero) {
        return numero > 19999999 && numero < 70000000;
    }
}
```

```
package solid;

public class Main {

    public static void main(String[] args) {
        LlamadaHaciaFijoCABA llamada = new LlamadaHaciaFijoCABA();
        llamada.efectuar(43112700);
        llamada.efectuar(1530366486);
    }
}
```

Debido a que la clase mencionada tiene dos responsabilidades (validar el número de teléfono fijo recibido y efectuar la llamada), podría haber más de un motivo para realizar cambios en la clase (por ejemplo, que cambie la forma de efectuar la llamada y que cambie el rango de números válidos).

¹ Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall

² Los principios S.O.L.I.D. en inglés son:

- 1) The Single Responsibility Principle: *A class should have one, and only one, reason to change.*
- 2) The Open Closed Principle: *You should be able to extend a classes behavior, without modifying it.*
- 3) The Liskov Substitution Principle: *Derived classes must be substitutable for their base classes.*
- 4) The Interface Segregation Principle: *Make fine grained interfaces that are client specific.*
- 5) The Dependency Inversion Principle: *Depend on abstractions, not on concretions.*



Para corregir el código anterior deben encapsularse las responsabilidades en clases separadas:

```
package solid;
public class LlamadaHaciaFijoCABA {
    public void efectuar(int numero) {
        ServicioDeValidacion v = new ServicioDeValidacion();
        if (v.validar(numero)) {
            System.out.println("Llamando al +54 11 " + numero + "...");
        } else {
            System.out.println("Error: " + numero + " no es un numero fijo valido...");
        }
    }
}
```

```
package solid;
public class ServicioDeValidacion {
    public boolean validar(int numero) {
        return numero > 19999999 && numero < 70000000;
    }
}
```

Ahora que sí se cumple el *Principio de Responsabilidad Única*, el código resultante es más legible y fácil de mantener. Además, al estar separadas, las responsabilidades pueden ser reutilizadas sin dificultad.

2. Principio abierto/cerrado

El comportamiento de una clase debe estar abierto a la extensión, pero cerrado a la modificación (se debe poder extender sin modificar el código existente). Por ello, en el siguiente ejemplo, la clase `CalculadoraDeAreas` viola este principio:

```
package solid;
import java.util.ArrayList;
public class CalculadoraDeAreas {
    private ArrayList c;
    public CalculadoraDeAreas(ArrayList c) {
        this.c = c;
    }
    public void calcular() {
        for (int i = 0; i < c.size(); i++) {
            Object p = c.get(i);
            if (p instanceof TrianguloEquilátero) {
                TrianguloEquilátero t = (TrianguloEquilátero) p;
                System.out.println(t.getNombre()+": "+Math.floor(t.calcular()*100)/100);
            } else if (p instanceof Cuadrado) {
                Cuadrado q = (Cuadrado) p;
                System.out.println(q.getNombre()+": "+Math.floor(q.calcularArea()*100)/100);
            } else if (p instanceof PentágonoRegular) {
                PentágonoRegular z = (PentágonoRegular) p;
                System.out.println(z.getNombre()+": "+Math.floor(z.calcularSup()*100)/100);
            }
        }
    }
}
```



```
package solid;
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList c = new ArrayList();
        c.add(new TrianguloEquilatero(5));
        c.add(new Cuadrado(5));
        c.add(new PentagonoRegular(5));
        CalculadoraDeAreas calcu = new CalculadoraDeAreas(c);
        calcu.calcular();
    }
}
```

```
package solid;
public class TrianguloEquilatero {
    private double lado;
    private String nombre;
    public TrianguloEquilatero(double lado) {
        this.lado = lado;
        nombre = "Triangulo equilatero";
    }
    public String getNombre() {
        return nombre;
    }
    public double calcular() {
        return lado * lado * Math.sqrt(3) / 4;
    }
}
```

```
package solid;
public class Cuadrado {
    private double lado;
    private String nombre;
    public Cuadrado(double lado) {
        this.lado = lado;
        nombre = "Cuadrado";
    }
    public String getNombre() {
        return nombre;
    }
    public double calcularArea() {
        return lado * lado;
    }
}
```

```
package solid;
public class PentagonoRegular {
    private double lado;
    private String nombre;
    public PentagonoRegular(double lado) {
        this.lado = lado;
        nombre = "Pentagono regular";
    }
    public String getNombre() {
        return nombre;
    }
    public double calcularSup() {
        return 5 * lado * lado / (4 * Math.tan(Math.PI / 5.0));
    }
}
```



En el ejemplo anterior ni siquiera se utiliza la herencia: las clases `TrianguloEquilatero`, `Cuadrado` y `PentagonoRegular` son independientes y utilizan métodos completamente diferentes (incluso en el nombre) para el cálculo del área. En la clase `CalculadoraDeAreas` se recorre `c`, un `ArrayList` recibido desde `Main` con objetos que podrían ser de cualquier tipo y, si el objeto contenido en determinada posición es una instancia de `TrianguloEquilatero`, se realiza una conversión explícita de tipo (*type cast*) para poder usar sus métodos. De forma análoga se procede si el objeto es una instancia de `Cuadrado` o de `PentagonoRegular`. Para que la clase `CalculadoraDeAreas` funcione adecuadamente si se encuentra en `c` un objeto que es una instancia de `HexagonoRegular` (una nueva clase), es necesario modificar su código, por lo cual no cumple con el principio que estamos viendo.

Para corregir el código anterior debe crearse una jerarquía de clases como la que se mostró en la pág. 35. Recordemos que es posible aprovechar el polimorfismo sobrescribiendo en todas las subclases concretas el método `calcularArea` de la clase abstracta `PoligonoRegular`. El código de las clases que aparecen en la pág. 34 más el código que se muestra a continuación constituyen el ejemplo corregido completo.

```
package solid;
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<PoligonoRegular> c = new ArrayList<>();
        c.add(new TrianguloEquilatero(5));
        c.add(new Cuadrado(5));
        c.add(new PentagonoRegular(5));
        CalculadoraDeAreas calcu = new CalculadoraDeAreas(c);
        calcu.calcular();
    }
}
```

```
package solid;
import java.util.ArrayList;
public class CalculadoraDeAreas {
    private ArrayList<PoligonoRegular> c;
    public CalculadoraDeAreas(ArrayList<PoligonoRegular> c) {
        this.c = c;
    }
    public void calcular() {
        for (int i = 0; i < c.size(); i++) {
            PoligonoRegular p = c.get(i);
            System.out.println(p.getNombre()+" "+Math.floor(p.calcularArea()*100)/100);
        }
    }
}
```

Esta versión de la clase `CalculadoraDeAreas` sí cumple con el *Principio abierto/cerrado*, ya que, gracias al polimorfismo, es capaz de calcular el área de cualquier instancia de `PoligonoRegular` que se encuentre en `c`. Por eso, si posteriormente se agregan en `c` instancias de `HexagonoRegular` (una nueva subclase de `PoligonoRegular`), no será necesario modificar el código de `CalculadoraDeAreas`.



3. Principio de sustitución de Liskov

Los objetos de las subclases deben poder sustituir a las instancias de las superclases sin alterar el correcto funcionamiento del programa. Por ello, en el siguiente ejemplo, la clase Winamp viola este principio:

```
package solid;

public class Main {

    public static void main(String[] args) {
        Reproductor r;
        r = new Reproductor();
        r.reproducirAudio();
        r.reproducirVideo();

        r = new VLC();
        r.reproducirAudio();
        r.reproducirVideo();

        r = new MediaPlayerClassic();
        r.reproducirAudio();
        r.reproducirVideo();

        r = new Winamp();
        r.reproducirAudio();
        r.reproducirVideo();
    }
}
```

```
package solid;

public class Reproductor {

    public void reproducirAudio() {
        System.out.println("Reproduciendo audio...");
    }

    public void reproducirVideo() {
        System.out.println("Reproduciendo video...");
    }
}
```

```
package solid;

public class MediaPlayerClassic extends Reproductor { }
```

```
package solid;

public class VLC extends Reproductor { }
```

```
package solid;

public class Winamp extends Reproductor {

    @Override
    public void reproducirVideo() {
        throw new RuntimeException("Winamp no puede reproducir video...");
    }
}
```

Básicamente, el problema en este caso es que la subclase tiene menos funcionalidad que la superclase, algo que, según este principio, nunca debería ocurrir.



Una manera de corregir el código anterior es reconocer que hay dos clases de reproductores: los que reproducen audio y video (como *Media Player Classic* y *VLC*) y los que sólo reproducen audio (como *Winamp*). Así, el código resultante es el siguiente:

```
package solid;
public class Main {
    public static void main(String[] args) {
        ReproductorDeAudioVideo r;
        r = new ReproductorDeAudioVideo();
        r.reproducirAudio();
        r.reproducirVideo();
        r = new VLC();
        r.reproducirAudio();
        r.reproducirVideo();
        r = new MediaPlayerClassic();
        r.reproducirAudio();
        r.reproducirVideo();
        ReproductorDeAudio a = new ReproductorDeAudio();
        a.reproducirAudio();
        a = new Winamp();
        a.reproducirAudio();
    //      a.reproducirVideo(); // Esta línea directamente no compila
    }
}
```

```
package solid;
public class ReproductorDeAudio {
    public void reproducirAudio() {
        System.out.println("Reproduciendo audio...");
    }
}
```

```
package solid;
public class ReproductorDeAudioVideo {
    public void reproducirVideo() {
        System.out.println("Reproducindo video...");
    }
    public void reproducirAudio() {
        System.out.println("Reproduciendo audio...");
    }
}
```

```
package solid;
class MediaPlayerClassic extends ReproductorDeAudioVideo {
```

```
package solid;
public class VLC extends ReproductorDeAudioVideo {
```

```
package solid;
public class Winamp extends ReproductorDeAudio {
```

Resumiendo, este principio establece que todas las subclases deben tener el comportamiento que los clientes esperan de la superclase. Por ello, como se afirma en un afiche muy conocido: “*Si tiene el aspecto de un pato y se oye como un pato, pero usa pilas, probablemente considerarlo un pato sea una abstracción equivocada.*”



4. Principio de segregación de interfaces

Deben crearse pequeñas interfaces específicas para los clientes en lugar de una general. De este modo, quienes implementen una interfaz no se verán forzados a implementar partes (métodos o propiedades) que no les sean útiles. Por ello, en el siguiente ejemplo, la interfaz `EmpleableParaTrabajar` viola este principio:

```
package solid;

public interface EmpleableParaTrabajar {
    void trabajar();
    void comer();
}
```

```
package solid;

public class Main {
    public static void main(String[] args) {
        EmpleableParaTrabajar trabajadorDiurno = new TrabajadorDiurno();
        trabajadorDiurno.trabajar();
        trabajadorDiurno.comer();

        EmpleableParaTrabajar trabajadorNocturno = new TrabajadorNocturno();
        trabajadorNocturno.trabajar();
        trabajadorNocturno.comer();
    }
}
```

```
package solid;

public class TrabajadorDiurno implements EmpleableParaTrabajar {
    @Override
    public void trabajar() {
        System.out.println("Trabaja de dia...");
    }

    @Override
    public void comer() {
        System.out.println("Hace una pausa para almorzar...");
    }
}
```

```
package solid;

public class TrabajadorNocturno implements EmpleableParaTrabajar {
    @Override
    public void trabajar() {
        System.out.println("Trabaja de noche...");
    }

    @Override
    public void comer() {
        System.out.println("Hace una pausa para cenar...");
    }
}
```

¿Qué ocurriría si se agregara una nueva clase `Robot`? Si ésta implementara `EmpleableParaTrabajar`, estaría obligada a implementar el método `comer`, lo cual no tiene ningún sentido.



Para corregir el código anterior, es necesario dividir `EmpleableParaTrabajar`, haciendo que `comer` sea un método de la interfaz `Alimentable`. De forma análoga, si los robots debieran recargarse, el método `recargar` debería declararse en una interfaz nueva (por ejemplo, `Recargable`). El código modificado quedaría así:

```
package solid;
public interface EmpleableParaTrabajar {
    void trabajar();
}
```

```
package solid;
public interface Alimentable {
    void comer();
}
```

```
package solid;
public interface EmpleableHumano extends EmpleableParaTrabajar, Alimentable {
}
```

```
package solid;
public class Main {
    public static void main(String[] args) {
        EmpleableHumano trabajadorDiurno = new TrabajadorDiurno();
        trabajadorDiurno.trabajar();
        trabajadorDiurno.comer();

        EmpleableHumano trabajadorNocturno = new TrabajadorNocturno();
        trabajadorNocturno.trabajar();
        trabajadorNocturno.comer();

        EmpleableParaTrabajar robot = new Robot();
        robot.trabajar();
    }
}
```

```
package solid;
public class TrabajadorDiurno implements EmpleableHumano {
    @Override
    public void trabajar() {
        System.out.println("Trabaja de dia...");
    }
    @Override
    public void comer() {
        System.out.println("Hace una pausa para almorzar...");
    }
}
```

```
package solid;
public class TrabajadorNocturno implements EmpleableHumano {
    @Override
    public void trabajar() {
        System.out.println("Trabaja de noche...");
    }
    @Override
    public void comer() {
        System.out.println("Hace una pausa para cenar...");
    }
}
```

```
package solid;
public class Robot implements EmpleableParaTrabajar {
    @Override
    public void trabajar() {
        System.out.println("Trabaja todo el dia sin pausa...");
    }
}
```



5. Principio de inversión de dependencias

Las clases deben depender de abstracciones, no de implementaciones concretas. Por ello, en el siguiente ejemplo, la clase `Gerente` viola este principio, ya que depende de la clase concreta `Operario`:

```
package solid;
public class Main {
    public static void main(String[] args) {
        Operario op = new Operario();
        Gerente ge = new Gerente();
        ge.asignarOperario(op);
        ge.trabajar();
    }
}
```

```
package solid;
public class Gerente {
    private Operario operarioACargo;
    public void asignarOperario(Operario o) {
        operarioACargo = o;
    }
    public void trabajar() {
        System.out.println("Gerente dando órdenes...");
        operarioACargo.trabajar();
    }
}
```

```
package solid;
public class Operario {
    public void trabajar() {
        System.out.println("Operario trabajando...");
    }
}
```

Si el trabajo del gerente consistiera en darles órdenes a varios tipos de trabajadores (operarios, secretarias, etc.), la clase `Gerente` debería sufrir modificaciones importantes. En cambio, si ésta dependiera solamente de una nueva interfaz `ContratableParaTrabajar`, podrían agregarse nuevas clases de trabajadores que implementaran esta interfaz y, sin embargo, en la clase `Gerente` no se necesitaría ningún cambio. El código modificado para cumplir con este principio es el siguiente:

```
package solid;
public class Gerente {
    private ContratableParaTrabajar trabajadorACargo;
    public void asignarTrabajador(ContratableParaTrabajar t) {
        trabajadorACargo = t;
    }
    public void trabajar() {
        System.out.println("Gerente dando órdenes...");
        trabajadorACargo.trabajar();
    }
}
```



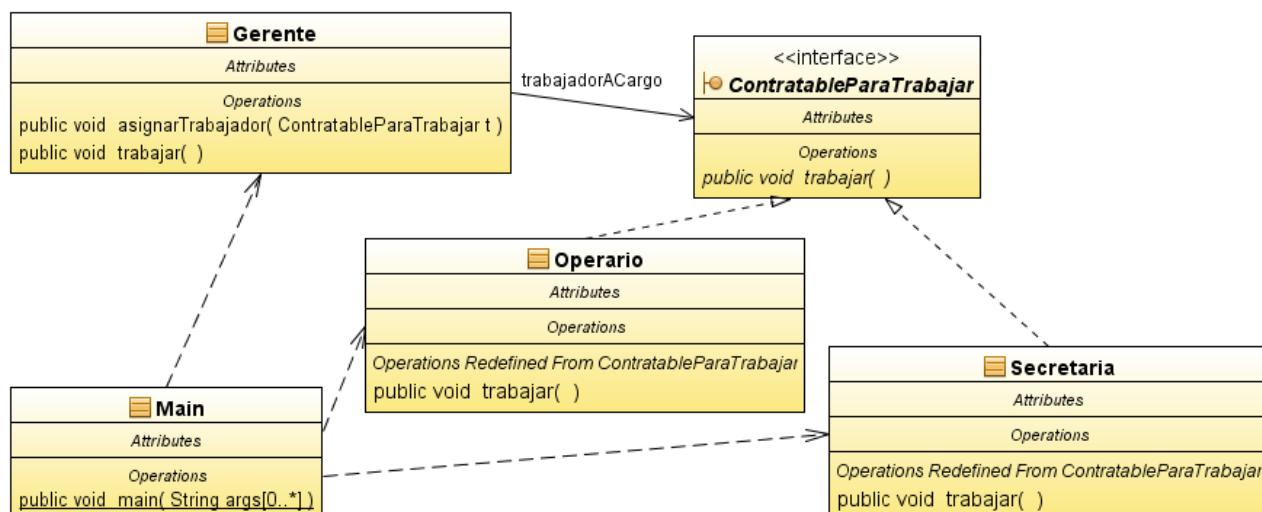
```
package solid;  
  
public interface ContratableParaTrabajar {  
    void trabajar();  
}
```

```
package solid;  
  
public class Operario implements ContratableParaTrabajar {  
    @Override  
    public void trabajar() {  
        System.out.println("Operario trabajando...");  
    }  
}
```

```
package solid;  
  
public class Secretaria implements ContratableParaTrabajar {  
    @Override  
    public void trabajar() {  
        System.out.println("Secretaria trabajando...");  
    }  
}
```

```
package solid;  
  
public class Main {  
    public static void main(String[] args) {  
        Operario op = new Operario();  
        Secretaria se = new Secretaria();  
        Gerente ge = new Gerente();  
  
        ge.asignarTrabajador(op);  
        ge.trabajar();  
  
        ge.asignarTrabajador(se);  
        ge.trabajar();  
    }  
}
```

En el diagrama de clases queda claro que ahora **Gerente** depende de la interfaz **ContratableParaTrabajar** y no de las clases concretas **Operario** y **Secretaria**.





En el diagrama anterior y en muchos otros (por ejemplo, en los de las pág. 35 y 36) puede observarse una gran cantidad de dependencias entre `Main` y las demás clases. Se dice que entre estas clases hay un alto acoplamiento (*high coupling*). Una característica del software de calidad es que sus clases tienen **bajo acoplamiento** (*low coupling*) y **alta cohesión** (*high cohesion*, es decir, son muy específicas).

Ciertos **patrones de diseño**¹ ayudan a desacoplar las clases de un sistema. A continuación, veremos un patrón conocido como **Factory** que permite el desacople entre las clases donde ciertos objetos son utilizados y las clases concretas a que éstos pertenecen. Veamos la siguiente variación del modelo que presentamos en la pág. 35.

```
package figuras;

public class FactoriaDePoligonos {

    public static PoligonoRegular crearPoligono(String tipo, double lado) {
        PoligonoRegular p = null;
        if (tipo.equals("TrianguloEquilatero")) {
            p = new TrianguloEquilatero(lado);
        } else if (tipo.equals("Cuadrado")) {
            p = new Cuadrado(lado);
        } else if (tipo.equals("PentagonoRegular")) {
            p = new PentagonoRegular(lado);
        } else {
            throw new IllegalArgumentException(tipo);
        }
        return p;
    }
}
```

```
package figuras;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        ArrayList<PoligonoRegular> c = new ArrayList<>();
        c.add(FactoriaDePoligonos.crearPoligono("TrianguloEquilatero", 5));
        c.add(FactoriaDePoligonos.crearPoligono("Cuadrado", 5));
        c.add(FactoriaDePoligonos.crearPoligono("PentagonoRegular", 5));

        for (int i = 0; i < c.size(); i++) {
            PoligonoRegular p = c.get(i);
            System.out.println(p.getNombre() + ": " + Math.floor(p.calcularArea()*100)/100);
        }
    }
}
```

Como se puede observar, la clase `Main` ahora sólo está acoplada a las clases `PoligonoRegular` y `FactoriaDePoligonos`, y es en el método estático `crearPoligono` perteneciente a ésta última donde se crean las instancias de las clases `TrianguloEquilatero`, `Cuadrado` y `Pentagono`, dependiendo del valor del primer argumento (.....).

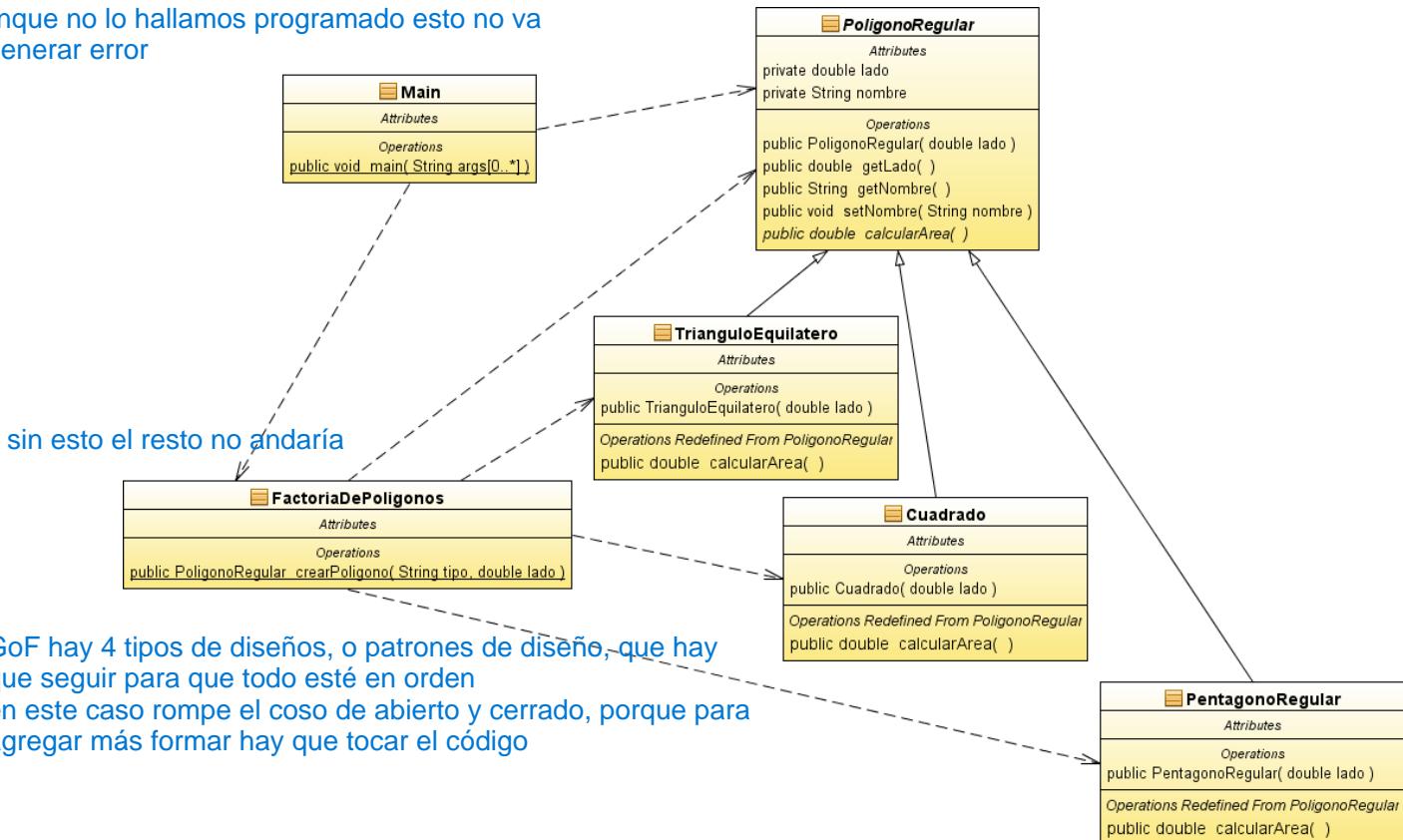
¹ Un patrón de diseño es una solución a un problema de diseño, considerada efectiva y reutilizable. Los más conocidos fueron publicados en el libro *Design Patterns* escrito por el grupo *Gang of Four (GoF)* compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, en el que se recogían 23 patrones de diseño comunes.



recordatorio, buscar Code smell (hace referencia a notar que el código va a tener un error si no me equivoco)

El nuevo diagrama de clases es el siguiente:

Aunque no lo hallamos programado esto no va a generar error



Sin embargo, la anterior implementación del método `crearPolígono` viola el *principio abierto/cerrado* (pág. 45). Lo ideal sería usar un enfoque diferente, donde no fuera necesario cambiar el código de `FactoriaDePoligonos` cada vez que se agregue una nueva subclase de `PoligonoRegular`, y que mantenga el bajo acoplamiento de `Main`. La siguiente implementación lo logra, aunque para ello es necesario aplicar técnicas de mayor complejidad.

```
package figuras;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        ArrayList<PoligonoRegular> c = new ArrayList<>();
        FactoriaDePoligonos fp = FactoriaDePoligonos.getInstance();
        c.add(fp.crearPoligono("TrianguloEquilatero", 5));
        c.add(fp.crearPoligono("Cuadrado", 5));
        c.add(fp.crearPoligono("PentagonoRegular", 5));

        for (int i = 0; i < c.size(); i++) {
            PoligonoRegular p = c.get(i);
            System.out.println(p.getNombre() + ": " + Math.floor(p.calcularArea()*100)/100);
        }
    }
}
```



```
package figuras;

public class FactoriaDePoligonos {

    private FactoriaDePoligonos() {
    }

    private static FactoriaDePoligonos f = null;

    public static FactoriaDePoligonos getInstance() {
        if (f == null) {
            f = new FactoriaDePoligonos();
        }
        return f;
    }

    public PoligonoRegular crearPoligono(String poligonoRegularID, double lado) {
        PoligonoRegular p = null;
        try {
            p = (PoligonoRegular) Class.forName(f.getClass().getPackage().getName() + "." +
                poligonoRegularID).getDeclaredConstructor().newInstance();
        } catch (Exception ex) {
            System.err.println(ex);
        }
        if (p == null) {
            throw new IllegalArgumentException(poligonoRegularID);
        } else {
            p.setLado(lado);
        }
        return p;
    }
}
```

```
package figuras;

public abstract class PoligonoRegular {

    private double lado;
    private String nombre;

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public abstract double calcularArea();
}
```



```
package figuras;

public class TrianguloEquilatero extends PoligonoRegular {

    public TrianguloEquilatero() {
        setNombre("Triangulo equilatero");
    }

    @Override
    public double calcularArea() {
        return getLado() * getLado() * Math.sqrt(3) / 4;
    }

}
```

```
package figuras;

public class Cuadrado extends PoligonoRegular {

    public Cuadrado() {
        setNombre("Cuadrado");
    }

    @Override
    public double calcularArea() {
        return getLado() * getLado();
    }

}
```

```
package figuras;

public class PentagonoRegular extends PoligonoRegular {

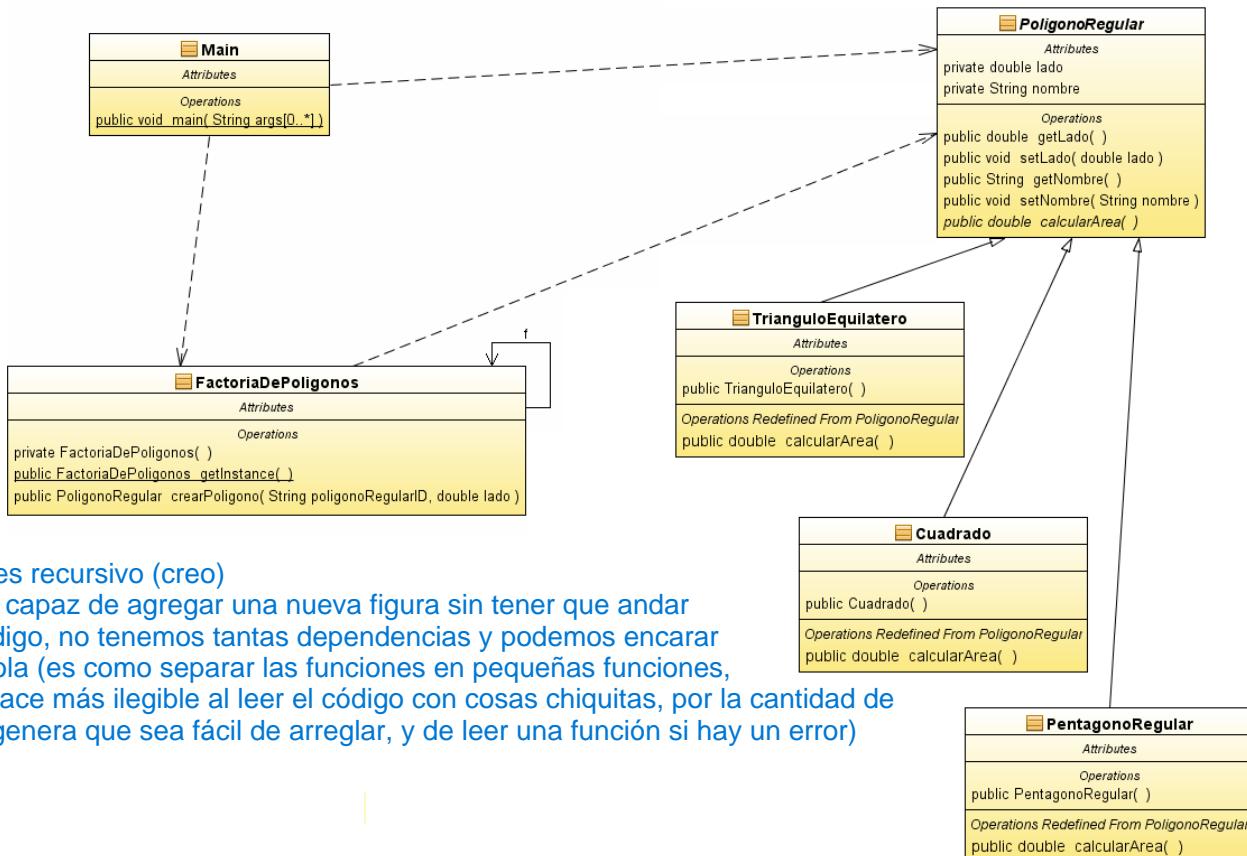
    public PentagonoRegular() {
        setNombre("Pentagono Regular");
    }

    @Override
    public double calcularArea() {
        return 5 * getLado() * getLado() / (4 * Math.tan(Math.PI / 5.0));
    }

}
```



En el diagrama de clases es posible observar que se han eliminado las dependencias que existían entre la clase FactoriaDePoligonos y las clases TrianguloEquilatero, Cuadrado y PentagonoRegular.



ahora esto es recursivo (creo)

y ahora soy capaz de agregar una nueva figura sin tener que andar tocando código, no tenemos tantas dependencias y podemos encarar esa parte sola (es como separar las funciones en pequeñas funciones, aunque lo hace más ilegible al leer el código con cosas chiquitas, por la cantidad de texto, esto genera que sea fácil de arreglar, y de leer una función si hay un error)

Esto permite que la clase FactoriaDePoligonos cumpla con el **principio abierto/cerrado**, es decir, su comportamiento puede extenderse sin modificar su código.

El método estático `forName` (perteneciente a `Class`) permite inicializar la clase cuyo nombre se recibe como argumento. El objeto resultante (que es una instancia de `Class`) es capaz de responder al mensaje devolviendo un objeto que resulta de instanciar la clase en cuestión mediante el constructor sin parámetros. Es por ello que FactoriaDePoligonos debe terminar la construcción de los polígonos cargando el valor de `lado` a través de la invocación de

A diferencia de `new`, que genera un alto acoplamiento entre las clases, `newInstance` permite compilar programas donde se requiera instanciar clases cuyos nombres recién se conocerán en tiempo de ejecución.

La clase FactoriaDePoligonos sigue el patrón de diseño **Singleton**, ya que no permite ser instanciada desde otras clases mediante `new` (sólo proporciona un constructor, que es privado), y la única instancia suya que puede existir debe obtenerse mediante el método público estático `getInstance`.



Para mostrar la importancia de aprender *patrones de diseño*, veamos el siguiente caso¹.

Una empresa produce un juego en el cual deben cazarse patos. El juego muestra una gran variedad de patos nadando y parpando.

```
package patos;
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<Pato> c = new ArrayList<>();
        FactoriaDePatos fp = FactoriaDePatos.getInstance();
        c.add(fp.crearPato("PatoOvero"));
        c.add(fp.crearPato("PatoCapuchino"));
        c.add(fp.crearPato("PatoDeGoma"));
        for (int i = 0; i < c.size(); i++) {
            Pato p = c.get(i);
            p.mostrar();
            p.parpar();
            p.nadar();
            System.out.println();
        }
    }
}
```

```
package patos;

public class FactoriaDePatos {

    private FactoriaDePatos() {
    }

    private static FactoriaDePatos f = null;

    public static FactoriaDePatos getInstance() {
        if (f == null) {
            f = new FactoriaDePatos();
        }
        return f;
    }

    public Pato crearPato(String patoID) {
        Pato p = null;
        try {
            p = (Pato) Class.forName(f.getClass().getPackage().getName() + "." +
                patoID).getDeclaredConstructor().newInstance();
        } catch (Exception ex) {
            System.err.println(ex);
        }
        if (p == null) {
            throw new IllegalArgumentException(patoID);
        }
        return p;
    }
}
```

¹ Freeman, E. et al. (2004): *Head First. Design Patterns*. O'Reilly Media, pp. 2-24 (adaptado)



```
package patos;

public abstract class Pato {

    public void parpar() {
        System.out.println("Cua cuá!");
    }

    public void nadar() {
        System.out.println("Pato nadando...");
    }

    public abstract void mostrar();
}
```

```
package patos;

public class PatoOvero extends Pato {

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato overo.");
    }
}
```

```
package patos;

public class PatoCapuchino extends Pato {

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato capuchino.");
    }
}
```

```
package patos;

public class PatoDeGoma extends Pato {

    @Override
    public void parpar() {
        System.out.println("<sonido artificial de pato>");
    }

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato de goma.");
    }
}
```



Recientemente, la competencia logró lanzar una imitación del juego de esta empresa y, ante la caída de las ventas, los ejecutivos decidieron que tendrían que mostrar algo *realmente* impresionante en la próxima reunión de accionistas, la cual ocurriría en una semana: los patos deberían VOLAR. Le encargaron la misión a José, uno de los programadores de la compañía.



Sin embargo, el día de la reunión de accionistas algo salió *muy mal*...



En un primer momento, José pensó en solucionar el problema redefiniendo el método *volar* en la clase *PatoDeGoma*, tal como ya se había hecho con *parpar*. Pero después pensó que andar redefiniendo métodos para suprimir funcionalidad no era una buena idea, ya que estaría violando el *Principio de Sustitución de Liskov*. Recordó entonces que el uso de **interfaces** siempre le había sido útil cada vez que tuvo un problema similar a éste, y decidió que cada pato que parpara implementaría la interfaz *CapazDeParpar*, los que nadaran implementarían *CapazDeNadar* y los que volaran implementarían *CapazDeVolar*. Además, de esta forma el sistema también sería compatible con los objetos de la clase *PatoDecorativoDeCeramica*, que no parpan, ni nadan, ni vuelan. ¡Y la clase *FactoriaDePatos* no necesitaría cambios!



```
package patos;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        ArrayList<Pato> c = new ArrayList<>();
        FactoriaDePatos fp = FactoriaDePatos.getInstance();
        c.add(fp.crearPato("PatoOvero"));
        c.add(fp.crearPato("PatoCapuchino"));
        c.add(fp.crearPato("PatoDeGoma"));
        c.add(fp.crearPato("PatoDecorativoDeCeramica"));

        for (int i = 0; i < c.size(); i++) {
            Pato p = c.get(i);
            p.mostrar();
            if (p instanceof CapazDeParpar) {
                ((CapazDeParpar) p).parpar();
            }
            if (p instanceof CapazDeNadar) {
                ((CapazDeNadar) p).nadar();
            }
            if (p instanceof CapazDeVolar) {
                ((CapazDeVolar) p).volar();
            }
            System.out.println();
        }
    }
}
```

```
package patos;

public abstract class Pato {

    public abstract void mostrar();
}
```

```
package patos;

public interface CapazDeParpar {

    public void parpar();
}
```

```
package patos;

public interface CapazDeNadar {

    public void nadar();
}
```

```
package patos;

public interface CapazDeVolar {

    public void volar();
}
```



```
package patos;

public class PatoDecorativoDeCeramica extends Pato {

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato decorativo de ceramica.");
    }

}
```



```
package patos;

public class PatoDeGoma extends Pato implements CapazDeParpar, CapazDeNadar {

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato de goma.");
    }

    @Override
    public void parpar() {
        System.out.println("<sonido artificial de pato >");
    }

    @Override
    public void nadar() {
        System.out.println("Pato nadando...");
    }

}
```



```
package patos;

public class PatoOvero extends Pato implements CapazDeParpar,CapazDeNadar,CapazDeVolar{

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato overo.");
    }

    @Override
    public void parpar() {
        System.out.println("Cua, cua!");
    }

    @Override
    public void nadar() {
        System.out.println("Pato nadando...");
    }

    @Override
    public void volar() {
        System.out.println("Pato volando...");
    }
}
```

```
package patos;

public class PatoCapuchino extends Pato implements CapazDeParpar,CapazDeNadar,CapazDeVolar{

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato capuchino.");
    }

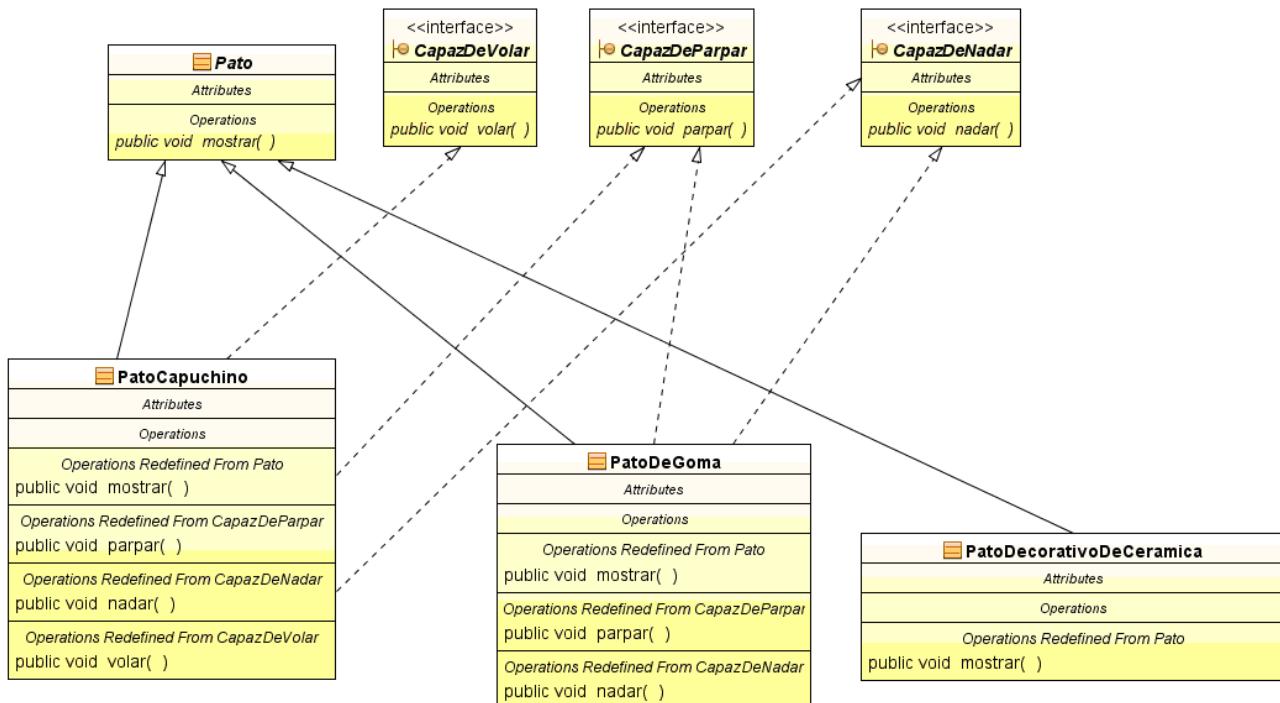
    @Override
    public void parpar() {
        System.out.println("Cua, cua!");
    }

    @Override
    public void nadar() {
        System.out.println("Pato nadando...");
    }

    @Override
    public void volar() {
        System.out.println("Pato volando...");
    }
}
```



Algunas de las generalizaciones y realizaciones que aparecen en el diagrama de clases correspondiente al código anterior son las siguientes:



¡Esta vez te superaste!

Nunca me habías presentado algo tan mal diseñado como esto... Si alguna vez cambiamos el algoritmo para parpar, nadar o volar, ivamos a tener que corregir, una por una, **MONTONES** de subclases!



LA ÚNICA CONSTANTE DEL DESARROLLO DE SOFTWARE
¿Con qué podemos contar siempre cuando desarrollamos software?

O I D M A O

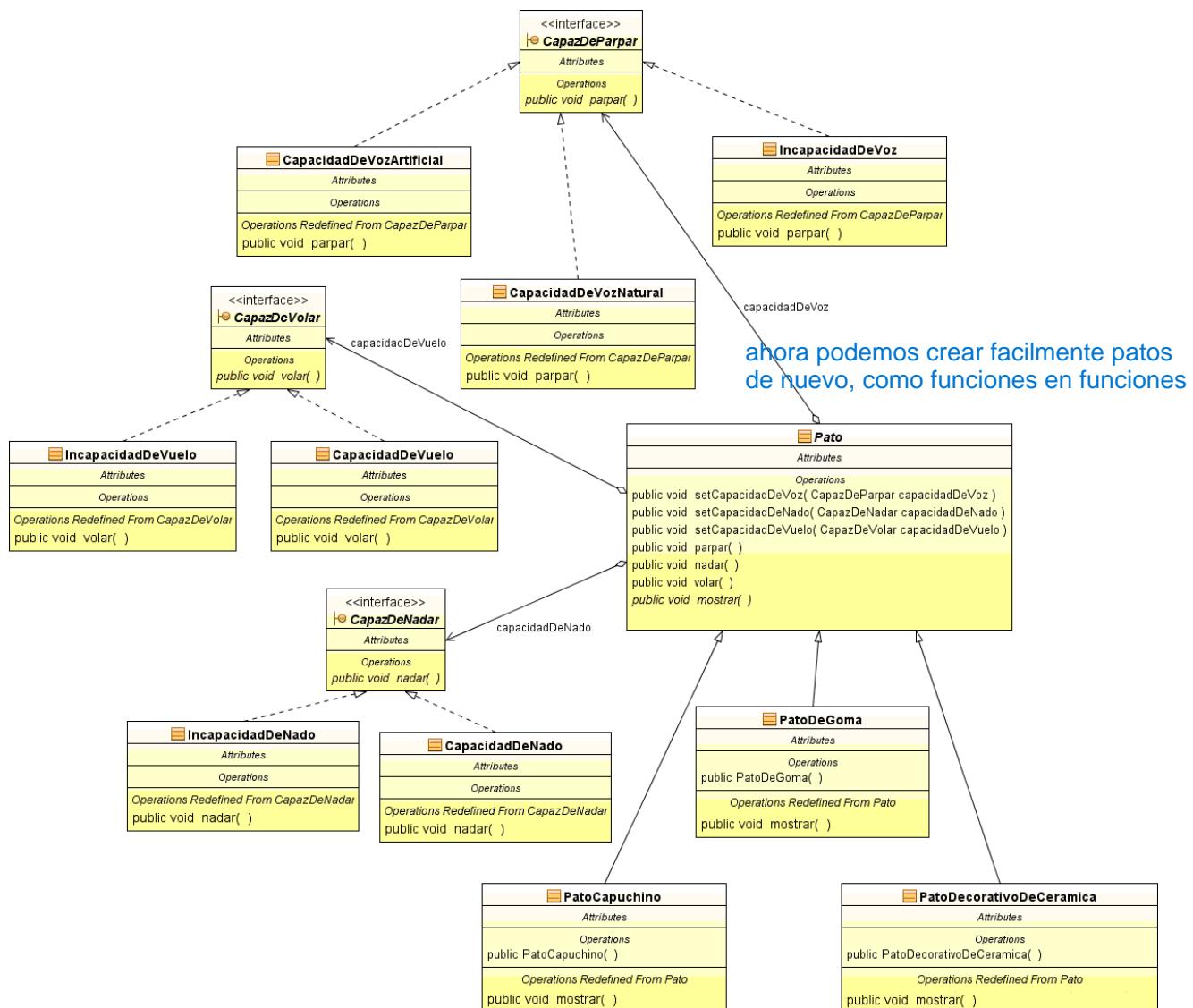
(usar un espejo para leer)

En ese momento, José se dio cuenta de que las interfaces, aunque ayudan a resolver ciertos efectos indeseados de la herencia, pueden crear una *pesadilla de mantenimiento* cuando atentan contra la reutilización de código.

La solución consiste en no implementar las interfaces en las subclases de Pato sino en otras clases nuevas. Las instancias de las subclases de Pato podrán enviarles mensajes a las instancias de estas clases nuevas para conseguir el comportamiento deseado. Esto se denomina **delegación**. Es decir que, en lugar de obtener el comportamiento por *herencia* (relación IS-A) las subclases de Pato lo reciben por **composición** (relación HAS-A). De hecho, muchos patrones de diseño se basan en el siguiente principio:

Debe usarse preferentemente la composición en lugar de la herencia.

El siguiente diagrama de clases muestra que la clase Pato tiene los atributos `capacidadDeVoz`, `capacidadDeVuelo` y `capacidadDeNado` (con sus correspondientes *setters*). Estos atributos son referencias a objetos creados al instanciar las subclases de Pato y proporcionan implementaciones de los métodos `parpar`, `volar` y `nadar`, respectivamente. Al encapsular el comportamiento en clases independientes y utilizarlo mediante delegación, logramos que éste pueda cambiarse sin tener que modificar el código de las clases que lo utilizan. Esto se conoce como *patrón de diseño Strategy*.





El código final es el siguiente (FactoriaDePatos y las tres interfaces CapazDeParpar, CapazDeNadar y CapazDeVolar no sufrieron ningún cambio):

```
package patos;  
  
import java.util.ArrayList;  
  
public class Main {  
  
    public static void main(String[] args) {  
        ArrayList<Pato> c = new ArrayList<>();  
        FactoriaDePatos fp = FactoriaDePatos.getInstance();  
        c.add(fp.crearPato("PatoOvero"));  
        c.add(fp.crearPato("PatoCapuchino"));  
        c.add(fp.crearPato("PatoDeGoma"));  
        c.add(fp.crearPato("PatoDecorativoDeCeramica"));  
  
        for (int i = 0; i < c.size(); i++) {  
            Pato p = c.get(i);  
            p.mostrar();  
            p.parpar();  
            p.nadar();  
            p.volar();  
            System.out.println();  
        }  
    }  
}
```

```
package patos;  
  
public abstract class Pato {  
  
    private CapazDeParpar capacidadDeVoz;  
    private CapazDeNadar capacidadDeNado;  
    private CapazDeVolar capacidadDeVuelo;  
  
    public void setCapacidadDeVoz(CapazDeParpar capacidadDeVoz) {  
        this.capacidadDeVoz = capacidadDeVoz;  
    }  
  
    public void setCapacidadDeNado(CapazDeNadar capacidadDeNado) {  
        this.capacidadDeNado = capacidadDeNado;  
    }  
  
    public void setCapacidadDeVuelo(CapazDeVolar capacidadDeVuelo) {  
        this.capacidadDeVuelo = capacidadDeVuelo;  
    }  
  
    public void parpar(){  
        capacidadDeVoz.parpar();  
    }  
  
    public void nadar(){  
        capacidadDeNado.nadar();  
    }  
  
    public void volar(){  
        capacidadDeVuelo.volar();  
    }  
  
    public abstract void mostrar();  
}
```



```
package patos;

public class PatoCapuchino extends Pato {

    public PatoCapuchino() {
        setCapacidadDeVoz(new CapacidadDeVozNatural());
        setCapacidadDeNado(new CapacidadDeNado());
        setCapacidadDeVuelo(new CapacidadDeVuelo());
    }

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato capuchino.");
    }
}
```

```
package patos;

public class PatoOvero extends Pato {

    public PatoOvero() {
        setCapacidadDeVoz(new CapacidadDeVozNatural());
        setCapacidadDeNado(new CapacidadDeNado());
        setCapacidadDeVuelo(new CapacidadDeVuelo());
    }

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato overo.");
    }
}
```

```
package patos;

public class PatoDeGoma extends Pato {

    public PatoDeGoma() {
        setCapacidadDeVoz(new CapacidadDeVozArtificial());
        setCapacidadDeNado(new CapacidadDeNado());
        setCapacidadDeVuelo(new IncapacidadDeVuelo());
    }

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato de goma.");
    }
}
```



```
package patos;

public class PatoDecorativoDeCeramica extends Pato {

    public PatoDecorativoDeCeramica() {
        setCapacidadDeVoz(new IncapacidadDeVoz());
        setCapacidadDeNado(new IncapacidadDeNado());
        setCapacidadDeVuelo(new IncapacidadDeVuelo());
    }

    @Override
    public void mostrar() {
        System.out.println("Se muestra un pato decorativo de ceramica.");
    }

}
```

```
package patos;

public class CapacidadDeVozNatural implements CapazDeParpar {

    @Override
    public void parpar() {
        System.out.println("Cua, cua!");
    }
}
```

```
package patos;

public class CapacidadDeVozArtificial implements CapazDeParpar {

    @Override
    public void parpar() {
        System.out.println("<sonido artificial de pato>");
    }
}
```

```
package patos;

public class IncapacidadDeVoz implements CapazDeParpar {

    @Override
    public void parpar() {
        System.out.println("<silencio>");
    }
}
```



```
package patos;

public class CapacidadDeNado implements CapazDeNadar {

    @Override
    public void nadar() {
        System.out.println("Pato nadando...");
    }
}
```

```
package patos;

public class IncapacidadDeNado implements CapazDeNadar {

    @Override
    public void nadar() {
        System.out.println("Pato incapaz de nadar...");
    }
}
```

```
package patos;

public class CapacidadDeVuelo implements CapazDeVolar {

    @Override
    public void volar() {
        System.out.println("Pato volando...");
    }
}
```

```
package patos;

