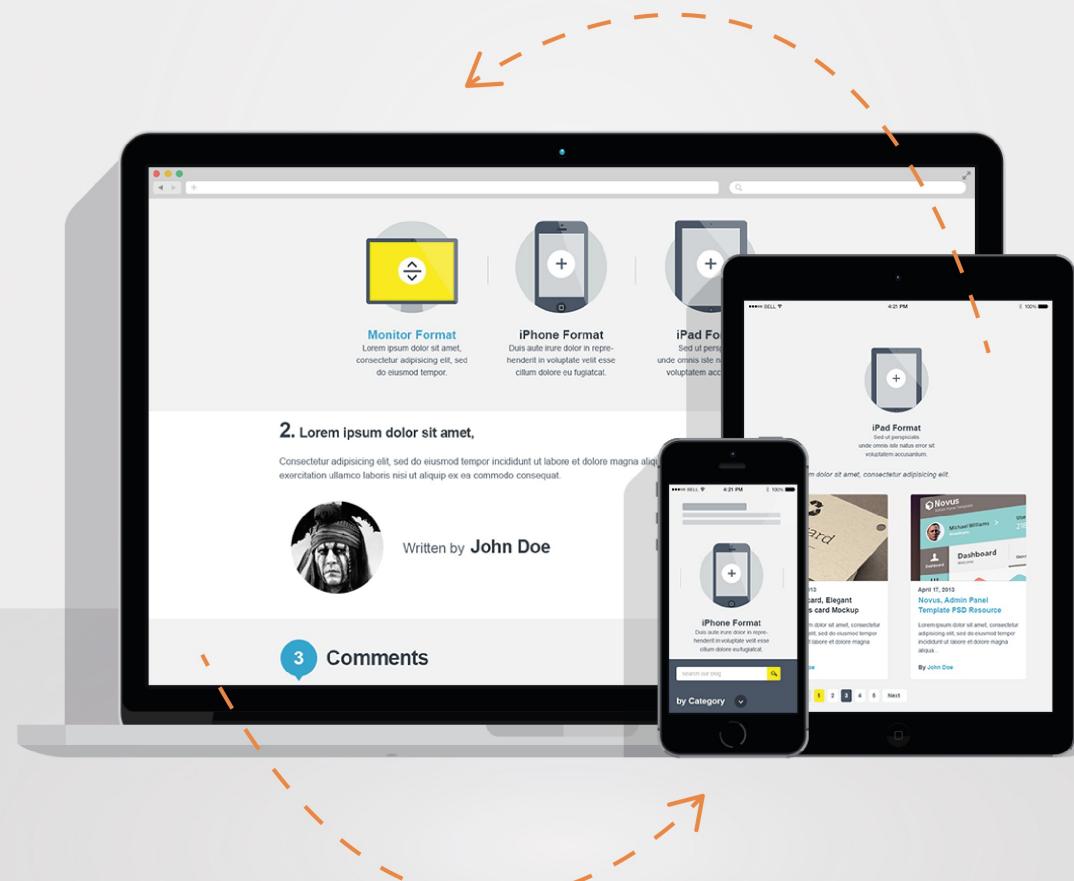


CONSTRUCCIÓN DE LA INTERFAZ GRÁFICA

05



1. Objetivos Pedagógicos

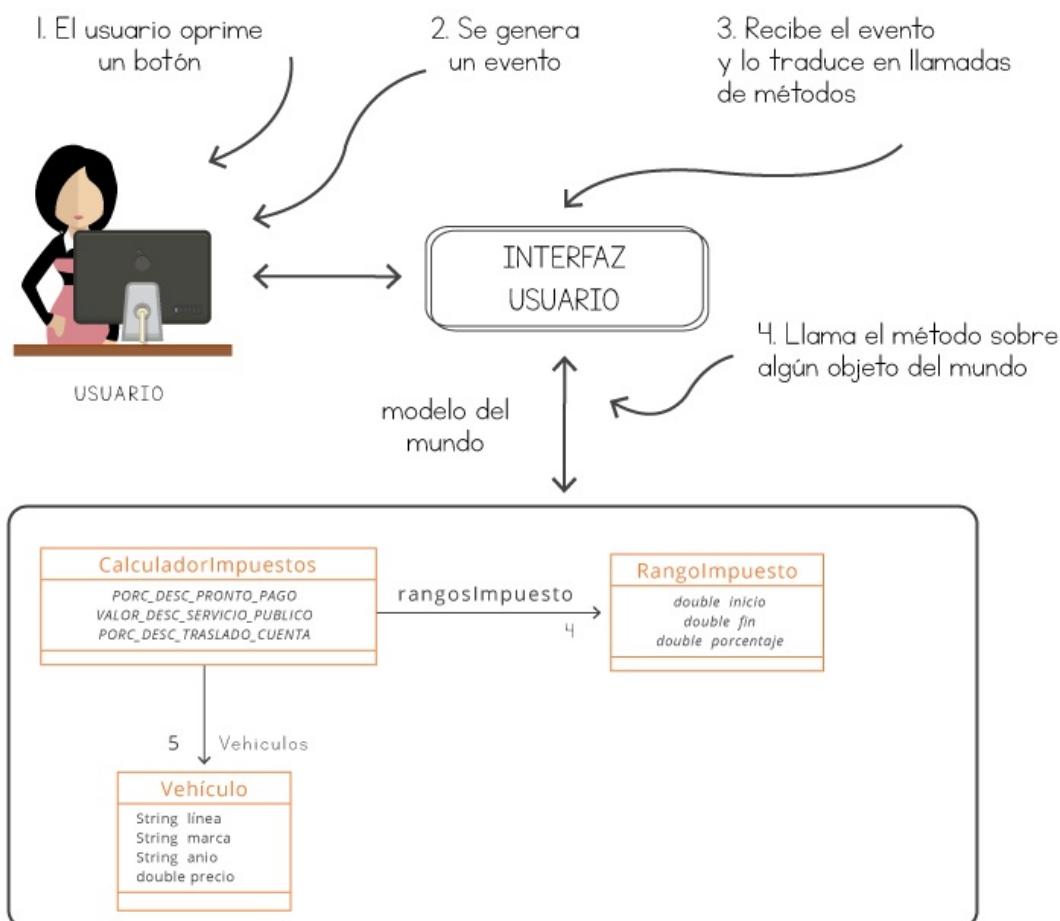
Al final de este nivel el lector será capaz de:

- Explicar la importancia de la [interfaz de usuario](#) dentro de un [programa de computador](#), teniendo en cuenta que es el medio de comunicación entre el usuario y el modelo del mundo.
- Proponer una [arquitectura](#) para un programa simple, repartiendo de manera adecuada las responsabilidades entre la [interfaz de usuario](#), el modelo del mundo y las pruebas unitarias. El lector deberá poder explicar la importancia de mantener separadas las clases de esos tres dominios.
- Construir las clases que implementan una [interfaz de usuario](#) sencilla e integrarlas con las clases que implementan el modelo del mundo del problema.

2. Motivación

La **interfaz de usuario** es el medio de comunicación entre el usuario y el modelo del mundo, tal como se sugiere en la [figura 5.1](#). A través de la interfaz, el usuario expresa las operaciones que desea realizar sobre el modelo del mundo y, por medio de la misma interfaz, el usuario puede apreciar el resultado de sus acciones. Es un medio que permite la comunicación en los dos sentidos. La **interfaz de usuario ideal** es aquella en la que la persona siente que está visualizando e interactuando directamente con los elementos del modelo del mundo, y que esto se hace a través de un proceso sencillo y natural.

Fig. 5.1 La interfaz de usuario como medio de comunicación



Siempre que utilizamos un [programa de computador](#), esperamos que sea agradable y fácil de utilizar. Aunque es difícil dar una definición precisa de lo que significa agradable, hay condiciones mínimas que influyen en esta percepción, que se relacionan con la combinación de colores, la organización de los elementos en las ventanas, los gráficos, los

tipos de letra, etc. La [propiedad](#) de facilidad de uso, por su parte, está más relacionada con el hecho de que los elementos de interacción se comporten de forma intuitiva (por ejemplo, si existe un botón con la [etiqueta](#) "cancelar" se espera que aquello que se está realizando se suspenda al oprimir este botón) y también, con la cantidad de conocimiento que el usuario debe tener para utilizar el programa.