public class IncapacidadDeVuelo implements CapazDeVolar {

    @Override
    public void volar() {
        System.out.println("Pato incapaz de volar...");
    }
}
```



DESARROLLADORA
ESCÉPTICA



GURÚ DE
LOS PATRONES
DE DISEÑO



Repaso

Ejemplo práctico de Ingeniería de Software

(Extraído de Deitel, P. & Deitel, H.: *Cómo programar en Java, 7^a ed.*, Pearson, 2008)

1. INTRODUCCIÓN A LA TECNOLOGÍA DE OBJETOS Y UML

La orientación a objetos es una manera natural de pensar acerca del mundo real y de escribir programas de cómputo. Nuestro objetivo aquí es ayudarle a desarrollar una forma de pensar orientada a objetos, y de presentarle el Lenguaje Unificado de Modelado (UML), un lenguaje gráfico que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.

En este ejemplo, presentaremos el diseño y la implementación orientados a objetos de un software para una máquina de cajero automático (ATM) simple. Para ello:

- analizaremos un documento de requerimientos típico que describe un sistema de software (el ATM) que construiremos
- determinaremos los objetos requeridos para implementar ese sistema
- estableceremos los atributos que deben tener estos objetos
- fijaremos los comportamientos que exhibirán estos objetos
- especificaremos la forma en que los objetos deben interactuar entre sí para cumplir con los requerimientos del sistema

Usted experimentará una concisa pero sólida introducción al diseño orientado a objetos con UML. Además, afinará sus habilidades para leer código al ver paso a paso la implementación del ATM en Java, cuidadosamente escrita y bien documentada.

Repaso de los conceptos básicos de la tecnología de objetos

Repasemos primero cierta terminología clave. En cualquier parte del mundo real puede ver **objetos**: gente, animales, plantas, automóviles, aviones, edificios, computadoras, etcétera. Los humanos pensamos en términos de objetos. Los teléfonos, casas, semáforos, hornos de microondas y enfriadores de agua son sólo unos cuantos objetos más. Los programas de cómputo están compuestos por muchos objetos de software con capacidad de interacción.

En ocasiones dividimos a los objetos en dos categorías: animados e inanimados. Los objetos animados están “vivos” en cierto sentido; se mueven a su alrededor y hacen cosas. Por otro lado, los objetos inanimados no se mueven por su propia cuenta. Sin embargo, los objetos de ambos tipos tienen ciertas cosas en común. Todos ellos tienen **atributos** (como tamaño, forma, color y peso), y todos exhiben **comportamientos** (por ejemplo, una pelota rueda, rebota, se infla y desinfla; un bebé llora, duerme, gatea, camina y parpadea; un automóvil acelera, frena y da vuelta; una toalla absorbe agua). Estudiemos los tipos de atributos y comportamientos que tienen los objetos de software.



Los humanos aprenden acerca de los objetos existentes estudiando sus atributos y observando sus comportamientos. Distintos objetos pueden tener atributos similares y pueden exhibir comportamientos similares. Por ejemplo, pueden hacerse comparaciones entre los bebés y los adultos, y entre los humanos y los chimpancés.

El **diseño orientado a objetos (DOO)** modela el software en términos similares a los que utilizan las personas para describir objetos del mundo real. Este diseño aprovecha las relaciones entre las clases, en donde los objetos de cierta clase (como una clase de vehículos) tienen las mismas características; los automóviles, camiones, pequeños vagones rojos y patines tienen mucho en común. El DOO también aprovecha las relaciones de **herencia**, en donde las nuevas clases de objetos se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. Un objeto de la clase “convertible” ciertamente tiene las características de la clase más general “automóvil” pero, de manera más específica, el techo de un convertible puede ponerse y quitarse.

El diseño orientado a objetos proporciona una manera natural e intuitiva de ver el proceso de diseño de software: a saber, modelando los objetos por sus atributos y comportamientos, de igual forma que como describimos los objetos del mundo real. El DOO también modela la comunicación entre los objetos. Así como las personas se envían mensajes unas a otras (por ejemplo, un sargento ordenando a un soldado que permanezca firme), los objetos también se comunican mediante mensajes. Un objeto cuenta de banco puede recibir un mensaje para reducir su saldo por cierta cantidad, debido a que el cliente ha retirado esa cantidad de dinero.

El DOO **encapsula** (es decir, envuelve) los atributos y las operaciones (comportamientos) en los objetos; los atributos y las operaciones de un objeto se enlazan íntimamente entre sí. Los objetos tienen la propiedad de **ocultamiento de información**. Esto significa que los objetos pueden saber cómo comunicarse entre sí a través de **interfaces** bien definidas, pero por lo general no se les permite saber cómo se implementan otros objetos; los detalles de la implementación se ocultan dentro de los mismos objetos. Por ejemplo, podemos conducir un automóvil con efectividad, sin necesidad de saber los detalles acerca de cómo funcionan internamente los motores, las transmisiones y los sistemas de escape; siempre y cuando sepamos cómo usar el pedal del acelerador, el pedal del freno, el volante, etcétera. Más adelante veremos por qué el ocultamiento de información es tan imprescindible para la buena ingeniería de software.

Los lenguajes como Java son **orientados a objetos**. La programación en dichos lenguajes se llama **programación orientada a objetos (POO)**, y permite a los programadores de computadoras implementar un diseño orientado a objetos como un sistema funcional. Por otra parte, los lenguajes como C son **por procedimientos**, de manera que la programación tiende a ser **orientada a la acción**. En C, la unidad de programación es la **función**. Los grupos de acciones que realizan cierta tarea común se forman en funciones, y las funciones se agrupan para formar programas. En Java, la unidad de programación es la clase a partir de la cual se **instancian** (crean) los objetos en un momento dado. Las clases en Java contienen **métodos** (que implementan operaciones y son similares a las funciones en C) y **campos** (que implementan atributos).



Los programadores de Java se concentran en crear clases. Cada clase contiene campos, además del conjunto de métodos que manipulan esos campos y proporcionan servicios a **clientes** (es decir, otras clases que utilizan esa clase). El programador utiliza las clases existentes como bloques de construcción para crear nuevas clases.

Las clases son para los objetos lo que los planos de construcción, para las casas. Así como podemos construir muchas casas a partir de un plano, podemos instanciar (crear) muchos objetos a partir de una clase. No puede cocinar alimentos en la cocina de un plano de construcción; puede cocinarlos en la cocina de una casa.

Las clases pueden tener relaciones con otras clases. Por ejemplo, en un diseño orientado a objetos de un banco, la clase “cajero” necesita relacionarse con las clases “cliente”, “cajón de efectivo”, “bóveda”, etcétera. A estas relaciones se les llama **asociaciones**.

Al empaquetar el software en forma de clases, los sistemas de software posteriores pueden **reutilizar** esas clases. Los grupos de clases relacionadas se empaquetan comúnmente como **componentes** reutilizables. Así como los correderos de bienes raíces dicen a menudo que los tres factores más importantes que afectan el precio de los bienes raíces son “la ubicación, la ubicación y la ubicación”, las personas en la comunidad de software dicen a menudo que los tres factores más importantes que afectan el futuro del desarrollo de software son “la reutilización, la reutilización y la reutilización”. Reutilizar las clases existentes cuando se crean nuevas clases y programas es un proceso que ahorra tiempo y esfuerzo; también ayuda a los programadores a crear sistemas más confiables y efectivos, ya que las clases y componentes existentes a menudo han pasado por un proceso extenso de prueba, depuración y optimización del rendimiento.

Evidentemente, con la tecnología de objetos podemos crear la mayoría del software que necesitaremos mediante la combinación de clases, así como los fabricantes de automóviles combinan las piezas intercambiables. Cada nueva clase que usted cree tendrá el potencial de convertirse en una valiosa pieza de software, que usted y otros programadores podrán usar para agilizar y mejorar la calidad de los futuros esfuerzos de desarrollo de software.

Introducción al análisis y diseño orientados a objetos (A/D OO)

Pronto estará escribiendo programas en Java. ¿Cómo creará el código para sus programas? Tal vez, como muchos programadores principiantes, simplemente encenderá su computadora y empezará a teclear. Esta metodología puede funcionar para programas pequeños, pero ¿qué haría usted si se le pidiera crear un sistema de software para controlar miles de máquinas de cajero automático para un importante banco? O suponga que le pidieron trabajar en un equipo de 1.000 desarrolladores de software para construir el nuevo sistema de control de tráfico aéreo. Para proyectos tan grandes y complejos, no podría simplemente sentarse y empezar a escribir programas.



Para crear las mejores soluciones, debe seguir un proceso detallado para **analizar** los **requerimientos** de su proyecto (es decir, determinar *qué* es lo que se supone debe hacer el sistema) y desarrollar un **diseño** que cumpla con esos requerimientos (es decir, decidir *cómo* debe hacerlo el sistema). Idealmente usted pasaría por este proceso y revisaría cuidadosamente el diseño (o haría que otros profesionales de software lo revisaran) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, lo llamamos un **proceso de análisis y diseño orientado a objetos (A/D OO)**. Los programadores experimentados saben que el análisis y el diseño pueden ahorrar innumerables horas, ya que les ayudan a evitar un método de desarrollo de un sistema mal planeado, que tiene que abandonarse en plena implementación, con la posibilidad de desperdiciar una cantidad considerable de tiempo, dinero y esfuerzo.

A/D OO es el término genérico para el proceso de analizar un problema y desarrollar un método para resolverlo. Los problemas pequeños no requieren de un proceso exhaustivo de A/D OO. Podría ser suficiente con escribir **pseudocódigo** antes de empezar a escribir el código en Java; el pseudocódigo es un medio informal para expresar la lógica de un programa. En realidad no es un lenguaje de programación, pero podemos usarlo como un tipo de bosquejo para guiarnos mientras escribimos nuestro código.

A medida que los problemas y los grupos de personas que los resuelven aumentan en tamaño, los métodos de A/D OO se vuelven más apropiados que el pseudocódigo. Idealmente, un grupo debería acordar un proceso estrictamente definido para resolver su problema, y establecer también una manera uniforme para que los miembros del grupo se comuniquen los resultados de ese proceso entre sí. Aunque existen diversos procesos de A/D OO, hay un lenguaje gráfico para comunicar los resultados de *cualquier* proceso A/D OO que se ha vuelto muy popular. Este lenguaje, conocido como Lenguaje Unificado de Modelado (UML), se desarrolló a mediados de la década de los noventa, bajo la dirección inicial de tres metodólogos de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

¿Qué es UML?

UML es ahora el esquema de representación gráfica más utilizado para modelar sistemas orientados a objetos. Evidentemente ha unificado los diversos esquemas de notación populares. Aquellos quienes diseñan sistemas utilizan el lenguaje (en forma de diagramas) para modelar sus sistemas.

Una característica atractiva es su flexibilidad. UML es **extensible** (es decir, capaz de mejorarse con nuevas características) e independiente de cualquier proceso de A/D OO específico. Los modeladores de UML tienen la libertad de diseñar sistemas utilizando varios procesos, pero todos los desarrolladores pueden ahora expresar esos diseños con un conjunto de notaciones gráficas estándar.

UML es un lenguaje gráfico complejo, con muchas características. Aquí presentaremos un subconjunto conciso y simplificado de estas características. Luego utilizaremos este subconjunto para guiarlo a través de la experiencia de su primer diseño con UML.



Recursos Web de UML

Para obtener más información acerca de UML, consulte los siguientes sitios Web:

- www.uml.org
Esta página de recursos de UML del Grupo de Administración de Objetos (OMG) proporciona documentos de la especificación para UML y otras tecnologías orientadas a objetos.
- es.wikipedia.org/wiki/UML
La definición de Wikipedia del UML en español.

Ejercicios de autorrepaso

1. Liste tres ejemplos de objetos reales que no mencionamos. Para cada objeto, liste varios atributos y comportamientos.
2. El pseudocódigo es _____.
 - a) otro término para el A/D OO
 - b) un lenguaje de programación utilizado para visualizar diagramas de UML
 - c) un medio informal para expresar la lógica de un programa
 - d) un esquema de representación gráfica para modelar sistemas orientados a objetos
3. El UML se utiliza principalmente para _____.
 - a) probar sistemas orientados a objetos
 - b) diseñar sistemas orientados a objetos
 - c) implementar sistemas orientados a objetos
 - d) a y b

Respuestas a los ejercicios de autorrepaso

1. [Nota: las respuestas pueden variar]. a) Los atributos de una televisión incluyen el tamaño de la pantalla, el número de colores que puede mostrar, su canal actual y su volumen actual. Una televisión se enciende y se apaga, cambia de canales, muestra video y reproduce sonidos. b) Los atributos de una cafetera incluyen el volumen máximo de agua que puede contener, el tiempo requerido para preparar una jarra de café y la temperatura del plato calentador bajo la jarra de café. Una cafetera se enciende y se apaga, prepara café y lo calienta. c) Los atributos de una tortuga incluyen su edad, el tamaño de su caparazón y su peso. Una tortuga camina, se mete en su caparazón, emerge del mismo y come vegetación.
2. c.
3. b.



2. CÓMO EXAMINAR UN DOCUMENTO DE REQUERIMIENTOS

Ahora empezaremos nuestro ejemplo práctico de diseño e implementación orientados a objetos que le ayudará a incursionar en la orientación a objetos, mediante el análisis de un ejemplo práctico de una máquina de cajero automático (*Automated Teller Machine* o ATM, por sus siglas en inglés). Este ejemplo práctico le brindará una experiencia de diseño e implementación substancial, cuidadosamente pautada y completa. Llevaremos a cabo los diversos pasos de un proceso de diseño orientado a objetos (DOO) utilizando UML, mientras relacionamos estos pasos con los conceptos orientados a objetos. Al final implementaremos el ATM utilizando las técnicas de la programación orientada a objetos (POO) en Java y presentaremos la solución completa al ejemplo práctico. Éste no es un ejercicio, sino una experiencia de aprendizaje de extremo a extremo, que concluye con un análisis detallado del código en Java que implementamos, con base en nuestro diseño. Este ejemplo práctico le ayudará a acostumbrarse a los tipos de problemas substanciales que se encuentran en la industria, y sus soluciones potenciales. Esperamos que disfrute esta experiencia de aprendizaje.

Empezaremos nuestro proceso de diseño con la presentación de un **documento de requerimientos**, el cual especifica el propósito general del sistema ATM y qué debe hacer. A lo largo del ejemplo práctico, nos referiremos al documento de requerimientos para determinar con precisión la funcionalidad que debe incluir el sistema.

Documento de requerimientos

Un banco local pretende instalar una nueva máquina de cajero automático (ATM), para permitir a los usuarios (es decir, los clientes del banco) realizar transacciones financieras básicas (figura 1). Cada usuario sólo puede tener una cuenta en el banco. Los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo (es decir, sacar dinero de una cuenta) y depositar fondos (es decir, meter dinero en una cuenta). La interfaz de usuario del cajero automático contiene los siguientes componentes:

- una pantalla que muestra mensajes al usuario
- un teclado que recibe datos numéricos de entrada del usuario
- un dispensador de efectivo que dispensa efectivo al usuario, y
- una ranura de depósito que recibe sobres para depósitos del usuario.

El dispensador de efectivo comienza cada día cargado con 500 billetes de \$20. [Nota: debido al alcance limitado de este ejemplo práctico, ciertos elementos del ATM que se describen aquí no imitan exactamente a los de un ATM real. Por ejemplo, generalmente un ATM contiene un dispositivo que lee el número de cuenta del usuario de una tarjeta para ATM, mientras que este ATM pide al usuario que escriba su número de cuenta. Un ATM real también imprime por lo general un recibo al final de una sesión, pero toda la salida de este ATM aparece en la pantalla].



Figura 1: Interfaz de usuario del cajero automático.

El banco desea que usted desarrolle software para realizar las transacciones financieras que inicien los clientes a través del ATM. Posteriormente, el banco integrará el software con el hardware del ATM. El software debe encapsular la funcionalidad de los dispositivos de hardware (por ejemplo: dispensador de efectivo, ranura para depósito) dentro de los componentes de software, pero no necesita estar involucrado en la manera en que estos dispositivos ejecutan su tarea. El hardware del ATM no se ha desarrollado aún, en vez de que usted escriba un software para ejecutarse en el ATM, deberá desarrollar una primera versión del software para que se ejecute en una computadora personal. Esta versión debe utilizar el monitor de la computadora para simular la pantalla del ATM y el teclado de la computadora para simular el teclado numérico del ATM.

Una sesión con el ATM consiste en la autenticación de un usuario (es decir, proporcionar la identidad del usuario) con base en un número de cuenta y un número de identificación personal (NIP), seguida de la creación y la ejecución de transacciones financieras. Para autenticar un usuario y realizar transacciones, el ATM debe interactuar con la base de datos de información sobre las cuentas del banco (es decir, una colección organizada de datos almacenados en una computadora). Para cada cuenta de banco, la base de datos almacena un número de cuenta, un NIP y un saldo que indica la cantidad de dinero en la cuenta. [Nota: asumiremos que el banco planea construir sólo un ATM, por lo que no necesitamos preocuparnos para que varios ATMs puedan acceder a esta base de datos al mismo tiempo. Lo que es más, supongamos que el banco no realizará modificaciones en la información que hay en la base de datos mientras un usuario accede al ATM. Además, cualquier sistema comercial como un ATM se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de un curso de programación de primer o segundo semestre. No obstante, para simplificar nuestro ejemplo supondremos que el banco confía en el ATM para que acceda a la información en la base de datos y la manipule sin necesidad de medidas de seguridad considerables].



Al acercarse al ATM (suponiendo que nadie lo está utilizando), el usuario deberá experimentar la siguiente secuencia de eventos (vea la figura 1):

- 1) La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de cuenta.
- 2) El usuario introduce un número de cuenta de cinco dígitos, mediante el uso del teclado.
- 3) En la pantalla aparece un mensaje, en el que se pide al usuario que introduzca su NIP (número de identificación personal) asociado con el número de cuenta especificado.
- 4) El usuario introduce un NIP de cinco dígitos mediante el teclado numérico.
- 5) Si el usuario introduce un número de cuenta válido y el NIP correcto para esa cuenta, la pantalla muestra el menú principal (figura 2). Si el usuario introduce un número de cuenta inválido o un NIP incorrecto, la pantalla muestra un mensaje apropiado y después el ATM regresa al *paso 1* para reiniciar el proceso de autenticación.



Figura 2: Menú principal del ATM.

Una vez que el ATM autentica al usuario, el menú principal (figura 2) debe contener una opción numerada para cada uno de los tres tipos de transacciones: solicitud de saldo (opción 1), retiro (opción 2) y depósito (opción 3). El menú principal también debe contener una opción para que el usuario pueda salir del sistema (opción 4). Después el usuario elegirá si desea realizar una transacción (oprime 1, 2 o 3) o salir del sistema (oprime 4).

Si el usuario oprime 1 para solicitar su saldo, la pantalla mostrará el saldo de esa cuenta bancaria. Para ello, el ATM deberá obtener el saldo de la base de datos del banco.



Los siguientes pasos describen las acciones que ocurren cuando el usuario elige la opción 2 para hacer un retiro:

- 1) La pantalla muestra un menú (vea la figura 3) que contiene montos de retiro estándar: \$20 (opción 1), \$40 (opción 2), \$60 (opción 3), \$100 (opción 4) y \$200 (opción 5). El menú también contiene una opción que permite al usuario cancelar la transacción (opción 6).
- 2) El usuario introduce la selección del menú mediante el teclado numérico.
- 3) Si el monto a retirar elegido es mayor que el saldo de la cuenta del usuario, la pantalla muestra un mensaje indicando esta situación y pide al usuario que seleccione un monto más pequeño. Entonces el ATM regresa al *paso 1*. Si el monto a retirar elegido es menor o igual que el saldo de la cuenta del usuario (es decir, un monto de retiro aceptable), el ATM procede al *paso 4*. Si el usuario opta por cancelar la transacción (opción 6), el ATM muestra el menú principal y espera la entrada del usuario.
- 4) Si el dispensador contiene suficiente efectivo para satisfacer la solicitud, el ATM procede al *paso 5*. En caso contrario, la pantalla muestra un mensaje indicando el problema y pide al usuario que seleccione un monto de retiro más pequeño. Despues el ATM regresa al *paso 1*.
- 5) El ATM carga el monto de retiro al saldo de la cuenta del usuario en la base de datos del banco (es decir, resta el monto de retiro al saldo de la cuenta del usuario).
- 6) El dispensador de efectivo entrega el monto deseado de dinero al usuario.
- 7) La pantalla muestra un mensaje para recordar al usuario que tome el dinero.



Figura 3: Menú de retiro del ATM.



Los siguientes pasos describen las acciones que ocurren cuando el usuario elige la opción 3 para hacer un depósito:

- 1) La pantalla muestra un mensaje que pide al usuario que introduzca un monto de depósito o que escriba 0 (cero) para cancelar la transacción.
- 2) El usuario introduce un monto de depósito o 0 mediante el teclado numérico. [Nota: el teclado no contiene un punto decimal o signo de dólares, por lo que el usuario no puede escribir una cantidad real en dólares (por ejemplo, \$1.25), sino que debe escribir un monto de depósito en forma de número de centavos (por ejemplo, 125). Después, el ATM divide este número entre 100 para obtener un número que represente un monto en dólares (por ejemplo, $125 \div 100 = 1.25$)].
- 3) Si el usuario especifica un monto a depositar, el ATM procede al *paso 4*. Si elige cancelar la transacción (escribiendo 0), el ATM muestra el menú principal y espera la entrada del usuario.
- 4) La pantalla muestra un mensaje indicando al usuario que introduzca un sobre de depósito en la ranura para depósitos.
- 5) Si la ranura de depósitos recibe un sobre dentro de un plazo de tiempo no mayor a 2 minutos, el ATM abona el monto del depósito al saldo de la cuenta del usuario en la base de datos del banco (es decir, suma el monto del depósito al saldo de la cuenta del usuario). [Nota: este dinero no está disponible de inmediato para retirarse. El banco debe primero verificar físicamente el monto de efectivo en el sobre de depósito, y cualquier cheque que éste contenga debe validarse (es decir, el dinero debe transferirse de la cuenta del emisor del cheque a la cuenta del beneficiario). Cuando ocurra uno de estos eventos, el banco actualizará de manera apropiada el saldo del usuario que está almacenado en su base de datos. Esto ocurre de manera independiente al sistema ATM]. Si la ranura de depósito no recibe un sobre dentro de un plazo de tiempo no mayor a dos minutos, la pantalla muestra un mensaje indicando que el sistema canceló la transacción debido a la inactividad. Después el ATM muestra el menú principal y espera la entrada del usuario.

Una vez que el sistema ejecuta una transacción en forma exitosa, debe volver a mostrar el menú principal para que el usuario pueda realizar transacciones adicionales. Si el usuario elige salir del sistema, la pantalla debe mostrar un mensaje de agradecimiento y después el mensaje de bienvenida para el siguiente usuario.

Análisis del sistema de ATM

En la declaración anterior se presentó un ejemplo simplificado de un documento de requerimientos. Por lo general, dicho documento es el resultado de un proceso detallado de **recopilación de requerimientos**, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar un documento de requerimientos para software bancario (por ejemplo, el sistema ATM que describimos aquí) podría entrevistar expertos financieros para obtener una mejor comprensión de qué es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de **requerimientos del sistema**, para guiar a los diseñadores de sistemas en el proceso de diseño del mismo.



El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del **ciclo de vida del software**. El ciclo de vida del software especifica las etapas a través de las cuales el software evoluciona desde que fue concebido hasta que deja de utilizarse. Por lo general, estas etapas incluyen: análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retiro. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones con respecto a cuándo y qué tan a menudo deben llevar a cabo los ingenieros de software las diversas etapas. Los **modelos de cascada** realizan cada etapa una vez en sucesión, mientras que los **modelos iterativos** pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir el problema a resolver. Al diseñar cualquier sistema, uno debe *resolver el problema de la manera correcta*, pero de igual manera uno debe *resolver el problema correcto*. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestro documento de requerimientos describe nuestro sistema ATM con el suficiente detalle como para que usted no necesite pasar por una etapa de análisis exhaustiva; ya lo hicimos por usted.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como **modelado de caso-uso**. Este proceso identifica los **casos de uso** del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que los ATMs tengan varios casos de uso, como “Ver saldo de cuenta”, “Retirar efectivo”, “Depositar fondos”, “Transferir fondos entre cuentas” y “Comprar estampas postales”. El sistema ATM simplificado que construiremos en este ejemplo práctico requiere sólo los tres primeros casos de uso.

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Usted ya leyó las descripciones de los casos de uso del sistema ATM en el documento de requerimientos; las listas de pasos requeridos para realizar cada tipo de transacción (como solicitud de saldo, retiro y depósito) describen en realidad los tres casos de uso de nuestro ATM: “Ver saldo de cuenta”, “Retirar efectivo” y “Depositar fondos”, respectivamente.

Diagramas de caso-uso

Ahora presentaremos el primero de varios diagramas de UML en el ejemplo práctico. Crearemos un **diagrama de caso-uso** para modelar las interacciones entre los clientes de un sistema (en este ejemplo práctico, los clientes del banco) y sus casos de uso. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (los cuales presentaremos a lo largo del ejemplo práctico). A menudo, los diagramas de caso-uso se acompañan de texto informal que describe los casos de uso con más detalle; como el texto que aparece en el documento de requerimientos. Los diagramas de caso-uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de caso-uso son herramientas indispensables que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

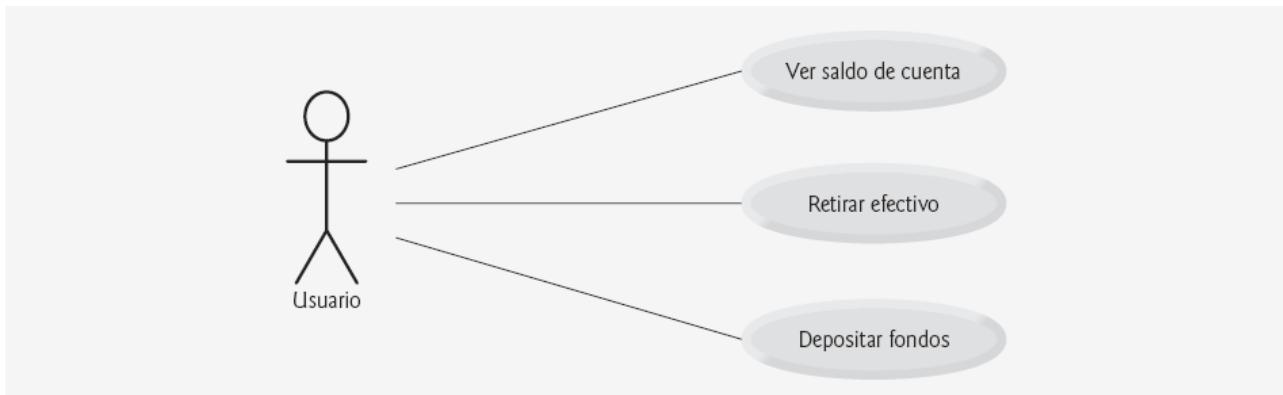


Figura 4: Diagrama de caso-uso para el ATM, desde la perspectiva del usuario.

La figura 4 muestra el diagrama de caso-uso para nuestro sistema ATM. La figura humana representa a un actor, el cual define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro cajero automático, el actor es un Usuario que puede ver el saldo de una cuenta, retirar efectivo y depositar fondos del ATM. El Usuario no es una persona real, sino que constituye los roles que puede desempeñar una persona real (al desempeñar el papel de un Usuario) mientras interactúa con el ATM. Hay que tener en cuenta que un diagrama de caso-uso puede incluir varios actores. Por ejemplo, el diagrama de caso-uso para un sistema ATM de un banco real podría incluir también un actor llamado Administrador, que rellene el dispensador de efectivo a diario.

Nuestro documento de requerimientos provee los actores: “los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo y depositar fondos”. Por lo tanto, el actor en cada uno de estos tres casos de uso es el usuario que interactúa con el ATM. Una entidad externa (una persona real) desempeña el papel del usuario para realizar transacciones financieras. La figura 4 muestra un actor, cuyo nombre (Usuario) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

Los ingenieros de software (más específicamente, los diseñadores de sistemas) deben analizar el documento de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen en un lenguaje de programación específico. Durante la etapa de análisis, los diseñadores de sistemas se enfocan en comprender el documento de requerimientos para producir una especificación de alto nivel que describa *qué* es lo que el sistema debe hacer. El resultado de la etapa de diseño (una **especificación de diseño**) debe detallar claramente *cómo* debe construirse el sistema para satisfacer estos requerimientos. A continuación, llevaremos a cabo los pasos de un proceso simple de diseño orientado a objetos (DOO) con el sistema ATM, para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de esos procesos, de los cuales el más conocido es Rational Unified Process™ (RUP), desarrollado por Rational Software Corporation. RUP es un proceso robusto para diseñar aplicaciones a nivel industrial. Para nuestro ejemplo práctico, presentaremos un proceso de diseño simplificado propio.



Diseño del sistema ATM

Ahora comenzaremos la etapa de diseño de nuestro sistema ATM. Un **sistema** es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema ATM tiene una interfaz de usuario (figura 1), contiene software para ejecutar transacciones financieras e interactúa con una base de datos de información de cuentas bancarias. La **estructura** del sistema describe los objetos del sistema y sus interrelaciones. El **comportamiento** del sistema describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen diversos tipos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

El estándar UML especifica 13 tipos de diagramas para documentar los modelos de un sistema. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema; seis diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. Aquí listaremos sólo los seis tipos de diagramas que utilizaremos en nuestro ejemplo práctico, uno de los cuales (el diagrama de clases) modela la estructura del sistema, mientras que los otros cinco modelan el comportamiento.

- 1) Los **diagramas de caso-uso**, como el de la figura 4, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como “Ver saldo de cuenta”, “Retirar efectivo” y “Depositar fondos”).
- 2) Los **diagramas de clases** modelan las clases o “bloques de construcción” que se usan en un sistema. Cada sustantivo u “objeto” que se describe en el documento de requerimientos es candidato para ser una clase en el sistema (por ejemplo, Cuenta, Teclado). Los diagramas de clases ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema ATM especificará que éste está compuesto físicamente de una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos.
- 3) Los **diagramas de máquina de estado** modelan las formas en que un objeto cambia de estado. El **estado** de un objeto se indica mediante los valores de todos los atributos del objeto, en un momento dado. Cuando un objeto cambia de estado, puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIP de un usuario, el ATM cambia del estado “usuario no autenticado” al estado “usuario autenticado”, punto en el cual el ATM permite al usuario realizar transacciones financieras (por ejemplo, ver el saldo de su cuenta, retirar efectivo, depositar fondos).
- 4) Los **diagramas de actividad** modelan la **actividad** de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el ATM debe obtener el saldo de la cuenta del usuario (de la base de datos de información de las cuentas del banco) antes de que la pantalla pueda mostrar el saldo al usuario.



- 5) Los **diagramas de comunicación** (antiguamente llamados diagramas de colaboración) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de *qué* interacciones ocurren. Estos diagramas muestran cuáles objetos deben interactuar para realizar una transacción en el ATM. Por ejemplo, el ATM debe comunicarse con la base de datos de información de las cuentas del banco para obtener el saldo de una cuenta.
- 6) Los **diagramas de secuencia** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan *cuándo* ocurren las interacciones. Estos diagramas ayudan a mostrar el orden en el que ocurren las interacciones al ejecutar una transacción financiera. Por ejemplo, la pantalla pide al usuario que escriba un monto de retiro antes de dispensar el efectivo.

A continuación, seguiremos diseñando nuestro sistema ATM, identificando las clases en el documento de requerimientos. Para lograr esto, extraeremos sustantivos clave y frases nominales del documento de requerimientos. Mediante el uso de estas clases, desarrollaremos nuestro primer borrador del diagrama de clases que modelará la estructura de nuestro sistema ATM.

Ejercicios de autoevaluación

1. Suponga que habilitamos a un usuario de nuestro sistema ATM para transferir dinero entre dos cuentas bancarias. Modifique el diagrama de caso-uso de la figura 4 para reflejar este cambio.
2. Los _____ modelan las interacciones entre los objetos en un sistema, con énfasis acerca de *cuándo* ocurren estas interacciones.
 - a) Diagramas de clases
 - b) Diagramas de secuencia
 - c) Diagramas de comunicación
 - d) Diagramas de actividad
3. ¿Cuál de las siguientes opciones lista las etapas de un típico ciclo de vida de software, en orden secuencial?
 - a) diseño, análisis, implementación, prueba
 - b) diseño, análisis, prueba, implementación
 - c) análisis, diseño, prueba, implementación
 - d) análisis, diseño, implementación, prueba

Respuestas a los ejercicios de autoevaluación

1. La figura 5 contiene un diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre cuentas.
2. b.
3. d.

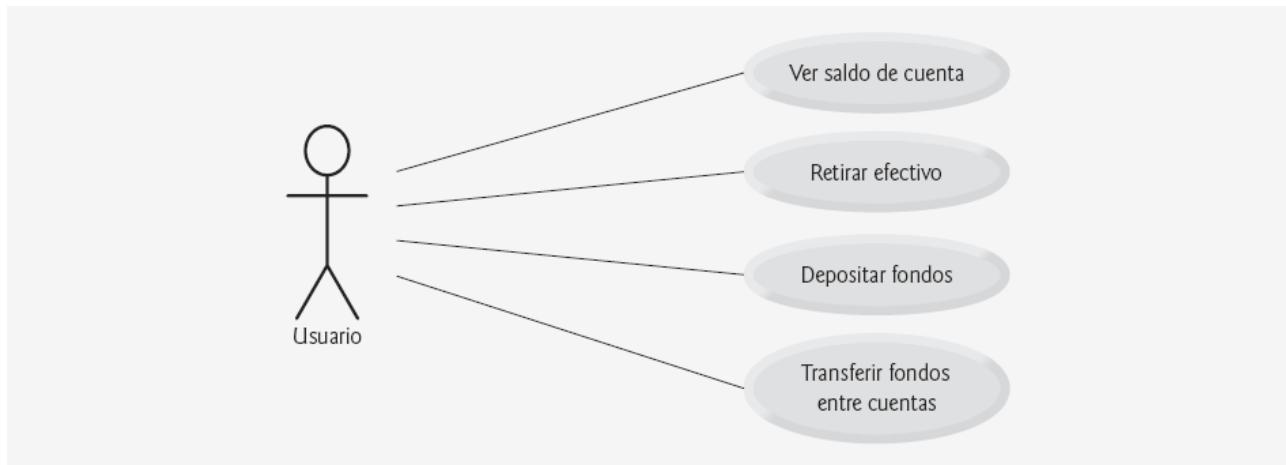


Figura 5: Diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre varias cuentas.

3. IDENTIFICACIÓN DE CLASES EN UN DOCUMENTO DE REQUERIMIENTOS

Para comenzar nuestro proceso de DOO, identificaremos las clases requeridas para crear el sistema ATM. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en Java. Primero debemos revisar el documento de requerimientos, para identificar los sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema ATM. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales.

La figura 6 lista los sustantivos y frases nominales que se encontraron en el documento de requerimientos. Los listaremos de izquierda a derecha, en el orden en el que los encontramos por primera vez en el documento de requerimientos. Sólo listaremos la forma singular de cada sustantivo o frase nominal.

Sustantivos y frases nominales en el documento de requerimientos del ATM		
banco	dinero / fondos	número de cuenta
ATM	pantalla	NIP
usuario	teclado numérico	base de datos del banco
cliente	dispensador de efectivo	solicitud de saldo
transacción	billete de \$20 / efectivo	retiro
cuenta	ranura de depósito	depósito
saldo	sobre de depósito	

Figura 6: Sustantivos y frases nominales en el documento de requerimientos del ATM.



Crearemos clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema ATM. No modelamos “banco” como una clase, ya que el banco no es una parte del sistema ATM; el banco sólo quiere que nosotros construyamos el ATM. “Cliente” y “usuario” también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema ATM, pero no necesitamos modelarlos como clases en el software del ATM. Recuerde que modelamos un usuario del ATM (es decir, un cliente del banco) como el actor en el diagrama de casos de uso de la figura 4.

No necesitamos modelar “billete de \$20” ni “sobre de depósito” como clases. Éstos son objetos físicos en el mundo real, pero no forman parte de lo que se va a automatizar. Podemos representar en forma adecuada la presencia de billetes en el sistema, mediante el uso de un atributo de la clase que modela el dispensador de efectivo (más adelante asignaremos atributos a las clases del sistema ATM). Por ejemplo, el dispensador de efectivo mantiene un conteo del número de billetes que contiene. El documento de requerimientos no dice nada acerca de lo que debe hacer el sistema con los sobres de depósito después de recibirlos. Podemos suponer que con sólo admitir la recepción de un sobre (una operación que realiza la clase que modela la ranura de depósito) es suficiente para representar la presencia de un sobre en el sistema (más adelante asignaremos operaciones a las clases del sistema ATM).

En nuestro sistema ATM simplificado, lo más apropiado sería representar varios montos de “dinero”, incluyendo el “saldo” de una cuenta, como atributos de clases. De igual forma, los sustantivos “número de cuenta” y “NIP” representan piezas importantes de información en el sistema ATM. Son atributos importantes de una cuenta bancaria. Sin embargo, no exhiben comportamientos. Por ende, podemos modelarlos de la manera más apropiada como atributos de una clase de cuenta.

Aunque, con frecuencia, el documento de requerimientos describe una “transacción” en un sentido general, no modelaremos la amplia noción de una transacción financiera en este momento. En vez de ello, modelaremos los tres tipos de transacciones (es decir, “solicitud de saldo”, “retiro” y “depósito”) como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las transacciones que representan. Por ejemplo, para un retiro se necesita conocer el monto de dinero que el usuario desea retirar. Sin embargo, una solicitud de saldo no requiere datos adicionales. Lo que es más, las tres clases de transacciones exhiben comportamientos únicos. Para un retiro se requiere entregar efectivo al usuario, mientras que para un depósito se requiere recibir un sobre de depósito del usuario. [Nota: más adelante, “factorizaremos” las características comunes de todas las transacciones en una clase de “transacción” general, mediante el uso del concepto orientado a objetos de **herencia**].

Determinaremos las clases para nuestro sistema con base en los sustantivos y frases nominales restantes de la figura 6. Cada una de ellas se refiere a uno o varios de los siguientes elementos:

- ATM
- pantalla
- teclado numérico
- dispensador de efectivo
- ranura de depósito
- cuenta
- base de datos del banco
- solicitud de saldo
- retiro
- depósito



Es probable que los elementos de esta lista sean clases que necesitaremos implementar en nuestro sistema.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que hemos creado. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de Java para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula. De esta manera, vamos a crear las clases ATM, Pantalla, Teclado, DispensadorEfectivo, RanuraDeposito, Cuenta, BaseDatosBanco, SolicitudSaldo, Retiro y Deposito. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

Modelado de las clases

UML nos permite modelar, a través de los **diagramas de clases**, las clases en el sistema ATM y sus interrelaciones. La figura 7 representa a la clase ATM. En UML, cada clase se modela como un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase. El compartimiento inferior contiene las operaciones de la clase. En la figura 7, los compartimientos intermedio e inferior están vacíos, ya que no hemos determinado los atributos y operaciones de esta clase todavía.

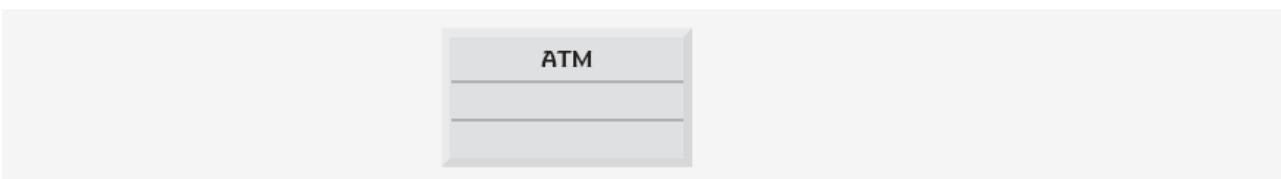


Figura 7: Representación de una clase en UML mediante un diagrama de clases.

Los diagramas de clases también muestran las relaciones entre las clases del sistema. La figura 8 muestra cómo nuestras clases ATM y Retiro se relacionan una con la otra. Por el momento modelaremos sólo este subconjunto de las clases del ATM, por cuestión de simpleza. Más adelante, presentaremos un diagrama de clases más completo. Observe que los rectángulos que representan a las clases en este diagrama no están subdivididos en compartimientos. UML permite suprimir los atributos y las operaciones de una clase de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina **diagrama con elementos omitidos (elided diagram)**: su información, como el contenido de los compartimientos segundo y tercero, no se modela. Más adelante, vamos a colocar información en estos compartimientos.

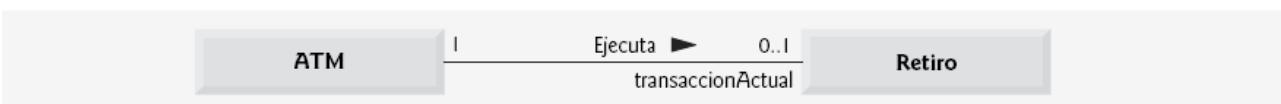


Figura 8: Diagrama de clases que muestra una asociación entre clases



En la figura 8, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto ATM participa en una asociación con cero o con un objeto Retiro; cero si el usuario actual no está realizando una transacción o si ha solicitado un tipo distinto de transacción, y uno si el usuario ha solicitado un retiro. UML puede modelar muchos tipos de multiplicidad. La figura 9 lista y explica los tipos de multiplicidad.

Símbolo	Significado
0	Ninguno.
1	Uno.
m	Un valor entero.
0..1	Cero o uno.
m, n	m o n .
$m..n$	Cuando menos m , pero no más que n .
*	Cualquier entero no negativo (cero o más).
0..*	Cero o más (idéntico a *).
1..*	Uno o más.

Figura 9: Tipos de multiplicidad

Una asociación puede tener nombre. Por ejemplo, la palabra **Ejecuta** por encima de la línea que conecta a las clases **ATM** y **Retiro** en la figura 8 indica el nombre de esa asociación. Esta parte del diagrama se lee así: “un objeto de la clase **ATM** ejecuta cero o un objeto de la clase **Retiro**”. Los nombres de las asociaciones son direccionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como “cero o un objeto de la clase **Retiro** ejecuta un objeto de la clase **ATM**”.

La palabra **transaccionActual** en el extremo de **Retiro** de la línea de asociación en la figura 8 es un **nombre de rol**, el cual identifica el rol que desempeña el objeto **Retiro** en su relación con el **ATM**. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de “profesor” con respecto a los estudiantes. La misma persona puede desempeñar el rol de “colega” cuando participa en una asociación con otro profesor, y de “entrenador” cuando entrena a los atletas estudiantes. En la figura 8, el nombre de rol **transaccionActual** indica que el objeto **Retiro** que participa en la asociación **Ejecuta** con un objeto de la clase **ATM** representa a la transacción que está procesando el **ATM** en ese momento. En otros contextos, un objeto **Retiro** puede desempeñar otros roles (por ejemplo, la transacción anterior). Observe que no especificamos un nombre de rol para el extremo del **ATM** de la asociación **Ejecuta**. A menudo, los nombres de los roles se omiten en los diagramas de clases, cuando el significado de una asociación está claro sin ellos.



Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considere un cajero automático real. ¿Qué “piezas” reúne un fabricante para construir un ATM funcional? Nuestro documento de requerimientos nos indica que el ATM está compuesto de una pantalla, un teclado, un dispensador de efectivo y una ranura de depósito.

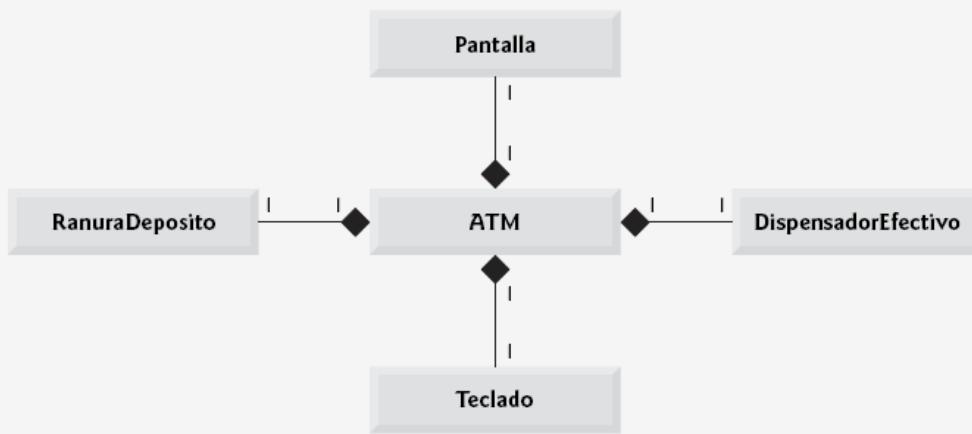


Figura 10: Diagrama de clases que muestra las relaciones de composición.

En la figura 10, los **diamantes sólidos** que se adjuntan a las líneas de asociación de la clase ATM indican que esta clase tiene una relación de **composición** con las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. La composición implica una relación todo/parte. La clase que tiene el símbolo de composición (el diamante sólido) en su extremo de la línea de asociación es el todo (en este caso, ATM), y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. Las composiciones en la figura 10 indican que un objeto de la clase ATM está formado por un objeto de la clase Pantalla, un objeto de la clase DispensadorEfectivo, un objeto de la clase Teclado y un objeto de la clase RanuraDeposito. El ATM “tiene una” pantalla, un teclado, un dispensador de efectivo y una ranura de depósito. La relación “*tiene un*” define la composición (más adelante, veremos que la relación “*es un*” define la herencia).

De acuerdo con la especificación del UML (www.uml.org), las relaciones de composición tienen las siguientes propiedades:

- 1) Sólo una clase en la relación puede representar el todo (es decir, el diamante puede colocarse sólo en un extremo de la línea de asociación). Por ejemplo, la pantalla es parte del ATM o el ATM es parte de la pantalla, pero la pantalla y el ATM no pueden representar ambos el “todo” dentro de la relación.
- 2) Las partes en la relación de composición existen sólo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un ATM incluye la manufactura de sus partes. Lo que es más, si el ATM se destruye, también se destruyen su pantalla, teclado, dispensador de efectivo y ranura de depósito.
- 3) Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

Los diamantes sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación “*tiene un*” no satisface uno o más de estos criterios, UML especifica que se deben adjuntar diamantes sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una computadora personal y un monitor de computadora participan en una relación de agregación: la computadora “tiene un” monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varias computadoras a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

La figura 11 muestra un diagrama de clases para el sistema ATM. Este diagrama modela la mayoría de las clases que identificamos antes en esta sección, así como las asociaciones entre ellas que podemos inferir del documento de requerimientos. [Nota: las clases **SolicitudSaldo** y **Depósito** participan en asociaciones similares a las de la clase **Retiro**, por lo que preferimos omitirlas en este diagrama por cuestión de simpleza. Más adelante, expandiremos nuestro diagrama de clases para incluir todas las clases en el sistema ATM].

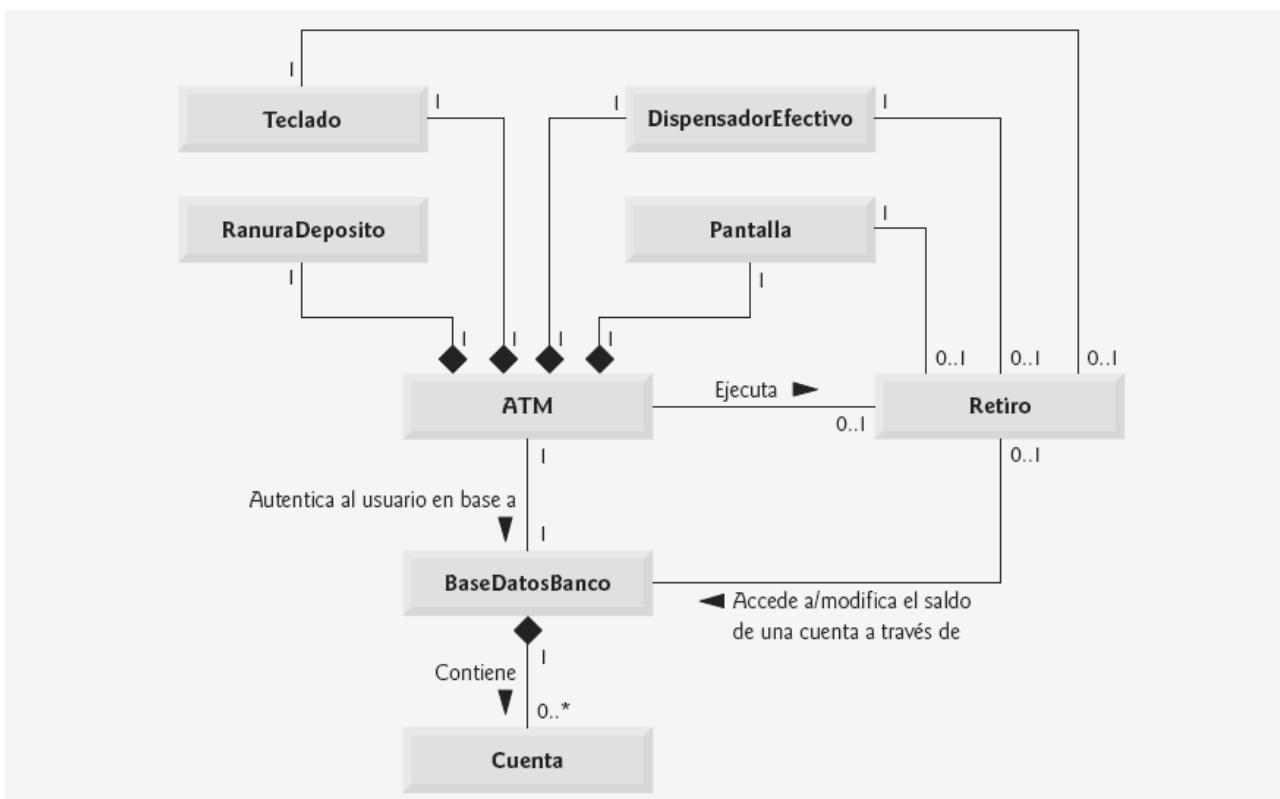


Figura 11: Diagrama de clases para el modelo del sistema ATM.

La figura 11 presenta un modelo gráfico de la estructura del sistema ATM. Este diagrama de clases incluye a las clases **BaseDatosBanco** y **Cuenta**, junto con varias asociaciones que no presentamos en las figuras 8 o 10. El diagrama de clases muestra que la clase **ATM** tiene una **relación de uno a uno** con la clase **BaseDatosBanco**: un objeto **ATM** autentica a los usuarios en base a un objeto **BaseDatosBanco**. En la figura 11 también modelamos el hecho de que la base de datos del banco contiene información sobre muchas cuentas; un objeto de la clase **BaseDatosBanco** participa en una relación



de composición con cero o más objetos de la clase **Cuenta**. Recuerde que en la figura 9 se muestra que el valor de multiplicidad $0..*$ en el extremo de la clase **Cuenta**, de la asociación entre las clases **BaseDatosBanco** y **Cuenta**, indica que cero o más objetos de la clase **Cuenta** participan en la asociación. La clase **BaseDatosBanco** tiene una **relación de uno a varios** con la clase **Cuenta**; **BaseDatosBanco** puede contener muchos objetos **Cuenta**. De manera similar, la clase **Cuenta** tiene una **relación de varios a uno** con la clase **BaseDatosBanco**; puede haber muchos objetos **Cuenta** en **BaseDatosBanco**. [Nota: si recuerda la figura 9, el valor de multiplicidad $*$ es idéntico a $0..*$. Incluimos $0..*$ en nuestros diagramas de clases por cuestión de claridad].

La figura 11 también indica que si el usuario va a realizar un retiro, “un objeto de la clase **Retiro** accede a / modifica el saldo de una cuenta a través de un objeto de la clase **BaseDatosBanco**”. Podríamos haber creado una asociación directamente entre la clase **Retiro** y la clase **Cuenta**. No obstante, el documento de requerimientos indica que el “ATM debe interactuar con la base de datos de información de las cuentas del banco” para realizar transacciones. Una cuenta de banco contiene información delicada, por lo que los ingenieros de sistemas deben considerar siempre la seguridad de los datos personales al diseñar un sistema. Por ello, sólo **BaseDatosBanco** puede acceder a una cuenta y manipularla en forma directa. Todas las demás partes del sistema deben interactuar con la base de datos para recuperar o actualizar la información de las cuentas (por ejemplo, el saldo de una cuenta).

El diagrama de clases de la figura 11 también modela las asociaciones entre la clase **Retiro** y las clases **Pantalla**, **DispensadorEfectivo** y **Teclado**. Una transacción de retiro implica pedir al usuario que seleccione el monto a retirar; también implica recibir entrada numérica. Estas acciones requieren el uso de la pantalla y del teclado, respectivamente. Además, para entregar efectivo al usuario se requiere acceso al dispensador de efectivo.

Aunque no se muestran en la figura 11, las clases **SolicitudSaldo** y **Deposito** participan en varias asociaciones con las otras clases del sistema ATM. Al igual que la clase **Retiro**, cada una de estas clases se asocia con las clases **ATM** y **BaseDatosBanco**. Un objeto de la clase **SolicitudSaldo** también se asocia con un objeto de la clase **Pantalla** para mostrar al usuario el saldo de una cuenta. La clase **Deposito** se asocia con las clases **Pantalla**, **Teclado** y **RanuraDeposito**. Al igual que los retiros, las transacciones de depósito requieren el uso de la pantalla y el teclado para mostrar mensajes y recibir datos de entrada, respectivamente. Para recibir sobres de depósito, un objeto de la clase **Deposito** accede a la ranura de depósitos.

Ya hemos identificado las clases en nuestro sistema ATM (aunque tal vez descubramos otras, a medida que avancemos con el diseño y la implementación). A continuación, determinaremos los atributos para cada una de estas clases, y luego utilizaremos estos atributos para examinar la forma en que cambia el sistema con el tiempo.



Ejercicios de autoevaluación

1. Suponga que tenemos una clase llamada **Auto**, la cual representa a un automóvil. Piense en algunas de las distintas piezas que podría reunir un fabricante para producir un automóvil completo. Cree un diagrama de clases (similar a la figura 10) que modele algunas de las relaciones de composición de la clase **Auto**.
2. Suponga que tenemos una clase llamada **Archivo**, la cual representa un documento electrónico en una computadora independiente, sin conexión de red, representada por la clase **Computadora**. ¿Qué tipo de asociación existe entre la clase **Computadora** y la clase **Archivo**?
 - a) La clase **Computadora** tiene una relación de uno a uno con la clase **Archivo**.
 - b) La clase **Computadora** tiene una relación de varios a uno con la clase **Archivo**.
 - c) La clase **Computadora** tiene una relación de uno a varios con la clase **Archivo**.
 - d) La clase **Computadora** tiene una relación de varios a varios con la clase **Archivo**.
3. Indique si la siguiente aseveración es *verdadera* o *falsa*. Si es *falsa*, explique por qué: un diagrama de clases de UML, en el que no se modelan los compartimientos segundo y tercero, se denomina diagrama con elementos omitidos (*elided diagram*).
4. Modifique el diagrama de clases de la figura 11 para incluir la clase **Depósito**, en vez de la clase **Retiro**.

Respuestas a los ejercicios de autoevaluación

1. [Nota: las respuestas pueden variar]. La figura 12 presenta un diagrama de clases que muestra algunas de las relaciones de composición de una clase **Auto**.

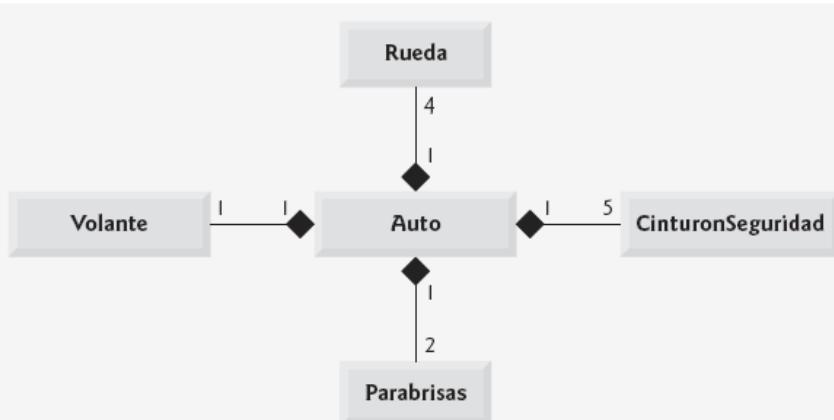


Figura 12: Diagrama de clases que muestra algunas relaciones de composición de una clase **Auto**.

2. c. [Nota: en una computadora con conexión de red, esta relación podría ser de varios a varios].
3. Verdadera.

4. La figura 13 presenta un diagrama de clases para el ATM, en el cual se incluye la clase **Deposito** en vez de la clase **Retiro** (como en la figura 11). Observe que la clase **Deposito** no se asocia con la clase **DispensadorEfectivo**, sino que se asocia con la clase **RanuraDeposito**.

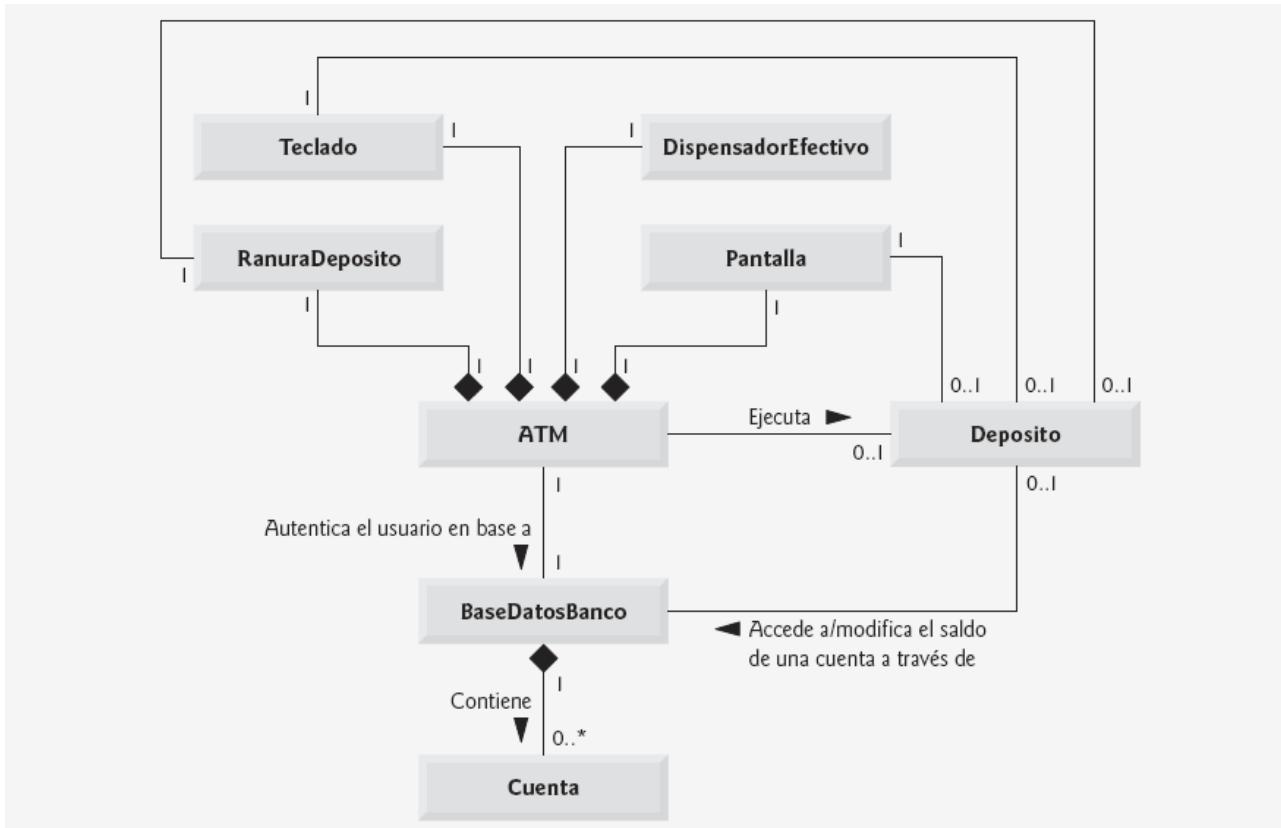


Figura 13: Diagrama de clases para el modelo del sistema ATM, incluyendo la clase **Deposito**.

4. IDENTIFICACIÓN DE LOS ATRIBUTOS DE LAS CLASES

Considere los atributos de algunos objetos reales: los atributos de una persona incluyen su altura, peso y si es zurdo, diestro o ambidiestro. Los atributos de un radio incluyen la estación, el volumen y si está en AM o FM. Los atributos de un automóvil incluyen las lecturas de su velocímetro y odómetro, la cantidad de gasolina en su tanque y la velocidad de marcha en la que se encuentra. Los atributos de una computadora personal incluyen su fabricante (por ejemplo, Dell, Apple o Lenovo), el tipo de pantalla (por ejemplo, LCD o CRT), el tamaño de su memoria principal y el de su disco duro.

Podemos identificar muchos atributos de las clases en nuestro sistema, analizando las palabras y frases descriptivas en el documento de requerimientos. Para cada palabra o frase que descubramos que desempeña un rol importante en el sistema ATM, creamos un atributo y lo asignamos a una o más de las clases ya identificadas anteriormente. También creamos atributos para representar los datos adicionales que pueda necesitar una clase, ya que dichas necesidades se van aclarando a lo largo del proceso de diseño.



La figura 14 lista las palabras o frases del documento de requerimientos que describen a cada una de las clases. Para formar esta lista, leemos el documento de requerimientos e identificamos cualquier palabra o frase que haga referencia a las características de las clases en el sistema. Por ejemplo, el documento de requerimientos describe los pasos que se llevan a cabo para obtener un “monto de retiro”, por lo que listamos “monto” enseguida de la clase Retiro.

Clase	Palabras y frases descriptivas
ATM	el usuario es autenticado
SolicitudSaldo	número de cuenta
Retiro	número de cuenta monto
Deposito	número de cuenta monto
BaseDatosBanco	[no hay palabras o frases descriptivas]
Cuenta	número de cuenta NIP saldo
Pantalla	[no hay palabras o frases descriptivas]
Teclado	[no hay palabras o frases descriptivas]
DispensadorEfectivo	empieza cada día cargado con 500 billetes de \$20
RanuraDeposito	[no hay palabras o frases descriptivas]

Figura 14: Palabras y frases descriptivas del documento de requerimientos del ATM

La figura 14 nos conduce a crear un atributo de la clase ATM. Esta clase mantiene información acerca del estado del ATM. La frase “el usuario es autenticado” describe un estado del ATM (más adelante hablaremos con detalle sobre los estados), por lo que incluimos **usuarioAutenticado** como un **atributo Boolean** (es decir, un atributo que tiene un valor de **true** o **false**) en la clase ATM. Observe que el atributo tipo **Boolean** en UML es equivalente al tipo **boolean** en Java. Este atributo indica si el ATM autenticó con éxito al usuario actual o no; **usuarioAutenticado** debe ser **true** para que el sistema permita al usuario realizar transacciones y acceder a la información de la cuenta. Este atributo nos ayuda a cerciorarnos de la seguridad de los datos en el sistema.

Las clases **SolicitudSaldo**, **Retiro** y **Deposito** comparten un atributo. Cada transacción requiere un “número de cuenta” que corresponde a la cuenta del usuario que realiza la transacción. Asignamos el atributo entero **numeroCuenta** a cada clase de transacción para identificar la cuenta a la que se aplica un objeto de la clase.

Las palabras y frases descriptivas en el documento de requerimientos también sugieren ciertas diferencias en los atributos requeridos por cada clase de transacción. El documento de requerimientos indica que para retirar efectivo o depositar fondos, los usuarios deben introducir un “monto” específico de dinero para retirar o depositar,



respectivamente. Por ende, asignamos a las clases **Retiro** y **Deposito** un atributo llamado **monto** para almacenar el valor suministrado por el usuario. Los montos de dinero relacionados con un retiro y un depósito son características que definen estas transacciones, que el sistema requiere para que se lleven a cabo. Sin embargo, la clase **SolicitudSaldo** no necesita datos adicionales para realizar su tarea; sólo requiere un número de cuenta para indicar la cuenta cuyo saldo hay que obtener.

La clase **Cuenta** tiene varios atributos. El documento de requerimientos establece que cada cuenta de banco tiene un “número de cuenta” y un “NIP”, que el sistema utiliza para identificar las cuentas y autenticar a los usuarios. A la clase **Cuenta** le asignamos dos atributos enteros: **numeroCuenta** y **nip**. El documento de requerimientos también especifica que una cuenta debe mantener un “saldo” del monto de dinero que hay en la cuenta, y que el dinero que el usuario deposita no estará disponible para su retiro sino hasta que el banco verifique la cantidad de efectivo en el sobre de depósito y cualquier cheque que contenga. Sin embargo, una cuenta debe registrar de todas formas el monto de dinero que deposita un usuario. Por lo tanto, decidimos que una cuenta debe representar un saldo utilizando dos atributos: **saldoDisponible** y **saldoTotal**. El atributo **saldoDisponible** rastrea el monto de dinero que un usuario puede retirar de la cuenta. El atributo **saldoTotal** se refiere al monto total de dinero que el usuario tiene “en depósito” (es decir, el monto de dinero disponible, más el monto de depósitos en efectivo o la cantidad de cheques esperando a ser verificados). Por ejemplo, suponga que un usuario del ATM deposita \$50.00 en efectivo, en una cuenta vacía. El atributo **saldoTotal** se incrementaría a \$50.00 para registrar el depósito, pero el **saldoDisponible** permanecería en \$0. [Nota: estamos suponiendo que el banco actualiza el atributo **saldoDisponible** de una **Cuenta** poco después de que se realiza la transacción del ATM, en respuesta a la confirmación de que se encontró un monto equivalente a \$50.00 en efectivo o cheques en el sobre de depósito. Asumimos que esta actualización se realiza a través de una transacción que realiza el empleado del banco mediante el uso de un sistema bancario distinto al del ATM. Por ende, no hablaremos sobre esta transacción en nuestro ejemplo práctico].

La clase **DispensadorEfectivo** tiene un atributo. El documento de requerimientos establece que el dispensador de efectivo “empieza cada día cargado con 500 billetes de \$20”. El dispensador de efectivo debe llevar el registro del número de billetes que contiene para determinar si hay suficiente efectivo disponible para satisfacer la demanda de los retiros. Asignamos a la clase **DispensadorEfectivo** el atributo entero **conteo**, el cual se establece al principio en 500.

Para los verdaderos problemas en la industria, no existe garantía alguna de que el documento de requerimientos será lo suficientemente robusto y preciso como para que el diseñador de sistemas orientados a objetos determine todos los atributos, o inclusive todas las clases. La necesidad de clases, atributos y comportamientos adicionales puede irse aclarando a medida que avance el proceso de diseño. A medida que progresemos a través de este ejemplo práctico, nosotros también seguiremos agregando, modificando y eliminando información acerca de las clases en nuestro sistema.



Modelado de los atributos

El diagrama de clases de la figura 15 enlista algunos de los atributos para las clases en nuestro sistema; las palabras y frases descriptivas en la figura 14 nos llevan a identificar estos atributos.

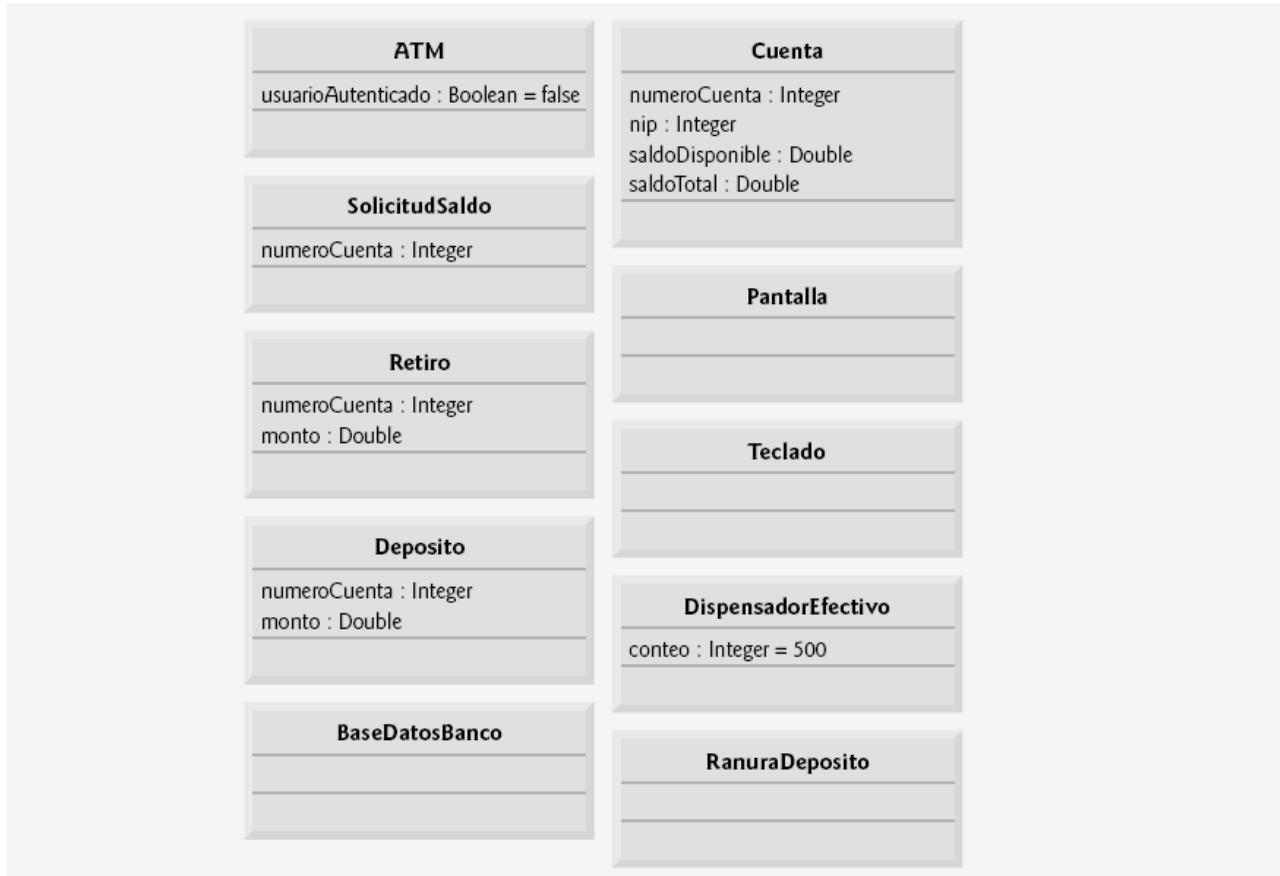


Figura 15: Clases con atributos

Por cuestión de simpleza, la figura 15 no muestra las asociaciones entre las clases; en la figura 11 mostramos estas asociaciones. Ésta es una práctica común de los diseñadores de sistemas, a la hora de desarrollar los diseños. Como vimos anteriormente, en UML los atributos de una clase se colocan en el compartimiento intermedio del rectángulo de la clase. Listamos el nombre de cada atributo y su tipo, separados por un signo de dos puntos (:), seguido en algunos casos de un signo de igual (=) y de un valor inicial.

Considere el atributo `usuarioAutenticado` de la clase `ATM`:

`usuarioAutenticado : Boolean = false`

La declaración de este atributo contiene tres informaciones acerca del atributo. El **nombre del atributo** es `usuarioAutenticado`. El **tipo del atributo** es `Boolean`. En Java, un atributo puede representarse mediante un tipo primitivo, como `boolean`, `int` o `double`, o por un tipo de referencia como una clase. Hemos optado por modelar sólo los atributos de tipo primitivo en la figura 15; en breve hablaremos sobre el razonamiento detrás de esta decisión. [Nota: los tipos de los atributos en la figura 15 están en notación de UML. Asociaremos los tipos `Boolean`, `Integer` y `Double` en el diagrama de UML con los tipos primitivos `boolean`, `int` y `double` en Java, respectivamente].



También podemos indicar un valor inicial para un atributo. El atributo `usuarioAutenticado` en la clase `ATM` tiene un valor inicial de `false`. Esto indica que al principio el sistema no considera que el usuario está autenticado. Si no se especifica un valor inicial para un atributo, sólo se muestran su nombre y tipo (separados por dos puntos). Por ejemplo, el atributo `numeroCuenta` de la clase `SolicitudSaldo` es un entero. Aquí no mostramos un valor inicial, ya que el valor de este atributo es un número que todavía no conocemos. Este número se determinará en tiempo de ejecución, con base en el número de cuenta introducido por el usuario actual del ATM.

La figura 15 no incluye atributos para las clases `Pantalla`, `Teclado` y `RanuraDepósito`. Éstos son componentes importantes de nuestro sistema, para los cuales nuestro proceso de diseño aún no ha revelado ningún atributo. No obstante, tal vez descubramos algunos en las fases restantes de diseño, o cuando implementemos estas clases en Java. Esto es perfectamente normal.

Observe que la figura 15 tampoco incluye atributos para la clase `BaseDatosBanco`. Como ya vimos, en Java los atributos pueden representarse mediante los tipos primitivos o los tipos por referencia. Hemos optado por incluir sólo los atributos de tipo primitivo en el diagrama de clases de la figura 15 (y en los diagramas de clases similares a lo largo del ejemplo práctico). Un atributo de tipo por referencia se modela con más claridad como una asociación (en particular, una composición) entre la clase que contiene la referencia y la clase del objeto al que apunta la referencia. Por ejemplo, el diagrama de clases de la figura 11 indica que la clase `BaseDatosBanco` participa en una relación de composición con cero o más objetos `Cuenta`. De esta composición podemos determinar que, cuando implementemos el sistema ATM en Java, tendremos que crear un atributo de la clase `BaseDatosBanco` para almacenar cero o más objetos `Cuenta`. De manera similar, podemos determinar los atributos de tipo por referencia de la clase `ATM` que correspondan a sus relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDepósito`. Estos atributos basados en composiciones serían redundantes si los modeláramos en la figura 15, ya que las composiciones modeladas en la figura 11 transmiten de antemano el hecho de que la base de datos contiene información acerca de cero o más cuentas, y que un ATM está compuesto por una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos. Por lo general, los desarrolladores de software modelan estas relaciones de todo/parte como asociaciones de composición, en vez de modelarlas como atributos requeridos para implementar las relaciones.

El diagrama de clases de la figura 15 proporciona una base sólida para la estructura de nuestro modelo, pero no está completo. A continuación, identificaremos los estados y las actividades de los objetos en el modelo, y luego identificaremos las operaciones que realizan los objetos. A medida que presentemos más acerca de UML y del diseño orientado a objetos, continuaremos reforzando la estructura de nuestro modelo.



Ejercicios de autoevaluación

1. Por lo general, identificamos los atributos de las clases en nuestro sistema mediante el análisis de _____ en el documento de requerimientos.
 - a) Los sustantivos y las frases nominales.
 - b) Las palabras y frases descriptivas.
 - c) Los verbos y las frases verbales.
 - d) Todo lo anterior.
2. ¿Cuál de los siguientes no es un atributo de un aeroplano?
 - a) Longitud.
 - b) Envergadura.
 - c) Volar.
 - d) Número de asientos.
3. Describa el significado de la siguiente declaración de un atributo de la clase **DispensadorEfectivo** en el diagrama de clases de la figura 15:

`conteo : Integer = 500`

Respuestas a los ejercicios de autoevaluación

1. b.
2. c. Volar es una operación o comportamiento de un aeroplano, no un atributo.
3. Esta declaración indica que el atributo **conteo** es de tipo **Integer**, con un valor inicial de 500. Este atributo lleva la cuenta del número de billetes disponibles en el **DispensadorEfectivo**, en cualquier momento dado.

5. CÓMO IDENTIFICAR LOS ESTADOS Y ACTIVIDADES DE LOS OBJETOS

Cada objeto en un sistema pasa a través de una serie de estados. El estado actual de un objeto se indica mediante los valores de los atributos del objeto en cualquier momento dado. Los **diagramas de máquina de estado** (que se conocen comúnmente como **diagramas de estado**) modelan varios estados de un objeto y muestran bajo qué circunstancias el objeto cambia de estado. A diferencia de los diagramas de clases, que se enfocan principalmente en la estructura del sistema, los diagramas de estado modelan parte del comportamiento del sistema.

La figura 16 es un diagrama de estado simple que modela algunos de los estados de un objeto de la clase ATM. UML representa a cada estado en un diagrama de estado como un **rectángulo redondeado** con el nombre del estado dentro de éste. Un **círculo relleno** con una punta de flecha designa el **estado inicial**. Recuerde que en el diagrama de clases de la figura 15 modelamos esta información de estado como el atributo Boolean de nombre **usuarioAutenticado**. Este atributo se inicializa en **false**, o en el estado “Usuario no autenticado”, de acuerdo con el diagrama de estado.

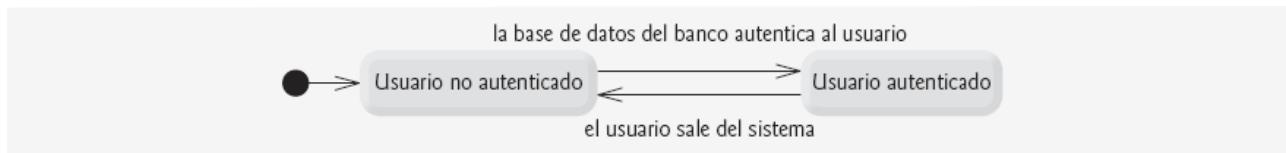


Figura 16: Diagrama de estado para el objeto ATM.

Las flechas indican las **transiciones** entre los estados. Un objeto puede pasar de un estado a otro, en respuesta a los diversos eventos que ocurren en el sistema. El nombre o la descripción del evento que ocasiona una transición se escribe cerca de la línea que corresponde a esa transición. Por ejemplo, el objeto ATM cambia del estado “Usuario no autenticado” al estado “Usuario autenticado”, una vez que la base de datos autentica al usuario. En el documento de requerimientos vimos que para autenticar a un usuario, la base de datos compara el número de cuenta y el NIP introducidos por el usuario con los de la cuenta correspondiente en la base de datos. Si la base de datos indica que el usuario ha introducido un número de cuenta válido y el NIP correcto, el objeto ATM pasa al estado “Usuario autenticado” y cambia su atributo `usuarioAutenticado` al valor `true`. Cuando el usuario sale del sistema al seleccionar la opción “salir” del menú principal, el objeto ATM regresa al estado “Usuario no autenticado”.

Diagramas de actividad

Al igual que un diagrama de estado, un diagrama de actividad modela los aspectos del comportamiento de un sistema. A diferencia de un diagrama de estado, un diagrama de actividad modela el **flujo de trabajo** (secuencia de objetos) de un objeto durante la ejecución de un programa. Un diagrama de actividad modela las **acciones** a realizar y en qué orden las realizará el objeto. El diagrama de actividad de la figura 17 modela las acciones involucradas en la ejecución de una transacción de solicitud de saldo. Asumimos que ya se ha inicializado un objeto `SolicitudSaldo` y que ya se le ha asignado un número de cuenta válido (el del usuario actual), por lo que el objeto sabe qué saldo extraer de la base de datos. El diagrama incluye las acciones que ocurren después de que el usuario selecciona la opción de solicitud de saldo del menú principal y antes de que el ATM devuelva al usuario al menú principal; un objeto `SolicitudSaldo` no realiza ni inicia estas acciones, por lo que no las modelamos aquí.



Figura 17: Diagrama de actividad para un objeto `SolicitudSaldo`.



El diagrama empieza extrayendo de la base de datos el saldo de la cuenta. Después, **SolicitudSaldo** muestra el saldo en la pantalla. Esta acción completa la ejecución de la transacción. Recuerde que hemos optado por representar el saldo de una cuenta como los atributos **saldoDisponible** y **saldoTotal** de la clase **Cuenta**, por lo que las acciones que se modelan en la figura 17 hacen referencia a la obtención y visualización de ambos atributos del saldo.

UML representa una acción en un diagrama de actividad como un estado de acción, el cual se modela mediante un rectángulo en el que sus lados izquierdo y derecho se sustituyen por arcos hacia fuera. Cada estado de acción contiene una expresión de acción; por ejemplo, “obtener de la base de datos el saldo de la cuenta”; eso especifica una acción a realizar. Una flecha conecta dos estados de acción, con lo cual indica el orden en el que ocurren las acciones representadas por los estados de acción. El círculo relleno (en la parte superior de la figura 17) representa el estado inicial de la actividad: el inicio del flujo de trabajo antes de que el objeto realice las acciones modeladas. En este caso, la transacción primero ejecuta la expresión de acción “obtener de la base de datos el saldo de la cuenta”. Después, la transacción muestra ambos saldos en la pantalla. El círculo relleno encerrado en un círculo sin relleno (en la parte inferior de la figura 17) representa el estado final: el fin del flujo de trabajo una vez que el objeto realiza las acciones modeladas. Utilizamos diagramas de actividad de UML para ilustrar el flujo de control para las instrucciones de control que ya presentamos.

La figura 18 muestra un diagrama de actividad para una transacción de retiro. Asumimos que ya se ha asignado un número de cuenta válido a un objeto **Retiro**. No modelaremos al usuario seleccionando la opción de retiro del menú principal ni al ATM devolviendo al usuario al menú principal, ya que estas acciones no las realiza un objeto **Retiro**. La transacción primero muestra un menú de montos estándar de retiro (que se muestra en la figura 3) y una opción para cancelar la transacción. Después la transacción recibe una selección del menú de parte del usuario. Ahora el flujo de actividad llega a una decisión (una bifurcación indicada por el pequeño símbolo de rombo). [Nota: antiguamente, una decisión se conocía como una bifurcación]. Este punto determina la siguiente acción con base en la condición de guardia asociada (entre corchetes, enseguida de la transición), que indica que la transición ocurre si se cumple esta condición de guardia. Si el usuario cancela la transacción al elegir la opción “cancelar” del menú, el flujo de actividad salta inmediatamente al siguiente estado. Observe la fusión (indicada mediante el pequeño símbolo de rombo), en donde el flujo de actividad de cancelación se combina con el flujo de actividad principal, antes de llegar al estado final de la actividad. Si el usuario selecciona un monto de retiro del menú, **Retiro** establece **monto** (un atributo modelado originalmente en la figura 15) al valor elegido por el usuario.

Después de establecer el monto de retiro, la transacción obtiene el saldo disponible de la cuenta del usuario (es decir, el atributo **saldoDisponible** del objeto **Cuenta** del usuario) de la base de datos. Después el flujo de actividad llega a otra decisión. Si el monto de retiro solicitado excede al saldo disponible del usuario, el sistema muestra un mensaje de error apropiado, en el cual informa al usuario sobre el problema y después regresa al principio del diagrama de actividad, y pide al usuario que introduzca un nuevo monto. Si el monto de retiro solicitado es menor o igual al saldo disponible del usuario, la transacción continúa. A continuación, la transacción evalúa si

el dispensador de efectivo tiene suficiente efectivo para satisfacer la solicitud de retiro. Si éste no es el caso, la transacción muestra un mensaje de error apropiado, después regresa al principio del diagrama de actividad y pide al usuario que seleccione un nuevo monto. Si hay suficiente efectivo disponible, la transacción interactúa con la base de datos para cargar el monto retirado de la cuenta del usuario (es decir, restar el monto tanto del atributo `saldoDisponible` como del atributo `saldoTotal` del objeto Cuenta del usuario). Después la transacción entrega el monto deseado de efectivo e instruye al usuario para que lo tome. Por último, el flujo de actividad se fusiona con el flujo de actividad de cancelación antes de llegar al estado final.

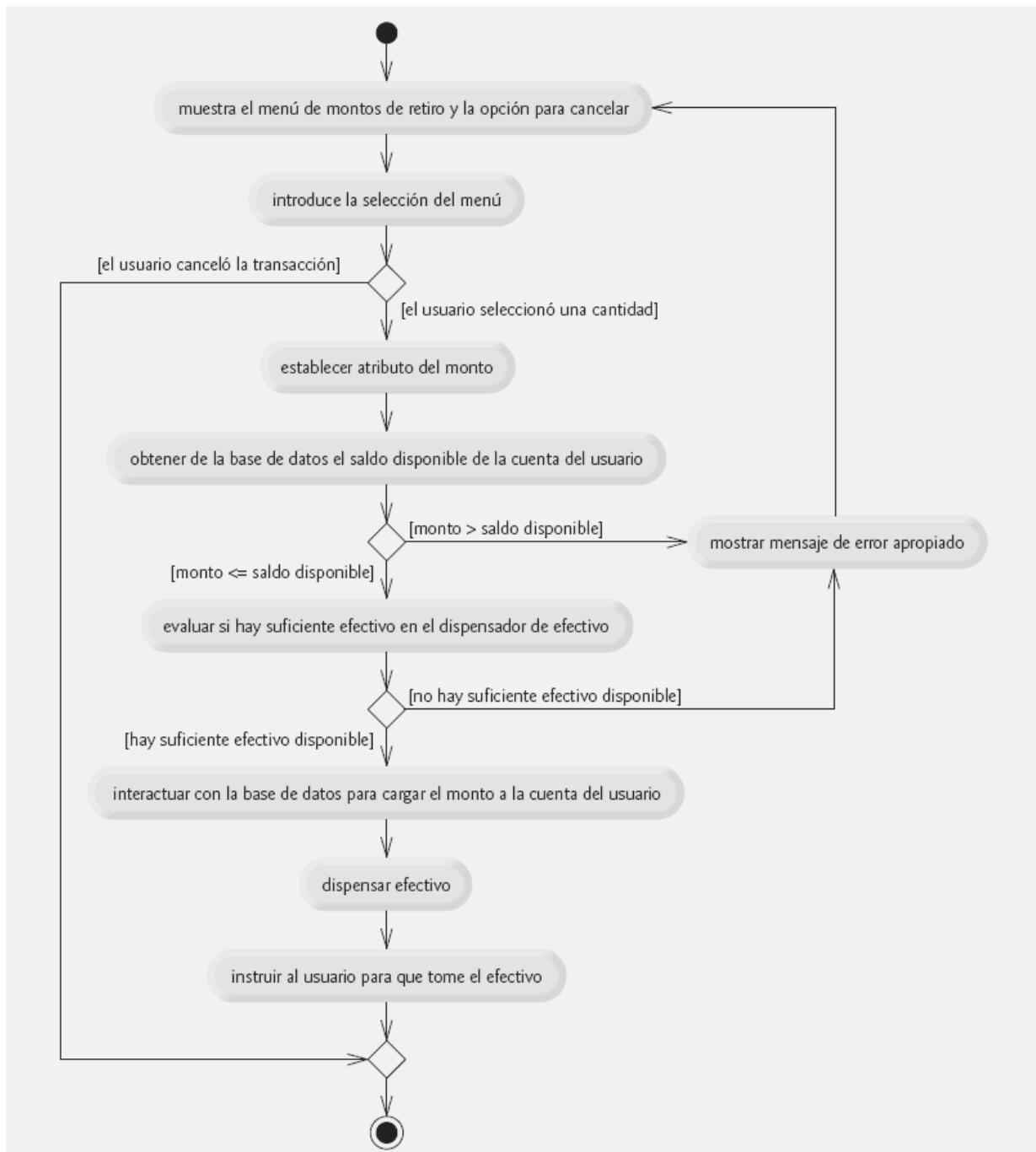


Figura 18: Diagrama de actividad para una transacción de retiro.



Hemos llevado a cabo los primeros pasos para modelar el comportamiento del sistema ATM y hemos mostrado cómo participan los atributos de un objeto para realizar las actividades del mismo. A continuación, investigaremos los comportamientos para todas las clases, de manera que obtengamos una interpretación más precisa del comportamiento del sistema, al “completar” los terceros compartimientos de las clases en nuestro diagrama de clases.

Ejercicios de autoevaluación

1. Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: los diagramas de estado modelan los aspectos estructurales de un sistema.
2. Un diagrama de actividad modela las (los) _____ que realiza un objeto y el orden en el que las(los) realiza.
 - a) acciones
 - b) atributos
 - c) estados
 - d) transiciones de estado
3. Con base en el documento de requerimientos, cree un diagrama de actividad para una transacción de depósito.

Respuestas a los ejercicios de autoevaluación

1. Falso. Los diagramas de estado modelan parte del comportamiento del sistema.
2. a.
3. La figura 19 presenta un diagrama de actividad para una transacción de depósito. El diagrama modela las acciones que ocurren una vez que el usuario selecciona la opción de depósito del menú principal, y antes de que el ATM regrese al usuario al menú principal. Recuerde que una parte del proceso de recibir un monto de depósito de parte del usuario implica convertir un número entero de centavos a una cantidad en dólares. Recuerde también que para acreditar un monto de depósito a una cuenta sólo hay que incrementar el atributo `saldoTotal` del objeto `Cuenta` del usuario. El banco actualiza el atributo `saldoDisponible` del objeto `Cuenta` del usuario sólo después de confirmar el monto de efectivo en el sobre de depósito y después de verificar los cheques que haya incluido; esto ocurre en forma independiente del sistema ATM.

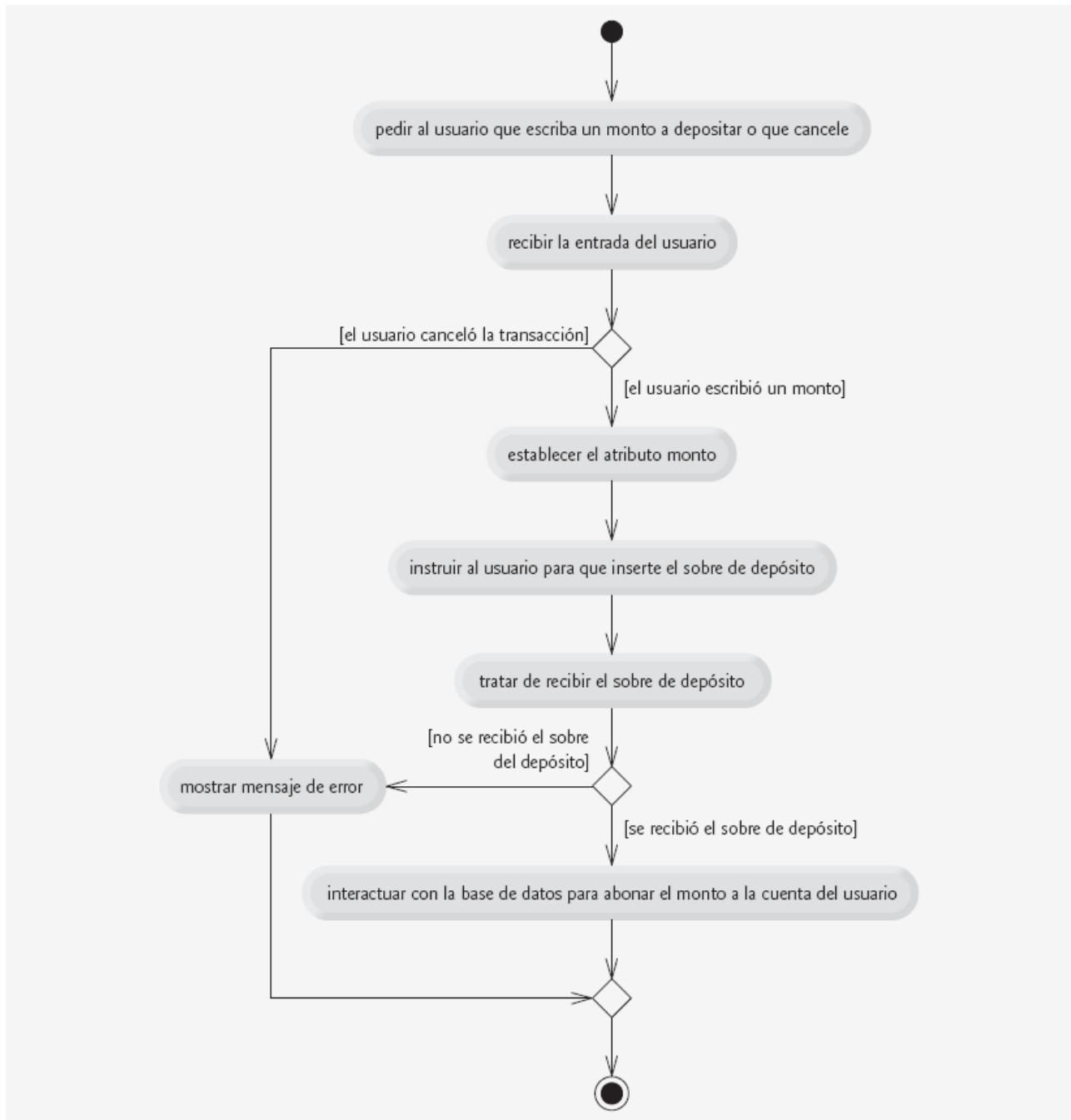


Figura 19: Diagrama de actividad para una transacción de depósito.

6. IDENTIFICACIÓN DE LAS OPERACIONES DE LAS CLASES

Una operación es un servicio que proporcionan los objetos de una clase a los clientes (usuarios) de esa clase. Considere las operaciones de algunos objetos reales. Las operaciones de un radio incluyen el sintonizar su estación y ajustar su volumen (que, por lo general, lo hace una persona que ajusta los controles del radio). Las operaciones de un automóvil incluyen acelerar (operación invocada por el conductor cuando oprime el pedal del acelerador), desacelerar (operación invocada por el conductor cuando oprime el pedal del freno o cuando suelta el pedal del acelerador), dar vuelta y cambiar velocidades. Los objetos de software también pueden ofrecer operaciones; por ejemplo,



un objeto de gráficos de software podría ofrecer operaciones para dibujar un círculo, dibujar una línea, dibujar un cuadrado, etcétera. Un objeto de software de hoja de cálculo podría ofrecer operaciones como imprimir la hoja de cálculo, totalizar los elementos en una fila o columna, y graficar la información de la hoja de cálculo como un gráfico de barras o de pastel.

Podemos derivar muchas de las operaciones de cada clase mediante un análisis de los verbos y las frases verbales clave en el documento de requerimientos. Después relacionamos cada una de ellas con las clases específicas en nuestro sistema (figura 20). Las frases verbales en la figura 20 nos ayudan a determinar las operaciones de cada clase.

Clase	Verbos y frases verbales
ATM	ejecuta transacciones financieras
SolicitudSaldo	[ninguna en el documento de requerimientos]
Retiro	[ninguna en el documento de requerimientos]
Deposito	[ninguna en el documento de requerimientos]
BaseDatosBanco	autentica a un usuario, obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
Cuenta	obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
Pantalla	muestra un mensaje al usuario
Teclado	recibe entrada numérica del usuario
DispensadorEfectivo	dispensa efectivo, indica si contiene suficiente efectivo para satisfacer una solicitud de retiro
RanuraDeposito	recibe un sobre de depósito

Figura 20: Verbos y frases verbales para cada clase en el sistema ATM.

Modelar las operaciones

Para identificar las operaciones, analizamos las frases verbales que se listan para cada clase en la figura 20. La frase “ejecuta transacciones financieras” asociada con la clase **ATM** implica que esta clase instruye a las transacciones a que se ejecuten. Por lo tanto, cada una de las clases **SolicitudSaldo**, **Retiro** y **Deposito** necesitan una operación para proporcionar este servicio al ATM. Colocamos esta operación (que hemos nombrado **ejecutar**) en el tercer compartimiento de las tres clases de transacciones en el diagrama de clases actualizado de la figura 21. Durante una sesión con el ATM, el objeto **ATM** invocará estas operaciones de transacciones, según sea necesario.

Para representar las operaciones (que se implementan en forma de métodos en Java), UML lista el nombre de la operación, seguido de una lista separada por comas de parámetros entre paréntesis, un signo de dos puntos y el tipo de valor de retorno:

nombreOperación(parámetro1, parámetro2, ..., parámetroN) : tipo de valor de retorno



Cada parámetro en la lista separada por comas consiste en un nombre de parámetro, seguido de un signo de dos puntos y del tipo del parámetro:

nombreParámetro : tipoParámetro

Por el momento, no listamos los parámetros de nuestras operaciones; en breve identificaremos y modelaremos los parámetros de algunas de las operaciones. Para algunas de estas operaciones no conocemos todavía los tipos de valores de retorno, por lo que también las omitiremos del diagrama. Estas omisiones son perfectamente normales en este punto. A medida que avancemos en nuestro proceso de diseño e implementación, agregaremos el resto de los tipos de valores de retorno.

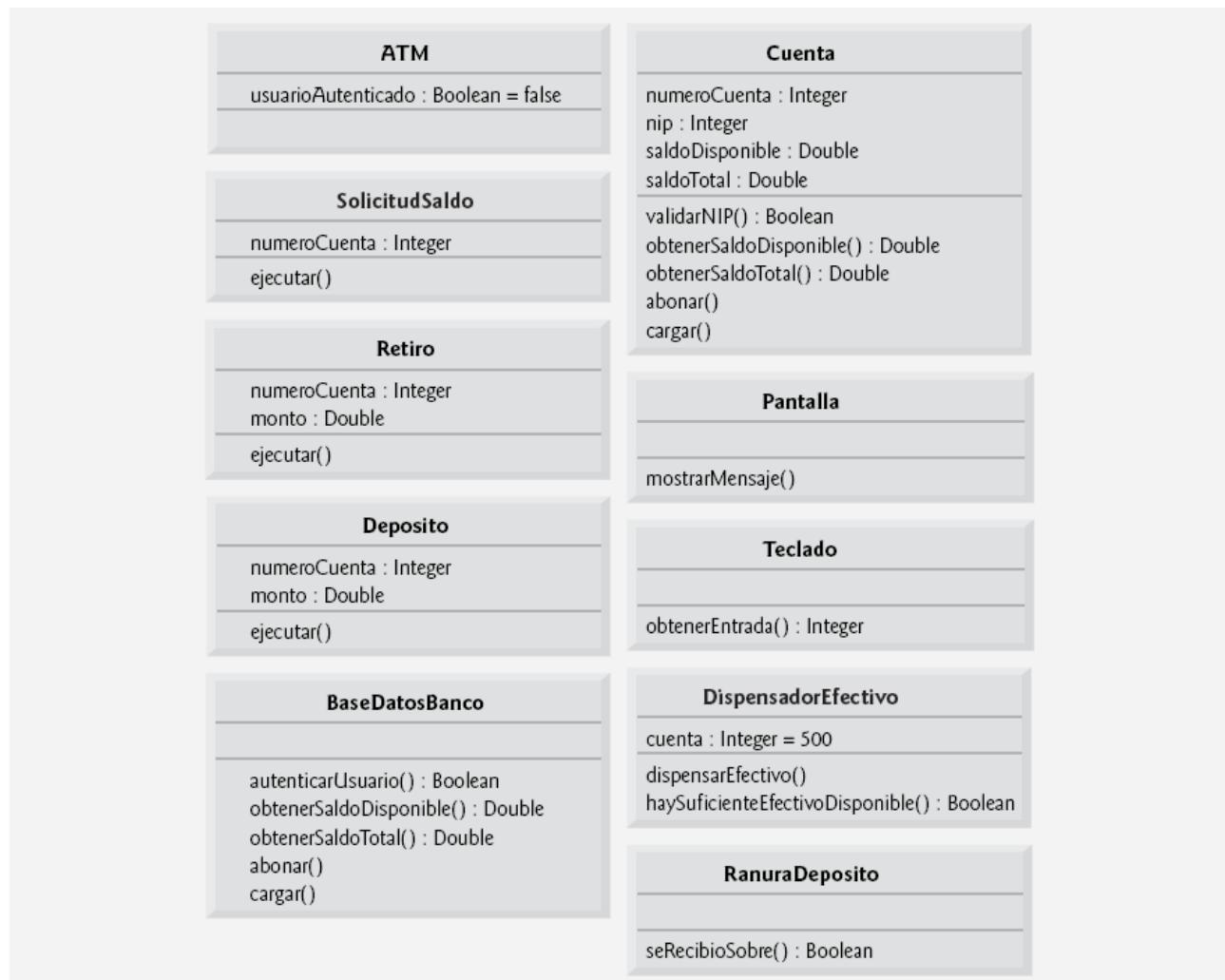


Figura 21: Las clases en el sistema ATM, con atributos y operaciones.

La figura 20 lista la frase “autentica a un usuario” enseguida de la clase `BaseDatosBanco`; la base de datos es el objeto que contiene la información necesaria de la cuenta para determinar si el número de cuenta y el NIP introducidos por un usuario concuerdan con los de una cuenta en el banco. Por lo tanto, la clase `BaseDatosBanco` necesita una operación que proporcione un servicio de autenticación al ATM. Colocamos la operación `autenticarUsuario` en el tercer compartimiento de la clase `BaseDatosBanco` (figura 21). No obstante, un objeto de la clase `Cuenta` y no de la clase



BaseDatosBanco es el que almacena el número de cuenta y el NIP a los que se debe acceder para autenticar a un usuario, por lo que la clase **Cuenta** debe proporcionar un servicio para validar un NIP obtenido como entrada del usuario, y compararlo con un NIP almacenado en un objeto **Cuenta**. Por ende, agregamos una operación **validarNIP** a la clase **Cuenta**. Observe que especificamos un tipo de retorno **Boolean** para las operaciones **autenticarUsuario** y **validarNIP**. Cada operación devuelve un valor que indica que la operación tuvo éxito al realizar su tarea (es decir, un valor de retorno **true**) o que no tuvo éxito (es decir, un valor de retorno **false**).

La figura 20 lista varias frases verbales adicionales para la clase **BaseDatosBanco**: “extrae el saldo de una cuenta”, “abona un monto de depósito a una cuenta” y “carga un monto de retiro a una cuenta”. Al igual que “autentica a un usuario”, estas frases restantes se refieren a los servicios que debe proporcionar la base de datos al ATM, ya que la base de datos almacena todos los datos de las cuentas que se utilizan para autenticar a un usuario y realizar transacciones con el ATM. No obstante, los objetos de la clase **Cuenta** son los que en realidad realizan las operaciones a las que se refieren estas frases. Por ello, asignamos una operación tanto a la clase **BaseDatosBanco** como a la clase **Cuenta**, que corresponda con cada una de estas frases. Antes ya vimos que, como una cuenta de banco contiene información delicada, no permitimos que el ATM acceda a las cuentas en forma directa. La base de datos actúa como un intermediario entre el ATM y los datos de la cuenta, evitando el acceso no autorizado. Como veremos más adelante, la clase **ATM** invoca las operaciones de la clase **BaseDatosBanco**, cada una de las cuales a su vez invoca a la operación con el mismo nombre en la clase **Cuenta**.

La frase “obtiene el saldo de una cuenta” sugiere que las clases **BaseDatosBanco** y **Cuenta** necesitan una operación **obtenerSaldo**. Sin embargo, recuerde que creamos dos atributos en la clase **Cuenta** para representar un saldo: **saldoDisponible** y **saldoTotal**. Una solicitud de saldo requiere el acceso a estos dos atributos del saldo, de manera que pueda mostrarlos al usuario, pero un retiro sólo requiere verificar el valor de **saldoDisponible**. Para permitir que los objetos en el sistema obtengan cada atributo de saldo en forma individual, agregamos las operaciones **obtenerSaldoDisponible** y **obtenerSaldoTotal** al tercer compartimiento de las clases **BaseDatosBanco** y **Cuenta** (figura 21). Específicamos un tipo de retorno **Double** para estas operaciones, debido a que los atributos de los saldos que van a obtener son de tipo **Double**.

Las frases “abona un monto de depósito a una cuenta” y “carga un monto de retiro a una cuenta” indican que las clases **BaseDatosBanco** y **Cuenta** deben realizar operaciones para actualizar una cuenta durante un depósito y un retiro, respectivamente. Por lo tanto, asignamos las operaciones **abonar** y **cargar** a las clases **BaseDatosBanco** y **Cuenta**. Tal vez recuerde que cuando se abona a una cuenta (como en un depósito) se suma un monto sólo al atributo **saldoTotal**. Por otro lado, cuando se carga a una cuenta (como en un retiro) se resta el monto tanto del saldo total como del saldo disponible. Ocultamos estos detalles de implementación dentro de la clase **Cuenta**. Éste es un buen ejemplo de encapsulamiento y ocultamiento de información.



Si éste fuera un sistema ATM real, las clases `BaseDatosBanco` y `Cuenta` también proporcionarían un conjunto de operaciones para permitir que otro sistema bancario actualizara el saldo de la cuenta de un usuario después de confirmar o rechazar todo, o parte de, un depósito. Por ejemplo, la operación `confirmarMontoDeposito` sumaría un monto al atributo `saldoDisponible`, y haría que los fondos depositados estuvieran disponibles para retirarlos. La operación `rechazarMontoDeposito` restaría un monto al atributo `saldoTotal` para indicar que un monto especificado, que se había depositado recientemente a través del ATM y se había sumado al `saldoTotal`, no se encontró en el sobre de depósito. El banco invocaría esta operación después de determinar que el usuario no incluyó el monto correcto de efectivo o que algún cheque no fue validado (es decir, que “rebotó”). Aunque al agregar estas operaciones nuestro sistema estaría más completo, no las incluiremos en nuestros diagramas de clases ni en nuestra implementación, ya que se encuentran más allá del alcance de este ejemplo práctico.

La clase `Pantalla` “muestra un mensaje al usuario” en diversos momentos durante una sesión con el ATM. Toda la salida visual se produce a través de la pantalla del ATM. El documento de requerimientos describe muchos tipos de mensajes (por ejemplo, un mensaje de bienvenida, un mensaje de error, un mensaje de agradecimiento) que la pantalla muestra al usuario. El documento de requerimientos también indica que la pantalla muestra *indicadores* y *menús* al usuario. No obstante, un *indicador* es en realidad sólo un mensaje que describe lo que el usuario debe introducir a continuación, y un *menú* es en esencia un tipo de indicador que consiste en una serie de mensajes (es decir, las opciones del menú) que se muestran en forma consecutiva. Por lo tanto, en vez de asignar a la clase `Pantalla` una operación individual para mostrar cada tipo de mensaje, *indicador* y *menú*, basta con crear una operación que pueda mostrar cualquier mensaje especificado por un parámetro. Colocamos esta operación (`mostrarMensaje`) en el tercer compartimiento de la clase `Pantalla` en nuestro diagrama de clases (figura 21). Observe que no nos preocupa el parámetro de esta operación en estos momentos; lo modelaremos más adelante.

De la frase “recibe entrada numérica del usuario” listada por la clase `Teclado` en la figura 20, podemos concluir que la clase `Teclado` debe realizar una operación `obtenerEntrada`. A diferencia del teclado de una computadora, el teclado del ATM sólo contiene los números del 0 al 9, por lo cual especificamos que esta operación devuelve un valor entero. Si recuerda, en el documento de requerimientos vimos que en distintas situaciones, tal vez se requiera que el usuario introduzca un tipo distinto de número (por ejemplo, un número de cuenta, un NIP, el número de una opción del menú, un monto de depósito como número de centavos). La clase `Teclado` sólo obtiene un valor numérico para un cliente de la clase; no determina si el valor cumple con algún criterio específico. Cualquier clase que utilice esta operación debe verificar que el usuario haya introducido un número apropiado según el caso, y después debe responder de manera acorde (por ejemplo, mostrar un mensaje de error a través de la clase `Pantalla`). [Nota: cuando implementemos el sistema, simularemos el teclado del ATM con el teclado de una computadora y, por cuestión de simpleza, asumiremos que el usuario no escribirá datos de entrada que no sean números, usando las teclas en el teclado de la computadora que no aparezcan en el teclado del ATM].



La figura 20 lista la frase “dispensa efectivo” para la clase **DispensadorEfectivo**. Por lo tanto, creamos la operación **dispensarEfectivo** y la listamos bajo la clase **DispensadorEfectivo** en la figura 21. La clase **DispensadorEfectivo** también “indica si contiene suficiente efectivo para satisfacer una solicitud de retiro”. Para esto incluimos a **haySuficienteEfectivoDisponible**, una operación que devuelve un valor de tipo Boolean de UML, en la clase **DispensadorEfectivo**. La figura 20 también lista la frase “recibe un sobre de depósito” para la clase **RanuraDeposito**. La ranura de depósito debe indicar si recibió un sobre, por lo que colocamos una operación **seRecibioSobre**, la cual devuelve un valor Boolean, en el tercer compartimiento de la clase **RanuraDeposito**. [Nota: es muy probable que una ranura de depósito de hardware real envíe una señal al ATM para indicarle que se recibió un sobre. No obstante, simularemos este comportamiento con una operación en la clase **RanuraDeposito**, que la clase **ATM** pueda invocar para averiguar si la ranura de depósito recibió un sobre].

No listamos ninguna operación para la clase **ATM** en este momento. Todavía no sabemos de algún servicio que proporcione la clase **ATM** a otras clases en el sistema. No obstante, cuando implementemos el sistema en código de Java, tal vez emergan las operaciones de esta clase junto con las operaciones adicionales de las demás clases en el sistema.

Identificar y modelar los parámetros de operación

Hasta ahora no nos hemos preocupado por los parámetros de nuestras operaciones; sólo hemos tratado de obtener una comprensión básica de las operaciones de cada clase. Ahora daremos un vistazo más de cerca a varios parámetros de operación. Para identificar los parámetros de una operación, analizamos qué datos requiere la operación para realizar su tarea asignada.

Considere la operación **autenticarUsuario** de la clase **BaseDatosBanco**. Para autenticar a un usuario, esta operación debe conocer el número de cuenta y el NIP que suministra el usuario. Por lo tanto, especificamos que la operación **autenticarUsuario** debe recibir los parámetros enteros **numeroCuentaUsuario** y **nipUsuario**, que la operación debe comparar con el número de cuenta y el NIP de un objeto **Cuenta** en la base de datos. Colocaremos después de estos nombres de parámetros la palabra **Usuario**, para evitar confusión entre los nombres de los parámetros de la operación y los nombres de los atributos que pertenecen a la clase **Cuenta**. Listamos estos parámetros en el diagrama de clases de la figura 22, el cual modela sólo a la clase **BaseDatosBanco**. [Nota: es perfectamente normal modelar sólo una clase en un diagrama de clases. En este caso lo que más nos preocupa es analizar los parámetros de esta clase específica, por lo que omitimos las demás clases. Más adelante en los diagramas de clase de este ejemplo práctico, en donde los parámetros dejarán de ser el centro de nuestra atención, los omitiremos para ahorrar espacio. No obstante, recuerde que las operaciones que se listan en estos diagramas siguen teniendo parámetros].

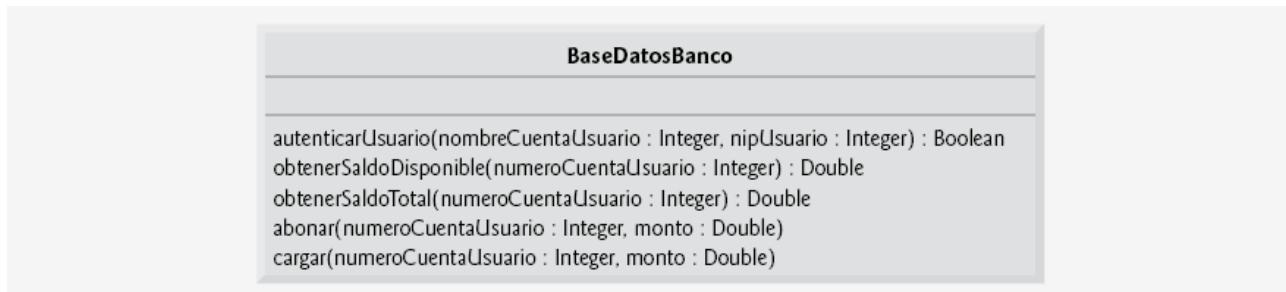


Figura 22: La clase **BaseDatosBanco** con parámetros de operación.

Recuerde que para modelar cada parámetro en una lista de parámetros separados por comas, UML lista el nombre del parámetro, seguido de un signo de dos puntos y el tipo del parámetro (en notación de UML). Así, la figura 22 especifica que la operación **autenticarUsuario** recibe dos parámetros: **numeroCuentaUsuario** y **nipUsuario**, ambos de tipo **Integer**. Cuando implementemos el sistema en Java, representaremos estos parámetros con valores **int**.

Las operaciones **obtenerSaldoDisponible**, **obtenerSaldoTotal**, **abonar** y **cargar** de la clase **BaseDatosBanco** también requieren un parámetro **nombreCuentaUsuario** para identificar la cuenta a la cual la base de datos debe aplicar las operaciones, por lo que incluimos estos parámetros en el diagrama de clases de la figura 22. Además, las operaciones **abonar** y **cargar** requieren un parámetro **Double** llamado **monto**, para especificar el monto de dinero que se abonará o cargará, respectivamente.

El diagrama de clases de la figura 23 modela los parámetros de las operaciones de la clase **Cuenta**. La operación **validarNIP** sólo requiere un parámetro **nipUsuario**, el cual contiene el NIP especificado por el usuario, que se comparará con el NIP asociado a la cuenta. Al igual que sus contrapartes en la clase **BaseDatosBanco**, las operaciones **abonar** y **cargar** en la clase **Cuenta** requieren un parámetro **Double** llamado **monto**, el cual indica la cantidad de dinero involucrada en la operación. Las operaciones **obtenerSaldoDisponible** y **obtenerSaldoTotal** en la clase **Cuenta** no requieren datos adicionales para realizar sus tareas. Observe que las operaciones de la clase **Cuenta** no requieren un parámetro de número de cuenta para diferenciar una cuenta de otra, ya que cada una de estas operaciones se puede invocar sólo en un objeto **Cuenta** específico.

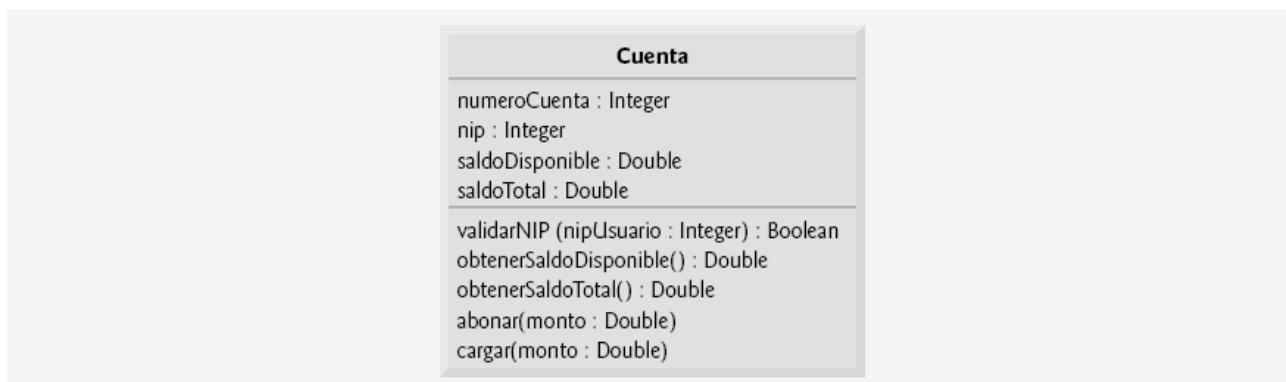


Figura 23: La clase **Cuenta** con parámetros de operación.



La figura 24 modela la clase **Pantalla** con un parámetro especificado para la operación **mostrarMensaje**. Esta operación requiere sólo un parámetro **String** llamado **mensaje**, el cual indica el texto que debe mostrarse en pantalla. Recuerde que los tipos de los parámetros que se enlistan en nuestros diagramas de clases están en notación de UML, por lo que el tipo **String** que se enumera en la figura 24 se refiere al tipo de UML. Cuando implementemos el sistema en Java, utilizaremos de hecho la clase **String** de Java para representar este parámetro.



Figura 24: La clase **Pantalla** con parámetros de operación.

El diagrama de clases de la figura 25 especifica que la operación **dispensarEfectivo** de la clase **DispensadorEfectivo** recibe un parámetro **Double** llamado **monto** para indicar el monto de efectivo (en dólares) que se dispensará al usuario. La operación **haySuficienteEfectivoDisponible** también recibe un parámetro **Double** llamado **monto** para indicar el monto de efectivo en cuestión.

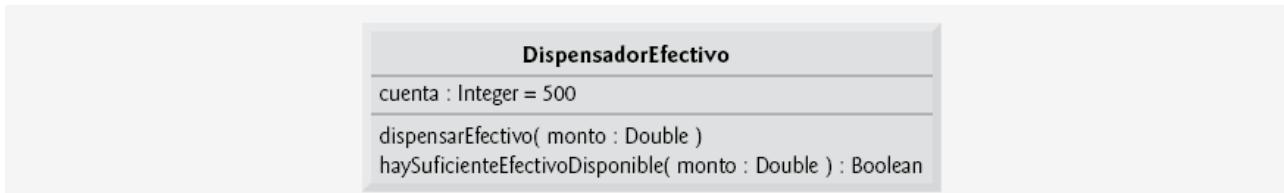


Figura 25: La clase **DispensadorEfectivo** con parámetros de operación.

Observe que no hablamos sobre los parámetros para la operación **ejecutar** de las clases **SolicitudSaldo**, **Retiro** y **Depósito**, de la operación **obtenerEntrada** de la clase **Teclado** y la operación **seRecibioSobre** de la clase **RanuraDeposito**. En este punto de nuestro proceso de diseño, no podemos determinar si estas operaciones requieren datos adicionales para realizar sus tareas, por lo que dejaremos sus listas de parámetros vacías. A medida que avancemos por el ejemplo práctico, tal vez decidamos agregar parámetros a estas operaciones.

En esta sección hemos determinado muchas de las operaciones que realizan las clases en el sistema ATM. Identificamos los parámetros y los tipos de valores de retorno de algunas operaciones. A medida que continuemos con nuestro proceso de diseño, el número de operaciones que pertenezcan a cada clase puede variar; podríamos descubrir que se necesitan nuevas operaciones o que ciertas operaciones actuales no son necesarias; y podríamos determinar que algunas de las operaciones de nuestras clases necesitan parámetros adicionales y tipos de valores de retorno distintos.



Ejercicios de autoevaluación

1. ¿Cuál de las siguientes opciones no es un comportamiento?
 - a) Leer los datos de un archivo.
 - b) Imprimir los resultados.
 - c) Imprimir texto.
 - d) Obtener la entrada del usuario.
2. Si quisiera agregar al sistema ATM una operación que devuelva el atributo `monto` de la clase `Retiro`, ¿cómo y en dónde especificaría esta operación en el diagrama de clases de la figura 21?
3. Describa el significado del siguiente listado de operaciones, el cual podría aparecer en un diagrama de clases para el diseño orientado a objetos de una calculadora:
`sumar(x : Integer, y : Integer) : Integer`

Respuestas a los ejercicios de autoevaluación

1. c.
2. Para especificar una operación que obtenga el atributo `monto` de la clase `Retiro`, se debe colocar el siguiente listado de operaciones en el (tercer) compartimiento de operaciones de la clase `Retiro`: `obtenerMonto() : Double`
3. Este listado de operaciones indica una operación llamada `sumar`, la cual recibe los enteros `x` e `y` como parámetros y devuelve un valor entero.

7. COLABORACIÓN ENTRE LOS OBJETOS

Cuando dos objetos se comunican entre sí para realizar una tarea, se dice que **colaboran** (para ello, un objeto invoca a las operaciones del otro). Una **colaboración** consiste en que un objeto de una clase envía un **mensaje** a un objeto de otra clase. En Java, los mensajes se envían mediante llamadas a métodos.

Anteriormente determinamos muchas de las operaciones de las clases en nuestro sistema. Ahora nos concentraremos en los mensajes que invocan a esas operaciones. Para identificar las colaboraciones en el sistema, regresaremos al documento de requerimientos. Recuerde que este documento especifica el rango de actividades que ocurren durante una sesión con el ATM (por ejemplo, autenticar a un usuario, realizar transacciones). Los pasos utilizados para describir cómo debe realizar el sistema cada una de estas tareas son nuestra primera indicación de las colaboraciones en nuestro sistema. A medida que avancemos, es probable que descubramos relaciones adicionales.



Identificar las colaboraciones en un sistema

Para identificar las colaboraciones en el sistema, leeremos con cuidado las secciones del documento de requerimientos que especifican lo que debe hacer el ATM para autenticar un usuario, y para realizar cada tipo de transacción. Para cada acción o paso descrito en el documento de requerimientos, decidimos qué objetos en nuestro sistema deben interactuar para lograr el resultado deseado. Identificamos un objeto como el emisor y otro como el receptor. Después seleccionamos una de las operaciones del objeto receptor (identificadas anteriormente) que el objeto emisor debe invocar para producir el comportamiento apropiado. Por ejemplo, el ATM muestra un mensaje de bienvenida cuando está inactivo. Sabemos que un objeto de la clase Pantalla muestra un mensaje al usuario a través de su operación `mostrarMensaje`. Por ende, decidimos que el sistema puede mostrar un mensaje de bienvenida si empleamos una colaboración entre el ATM y la Pantalla, en donde el ATM envía un mensaje `mostrarMensaje` a la Pantalla, invocando la operación `mostrarMensaje` de la clase Pantalla. [Nota: para evitar repetir la frase “un objeto de la clase...”, nos referiremos a cada objeto sólo utilizando su nombre de clase, precedido por un artículo (por ejemplo, “un”, “una”, “el” o “la”); por ejemplo, “el ATM” hace referencia a un objeto de la clase ATM].

Un objeto de la clase...	envía el mensaje...	a un objeto de la clase...
ATM	<code>mostrarMensaje</code>	Pantalla
	<code>obtenerEntrada</code>	Teclado
	<code>autenticarUsuario</code>	BaseDatosBanco
	<code>ejecutar</code>	SolicitudSaldo
	<code>ejecutar</code>	Retiro
	<code>ejecutar</code>	Deposito
SolicitudSaldo	<code>obtenerSaldoDisponible</code>	BaseDatosBanco
	<code>obtenerSaldoTotal</code>	BaseDatosBanco
	<code>mostrarMensaje</code>	Pantalla
Retiro	<code>MostrarMensaje</code>	Pantalla
	<code>obtenerEntrada</code>	Teclado
	<code>obtenerSaldoDisponible</code>	BaseDatosBanco
	<code>haySuficienteEfectivoDisponible</code>	DispensadorEfectivo
	<code>cargar</code>	BaseDatosBanco
	<code>dispensarEfectivo</code>	DispensadorEfectivo
Deposito	<code>mostrarMensaje</code>	Pantalla
	<code>obtenerEntrada</code>	Teclado
	<code>seRecibioSobreDeposito</code>	RanuraDeposito
	<code>abonar</code>	BaseDatosBanco
BaseDatosBanco	<code>validarNIP</code>	Cuenta
	<code>obtenerSaldoDisponible</code>	Cuenta
	<code>obtenerSaldoTotal</code>	Cuenta
	<code>cargar</code>	Cuenta
	<code>abonar</code>	Cuenta

Figura 26: Colaboraciones en el sistema ATM.



La figura 26 lista las colaboraciones que pueden derivarse del documento de requerimientos. Para cada objeto emisor, listamos las colaboraciones en el orden en el que ocurren primero durante una sesión con el ATM (es decir, el orden en el que se describen en el documento de requerimientos). Listamos cada colaboración en la que se involucre un emisor único, un mensaje y un receptor sólo una vez, aun cuando la colaboración puede ocurrir varias veces durante una sesión con el ATM. Por ejemplo, la primera fila en la figura 26 indica que el objeto **ATM** colabora con el objeto **Pantalla** cada vez que el **ATM** necesita mostrar un mensaje al usuario.

Consideraremos las colaboraciones en la figura 26. Antes de permitir que un usuario realice transacciones, el **ATM** debe pedirle que introduzca un número de cuenta y que después introduzca un NIP. Para realizar cada una de estas tareas envía un mensaje a la **Pantalla** a través de **mostrarMensaje**. Ambas acciones se refieren a la misma colaboración entre el **ATM** y la **Pantalla**, que ya se listan en la figura 26. El **ATM** obtiene la entrada en respuesta a un indicador, mediante el envío de un mensaje **obtenerEntrada del Teclado**. A continuación, el **ATM** debe determinar si el número de cuenta especificado por el usuario y el NIP coinciden con los de una cuenta en la base de datos. Para ello envía un mensaje **autenticarUsuario** a la **BaseDatosBanco**. Recuerde que **BaseDatosBanco** no puede autenticar a un usuario en forma directa; sólo la **Cuenta** del usuario (es decir, la **Cuenta** que contiene el número de cuenta especificado por el usuario) puede acceder al NIP registrado del usuario para autenticarlo. Por lo tanto, la figura 26 lista una colaboración en la que **BaseDatosBanco** envía un mensaje **validarNIP** a una **Cuenta**.

Una vez autenticado el usuario, el **ATM** muestra el menú principal enviando una serie de mensajes **mostrarMensaje** a la **Pantalla** y obtiene la entrada que contiene una selección de menú; para ello envía un mensaje **obtenerEntrada al Teclado**. Ya hemos tomado en cuenta estas colaboraciones, por lo que no agregamos nada a la figura 26. Una vez que el usuario selecciona un tipo de transacción a realizar, el **ATM** ejecuta la transacción enviando un mensaje **ejecutar** a un objeto de la clase de transacción apropiada (es decir, un objeto **SolicitudSaldo**, **Retiro** o **Deposito**). Por ejemplo, si el usuario elige realizar una solicitud de saldo, el **ATM** envía un mensaje **ejecutar** a un objeto **SolicitudSaldo**.

Un análisis más a fondo del documento de requerimientos revela las colaboraciones involucradas en la ejecución de cada tipo de transacción. Un objeto **SolicitudSaldo** extrae la cantidad de dinero disponible en la cuenta del usuario, al enviar un mensaje **obtenerSaldoDisponible** al objeto **BaseDatosBanco**, el cual responde enviando un mensaje **obtenerSaldoDisponible** a la **Cuenta** del usuario. De manera similar, el objeto **SolicitudSaldo** extrae la cantidad de dinero depositado al enviar un mensaje **obtenerSaldoTotal** al objeto **BaseDatosBanco**, el cual envía el mismo mensaje a la **Cuenta** del usuario. Para mostrar en pantalla ambas cantidades del saldo del usuario al mismo tiempo, el objeto **SolicitudSaldo** envía a la **Pantalla** un mensaje **mostrarMensaje**.



Un objeto **Retiro** envía a la **Pantalla** una serie de mensajes **mostrarMensaje** para mostrar un menú de montos estándar de retiro (es decir, \$20, \$40, \$60, \$100, \$200). El objeto **Retiro** envía al **Teclado** un mensaje **obtenerEntrada** para obtener la selección del menú elegida por el usuario. A continuación, el objeto **Retiro** determina si el monto de retiro solicitado es menor o igual al saldo de la cuenta del usuario. Para obtener el monto de dinero disponible en la cuenta del usuario, el objeto **Retiro** envía un mensaje **obtenerSaldoDisponible** al objeto **BaseDatosBanco**. Después el objeto **Retiro** evalúa si el dispensador contiene suficiente efectivo, enviando al **DispensadorEfectivo** un mensaje **haySuficienteEfectivoDisponible**. Un objeto **Retiro** envía un mensaje **cargar** al objeto **BaseDatosBanco** para reducir el saldo de la cuenta del usuario. El objeto **BaseDatosBanco** envía a su vez el mismo mensaje al objeto **Cuenta** apropiado. Recuerde que al hacer un cargo a una **Cuenta** se reduce tanto el saldo total como el saldo disponible. Para dispensar la cantidad solicitada de efectivo, el objeto **Retiro** envía un mensaje **dispensarEfectivo** al objeto **DispensadorEfectivo**. Por último, el objeto **Retiro** envía a la **Pantalla** un mensaje **mostrarMensaje**, instruyendo al usuario para que tome el efectivo.

Para responder a un mensaje **ejecutar**, un objeto **Deposito** primero envía a la **Pantalla** un mensaje **mostrarMensaje** para pedir al usuario que introduzca un monto a depositar. El objeto **Deposito** envía al **Teclado** un mensaje **obtenerEntrada** para obtener la entrada del usuario. Después, el objeto **Deposito** envía a la **Pantalla** un mensaje **mostrarMensaje** para pedir al usuario que inserte un sobre de depósito. Para determinar si la ranura de depósito recibió un sobre de depósito entrante, el objeto **Deposito** envía al objeto **RanuraDeposito** un mensaje **seRecibioSobreDeposito**. El objeto **Deposito** actualiza la cuenta del usuario enviando un mensaje **abonar** al objeto **BaseDatosBanco**, el cual a su vez envía un mensaje **abonar** al objeto **Cuenta** del usuario. Recuerde que al abonar a una **Cuenta** se incrementa el **saldoTotal**, pero no el **saldoDisponible**.

Diagramas de interacción

Ahora que identificamos un conjunto de posibles colaboraciones entre los objetos en nuestro sistema ATM, modelaremos en forma gráfica estas interacciones. UML cuenta con varios tipos de **diagramas de interacción**, que para modelar el comportamiento de un sistema modelan la forma en que los objetos interactúan entre sí. El **diagrama de comunicación** enfatiza *cuáles objetos* participan en las colaboraciones. [Nota: antiguamente, a los diagramas de comunicación se los llamaba “diagramas de colaboración”]. Al igual que el diagrama de comunicación, el **diagrama de secuencia** muestra las colaboraciones entre los objetos, pero enfatiza *cuándo* se deben enviar los mensajes entre los objetos *a través del tiempo*.

Diagramas de comunicación

La figura 27 muestra un diagrama de comunicación que modela la forma en que el ATM ejecuta una **SolicitudSaldo**. Los objetos se modelan en UML como rectángulos que contienen nombres de la forma **nombreObjeto : NombreClase**. En este ejemplo, que involucra sólo a un objeto de cada tipo, descartamos el nombre del objeto y listamos sólo



un signo de dos puntos (:) seguido del nombre de la clase. [Nota: se recomienda especificar el nombre de cada objeto en un diagrama de comunicación cuando se modelan varios objetos del mismo tipo]. Los objetos que se comunican se conectan con líneas sólidas y los mensajes se pasan entre los objetos a lo largo de estas líneas, en la dirección mostrada por las flechas. El nombre del mensaje, que aparece enseguida de la flecha, es el nombre de una operación (es decir, un método en Java) que pertenece al objeto receptor; considere el nombre como un “servicio” que el objeto receptor proporciona a los objetos emisores (sus “clientes”).



Figura 27: Diagrama de comunicación del ATM, ejecutando una solicitud de saldo.

La flecha rellena en la figura 27 representa un mensaje (o **llamada síncrona**) en UML y una llamada a un método en Java. Esta flecha indica que el flujo de control va desde el objeto emisor (el ATM) hasta el objeto receptor (una **SolicitudSaldo**). Como ésta es una llamada síncrona, el objeto emisor no puede enviar otro mensaje, ni hacer cualquier otra cosa, hasta que el objeto receptor procese el mensaje y devuelva el control al objeto emisor. El emisor sólo espera. Por ejemplo, en la figura 27 el objeto ATM llama al método **ejecutar** de un objeto **SolicitudSaldo** y no puede enviar otro mensaje sino hasta que **ejecutar** termine y devuelva el control al objeto ATM. [Nota: si ésta fuera una llamada asíncrona, representada por una flecha, el objeto emisor no tendría que esperar a que el objeto receptor devolviera el control; continuaría enviando mensajes adicionales inmediatamente después de la llamada asíncrona. Dichas llamadas se implementan en Java mediante el uso de una técnica conocida como subprocesamiento múltiple (*multi-threading*)].

Secuencia de mensajes en un diagrama de comunicación

La figura 28 muestra un diagrama de comunicación que modela las interacciones entre los objetos en el sistema, cuando se ejecuta un objeto de la clase **SolicitudSaldo**. Asumimos que el atributo **numeroCuenta** del objeto contiene el número de cuenta del usuario actual. Las colaboraciones en la figura 28 empiezan después de que el objeto ATM envía un mensaje **ejecutar** a un objeto **SolicitudSaldo** (es decir, la interacción modelada en la figura 27). El número a la izquierda del nombre de un mensaje indica el orden en el que éste se pasa. La secuencia de mensajes en un diagrama de comunicación progresan en orden numérico, de menor a mayor. En este diagrama, la numeración comienza con el mensaje 1 y termina con el mensaje 3. El objeto **SolicitudSaldo** envía primero un mensaje **obtenerSaldoDisponible** al objeto **BaseDatosBanco** (mensaje 1), después envía un mensaje **obtenerSaldoTotal** al objeto **BaseDatosBanco** (mensaje 2). Dentro de los paréntesis que van después del nombre de un mensaje, podemos especificar una lista separada por comas de los nombres de los parámetros que se envían con el mensaje (es decir, los argumentos en una llamada a un método en Java); el objeto **SolicitudSaldo** pasa el atributo **numeroCuenta** con sus mensajes al objeto **BaseDatosBanco** para indicar de cuál objeto **Cuenta** se extraerá la información del saldo. En la figura 22 vimos que las operaciones **obtenerSaldoDisponible** y



obtenerSaldoTotal de la clase BaseDatosBanco requieren cada una de ellas un parámetro para identificar una cuenta. El objeto SolicitudSaldo muestra a continuación el saldoDisponible y el saldoTotal al usuario; para ello pasa un mensaje mostrarMensaje a la Pantalla (mensaje 3) que incluye un parámetro, el cual indica el mensaje a mostrar.

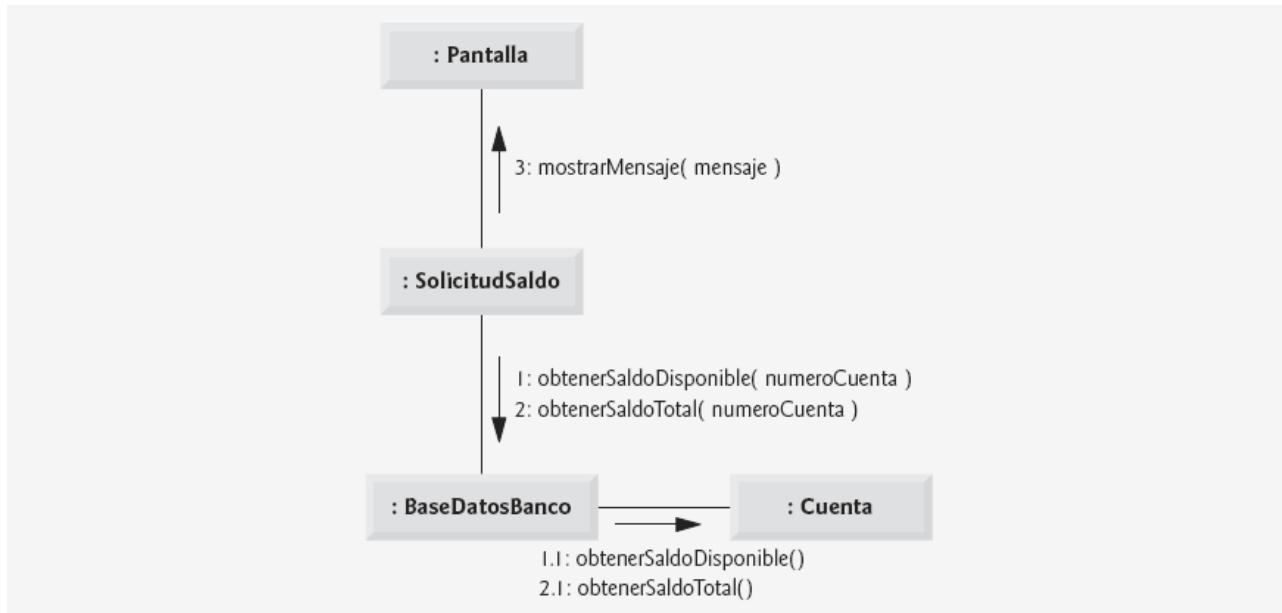


Figura 28: Diagrama de comunicación para ejecutar una solicitud de saldo.

Observe que la figura 28 modela dos mensajes adicionales que se pasan del objeto BaseDatosBanco a un objeto Cuenta (mensaje 1.1 y mensaje 2.1). para proveer al ATM los dos saldos de la Cuenta del usuario (según lo solicitado por los mensajes 1 y 2), el objeto BaseDatosBanco debe pasar un mensaje obtenerSaldoDisponible y un mensaje obtenerSaldoTotal a la Cuenta del usuario. Dichos mensajes que se pasan dentro del manejo de otro mensaje se llaman mensajes anidados. UML recomienda utilizar un esquema de numeración decimal para indicar mensajes anidados. Por ejemplo, el mensaje 1.1 es el primer mensaje anidado en el mensaje 1; el objeto BaseDatosBanco pasa un mensaje obtenerSaldoDisponible durante el procesamiento de BaseDatosBanco de un mensaje con el mismo nombre. [Nota: si el objeto BaseDatosBanco necesita pasar un segundo mensaje anidado mientras procesa el mensaje 1, el segundo mensaje se numera como 1.2]. Un mensaje puede pasarse sólo cuando se han pasado ya todos los mensajes anidados del mensaje anterior. Por ejemplo, el objeto SolicitudSaldo pasa el mensaje 3 sólo hasta que se han pasado los mensajes 2 y 2.1, en ese orden.

El esquema de numeración anidado que se utiliza en los diagramas de comunicación ayuda a aclarar con precisión cuándo y en qué contexto se pasa cada mensaje. Por ejemplo, si numeramos los cinco mensajes de la figura 28 usando un esquema de numeración plano (es decir, 1, 2, 3, 4, 5), podría ser posible que alguien que vieriera el diagrama no pudiera determinar que el objeto BaseDatosBanco pasa el mensaje obtenerSaldoDisponible (mensaje 1.1 a una Cuenta durante el procesamiento del mensaje 1 por parte del objeto BaseDatosBanco, en vez de hacerlo después de completar el procesamiento del mensaje 1. Los números decimales anidados hacen ver que el

segundo mensaje obtenerSaldoDisponible (mensaje 1.1) se pasa a una Cuenta dentro del manejo del primer mensaje obtenerSaldoDisponible (mensaje 1) por parte del objeto BaseDatosBanco.

Diagramas de secuencia

Los diagramas de comunicación enfatizan los participantes en las colaboraciones, pero modelan su sincronización de una forma bastante extraña. Un diagrama de secuencia ayuda a modelar la sincronización de las colaboraciones con más claridad. La figura 29 muestra un diagrama de secuencia que modela la secuencia de las interacciones que ocurren cuando se ejecuta un Retiro. La línea punteada que se extiende hacia abajo desde el rectángulo de un objeto es la **línea de vida** de ese objeto, la cual representa la evolución en el tiempo. Las acciones ocurren a lo largo de la línea de vida de un objeto, en orden cronológico de arriba hacia abajo; una acción cerca de la parte superior ocurre antes que una cerca de la parte inferior.

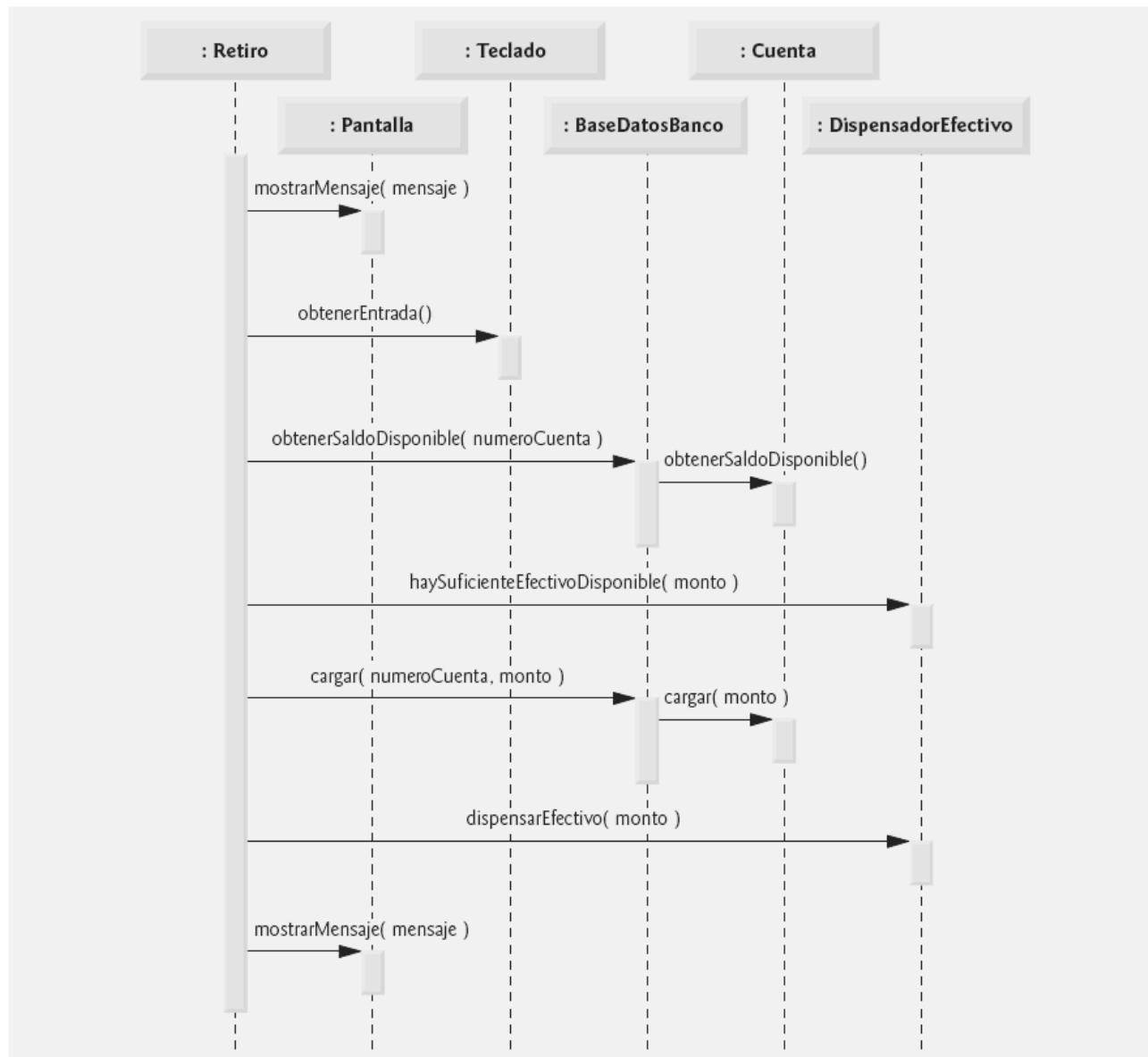


Figura 29: Diagrama de secuencia que modela la ejecución de un Retiro.



El paso de mensajes en los diagramas de secuencia es similar al paso de mensajes en los diagramas de comunicación. Una flecha con punta rellena, que se extiende desde el objeto emisor hasta el objeto receptor, representa un mensaje entre dos objetos. La punta de flecha apunta a una activación en la línea de vida del objeto receptor. Una **activación**, que se muestra como un rectángulo vertical delgado, indica que se está ejecutando un objeto. Cuando un objeto devuelve el control, un mensaje de retorno (representado como una línea punteada con una punta de flecha) se extiende desde la activación del objeto que devuelve el control hasta la activación del objeto que envió originalmente el mensaje. Para eliminar el desorden, omitimos las flechas de los mensajes de retorno; UML permite esta práctica para que los diagramas sean más legibles. Al igual que los diagramas de comunicación, los de secuencia pueden indicar parámetros de mensaje entre los paréntesis que van después del nombre de un mensaje.

La secuencia de mensajes de la figura 29 empieza cuando un objeto **Retiro** pide al usuario que seleccione un monto de retiro; para ello envía a la **Pantalla** un mensaje **mostrarMensaje**. Después el objeto **Retiro** envía al **Teclado** un mensaje **obtenerEntrada**, el cual obtiene los datos de entrada del usuario. Antes ya modelamos la lógica de control involucrada en un objeto **Retiro**, por lo que no mostraremos esta lógica en el diagrama de secuencia de la figura 29. En vez de ello modelaremos el escenario para el mejor caso, en el cual el saldo de la cuenta del usuario es mayor o igual al monto de retiro seleccionado, y el dispensador de efectivo contiene un monto de efectivo suficiente como para satisfacer la solicitud.

Después de obtener un monto de retiro, el objeto **Retiro** envía un mensaje **obtenerSaldoDisponible** al objeto **BaseDatosBanco**, el cual a su vez envía un mensaje **obtenerSaldoDisponible** a la **Cuenta** del usuario. Suponiendo que la cuenta del usuario tiene suficiente dinero disponible para permitir la transacción, el objeto **Retiro** envía al objeto **DispensadorEfectivo** un mensaje **haySuficienteEfectivoDisponible**. Suponiendo que hay suficiente efectivo disponible, el objeto **Retiro** reduce el saldo de la cuenta del usuario (tanto el **saldoTotal** como el **saldoDisponible**) enviando un mensaje **cargar** a la **Cuenta** del usuario. Por último, el objeto **Retiro** envía al **DispensadorEfectivo** un mensaje **dispensarEfectivo** y a la **Pantalla** un mensaje **mostrarMensaje**, indicando al usuario que retire el efectivo de la máquina.

Hemos identificado las colaboraciones entre los objetos en el sistema ATM, y modelamos algunas de estas colaboraciones usando los diagramas de interacción de UML: los diagramas de comunicación y los diagramas de secuencia. A continuación, mejoraremos la estructura de nuestro modelo para completar un diseño orientado a objetos preliminar, y después empezaremos a implementar el sistema ATM en Java.



Ejercicios de autoevaluación

1. Un(a) _____ consiste en que un objeto de una clase envía un mensaje a un objeto de otra clase.
 - a) asociación
 - b) agregación
 - c) colaboración
 - d) composición
2. ¿Cuál forma de diagrama de interacción es la que enfatiza qué colaboraciones se llevan a cabo? ¿Cuál forma enfatiza cuándo ocurren las interacciones?
3. Cree un diagrama de secuencia para modelar las interacciones entre los objetos del sistema ATM, que ocurran cuando se ejecute un **Depósito** con éxito. Explique la secuencia de los mensajes modelados por el diagrama.

Respuestas a los ejercicios de autoevaluación

1. c.
2. Los diagramas de comunicación enfatizan qué colaboraciones se llevan a cabo. Los diagramas de secuencia enfatizan cuándo ocurren las colaboraciones.
3. Un diagrama de secuencia que modele las interacciones entre los objetos del sistema ATM cuando un **Depósito** se ejecuta con éxito debe indicar que un **Depósito** primero envía un mensaje **mostrarMensaje** a la **Pantalla**, para pedirle al usuario que introduzca un monto de depósito. A continuación, el **Depósito** envía un mensaje **obtenerEntrada** al **Teclado** para recibir la entrada del usuario. Despues, el **Depósito** pide al usuario que inserte un sobre de depósito; para ello envía un mensaje **mostrarMensaje** a la **Pantalla**. Luego, el **Depósito** envía un mensaje **seRecibioSobreDeposito** al objeto **RanuraDeposito** para confirmar que el ATM haya recibido el sobre de depósito. Por último, el objeto **Depósito** incrementa el atributo **saldoTotal** (pero no el atributo **saldoDisponible**) de la **Cuenta** del usuario, enviando al objeto **BaseDatosBanco** un mensaje **abonar**. El objeto **BaseDatosBanco** responde enviando el mismo mensaje a la **Cuenta** del usuario.

8. INICIO DE LA PROGRAMACIÓN DE LAS CLASES DEL SISTEMA ATM

Ahora aplicaremos modificadores de acceso **public** y **private** a los miembros de nuestras clases. Los modificadores de acceso determinan la **visibilidad**, o accesibilidad, de los atributos y métodos de un objeto para otros objetos. Antes de empezar a implementar nuestro diseño, debemos considerar cuáles atributos y métodos de nuestras clases deben ser **public** y cuáles deben ser **private**.



Por lo general, los atributos deben ser **private**, y que los métodos invocados por los clientes de una clase dada deben ser **public**. Sin embargo, los métodos que se llaman sólo por otros métodos de la clase como “métodos utilitarios” deben ser **private**. UML emplea **marcadores de visibilidad** para modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica mediante la colocación de un signo más (+) antes de una operación o atributo, mientras que un signo menos (-) indica una visibilidad privada. La figura 30 muestra nuestro diagrama de clases actualizado, en el cual se incluyen los marcadores de visibilidad. [Nota: no incluimos parámetros de operación en la figura 30; esto es perfectamente normal. Agregar los marcadores de visibilidad no afecta a los parámetros que ya están modelados en los diagramas de clases de las figuras 22 a 25].

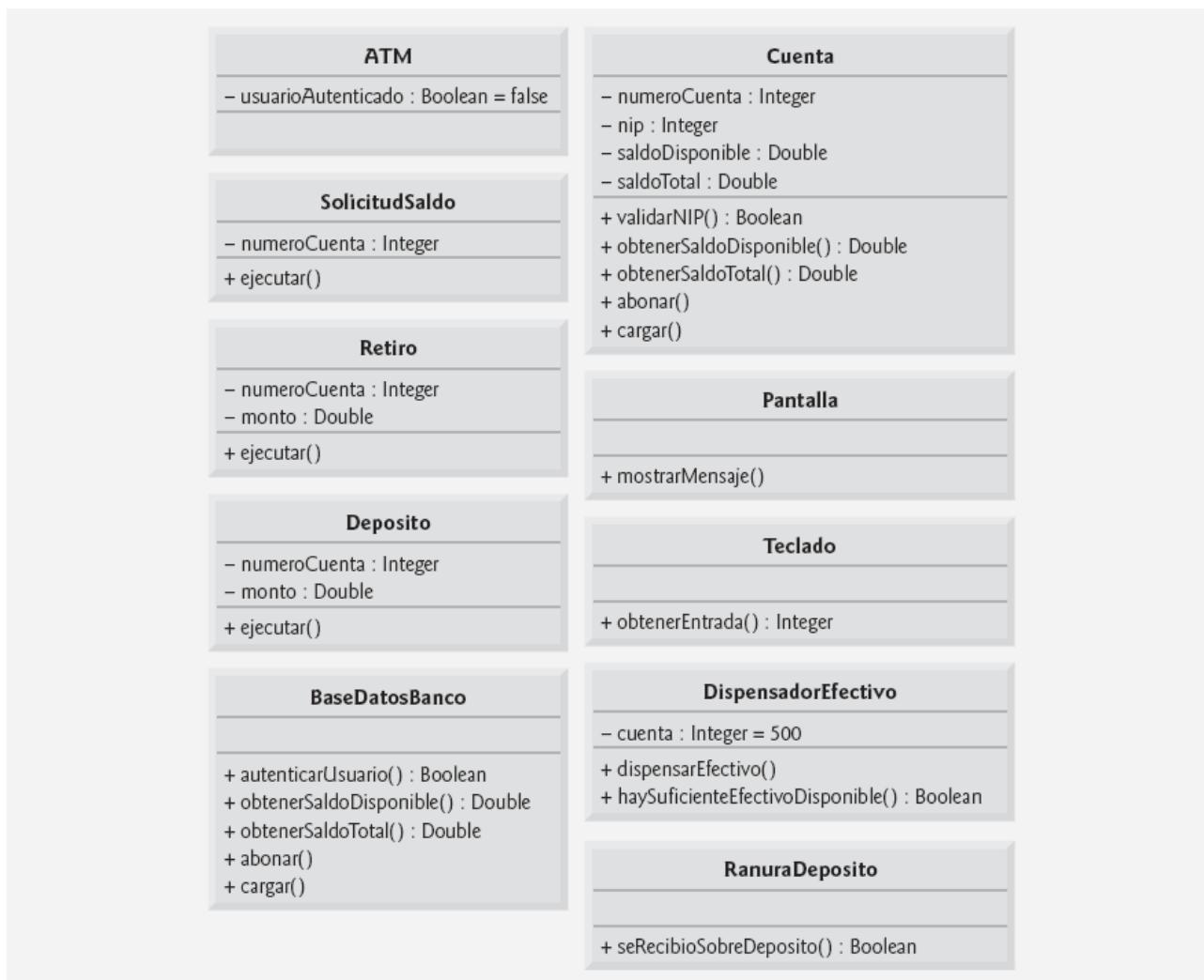


Figura 30: Diagrama de clases con marcadores de visibilidad

Navegabilidad

Antes de empezar a implementar nuestro diseño en Java, presentaremos una notación adicional de UML. El diagrama de clases de la figura 31 refina aún más las relaciones entre las clases del sistema ATM, al agregar flechas de navegabilidad a las líneas de asociación. Las **flechas de navegabilidad** (representadas como flechas con puntas delgadas en el diagrama de clases) indican en qué dirección puede recorrerse

una asociación. Al implementar un sistema diseñado mediante el uso de UML, los programadores utilizan flechas de navegabilidad para ayudar a determinar cuáles objetos necesitan referencias a otros objetos. Por ejemplo, la flecha de navegabilidad que apunta de la clase ATM a la clase BaseDatosBanco indica que podemos navegar de una a la otra, con lo cual se permite a la clase ATM invocar a las operaciones de BaseDatosBanco. No obstante, como la figura 31 no contiene una flecha de navegabilidad que apunte de la clase BaseDatosBanco a la clase ATM, la clase BaseDatosBanco no puede acceder a las operaciones de la clase ATM. Observe que las asociaciones en un diagrama de clases que tienen flechas de navegabilidad en ambos extremos, o que no tienen ninguna flecha, indican una navegabilidad bidireccional: la navegación puede proceder en cualquier dirección a lo largo de la asociación.

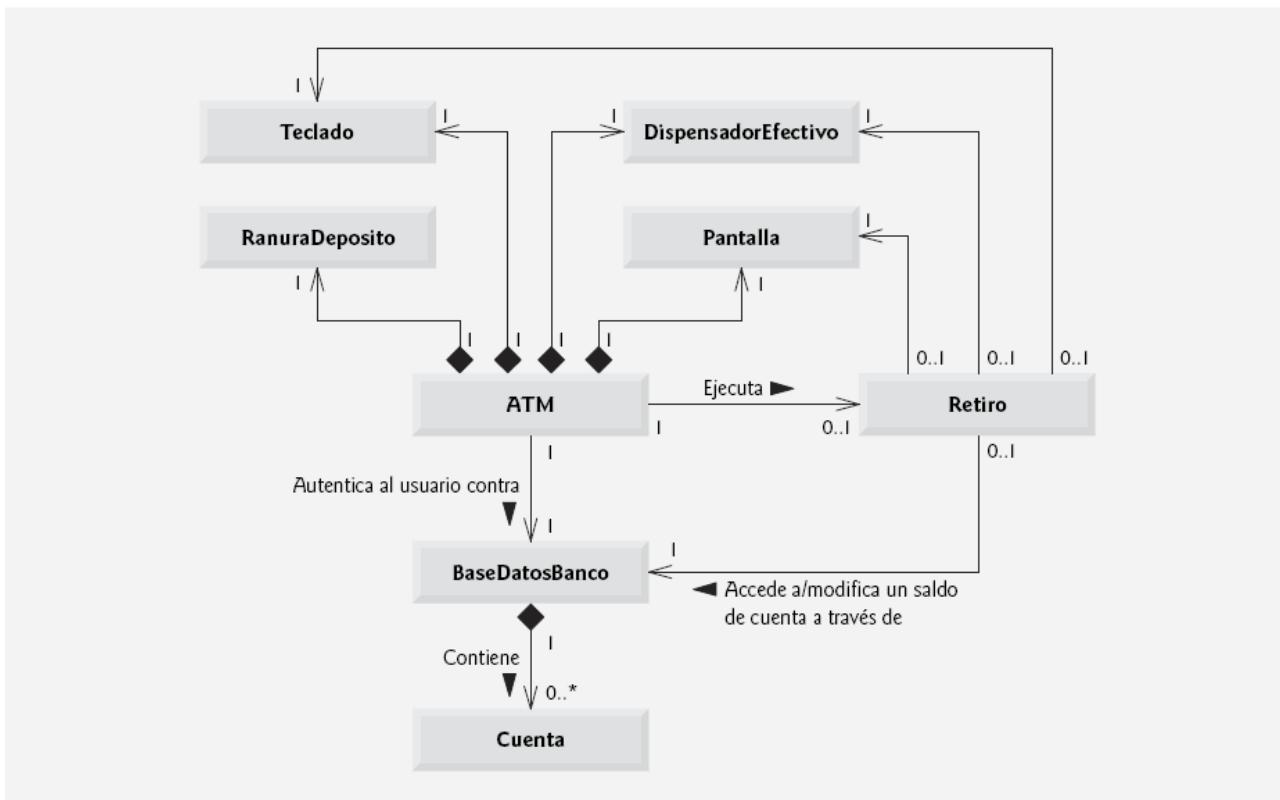


Figura 31: Diagrama de clases con flechas de navegabilidad.

Al igual que el diagrama de clases de la figura 11, el de la figura 31 omite las clases **SolicitudSaldo** y **Deposito** para simplificarlo. La navegabilidad de las asociaciones en las que participan estas dos clases se asemeja mucho a la navegabilidad de las asociaciones de la clase **Retiro**. Anteriormente vimos que **SolicitudSaldo** tiene una asociación con la clase **Pantalla**. Podemos navegar de la clase **SolicitudSaldo** a la clase **Pantalla** a lo largo de esta asociación, pero no podemos navegar de la clase **Pantalla** a la clase **SolicitudSaldo**. Por ende, si modeláramos la clase **SolicitudSaldo** en la figura 31, colocaríamos una flecha de navegabilidad en el extremo de la clase **Pantalla** de esta asociación. Recuerde también que la clase **Deposito** se asocia con las clases **Pantalla**, **Teclado** y **RanuraDeposito**. Podemos navegar de la clase **Deposito** a cada una de estas clases, pero no al revés. Por lo tanto, podríamos colocar flechas de navegabilidad en los extremos de las clases **Pantalla**, **Teclado** y **RanuraDeposito** de estas asociaciones. [Nota: más adelante, modelaremos estas clases



y asociaciones adicionales en nuestro diagrama de clases final, una vez que hayamos simplificado la estructura de nuestro sistema, al incorporar el concepto orientado a objetos de la herencia].

Implementación del sistema ATM a partir de su diseño de UML

Ahora estamos listos para empezar a implementar el sistema ATM. Primero convertiremos las clases de los diagramas de las figuras 30 y 31 en código de Java. Este código representará el “esqueleto” del sistema.

Como ejemplo, empezaremos a desarrollar el código a partir de nuestro diseño de la clase **Retiro** en la figura 30. Utilizaremos esta figura para determinar los atributos y operaciones de la clase. Usaremos el modelo de UML en la figura 31 para determinar las asociaciones entre las clases. Seguiremos estos cuatro lineamientos para cada clase:

1. Use el nombre que se localiza en el primer compartimiento para declarar la clase como **public**, con un constructor sin parámetros vacío. Incluimos este constructor tan sólo como un receptáculo para recordarnos que la mayoría de las clases necesitarán en definitiva constructores. Más adelante, cuando completaremos una versión funcional de esta clase, agregaremos todos los argumentos y el código necesarios al cuerpo del constructor. Por ejemplo, la clase **Retiro** produce el código de la figura 32. [Nota: si encontramos que las variables de instancia de la clase sólo requieren la inicialización predeterminada, eliminaremos el constructor sin parámetros vacío, ya que es innecesario].
2. Use los atributos que se localizan en el segundo compartimiento para declarar las variables de instancia. Por ejemplo, los atributos **private numeroCuenta** y **monto** de la clase **Retiro** producen el código de la figura 33. [Nota: el constructor de la versión funcional completa de esta clase asignará valores a estos atributos].
3. Use las asociaciones descritas en el diagrama de clases para declarar las referencias a otros objetos. Por ejemplo, de acuerdo con la figura 31, **Retiro** puede acceder a un objeto de la clase **Pantalla**, a un objeto de la clase **Teclado**, a un objeto de la clase **DispensadorEfectivo** y a un objeto de la clase **BaseDatosBanco**. Esto produce el código de la figura 34. [Nota: el constructor de la versión funcional completa de esta clase inicializará estas variables de instancia con referencias a objetos reales].
4. Use las operaciones que se localizan en el tercer compartimiento de la figura 30 para declarar las armazones de los métodos. Si todavía no hemos especificado un tipo de valor de retorno para una operación, declaramos el método con el tipo de retorno **void**. Consulte los diagramas de clases de las figuras 22 a 25 para declarar cualquier parámetro necesario. Por ejemplo, al agregar la operación **public ejecutar** en la clase **Retiro**, que tiene una lista de parámetros vacía, se produce el código de la figura 35. [Nota: codificaremos los cuerpos de los métodos más adelante, cuando implementemos el sistema ATM completo].

Esto concluye nuestra discusión sobre los fundamentos de la generación de clases a partir de diagramas de UML.