- La interfaz es el medio para que el usuario pueda interactuar con el modelo del mundo.
- Es también una [ventana](#) para que el usuario pueda visualizar el estado del mundo.
- La interfaz debe ser amigable y fácil de usar, de manera que el usuario se sienta cómodo utilizando el programa y no cometa errores debido a cuestiones que no son claras para él.
- La interfaz debe ser capaz de interpretar las acciones de los usuarios (expresadas como eventos) y llamar los métodos que ejecutan lo que él pide.

La razón de darle importancia a este aspecto es muy sencilla: si el usuario no se siente cómodo con el programa, no lo va a utilizar o lo va a utilizar de manera incorrecta. En la mayoría de los proyectos, se dedica igual cantidad de esfuerzo a la construcción de la interfaz que al desarrollo del modelo del mundo.

Hay dos aspectos de gran importancia en el [diseño](#) de la interfaz: el primer aspecto tiene que ver con el [diseño](#) funcional y gráfico (los colores que se deben usar, la distribución de los elementos gráficos, etc.). En eso debe participar la mayoría de las veces un diseñador gráfico y es un tema que está por fuera del alcance de este libro.

El segundo aspecto es la parte de la organización de las clases que van a conformar la interfaz y, de nuevo, este aspecto tiene que ver con la [asignación](#) de responsabilidades que discutimos en el nivel anterior.

Hay muchas formas distintas de estructurar una interfaz gráfica. Podríamos, por ejemplo, construir una sola [clase](#) con todos los elementos que el usuario va a ver en la pantalla y todo el código relacionado con los servicios para recibir información, presentar información, etc. El problema de esta solución es que sería muy difícil de construir y de mantener. Un buen [diseño](#) en este caso se refiere a una estructura clara, fácil de mantener y que sigue reglas que facilitan localizar los elementos que en ella participan. Esa es la principal preocupación de este nivel: cómo estructurar la [interfaz de usuario](#) y cómo comunicarla con las clases del modelo del mundo, sin mezclar en ningún momento las responsabilidades de esos dos componentes de un programa. De algún modo las acciones del usuario se deben convertir en eventos, los cuales deben ser interpretados por algún elemento de la interfaz y traducidos en llamadas a métodos de los objetos del modelo del mundo (ver [figura 5.1](#)).

Para la construcción de las interfaces de usuario, los lenguajes de programación proveen un conjunto de clases y mecanismos ya implementados, que van a facilitar, en gran medida, el trabajo del programador. Dicho conjunto se denomina un framework o, también, biblioteca

gráfica. Construir una [interfaz de usuario](#) se convierte entonces en el uso adecuado y en la especialización de los elementos que allí aparecen disponibles. Nosotros trabajaremos en este nivel sobre swing y awt, el framework sobre el que se basan la mayoría de las interfaces gráficas escritas en Java.

3. El Caso de Estudio

En este caso de estudio queremos construir un programa que permita a una persona calcular el valor de los impuestos que debe pagar por un automóvil. Para esto, el programa debe tener en cuenta el valor del vehículo y los descuentos que contempla la ley.

Un vehículo se caracteriza por una marca (por ejemplo, Peugeot, Mazda), una línea (por ejemplo, 206, 307, Allegro), un modelo, que corresponde al año de fabricación (por ejemplo, 2016, 2017), y un precio.

Para calcular el valor de los impuestos se establecen ciertos rangos, donde cada uno tiene asociado un porcentaje que se aplica sobre el valor del vehículo. Por ejemplo, si se tiene que los vehículos con precio entre 0 y 30 millones deben pagar el 1,5% del valor del vehículo como impuesto anual, un automóvil valuado en 10 millones debe pagar \$150.000 al año. La siguiente tabla resume el porcentaje de impuestos para los cuatro rangos de valores en que han sido divididos los automóviles.

- Entre 0 y 30 millones, pagan el 1,5% de impuesto.
- Más de 30 millones y hasta 70 millones, pagan el 2,0% de impuesto.
- Más de 70 millones y hasta 200 millones, pagan el 2,5% de impuesto.
- Más de 200 millones, pagan el 4% de impuesto.

Esta tabla se debe poder cambiar sin necesidad de modificar el programa, lo cual implica que pueden aparecer nuevos rangos, modificarse los límites o cambiar los porcentajes.

En el caso que queremos trabajar, están definidos tres tipos de descuentos:

1. Descuento por pronto pago (10% de descuento en el valor del impuesto si se paga antes del 31 de marzo).
2. Descuento para vehículos de servicio público (\$50.000 de descuento en el impuesto anual).
3. Descuento por traslado del registro de un automóvil a una nueva ciudad (5% de descuento en el pago).