```
1 // La clase Retiro representa una transacción de retiro del ATM
2 public class Retiro
3 {
4     // constructor sin argumentos
5     public Retiro()
6     {
7         } // fin del constructor de Retiro sin argumentos
8 } // fin de la clase Retiro
```

Figura 32: Código de Java para la clase Retiro, con base en las figuras 30 y 31.

```
1 // La clase Retiro representa una transacción de retiro del ATM
2 public class Retiro
3 {
4     // atributos
5     private int numeroCuenta; // cuenta de la que se van a retirar los fondos
6     private double monto; // monto que se va a retirar de la cuenta
7
8     // constructor sin argumentos
9     public Retiro()
10    {
11        } // fin del constructor de Retiro sin argumentos
12    } // fin de la clase Retiro
```

Figura 33: Código de Java para la clase Retiro, con base en las figuras 30 y 31.

```
1 // La clase Retiro representa una transacción de retiro del ATM
2 public class Retiro
3 {
4     // atributos
5     private int numeroCuenta; // cuenta de la que se retirarán los fondos
6     private double monto; // monto a retirar
7
8     // referencias a los objetos asociados
9     private Pantalla pantalla; // pantalla del ATM
10    private Teclado teclado; // teclado del ATM
11    private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
12    private BaseDatosBanco baseDatosBanco; // base de datos de información de las
13    cuentas
14
15    // constructor sin argumentos
16    public Retiro()
17    {
18        } // fin del constructor de Retiro sin argumentos
19    } // fin de la clase Retiro
```

Figura 34: Código de Java para la clase Retiro, con base en las figuras 30 y 31.



```
1 // La clase Retiro representa una transacción de retiro del ATM
2 public class Retiro
3 {
4     // atributos
5     private int numeroCuenta; // cuenta de la que se van a retirar los fondos
6     private double monto; // monto a retirar
7
8     // referencias a los objetos asociados
9     private Pantalla pantalla; // pantalla del ATM
10    private Teclado teclado; // teclado del ATM
11    private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
12    private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas
13
14    // constructor sin argumentos
15    public Retiro()
16    {
17        } // fin del constructor de Retiro sin argumentos
18
19    // operaciones
20    public void ejecutar()
21    {
22        } // fin del método ejecutar
23 } // fin de la clase Retiro
```

Figura 35: Código de Java para la clase **Retiro**, con base en las figuras 30 y 31.

Ejercicios de autoevaluación

1. Indique si el siguiente enunciado es *verdadero* o *falso*, y si es *falso*, explique por qué: si un atributo de una clase se marca con un signo menos (-) en un diagrama de clases, el atributo no es directamente accesible fuera de la clase.
2. En la figura 31, la asociación entre los objetos **ATM** y **Pantalla** indica:
 - a) que podemos navegar de la **Pantalla** al **ATM**.
 - b) que podemos navegar del **ATM** a la **Pantalla**.
 - c) (a) y (b); la asociación es bidireccional.
 - d) Ninguna de las anteriores.
3. Escriba código de Java para empezar a implementar el diseño para la clase **Teclado**.

Respuestas a los ejercicios de autoevaluación

1. Verdadero. El signo menos (-) indica visibilidad privada.
2. b.
3. El diseño para la clase **Teclado** produce el código de la figura 36. Recuerde que la clase **Teclado** no tiene atributos en estos momentos, pero pueden volverse aparentes a medida que continuemos con la implementación. Observe además que, si fuéramos a diseñar un ATM real, el método **obtenerEntrada** tendría que interactuar con el hardware del teclado del ATM. En realidad recibiremos la entrada del teclado de una computadora personal, cuando escribamos el código de Java completo.



```
1 // la clase Teclado representa el teclado de un ATM
2 public class Teclado
3 {
4     // no se han especificado atributos todavía
5
6     // constructor sin argumentos
7     public Teclado()
8     {
9         } // fin del constructor de Teclado sin argumentos
10
11    // operaciones
12    public int obtenerEntrada()
13    {
14        } // fin del método obtenerEntrada
15    } // fin de la clase Teclado
```

Figura 36: Código de Java para la clase Teclado, con base en las figuras 30 y 31.

9. INCORPORACIÓN DE LA HERENCIA EN EL SISTEMA ATM

Ahora mejoraremos nuestro diseño del sistema ATM para ver cómo podría beneficiarse de la herencia. Para aplicar la herencia, primero buscamos características comunes entre las clases del sistema. Creamos una jerarquía de herencia para modelar las clases similares (pero no idénticas). Después modificamos nuestro diagrama de clases para incorporar las nuevas relaciones de herencia. Por último, demostramos cómo traducir nuestro diseño actualizado en código de Java.

Anteriormente nos topamos con el problema de representar una transacción financiera en el sistema. Creamos tres clases de transacciones (*SolicitudSaldo*, *Retiro* y *Deposito*) para representar las transacciones que puede realizar el sistema ATM. La figura 37 muestra los atributos y operaciones de las clases *SolicitudSaldo*, *Retiro* y *Deposito*. Observe que estas clases tienen un atributo (*numeroCuenta*) y una operación (*ejecutar*) en común. Cada clase requiere que el atributo *numeroCuenta* especifique la cuenta a la que se aplica la transacción. Cada clase contiene la operación *ejecutar*, que el ATM invoca para realizar la transacción. Es evidente que *SolicitudSaldo*, *Retiro* y *Deposito* representan *tipos* de transacciones. La figura 37 revela las características comunes entre las clases de transacciones, por lo que el uso de la herencia para factorizar las características comunes parece apropiado para diseñar estas clases. Colocamos la funcionalidad común en una superclase, *Transaccion*, que las clases *SolicitudSaldo*, *Retiro* y *Deposito* extienden.

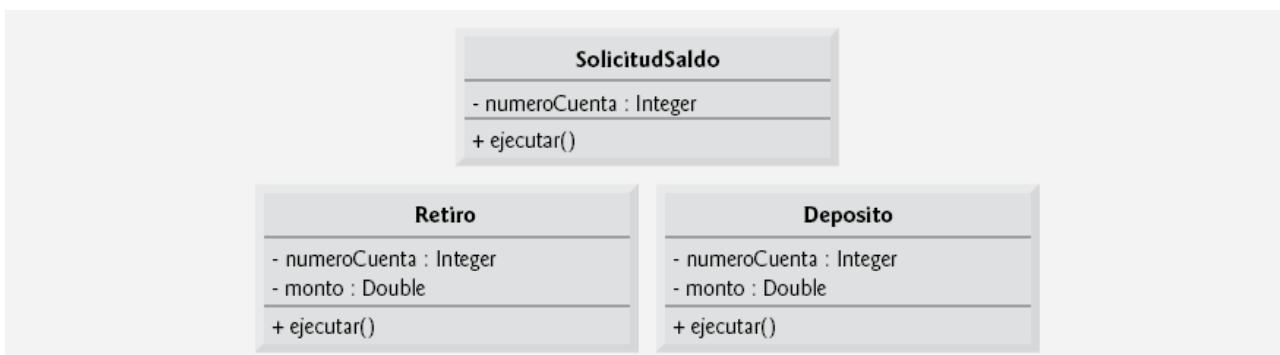


Figura 37: Atributos y operaciones de las clases *SolicitudSaldo*, *Retiro* y *Deposito*.

UML especifica una relación conocida como **generalización** para modelar la herencia. La figura 38 es el diagrama de clases que modela la generalización de la superclase **Transaccion** y las subclases **SolicitudSaldo**, **Retiro** y **Deposito**. Las flechas con puntas triangulares huecas indican que las clases **SolicitudSaldo**, **Retiro** y **Deposito** exienden a la clase **Transaccion**. Se dice que la clase **Transaccion** es una generalización de las clases **SolicitudSaldo**, **Retiro** y **Deposito**. Se dice que las clases **SolicitudSaldo**, **Retiro** y **Deposito** son **especializaciones** de la clase **Transaccion**.

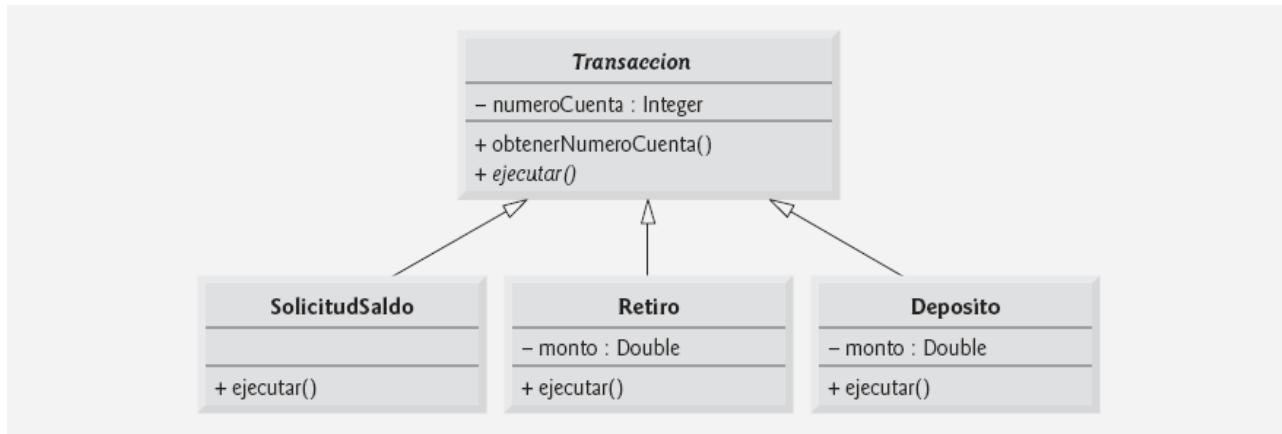


Figura 38: Diagrama de clases que modela la generalización de la superclase **Transaccion** y las subclases **SolicitudSaldo**, **Retiro** y **Deposito**. Observe que los nombres de las clases abstractas (por ejemplo, **Transaccion**) y los nombres de los métodos abstractos (por ejemplo, **ejecutar** en la clase **Transaccion**) aparece en cursivas.

Las clases **SolicitudSaldo**, **Retiro** y **Deposito** comparten el atributo entero `numeroCuenta`, por lo que factorizamos este atributo común y lo colocamos en la superclase **Transaccion**. Ya no listamos a `numeroCuenta` en el segundo compartimiento de cada subclase, ya que las tres subclases heredan este atributo de **Transaccion**. Sin embargo, recuerde que las subclases no pueden acceder a los atributos `private` de una superclase. Por lo tanto, incluimos el método `public obtenerNumeroCuenta` en la clase **Transaccion**. Cada subclase heredará este método, con lo cual podrá acceder a su `numeroCuenta` según sea necesario para ejecutar una transacción.

De acuerdo con la figura 37, las clases **SolicitudSaldo**, **Retiro** y **Deposito** también comparten la operación `ejecutar`, por lo que decidimos que la superclase **Transaccion** debe contener el método `public ejecutar`. Sin embargo, no tiene sentido implementar a `ejecutar` en la clase **Transaccion**, ya que la funcionalidad que proporciona este método depende del tipo específico de la transacción actual. Por lo tanto, declaramos el método `ejecutar` como `abstract` en la superclase **Transaccion**. Cualquier clase que contenga cuando menos un método abstracto también debe declararse como `abstract`. Esto obliga a que cualquier clase de **Transaccion** que deba ser una clase concreta (es decir, **SolicitudSaldo**, **Retiro** y **Deposito**) a implementar el método `ejecutar`. UML requiere que coloquemos los nombres de clase abstractos (y los métodos abstractos) en cursivas, por lo cual **Transaccion** y su método `ejecutar` aparecen en cursivas en la figura 38. Observe que el método `ejecutar` no está en cursivas en las subclases **SolicitudSaldo**, **Retiro** y **Deposito**. Cada subclase sobrescribe el método `ejecutar` de la superclase **Transaccion** con una implementación concreta que realiza los pasos apropiados para completar ese tipo de transacción.



Observe que la figura 38 incluye la operación **ejecutar** en el tercer compartimiento de las clases **SolicitudSaldo**, **Retiro** y **Deposito**, ya que cada clase tiene una implementación concreta distinta del método sobrescrito.

Al incorporar la herencia, se proporciona al ATM una manera elegante de ejecutar todas las transacciones “en general”. Por ejemplo, suponga que un usuario elige realizar una solicitud de saldo. El ATM establece una referencia **Transaccion** a un nuevo objeto de la clase **SolicitudSaldo**. Cuando el ATM utiliza su referencia **Transaccion** para invocar el método **ejecutar**, se hace una llamada a la versión de ejecutar de **SolicitudSaldo**.

Este enfoque polimórfico también facilita la extensibilidad del sistema. Si deseamos crear un nuevo tipo de transacción (por ejemplo, una transferencia de fondos o el pago de un recibo), sólo tenemos que crear una subclase de **Transaccion** adicional que sobrescriba el método **ejecutar** con una versión apropiada para ejecutar el nuevo tipo de transacción. Sólo tendríamos que realizar pequeñas modificaciones al código del sistema, para permitir que los usuarios seleccionen el nuevo tipo de transacción del menú principal y para que la clase ATM cree instancias y ejecute objetos de la nueva subclase. La clase ATM podría ejecutar transacciones del nuevo tipo utilizando el código actual, ya que éste ejecuta todas las transacciones de manera polimórfica, usando una referencia **Transaccion** general.

Como aprendió antes, una clase abstracta como **Transaccion** es una para la cual el programador nunca tendrá la intención de crear instancias de objetos. Una clase abstracta sólo declara los atributos y comportamientos comunes de sus subclases en una jerarquía de herencia. La clase **Transaccion** define el concepto de lo que significa ser una transacción que tiene un número de cuenta y puede ejecutarse. Tal vez usted se pregunte por qué nos tomamos la molestia de incluir el método **abstract ejecutar** en la clase **Transaccion**, si carece de una implementación concreta. En concepto, incluimos este método porque corresponde al comportamiento que define a todas las transacciones: ejecutarse. Técnicamente, debemos incluir el método **ejecutar** en la superclase **Transaccion**, de manera que la clase ATM (o cualquier otra clase) pueda invocar mediante el polimorfismo a la versión sobrescrita de este método en cada subclase, a través de una referencia **Transaccion**. Además, desde la perspectiva de la ingeniería de software, al incluir un método abstracto en una superclase, el que implementa las subclases se ve obligado a sobrescribir ese método con implementaciones concretas en las subclases, o de lo contrario, las subclases también serán abstractas, lo cual impedirá que se creen instancias de objetos de esas subclases.

Las subclases **SolicitudSaldo**, **Retiro** y **Deposito** heredan el atributo **numeroCuenta** de la superclase **Transaccion**, pero las clases **Retiro** y **Deposito** contienen el atributo adicional **monto** que las diferencia de la clase **SolicitudSaldo**. Las clases **Retiro** y **Deposito** requieren este atributo adicional para almacenar el monto de dinero que el usuario desea retirar o depositar. La clase **SolicitudSaldo** no necesita dicho atributo, puesto que sólo requiere un número de cuenta para ejecutarse. Aun cuando dos de las tres subclases de **Transaccion** comparten el atributo **monto**, no lo colocamos en la superclase **Transaccion**; en la superclase sólo colocamos las características comunes para todas las subclases, ya que de otra forma las subclases podrían heredar atributos (y métodos) que no necesitan y no deben tener.

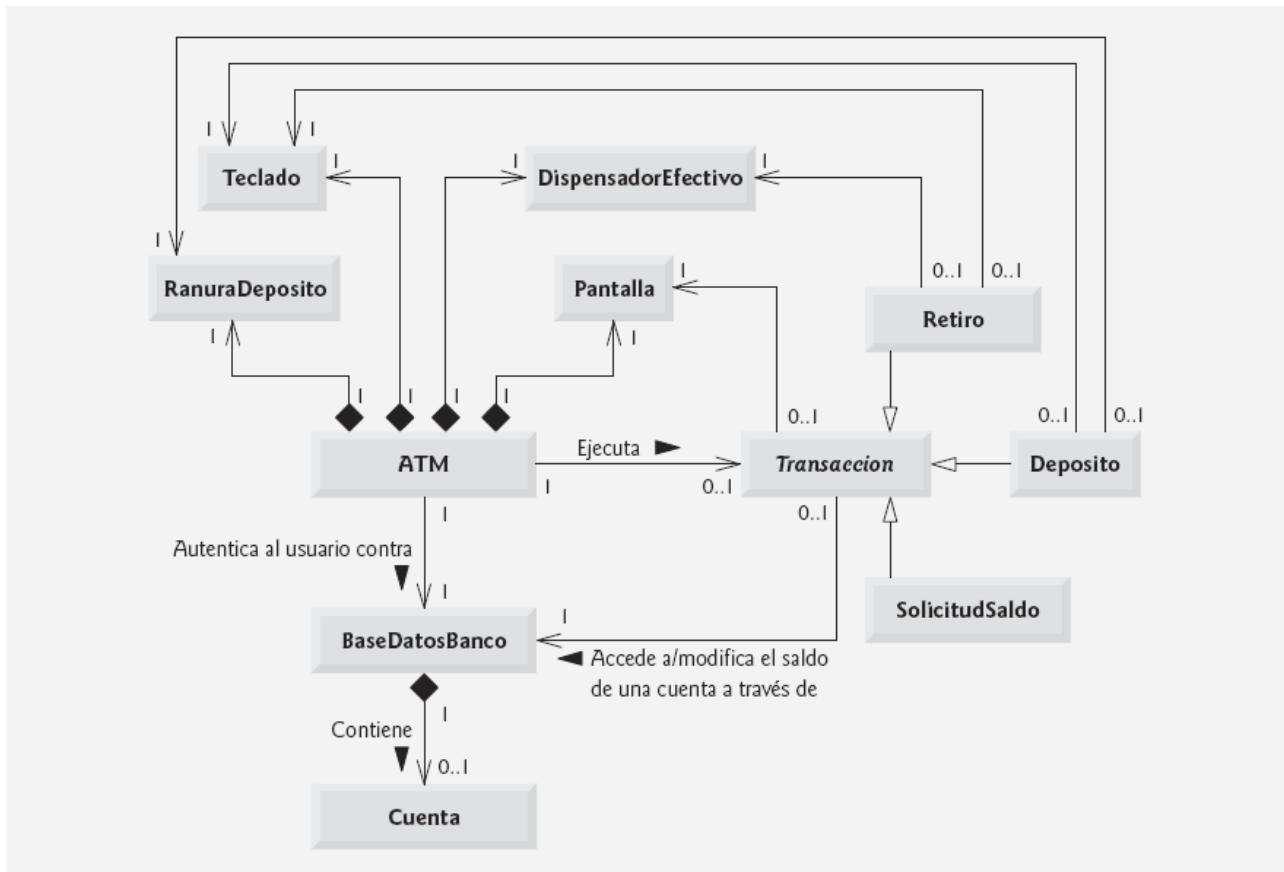


Figura 39: Diagrama de clases del sistema ATM (en el que se incorpora la herencia).

La figura 39 presenta un diagrama de clases actualizado de nuestro modelo, en el cual se incorpora la herencia y se introduce la clase **Transaccion**. Modelamos una asociación entre la clase **ATM** y la clase **Transaccion** para mostrar que la clase **ATM**, en cualquier momento dado, está ejecutando una transacción o no lo está (es decir, existen cero o un objetos de tipo **Transaccion** en el sistema, en un momento dado). Como un **Retiro** es un tipo de **Transaccion**, ya no dibujamos una línea de asociación directamente entre la clase **ATM** y la clase **Retiro**. La subclase **Retiro** hereda la asociación de la superclase **Transaccion** con la clase **ATM**. Las subclases **SolicitudSaldo** y **Deposito** también heredan esta asociación, por lo que ya no existen las asociaciones entre la clase **ATM** y las clases **SolicitudSaldo** y **Deposito**, que se habían omitido anteriormente.

También agregamos una asociación entre la clase **Transaccion** y la clase **BaseDatosBanco** (figura 39). Todos los objetos **Transaccion** requieren una referencia a **BaseDatosBanco**, de manera que puedan acceder a (y modificar) la información de las cuentas. Debido a que cada subclase de **Transaccion** hereda esta referencia, ya no tenemos que modelar la asociación entre la clase **Retiro** y **BaseDatosBanco**. De manera similar, ya no existen las asociaciones entre **BaseDatosBanco** y las clases **SolicitudSaldo** y **Deposito**, que omitimos anteriormente.

Mostramos una asociación entre la clase **Transaccion** y la clase **Pantalla**. Todos los objetos **Transaccion** muestran los resultados al usuario a través de la **Pantalla**. Por ende, ya no incluimos la asociación que modelamos antes entre **Retiro** y **Pantalla**,



aunque Retiro aún participa en las asociaciones con DispensadorEfectivo y Teclado. Nuestro diagrama de clases que incorpora la herencia también modela a Deposito y SolicitudSaldo. Mostramos las asociaciones entre Deposito y tanto RanuraDeposito como Teclado. Observe que la clase SolicitudSaldo no participa en asociaciones más que las heredadas de la clase Transaccion; un objeto SolicitudSaldo sólo necesita interactuar con la BaseDatosBanco y con la Pantalla.

El diagrama de clases de la figura 30 muestra los atributos y las operaciones con marcadores de visibilidad. Ahora presentamos un diagrama de clases modificado que incorpora la herencia en la figura 40.

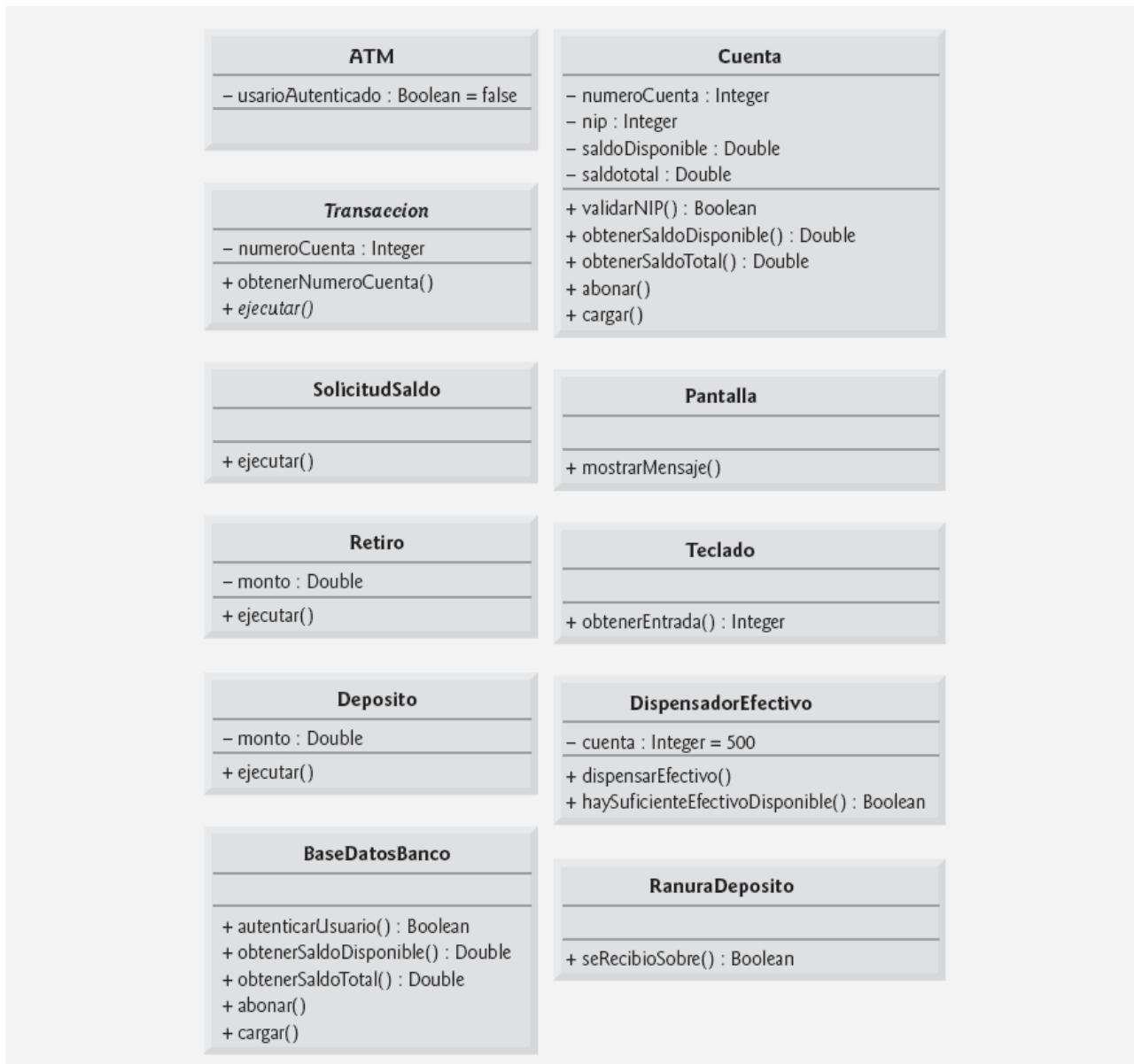


Figura 40: Diagrama de clases con atributos y operaciones (incorporando la herencia).

Este diagrama abreviado no muestra las relaciones de herencia, sino los atributos y los métodos después de haber empleado la herencia en nuestro sistema. Para ahorrar espacio, como hicimos en la figura 15, no incluimos los atributos mostrados por las asociaciones en la figura 39; sin embargo, los incluimos en la implementación completa



en Java. También omitimos todos los parámetros de las operaciones, como hicimos en la figura 30; al incorporar la herencia no se afectan los parámetros que ya estaban modelados en las figuras 22 a 25.

Implementación del diseño del sistema ATM (en el que se incorpora la herencia)

Anteriormente empezamos a implementar el diseño del sistema ATM en código de Java. Ahora modificaremos nuestra implementación para incorporar la herencia, usando la clase **Retiro** como ejemplo.

1. Si la clase A es una generalización de la clase B, entonces la clase B extiende a la clase A en la declaración de la clase. Por ejemplo, la superclase abstracta **Transaccion** es una generalización de la clase **Retiro**. La figura 41 contiene la estructura de la clase **Retiro**, la cual contiene la declaración de clase apropiada.

```
1 // La clase Retiro representa una transacción de retiro en el ATM
2 public class Retiro extends Transaccion
3 {
4 } // fin de la clase Retiro
```

Figura 41: Código de Java para la estructura de la clase **Retiro**.

2. Si la clase A es una clase abstracta y la clase B es una subclase de la clase A, entonces la clase B debe implementar los métodos abstractos de la clase A, si la clase B va a ser una clase concreta. Por ejemplo, la clase **Transaccion** contiene el método abstracto **ejecutar**, por lo que la clase **Retiro** debe implementar este método si queremos crear una instancia de un objeto **Retiro**. La figura 42 es el código en Java para la clase **Retiro** de las figuras 39 y 40. La clase **Retiro** hereda el campo **numeroCuenta** de la superclase **Transaccion**, por lo que **Retiro** no necesita declarar este campo. La clase **Retiro** también hereda referencias a las clases **Pantalla** y **BaseDatosBanco** de su superclase **Transaccion**, por lo que no incluimos estas referencias en nuestro código. La figura 40 especifica el atributo **monto** y la operación **ejecutar** para la clase **Retiro**. La línea 6 de la figura 42 declara un campo para el atributo **monto**. Las líneas 16 a 18 declaran la estructura de un método para la operación **ejecutar**. Recuerde que la subclase **Retiro** debe proporcionar una implementación concreta del método **abstract ejecutar** de la superclase **Transaccion**. Las referencias **teclado** y **dispensadorEfectivo** (líneas 7 y 8) son campos derivados de las asociaciones de **Retiro** en la figura 39. [Nota: el constructor en la versión completa de esta clase inicializará estas referencias con objetos reales].



```
1 // Retiro.java
2 // Se generó usando los diagramas de clases en las figuras 10.21 y 10.22
3 public class Retiro extends Transaccion
4 {
5     // atributos
6     private double monto; // monto a retirar
7     private Teclado teclado; // referencia al teclado
8     private DispensadorEfectivo dispensadorEfectivo; // referencia al dispensador de
9     efectivo
10    // constructor sin argumentos
11    public Retiro()
12    {
13    } // fin del constructor de Retiro sin argumentos
14
15    // método que sobrescribe a ejecutar
16    public void ejecutar()
17    {
18    } // fin del método ejecutar
19 } // fin de la clase Retiro
```

Figura 42: Código de Java para la clase Retiro, basada en las figuras 39 y 40.

Esto concluye nuestro diseño orientado a objetos del sistema ATM. Le recomendamos leer con cuidado el código fuente completo y su descripción (pág. 133), ya que contiene muchos comentarios y sigue con precisión el diseño, con el cual usted ya está familiarizado. La descripción que lo acompaña está escrita cuidadosamente, para guiar su comprensión acerca de la implementación con base en el diseño de UML.

Ejercicios de autoevaluación

1. UML utiliza una flecha con una _____ para indicar una relación de generalización.
 - a) punta con relleno sólido
 - b) punta triangular sin relleno
 - c) punta hueca en forma de diamante
 - d) punta lineal
2. Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: UML requiere que subrayemos los nombres de las clases abstractas y los nombres de los métodos abstractos.
3. Escriba código en Java para empezar a implementar el diseño para la clase Transaccion que se especifica en las figuras 39 y 40. Asegúrese de incluir los atributos tipo referencias `private`, con base en las asociaciones de la clase Transaccion. Asegúrese también de incluir los *métodos establecer* `public` que proporcionan acceso a cualquiera de estos atributos `private` que requieren las subclases para realizar sus tareas.



Respuestas a los ejercicios de autoevaluación

1. b.
2. Falso. UML requiere que se escriban los nombres de las clases abstractas y de los métodos abstractos en cursiva.
3. El diseño para la clase **Transaccion** produce el código de la figura 43. Los cuerpos del constructor de la clase y los métodos se completarán más adelante. Cuando estén implementados por completo, los métodos **obtenerPantalla** y **obtenerBaseDatosBanco** devolverán los atributos de referencias **private** de la superclase **Transaccion**, llamados **pantalla** y **baseDatosBanco**, respectivamente. Estos métodos permiten que las subclases de **Transaccion** accedan a la pantalla del ATM e interactúen con la base de datos del banco.

```
1 // La clase abstracta Transaccion representa una transacción con el ATM
2 public abstract class Transaccion
3 {
4     // atributos
5     private int numeroCuenta; // indica la cuenta involucrada
6     private Pantalla pantalla; // la pantalla del ATM
7     private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas
8
9     // constructor sin argumentos invocado por las subclases, usando super()
10    public Transaccion()
11    {
12    } // fin del constructor Transaccion sin argumentos
13
14    // devuelve el número de cuenta
15    public int obtenerNumeroCuenta()
16    {
17    } // fin del método obtenerNumeroCuenta
18
19    // devuelve referencia a la pantalla
20    public Pantalla obtenerPantalla()
21    {
22    } // fin del metodo getScreen
23
24    // devuelve referencia a la base de datos del banco
25    public BaseDatosBanco obtenerBaseDatosBanco()
26    {
27    } // fin del método obtenerBaseDatosBanco
28
29    // método abstracto sobreescrito por las subclases
30    public abstract void ejecutar();
31 } // fin de la clase Transaccion
```

Figura 43: Código de Java para la clase **Transaccion**, basada en las figuras 39 y 40.



10. DESCRIPCIÓN DEL CÓDIGO FUENTE COMPLETO

Consideramos aquí las clases en el orden en el que fueron identificadas en la pág. 86: ATM, Pantalla, Teclado, DispensadorEfectivo, RanuraDeposito, Cuenta, BaseDatosBanco, Transaccion, SolicitudSaldo, Retiro y Deposito. Presentamos también una aplicación de Java (CasoEstudioATM) que inicia el ATM y pone en uso las demás clases del sistema. Recuerde que estamos desarrollando la primera versión del sistema ATM que se ejecuta en una computadora personal, y utiliza el teclado y el monitor para lograr la mayor semejanza posible con el teclado y la pantalla de un ATM. Además, sólo simulamos las acciones del dispensador de efectivo y la ranura de depósito del ATM. Sin embargo, tratamos de implementar el sistema de manera tal que las versiones reales de hardware de esos dispositivos pudieran integrarse sin necesidad de cambios considerables en el código.

10.1. La clase ATM

La clase ATM representa al ATM como un todo. Las líneas 6 a 12 implementan los atributos de la clase. Determinamos todos estos atributos (excepto uno) de los diagramas de clase de las figuras 39 y 40. En la figura 40 implementamos el atributo Boolean `usuarioAutenticado` de UML como un atributo `boolean` en Java (línea 6). En la línea 7 se declara un atributo que no se incluye en nuestro diseño UML: el atributo `int numeroCuentaActual`, que lleva el registro del número de cuenta del usuario autenticado actual. Pronto veremos cómo es que la clase utiliza este atributo. En las líneas 8 a 12 se declaran atributos de tipo de referencia, correspondientes a las asociaciones de la clase ATM modeladas en el diagrama de clases de la figura 39. Estos atributos permiten al ATM acceder a sus partes (es decir, su Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito) e interactuar con la base de datos de información de cuentas bancarias (es decir, un objeto BaseDatosBanco). Las líneas 14 a 17 declaran constantes enteras que corresponden a las cuatro opciones en el menú principal del ATM (es decir, solicitud de saldo, retiro, depósito y salir). Las líneas 20 a 28 declaran el constructor, el cual inicializa los atributos de la clase. Cuando se crea un objeto ATM por primera vez, no se autentica ningún usuario, por lo que la línea 21 inicializa `usuarioAutenticado` a `false`. De igual forma, la línea 22 inicializa `numeroCuentaActual` a 0 debido a que todavía no hay un usuario actual. Las líneas 23 a 26 crean instancias de nuevos objetos para representar las partes del ATM. Recuerde que la clase ATM tiene relaciones de composición con las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito, por lo que la clase ATM es responsable de su creación. La línea 27 crea un nuevo objeto BaseDatosBanco. [Nota: si éste fuera un sistema ATM real, la clase ATM recibiría una referencia a un objeto base de datos existente creado por el banco. Sin embargo, en esta implementación sólo estamos simulando la base de datos del banco, por lo que ATM crea el objeto BaseDatosBanco con el que interactúa].

El método run de ATM

El diagrama de clases de la figura 40 no lista ninguna operación para la clase ATM. Ahora vamos a implementar una operación (es decir, un método `public`) en la clase ATM que permite a un cliente externo de la clase (en este caso, la clase CasoEstudioATM) indicar al ATM que se ejecute. El método `run` de ATM (líneas 31 a 45) usa un ciclo infinito (líneas 33 a 44) para dar la bienvenida repetidas veces a un usuario, tratar de



autenticarlo y, si la autenticación tiene éxito, permite al usuario realizar transacciones. Una vez que un usuario autenticado realiza las transacciones deseadas y selecciona la opción para salir, el ATM se reinicia a sí mismo, muestra un mensaje de despedida al usuario y reinicia el proceso. Aquí usamos un ciclo infinito para simular el hecho de que un ATM parece ejecutarse en forma continua hasta que el banco lo apaga (una acción que está más allá del control del usuario). Un usuario del ATM tiene la opción de salir del sistema, pero no la habilidad de apagar el ATM por completo.

Autenticación de un usuario

En el ciclo infinito del método `run`, las líneas 35 a 38 provocan que el ATM dé la bienvenida al usuario y trate de autenticarlo repetidas veces, siempre y cuando éste no haya sido autenticado antes (es decir, que `!usuarioAutenticado` sea `true`). La línea 36 invoca al método `mostrarLineaMensaje` de la pantalla del ATM para mostrar un mensaje de bienvenida. Al igual que el método `mostrarMensaje` de `Pantalla` diseñado en el caso de estudio, el método `mostrarLineaMensaje` muestra un mensaje al usuario, sólo que este método también produce una nueva línea después del mensaje. Agregamos este método durante la implementación para dar a los clientes de la clase `Pantalla` un mayor control sobre la disposición de los mensajes visualizados. La línea 37 invoca el método utilitario `private autenticarUsuario` de la clase `ATM` (declarado en las líneas 48 a 64) para tratar de autenticar al usuario. Nos referimos al documento de requerimientos para determinar los pasos necesarios para autenticar al usuario, antes de permitir que ocurran transacciones. La línea 49 del método `autenticarUsuario` invoca al método `mostrarMensaje` de la `pantalla`, para pedir al usuario que introduzca un número de cuenta. La línea 50 invoca el método `obtenerEntrada` del `teclado` para obtener la entrada del usuario, y después almacena el valor entero introducido por el usuario en una variable local llamada `numeroCuenta`. A continuación, el método `autenticarUsuario` pide al usuario que introduzca un NIP (línea 51), y almacena el NIP introducido por el usuario en la variable local `nip` (línea 52). En seguida, la línea 55 trata de autenticar al usuario pasando el `numeroCuenta` y `nip` introducidos por el usuario al método `autenticarUsuario` de la `baseDatosBanco`. La clase `ATM` establece su atributo `usuarioAutenticado` al valor booleano devuelto por este método; `usuarioAutenticado` se vuelve `true` si la autenticación tiene éxito (es decir, si `numeroCuenta` y `nip` coinciden con los de una `Cuenta` existente en `baseDatosBanco`), y permanece como `false` en caso contrario. Si `usuarioAutenticado` es `true`, la línea 59 guarda el número de cuenta introducido por el usuario (es decir, `numeroCuenta`) en el atributo `numeroCuentaActual` del `ATM`. Los otros métodos de `ATM` usan esta variable cada vez que una sesión del `ATM` requiere acceso al número de cuenta del usuario. Si `usuarioAutenticado` es `false`, la línea 62 usa el método `mostrarLineaMensaje` de `pantalla` para indicar que se introdujo un número de cuenta inválido y/o NIP, por lo que el usuario debe intentar de nuevo. Establecemos `numeroCuentaActual` sólo después de autenticar el número de cuenta del usuario y el NIP asociado; si la base de datos no puede autenticar al usuario, `numeroCuentaActual` permanece como `0`. Después de que el método `run` intenta autenticar al usuario (línea 37), si `usuarioAutenticado` sigue siendo `false`, el ciclo `while` en las líneas 35 a 38 se ejecuta de nuevo. Si ahora `usuarioAutenticado` es `true`, el ciclo termina y el control continúa en la línea 40, que llama al método utilitario `realizarTransacciones` de la clase `ATM`.



Realizar transacciones

El método `realizarTransacciones` (líneas 67 a 98) lleva a cabo una sesión con el ATM para un usuario autenticado. La línea 69 declara una variable local del tipo `Transaccion` a la que asignaremos un objeto `SolicitudSaldo`, `Retiro` o `Deposito`, el cual representa la transacción del ATM que el usuario seleccionó. Usamos una variable del tipo `Transaccion` para poder sacar provecho del polimorfismo. Además, nombramos a esta variable con base en el *nombre de rol* incluido en el diagrama de clases de la figura 8: `transaccionActual`. La línea 71 declara otra variable booleana llamada `usuarioSalio`, la cual lleva el registro que indica si el usuario ha elegido salir o no. Esta variable controla un ciclo `while` (líneas 74 a 97), el cual permite al usuario ejecutar un número ilimitado de transacciones antes de que elija salir del sistema. Dentro de este ciclo, la línea 76 muestra el menú principal y obtiene la selección del menú del usuario al llamar a un método utilitario de ATM, llamado `mostrarMenuPrincipal` (declarado en las líneas 101 a 109). Este método muestra el menú principal invocando a los métodos de la pantalla del ATM, y devuelve una selección del menú que obtiene del usuario, a través del teclado del ATM. La línea 76 almacena la selección del usuario devuelta por `mostrarMenuPrincipal` en la variable local `seleccionMenuPrincipal`. Después de obtener una selección del menú principal, el método `realizarTransacciones` usa una instrucción `switch` (líneas 79 a 96) para responder a esa selección en forma apropiada. Si `seleccionMenuPrincipal` es igual a cualquiera de las tres constantes enteras que representan los tipos de transacciones (es decir, si el usuario elige realizar una transacción), la línea 85 llama al método utilitario `crearTransaccion` (declarado en las líneas 112 a 129) para regresar un objeto recién instanciado del tipo que corresponde a la transacción seleccionada. A la variable `transaccionActual` se le asigna la referencia devuelta por `crearTransaccion`, y después la línea 87 invoca al método `ejecutar` de esta transacción para ejecutarla. En breve hablaremos sobre el método `ejecutar` de `Transaccion` y sobre las tres subclases de `Transaccion`. Asignamos a la variable `transaccionActual` de `Transaccion` un objeto de una de las tres subclases de `Transaccion`, de modo que podamos ejecutar las transacciones mediante el *polimorfismo*. Por ejemplo, si el usuario opta por realizar una solicitud de saldo, `seleccionMenuPrincipal` es igual a `SOLICITUD_SALDO`, lo cual conduce a que `crearTransaccion` devuelva un objeto `SolicitudSaldo`. Por ende, `transaccionActual` se refiere a una `SolicitudSaldo`, y la invocación de `transaccionActual.ejecutar()` produce como resultado la invocación a la versión de `ejecutar` que corresponde a `SolicitudSaldo`.

Creación de una transacción

El método `crearTransaccion` (líneas 112 a 129) usa una instrucción `switch` (líneas 116 a 126) para instanciar un nuevo objeto de la subclase `Transaccion` del tipo indicado por el parámetro `tipo`. Recuerde que el método `realizarTransacciones` pasa la `seleccionMenuPrincipal` a este método sólo cuando `seleccionMenuPrincipal` contiene un valor que corresponde a uno de los tres tipos de transacción. Por lo tanto, `tipo` es `SOLICITUD_SALDO`, `RETIRO` o `DEPOSITO`. Cada `case` en la instrucción `switch` crea una instancia de un nuevo objeto llamando al constructor de la subclase apropiada de `Transaccion`. Cada constructor tiene una lista de parámetros únicos, con base en los datos específicos requeridos para inicializar el objeto de la subclase. Un objeto `SolicitudSaldo` sólo requiere el número de cuenta del usuario actual y referencias



tanto a la pantalla como a la baseDatosBanco del ATM. Además de estos parámetros, un objeto Retiro requiere referencias al teclado y dispensadorEfectivo del ATM, y un objeto Deposito requiere referencias al teclado y la ranuraDeposito del ATM. Más adelante analizaremos las clases de transacciones con más detalle.

Salir del menú principal y procesar selecciones inválidas

Después de ejecutar una transacción (línea 87 en realizarTransacciones), usuarioSalio sigue siendo false y se repiten las líneas 74 a 97, en donde el usuario regresa al menú principal. No obstante, si un usuario selecciona la opción del menú principal para salir en vez de realizar una transacción, la línea 91 establece usuarioSalio a true, lo cual provoca que la condición del ciclo while (!usuarioSalio) se vuelva false. Este while es la instrucción final del método realizarTransacciones, por lo que el control regresa al método run que hizo la llamada. Si el usuario introduce una selección de menú inválida (es decir, que no sea un entero del 1 al 4), la línea 94 muestra un mensaje de error apropiado, usuarioSalio sigue siendo false y el usuario regresa al menú principal para intentar de nuevo.

Esperar al siguiente usuario del ATM

Cuando realizarTransacciones devuelve el control al método run, el usuario ha elegido salir del sistema, por lo que las líneas 41 y 42 reinician los atributos usuarioAutenticado y numeroCuentaActual del ATM como preparación para el siguiente usuario del ATM. La línea 43 muestra un mensaje de despedida antes de que el ATM inicie de nuevo y dé la bienvenida al nuevo usuario.

10.2. La clase Pantalla

La clase Pantalla representa la pantalla del ATM y encapsula todos los aspectos relacionados con el proceso de mostrar la salida al usuario. La clase Pantalla simula la pantalla de un ATM real mediante un monitor de computadora y muestra los mensajes de texto mediante los métodos estándar de salida a la consola System.out.print, System.out.println y System.out.printf. En este caso de estudio diseñamos la clase Pantalla de modo que tenga una operación: mostrarMensaje. Para una mayor flexibilidad al mostrar mensajes en la Pantalla, ahora declararemos tres métodos: mostrarMensaje, mostrarLineaMensaje y mostrarMontoDolares. El método mostrarMensaje (líneas 7 a 9) recibe un argumento String y lo imprime en la consola. El cursor permanece en la misma línea, lo que hace a este método apropiado para mostrar indicadores al usuario. El método mostrarLineaMensaje (líneas 12 a 14) hace lo mismo mediante System.out.println, que imprime una nueva línea para mover el cursor a la siguiente línea. Por último, el método mostrarMontoDolares (líneas 17 a 19) imprime un monto en dólares con un formato apropiado (por ejemplo, \$1,234.56). La línea 18 utiliza a System.out.printf para imprimir un valor double al que se le aplica un formato con comas para mejorar la legibilidad, junto con dos lugares decimales.



10.3. La clase Teclado

La clase **Teclado** representa el teclado del ATM y es responsable de recibir toda la entrada del usuario. Recuerde que estamos simulando este hardware, por lo que usaremos el teclado de la computadora para simular el teclado del ATM. Usamos la clase **Scanner** para obtener la entrada de consola del usuario. Un teclado de computadora contiene muchas teclas que no se encuentran en el teclado del ATM. No obstante, vamos a suponer que el usuario sólo presionará las teclas en el teclado de computadora que aparezcan también en el teclado del ATM: las teclas enumeradas del 0 al 9, y la tecla *Intro*. La línea 4 de la clase **Teclado** importa la clase **Scanner** para usarla en la clase **Teclado**. La línea 7 declara la variable **Scanner entrada** como una variable de instancia. La línea 11 en el constructor crea un nuevo objeto **Scanner** que lee la entrada del flujo de entrada estándar (**System.in**) y asigna la referencia del objeto a la variable **entrada**. El método **obtenerEntrada** (líneas 15 a 17) invoca al método **nextInt** de **Scanner** (línea 16) para devolver el siguiente entero introducido por el usuario. [Nota: el método **nextInt** puede lanzar una excepción **InputMismatchException** si el usuario introduce una entrada que no sea número entero. Puesto que el teclado del ATM real sólo permite introducir enteros, vamos a suponer que no ocurrirá una excepción y no intentaremos corregir este problema]. Recuerde que **nextInt** contiene toda la entrada utilizada por el ATM. El método **obtenerEntrada** de **Teclado** sólo devuelve el entero introducido por el usuario. Si un cliente de la clase **Teclado** requiere entrada que cumpla con ciertos criterios (por decir, un número que corresponda a una opción válida del menú), el cliente deberá realizar la comprobación de errores.

10.4. La clase DispensadorEfectivo

La clase **DispensadorEfectivo** representa el dispensador de efectivo del ATM. La línea 7 declara la constante **CUENTA_INICIAL**, la cual indica la cuenta inicial de billetes en el dispensador de efectivo cuando el ATM inicia su operación (es decir, 500). La línea 8 implementa el atributo **cuenta** (modelado en la figura 40), que lleva la cuenta del número de billetes restantes en el **DispensadorEfectivo** en cualquier momento. El constructor (líneas 11 a 13) establece **cuenta** en la cuenta inicial. **DispensadorEfectivo** tiene 2 métodos **public**: **dispensarEfectivo** (líneas 16 a 19) y **haySuficienteEfectivoDisponible** (líneas 22 a 30). La clase confía en que un cliente (es decir, **Retiro**) llamará a **dispensarEfectivo** sólo después de establecer que hay suficiente efectivo disponible mediante una llamada a **haySuficienteEfectivoDisponible**. Por ende, **dispensarEfectivo** tan sólo simula el proceso de dispensar la cantidad solicitada sin verificar si en realidad hay suficiente efectivo disponible. El método **haySuficienteEfectivoDisponible** (líneas 22 a 30) tiene un parámetro llamado **monto**, el cual especifica el monto de efectivo en cuestión. La línea 23 calcula el número de billetes de \$20 que se requieren para dispensar el monto solicitado. El ATM permite al usuario elegir sólo montos de retiro que sean múltiplos de \$20, por eso dividimos **monto** por 20 para obtener el número de **billetesRequeridos**. Las líneas 25 a 29 devuelven **true** si la cuenta del **DispensadorEfectivo** es mayor o igual a **billetesRequeridos** (es decir, si hay suficientes billetes disponibles), y **false** en caso contrario (si no hay suficientes billetes). Por ejemplo, si un usuario desea retirar \$80 (**billetesRequeridos** es 4) y sólo quedan tres billetes (**cuenta** es 3), el método devuelve **false**. El método **dispensarEfectivo** (líneas 16 a 19) simula el proceso de dispensar el efectivo. Si nuestro sistema se



conectara al hardware real de un dispensador de efectivo, este método interactuaría con el dispositivo para dispensar físicamente el efectivo. Nuestra versión del método tan sólo reduce la cuenta de billetes restantes con base en el número requerido para dispensar el monto especificado (línea 18). Es responsabilidad del cliente de la clase (es decir, **Retiro**) informar al usuario que se dispensó el efectivo; la clase **DispensadorEfectivo** no puede interactuar de manera directa con **Pantalla**.

10.5. La clase RanuraDeposito

La clase **RanuraDeposito** representa a la ranura de depósito del ATM. Al igual que la clase **DispensadorEfectivo**, la clase **RanuraDeposito** tan sólo simula la funcionalidad del hardware real de una ranura de depósito. **RanuraDeposito** no tiene atributos y sólo cuenta con un método: **seRecibioSobre** (líneas 8 a 10), el cual indica si se recibió un sobre de depósito. En el documento de requerimientos vimos que el ATM permite al usuario hasta dos minutos para insertar un sobre. La versión actual del método **seRecibioSobre** sólo devuelve **true** de inmediato (línea 9), ya que ésta es sólo una simulación de software, por lo que asumimos que el usuario insertó un sobre dentro del límite de tiempo requerido. Si se conectara el hardware de una ranura de depósito real a nuestro sistema, podría implementarse el método **seRecibioSobre** para esperar un máximo de dos minutos a recibir una señal del hardware de la ranura de depósito, indicando que en definitiva el usuario insertó un sobre de depósito. Si **seRecibioSobre** recibiera dicha señal en un tiempo máximo de dos minutos, el método devolvería **true**. Si transcurrieran los dos minutos y el método no recibiera ninguna señal, entonces devolvería **false**.

10.6. La clase Cuenta

La clase **Cuenta** representa una cuenta bancaria. Cada **Cuenta** tiene cuatro atributos (modelados en la figura 40): **numeroCuenta**, **nip**, **saldoDisponible** y **saldoTotal**. Las líneas 6 a 9 implementan estos atributos como campos **private**. La variable **saldoDisponible** representa el monto de fondos disponibles para retirar. La variable **saldoTotal** representa el monto de fondos disponibles, junto con el monto de los fondos depositados pendientes de confirmación o liberación. La clase **Cuenta** tiene un constructor (líneas 12 a 17) que recibe como argumentos un número de cuenta, el NIP establecido para la cuenta, el saldo disponible inicial y el saldo total inicial de la cuenta. Las líneas 13 a 16 asignan estos valores a los atributos de la clase (es decir, los campos). El método **validarNIP** (líneas 20 a 26) determina si un NIP especificado por el usuario (es decir, el parámetro **nipUsuario**) coincide con el NIP asociado con la cuenta (es decir, el atributo **nip**). Recuerde que modelamos el parámetro **nipUsuario** de este método en la figura 23. Si los dos NIP coinciden, el método devuelve **true** (línea 22); en caso contrario devuelve **false** (línea 24). Los métodos **obtenerSaldoDisponible** (líneas 29 a 31) y **obtenerSaldoTotal** (líneas 34 a 36) devuelven los valores de los atributos **double saldoDisponible** y **saldoTotal**, respectivamente. El método **abonar** (líneas 39 a 41) agrega un monto de dinero (el parámetro **monto**) a una **Cuenta** como parte de una transacción de depósito. Este método agrega el **monto** sólo al atributo **saldoTotal** (línea 40). El dinero abonado a una cuenta durante un depósito no se vuelve disponible de inmediato, por lo que sólo modificamos el saldo total. Supondremos que el banco actualiza después el saldo disponible de manera apropiada. Nuestra implementación de



la clase **Cuenta** sólo incluye los métodos requeridos para realizar transacciones con el ATM. Por lo tanto, omitiremos los métodos que invocaría cualquier otro sistema bancario para sumar al atributo **saldoDisponible** (confirmar un depósito) o restar del atributo **saldoTotal** (rechazar un depósito). El método **cargar** (líneas 44 a 47) resta un monto de dinero (el parámetro **monto**) de una **Cuenta**, como parte de una transacción de retiro. Este método resta el **monto** tanto del atributo **saldoDisponible** (línea 45) como del atributo **saldoTotal** (línea 46), debido a que un retiro afecta ambas unidades del saldo de una cuenta. El método **obtenerNúmeroCuenta** (líneas 50 a 52) proporciona acceso al **numeroCuenta** de una **Cuenta**. Incluimos este método en nuestra implementación de modo que un cliente de la clase (por ejemplo, **BaseDatosBanco**) pueda identificar a una **Cuenta** específica. Por ejemplo, **BaseDatosBanco** contiene muchos objetos **Cuenta**, y puede invocar este método en cada uno de sus objetos **Cuenta** para localizar el que tenga cierto número de cuenta específico.

10.7. La clase **BaseDatosBanco**

La clase **BaseDatosBanco** modela la base de datos del banco con la que el ATM interactúa para acceder a la información de la cuenta de un usuario y modificarla. Como aún no hemos estudiado el acceso a las bases de datos en Java, por ahora implementaremos la base de datos con un arreglo. Más adelante podrá volver a implementar esta parte del ATM, usando una base de datos real. Determinaremos un atributo de tipo referencia para la clase **BaseDatosBanco** con base en su relación de composición con la clase **Cuenta**. En la figura 39 vimos que una **BaseDatosBanco** está compuesta de cero o más objetos de la clase **Cuenta**. La línea 6 implementa el atributo **cuentas** (un arreglo de objetos **Cuenta**) para implementar esta relación de composición. La clase **BaseDatosBanco** tiene un constructor sin argumentos (líneas 9 a 13) que inicializa **cuentas** para que contenga un conjunto de nuevos objetos **Cuenta**. A fin de probar el sistema, instanciamos **cuentas** de modo que contenga sólo dos elementos en el arreglo (línea 10), que instanciamos como nuevos objetos **Cuenta** con datos de prueba (líneas 11 y 12). El constructor de **Cuenta** tiene cuatro parámetros: el número de cuenta, el NIP asignado a la cuenta, el saldo disponible inicial y el saldo total inicial. Recuerde que la clase **BaseDatosBanco** sirve como intermediario entre la clase **ATM** y los mismos objetos **Cuenta** que contienen la información sobre la cuenta de un usuario. Por ende, los métodos de la clase **BaseDatosBanco** no hacen más que invocar a los métodos correspondientes del objeto **Cuenta** que pertenece al usuario actual del ATM. Incluimos el método **private** utilitario **obtenerCuenta** (líneas 16 a 25) para permitir que la **BaseDatosBanco** obtenga una referencia a una **Cuenta** específica dentro del arreglo **cuentas**. Para localizar la **Cuenta** del usuario, la **BaseDatosBanco** compara el valor devuelto por el método **obtenerNúmeroCuenta** para cada elemento de **cuentas** con un número de cuenta específico, hasta encontrar una coincidencia. Las líneas 18 a 23 recorren el arreglo **cuentas**. Si el número de cuenta de **cuentaActual** es igual al valor del parámetro **numeroCuenta**, el método devuelve de inmediato la **cuentaActual**. Si ninguna cuenta tiene el número de cuenta dado, entonces la línea 24 devuelve **null**. El método **autenticarUsuario** (líneas 29 a 38) aprueba o desaprueba la identidad de un usuario del ATM. Este método recibe un número de cuenta y un NIP especificados por el usuario como argumentos, e indica si coinciden con el número de cuenta y NIP de una **Cuenta** en la base de datos. La línea 31 llama al método **obtenerCuenta**, el cual devuelve



una Cuenta con numeroCuentaUsuario como su número de cuenta, o null para indicar que el numeroCuentaUsuario es inválido. Si obtenerCuenta devuelve un objeto Cuenta, la línea 34 regresa el valor boolean devuelto por el método validarNIP de ese objeto. El método autenticarUsuario de BaseDatosBanco no realiza la comparación de NIP por sí solo, sino que envía nipUsuario al método validarNIP del objeto Cuenta para que lo haga. El valor devuelto por el método validarNIP de Cuenta indica si el NIP especificado por el usuario coincide con el NIP de la Cuenta del usuario, por lo que el método autenticarUsuario simplemente devuelve este valor al cliente de la clase (es decir, el ATM). BaseDatosBanco confía en que el ATM invoque al método autenticarUsuario y reciba un valor de retorno true antes de permitir que el usuario realice transacciones. BaseDatosBanco también confía que cada objeto Transaccion creado por el ATM contendrá el número de cuenta válido del usuario actual autenticado, y que éste será el número de cuenta que se pase al resto de los métodos de BaseDatosBanco como el argumento numeroCuentaUsuario. Por lo tanto, los métodos obtenerSaldoDisponible (líneas 41 a 43), obtenerSaldoTotal (líneas 46 a 48), abonar (líneas 51 a 53) y cargar (líneas 56 a 58) tan sólo obtienen el objeto Cuenta del usuario con el método utilitario obtenerCuenta, y después invocan al método de Cuenta apropiado con base en ese objeto. Sabemos que las llamadas a obtenerCuenta desde estos métodos nunca devolverán null, puesto que numeroCuentaUsuario se debe referir a una Cuenta existente. Los métodos obtenerSaldoDisponible y obtenerSaldoTotal devuelven los valores que regresan los correspondientes métodos de Cuenta. Además, abonar y cargar simplemente redirigen el parámetro monto a los métodos de Cuenta que invocan.

10.8. La clase Transaccion

La clase Transaccion es una superclase abstracta que representa la noción de una transacción con el ATM. Contiene las características comunes de las subclases SolicitudSaldo, Retiro y Deposito. Esta clase se expande a partir del código “estructural” que se desarrolló por primera vez en la sección 9. La línea 4 declara esta clase como **abstract**. Las líneas 6 a 8 declaran los atributos **private** de las clases. En el diagrama de clases de la figura 40 vimos que la clase Transaccion contiene un atributo llamado numeroCuenta (línea 6), el cual indica la cuenta involucrada en la Transaccion. Luego derivamos los atributos de Pantalla (línea 7) y de BaseDatosBanco (línea 8) de la clase Transaccion asociada y que se modela en la figura 39. Todas las transacciones requieren acceso a la pantalla del ATM y a la base de datos del banco. La clase Transaccion tiene un constructor (líneas 11 a 15) que recibe como argumentos el número de cuenta del usuario actual y referencias tanto a la pantalla del ATM como a la base de datos del banco. Puesto que Transaccion es una clase abstracta, este constructor se llama sólo a través de los constructores de las subclases de Transaccion. La clase tiene tres *métodos obtener* públicos: obtenerNumeroCuenta (líneas 18 a 20), obtenerPantalla (líneas 23 a 25) y obtenerBaseDatosBanco (líneas 28 a 30). Estos métodos son heredados por las subclases de Transaccion y se utilizan para obtener acceso a los atributos **private** de esta clase. La clase Transaccion también declara el método **abstract ejecutar** (línea 33). No tiene caso proveer la implementación de este método, ya que una transacción genérica no se puede ejecutar. Por ende, declaramos este método como **abstract** y forzamos a cada subclase de Transaccion a proveer una implementación concreta que ejecute este tipo específico de transacción.



10.9. La clase SolicitudSaldo

La clase `SolicitudSaldo` extiende a `Transaccion` y representa una transacción de solicitud de saldo con el ATM. `SolicitudSaldo` no tiene atributos propios, pero hereda los atributos `numeroCuenta`, `pantalla` y `baseDatosBanco` de `Transaccion`, a los cuales se puede acceder por medio de los *métodos obtener* públicos de `Transaccion`. El constructor de `SolicitudSaldo` recibe los argumentos correspondientes a estos atributos, y lo único que hace es reenviarlos al constructor de `Transaccion` mediante el uso de `super` (línea 8). La clase `SolicitudSaldo` sobrescribe el método abstracto `ejecutar` de `Transaccion` para proveer una implementación concreta (líneas 13 a 31) que realiza los pasos involucrados en una solicitud de saldo. Las líneas 15 y 16 obtienen referencias a la base de datos del banco y la pantalla del ATM, al invocar a los métodos heredados de la superclase `Transaccion`. La línea 19 obtiene el saldo disponible de la cuenta implicada, mediante una invocación al método `obtenerSaldoDisponible` de `baseDatosBanco`. La línea 19 usa el método heredado `obtenerNumeroCuenta` para obtener el número de cuenta del usuario actual, que después pasa a `obtenerSaldoDisponible`. La línea 22 obtiene el saldo total de la cuenta del usuario actual. Las líneas 25 a 30 muestran la información del saldo en la pantalla del ATM. Recuerde que `mostrarMontoDolares` recibe un argumento `double` y lo imprime en la pantalla, con formato de monto en dólares. Por ejemplo, si el `saldoDisponible` de un usuario es 1000.5, la línea 27 imprime \$1,000.50. La línea 30 inserta una línea en blanco de salida para separar la información del saldo de la salida subsiguiente (es decir, el menú principal repetido por la clase `ATM` después de ejecutar la `SolicitudSaldo`).

10.10. La clase Retiro

La clase `Retiro` extiende a `Transaccion` y representa una transacción de retiro del ATM. Esta clase se expande a partir del código “estructural” para la misma, desarrollado en la figura 42. En el diagrama de clases de la figura 40 vimos que la clase `Retiro` tiene un atributo, `monto`, que la línea 6 implementa como campo `int`. La figura 39 modela las asociaciones entre la clase `Retiro` y las clases `Teclado` y `DispensadorEfectivo`, para las que las líneas 7 y 8 implementan los atributos de tipo referencia `teclado` y `dispensadorEfectivo`, respectivamente. La línea 10 declara una constante que corresponde a la opción de cancelación en el menú. Pronto veremos cómo es que la clase utiliza esta constante. El constructor de la clase `Retiro` (líneas 13 a 20) tiene cinco parámetros. Usa `super` para pasar los parámetros `numeroCuentaUsuario`, `pantallaATM` y `baseDatosBancoATM` al constructor de la superclase `Transaccion` para establecer los atributos que `Retiro` hereda de `Transaccion`. El constructor también recibe las referencias `tecladoATM` y `dispensadorEfectivoATM` como parámetros, y las asigna a los atributos de tipo referencia `teclado` y `dispensadorEfectivo`. La clase `Retiro` sobrescribe el método `ejecutar` de `Transaccion` con una implementación concreta (líneas 24 a 70) que realiza los pasos de un retiro. La línea 25 declara e inicializa una variable `boolean` local llamada `efectivoDispensado`, la cual indica si se dispensó efectivo (es decir, si la transacción se completó con éxito), y en un principio es `false`. La línea 26 declara la variable `double` local llamada `saldoDisponible`, la cual almacenará el saldo disponible del usuario durante una transacción de retiro. Las líneas 29 y 30 obtienen referencias a la base de datos del banco y a la pantalla del ATM, invocando los métodos heredados de la superclase `Transaccion`. Las líneas 33 a 69 contienen un ciclo `do-while` que ejecuta su cuerpo hasta dispensar efectivo (es decir,



hasta que `efectivoDispensado` se vuelva `true`) o hasta que el usuario elija cancelar (en cuyo caso, el ciclo termina). Usamos este ciclo para regresar en forma continua al usuario al inicio de la transacción en caso de que ocurra un error (por ejemplo, si el monto de retiro solicitado es mayor que el saldo disponible del usuario, o mayor que la cantidad de efectivo en el dispensador). La línea 35 muestra un menú de montos de retiro y obtiene una selección del usuario mediante una llamada al método `private utilitario mostarMenuDeMontos` (declarado en las líneas 74 a 114). Este método muestra el menú de montos y devuelve un monto de retiro `int`, o la constante `int CANCEL0` para indicar que el usuario optó por cancelar la transacción. El método `mostrarMenuDeMontos` (líneas 74 a 114) declara primero la variable local `opcionUsuario` (que en un principio vale 0) para almacenar el valor que devolverá el método (línea 75). La línea 77 obtiene una referencia a la pantalla mediante una llamada al método `obtenerPantalla` heredado de la superclase `Transaccion`. La línea 80 declara un arreglo entero de montos de retiro que corresponden a los montos mostrados en el menú de retiro. Ignoramos el primer elemento en el arreglo (índice 0), debido a que el menú no tiene opción 0. La instrucción `while` en las líneas 83 a 111 se repite hasta que `opcionUsuario` recibe un valor distinto de 0. En breve veremos que esto ocurre cuando el usuario realiza una selección válida del menú. Las líneas 85 a 92 muestran el menú de retiro en la pantalla y piden al usuario que introduzca una opción. La línea 94 obtiene el entero `entrada` por medio del teclado. La instrucción `switch` en las líneas 97 a 110 determina cómo proceder con base en la entrada del usuario. Si éste selecciona un número entre 1 y 5, la línea 103 establece `opcionUsuario` en el valor que `montos` contiene en el índice `entrada`. Por ejemplo, si el usuario introduce 3 para retirar \$60, la línea 103 establece `opcionUsuario` en el valor de `montos[3]` (es decir, 60). La línea 104 termina la instrucción `switch`. La variable `opcionUsuario` ya no es igual a 0, por lo que la instrucción `while` en las líneas 83 a 111 termina y la línea 113 devuelve `opcionUsuario`. Si el usuario selecciona la opción del menú para cancelar, se ejecutan las líneas 106 y 107, las cuales establecen `opcionUsuario` en `CANCEL0` y hacen que el método devuelva este valor. Si el usuario no introduce una selección válida del menú, la línea 109 muestra un mensaje de error y el usuario regresa al menú de retiro. La línea 38 en el método `ejecutar` determina si el usuario seleccionó un monto de retiro o eligió cancelar. Si el usuario cancela, se ejecutan las líneas 66 y 67 para mostrar un mensaje apropiado al usuario, antes de devolver el control al método que hizo la llamada (el método `realizarTransacciones` de ATM). Si el usuario selecciona un monto de retiro, la línea 40 obtiene el saldo disponible de la `Cuenta` del usuario actual y lo almacenan en la variable `saldoDisponible`. Luego, la línea 43 determina si el monto seleccionado es menor o igual que el saldo disponible del usuario. Si no es así, la línea 61 muestra un mensaje de error apropiado. Después el control continúa hasta el final del ciclo `do-while`, y éste se repite debido a que `efectivoDispensado` sigue siendo `false`. Si el saldo del usuario es suficientemente alto, la instrucción `if` en la línea 45 determina si el dispensador de efectivo tiene dinero suficiente para cumplir con la solicitud de retiro, invocando al método `haySuficienteEfectivoDisponible` de `dispensadorEfectivo`. Si este método devuelve `false`, la línea 56 muestra un mensaje de error apropiado y se repite el ciclo `do-while`. Si hay suficiente efectivo disponible, entonces se cumplen los requisitos para el retiro y la línea 47 carga `monto` a la cuenta del usuario en la base de datos. Despues, las líneas 49 y 50 instruyen al dispensador de efectivo para que dispense el efectivo al usuario y establecen `efectivoDispensado` en `true`. Por último, la línea 52



muestra un mensaje al usuario para indicarle que se dispensó el efectivo. Puesto que ahora `efectivoDispensado` es `true`, el control continúa después del ciclo `do-while`. No aparecen instrucciones adicionales debajo del ciclo, por lo que el método regresa.

10.11. La clase Deposito

La clase `Deposito` extiende a `Transaccion` y representa una transacción de depósito. En la figura 40 vimos que la clase `Deposito` tiene un atributo llamado `monto`, el cual se implementa en la línea 6 como un campo `int`. Las líneas 7 y 8 crean los atributos de referencia `teclado` y `ranuraDeposito`, que implementan las asociaciones entre la clase `Deposito` y las clases `Teclado` y `RanuraDeposito` modeladas en la figura 39. La línea 9 declara una constante `CANCELO`, la cual corresponde al valor que introduce un usuario para cancelar. Pronto veremos cómo es que la clase usa esta constante. Al igual que `Retiro`, la clase `Deposito` contiene un constructor (líneas 12 a 19) que pasa tres parámetros al constructor de la superclase `Transaccion`. El constructor también tiene los parámetros `tecladoATM` y `ranuraDepositoATM`, que asigna a los atributos correspondientes (líneas 17 y 18). El método `ejecutar` (líneas 23 a 56) sobrescribe la versión `abstract` en la superclase `Transaccion` con una implementación concreta que realiza los pasos requeridos en una transacción de depósito. Las líneas 24 y 25 obtienen referencias a la base de datos y la pantalla. La línea 27 pide al usuario que introduzca un monto de depósito, para lo cual invoca al método `private` utilitario `pedirMontoADepositar` (declarado en las líneas 59 a 72) y establece el atributo `monto` con el valor devuelto. El método `pedirMontoADepositar` pide al usuario que introduzca un monto de depósito como un número entero de centavos (debido a que el teclado del ATM no contiene un punto decimal; esto es consistente con muchos ATM reales) y devuelve el valor `double` que representa el monto en dólares a depositar. La línea 60 en el método `pedirMontoADepositar` obtiene una referencia a la pantalla del ATM. La línea 63 muestra un mensaje que pide al usuario introducir un monto de depósito como un número de centavos, o “0” para cancelar la transacción. La línea 64 obtiene la entrada del usuario desde el teclado. Las líneas 67 a 71 determinan si el usuario introdujo un monto de depósito real o eligió cancelar. Si optó por esto último, la línea 68 devuelve la constante `CANCELO`. De lo contrario, la línea 70 devuelve el monto de depósito después de convertirlo del número de centavos a un monto en dólares, mediante la conversión de entrada a `double`, y después lo divide por 100. Por ejemplo, si un usuario introduce 125 como el número de centavos, la línea 70 devuelve el resultado de dividir 125.0 por 100, o 1.25; 125 centavos son \$1.25. Las líneas 30 a 55 en el método `ejecutar` determinan si el usuario eligió cancelar la transacción en vez de introducir un monto de depósito. Si el usuario cancela, la línea 54 muestra un mensaje apropiado y el método regresa. Si el usuario introduce un monto de depósito, las líneas 32 a 34 instruyen al usuario para que inserte un sobre de depósito con el monto correcto. Recuerde que el método `mostrarMontoDolares` de `Pantalla` imprime un valor `double` con formato de monto en dólares. La línea 37 establece una variable `boolean` local `seRecibioSobre` en el valor devuelto por el método `seRecibioSobre` de `ranuraDeposito`, para indicar si se recibió un sobre de depósito. Recuerde que codificamos el método `seRecibioSobre` (líneas 8 a 10 de `RanuraDeposito.java`) para que siempre devuelva `true`, debido a que estamos simulando la funcionalidad de la ranura de depósito y suponemos que el usuario siempre inserta un sobre. Sin embargo, codificamos el método `ejecutar` de la clase `Deposito` para que evalúe la posibilidad de que el usuario no inserte un sobre; la



buen ingeniería de software exige que los programas tomen en cuenta todos los posibles valores de retorno. Por ende, la clase `Deposito` está preparada para futuras versiones de `seRecibioSobre` que pudieran devolver `false`. Las líneas 41 a 45 se ejecutan si la ranura de depósito recibe un sobre. Las líneas 41 a 43 muestran un mensaje apropiado al usuario. Después, la línea 45 abona el monto de depósito a la cuenta del usuario en la base de datos. La línea 49 se ejecutará si la ranura de depósito no recibe un sobre de depósito. En este caso, mostramos un mensaje al usuario para indicarle que el ATM canceló la transacción. A continuación, el método regresa sin modificar la cuenta del usuario.

10.12. La clase CasoEstudioATM

La clase `CasoEstudioATM` es una clase simple que nos permite iniciar, o “encender”, el ATM y evaluar la implementación de nuestro modelo del sistema ATM. El método `main` de la clase `CasoEstudioATM` (líneas 7 a 10) no hace nada más que instanciar un nuevo objeto ATM llamado `elATM` (línea 8), e invoca a su método `run` (línea 9) para iniciar el ATM.

10.13. Descarga del código fuente completo

El código fuente completo descrito anteriormente puede ser descargado de Internet accediendo al siguiente enlace:

<http://www.mediafire.com/download/1n2c5x4pdb7t9db/ATM.zip>



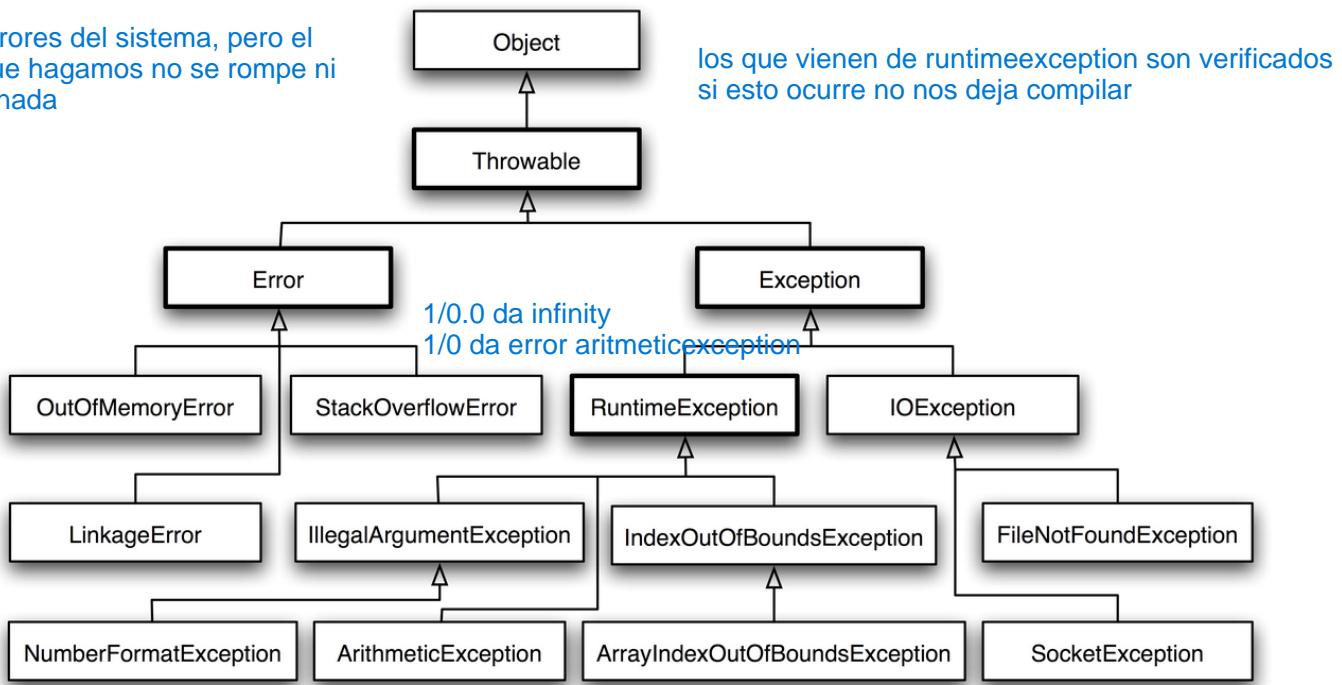
Unidad 7 Excepciones

En Java, los errores que se producen durante la ejecución de un programa pueden ser tratados mediante un poderoso mecanismo denominado **manejo de excepciones**, que permite evitar la terminación anormal del programa y continuar con su ejecución.

Las excepciones son objetos pertenecientes a la siguiente jerarquía de clases (vista parcial):

estos son errores del sistema, pero el programa que hagamos no se rompe ni el código ni nada

los que vienen de runtimeexception son verificados si esto ocurre no nos deja compilar



Las excepciones se *lanzan* (generan) automáticamente cuando ocurre un error o manualmente mediante el uso de una sentencia `throw`.

Existen dos tipos de excepciones: **verificadas (checked)** y **no verificadas (unchecked)**. Las excepciones verificadas extienden `Exception` (pero no `RuntimeException`) y, para poder compilar el programa, deben ser declaradas (con `throws`) o capturadas (con `try..catch..finally`). Las excepciones no verificadas extienden `Error` o `RuntimeException`, y no es obligatorio declararlas ni capturarlas.

En el bloque `try` se encierra el código que podría provocar errores durante la ejecución y lanzar excepciones. En el o los bloque(s) `catch` se encierra el código que se ejecuta cuando se captura una excepción del tipo indicado en el parámetro. En el bloque `finally` (que es optativo cuando ya se definió, por lo menos, un bloque `catch`) se encierra el código que debe ejecutarse siempre, se haya o no lanzado y capturado una excepción. El bloque `try` no puede aparecer solo: debe preceder a, por lo menos, un bloque `catch` o al bloque `finally`. Si se definen dos o más bloques `catch`, su orden no es arbitrario: los bloques que capturan las excepciones más específicas deben aparecer primero y los que capturan las más generales debe aparecer por último.



El siguiente ejemplo muestra prácticamente todos los usos posibles de las excepciones:

```
package excepciones;

public class AprobadoException extends Exception {
    public AprobadoException(String message) {
        super(message);
    }
}
```

Annotations: 1 points to the line 'extends Exception' and 2 points to the line 'super(message);'

```
package excepciones;

public class ReprobadoException extends Exception {
    public ReprobadoException(String message) {
        super(message);
    }
}
```

```
package excepciones;

import java.io.FileNotFoundException;

public class Alumno {

    private String nombre;
    private int nota;

    public Alumno(String nombre, int nota) {
        this.nombre = nombre;
        this.nota = nota;
    }

    public String getNombre() {
        return nombre;
    }

    public int getNota() {
        return nota;
    }

    public void sonreir() {
        int x = 1/0;
    }

    public void llorar() throws FileNotFoundException {
        java.io.FileReader fileReader = new java.io.FileReader("noexiste.fil");
    }
}
```

Annotations: 3 points to the line 'int x = 1/0;' and 4 points to the line 'throws FileNotFoundException'.



```
package excepciones;

public class Main {

    public static void main(String[] args) {
        Alumno a1 = new Alumno("Pedro", );
        Alumno a2 = new Alumno("Pablo", );

        try {
            if (a1.getNota() >= 4) {
                throw new AprobadoException(a1.getNombre() + " ha aprobado!");
            } else {
                throw new ReprobadoException(a1.getNombre() + " fue reprobado!");
            }
        } catch (AprobadoException | ReprobadoException ex) { 7
            System.out.println(ex.getMessage());
        }

        if (a2.getNota() >= 4) {
            try {
                a2.sonreir();
            } catch (RuntimeException ex) { 9
                System.out.println("Sonrie " + a2.getNombre() + "...");
            } catch (Exception ex) { 10
                System.out.println("Se rie " + a2.getNombre() + "...");
            } finally {
                System.out.println("Porque se ha sacado un " + a2.getNota() + "!");
            }
        } else {
            try {
                a2.llorar();
            } catch (Exception ex) {
                System.out.println("Llora " + a2.getNombre() + "...");
            } finally {
                System.out.println("Porque le pusieron un " + a2.getNota() + "!");
            }
        }
    }
}
```

El sistema está compuesto por las siguientes cuatro clases: **Alumno**, que posee dos atributos (**nombre** y **nota**) con sus correspondientes *getters* y dos comportamientos (**sonreir** y **llorar**), **AprobadoException** y **ReprobadoException**, que son clases de excepciones definidas *ad-hoc* y, por último, **Main**, donde se instancian dos alumnos **a1** y **a2**, y donde, además, según cómo sea la nota de **a1**, se lanza una de las dos excepciones anteriores y según cómo sea la nota de **a2**, se le envía a éste un mensaje **sonreir** o **llorar**. Estos dos últimos métodos lanzan excepciones estándar de Java, las cuales son capturadas en **Main**.



OBSERVACIONES

1) Complete **⑤** y **⑥** con los valores indicados y registre la salida obtenida en cada caso:

1.a) **⑤**: 1 y **⑥**: 10

.....
.....
.....

1.b) **⑤**: 2 y **⑥**: 1

.....
.....
.....

1.c) **⑤**: 8 y **⑥**: 5

.....
.....
.....

1.d) **⑤**: 6 y **⑥**: 2

.....
.....
.....

2) Por extender `Exception` (**①**), ¿qué tipo de excepción es `AprobadoException`?
¿Cómo es manejada en este sistema?

.....

3) ¿El constructor de cuál clase está siendo invocado en **②**?

.....

4) ¿Por qué en **③** no se declara (con `throws`) que el método lanza una excepción?

.....

5) ¿Por qué en **④** se tuvo que declarar que el método lanza esta excepción?

.....

6) La captura múltiple declarada en **⑦**, ¿a cuál captura simple es preferible?

.....

7) ¿Qué parte del bloque `try..catch..finally` se ejecuta igual si se borra o comenta **⑧**?

.....

8) ¿Qué ocurre si se intercambian **⑨** y **⑩**? ¿Por qué?

.....

9) ¿Qué excepción más específica podría usarse en **⑨** en lugar de `RuntimeException`?

.....

10) ¿Qué excepción más específica podría usarse en **⑪** en lugar de `Exception`?

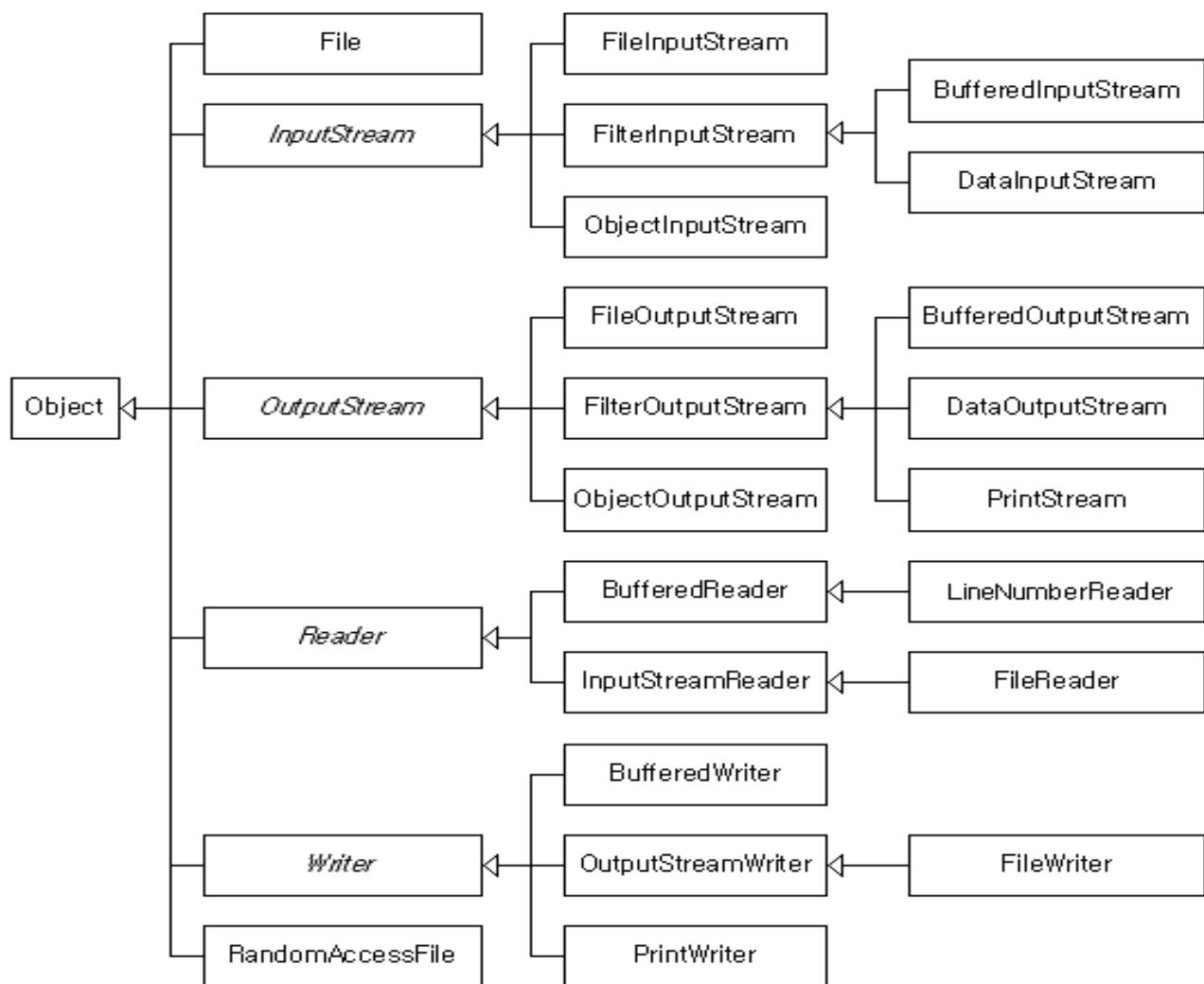
.....

Unidad 8

Archivos

Java proporciona un amplio conjunto de clases e interfaces para manipular archivos. El paquete `java.io` contiene 51 clases (por ejemplo: `File`), 12 interfaces (por ejemplo: `Serializable`), 16 excepciones (por ejemplo: `FileNotFoundException`) y un error (`IOException`). Otros paquetes también proveen clases e interfaces que sirven para trabajar con archivos: `java.util.zip` contiene clases para trabajar con archivos comprimidos (por ejemplo: `ZipInputStream`), `javax.sound.sampled` contiene clases para trabajar con archivos de audio (por ejemplo: `AudioInputStream`), etc..

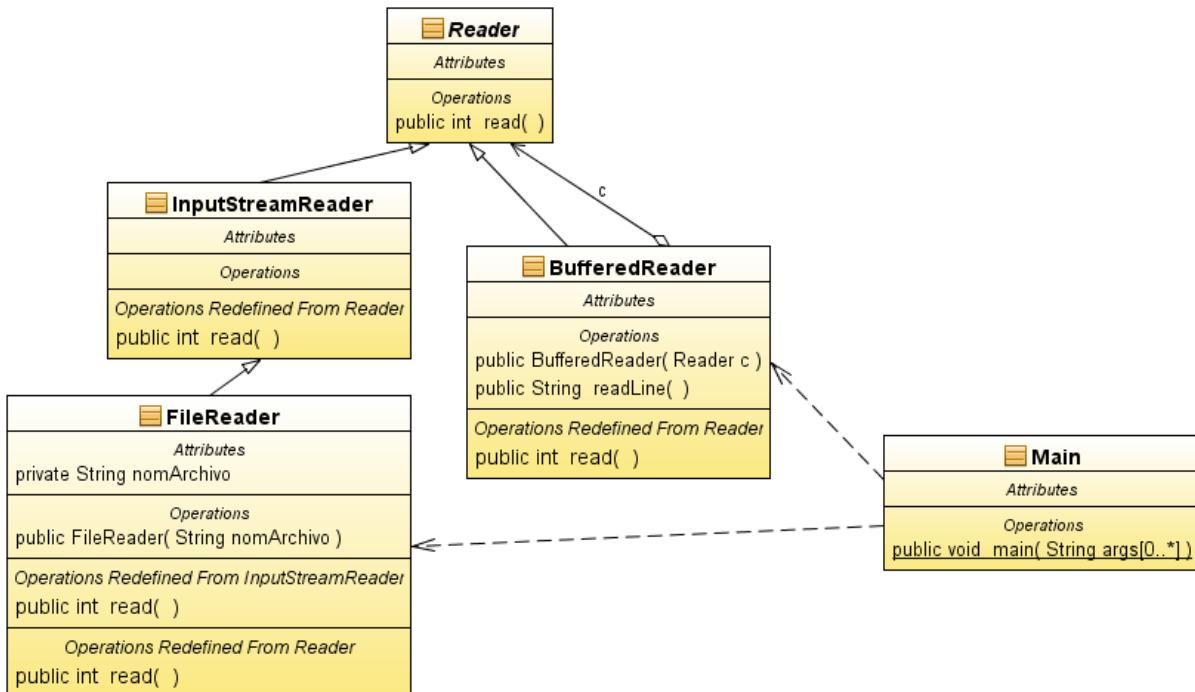
Las siguientes son algunas de las clases disponibles en `java.io`:



Los diseñadores del lenguaje pensaron que utilizar la herencia para componer las combinaciones de funcionalidades más comunes requeridas para el manejo de archivos habría resultado en una verdadera *explosión de clases*. Por eso, diseñaron las clases para el manejo de archivos siguiendo el *patrón de diseño Decorator*. En lugar de usar la herencia para agregar funcionalidad a las clases, este patrón permite agregar funcionalidad a los objetos individuales (en tiempo de ejecución).



El siguiente ejemplo, basado en el diagrama anterior, muestra cómo está constituido el *patrón de diseño Decorator*:



Reader es la superclase abstracta común para los componentes concretos y los decoradores:

```
package decorator;
public abstract class Reader {
    public int read() {
        return 32 + 32 + 1;
    }
}
```

InputStreamReader y **FileReader** son las clases de los componentes concretos que pueden ser decorados en tiempo de ejecución:

```
package decorator;
public class InputStreamReader extends Reader {
    @Override
    public int read() {
        return super.read();
    }
}
```

```
package decorator;
public class FileReader extends InputStreamReader {
    private String nomArchivo;
    public FileReader(String nomArchivo) {
        this.nomArchivo = nomArchivo;
    }
    @Override
    public int read() {
        System.out.println("read() de FileReader");
        System.out.println("Devuelve un carácter del archivo: " + nomArchivo);
        return super.read();
    }
}
```



El patrón **Decorator** original establece que debe haber una clase abstracta extendida de la primera para actuar como superclase de los decoradores. En este ejemplo sólo hay un decorador (`BufferedReader`) y, por lo tanto, su clase hereda directamente de la primera clase abstracta (`Reader`):

```
package decorator;

public class BufferedReader extends Reader {
    private Reader c;
    public BufferedReader(Reader c) {
        this.c = c;
    }
    @Override
    public int read() {
        System.out.println("read() de BufferedReader");
        System.out.println("Devuelve un carácter del buffer cargado a partir de: " + c);
        return 65;
    }
    public String readLine() {
        System.out.println("readLine() de BufferedReader");
        System.out.println("Devuelve un renglón del buffer cargado a partir de: " + c);
        return "Hola mundo";
    }
}
```

La clave de este patrón de diseño es que el objeto decorador y el objeto decorado tienen en común la misma superclase (tienen la misma relación *is-a*), y que el objeto decorador tiene como atributo una referencia a un objeto de esa misma superclase (tienen una relación *has-a*). Por ello, el objeto decorador, al ser construido, recibe una referencia a un objeto que será decorado y, luego de inicializar con ella su atributo, puede ejecutar tanto los métodos del objeto decorado (por *delegación*), como los propios (heredados de sus superclases o definidos en su propia clase).

En el método `main` de la clase `Main`, se crea un objeto de la clase `FileReader`, se guarda una referencia al mismo en la variable `fr` y se utiliza su método `read` para leer un carácter (un `int` que debe convertirse explícitamente a `char`). Luego se instancia un segundo objeto de la misma clase, pero en lugar de usar su variable de referencia `fr2` para invocar métodos, la misma se utiliza como argumento para construir un objeto de la clase `BufferedReader`, con cuya variable de referencia `br` es posible invocar no sólo el método `read`, sino también el método `readLine`.