Estos descuentos se aplican en el orden en el que acabamos de presentarlos. Por ejemplo, si el vehículo debe pagar \$150.000 de impuestos, pero tiene derecho a los tres descuentos, debería pagar \$80.750, calculados de la siguiente manera:

- $150.000 - 15.000 = 135.000$ (Primer descuento: $150.000 * 10\% = 15.000$)
- $135.000 - 50.000 = 85.000$ (Segundo descuento: 50.000)
- $85.000 - 4.250 = 80.750$ (Tercer descuento: $85.000 * 5\% = 4.250$)

El [diseño](#) de la interfaz de la aplicación ([figura 5.2](#)) trata de organizar los elementos del problema en zonas de trabajo fáciles de entender y utilizar por el usuario. Como se puede ver, después de la la imagen con el nombre de la aplicación hay una zona que tiene como objetivo mostrar la información sobre un vehículo y permite navegar por los vehículos existentes en la aplicación. Después, hay una zona que permite buscar un vehículo por línea o marca y encontrar el vehículo más caro. Luego hay una zona que permite seleccionar los descuentos que se desean aplicar. Finalmente, en la parte inferior se tiene una zona donde se ofrece la opción de calcular el impuesto a pagar por el vehículo actual, así como 2 opciones adicionales.

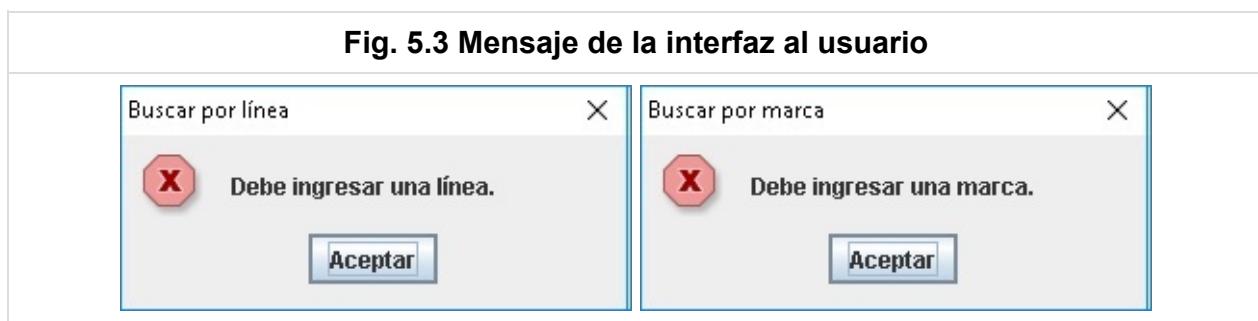
Fig. 5.2 Diseño de la interfaz de usuario del caso de estudio