```
package decorator;
public class Main {
    public static void main(String[] args) {
        String nomArch = "hola.txt";
        FileReader fr = new FileReader(nomArch);
        System.out.println((char) fr.read());
        System.out.println();
        FileReader fr2 = new FileReader(nomArch);
        BufferedReader br = new BufferedReader(fr2);
        System.out.println((char) br.read());
        System.out.println(br.readLine());
    }
}
```



El ejemplo visto sirve para entender el funcionamiento del patrón **Decorator**, ya que es posible observar cómo el objeto `br` se obtiene decorando el objeto `fr2`.

PREGUNTA DE REPASO

¿Por qué en vez de crear un `FileReader` y después decorarlo con un `BufferedReader`:

```
FileReader fr2 = new FileReader(nomArch);
BufferedReader br = new BufferedReader(fr2);
```

no se crea directamente un `BufferedReader`:

```
BufferedReader br = new BufferedReader(nomArch);
```

si, al fin y al cabo, los métodos utilizados están disponibles en este último?

Aunque fue útil para ver el patrón de diseño en que se basan las clases del paquete `java.io`, el ejemplo anterior no realizó ningún trabajo con archivos. Al recibir el mensaje `read`, los objetos siempre devolverán 65 (que convertido a `char` representa una letra A), y al recibir el mensaje `readLine`, el objeto decorado siempre devolverá la cadena `Hola mundo`.

El siguiente ejemplo, en cambio, usa las clases `FileReader` y `BufferedReader` estándar, importadas desde el paquete `java.io`.

```
package decorator;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JFileChooser;

public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/decorator");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }
        FileReader fr = new FileReader(nomArch);
        System.out.println((char) fr.read());
        System.out.println();
        FileReader fr2 = new FileReader(nomArch);
        BufferedReader br = new BufferedReader(fr2);
        System.out.println((char) br.read());
        System.out.println(br.readLine());
    }
}
```

Dado que se requiere el acceso a un archivo de texto, para obtener la misma salida que el ejemplo anterior, debería crearse un archivo de texto como el siguiente:

hola.txt
AHola mundo



Veamos ahora cómo utilizar las principales clases mostradas en la pág. 149.

Primer caso: La clase File

```
package claseFile;
import java.io.File;
import java.text.DateFormat;
import java.util.Date;
import javax.swing.JFileChooser;

public class Main {
    public static void main(String[] args) {
        String nomElegido = "";
        JFileChooser fc = new JFileChooser("src/clasefile");
        fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomElegido = fc.getSelectedFile().getPath();
        }
        File f = new File(nomElegido);
        System.out.println("Nombre: " + f.getName());
        System.out.println("Ubicado en: " + f.getParent());
        System.out.println("Última modificación: " +
                           DateFormat.getInstance().format(new Date(f.lastModified())));
        if (f.isFile()) {
            System.out.println("ES UN ARCHIVO");
            System.out.println("Tamaño: " + f.length() + " bytes");
        } else if (f.isDirectory()) {
            System.out.println("ES UN DIRECTORIO");
            System.out.println("Contenido:");
            int cantArchivos = 0;
            long tamArchivos = 0;
            int cantDirectorios = 0;
            for (File elemento : f.listFiles()) {
                System.out.print(DateFormat.getInstance().format(new Date(elemento.lastModified())));
                if (elemento.isFile()) {
                    cantArchivos++;
                    tamArchivos += elemento.length();
                    System.out.printf(" %,14d ", elemento.length());
                } else if (elemento.isDirectory()) {
                    cantDirectorios++;
                    System.out.printf(" <DIR> ");
                }
                System.out.println(elemento.getName());
            }
            System.out.printf(" %,7d archivos %,14d bytes\n", cantArchivos, tamArchivos);
            System.out.printf(" %,7d dirs %,14d bytes libres\n",
                            cantDirectorios, f.getFreeSpace());
        }
    }
}
```

Los siguientes métodos permiten obtener información del sistema de archivos:

getName: getParent:
lastModified: length:
isFile: isDirectory:
listFiles: getFreeSpace:

Además, los siguientes métodos permiten modificar el sistema de archivos:

.....
.....



Segundo caso: Archivos de texto plano (Unicode)

```
package archivosDeTexto;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.LineNumberReader;
import java.io.PrintWriter;
import javax.swing.JFileChooser;

public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosDeTexto");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }

        FileReader fr = new FileReader(nomArch);
        BufferedReader br = new BufferedReader(fr);
        LineNumberReader lr = new LineNumberReader(br);

        FileWriter fw = new FileWriter(nomArch + ".bak");
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter pw = new PrintWriter(bw);

        String renglon;
        while ((renglon = lr.readLine()) != null) {
            pw.println(lr.getLineNumber() + ") " + renglon);
        }

        pw.close();
    }
}
```

Para leer datos desde este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia y poder usar) y (para poder usar los métodos).

No quedan datos por leer, si

Para grabar datos en este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia y poder usar) y (para poder usar los métodos).

Para garantizar que los datos se graben en los archivos, éstos deben cerrarse mediante



Tercer caso: Archivos binarios secuenciales

```
package archivosBinariosSecuenciales;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.swing.JFileChooser;

public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosBinariosSecuenciales");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }
        FileInputStream fis = new FileInputStream(nomArch);
        BufferedInputStream bis = new BufferedInputStream(fis);
        DataInputStream dis = new DataInputStream(bis);
        FileOutputStream fos = new FileOutputStream(nomArch + ".bak");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        DataOutputStream dos = new DataOutputStream(bos);
        int byteLeido; // byte no permite contener 128..255. Por eso: int
        boolean finDeArchivo = false;
        while (!finDeArchivo) {
            try {
                byteLeido = dis.readByte();
                dos.writeByte(byteLeido);
            } catch (EOFException ex) {
                dos.close();
                finDeArchivo = true;
            }
        }
    }
}
```

Para leer datos desde este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia) y (para poder usar los métodos).

No quedan datos por leer, si

Para grabar datos en este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia) y (para poder usar los métodos).

Para garantizar que los datos se graben en los archivos, éstos deben cerrarse mediante



Cuarto caso: Archivos binarios de acceso aleatorio

```
package archivosBinariosDeAccesoAleatorio;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import javax.swing.JFileChooser;

public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosBinariosDeAccesoAleatorio");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }

        RandomAccessFile raf = new RandomAccessFile(nomArch, "rw");

        long tam = raf.length();
        for (long i = 0; i < tam / 2; i++) {
            raf.seek(i);
            int b1 = raf.read();
            raf.seek(tam - i - 1);
            int b2 = raf.read();
            raf.seek(i);
            raf.write(b2);
            raf.seek(tam - i - 1);
            raf.write(b1);
        }
        raf.close();
    }
}
```

Este tipo de archivos no requiere que se utilice el patrón de diseño **Decorator**. La clase `RandomAccessFile` posee prácticamente todos los métodos necesarios para trabajar con archivos de acceso aleatorio, por ejemplo:

length:
setLength:
getFilePointer:
seek:
read:
write:
readBoolean / writeBoolean:
readByte / writeByte:
readChar / writeChar:
readDouble / writeDouble:
readFloat / writeFloat:
readInt / writeInt:
readLong / writeLong:
readShort / writeShort:
readUTF / writeUTF:

El segundo argumento del constructor puede ser "r" (modo de sólo lectura) o "rw" (modo de lectura y escritura).



Quinto caso: Archivos de objetos / Serialización

```
package serializacion;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        Lista li = new Lista();
        try {
            li = li.deSerializar("lista.txt");
            if (!EntradaSalida.leerBoolean("Ya hay una lista. ¿Desea reutilizarla?")) {
                li = new Lista();
            }
        } catch (IOException | ClassNotFoundException e) {
            EntradaSalida.mostrarString("Lista nueva!");
        }
        do {
            String dato = EntradaSalida.leerString("Ingrese una cadena");
            li.agregar(dato);
        } while (EntradaSalida.leerBoolean("Desea seguir agregando valores?"));
        li.mostrar();
        try {
            li.serializar("lista.txt");
        } catch (Exception e) {
            EntradaSalida.mostrarString(e.getMessage() + "\nERROR AL GRABAR!");
        }
    }
}
```

```
package serializacion;
import java.io.*;
import java.util.ArrayList;
public class Lista implements Serializable {
    private ArrayList<String> valores;
    public Lista() {
        valores = new ArrayList<>();
    }
    public void agregar(String s) {
        valores.add(s);
    }
    public void mostrar() {
        String cadena = "La lista contiene:\n";
        for (String s : valores) {
            cadena += s + "\n";
        }
        EntradaSalida.mostrarString(cadena);
    }
    public Lista deSerializar(String nomArch) throws IOException, ClassNotFoundException {
        ObjectInputStream o =
            new ObjectInputStream(new BufferedInputStream(new FileInputStream(nomArch)));
        Lista j = (Lista) o.readObject();
        o.close();
        return j;
    }
    public void serializar(String nomArch) throws IOException {
        ObjectOutputStream o =
            new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(nomArch)));
        o.writeObject(this);
        o.close();
    }
}
```



```
package serializacion;

import javax.swing.JOptionPane;

public class EntradaSalida {

    public static String leerString(String texto) {
        String st = JOptionPane.showInputDialog(texto);
        return (st == null ? "" : st);
    }

    public static boolean leerBoolean(String texto) {
        int i=JOptionPane.showConfirmDialog(null, texto, "Consulta", JOptionPane.YES_NO_OPTION);
        return i == JOptionPane.YES_OPTION;
    }

    public static void mostrarString(String s) {
        JOptionPane.showMessageDialog(null, s);
    }
}
```

Para leer datos desde este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia) y (para poder usar el método).

Para grabar datos en este tipo de archivos, puede decorarse un objeto de la clase mediante los decoradores (para mejorar la eficiencia) y (para poder usar el método).

Para garantizar que los datos se graben en los archivos, éstos deben cerrarse mediante

Para que el estado de un objeto pueda ser grabado, la clase del objeto debe implementar la interfaz Ésta no declara métodos ni atributos, sólo indica que los objetos de las clases que la implementen podrán ser serializados (sus estados serán grabados) y deserializados (sus estados serán leídos).

Cuando el estado de un objeto está formado por referencias a otros objetos, durante la serialización estos objetos

Cuando el estado de un objeto está formado por referencias a otros objetos, durante la deserialización estos objetos

Si una clase tiene atributos declarados como `transient`, éstos



Usualmente, las aplicaciones se diseñan en capas lógicas (*layers*) alojadas en una o más capas físicas (*tiers*). Una arquitectura común es la de tres capas lógicas: **capa de presentación** (a través de la cual el usuario ingresa solicitudes y obtiene respuestas), **capa de negocio** (donde residen los programas que se ejecutan, los cuales constituyen la lógica del negocio) y **capa de datos** (donde residen los datos a los cuales esta capa se encarga de acceder). Así, los usuarios interactúan con el *front-end* (el software implementado en la capa de presentación) y éste con el *back-end* (el software implementado en las capas de negocio y de datos).

En el siguiente ejemplo, el *front-end* muestra una lista de carreras (obtenida desde el *back-end*), solicita al usuario que elija una carrera, y muestra el listado de alumnos que cursan esa carrera (obtenido desde el *back-end*). El *front-end* utiliza la entrada/salida estándar (la consola) para interactuar con el usuario, pero esto podría cambiarse fácilmente por otra implementación (por ejemplo, una interfaz gráfica). Por otro lado, en este ejemplo el *back-end* sigue el *patrón de diseño Façade*, ya que la clase *BackEnd* proporciona un punto de acceso (mediante sus métodos) a los objetos que manipulan los datos (DTO: *Data Transfer Objects* y DAO: *Data Access Objects*). Los DAO utilizan archivos de texto plano, pero fácilmente podrían cambiarse por otra implementación (por ejemplo, una base de datos).

```
package trescapas;
import java.io.*;
public class Main {
    public static void main(String[] args) throws FileNotFoundException, IOException {
        BackEnd backEnd = new BackEnd();
        FrontEnd frontEnd = new FrontEnd(backEnd);
        frontEnd.arrancar();
    }
}
```

```
package trescapas;
import java.io.*;
import java.util.*;
public class FrontEnd {
    private BackEnd backEnd;
    public FrontEnd(BackEnd backEnd) {
        this.backEnd = backEnd;
    }
    public void arrancar() throws FileNotFoundException, IOException {
        Collection<CarreraDTO> carreras = backEnd.obtenerCarreras();
        for (CarreraDTO elem : carreras) {
            System.out.println(elem.getCodigo() + " - " + elem.getNombre());
        }
        System.out.print("Elija una carrera:");
        int codCarrera = new Scanner(System.in).nextInt();
        Collection<AlumnoDTO> alumnos = backEnd.obtenerAlumnos(codCarrera);
        for (AlumnoDTO elem : alumnos) {
            System.out.println(elem.getLegajo() + " - " + elem.getApellido() +
                               ", " + elem.getNombres());
        }
    }
}
```



```
package trescapas;
import java.io.*;
import java.util.Collection;
public class BackEnd {
    public Collection<CarreraDTO> obtenerCarreras()
        throws FileNotFoundException, IOException {
        CarrerasDAO cDAO = new CarrerasDAO();
        return cDAO.cargarCarreras();
    }
    public Collection<AlumnoDTO> obtenerAlumnos(int codCarrera)
        throws FileNotFoundException, IOException {
        AlumnosDAO aDAO = new AlumnosDAO();
        return aDAO.cargarAlumnos(codCarrera);
    }
}
```

```
package trescapas;
public class CarreraDTO {
    private int codigo;
    private String nombre;
    public int getCodigo() {
        return codigo;
    }
    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```
package trescapas;
import java.io.*;
import java.util.*;
public class CarrerasDAO {
    public Collection<CarreraDTO> cargarCarreras()
        throws FileNotFoundException, IOException {
        Collection<CarreraDTO> carreras = new ArrayList<>();
        BufferedReader br=new BufferedReader(new FileReader("src/trescapas/carreras.txt"));
        String renglon;
        while ((renglon = br.readLine()) != null) {
            CarreraDTO ca = new CarreraDTO();
            ca.setCodigo(Integer.parseInt(renglon));
            renglon = br.readLine();
            ca.setNombre(renglon);
            renglon = br.readLine();
            carreras.add(ca);
        }
        return carreras;
    }
}
```



```
package trescapas;
public class AlumnoDTO {
    private int legajo;
    private String apellido;
    private String nombres;
    public int getLegajo() {
        return legajo;
    }
    public void setLegajo(int legajo) {
        this.legajo = legajo;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public String getNombres() {
        return nombres;
    }
    public void setNombres(String nombres) {
        this.nombres = nombres;
    }
}
```

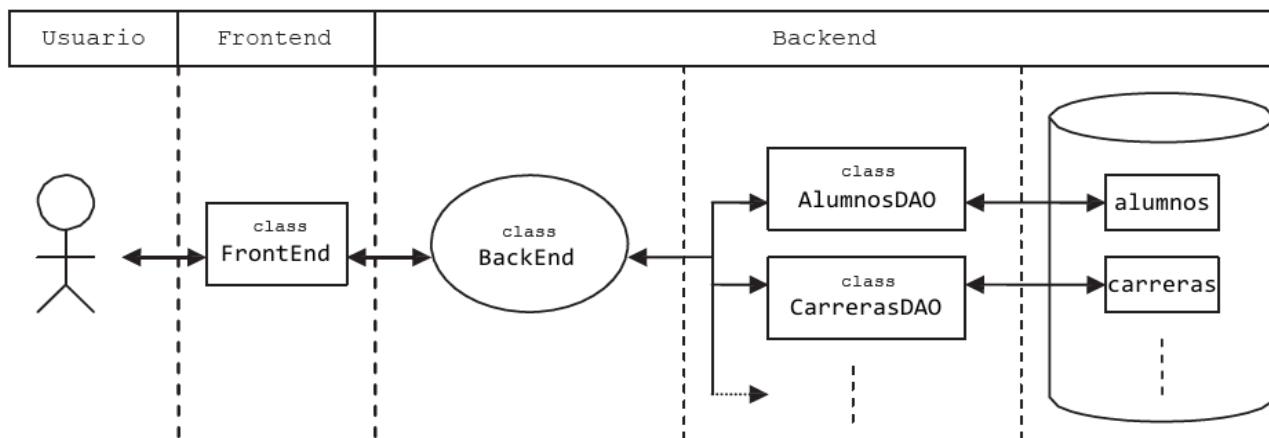
```
package trescapas;
import java.io.*;
import java.util.*;
public class AlumnosDAO {
    public Collection<AlumnoDTO> cargarAlumnos(int codCarrera)
        throws FileNotFoundException, IOException {
        Collection<AlumnoDTO> alumnos = new ArrayList<>();
        BufferedReader br=new BufferedReader(new FileReader("src/trescapas/alumnos.txt"));
        String renglon;
        while ((renglon = br.readLine()) != null) {
            if (Integer.parseInt(renglon) == codCarrera) {
                AlumnoDTO alu = new AlumnoDTO();
                renglon = br.readLine();
                alu.setLegajo(Integer.parseInt(renglon));
                renglon = br.readLine();
                alu.setApellido(renglon);
                renglon = br.readLine();
                alu.setNombres(renglon);
                renglon = br.readLine();
                alumnos.add(alu);
            } else {
                renglon = br.readLine();
                renglon = br.readLine();
                renglon = br.readLine();
                renglon = br.readLine();
            }
        }
        return alumnos;
    }
}
```



Observe que para que el código del ejemplo no quedara demasiado largo y complejo, no fueron tratadas las excepciones, sólo se las declaró. Como CarrerasDAO debe acceder al archivo `src/trescapas/carreras.txt` y AlumnosDAO debe acceder a `src/trescapas/alumnos.txt`, es necesario colocar los archivos en esas ubicaciones, o modificarlas en el código.

carreras.txt	
1	60
2	Informática Aplicada
3	
4	61
5	Control Eléctrico y Accionamientos
6	
7	62
8	Mecánica, Automotores y Máquinas Térmicas
9	
10	63
11	Automatización y Robótica
12	
13	64
14	Electrónica
15	
16	65
17	Química y Química Aplicada
18	
19	66
20	Física y Física Aplicada
21	
22	67
23	Diseño Tecnológico
24	
25	68
26	Profesorado en Disciplinas Industriales
27	
28	69
29	Inglés e Inglés Técnico
30	
31	70
32	Matemática y Matemática Aplicada

alumnos.txt	
1	60
2	11440
3	Rossi
4	Pablo Ricardo
5	
6	65
7	12787
8	Alvarez
9	Tomás Pedro
10	
11	62
12	13797
13	Cáceres
14	María Celeste
15	
16	63
17	13998
18	Vilas
19	Luis Pablo
20	
21	60
22	14247
23	Liao
24	Martín Adrián
25	
26	60
27	14478
28	Lee
29	Analía Rita
30	
31	61
32	14500
33	Quito
34	Esteban





En el ejemplo anterior, dentro del método `arrancar` de la clase `FrontEnd` se utilizó un objeto de la clase `Scanner` :

```
int codCarrera = new Scanner(System.in).nextInt();
```

Un escáner puede leer texto desde cualquier objeto cuya clase implemente la interfaz `Readable` (por ejemplo: `BufferedReader`, `FileReader`, `InputStreamReader`, `LineNumberReader`, etc.), desde instancias de `File` y de las subclases de `InputStream`, e incluso desde objetos de la clase `String`, entre otros. En este caso, `System.in` es el flujo de entrada estándar (el mismo cumple la condición *is-a* con la clase `InputStream`). El texto leído es dividido en partes (*tokens*) mediante un separador determinado (por defecto, un carácter de espacio en blanco, como por ejemplo: el espacio propiamente dicho, el carácter de tabulación o el de salto de línea), y estos *tokens* pueden convertirse en valores de distintos tipos usando los métodos *next* correspondientes (`nextInt`, `nextDouble`, `nextShort`, `nextLong`, etc.). Los métodos `hasNext` (`hasNextInt`, `hasNextDouble`, etc.) sirven para verificar si el flujo contiene más *tokens* del tipo correspondiente (lo cual tiene sentido cuando el escáner está siendo usado para leer texto desde un archivo).

EJERCICIO N° 4

En el ejemplo anterior, la clase `BackEnd` depende de las clases concretas `CarrerasDAO` y `AlumnosDAO`. Este acoplamiento viola el *Principio de Inversión de Dependencias* (pág. 52). Refactorice el código dado, siguiendo el patrón de diseño `Factory`, para que sea posible cambiar fácilmente de dónde provienen los datos: de los dos archivos `src/trescapas/carreras.txt` y `src/trescapas/alumnos.txt` (como hasta ahora), o de un único archivo `src/trescapas/datos.xml`, que tenga la siguiente estructura:

`src/trescapas/datos.xml` (vista parcial)

```
<?xml version="1.0" encoding="UTF-8"?>
<datos>
    <carrera>
        <codigo>60</codigo>
        <denominacion>Informática Aplicada</denominacion>
    </carrera>
    ...
    <alumno>
        <cod_carrera>60</cod_carrera>
        <legajo>11440</legajo>
        <apellido>Rossi</apellido>
        <nombres>Pablo Ricardo</nombres>
    </alumno>
</datos>
```

En Java, existe más de una manera de obtener datos desde archivos XML. Una de las formas más sencillas consiste en cargar el documento en memoria usando un *parser* y luego acceder a su contenido mediante la API (*Application Programming Interface*) denominada DOM (*Document Object Model*). En el siguiente ejemplo, para referirse al documento contenido en el archivo `src/trescapas/datos.xml` se utiliza la variable del tipo (es una interface).



```
package trescapas;

import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XML {

    public static void main(String[] args) {
        Map<String, String> carreras = new HashMap<>();
        try {
            DocumentBuilderFactory factoriaDeParsers = DocumentBuilderFactory.newInstance();
            DocumentBuilder parser = factoriaDeParsers.newDocumentBuilder();
            Document doc = parser.parse(new File("src/trescapas/datos.xml"));
            Element raiz = (Element) doc.getElementsByTagName("datos").item(0);

            NodeList nodosCarrera = raiz.getElementsByTagName("carrera");
            for (int i = 0; i < nodosCarrera.getLength(); i++) {
                Element e = (Element) nodosCarrera.item(i);
                String codigo = e.getElementsByTagName("codigo").item(0).getTextContent();
                String denominacion = e.getElementsByTagName("denominacion").item(0).getTextContent();
                carreras.put(codigo, denominacion);
            }

            NodeList nodosAlumno = raiz.getElementsByTagName("alumno");
            for (int i = 0; i < nodosAlumno.getLength(); i++) {
                Element e = (Element) nodosAlumno.item(i);
                System.out.println("Apellido: " +
                    e.getElementsByTagName("apellido").item(0).getTextContent());
                System.out.println(" Nombres: " +
                    e.getElementsByTagName("nombres").item(0).getTextContent());
                System.out.println(" Legajo: " +
                    e.getElementsByTagName("legajo").item(0).getTextContent());
                System.out.println(" Carrera: " + carreras.get(
                    e.getElementsByTagName("cod_carrera").item(0).getTextContent()));
                System.out.println();
            }
        } catch (ParserConfigurationException | SAXException | IOException ex) {
            System.err.println(ex);
        }
    }
}
```

En DOM, todos los documentos tienen una estructura de árbol. Los nodos del árbol pueden ser, entre otros, elementos de XML (implementan la interfaz o nodos de texto. Cuando un elemento de XML recibe un mensaje con un argumento de la clase, devuelve una lista (que implementa) con los nodos cuyos nombres coinciden con el argumento. Esta lista devuelve su longitud cuando recibe el mensaje, y para obtener un nodo debe enviársele a la lista el mensaje usando el índice del nodo como argumento. Cuando un nodo recibe el mensaje, devuelve un objeto de la clase con el texto que contiene.



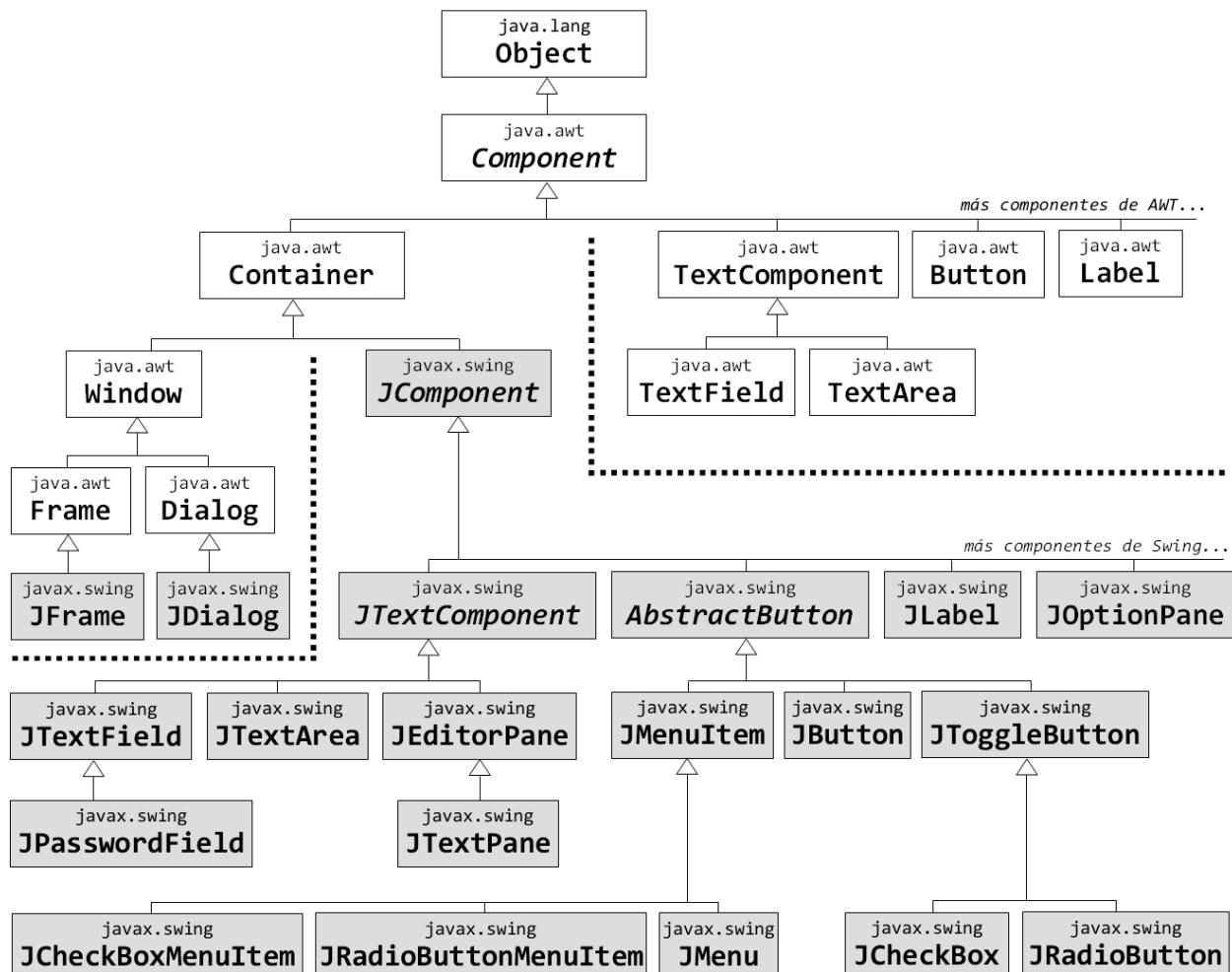
Unidad 9

Interfaces Gráficas de Usuario

Una interfaz gráfica o *GUI (Graphical User Interface)* es un conjunto de componentes gráficos (ventanas, botones, áreas de texto, etiquetas, etc.) que facilitan la interacción entre el usuario y la aplicación. En Java existen varias API que podemos utilizar para desarrollar interfaces gráficas (*AWT*, *Swing*, *SWT*, etc.). En este capítulo utilizaremos *Swing*, una de las más populares.

Los nombres de la mayoría de las clases en *Swing* se diferencian de sus equivalentes en *AWT* porque empiezan con la letra J, por ejemplo: **JButton** es una clase de *Swing* y **Button** es una clase de *AWT*.

La siguiente vista parcial de la jerarquía de clases de *Swing* permite notar que las clases **JFrame** y **JDialog** son (indirectamente) subclases de **Window** y de **Container**, por lo que se las considera *contenedores* de nivel superior. Casi todas las demás clases en esta API son *componentes de Swing* porque heredan de **JComponent**. Sin embargo, éstos también heredan de **Container**, por lo cual pueden servir de *contenedores* de otros *componentes* (aunque no todas las posibles combinaciones tienen sentido).

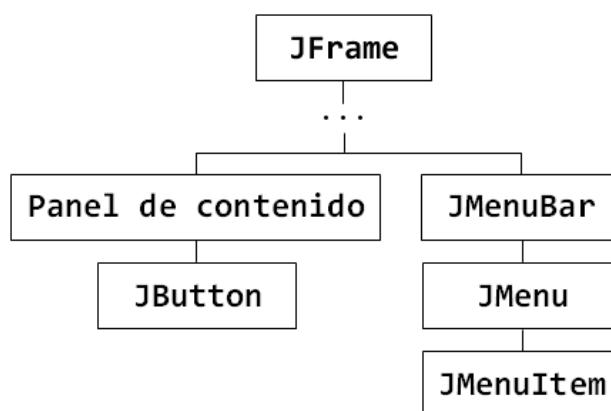




Las subclases directas de `JComponent` que se omitieron en la jerarquía anterior se listan a continuación:

- `BasicInternalFrameTitlePane`
- `Box`
- `Box.Filler`
- `JColorChooser`
- `JComboBox`
- `JFileChooser`
- `JInternalFrame`
- `JLayer`
- `JLayeredPane`
- `JList`
- `JMenuBar`
- `JPanel`
- `JPopupMenu`
- `JProgressBar`
- `JRootPane`
- `JScrollBar`
- `JScrollPane`
- `JSeparator`
- `JSlider`
- `JSpinner`
- `JSplitPane`
- `JTabbedPane`
- `JTable`
- `JTableHeader`
- `JToolBar`
- `JToolTip`
- `JTree`
- `JViewport`

En el próximo ejemplo se crea una GUI denominada `Vista`. Su contenedor de nivel superior es un `JFrame` al cual primeramente se le agrega una `JMenuBar` (mediante el método del `JFrame`). Esta `JMenuBar` contiene un único `JMenu` (agregado mediante el método de la `JMenuBar`), el cual contiene un único `JMenuItem` (agregado mediante el método del `JMenu`). Además, en el panel de contenido del `JFrame` se coloca luego un `JButton` (mediante el método del panel obtenido al invocar el método `getContentPane` del `JFrame`). La siguiente figura muestra esta jerarquía de elementos contenedores y componentes contenidos.



```
package demo1;  
public class Main {  
    public static void main(String[] args) {  
        Vista v = new Vista();  
    }  
}
```



```
package demo1;

import java.awt.Color;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

public class Vista {

    private JButton b;
    private JFrame f;
    private JMenu m;
    private JMenuBar mB;
    private JMenuItem mI;

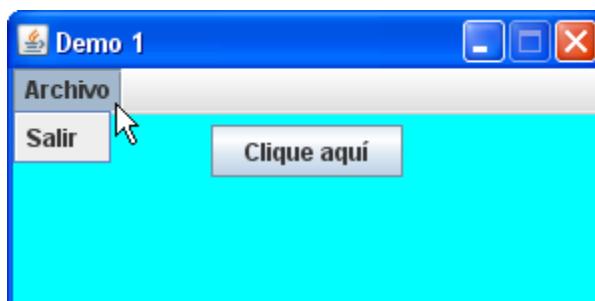
    public Vista() {
        b = new JButton();
        mI = new JMenuItem();
        m = new JMenu();
        mB = new JMenuBar();
        f = new JFrame();

        b.setText("Clique aquí"); // Establece el texto del botón
        mI.setText("Salir"); // Establece el texto del ítem del menú
        m.setText("Archivo"); // Establece el texto del menú
        m.add(mI); // Agrega, en el menú, el ítem del menú
        mB.add(m); // Agrega, en la barra de menús, el menú

        f.getContentPane().setBackground(Color.CYAN); // Establece el fondo del panel
        f.getContentPane().setLayout(new FlowLayout()); // Establece el layout del panel
        f.getContentPane().add(b); // Agrega, en el panel, el botón

        f.setJMenuBar(mB); // Agrega, en el JFrame, la barra de menús
        f.setTitle("Demo 1"); // Establece el título del JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Al cerrar el JFrame, salir
        f.setSize(300, 150); // Establece el tamaño del JFrame
        f.setLocationRelativeTo(null); // Centra el JFrame en la pantalla
        f.setResizable(false); // Impide que se cambie el tamaño del JFrame
        f.setVisible(true); // Muestra el JFrame
    }
}
```

El resultado que se obtiene al ejecutar el programa anterior es el siguiente:





OBSERVACIÓN

Por una cuestión de conveniencia, la clase `JFrame` sobrescribe los tres métodos `add`, `remove` y `setLayout` para que automáticamente obtengan una referencia al panel de contenido e invoquen a sus métodos homónimos. Por ello, las líneas de código:

```
f.getContentPane().setBackground(Color.CYAN); // Establece el fondo del panel  
f.getContentPane().setLayout(new FlowLayout()); // Establece el layout del panel  
f.getContentPane().add(b); // Agrega, en el panel, el botón
```

se podrían simplificar de la siguiente manera:

```
f.getContentPane().setBackground(Color.CYAN); // Establece el fondo del panel  
f.setLayout(new FlowLayout()); // Establece el layout del panel  
f.add(b); // Agrega, en el panel, el botón
```

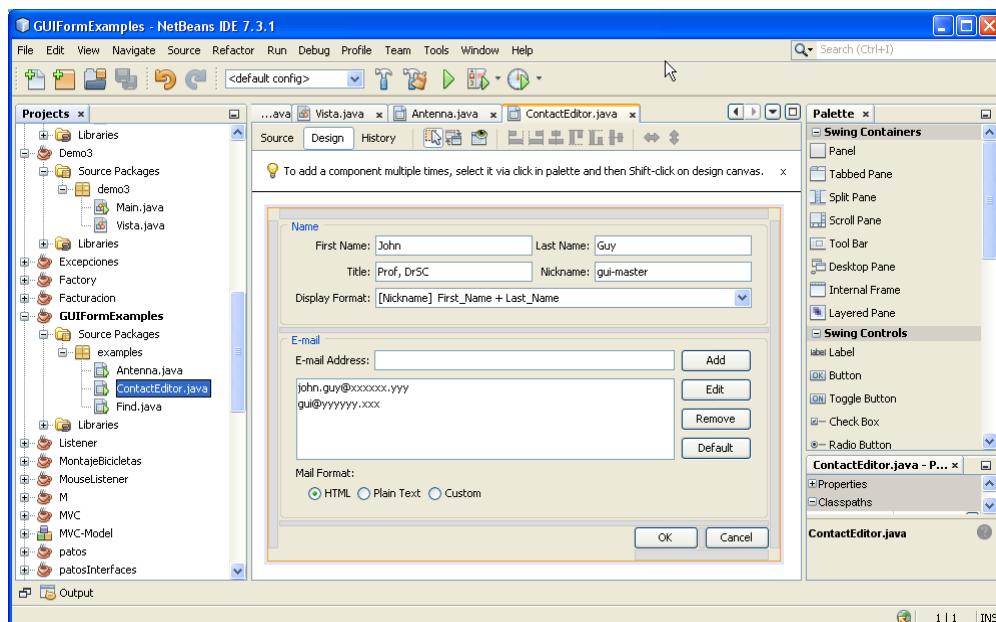
Nótese que `setBackground` no es uno de los métodos sobrescritos, por lo que es necesario obtener el panel de contenido de forma explícita mediante `getContentPane`.

Para que los componentes de la GUI se muestren de manera correcta, independientemente de la plataforma donde corra la aplicación, no se suele especificar la posición absoluta de los mismos dentro del `JFrame`, sino que se deja su distribución a cargo de un *layout manager* establecido mediante el método

Los principales *layout managers* disponibles son los siguientes:

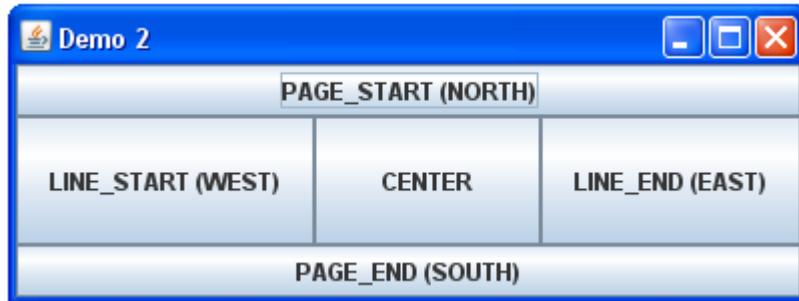
- **java.awt.BorderLayout**
- **java.awt.CardLayout**
- **java.awt.FlowLayout**
- **java.awt.GridBagLayout**
- **java.awt.GridLayout**
- **javax.swing.BoxLayout**
- **javax.swing.GroupLayout**
- **javax.swing.OverlayLayout**
- **javax.swing.ScrollPaneLayout**
- **javax.swing.SpringLayout**
- **javax.swing.ViewportLayout**

Aquí solamente trataremos los *layout managers* que están marcados en negrita. Los demás casi no son usados cuando se programa una GUI a mano, pero algunas herramientas de diseño (por ejemplo, el *GUI Builder* de NetBeans) sí los utilizan:





BorderLayout divide un panel de contenido en hasta cinco regiones estándar y permite agregar como máximo un elemento en cada una. Al agregar un elemento, éste se estira hasta abarcar completamente su región, como se ve en la próxima figura. Si no se indica una región, el elemento se agrega en la región central. BorderLayout es el *layout manager* utilizado por defecto en el panel de contenido de JFrame.

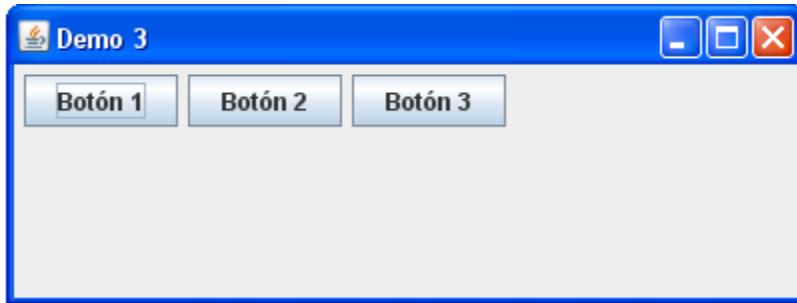


```
package demo2;  
  
public class Main {  
    public static void main(String[] args) {  
        Vista v = new Vista();  
    }  
}
```

```
package demo2;  
  
import java.awt.BorderLayout;  
import javax.swing.JButton;  
import javax.swing.JFrame;  
  
public class Vista {  
    private JButton b1, b2, b3, b4, b5;  
    private JFrame f;  
  
    public Vista() {  
        b1 = new JButton("PAGE_START (NORTH)");  
        b2 = new JButton("LINE_START (WEST)");  
        b3 = new JButton("PAGE_END (SOUTH)");  
        b4 = new JButton("LINE_END (EAST)");  
        b5 = new JButton("CENTER");  
        f = new JFrame();  
  
        f.getContentPane().setLayout(new BorderLayout());  
        f.getContentPane().add(b1, BorderLayout.PAGE_START);  
        f.getContentPane().add(b2, BorderLayout.LINE_START);  
        f.getContentPane().add(b3, BorderLayout.PAGE_END);  
        f.getContentPane().add(b4, BorderLayout.LINE_END);  
        f.getContentPane().add(b5, BorderLayout.CENTER);  
  
        f.setTitle("Demo 2");  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setSize(400, 150);  
        f.setLocationRelativeTo(null);  
        f.setVisible(true);  
    }  
}
```



FlowLayout no divide el panel en regiones ni estira los elementos. Éstos simplemente se van agregando uno al lado del otro mientras quepan, continuando abajo cuando se acaba el espacio, y manteniendo la alineación indicada al instanciar el *layout manager*.FlowLayout es el *layout manager* utilizado por defecto en JPanel. Si no se especifica cómo se alinearán los elementos, éstos se centran. En el siguiente ejemplo, se especificó que los elementos se alinearían a la izquierda.



```
package demo3;

public class Main {

    public static void main(String[] args) {
        Vista v = new Vista();
    }
}
```

```
package demo3;

import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Vista {

    private JButton b1, b2, b3;
    private JFrame f;

    public Vista() {

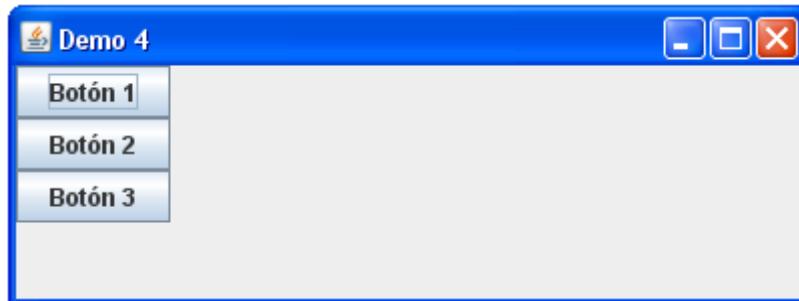
        b1 = new JButton("Botón 1");
        b2 = new JButton("Botón 2");
        b3 = new JButton("Botón 3");
        f = new JFrame();

        f.getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT));
        f.getContentPane().add(b1);
        f.getContentPane().add(b2);
        f.getContentPane().add(b3);

        f.setTitle("Demo 3");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 150);
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```



BoxLayout tampoco divide el panel en regiones ni estira los elementos, pero a diferencia de FlowLayout, permite especificar si los elementos se irán agregando al lado de los anteriores o debajo de ellos. En el próximo ejemplo, los botones se van agregando a lo largo del eje vertical (Y_AXIS), como se ve en la siguiente figura:



```
package demo4;

public class Main {

    public static void main(String[] args) {
        Vista v = new Vista();
    }
}
```

```
package demo4;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Vista {

    private JButton b1, b2, b3;
    private JFrame f;

    public Vista() {

        b1 = new JButton("Botón 1");
        b2 = new JButton("Botón 2");
        b3 = new JButton("Botón 3");
        f = new JFrame();

        f.getContentPane().setLayout(new BoxLayout(f.getContentPane(), BoxLayout.Y_AXIS));
        f.getContentPane().add(b1);
        f.getContentPane().add(b2);
        f.getContentPane().add(b3);

        f.setTitle("Demo 4");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 150);
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```



GridLayout divide un panel de contenido en una grilla de celdas de igual tamaño, según la cantidad de filas y columnas que se indiquen, y los elementos se van agregando consecutivamente, de a uno por celda. Al agregar un elemento, éste se estira hasta abarcar completamente su celda, como se ve en la siguiente figura:

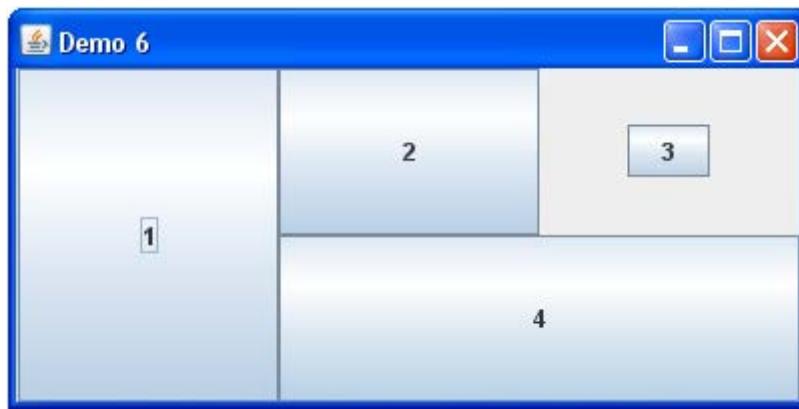


```
package demo5;  
  
public class Main {  
    public static void main(String[] args) {  
        Vista v = new Vista();  
    }  
}
```

```
package demo5;  
  
import java.awt.GridLayout;  
import javax.swing.JButton;  
import javax.swing.JFrame;  
  
public class Vista {  
    private JButton b[];  
    private JFrame f;  
  
    public Vista() {  
        b = new JButton[12];  
        f = new JFrame("Demo 5");  
  
        for (int i = 0; i < 12; i++) {  
            b[i] = new JButton(Character.toString("123456789*0#".charAt(i)));  
        }  
  
        f.getContentPane().setLayout(new GridLayout(4, 3));  
  
        for (int i = 0; i < 12; i++) {  
            f.getContentPane().add(b[i]);  
        }  
  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setSize(200, 220);  
        f.setLocationRelativeTo(null);  
        f.setVisible(true);  
    }  
}
```



El último *layout manager* que veremos aquí es `GridBagLayout`, uno de los más flexibles y complejos. Éste divide el panel de contenido en una grilla, y cada elemento que se vaya agregando debe ir acompañado de una instancia de `GridBagConstraints`, en cuyos atributos se indica dónde y cómo mostrar el elemento, el cual, por ejemplo, puede ocupar más de una celda, estirarse o no, alinearse hacia cualquiera de los bordes de la celda, etc. La siguiente figura muestra algunas de las posibilidades que permite este potente *layout manager*:



En este ejemplo, se utiliza una grilla de dos filas y tres columnas, cuyas celdas tienen las siguientes coordenadas:

<code>gridx = 0; gridy = 0</code>	<code>gridx = 1; gridy = 0</code>	<code>gridx = 2; gridy = 0</code>
<code>gridx = 0; gridy = 1</code>	<code>gridx = 1; gridy = 1</code>	<code>gridx = 2; gridy = 1</code>

Se agregan cuatro botones al panel de contenido del `JFrame`, cada uno acompañado de su correspondiente instancia de `GridBagConstraints`. El primer botón agregado ocupa dos filas, y el último ocupa dos columnas:

<code>b[0]</code> <code>gridheight = 2</code>	<code>b[1]</code>	<code>b[2]</code>
	<code>b[3]</code> <code>gridwidth = 2</code>	

El tercer botón agregado (`b[2]`) es el único que no debe estirarse para llenar su celda en ambas direcciones (porque no usa `fill = GridBagConstraints.BOTH`). Debido a ello, queda de un tamaño menor que la celda que lo contiene, y entonces se lo centra usando `anchor = GridBagConstraints.CENTER`.

```
package demo6;

public class Main {

    public static void main(String[] args) {
        Vista v = new Vista();
    }
}
```



```
package demo6;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Vista {

    private JButton b[];
    private GridBagConstraints c[];
    private JFrame f;

    public Vista() {

        b = new JButton[4];
        c = new GridBagConstraints[4];
        f = new JFrame("Demo 6");

        for (int i = 0; i < 4; i++) {
            b[i] = new JButton(Integer.toString(i + 1));
            c[i] = new GridBagConstraints();
        }

        c[0].gridx = 0;
        c[0].gridy = 0;
        c[0].gridheight = 2;
        c[0].fill = GridBagConstraints.BOTH;
        c[0].weightx = 1.0;
        c[0].weighty = 1.0;

        c[1].gridx = 1;
        c[1].gridy = 0;
        c[1].fill = GridBagConstraints.BOTH;
        c[1].weightx = 1.0;
        c[1].weighty = 1.0;

        c[2].gridx = 2;
        c[2].gridy = 0;
        c[2].anchor = GridBagConstraints.CENTER;
        c[2].weightx = 1.0;
        c[2].weighty = 1.0;

        c[3].gridx = 1;
        c[3].gridy = 1;
        c[3].gridwidth = 2;
        c[3].fill = GridBagConstraints.BOTH;
        c[3].weightx = 1.0;
        c[3].weighty = 1.0;

        f.getContentPane().setLayout(new GridBagLayout());

        for (int i = 0; i < 4; i++) {
            f.getContentPane().add(b[i], c[i]);
        }

        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 200);
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```



Hasta aquí hemos visto cómo los *layout managers* permiten determinar la distribución de los componentes en un contenedor¹. Ahora veremos un ejemplo donde se utilizan los contenedores y componentes más comunes.

```
package demo7;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Vista v = new Vista();  
    }  
}
```

```
package demo7;  
  
import java.awt.*;  
import javax.swing.*;  
import javax.swing.plaf.basic.BasicArrowButton;  
import javax.swing.tree.DefaultMutableTreeNode;  
  
public class Vista {  
  
    private JFrame frame;  
    private JTabbedPane tabbedPane;  
    private JPanel panel;  
    private JScrollPane scrollPane;  
    private JCheckBox cB1;  
    private JCheckBox cB2;  
    private JCheckBox cB3;  
    private ButtonGroup bG;  
    private JRadioButton rB1;  
    private JRadioButton rB2;  
    private JRadioButton rB3;  
    private JRadioButton rB4;  
    private JPanel panelCheckboxes;  
    private JPanel panelRadioButtons;  
    private JPanel panelPanelRadioButtons;  
    private JPanel panelButton;  
    private JButton jb;  
    private JPasswordField passwordField;  
    private JTextArea textArea;  
    private JTextField textField;  
    private JPanel panelConBorde;  
    private JLabel nombre;  
    private JLabel antecedentes;  
    private JLabel contraseña;  
    private JPanel panelTextField;  
    private JPanel panelScrollPane;  
    private JPanel panelPasswordField;  
    private JPanel panelPanelScrollPane;  
    private JSplitPane splitPane;  
    private JPanel panelSuperior;  
    private JPanel panelInferior;  
    private BasicArrowButton basicArrowButton;  
    private JLabel ingresoPorSeleccion;
```

¹ En todos los ejemplos vistos, se utilizó el *layout manager* del panel de contenido del `JFrame`. Sin embargo, algunos componentes (por ejemplo, `JPanel`) son considerados contenedores y, por lo tanto, utilizan su propio *layout manager* independiente.



```
private JLabel visualizacionDeModelos;
private JComboBox comboBox;
private JList list;
private JSlider slider;
private JSpinner spinner;
private JPanel panelIngresoPorSeleccion;
private JPanel panelVisualizacionDeModelos;
private JPanel panelComboBox;
private JPanel panelList;
private JPanel panelSlider;
private JPanel panelSpinner;
private JTable table;
private JTree tree;
private JPanel panelTable;
private JPanel panelTree;
private JScrollPane scrollPaneList;
private JScrollPane scrollPaneTable;
private JScrollPane scrollPaneTree;
private JProgressBar progressBar;
private JMenuBar menuBar;
private JMenu verMenu;
private JSeparator separator;
private JMenu ayudaMenu;
private JCheckBoxMenuItem jCheckBoxMenuItem;
private JRadioButtonMenuItem jRadioButtonMenuItem1;
private JRadioButtonMenuItem jRadioButtonMenuItem2;
private JRadioButtonMenuItem jRadioButtonMenuItem3;
private ButtonGroup bG2;
private JMenu resolucionMenu;
private JMenuItem soloTextoMenuItem1;
private JMenuItem soloTextoMenuItem2;

public Vista() {
    frame = new JFrame("Demo 7");
    tabbedPane = new JTabbedPane();
    panel = new JPanel();
    panel.setLayout(new BorderLayout());
    cB1 = new JCheckBox("Java");
    cB2 = new JCheckBox("C#");
    cB3 = new JCheckBox("Python");
    cB1.setMaximumSize(new Dimension(80, 25));
    cB2.setMaximumSize(new Dimension(80, 25));
    cB3.setMaximumSize(new Dimension(80, 25));
    cB1.setBackground(Color.CYAN);
    cB2.setBackground(Color.CYAN);
    cB3.setBackground(Color.CYAN);
    cB1.setAlignmentX(Component.CENTER_ALIGNMENT);
    cB2.setAlignmentX(Component.CENTER_ALIGNMENT);
    cB3.setAlignmentX(Component.CENTER_ALIGNMENT);

    panelCheckboxes = new JPanel();
    panelCheckboxes.setLayout(new BoxLayout(panelCheckboxes, BoxLayout.Y_AXIS));
    panelCheckboxes.setBackground(Color.CYAN);
    panelCheckboxes.add(cB1);
    panelCheckboxes.add(cB2);
    panelCheckboxes.add(cB3);
```



```
rB1 = new JRadioButton("Malo");
rB2 = new JRadioButton("Regular");
rB3 = new JRadioButton("Bueno");
rB4 = new JRadioButton("Excelente");
rB1.setBackground(Color.YELLOW);
rB2.setBackground(Color.YELLOW);
rB3.setBackground(Color.YELLOW);
rB4.setBackground(Color.YELLOW);

bG = new ButtonGroup();
bG.add(rB1);
bG.add(rB2);
bG.add(rB3);
bG.add(rB4);

panelPanelRadioButtons = new JPanel();
panelPanelRadioButtons.setLayout(new GridBagLayout());
panelPanelRadioButtons.setBackground(Color.YELLOW);
panelRadioButtons = new JPanel();
panelRadioButtons.setLayout(new FlowLayout());
panelRadioButtons.setBackground(Color.YELLOW);
panelRadioButtons.add(rB1);
panelRadioButtons.add(rB2);
panelRadioButtons.add(rB3);
panelRadioButtons.add(rB4);
panelPanelRadioButtons.add(panelRadioButtons, new GridBagConstraints());

jB = new JButton("Enviar");
panelButton = new JPanel();
panelButton.setBackground(Color.GREEN);
panelButton.add(jB);

panel.add(panelCheckBoxes, BorderLayout.PAGE_START);
panel.add(panelPanelRadioButtons, BorderLayout.CENTER);
panel.add(panelButton, BorderLayout.PAGE_END);

panelConBorde = new JPanel();
panelConBorde.setBorder(
    BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder("Complete los campos"),
        BorderFactory.createEmptyBorder(5, 5, 5, 5)));
panelConBorde.setLayout(new BorderLayout());
panelConBorde.setBackground(new Color(192, 255, 192));

nombre = new JLabel("Nombre: ");
nombre.setPreferredSize(new Dimension(100, 25));
textField = new JTextField();
textField.setColumns(32);

antecedentes = new JLabel("Antecedentes: ");
antecedentes.setPreferredSize(new Dimension(100, 25));
textArea = new JTextArea("");
textArea.setColumns(30);
textArea.setRows(12);
scrollPane = new JScrollPane(textArea);

contraseña = new JLabel("Contraseña: ");
contraseña.setPreferredSize(new Dimension(100, 25));
passwordField = new JPasswordField();
passwordField.setColumns(8);
```



```
FlowLayout fL = new FlowLayout();
fL.setAlignment(FlowLayout.LEFT);
panelTextField = new JPanel();
panelScrollPane = new JPanel();
panelPasswordField = new JPanel();
panelTextField.setLayout(fL);
panelScrollPane.setLayout(fL);
panelPasswordField.setLayout(fL);
panelTextField.setBackground(new Color(192, 255, 192));
panelScrollPane.setBackground(new Color(192, 255, 192));
panelPasswordField.setBackground(new Color(192, 255, 192));

panelTextField.add(nombre);
panelTextField.add(textField);
panelScrollPane.add(antecedentes);
panelScrollPane.add(scrollPane);
panelPasswordField.add(contraseña);
panelPasswordField.add(passwordField);
basicArrowButton = new BasicArrowButton(BasicArrowButton.EAST);
basicArrowButton.setToolTipText("Login");
panelPasswordField.add(basicArrowButton);

panelPanelScrollPane = new JPanel();
panelPanelScrollPane.setLayout(new GridBagLayout());
GridBagConstraints gBC = new GridBagConstraints();
gBC.anchor = GridBagConstraints.WEST;
gBC.weightx = 1;
panelPanelScrollPane.add(panelScrollPane, gBC);
panelPanelScrollPane.setBackground(new Color(192, 255, 192));

panelConBorde.add(panelTextField, BorderLayout.PAGE_START);
panelConBorde.add(panelPanelScrollPane, BorderLayout.CENTER);
panelConBorde.add(panelPasswordField, BorderLayout.PAGE_END);

panelSuperior = new JPanel();
panelSuperior.setLayout(new BorderLayout());
panelSuperior.setBackground(new Color(0, 64, 128));
panelInferior = new JPanel();
panelInferior.setLayout(new BorderLayout());
panelInferior.setBackground(new Color(0, 128, 64));
ingresoPorSeleccion = new JLabel("Ingreso por selección");
ingresoPorSeleccion.setForeground(Color.WHITE);
ingresoPorSeleccion.setHorizontalTextPosition(SwingConstants.CENTER);
panelSuperior.add(ingresoPorSeleccion, BorderLayout.PAGE_START);
panelIngresoPorSeleccion = new JPanel();
panelIngresoPorSeleccion.setLayout(new GridLayout(2, 2));

String[] dias = {"Sábado", "Viernes", "Jueves", "Miércoles", "Martes",
                 "Lunes", "Domingo"};

comboBox = new JComboBox(dias);
comboBox.setToolTipText("JComboBox");
panelComboBox = new JPanel();
panelComboBox.setLayout(new GridBagLayout());
panelComboBox.add(comboBox, new GridBagConstraints());
panelComboBox.setBackground(new Color(192, 255, 192));
panelIngresoPorSeleccion.add(panelComboBox);

list = new JList(dias);
list.setToolTipText("JList");
scrollPanelList = new JScrollPane(list);
scrollPanelList.setPreferredSize(new Dimension(80, 60));
```



```
panelList = new JPanel();
panelList.setLayout(new GridBagLayout());
panelList.add(scrollPanelList, new GridBagConstraints());
panelList.setBackground(new Color(255, 192, 192));
panelIngresoPorSeleccion.add(panelList);

slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 50);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setToolTipText("JSlider");
panelSlider = new JPanel();
panelSlider.setLayout(new GridBagLayout());
panelSlider.add(slider, new GridBagConstraints());
panelSlider.setBackground(new Color(192, 192, 255));
panelIngresoPorSeleccion.add(panelSlider);

SpinnerListModel modeloDias = new SpinnerListModel(dias);
spinner = new JSpinner(modeloDias);
spinner.setPreferredSize(new Dimension(80, 25));
spinner.setToolTipText("JSpinner");
panelSpinner = new JPanel();
panelSpinner.setLayout(new GridBagLayout());
panelSpinner.add(spinner, new GridBagConstraints());
panelSpinner.setBackground(new Color(192, 192, 192));
panelIngresoPorSeleccion.add(panelSpinner);

panelSuperior.add(panelIngresoPorSeleccion, BorderLayout.CENTER);

visualizacionDeModelos = new JLabel("Visualización de modelos");
visualizacionDeModelos.setForeground(Color.WHITE);
visualizacionDeModelos.setHorizontalTextPosition(SwingConstants.CENTER);
panelInferior.add(visualizacionDeModelos, BorderLayout.PAGE_START);

panelVisualizacionDeModelos = new JPanel();
panelVisualizacionDeModelos.setLayout(new GridLayout(1, 2));

String[] columnas = {"Nombre", "Apellido", "DNI"};
Object[][] datos = {{ {"Santiago", "Lima", 20146853},
                     {"Antonia", "Arroyo", 31046772}, {"Sandra", "Luppi", 23065768},
                     {"Susana", "Torio", 28011727}, {"Esteban", "Quito", 17271884} }};
table = new JTable(datos, columnas);
table.setToolTipText("JTable");
table.getTableHeader().setBackground(Color.BLUE);
table.getTableHeader().setForeground(Color.WHITE);
scrollPaneTable = new JScrollPane(table);
scrollPaneTable.setPreferredSize(new Dimension(230, 90));
panelTable = new JPanel();
panelTable.setLayout(new GridBagLayout());
panelTable.add(scrollPaneTable, new GridBagConstraints());
panelTable.setBackground(new Color(128, 255, 192));
panelVisualizacionDeModelos.add(panelTable);

DefaultMutableTreeNode top = new DefaultMutableTreeNode("Informática");
DefaultMutableTreeNode año = new DefaultMutableTreeNode("Primer Año");

año.add(new DefaultMutableTreeNode("Programación I"));
año.add(new DefaultMutableTreeNode("Inglés Técnico I"));
año.add(new DefaultMutableTreeNode("Análisis Matemático I"));
año.add(new DefaultMutableTreeNode("Laboratorio"));
top.add(año);
```



```
año = new DefaultMutableTreeNode("Segundo Año");
año.add(new DefaultMutableTreeNode("Programación II"));
año.add(new DefaultMutableTreeNode("Sistemas de Computación I"));
año.add(new DefaultMutableTreeNode("Estructuras y Bases de Datos"));
top.add(año);

año = new DefaultMutableTreeNode("Tercer Año");
año.add(new DefaultMutableTreeNode("Programación III"));
año.add(new DefaultMutableTreeNode("Sistemas de Computación II"));
año.add(new DefaultMutableTreeNode("Seminario"));
top.add(año);

tree = new JTree(top);
tree.setToolTipText("JTree");

scrollPaneTree = new JScrollPane(tree);
scrollPaneTree.setPreferredSize(new Dimension(230, 90));
panelTree = new JPanel();
panelTree.setLayout(new GridBagLayout());
panelTree.add(scrollPaneTree, new GridBagConstraints());
panelTree.setBackground(new Color(192, 128, 255));
panelVisualizacionDeModelos.add(panelTree);

panelInferior.add(panelVisualizacionDeModelos, BorderLayout.CENTER);

progressBar = new JProgressBar();
progressBar.setIndeterminate(true);
panelInferior.add(progressBar, BorderLayout.PAGE_END);

splitPane =
    new JSplitPane(JSplitPane.VERTICAL_SPLIT, panelSuperior, panelInferior);
splitPane.setDividerLocation(150);

tabbedPane.addTab("Botones", null, panel, "Componentes clicables");
tabbedPane.addTab("Casillas de texto", null, panelConBorde,
    "Componentes para ingresar textos");
tabbedPane.addTab("Otros", null, splitPane, "Otros componentes");

menuBar = new JMenuBar();
menuBar.setLayout(fL);
menuBar.setBackground(new Color(192, 192, 192));
verMenu = new JMenu("Ver");
ayudaMenu = new JMenu("Ayuda");
separador = new JSeparator(SwingConstants.VERTICAL);
separador.setPreferredSize(new Dimension(1, 12));

jCheckBoxMenuItem = new JCheckBoxMenuItem("Con compresión");
resolucionMenu = new JMenu("Resolución");
jRadioButtonMenuItem1 = new JRadioButtonMenuItem("640x480");
jRadioButtonMenuItem2 = new JRadioButtonMenuItem("800x600");
jRadioButtonMenuItem3 = new JRadioButtonMenuItem("1024x768");
bG2 = new ButtonGroup();
bG2.add(jRadioButtonMenuItem1);
bG2.add(jRadioButtonMenuItem2);
bG2.add(jRadioButtonMenuItem3);

soloTextoMenuItem1 = new JMenuItem("Asistencia en línea");
soloTextoMenuItem2 = new JMenuItem("Acerca de...");

verMenu.add(jCheckBoxMenuItem);
verMenu.addSeparator();
resolucionMenu.add(jRadioButtonMenuItem1);
resolucionMenu.add(jRadioButtonMenuItem2);
resolucionMenu.add(jRadioButtonMenuItem3);
verMenu.add(resolucionMenu);
```



```
ayudaMenu.add(soloTextoMenuItem1);
ayudaMenu.add(soloTextoMenuItem2);

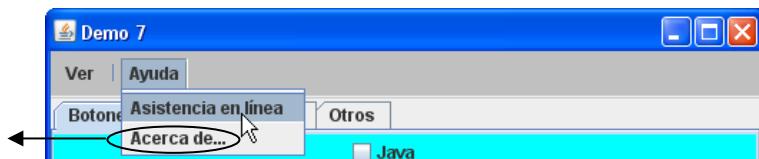
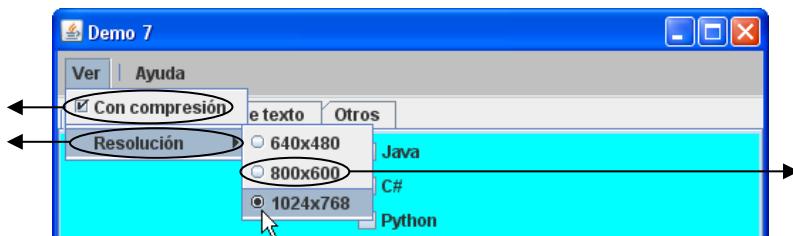
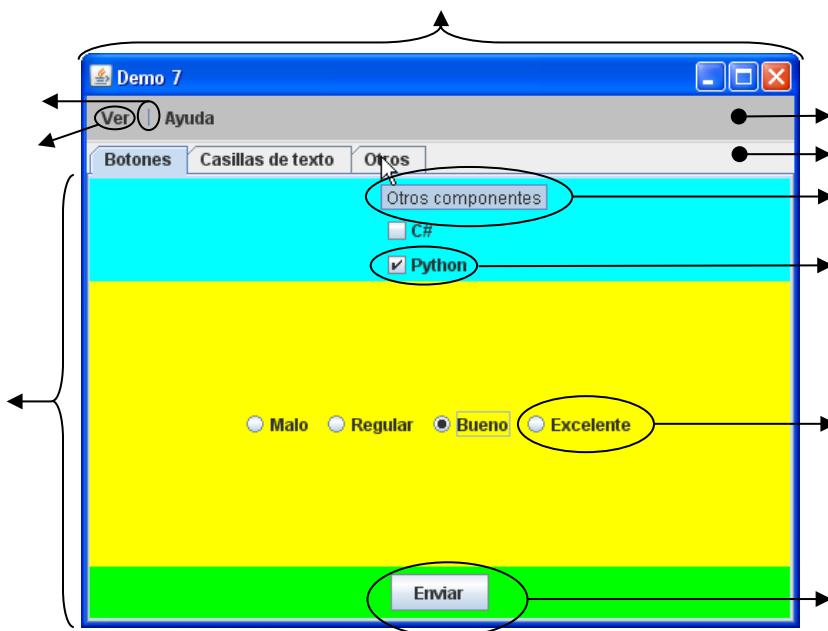
menuBar.add(verMenu);
menuBar.add(separator);
menuBar.add(ayudaMenu);

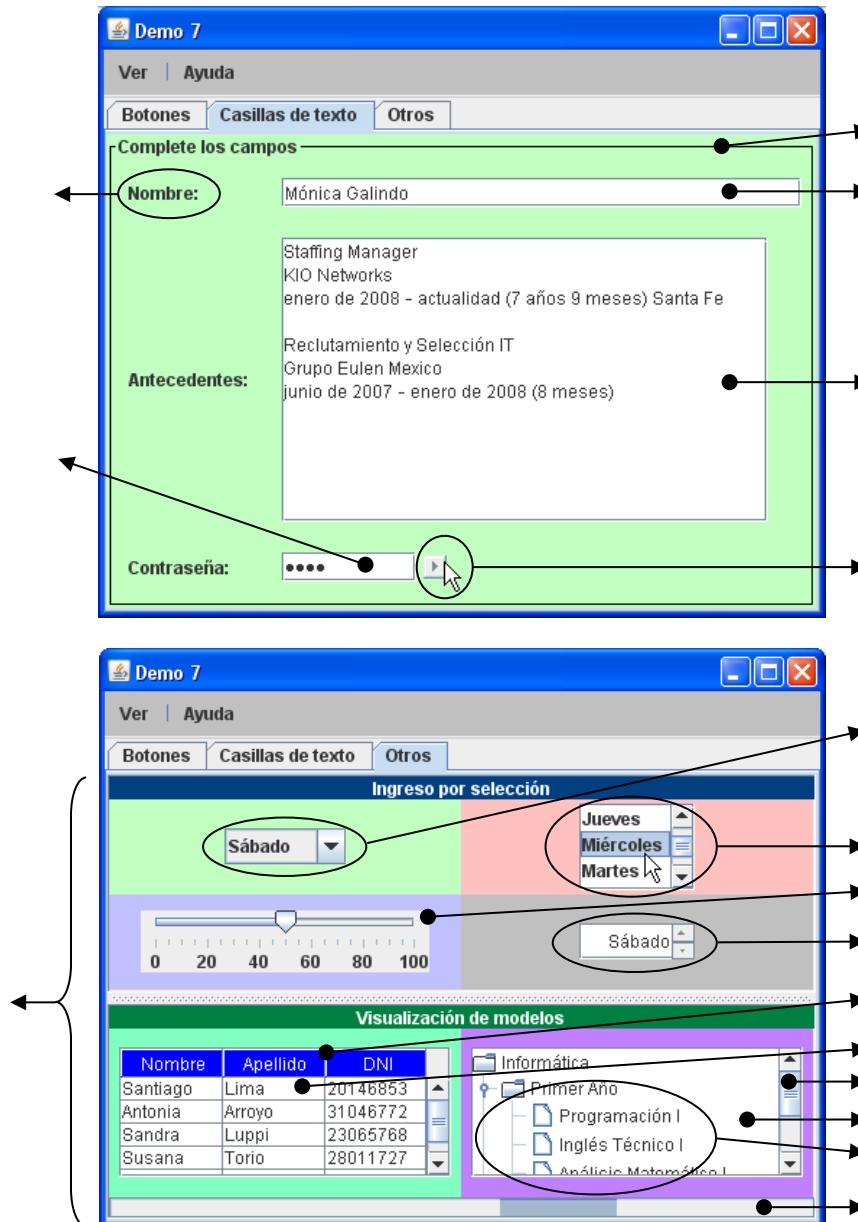
frame.setJMenuBar(menuBar);
frame.getContentPane().add(tabbedPane);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(500, 500);
frame.setLocationRelativeTo(null);
frame.pack();
// frame.setResizable(false);
frame.setVisible(true);
}
}
```

EJERCICIO N° 5

Identifique, en las siguientes figuras, las clases a que pertenecen los contenedores y componentes señalados. A continuación, complete las explicaciones.





En el ejemplo anterior, los componentes que son instancias de las clases `JToolTip`, `JScrollBar` y `JTableHeader` no fueron instanciados mediante `new` y el nombre de esas clases. El texto de las ayudas contextuales (objetos `JToolTip`) se puede establecer mediante el método heredado de `JComponent`. Las barras de desplazamiento (objetos `JScrollBar`) pueden aparecer automáticamente cuando son necesarias, si un componente es decorado (patrón de diseño *Decorator*) mediante una instancia de A los encabezados de las tablas (objetos `JTableHeader`) se puede acceder (por ejemplo, para cambiar sus colores) mediante el método de `JTable`.

Aunque no son considerados componentes de *Swing*, los bordes que pueden agregarse a otros objetos son una característica interesante a considerar cuando se diseña una GUI. En el ejemplo visto, se usó un `CompoundBorder` formado por un `EmptyBorder` y un `TitledBorder`, obtenidos mediante métodos de (patrón de diseño *Factory Method*).



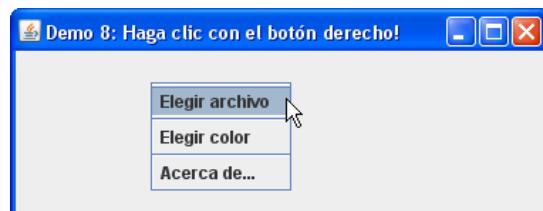
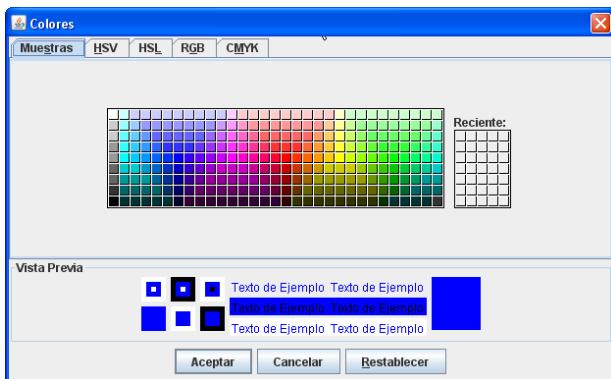
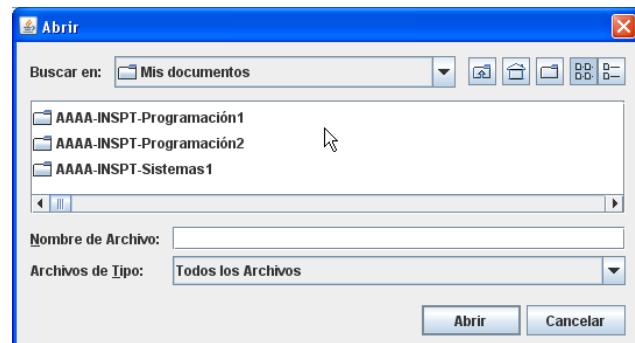
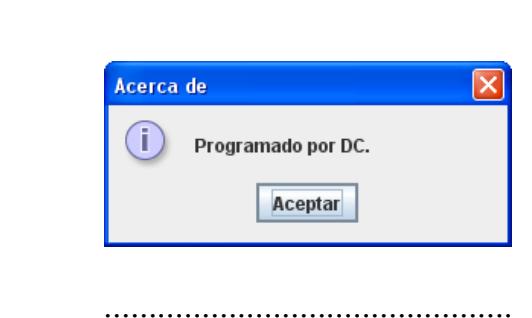
Los objetos de las clases y comparten varios métodos que heredan de `JToggleButton`, una clase concreta de *Swing* que podría extenderse para crear otros tipos de botones.

A diferencia de `JTextArea`, que sólo permite trabajar con texto plano, en *Swing* también existen las clases y (no mostradas en el ejemplo anterior) las cuales soportan contenidos en RTF y HTML. La ventaja de esta última es que, además, es capaz de contener a otros componentes, como por ejemplo, audios e íconos.

Además de utilizar, como en el ejemplo, una barra de menús desplegables (objeto), en *Swing* es posible utilizar barras de herramientas (objetos `JToolBar`, no mostrados en el ejemplo anterior) que permiten el acceso directo (con un único clic) a ciertas opciones que, en el caso de estar solamente disponibles en los menús, requerirían de varios clics para ser alcanzadas.

EJERCICIO N° 6

En el ejemplo anterior no se mostraron los siguientes componentes emergentes: `JPopupMenu`, `JFileChooser`, `JColorChooser` y `JOptionPane`. Identifíquelos en las siguientes figuras:



El siguiente ejemplo muestra cómo se utilizan estos componentes:



```
package demo8;

import java.awt.Color;
import java.awt.event.*;
import javax.swing.*;

public class Vista {
    private JFrame frame;
    private JPopupMenu popupMenu;
    private JMenuItem elegirArchivo;
    private JMenuItem elegirColor;
    private JMenuItem acercaDe;

    public Vista() {
        frame = new JFrame("Demo 8: Haga clic con el botón derecho!");
        elegirArchivo = new JMenuItem("Elegir archivo");
        elegirColor = new JMenuItem("Elegir color");
        acercaDe = new JMenuItem("Acerca de...");

        elegirArchivo.addActionListener(new EscuchaElegirArchivo());
        elegirColor.addActionListener(new EscuchaElegirColor());
        acercaDe.addActionListener(new EscuchaAcercaDe());

        popupMenu = new JPopupMenu();
        popupMenu.add(elegirArchivo);
        popupMenu.addSeparator();
        popupMenu.add(elegirColor);
        popupMenu.addSeparator();
        popupMenu.add(acercaDe);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(380, 150);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
        frame.getContentPane().addMouseListener(new EscuchaFrame());
    }

    private class EscuchaElegirArchivo implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser fc = new JFileChooser();
            fc.showOpenDialog(frame);
        }
    }

    private class EscuchaElegirColor implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            Color c = JColorChooser.showDialog(frame, "Colores", Color.BLUE);
        }
    }

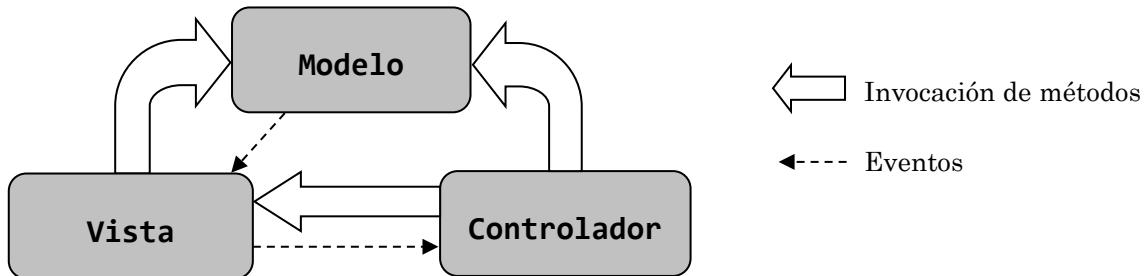
    private class EscuchaAcercaDe implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(frame, "Programado por DC.", "Acerca de",
                                         JOptionPane.INFORMATION_MESSAGE);
        }
    }

    private class EscuchaFrame extends MouseAdapter {
        public void mouseReleased(MouseEvent e) {
            if (e.isPopupTrigger()) {
                popupMenu.show(e.getComponent(), e.getX(), e.getY());
            }
        }
    }
}
```

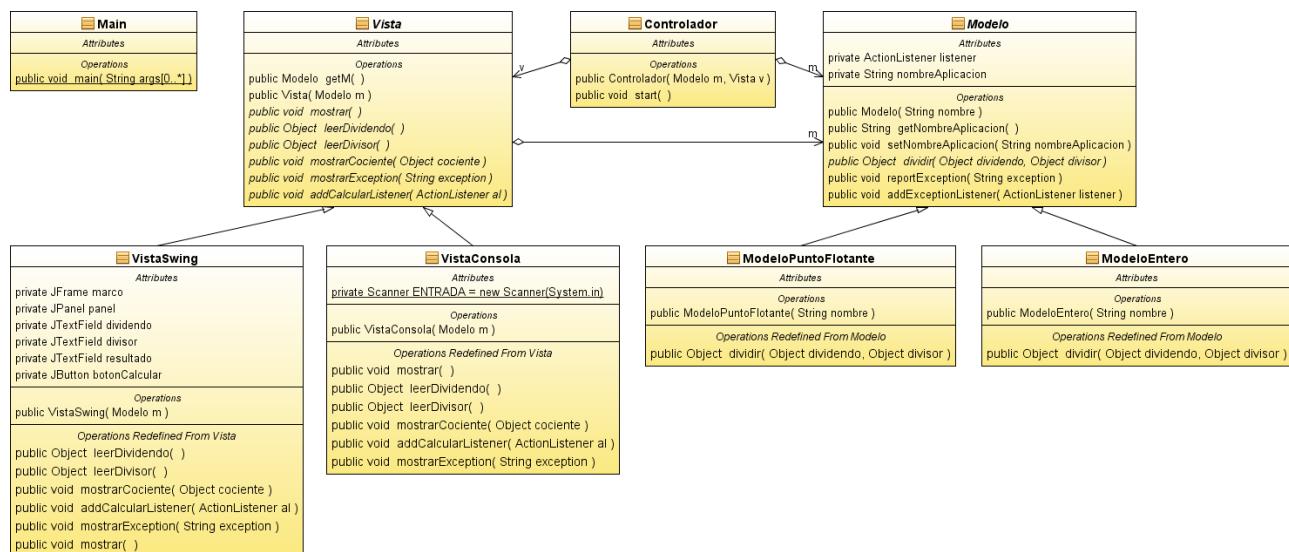
```
package demo8;
public class Main {
    public static void main(String[] args) {
        Vista v = new Vista();
    }
}
```

En el ejemplo anterior coexisten, en la misma clase, los componentes de *Swing* y el código correspondiente al comportamiento de la aplicación cuando el usuario interactúa con esos componentes. Este último se encuentra ubicado en *clases internas* de la vista, cuyas instancias (objetos *listeners*) deben registrarse en los componentes correspondientes mediante métodos que éstos proveen específicamente para ello (*addMouseListener*, *addActionListener*, etc.). De esta manera, cuando ocurre cada evento relacionado con un componente, el objeto *listener* registrado ejecuta su método correspondiente (*mouseReleased*, *actionPerformed*, etc.).

En cambio, en el patrón de arquitectura **MVC** (Modelo-Vista-Controlador), las responsabilidades están divididas. Los datos con que opera el sistema y las operaciones para procesarlos se implementan en el **modelo**. El usuario interactúa con la **vista**, la cual provee *getters* y *setters* para acceder a los contenidos de sus componentes, incluyendo métodos para registrar los objetos *listeners*, que en esta arquitectura son parte del **controlador**, ya que es éste quien responde a los eventos que se producen en la vista, y accede a ésta o al modelo según corresponda.



El siguiente ejemplo muestra una aplicación que permite al usuario elegir la implementación del modelo y la vista que desea usar. El controlador es único y funciona para cualquier modelo y vista concretos que se utilicen.





```
package mvc;
import java.awt.event.*;
public class Controlador {
    private Modelo m;
    private Vista v;
    public Controlador(Modelo m, Vista v) {
        this.m = m;
        this.v = v;
    }
    public void start() {
        m.setNombreAplicacion("MVC - " + m.getNombreAplicacion());
        v.mostrar();
        v.addCalcularListener(new CalcularListener());
    }
    private class CalcularListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            Object cociente = m.dividir(v.leerDividendo(), v.leerDivisor());
            if (cociente != null) {
                v.mostrarCociente(cociente);
            }
        }
    }
}
```

```
package mvc;
import java.awt.event.*;
public abstract class Modelo {
    private ActionListener listener;
    private String nombreAplicacion;
    public Modelo(String nombre) {
        nombreAplicacion = nombre;
    }
    public String getNombreAplicacion() {
        return nombreAplicacion;
    }
    public void setNombreAplicacion(String nombreAplicacion) {
        this.nombreAplicacion = nombreAplicacion;
    }
    public abstract Object dividir(Object dividendo, Object divisor);
    protected void reportException(String exception) {
        if (listener != null) {
            ActionEvent evt = new ActionEvent(this, 0, exception);
            listener.actionPerformed(evt);
        }
    }
    public void addExceptionListener(ActionListener listener) {
        this.listener = listener;
    }
}
```



```
package mvc;
public class ModeloEntero extends Modelo {
    public ModeloEntero(String nombre) {
        super(nombre);
    }
    public Object dividir(Object dividendo, Object divisor) {
        Integer cociente = null;
        try {
            cociente =
                Integer.parseInt(dividendo.toString())/Integer.parseInt(divisor.toString());
        } catch (Exception e) {
            reportException(e.getMessage());
        }
        return cociente;
    }
}
```

```
package mvc;
public class ModeloPuntoFlotante extends Modelo {
    public ModeloPuntoFlotante(String nombre) {
        super(nombre);
    }
    public Object dividir(Object dividendo, Object divisor) {
        Double cociente = null;
        try {
            cociente =
                Double.parseDouble(dividendo.toString())/Double.parseDouble(divisor.toString());
        } catch (Exception e) {
            reportException(e.getMessage());
        }
        return cociente;
    }
}
```

```
package mvc;
import java.awt.event.*;
public abstract class Vista {
    private Modelo m;
    public Modelo getM() {
        return m;
    }
    public Vista(Modelo m) {
        this.m = m;
        this.m.addExceptionListener(new ExceptionListener());
    }
    public abstract void mostrar();
    public abstract Object leerDividendo();
    public abstract Object leerDivisor();
    public abstract void mostrarCociente(Object cociente);
    public abstract void mostrarException(String exception);
    public abstract void addCalcularListener(ActionListener al);
    private class ExceptionListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            mostrarException(event.getActionCommand());
        }
    }
}
```



```
package mvc;

import java.awt.event.ActionListener;
import javax.swing.*;

public class VistaSwing extends Vista {

    private JFrame marco;
    private JPanel panel;
    private JTextField dividendo;
    private JTextField divisor;
    private JTextField resultado;
    private JButton botonCalcular;

    public VistaSwing(Modelo m) {
        super(m);
        marco = new JFrame();
        marco.setResizable(false);
        marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        dividendo = new JTextField(10);
        divisor = new JTextField(10);
        botonCalcular = new JButton("/");
        resultado = new JTextField(15);

        panel = new JPanel();
        panel.add(dividendo);
        panel.add(divisor);
        panel.add(botonCalcular);
        panel.add(resultado);
    }

    public Object leerDividendo() {
        return dividendo.getText();
    }

    public Object leerDivisor() {
        return divisor.getText();
    }

    public void mostrarCociente(Object cociente) {
        resultado.setText(cociente.toString());
    }

    public void addCalcularListener(ActionListener al) {
        botonCalcular.addActionListener(al);
    }

    public void mostrarException(String exception) {
        JOptionPane.showMessageDialog(null, "ERROR: " + exception,
                                   "Vista de Error", JOptionPane.ERROR_MESSAGE);
    }

    public void mostrar() {
        marco.setTitle(getM().getNombreAplicacion());
        marco.setContentPane(panel);
        marco.pack();
        marco.setLocationRelativeTo(null);
        marco.setVisible(true);
    }
}
```



```
package mvc;

import java.awt.event.*;
import java.util.Scanner;

public class VistaConsola extends Vista {

    private ActionListener al;

    public VistaConsola(Modelo m) {
        super(m);
    }
    private static final Scanner ENTRADA = new Scanner(System.in);

    public void mostrar() {
        System.out.println(getM().getNombreAplicacion());
    }

    public Object leerDividendo() {
        System.out.print("Dividendo:");
        return ENTRADA.nextLine();
    }

    public Object leerDivisor() {
        System.out.print("Divisor:");
        return ENTRADA.nextLine();
    }

    public void mostrarCociente(Object cociente) {
        System.out.println("Cociente: " + cociente);
        System.out.print("Mas cuentas? [S/N]: ");
        String st = ENTRADA.nextLine();
        char opc = (st.length() > 0 ? st.charAt(0) : '0');
        if (opc == 'S' || opc == 's') {
            System.out.println();
            mostrar();
            addCalcularListener(al);
        }
    }

    public void addCalcularListener(ActionListener al) {
        this.al = al;
        ActionEvent evt = new ActionEvent(this, 0, "Calcular");
        al.actionPerformed(evt);
    }

    public void mostrarException(String exception) {
        System.err.println("ERROR: " + exception);
    }
}
```



```
package mvc;
import javax.swing.*;
public class Main {
    public static void main(String args[]) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        Modelo m = null;
        int opcion = JOptionPane.showOptionDialog(
            null, "Que modelo desea usar?", "MVC",
            JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE, null,
            new String[]{"Entero", "Punto flotante", "Salir"}, "Entero");
        switch (opcion) {
            case 0:
                m = new ModeloEntero("Cociente Entero");
                break;
            case 1:
                m = new ModeloPuntoFlotante("Cociente Punto flotante");
                break;
            case 2:
                System.exit(0);
        }
        Vista v = null;
        opcion = JOptionPane.showOptionDialog(
            null, "Que vista desea usar?", "MVC",
            JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE, null,
            new String[]{"Consola", "Swing", "Salir"}, "Consola");
        switch (opcion) {
            case 0:
                v = new VistaConsola(m);
                break;
            case 1:
                v = new VistaSwing(m);
                break;
            case 2:
                System.exit(0);
        }
        Controlador c = new Controlador(m, v);
        c.start();
    }
}
```

CUESTIONARIO FINAL DE LA UNIDAD

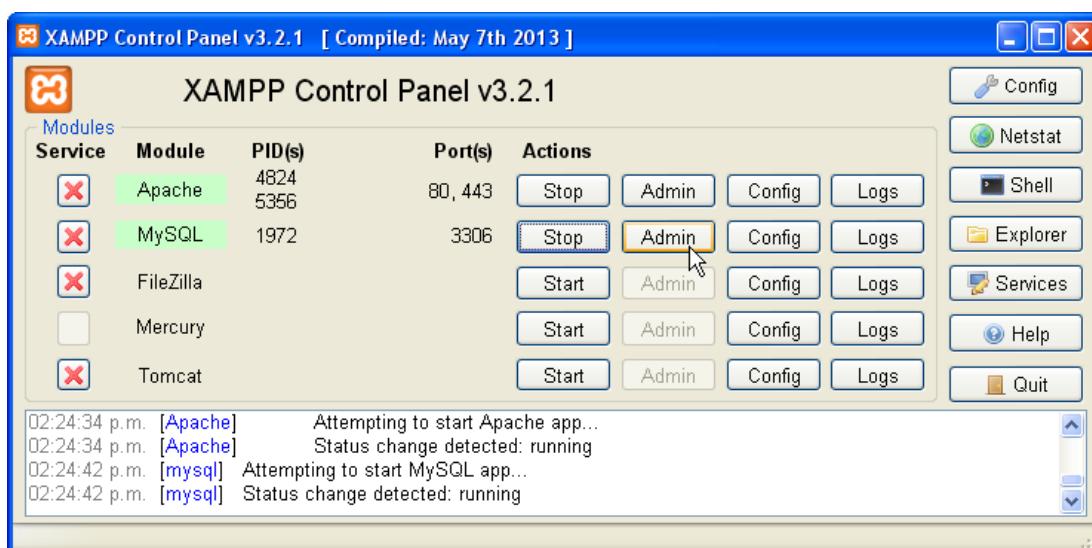
- 1) ¿Cómo y para qué se comunica el controlador con la vista?
.....
- 2) ¿Cómo y para qué se comunica la vista con el controlador?
.....
- 3) ¿Cómo y para qué se comunica el controlador con el modelo?
.....
- 4) ¿Cómo y para qué se comunica la vista con el modelo?
.....
- 5) ¿Cómo y para qué se comunica el modelo con la vista?
.....



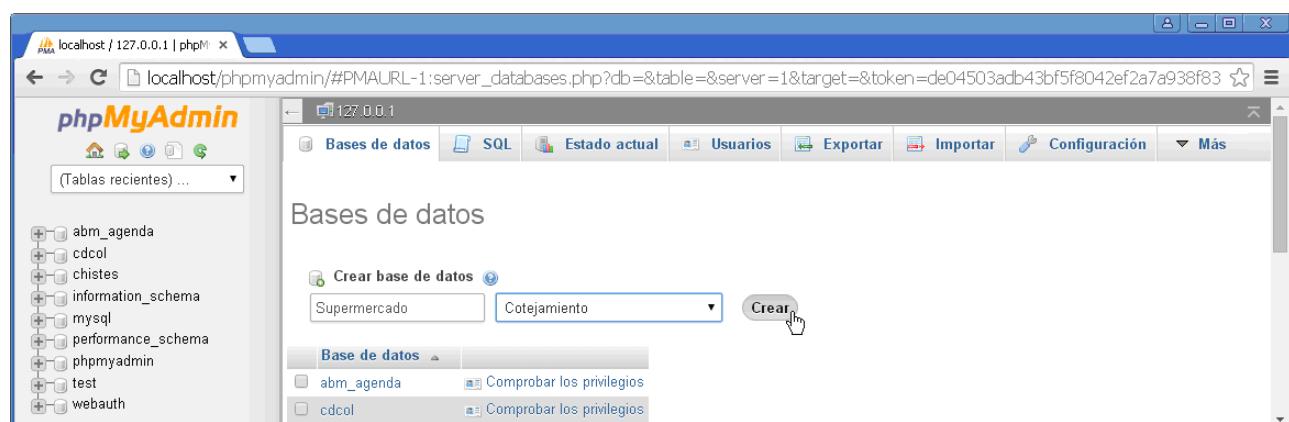
Unidad 10

Acceso a Bases de Datos

Una base de datos relacional no es más que un conjunto de datos organizados en forma de tablas relacionadas entre sí. Toda fila de una tabla es un *registro* que contiene un dato cargado en cada columna o *campo*. Esta organización permite gestionar información de manera eficiente mediante un SGBD (*Sistema Gestor de Base de Datos* o, en inglés, DBMS, *Data Base Management System*) como, por ejemplo, *MySQL*, el cual forma parte de la pila de soluciones *XAMPP* (que incluye, además, el servidor web *Apache* e intérpretes de los lenguajes *PHP* y *Perl*, entre otras soluciones). A través del panel de control de *XAMPP*, es posible administrar *MySQL* usando un simple navegador web, por medio de *phpMyAdmin*¹.



Esta herramienta (*phpMyAdmin*) permite interactuar fácilmente con *MySQL* para realizar con facilidad las operaciones básicas conocidas como CRUD (*Create, Read, Update* y *Delete*), equivalentes en español a ABMC (*Altas, Bajas, Modificaciones y Consultas*).



¹ Con *XAMPP*, para poder usar *phpMyAdmin*, el servidor Apache debe estar en ejecución.



Mis anotaciones: Cómo se hace en phpMyAdmin para:

Crear una base de datos

.....
.....

Eliminar una base de datos

.....
.....

Exportar una base de datos

.....
.....

Importar una base de datos

.....
.....

Crear un usuario para una base de datos

.....
.....

Eliminar un usuario de una base de datos

.....
.....

Crear una tabla en una base de datos

.....
.....

Eliminar una tabla en una base de datos

.....
.....

Agregar un campo en una tabla de una base de datos

.....
.....

Eliminar un campo en una tabla de una base de datos

.....
.....

Agregar un registro en una tabla de una base de datos

.....
.....

Eliminar un registro en una tabla de una base de datos

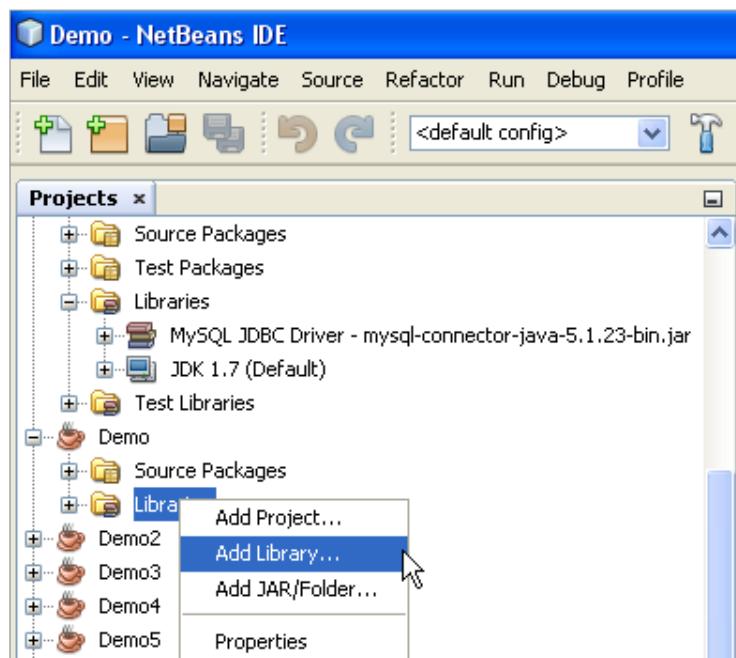
.....
.....



Para trabajar con bases de datos, existe un lenguaje estructurado de consultas llamado *SQL* (*Structured Query Language*). Este lenguaje está compuesto por un DDL (*Data Definition Language*) que provee sentencias para crear, modificar y eliminar tablas y por un DML (*Data Manipulation Language*) que provee sentencias para manipular los datos realizando consultas (*queries*) de campos o modificaciones (*updates*) de registros (como por ejemplo, insertar, modificar y eliminar registros). El siguiente cuadro resume las operaciones que el DML permite realizar:

Operación	Sentencia SQL	Tipo de manipulación
C (Create)	INSERT	Modificación (<i>update</i>)
R (Read)	SELECT	Consulta (<i>query</i>)
U (Update)	UPDATE	Modificación (<i>update</i>)
D (Delete)	DELETE	Modificación (<i>update</i>)

La API que permite utilizar bases de datos en nuestros programas escritos en Java se llama *JDBC* (*Java Database Connectivity*) y se encuentra dentro del paquete `java.sql`. Se trata de un conjunto de interfaces que definen (por ejemplo) la manera de establecer la conexión, la forma de ejecutar sentencias (*queries* o *updates*), etc. Para poder trabajar con cada SGBD específico, es necesario disponer de un conjunto de clases que implementen todas estas interfaces. A este conjunto de clases se lo denomina *driver*.



A continuación, se ejemplifica la utilización de *JDBC* a través de una aplicación diseñada según el patrón MVC, de tal manera que en el modelo se concentran todos los datos, algunos en el propio objeto m - en forma de atributos - (como, por ejemplo, los textos que se utilizan en la vista) y otros en una base de datos gestionada mediante *MySQL* y denominada *agenda*. La aplicación usa una única tabla (*datos*) que posee dos campos (*nombre* y *telefono*). Para que la conexión se pueda llevar a cabo, es necesario que el SGBD se encuentre corriendo localmente y que la base de datos permita accesos anónimos (cualquier usuario, sin contraseña).



```
package agenda;

public class Main {

    public static void main(String[] args) {
        Modelo m = new Modelo();
        Vista v = new Vista(m);
        Controlador c = new Controlador(m, v);
        c.ejecutar();
    }
}
```

```
package agenda;

import java.awt.event.*;
import java.sql.*;

public class Modelo {

    private String tituloApp;
    private String tituloSalida;
    private String tituloExcepcion;
    private String tituloNombre;
    private String tituloTelefono;
    private String tituloNuevoTelefono;
    private String tituloBotonAlta;
    private String tituloBotonBaja;
    private String tituloBotonModificacion;
    private String tituloBotonConsulta;
    private String tituloRegistros;
    private String tituloConsultaVacia;
    private String driver;
    private String prefijoConexion;
    private String ip;
    private String usr;
    private String psw;
    private String bd;
    private String tabla;
    private String campoNombre;
    private String campoTelefono;
    private String resultadoConsulta;
    private int cantidadRegistros;
    private Connection connection;
    private ActionListener listener;

    public Modelo() {
        tituloApp = "Mi Agenda";
        tituloSalida = "Resultado de la operación";
        tituloExcepcion = "Error!";
        tituloNombre = "Nombre: ";
        tituloTelefono = "Teléfono: ";
        tituloNuevoTelefono = "Nuevo teléfono: ";
        tituloBotonAlta = "Alta";
        tituloBotonBaja = "Baja";
        tituloBotonModificacion = "Modificación";
        tituloBotonConsulta = "Consulta";
        tituloRegistros = "Cantidad de registros actualizados: ";
        tituloConsultaVacia = "No se ha encontrado a ";
    }
}
```



```
driver = "com.mysql.cj.jdbc.Driver";
prefijoConexion = "jdbc:mysql://";
ip = "127.0.0.1";           // Dirección IP donde está corriendo el SGBD
usr = "";                   // Usuario
psw = "";                   // Password
bd = "agenda";              // Base de datos que contiene la tabla a usar
tabla = "datos";             // Tabla de la agenda
campoNombre = "nombre";      // Campo que contiene el nombre
campoTelefono = "telefono";   // Campo que contiene el teléfono
}

public String getTituloApp() {
    return tituloApp;
}

public String getTituloSalida() {
    return tituloSalida;
}

public String getTituloExcepcion() {
    return tituloExcepcion;
}

public String getTituloNombre() {
    return tituloNombre;
}

public String getTituloTelefono() {
    return tituloTelefono;
}

public String getTituloNuevoTelefono() {
    return tituloNuevoTelefono;
}

public String getTituloBotonAlta() {
    return tituloBotonAlta;
}

public String getTituloBotonBaja() {
    return tituloBotonBaja;
}

public String getTituloBotonModificacion() {
    return tituloBotonModificacion;
}

public String getTituloBotonConsulta() {
    return tituloBotonConsulta;
}

public String getTituloRegistros() {
    return tituloRegistros;
}

public String getResultadoConsulta() {
    return resultadoConsulta;
}

public int getCantidadRegistros() {
    return cantidadRegistros;
}
```



```
public void darDeAlta(String nombre, String telefono) {  
    connection = obtenerConexion(); ← 1  
    String u = "INSERT INTO " + tabla + " (" + campoNombre + ", " + campoTelefono +  
        ") " + " VALUES ('" + nombre + "', '" + telefono + "')";  
    try {  
        Statement statement = connection.createStatement(); ← 6  
        cantidadRegistros = statement.executeUpdate(u);  
        statement.close(); ← 10 ← 7  
    } catch (SQLException ex) {  
        reportException(ex.getMessage());  
    }  
}  
  
public void darDeBaja(String nombre) {  
    connection = obtenerConexion(); ← 1  
    String u = "DELETE FROM " + tabla + " WHERE " + campoNombre + "=''" + nombre + "'";  
    try {  
        Statement statement = connection.createStatement(); ← 6  
        cantidadRegistros = statement.executeUpdate(u);  
        statement.close(); ← 10 ← 7  
    } catch (SQLException ex) {  
        reportException(ex.getMessage());  
    }  
}  
  
public void modificar(String nombre, String telefono) {  
    connection = obtenerConexion(); ← 1  
    String u = "UPDATE " + tabla + " SET " + campoNombre + "=''" + nombre + ", " +  
        campoTelefono + "=''" + telefono + "' " +  
        " WHERE " + campoNombre + "=''" + nombre + "'";  
    try {  
        Statement statement = connection.createStatement(); ← 6  
        cantidadRegistros = statement.executeUpdate(u);  
        statement.close(); ← 10 ← 7  
    } catch (SQLException ex) {  
        reportException(ex.getMessage());  
    }  
}  
  
public void consultar(String nombre) {  
    connection = obtenerConexion(); ← 1  
    String q = "SELECT " + campoNombre + ", " + campoTelefono + " FROM " + tabla +  
        " WHERE " + campoNombre + "=''" + nombre + "'";  
    try {  
        Statement statement = connection.createStatement(); ← 6  
        ResultSet resultSet = statement.executeQuery(q);  
        resultadoConsulta = "";  
        while (resultSet.next()) {  
            resultadoConsulta = resultadoConsulta + resultSet.getString(1) + ":" +  
                resultSet.getString(2) + System.getProperty("line.separator");  
        }  
        if (resultadoConsulta.isEmpty() && !nombre.isEmpty()) {  
            resultadoConsulta = tituloConsultaVacia + nombre + ".";  
        }  
        resultSet.close(); ← 9 ← 10  
        statement.close();  
    } catch (SQLException ex) {  
        reportException(ex.getMessage());  
    }  
}
```



```
public void addExceptionListener(ActionListener listener) {
    this.listener = listener;
}

private void reportException(String exception) {
    if (listener != null) {
        ActionEvent evt = new ActionEvent(this, 0, exception);
        listener.actionPerformed(evt);
    }
}

private Connection obtenerConexion() {
    if (connection == null) { ← ②
        try {
            Class.forName(driver); ← ③
        } catch (ClassNotFoundException ex) {
            reportException(ex.getMessage());
        }
        try {
            connection = ← ④
            DriverManager.getConnection(prefijoConexion + ip + "/" + bd, usr, psw);
        } catch (Exception ex) {
            reportException(ex.getMessage());
        }
        Runtime.getRuntime().addShutdownHook(new ShutDownHook()); ← ⑤
    }
    return connection;
}

private class ShutDownHook extends Thread {
    @Override
    public void run() {
        try {
            if (connection != null) {
                connection.close(); ← ⑥
            }
        } catch (SQLException ex) {
            reportException(ex.getMessage());
        }
    }
}
```

```
package agenda;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Vista {

    private Modelo m;
    private JFrame f;
    private JPanel p;
    private JPanel panelLabelAlta;
    private JPanel panelAlta;
    private JPanel panelLabelModificacion;
    private JPanel panelModificacion;
    private JLabel labelNombreAlta;
    private JLabel labelTelefonoAlta;
```



```
private JLabel labelNombreBaja;
private JLabel labelNombreModificacion;
private JLabel labelTelefonoModificacion;
private JLabel labelNombreConsulta;
private JTextField nombreAlta;
private JTextField telefonoAlta;
private JTextField nombreBaja;
private JTextField nombreModificacion;
private JTextField telefonoModificacion;
private JTextField nombreConsulta;
private JButton botonAlta;
private JButton botonBaja;
private JButton botonModificacion;
private JButton botonConsulta;

public Vista(Modelo m) {
    this.m = m;
    this.m.addExceptionListener(new ExceptionListener());
    f = new JFrame();
    p = new JPanel();
    panelLabelAlta = new JPanel();
    panelAlta = new JPanel();
    panelLabelModificacion = new JPanel();
    panelModificacion = new JPanel();
    labelNombreAlta = new JLabel(m.getTituloNombre());
    labelTelefonoAlta = new JLabel(m.getTituloTelefono());
    labelNombreBaja = new JLabel(m.getTituloNombre());
    labelNombreModificacion = new JLabel(m.getTituloNombre());
    labelTelefonoModificacion = new JLabel(m.getTituloNuevoTelefono());
    labelNombreConsulta = new JLabel(m.getTituloNombre());
    labelNombreAlta.setHorizontalAlignment(SwingConstants.RIGHT);
    labelTelefonoAlta.setHorizontalAlignment(SwingConstants.RIGHT);
    labelNombreBaja.setHorizontalAlignment(SwingConstants.RIGHT);
    labelNombreModificacion.setHorizontalAlignment(SwingConstants.RIGHT);
    labelTelefonoModificacion.setHorizontalAlignment(SwingConstants.RIGHT);
    labelNombreConsulta.setHorizontalAlignment(SwingConstants.RIGHT);
    nombreAlta = new JTextField();
    telefonoAlta = new JTextField();
    nombreBaja = new JTextField();
    nombreModificacion = new JTextField();
    telefonoModificacion = new JTextField();
    nombreConsulta = new JTextField();
    botonAlta = new JButton(m.getTituloBotonAlta());
    botonBaja = new JButton(m.getTituloBotonBaja());
    botonModificacion = new JButton(m.getTituloBotonModificacion());
    botonConsulta = new JButton(m.getTituloBotonConsulta());
    panelLabelAlta.setBackground(new Color(255, 128, 128));
    labelNombreBaja.setOpaque(true);
    labelNombreBaja.setBackground(new Color(128, 255, 128));
    panelLabelModificacion.setBackground(new Color(128, 128, 255));
    labelNombreConsulta.setOpaque(true);
    labelNombreConsulta.setBackground(new Color(255, 255, 128));
    nombreAlta.setBackground(new Color(255, 144, 144));
    telefonoAlta.setBackground(new Color(255, 144, 144));
    nombreBaja.setBackground(new Color(144, 255, 144));
    nombreModificacion.setBackground(new Color(144, 144, 255));
    telefonoModificacion.setBackground(new Color(144, 144, 255));
    nombreConsulta.setBackground(new Color(255, 255, 144));
    p.setLayout(new GridLayout(4, 3));
    panelLabelAlta.setLayout(new GridLayout(2, 1));
```



```
panelAlta.setLayout(new GridLayout(2, 1));
panelLabelModificacion.setLayout(new GridLayout(2, 1));
panelModificacion.setLayout(new GridLayout(2, 1));
panelLabelAlta.add(labelNombreAlta);
panelLabelAlta.add(labelTelefonoAlta);
panelAlta.add(nombreAlta);
panelAlta.add(telefonoAlta);
panelLabelModificacion.add(labelNombreModificacion);
panelLabelModificacion.add(labelTelefonoModificacion);
panelModificacion.add(nombreModificacion);
panelModificacion.add(telefonoModificacion);
p.add(panelLabelAlta);
p.add(panelAlta);
p.add(botonAlta);
p.add(labelNombreBaja);
p.add(nombreBaja);
p.add(botonBaja);
p.add(panelLabelModificacion);
p.add(panelModificacion);
p.add(botonModificacion);
p.add(labelNombreConsulta);
p.add(nombreConsulta);
p.add(botonConsulta);
f.getContentPane().add(p);
}

public void mostrar() {
    f.setTitle(m.getTituloApp());
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setLocationRelativeTo(null);
    f.pack();
    f.setVisible(true);
}

public String getNombreAltaStr() {
    return nombreAlta.getText();
}

public String getTelefonoAltaStr() {
    return telefonoAlta.getText();
}

public String getNombreBajaStr() {
    return nombreBaja.getText();
}

public String getNombreModificacionStr() {
    return nombreModificacion.getText();
}

public String getTelefonoModificacionStr() {
    return telefonoModificacion.getText();
}

public String getNombreConsultaStr() {
    return nombreConsulta.getText();
}

public void addBotonAltaListener(ActionListener al) {
```



```
        botonAlta.addActionListener(al);
    }

    public void addBotonBajaListener(ActionListener al) {
        botonBaja.addActionListener(al);
    }

    public void addBotonModificacionListener(ActionListener al) {
        botonModificacion.addActionListener(al);
    }

    public void addBotonConsultaListener(ActionListener al) {
        botonConsulta.addActionListener(al);
    }

    public void mostrarCadena(String s) {
        JOptionPane.showMessageDialog(f, s, m.getTituloSalida(),
                                      JOptionPane.PLAIN_MESSAGE);
    }

    public void mostrarCantidadDeRegistros(int cant) {
        JOptionPane.showMessageDialog(f, m.getTituloRegistros() + cant,
                                      m.getTituloSalida(), JOptionPane.INFORMATION_MESSAGE);
    }

    public void mostrarExcepcion(String s) {
        JOptionPane.showMessageDialog(f, s, m.getTituloExcepcion(),
                                      JOptionPane.ERROR_MESSAGE);
    }

    private class ExceptionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent event) {
            mostrarExcepcion(event.getActionCommand());
        }
    }
}
```

```
package agenda;

import java.awt.event.*;

public class Controlador {

    private Modelo m;
    private Vista v;

    public Controlador(Modelo m, Vista v) {
        this.m = m;
        this.v = v;
    }

    public void ejecutar() {
        v.mostrar();
        v.addBotonAltaListener(new AltaListener());
        v.addBotonBajaListener(new BajaListener());
        v.addBotonModificacionListener(new ModificacionListener());
        v.addBotonConsultaListener(new ConsultaListener());
    }

    private class AltaListener implements ActionListener {
```



```
    @Override
    public void actionPerformed(ActionEvent event) {
        m.darDeAlta(v.getNombreAltaStr(), v.getTelefonoAltaStr());
        v.mostrarCantidadDeRegistros(m.getCantidadRegistros());
    }
}

private class BajaListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent event) {
        m.darDeBaja(v.getNombreBajaStr());
        v.mostrarCantidadDeRegistros(m.getCantidadRegistros());
    }
}

private class ModificacionListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent event) {
        m.modificar(v.getNombreModificacionStr(), v.getTelefonoModificacionStr());
        v.mostrarCantidadDeRegistros(m.getCantidadRegistros());
    }
}

private class ConsultaListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent event) {
        m.consultar(v.getNombreConsultaStr());
        v.mostrarCadena(m.getResultadoConsulta());
    }
}
```



Al arrancar el controlador, éste le envía a la vista los objetos que escucharán los clics en los botones. Estos objetos *listeners* pertenecen a clases internas del controlador y, por lo tanto, tienen acceso al modelo y a la vista (a través de los atributos *m* y *v*, respectivamente). Así, cuando el usuario, por ejemplo, hace clic en el botón de *Alta*, el controlador puede obtener los datos de la vista y pasárselos al modelo para que los guarde en la base de datos, y luego puede obtener del modelo los resultados y pasárselos a la vista para que ésta los muestre.



Mis anotaciones: ¿Cómo se utiliza JDBC para ejecutar sentencias en SQL?

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

CUESTIONARIO FINAL DE LA UNIDAD

- 1) En SQL, ¿cuál es la sintaxis de la sentencia INSERT?
- 2) En SQL, ¿cuál es la sintaxis de la sentencia DELETE?
- 3) En SQL, ¿cuál es la sintaxis de la sentencia UPDATE?
- 4) En SQL, ¿cuál es la sintaxis de la sentencia SELECT?
- 5) En SQL, ¿cuál es la sintaxis de las sentencias correspondientes a cada una de las operaciones listadas en la pág. 192? Anótelas allí como una alternativa a phpMyAdmin.



Unidad 11

Aplicaciones Web

Las *aplicaciones web* son unas herramientas informáticas que están basadas en la **arquitectura cliente-servidor** y que se pueden utilizar mediante un **navegador web** (como, por ejemplo, *Chrome*, *Firefox* o *Safari*) a través de Internet o de una intranet. Son populares debido, entre otros motivos, a la practicidad de usar un navegador web como cliente ligero, a la independencia del sistema operativo y a la facilidad de su mantenimiento sin necesidad de distribuir e instalar software a los usuarios. Los *webmails* y las tiendas en línea son ejemplos de *aplicaciones web*.

Básicamente, el navegador web (o sea, el cliente) tiene dos funciones:

1. recibir una petición (*request*) del usuario, darle formato según el protocolo de transferencia que se usará (generalmente *HTTP - HyperText Transfer Protocol*) y enviársela al servidor;
2. recibir una respuesta (*response*) del servidor. Si ésta contiene una página web, procesarla con un motor de renderizado (por ejemplo, *Blink*, *Gecko* o *WebKit*) y mostrarla. De lo contrario, seguir el protocolo (por ejemplo, descargar un archivo).

El servidor también tiene básicamente dos funciones:

1. recibir la petición (*request*) que fue enviada desde el cliente e intentar localizar un recurso para responderla (si es una petición estática, el recurso será algún tipo de archivo como, por ejemplo, una página web, una imagen o un video; si es una petición dinámica, el recurso a localizar será un *script* o un programa);
2. darle formato, según el protocolo de transferencia que se usará, a la respuesta (*response*) a entregar o, si el recurso no se pudo localizar, a una página indicadora del error (por ejemplo, *HTTP Error 404*), y enviársela al cliente.

Del lado del servidor, es posible encontrar:

- **Servidores web** (también conocidos como *servidores HTTP*): son sistemas que, usando *HTTP*, reciben peticiones y entregan páginas web como respuesta, tanto estáticas (las páginas son archivos) como dinámicas (las páginas son generadas por un *script* escrito en un lenguaje cuyo intérprete está integrado en el servidor, como, por ejemplo, *PHP* o *Perl*). Los más ampliamente utilizados son *Apache HTTP Server* y *Nginx* (pronunciado en inglés “engine X”).
- **Contenedores web** (también conocidos como *contenedores de servlets*): son sistemas que proveen un entorno de ejecución para *servlets* (programas escritos en Java para recibir peticiones/*requests* y entregar respuestas/*responses*) y que administran el ciclo de vida de éstos (en lugar del método *main*, los *servlets* poseen los métodos *init*, *service* y *destroy*, que son invocados, cuando corresponde, por el contenedor). Los más ampliamente usados son *Apache Tomcat* y *Eclipse Jetty*, los cuales también poseen la funcionalidad de servidor web.
- **Servidores de aplicaciones**: son sistemas ubicados entre los servidores del *back-end* (aplicaciones de negocios o bases de datos) y los clientes del *front-end* (los cuales no necesariamente usan *HTTP*). Si bien existen servidores de aplicaciones específicos para casi todos los lenguajes de programación (para Python, por ejemplo, existe *Zope*), la mayoría corren aplicaciones escritas en Java (EE), como es, por ejemplo, el caso de *Glassfish*, el cual también posee las funcionalidades de contenedor web y de servidor web.



A continuación, desarrollaremos una sencilla aplicación web utilizando *NetBeans*. Simplemente hay que ir al siguiente asistente para la creación del proyecto:

```
File → New Project → Java with Maven → Web Application
```

Allí hay que completar los datos solicitados:

Name and Location

Project Name: *cerveceria*
Project location: [no cambiarla]
Group Id: *prog2*
Version: 1.0
Package: *prog2.cerveceria*

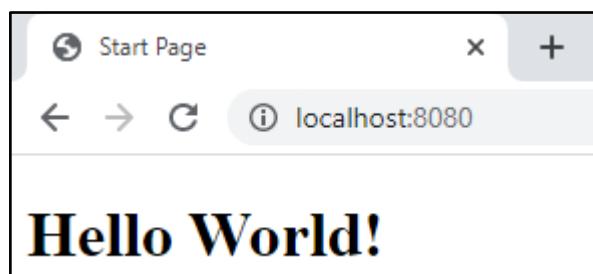
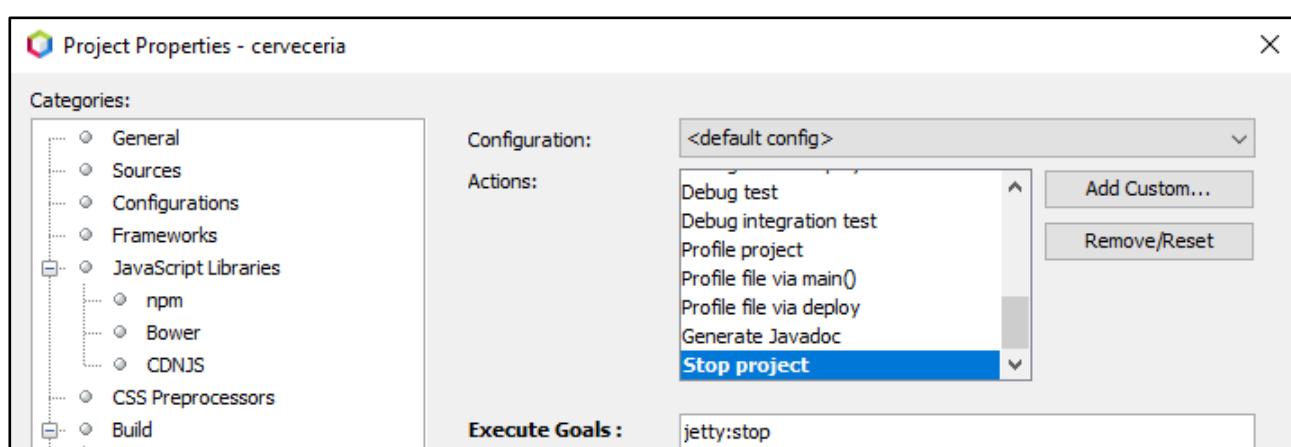
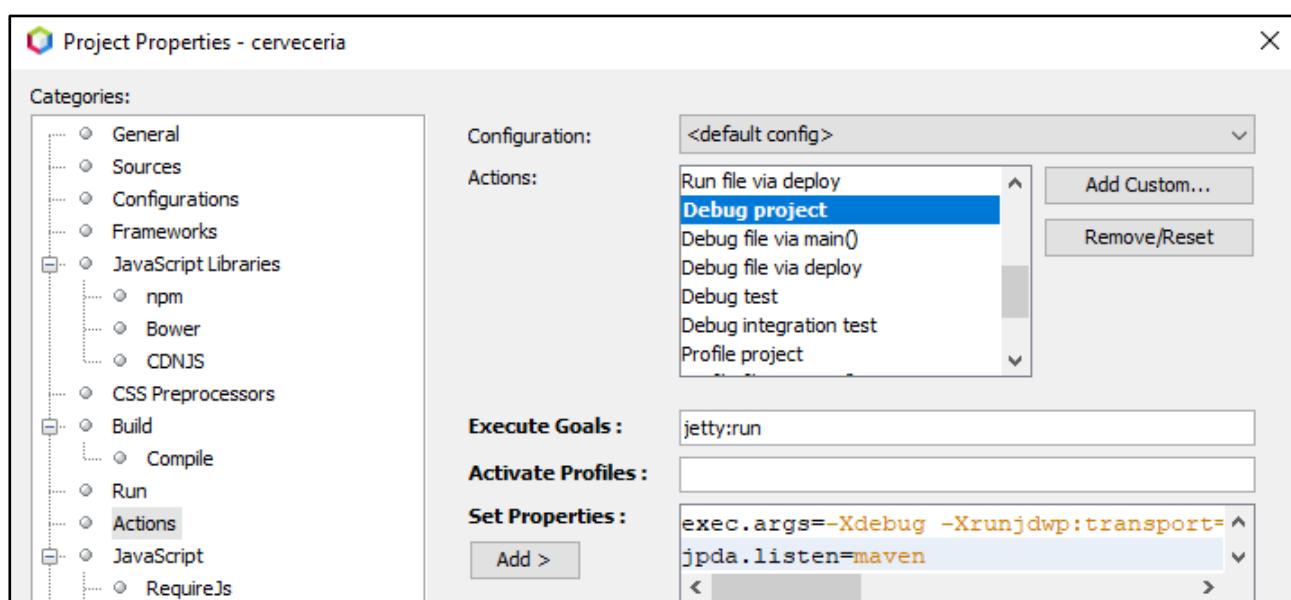
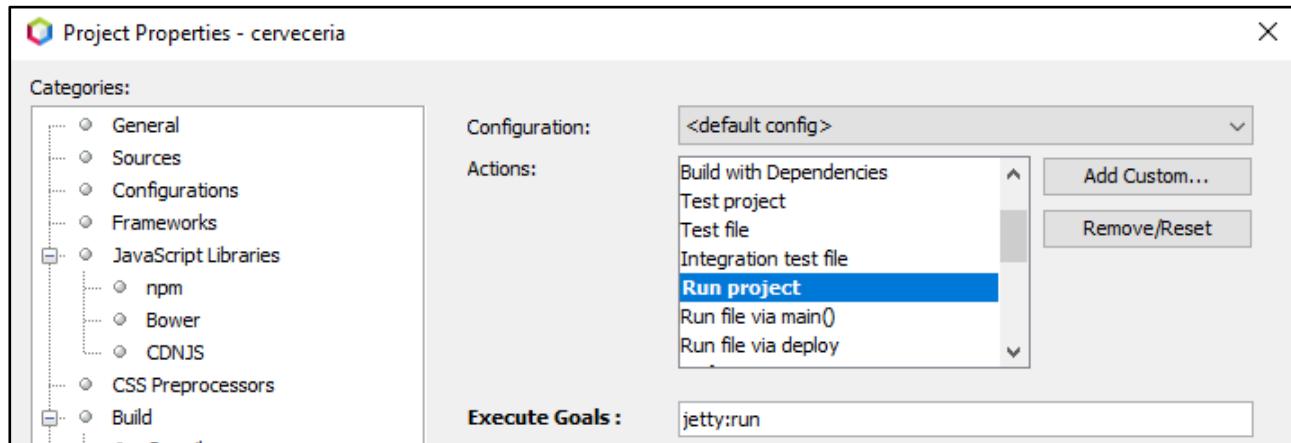
Settings

Web Server: <No server selected>
Java EE Version: Java EE 7 Web

La aplicación generada automáticamente sólo contiene una página web (*index.html*) que recién correrá (en *localhost:8080*) cuando se haya instalado un Web Server (contenedor de *servlets*). Por su practicidad, la opción elegida aquí es *Eclipse Jetty*, que se puede hacer funcionar agregando el siguiente *plugin* en el archivo *pom.xml* (en *plugins*):

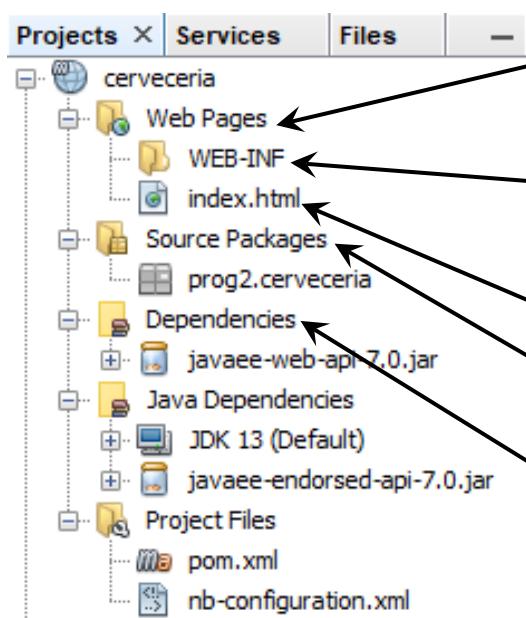
```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.4.34.v20201102</version>
  <configuration>
    <stopKey>stop</stopKey>
    <stopPort>8081</stopPort>
  </configuration>
</plugin>
```

Por último, en las propiedades del proyecto, se deben editar dos de las *Actions* predeterminadas (*Run project* y *Debug project*) y cargar una nueva acción personalizada (*Stop project*, creada en *Add Custom..*). En *Run project* debe completarse, en *Execute Goals*, la meta *jetty:run*; en *Debug project*, además de completarse en *Execute Goals* la meta *jetty:run*, debe cambiarse *jpda.listen=true* por *jpda.listen=maven*, y en *Stop project* debe completarse, en *Execute Goals*, la meta *jetty:stop*. De esta manera, *Jetty* arrancará al correr la aplicación; para detenerlo existirá la acción *Stop project* en *Run Maven* (al hacerle clic derecho al proyecto), y además podrá depurarse el proyecto (con las opciones del menú *Debug* de *NetBeans*).





A pesar de su pequeño tamaño, esta aplicación permite mostrar la distribución de los componentes de cualquier proyecto de este tipo:



Carpeta donde estarán las páginas web (*JSP, HTML, etc.*) y las subcarpetas con sus recursos (*CSS, scripts, imágenes, videos, etc.*)

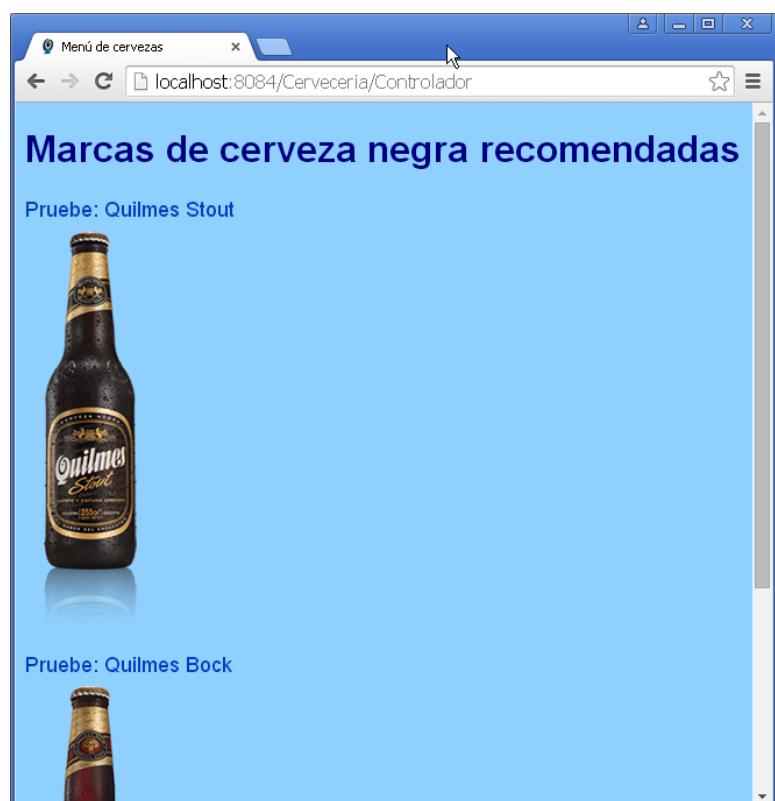
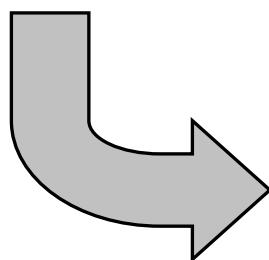
Carpeta que contendrá el descriptor de despliegue *web.xml* con el registro de *servlets* (solo es necesario si no se usan *annotations*)

Página de inicio

Carpeta que contendrá las clases de Java (*servlets, JavaBeans y POJOs*)

Carpeta donde se agregarán bibliotecas necesarias para acceder a los SGBD y a otros recursos

La aplicación web que desarrollaremos nos permitirá seleccionar un color de cerveza y nos recomendará marcas de cerveza del color seleccionado.





Las consultas se realizarán en la tabla **cervezas** de la base de datos **cervecería**.

The screenshot shows the MySQL Workbench interface. On the left, there's a tree view with nodes for '127.0.0.1', 'cerveceria', and 'cervezas'. Under 'cervezas', there are two tabs: 'Estructura' (Structure) and 'Datos' (Data). The 'Estructura' tab displays the table structure with three columns: '#', 'Nombre' (Name), and 'Tipo' (Type). The 'Datos' tab displays the data with three columns: 'color', 'marca' (brand), and 'foto' (photo).

#	Nombre	Tipo
1	color	varchar(20)
2	marca	varchar(30)
3	foto	varchar(30)

color	marca	foto
roja	Quilmes Red Lager	redlager.png
rubia	Quilmes Cristal	cristal.png
negra	Quilmes Stout	stout.png
negra	Quilmes Bock	bock.png

La tabla **cervezas** deberá poder ser consultada por cualquier usuario anónimo:

The screenshot shows the 'Agregar un nuevo usuario' (Add new user) dialog in MySQL Workbench. It has tabs for 'Juegos de caracteres' (Character sets), 'Motores' (Engines), 'Privilegios' (Privileges), and 'Procesos' (Processes). Below these tabs, there are buttons for 'Exportar' (Export) and 'Importar' (Import). The main form contains fields for 'Nombre de usuario' (User name), 'Servidor' (Server), and 'Contraseña' (Password), all set to 'Cualquier usuario' (Any user).

Para poder usar la base de datos, en *Dependencies* debe agregarse el *driver JDBC* para MySQL. Esto puede hacerse colocando la dependencia en el archivo *pom.xml* o mediante el siguiente asistente que se abre en *Dependencies* → *Add Dependency*

The screenshot shows the 'Search' interface for Maven dependencies. The 'Query' field contains 'mysql'. The 'Search Results' section lists several MySQL-related dependencies, with 'mysql:mysql-connector-java:8.0.22 [jar] - central' highlighted.



Veremos tres arquitecturas posibles para desarrollar esta aplicación web: usando sólo *servlets* (como se hacía antes del lanzamiento de *JSP / JavaServer Pages* en 1998), usando sólo *JSP* (arquitectura *JSP Modelo-1*) y usando ambos para seguir el patrón *MVC* (arquitectura *JSP Modelo-2*). Las dos primeras sólo tienen valor ilustrativo e histórico y, siempre que sea posible, se recomienda evitarlas, ya que utilizar la arquitectura *JSP Modelo-2 (MVC)* ofrece múltiples ventajas.

La primera variante utiliza una página web estática (en *HTML*), desde la cual se envía una petición que será atendida por el *servlet CerveceriaServlet*.

index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <h1>Menú de cervezas</h1>
        <form method="post" action="CerveceriaServlet">
            <p>
                Seleccione el color:
                <select name="color" size="1">
                    <option value="rubia"> rubia </option>
                    <option value="roja"> roja </option>
                    <option value="negra"> negra </option>
                    <option value="verde"> verde </option>
                </select>
                <input type="hidden" name="dirIP" value="localhost">
                <input type="hidden" name="nomBD" value="cerveceria">
                <input type="submit" value ="Enviar">
            </p>
        </form>
    </body>
</html>
```

CerveceriaServlet.java

```
package prog2.cerveceria;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(name = "CerveceriaServlet", urlPatterns = {"/CerveceriaServlet"})
public class CerveceriaServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
}
```



```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    String ip = request.getParameter("dirIP");
    String bd = request.getParameter("nomBD");
    String c = request.getParameter("color");
    response.setContentType("text/html; charset=UTF-8");
    String jdbcDriver = "com.mysql.cj.jdbc.Driver";
    String url = "jdbc:mysql://" + ip + "/" + bd;
    PrintWriter out = response.getWriter();
    try {
        out.println("<!DOCTYPE html>");
        out.println("<html>\n<head>");
        out.println("<meta http-equiv=\"Content-Type\" "
            + "content=\"text/html; charset=utf-8\">");
        out.println("<link rel=\"stylesheet\" "
            + "type=\"text/css\" href=\"css/estilo.css\"");
        out.println("<link rel=\"icon\" "
            + "type=\"image/x-icon\" href=\"img/favicon.ico\"");
        out.println("<link rel=\"shortcut icon\" "
            + "type=\"image/x-icon\" href=\"img/favicon.ico\"");
        out.println("<title>Menú de cervezas</title>");
        out.println("</head>\n<body>");
        try {
            Class.forName(jdbcDriver);
            Connection con = DriverManager.getConnection(url, "", "");
            Statement stmt = con.createStatement();
            stmt.execute("SELECT marca, color, foto FROM cervezas WHERE color='" + c + "'");
            ResultSet rs = stmt.getResultSet();
            if (rs.isBeforeFirst()) {
                out.println("<h1>Marcas de cerveza " + c + " recomendadas</h1>");
                while (rs.next()) {
                    out.println("<p>Pruebe: " + rs.getString(1) + "<br><img src=\"img/" +
                        + rs.getString(3) + "\" alt=\"\" " + rs.getString(1) + "\"</p>");
                }
                rs.close();
            } else {
                out.println("<h1>ERROR</h1><p>Lamentablemente, no hay ninguna cerveza " +
                    + c + " en nuestra base de datos.</p>");
            }
            stmt.close();
            con.close();
        } catch (ClassNotFoundException | SQLException ex) {
            out.println("<h1>ERROR</h1><p>" + ex.getMessage() + "</p>");
        }
        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}
```

Los datos recibidos desde el cliente están almacenados en una instancia de `HttpServletRequest`, de la cual pueden ser obtenidos individualmente invocando el método con el nombre del dato.



La hoja de estilos utilizada por `index.html` es muy básica:

`estilo.css`

```
body{
    background:#90D0FF;
}

h1{
    font-family:Arial, "Trebuchet MS", Helvetica, sans-serif;
    font-size:32px;
    color:#000080;
}

p{
    font-family:Arial, "Trebuchet MS", Helvetica, sans-serif;
    font-size:18px;
    color:#0040C0;
}
```

Evidentemente, aunque la aplicación funciona correctamente, colocar todo el código *HTML* de las respuestas dentro del *servlet* no es una buena idea, ya que, al mezclar la presentación y la lógica, la aplicación se vuelve difícil de mantener y de cambiar.

A continuación, se muestra la arquitectura *JSP Modelo-1*, que sólo utiliza páginas *JSP* y *JavaBeans*. Las clases *JavaBeans* son clases que siguen ciertas convenciones: deben tener un constructor sin parámetros, atributos privados, *getters* y *setters* con nomenclatura estándar y, además, deben ser serializables.

`index.jsp`

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <h1>Menú de cervezas</h1>
        <form method="post" action="procesar.jsp">
            <p>
                Seleccione el color:
                <select name="color" size="1">
                    <option value="rubia"> rubia </option>
                    <option value="roja"> roja </option>
                    <option value="negra"> negra </option>
                    <option value="verde"> verde </option>
                </select>
                <input type="hidden" name="dirIP" value="localhost">
                <input type="hidden" name="nomBD" value="cerveceria">
                <input type="submit" value ="Enviar">
            </p>
        </form>
    </body>
</html>
```



procesar.jsp

```
<%@ page session="false" %>
<jsp:useBean id="cerveceriaBean" scope="page" class="prog2.cerveceria.CerveceriaBean" />
<%
    if (cerveceriaBean.esCargarOK(request.getParameter("dirIP"),
        request.getParameter("nomBD"),
        request.getParameter("color"))) {
        request.setAttribute("cervezas", cerveceriaBean.getResultado());
        request.setAttribute("colorElegido", request.getParameter("color"));
        request.getRequestDispatcher("resultados.jsp").forward(request, response);
    } else {
        request.setAttribute("mensajeError", cerveceriaBean.getMensajeError());
        request.getRequestDispatcher("error.jsp").forward(request, response);
    }
%>
```

resultados.jsp

```
<%@page import="java.util.ArrayList"%>
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <%
            if (request.getAttribute("cervezas") == null) {
                request.setAttribute("mensajeError",
                    "Lamentablemente, no hay ninguna cerveza "
                    + request.getAttribute("colorElegido")
                    + " en nuestra base de datos.");
                request.getRequestDispatcher("error.jsp").forward(request, response);
            } else {
        %>
        <h1>
            Marcas de cerveza
            <% out.print(request.getAttribute("colorElegido")); %>
            recomendadas
        </h1>
        <%
            ArrayList<prog2.cerveceria.CervezaBean> cervezas =
                (ArrayList<prog2.cerveceria.CervezaBean>) request.getAttribute("cervezas");
            for (int i = 0; i < cervezas.size(); i++) {
        %>
        <p>
            Pruebe: <% out.print(cervezas.get(i).getMarca()); %><br>
            ">
        </p>
        <%
            }
        %}
        <%
    }
</body>
</html>
```



error.jsp

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <h1>ERROR</h1>
        <p>
            <% out.println(request.getAttribute("mensajeError")); %>
        </p>
    </body>
</html>
```

Como se puede observar en las páginas *JSP* anteriores, la arquitectura *JSP* Modelo-1 no logra una completa separación entre la presentación y la lógica (a pesar de mantener gran parte de ésta en los *JavaBeans* que se muestran a continuación), ya que requiere que se coloque bastante código escrito en Java dentro del *HTML*. Estos *scriptlets* hacen que la aplicación sea difícil de mantener y de cambiar.

CervezaBean.java

```
package prog2.cerveceria;

public class CervezaBean implements java.io.Serializable {

    private String marca;
    private String color;
    private String foto;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getFoto() {
        return foto;
    }

    public void setFoto(String foto) {
        this.foto = foto;
    }
}
```



CerveceriaBean.java

```
package prog2.cerveceria;

import java.sql.*;
import java.util.ArrayList;

public class CerveceriaBean implements java.io.Serializable {

    private String mensajeError;
    private ArrayList<CervezaBean> resultado;

    public ArrayList<CervezaBean> getResultado() {
        return resultado;
    }

    public String getMensajeError() {
        return mensajeError;
    }

    public boolean esCargarOK(String ip, String bd, String c) {
        String jdbcDriver = "com.mysql.cj.jdbc.Driver";
        String url = "jdbc:mysql://" + ip + "/" + bd;
        boolean isOK = false;
        try {
            Class.forName(jdbcDriver);
            Connection con = DriverManager.getConnection(url, "", "");
            Statement stmt = con.createStatement();
            stmt.execute("SELECT marca, color, foto FROM cervezas WHERE color='" + c + "'");
            ResultSet rs = stmt.getResultSet();
            if (rs.isBeforeFirst()) {
                resultado = new ArrayList<CervezaBean>();
                while (rs.next()) {
                    CervezaBean cerveza = new CervezaBean();
                    cerveza.setMarca(rs.getString(1));
                    cerveza.setColor(rs.getString(2));
                    cerveza.setFoto(rs.getString(3));
                    resultado.add(cerveza);
                }
                rs.close();
            }
            stmt.close();
            con.close();
            isOK = true;
        } catch (ClassNotFoundException ex) {
            mensajeError = ex.getMessage();
        } catch (SQLException ex) {
            mensajeError = ex.getMessage();
        }
        return isOK;
    }
}
```

La arquitectura más adecuada para desarrollar aplicaciones fáciles de mantener y de cambiar es la *JSP Modelo-2*, la cual se basa en el patrón *MVC*, utilizando *servlets* como controladores, *JavaBeans* y *POJOs (Plain Old Java Objects)* como modelos y *JSP* como vistas. En estas últimas se utiliza el *EL (Expression Language)* en lugar de *scriptlets*, con lo cual se elimina la necesidad de utilizar Java para cuestiones de presentación, limitándose su uso a la lógica de los controladores y los modelos.



index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <h1>Menú de cervezas</h1>
        <form method="post" action="Controlador">
            <p>
                Seleccione el color:
                <select name="color" size="1">
                    <option value="rubia"> rubia </option>
                    <option value="roja"> roja </option>
                    <option value="negra"> negra </option>
                    <option value="verde"> verde </option>
                </select>
                <input type="hidden" name="dirIP" value="localhost">
                <input type="hidden" name="nomBD" value="cerveceria">
                <input type="submit" value ="Enviar">
            </p>
        </form>
    </body>
</html>
```

Controlador.java

```
package prog2.cerveceria;

import java.awt.event.*;
import java.util.logging.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import java.io.*;

@WebServlet(name = "Controlador", urlPatterns = {"/Controlador"})
public class Controlador extends HttpServlet {

    HttpServletRequest request;
    HttpServletResponse response;

    @Override
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        this.request = request;
        this.response = response;
        String ip = request.getParameter("dirIP");
        String bd = request.getParameter("nomBD");
        String color = request.getParameter("color");
```



```
Modelo m = new Modelo(ip, bd);
m.addExceptionListener(new ExceptionListener());
m.consultarPorColor(color);
request.setAttribute("cervezas", m.getResultado());
request.setAttribute("colorElegido", color);
RequestDispatcher vista = request.getRequestDispatcher("vistaResultados.jsp");
vista.forward(request, response);
}

private class ExceptionListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent event) {
        String exception = event.getActionCommand();
        request.setAttribute("mensajeError", exception);
        RequestDispatcher vista = request.getRequestDispatcher("vistaError.jsp");
        try {
            vista.forward(request, response);
        } catch (ServletException | IOException ex) {
            Logger.getLogger(Controlador.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

CervezaBean.java

```
package prog2.cerveceria;

public class CervezaBean implements java.io.Serializable {

    private String marca;
    private String color;
    private String foto;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getFoto() {
        return foto;
    }

    public void setFoto(String foto) {
        this.foto = foto;
    }
}
```



Modelo.java

```
package prog2.cerveceria;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.*;
import java.util.ArrayList;

public class Modelo {

    private String jdbcDriver;
    private String dbName;
    private String urlRoot;
    private ArrayList<CervezaBean> resultado;
    private ActionListener listener;

    public Modelo(String url, String dbName) {
        jdbcDriver = "com.mysql.cj.jdbc.Driver"; urlRoot = "jdbc:mysql://" + url + "/";
        this.dbName = dbName; listener = null; resultado = new ArrayList<>();
        try {
            Class.forName(jdbcDriver);
        } catch (ClassNotFoundException e) {
            reportException(e.getMessage());
        }
    }

    public void consultarPorColor(String c) {
        try {
            Connection con = DriverManager.getConnection(urlRoot + dbName, "", "");
            Statement stmt = con.createStatement();
            stmt.execute("SELECT marca, color, foto FROM cervezas WHERE color='" + c + "'");
            ResultSet rs = stmt.getResultSet();
            while (rs.next()) {
                CervezaBean cerveza = new CervezaBean();
                cerveza.setMarca(rs.getString(1)); cerveza.setColor(rs.getString(2));
                cerveza.setFoto(rs.getString(3)); resultado.add(cerveza);
            }
            con.close();
        } catch (SQLException e) {
            reportException(e.getMessage());
        }
    }

    public ArrayList<CervezaBean> getResultado() {
        return resultado;
    }

    private void reportException(String exception) {
        if (listener != null) {
            ActionEvent evt = new ActionEvent(this, 0, exception);
            listener.actionPerformed(evt);
        }
    }

    public void addExceptionListener(ActionListener listener) {
        this.listener = listener;
    }
}
```



vistaError.jsp

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <h1>ERROR</h1>
        <p>
            ${mensajeError}
        </p>
    </body>
</html>
```

vistaResultados.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="css/estilo.css">
        <link rel="icon" type="image/x-icon" href="img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="img/favicon.ico">
        <title>Menú de cervezas</title>
    </head>
    <body>
        <c:choose>
            <c:when test="${empty cervezas}">
                <h1>ERROR</h1>
                <p>
                    Lamentablemente, no hay ninguna cerveza ${colorElegido} en nuestra base de datos.
                </p>
            </c:when>
            <c:otherwise>
                <h1>
                    Marcas de cerveza ${colorElegido} recomendadas
                </h1>
                <c:forEach var="cerveza" items="${cervezas}" >
                    <p>
                        Pruebe: ${cerveza.marca} <br>
                        
                    </p>
                </c:forEach>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```

Como se puede observar en las dos últimas vistas, el uso de *EL* permitió eliminar la necesidad de utilizar *scriptlets*.



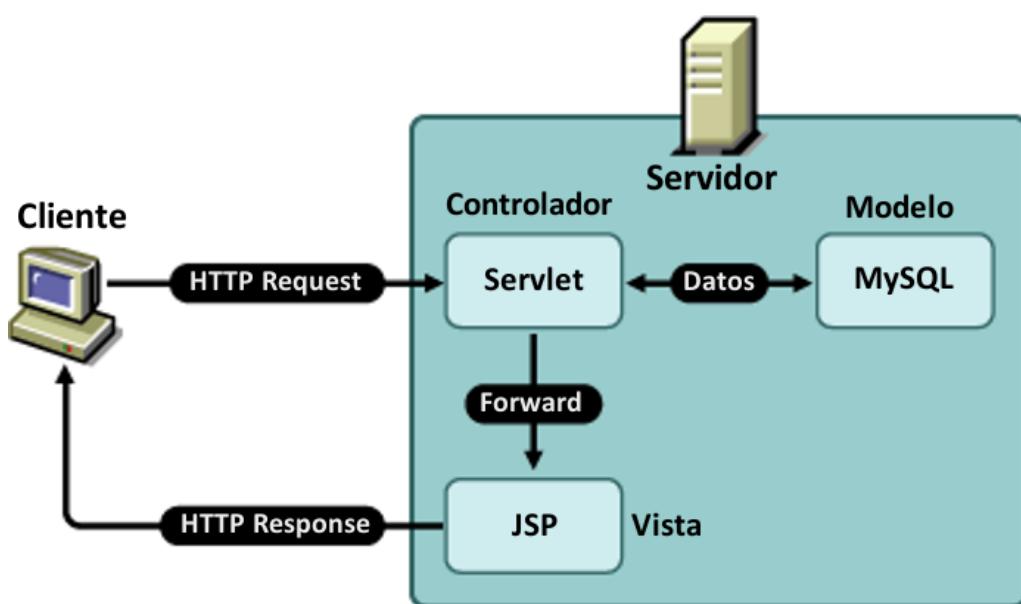
En el archivo anterior (`vistaResultados.jsp`), se obtienen del objeto `request` (usando `EL`) el atributo `colorElegido` (que el `servlet` ya había usado para consultar al modelo) y el atributo `cervezas`, un `ArrayList` con los objetos de la clase `CervezaBean` que el modelo obtuvo al realizar la consulta a la base de datos.

La etiqueta `<c:choose>` permite encerrar una o más etiquetas `<c:when>` con condiciones y cero o una etiqueta `<c:otherwise>`, para generar la página según los resultados de evaluar las condiciones.

La etiqueta `<c:forEach>` permite recorrer una colección (en este caso, `cervezas` es un `ArrayList`). Como la clase `CervezaBean`, a la que pertenecen los objetos extraídos de la colección en la variable `cerveza`, es un *JavaBean* (y, por lo tanto, sus atributos son accesibles mediante *getters* y *setters* que siguen una convención de nomenclatura estándar), sus atributos pueden obtenerse mediante el operador punto (.), por ejemplo, así:

`${cerveza.foto}`

La siguiente figura resume la arquitectura *JSP* Modelo-2, de la cual derivan varios de los *frameworks* que veremos en la siguiente unidad.



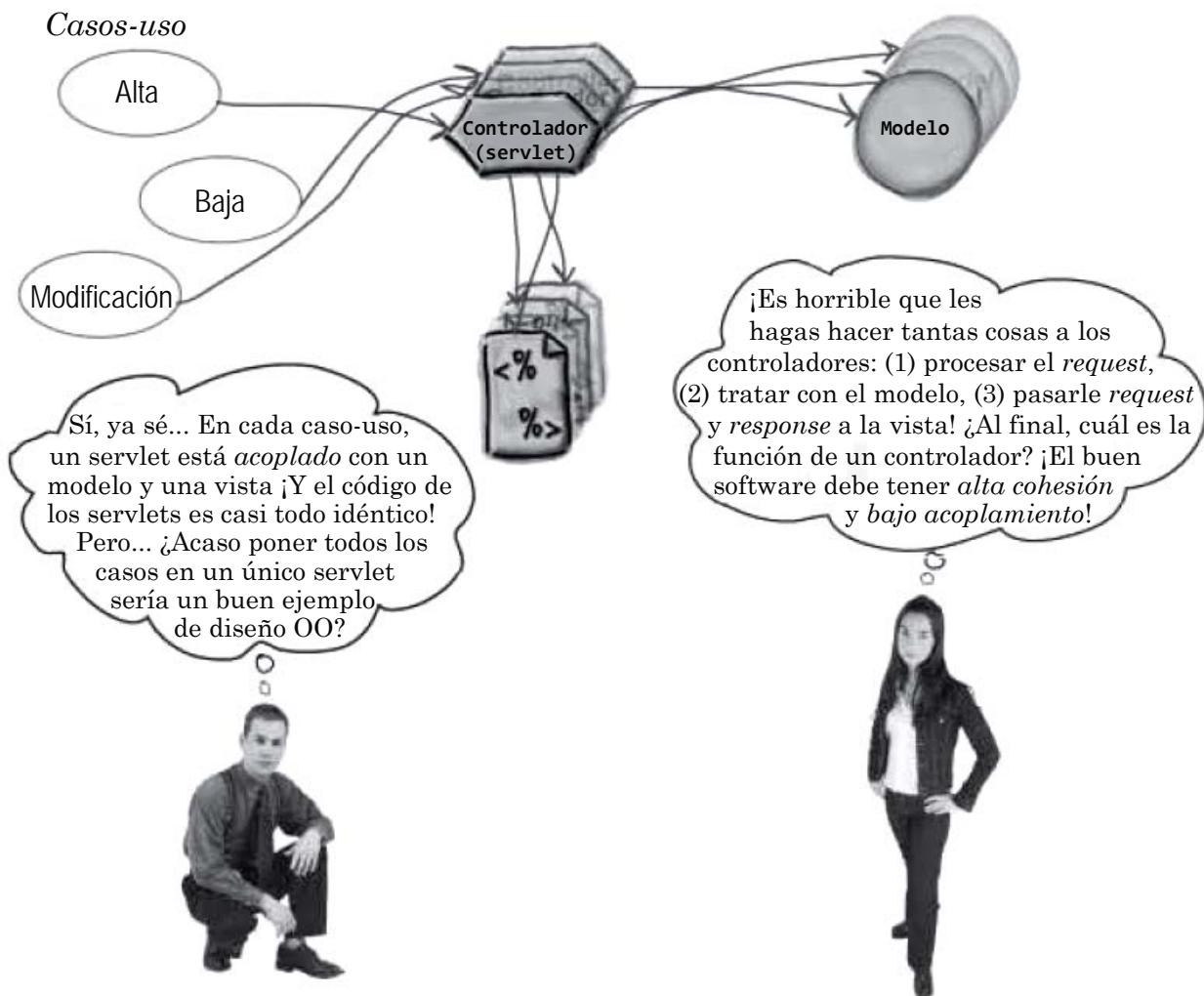
CUESTIONARIO FINAL DE LA UNIDAD

- 1) En los ejemplos de esta unidad, cada vez que se debe acceder a la base de datos se obtiene una nueva conexión. ¿Le parece correcto trabajar así? (Investigue `javax.sql.DataSource`)
.....
- 2) En una aplicación formada por múltiples páginas *JSP* con formularios, ¿qué diseño le parece mejor: uno que utilice un *servlet* por cada formulario o uno en el que todos los formularios compartan un único *servlet*?
.....



Unidad 12 Frameworks

La siguiente figura representa la cuestión mencionada en la última pregunta de la unidad anterior:



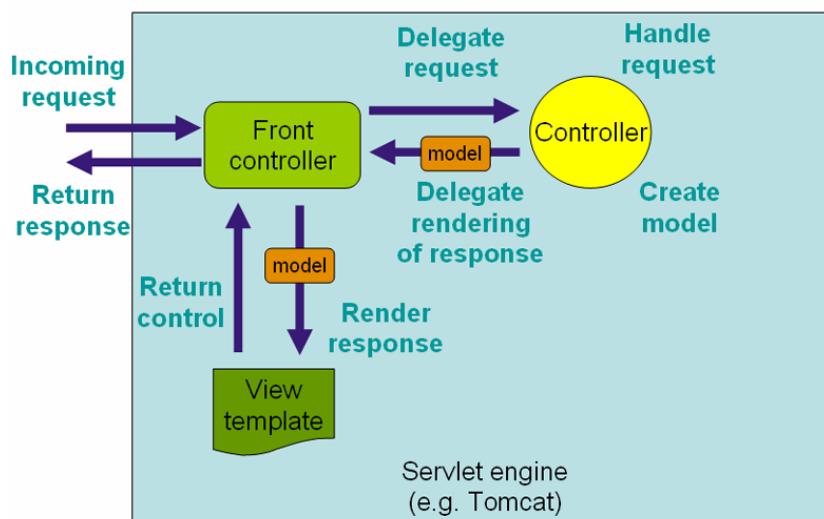
En el código que se muestra a continuación, pueden observarse las tareas que tradicionalmente realiza un controlador implementado como un servlet:

```
public class Controlador extends HttpServlet {  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException {  
        String ip = request.getParameter("dirIP");  
        String bd = request.getParameter("nomBD");  
        String color = request.getParameter("color"); } ①  
        Modelo m = new Modelo(ip, bd);  
        m.addExceptionListener(new ExceptionListener()); } ②  
        m.consultarPorColor(color);  
        request.setAttribute("cervezas", m.getResultado());  
        request.setAttribute("colorElegido", color);  
        RequestDispatcher vista = request.getRequestDispatcher("vistaResultados.jsp"); } ③  
        vista.forward(request, response);  
    }  
}
```

Un mejor diseño consistiría en usar **un único servlet** (*Front controller*) que:

1. delegue el procesamiento del *request* a otro componente (*Controller*);
2. no esté acoplado a un modelo específico;
3. no esté acoplado a una vista específica.

En este diseño se basa *Spring Web MVC*. Este *framework*¹ actúa como una enorme *Factory* que instancia los objetos de aquellas clases decoradas mediante ciertas *annotations*, porque ofrece **inversión de control** (*Inversion of Control* o *IoC*), y usa un *front controller* de la clase `org.springframework.web.servlet.DispatcherServlet`.



El funcionamiento básico de *Spring Web MVC* es así:

1. Cuando se arranca la aplicación, se detecta la clase que implementa la interfaz `WebApplicationInitializer` y se la utiliza para la inicialización del contenedor de *servlets*, ejecutando el método `onStartup`, donde se instancian dos objetos: uno de la clase `AnnotationConfigWebApplicationContext` (encargado de registrar una clase de configuración que implemente la interfaz `WebMvcConfigurer`) y otro de la clase `DispatcherServlet` (el *front controller*).
2. A través de la *annotation* `@ComponentScan`, en la clase de configuración (decorada con la *annotation* `@Configuration`) se indica el paquete donde buscar clases adicionales "anotadas" mediante `@Controller`, `@Repository`, etc. para ser registradas por el *framework*, y se instancian distintos objetos (*beans* de Spring), por ejemplo un `ViewResolver` de la clase `InternalResourceViewResolver` necesario para determinar las vistas. Estos objetos, administrados por el **contenedor de IoC** de Spring, no son creados por los objetos que dependen de ellos (por ejemplo, el modelo no es instanciado por el controlador), sino que son suministrados a éstos mediante **inyección de dependencias** (*Dependency Injection* o *DI*), lo que se suele indicar a través de la *annotation* `@Autowired`.

¹ Un *framework* es una colección de interfaces y clases diseñadas para trabajar juntas en el tratamiento de un tipo de problema en particular. En el caso de *Spring Web MVC*, el problema en cuestión son las aplicaciones web. El principal objetivo de un *web application framework* (*WAF*) es facilitar a los programadores el desarrollo y mantenimiento de aplicaciones web complejas.



3. Al recibir una petición (*request*), el **DispatcherServlet** (el *front controller*) averigua a cuál de los *Controllers* registrados hay que llamar para servirla.
4. A continuación, se identifica (por su *annotation*) y se llama al método de ese *Controller* que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al **DispatcherServlet**, encapsulados en una instancia de **Model**. Además, se devolverá el nombre lógico de la vista a mostrar (normalmente como un **String**).
5. El **InternalResourceViewResolver** obtiene el nombre real de la vista, correspondiente al nombre lógico del paso anterior.
6. Finalmente, se genera la respuesta (*response*) completando el contenido del archivo de la vista con los datos encapsulados en la instancia de **Model**, y el **DispatcherServlet** la retorna.

Veamos ahora cómo volver a construir con *NetBeans* la aplicación presentada en la página 204 para que utilice *Spring Web MVC*.

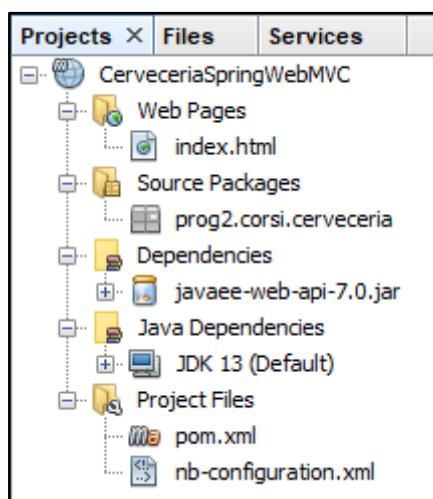
Paso 1: Creación del proyecto con Maven en NetBeans

Simplemente hay que ir al siguiente asistente para la creación del proyecto:

File → New Project → Java with Maven → Web Application

Allí hay que completar los datos solicitados:

Name and Location
Project Name: <i>CerveceriaSpringWebMVC</i>
Project location: [no cambiarla]
Group Id: <i>prog2.corsi</i>
Version: 1.0
Package: <i>prog2.corsi.cerveceria</i>
Settings
Web Server: <No server selected>
Java EE Version: Java EE 7 Web



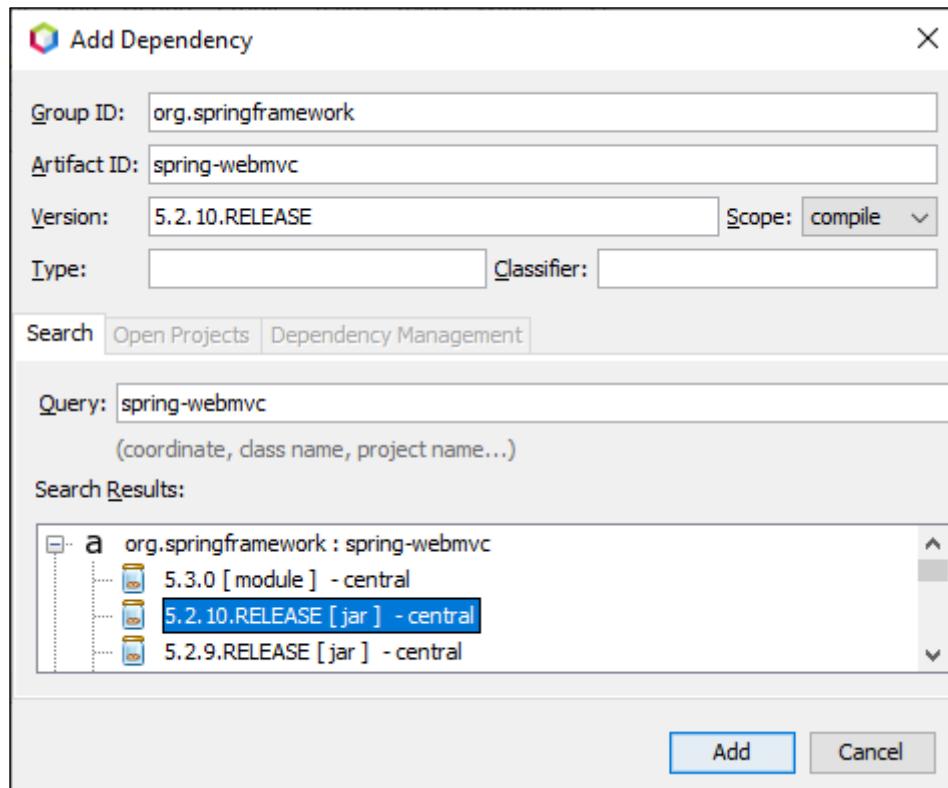


Paso 2: Borrar index.html y agregar spring-webmvc en Dependencies

Simplemente hay que ir al siguiente asistente haciendo clic derecho en el proyecto:

Dependencies → Add dependency...

Allí se debe escribir `spring-webmvc` en *Query* y elegir la versión que se desea agregar:



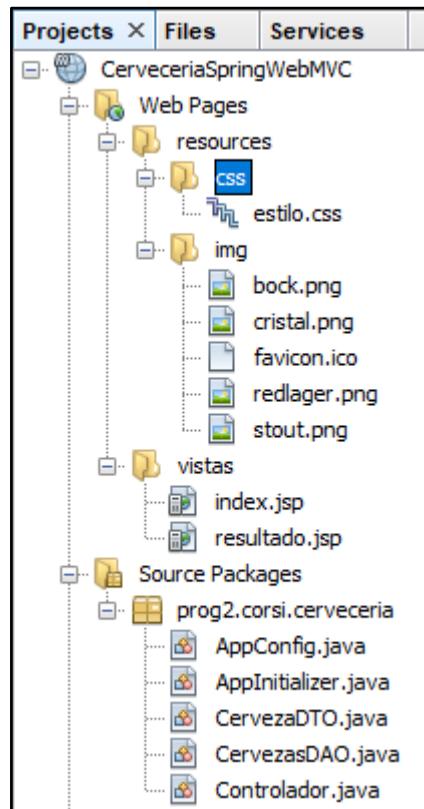
En lugar de usar el asistente de NetBeans, también puede colocarse la siguiente dependencia en el archivo `pom.xml`:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version></version>
    <type>jar</type>
</dependency>
```

En el caso anterior, la versión está vacía para que aparezca un menú emergente al situar el cursor entre las etiquetas XML `<version>` y `</version>`, en el que deberá elegirse la versión *RELEASE* más reciente que esté disponible en los repositorios de Maven.



Paso 3: Crear las carpetas y agregar en ellas los archivos del proyecto



Si bien los archivos de recursos (imágenes, estilos) son los mismos que los del proyecto original, en la cantidad de vistas habrá un cambio, pues estas serán ahora solo las dos siguientes:

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <link rel="stylesheet" type="text/css" href="/resources/css/estilo.css">
        <link rel="icon" type="image/x-icon" href="/resources/img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="/resources/img/favicon.ico">
        <title>Cervecería Spring Web MVC</title>
    </head>
    <body>
        <h1>Elija un color:</h1>
        <form method="post" action="/consulta">
            <p>
                <select name="color">
                    <c:forEach var="color" items="${colores}" >
                        <option value="${color}">${color}</option>
                    </c:forEach>
                </select>
                <input type="submit" value="Enviar">
            </p>
        </form>
    </body>
</html>
```



resultado.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" href="/resources/css/estilo.css">
        <link rel="icon" type="image/x-icon" href="/resources/img/favicon.ico">
        <link rel="shortcut icon" type="image/x-icon" href="/resources/img/favicon.ico">
        <title>Cervecería Spring Web MVC</title>
    </head>
    <body>
        <h1>
            Marcas de cerveza ${colorElegido} recomendadas
        </h1>
        <c:forEach var="cerveza" items="${cervezas}" >
            <p>
                Pruebe: ${cerveza.marca} <br>
                
            </p>
        </c:forEach>
    </body>
</html>
```

La clase que implementa la interfaz `WebApplicationInitializer` y que se utiliza para la inicialización del contenedor web, ejecutando el método `onStartup`, donde se instancian dos objetos: uno de la clase `AnnotationConfigWebApplicationContext` (encargado de registrar una clase de configuración que implemente la interfaz `WebMvcConfigurer`) y otro de la clase `DispatcherServlet` (el *front controller*) es la siguiente:

AppInitializer.java

```
package prog2.corsi.cerveceria;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import org.springframework.web.servlet.DispatcherServlet;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext context =
            new AnnotationConfigWebApplicationContext();
        context.register(AppConfig.class);
        context.setServletContext(servletContext);

        ServletRegistration.Dynamic servlet =
            servletContext.addServlet("dispatcher", new DispatcherServlet(context));
        servlet.setLoadOnStartup(1);
        servlet.addMapping("/");
    }
}
```



La clase de configuración (que implementa la interfaz `WebMvcConfigurer`) decorada con la `annotation @Configuration` y en la cual, a través de la `annotation @ComponentScan`, se indica el paquete donde buscar clases adicionales para ser registradas por el *framework*, es la siguiente:

AppConfig.java

```
package prog2.corsi.cerveceria;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@ComponentScan("prog2.corsi.cerveceria")
@EnableWebMvc
public class AppConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(final ResourceHandlerRegistry reg) {
        reg.addResourceHandler("/resources/**").addResourceLocations("/resources/");
    }

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver vr = new InternalResourceViewResolver();
        vr.setViewClass(JstlView.class);
        vr.setPrefix("/vistas/");
        vr.setSuffix(".jsp");
        return vr;
    }

    @Bean(name = "dbName")
    public String getDBName() { return "cerveceria"; }

    @Bean(name = "dbURL")
    public String getDBURL() { return "127.0.0.1"; }

    @Bean(name = "dbUser")
    public String getDBUser() { return ""; }

    @Bean(name = "dbPswd")
    public String getDBPswd() { return ""; }

}
```

Los *beans* decorados en la clase de configuración mediante las `annotations @Bean` podrán ser "inyectados" en las clases que dependan de ellos.



La clase que define el controlador donde se procesan las distintas peticiones (*requests*) HTTP (dado que se trata de un controlador *multi-action*) y a la cual se le "inyecta" un DAO (mediante la *annotation* `@Autowired`) es la siguiente:

Controlador.java

```
package prog2.corsi.cerveceria;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class Controlador {

    @Autowired
    private CervezasDAO cervezasDAO;

    @GetMapping("/")
    public String mostrarInicio(Model model) {
        model.addAttribute("colores", cervezasDAO.getColores());
        return "index";
    }

    @PostMapping("/consulta")
    public String mostrarRespuesta(
        Model model,
        @RequestParam(value = "color", required = true) String color) {
        model.addAttribute("colorElegido", color);
        model.addAttribute("cervezas", cervezasDAO.getCervezasByColor(color));
        return "resultado";
    }
}
```

La clase decorada mediante la *annotation* `@Repository` y que puede ser "inyectada" en las clases que dependan de ella para proporcionarles los métodos de acceso al repositorio de datos, es la siguiente:

CervezasDAO.java

```
package prog2.corsi.cerveceria;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Repository;
```



```
@Repository
public class CervezasDAO {

    private final String dbFullURL;
    private final String dbUser;
    private final String dbPswd;

    @Autowired
    public CervezasDAO(
        @Qualifier("dbName") String dbName,
        @Qualifier("dbURL") String dbURL,
        @Qualifier("dbUser") String dbUser,
        @Qualifier("dbPswd") String dbPswd) {
        dbFullURL = "jdbc:mysql://" + dbURL + "/" + dbName;
        this.dbUser = dbUser;
        this.dbPswd = dbPswd;
    }

    public List<CervezaDTO> getCervezasByColor(String color) {
        ArrayList<CervezaDTO> resultado = new ArrayList<>();
        try {
            Connection con = DriverManager.getConnection(dbFullURL, dbUser, dbPswd);
            Statement stmt = con.createStatement();
            stmt.execute("SELECT marca, color, foto FROM cervezas WHERE color='"
                + color + "'");
            ResultSet rs = stmt.getResultSet();
            while (rs.next()) {
                CervezaDTO cerveza = new CervezaDTO();
                cerveza.setMarca(rs.getString(1));
                cerveza.setColor(rs.getString(2));
                cerveza.setFoto(rs.getString(3));
                resultado.add(cerveza);
            }
            con.close();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return resultado;
    }

    public List<String> getColores() {
        ArrayList<String> resultado = new ArrayList<>();
        try {
            Connection con = DriverManager.getConnection(dbFullURL, dbUser, dbPswd);
            Statement stmt = con.createStatement();
            stmt.execute("SELECT DISTINCT color FROM cervezas;");
            ResultSet rs = stmt.getResultSet();
            while (rs.next()) {
                resultado.add(rs.getString(1));
            }
            con.close();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return resultado;
    }
}
```



En la clase anterior puede observarse cómo la existencia de múltiples objetos de la clase `String` en el contenedor de IoC de Spring hace que, al usar la opción `@Autowired`, también sea necesario utilizar la *annotation* `@Qualifier` para eliminar la ambigüedad y definir cuál de ellos se "inyectará".

La quinta y última clase de este proyecto es la siguiente:

CervezaDTO.java

```
package prog2.corsi.cerveceria;

public class CervezaDTO implements java.io.Serializable {
    private String marca;
    private String color;
    private String foto;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getFoto() {
        return foto;
    }

    public void setFoto(String foto) {
        this.foto = foto;
    }
}
```

Como puede observarse, esta clase es un *JavaBean*, por lo que puede aprovecharse la versión del proyecto original prácticamente sin cambios.



La aplicación recién podrá correr (en `localhost:8080`) cuando se haya instalado un Web Server (contenedor de *servlets*). Por su practicidad, la opción sugerida aquí es *Eclipse Jetty*, que se puede hacer funcionar agregando su *plugin* en el archivo `pom.xml` (en *plugins*) y configurando las acciones necesarias, como se indicó en las páginas 204-205.

Además, para poder usar la base de datos, en *Dependencies* debe agregarse el *driver JDBC* para MySQL. Esto puede hacerse colocando la dependencia en el archivo `pom.xml` o mediante el asistente que se abre en *Dependencies* → *Add Dependency*, como se vio en la página 207.

OBSERVACIÓN

La estructura presentada aquí es minimalista y solamente adecuada para proyectos de pequeño tamaño. En proyectos de mayor complejidad, se recomienda no utilizar un controlador *multi-action* sino un controlador específico para cada caso-uso, decorado mediante la *annotation @Controller*, lo que aumenta la cohesión de las clases.

Asimismo, se recomienda utilizar paquetes para mantener las clases separadas y mejor organizadas.

Por último, el *Principio de inversión de dependencias* (la "D" en S.O.L.I.D.) establece que las clases deben depender de abstracciones, no de implementaciones concretas, por lo que se recomienda usar interfaces para la inyección de dependencias, en lugar de clases concretas.

EJERCICIO N° 7

Utilizando *Spring Web MVC*, transforme la aplicación presentada en la pág. 193 en una aplicación web:





Como vimos, un *framework* es una colección de interfaces y clases diseñadas para trabajar juntas en el tratamiento de un tipo de problema en particular. No debe confundirse con una API (*Application Programming Interface*), un SDK (*Software Development Kit*), una biblioteca (*Library*) ni un motor (*Engine*), los cuales son términos relacionados, aunque conceptualmente diferentes:

Mis anotaciones: Comparación entre Frameworks y otras herramientas

Framework:

.....
.....
.....
.....

Library (Ej: *Lightweight Java Game Library* [lwjgl.org]):

.....
.....
.....
.....

SDK (Ej: *GWT* [gwtproject.org]):

.....
.....
.....
.....

API (Ej: *YouTube Data API* [developers.google.com/youtube/v3]):

.....
.....
.....
.....

Engine (Ej: *Apache Spark* [spark.apache.org]):

.....
.....
.....
.....



Los *frameworks* también difieren en sus objetivos. A continuación, se listan 40 *frameworks* para Java.

Java Frameworks - Top 40

Sitio Web	Framework	¿Para qué sirve?
ace.apache.org	Apache ACE	Distribución de software. Permite entregar actualizaciones y nuevos componentes. Facilita el montaje de un entorno automatizado de desarrollo, QA (<i>Quality Assurance</i>), prueba y producción.
asm.ow2.org	ASM	Manipulación y análisis de bytecode. Permite modificar clases existentes o generar clases de forma dinámica, directamente en binario.
audit4j.org	Audit4j	Auditoría. Registro seguro y preservable de eventos en servidores, aplicaciones y bases de datos. A diferencia de un <i>log</i> , permite registrar el actor, la acción y el origen.
camel.apache.org	Apache Camel	Integración. Permite implementar cualquiera de los 65 EIP (<i>Enterprise Integration Patterns</i>) estándar en pocas líneas de código. Soporta HTTP, REST, JMS y <i>Web Services</i> , entre otros.
castor-data-binding.github.io/castor	Castor	Enlace de datos (Data binding). Permite pasar un objeto a XML (<i>marshalling</i>) y viceversa (<i>unmarshalling</i>). Mapeo objeto-relacional (ORM). Permite la persistencia de objetos como tablas en bases de datos relacionales.
cayenne.apache.org	Apache Cayenne	Mapeo objeto-relacional (ORM). Permite la persistencia de objetos como tablas en bases de datos relacionales, incluso remotamente mediante <i>Web Services</i> . Su <i>GUI</i> hace innecesaria la configuración mediante XML o <i>annotations</i> .
cocoon.apache.org	Apache Cocoon	Publicación basada en XML. Aplicando los conceptos de Separación de Intereses (<i>Separation of Concerns</i>) y Desarrollo Basado en Componentes (<i>Component-based Development</i>), permite construir soluciones a través de la unión de componentes en un <i>pipeline</i> , sin programación.
commons.apache.org/proper/commons-bsf	Apache BSF	Ejecución de scripts. Permite que las aplicaciones desarrolladas en Java usen <i>scripts</i> escritos en lenguajes como Javascript, Python, Ruby o Prolog. También facilita el acceso a objetos de Java desde estos <i>scripts</i> .
cxfr.apache.org	Apache CXF	Desarrollo de servicios. Utilizando APIs de programación <i>front end</i> como JAX-WS y JAX-RS, permite desarrollar servicios que se comuniquen mediante SOAP, XML/HTTP, HTTP <i>RESTful</i> o CORBA a través de HTTP, JMS or JBI.
gora.apache.org	Apache Gora	Mapeo objeto-almacén de datos (OTD: Object-to-Datastore). Provee un modelo de datos <i>in-memory</i> que permite desacoplar las aplicaciones <i>Big Data</i> de la capa de persistencia, donde pueden usarse tanto bases de datos NoSQL como relacionales, e incluso archivos.
github.com/google/guice	Guice	Inyección de dependencias. Permite escribir código modular y fácilmente testeable. Desarrollado por Google.
hadoop.apache.org	Apache Hadoop	Almacenamiento y procesamiento de grandes volúmenes de datos (Big Data). Permite hacerlo en forma distribuida mediante <i>clusters</i> de computadoras y el modelo de programación <i>MapReduce</i> .
helix.apache.org	Apache Helix	Administración de clusters. Permite gestionar recursos particionados, replicados y distribuidos alojados en un <i>cluster</i> . Automatiza la reasignación de recursos ante fallas en nodos, la expansión del <i>cluster</i> y la reconfiguración.



hibernate.org	Hibernate	Mapeo objeto-relacional (ORM). Permite la persistencia de objetos como tablas en bases de datos relacionales. Mapeo objeto-grid (OGM). Permite la persistencia de objetos en bases de datos NoSQL.
javaserverfaces.java.net	Mojarra	Desarrollo de aplicaciones web (component-based). Es la implementación de Oracle del estándar JSF.
jena.apache.org	Apache Jena	Desarrollo de aplicaciones de Web Semántica. Permite procesar datos <i>RDF</i> y también trae incorporados motores de inferencia para razonar sobre ontologías RDFS y OWL.
jersey.java.net	Jersey	Desarrollo de servicios web (APIs). Es la implementación de referencia (de Sun/Oracle) de <i>JAX-RS</i> . Permite desarrollar servicios web <i>RESTful</i> .
junit.org	JUnit	Automatización de pruebas. Permite diseñar y ejecutar pruebas unitarias para verificar que cada uno de los métodos de una clase se comporta de la manera esperada.
libgdx.badlogicgames.com	libGDX	Desarrollo de juegos. A través de una API unificada, permite desarrollar juegos que corren en varias plataformas (Android, iOS, HTML, etc.)
marf.sourceforge.net	MARF	Reconocimiento de audio. Permite usar algoritmos para el procesamiento de sonidos, incluyendo el lenguaje natural.
mina.apache.org	Apache Mina	Desarrollo de aplicaciones de red. Ofrece una API unificada para TCP, UDP, comunicación serial (RS-232), etc., lo que facilita la creación de aplicaciones de alto desempeño y escalabilidad. Es una alternativa a Netty.
mockito.org	Mockito	Automatización de pruebas. Permite crear fácilmente <i>mocks</i> (objetos que sustituyen a otros en pruebas unitarias y que tienen el comportamiento que se espera de ellos).
mybatis.org	MyBatis	Mapeo objeto-sentencia SQL. Permite la persistencia de objetos en bases de datos relacionales. Es posible utilizar todas las funcionalidades del motor de base de datos, como <i>stored procedures</i> , vistas o funcionalidades específicas.
myfaces.apache.org	Apache MyFaces	Desarrollo de aplicaciones web (component-based). Es la implementación del estándar JSF patrocinada por la <i>Apache Software Foundation</i> .
netty.io	Netty	Desarrollo de aplicaciones de red. Ofrece una API unificada para varios protocolos de transporte, lo que permite crear aplicaciones de alto desempeño y fácil mantenimiento. Es el ancestro inmediato de Apache Mina.
neuroph.sourceforge.net	Neuroph	Desarrollo de redes neuronales. Ofrece una colección de clases básicas y un editor con GUI para crear redes neuronales con facilidad.
playframework.com	Play	Desarrollo de aplicaciones web (action-based). Ofrece gran simplicidad, productividad y usabilidad. Fuertemente inspirado en <i>Ruby on Rails</i> y <i>Django</i> (para Ruby y Python, respectivamente), permite realizar la programación tanto en Scala como en Java.



rdf4j.org	Sesame	Desarrollo de aplicaciones de Web Semántica. Permite procesar (crear, analizar, almacenar, razonar y consultar) datos <i>RDF</i> . Ofrece una API que facilita la conexión con las principales soluciones de almacenamiento <i>RDF</i> . Es menos completo que Apache Jena (p. ej., no trabaja con <i>OWL</i>).
restlet.com/projects/restlet-framework	Restlet	Desarrollo de servicios web (APIs). Ofrece un conjunto reutilizable de clases e interfaces que sirven como base para crear APIs <i>RESTful</i> seguras y escalables. Fue el primer <i>framework</i> web <i>RESTful</i> para Java, disponible desde 2005 (antes de la publicación de <i>JAX-RS</i>).
shiro.apache.org	Apache Shiro	Implementación de seguridad en aplicaciones. Permite incorporar fácilmente autenticación, criptografía y gestión de sesiones en aplicaciones de cualquier tipo, desde móviles hasta web y empresariales.
sitemesh.org	SiteMesh	Diseño de páginas web. Basado en el patrón de diseño <i>Decorator</i> , permite una clara separación entre contenido y presentación: después de trabajar con el contenido, se lo “decora” con un <i>look and feel</i> apropiado.
sparkjava.com	Spark	Desarrollo rápido de aplicaciones web. Inspirado en <i>Sinatra</i> (para Ruby), no sigue el patrón MVC. Facilita la creación de aplicaciones web con el mínimo esfuerzo.
spring.io	Spring	Desarrollo de aplicaciones. Está compuesto por módulos: un <i>framework</i> para AOP (<i>Aspect-oriented programming</i>), un contenedor de inversión de control (fue uno de los pioneros) y un <i>framework</i> web MVC <i>action-based</i> , entre otros.
stripesframework.org	Stripes	Desarrollo de aplicaciones web (<i>action-based</i>). Inspirado en <i>Struts</i> , provee soluciones simples a sus principales problemas, por ejemplo, reemplazando por <i>annotations</i> la configuración mediante archivos XML.
struts.apache.org	Apache Struts	Desarrollo de aplicaciones web (<i>action-based</i>). Fue el primer <i>framework</i> web MVC disponible. Poco flexible, sólo era posible configurarlo mediante archivos XML. La segunda versión es un sistema completamente distinto, fácil de extender mediante <i>plugins</i> .
tapestry.apache.org	Apache Tapestry	Desarrollo de aplicaciones web (<i>component-based</i>). Adhiere al paradigma de <i>Convention over Configuration</i> , eliminando la necesidad de la configuración mediante XML. Las UIs se pueden programar declarativamente en TML (<i>Tapestry Markup Language</i>). Incorpora un contenedor de IoC propio para permitir la inyección de dependencias.
tiles.apache.org	Apache Tiles	Diseño de páginas web. Basado en el patrón de diseño <i>Composite</i> , permite definir fragmentos de página (<i>tiles</i>) con los cuales es posible ensamblar páginas completas en tiempo de ejecución.
vaadin.com	Vaadin	Desarrollo de aplicaciones web (<i>component-based</i>). Permite construir fácilmente RIAs (<i>Rich Internet Applications</i>) con Ajax. Las UIs se programan en Java (<i>event-driven programming</i>), pudiendo usar una variedad de <i>wIDGETS</i> . Utiliza GWT para generar las páginas web.
wicket.apache.org	Apache Wicket	Desarrollo de aplicaciones web (<i>component-based</i>). Da énfasis al uso de "puro Java" para la lógica y "puro HTML" para la presentación. Mantiene esta estricta separación al no permitir el uso de estructuras de control o bibliotecas de etiquetas (<i>tag libraries</i>) en las plantillas del <i>front end</i> .
zkoss.org	ZK	Desarrollo de aplicaciones web (<i>component-based</i>). Permite construir fácilmente RIAs (<i>Rich Internet Applications</i>) con Ajax. Las UIs se programan declarativamente en ZUML (<i>ZK User Interface Markup Language</i>). Utiliza jQuery para generar las páginas web.



Mi glosario: Términos que aparecen en las descripciones de los *frameworks* anteriores

IoC, DI, AOP:

.....
.....
.....
.....
.....

Action-based, Component-based:

.....
.....
.....
.....
.....

Big Data, MapReduce, clusters, bases de datos NoSQL:

.....
.....
.....
.....
.....

Web Semántica, RDF, ontologías RDFS y OWL:

.....
.....
.....
.....
.....

Web services, JAX-WS, SOAP, CORBA:

.....
.....
.....
.....
.....

RESTful APIs, JAX-RS:

.....
.....
.....
.....
.....



Bibliografía

Barker, J. (2005): *Beginning Java Objects: From Concepts to Code, 2^a ed.*, Apress

Basham, B. et al. (2008): *Head First. Servlets and JSP, 2^a ed.*, O'Reilly Media

Deitel, P. & Deitel, H. (2008): *Cómo programar en Java, 7^a ed.*, Pearson

----- (2015): *Java. How to program, 10^a ed.*, Pearson

Freeman, E. et al. (2004): *Head First. Design Patterns*. O'Reilly Media

Martin, R. C. (2003): *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall

Sun Microsystems, Inc. (2007): *Java™ Programming Language. SL-275-SE6 StudentGuide, Revision G*

Sznajdleder, P. (2013): *Java a fondo: estudio del lenguaje y desarrollo de aplicaciones, 2^a ed.* Alfaomega