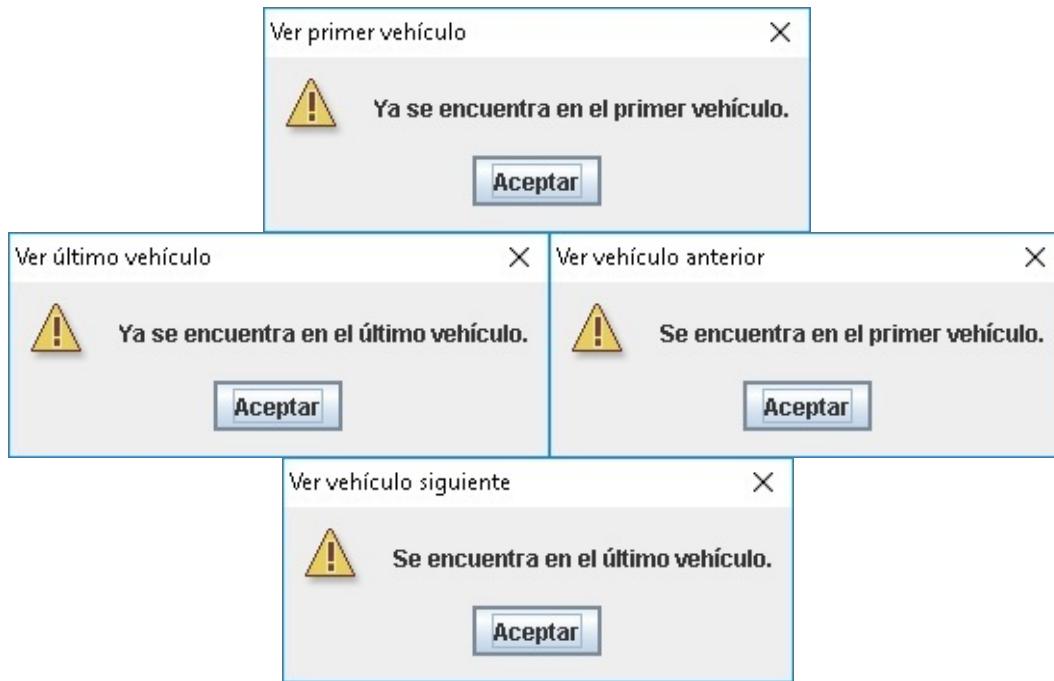
- La [ventana](#) del programa está dividida en cinco zonas: en la primera está la imagen con el título, en la segunda los datos del vehículo, en la tercera las opciones de búsqueda, en la cuarta la selección de descuentos, y en la quinta las opciones que provee la aplicación.
- El botón << permite visualizar el primer vehículo de la lista.
- El botón < permite visualizar el vehículo anterior.
- El botón > permite visualizar el vehículo siguiente.
- El botón >> permite visualizar el último vehículo de la lista.
- El botón Buscar por línea permite visualizar el primer vehículo que encuentre con la línea ingresada por el usuario.
- El botón Buscar por marca permite visualizar el primer vehículo que encuentre con la marca ingresada por el usuario.
- El botón Buscar vehículo más caro permite visualizar el vehículo con mayor valor.
- En la zona de descuentos, el usuario debe seleccionar los descuentos que quiere aplicar.
- El botón calcular muestra un mensaje con el valor total que debe pagar el usuario por el vehículo mostrado actualmente, incluyendo los descuentos seleccionados.
- Los botones Opción 1 y Opción 2 todavía no tienen una funcionalidad asignada.
- Fíjese que cada zona (menos la primera) tiene un borde y un título.

Si el usuario intenta hacer una búsqueda sin haber dado la información necesaria, el programa debe informarle del problema utilizando una [ventana](#) de diálogo, como se muestra en la [figura 5.3](#).

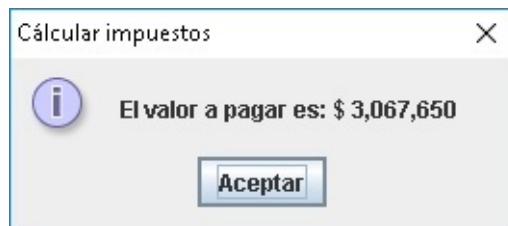
Fig. 5.3 Mensaje de la interfaz al usuario



Si el usuario trata de navegar una consulta sobre un vehículo cuya información no está registrada en el programa, se debe presentar la advertencia que aparece en la [figura 5.4](#).

Fig. 5.4 – Mensaje de la interfaz al usuario

Cuando el usuario selecciona la opción calcular, se debe presentar un mensaje con el valor a pagar, como se muestra en la [figura 5.5](#).

Fig. 5.5 – Mensaje de la interfaz al usuario

3.1. Comprensión de los requerimientos

A partir de la descripción del caso de estudio, podemos identificar al menos seis requerimientos funcionales:

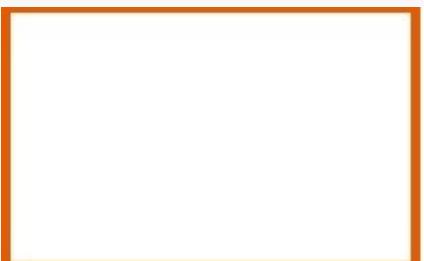
1. Navegar entre vehículos.
2. Buscar un vehículo por línea.
3. Buscar un vehículo por marca.
4. Buscar el vehículo más caro.
5. Calcular el impuesto de un carro.
6. Visualizar la información de un vehículo.

Tarea 1

Objetivo: Entender los requerimientos funcionales del caso de estudio.

Lea detenidamente el enunciado del caso de estudio, identifique los seis requerimientos funcionales y complete su documentación.

Requerimiento funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 2

	
Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 3

	
Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 4

	
Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 5

	
Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 6

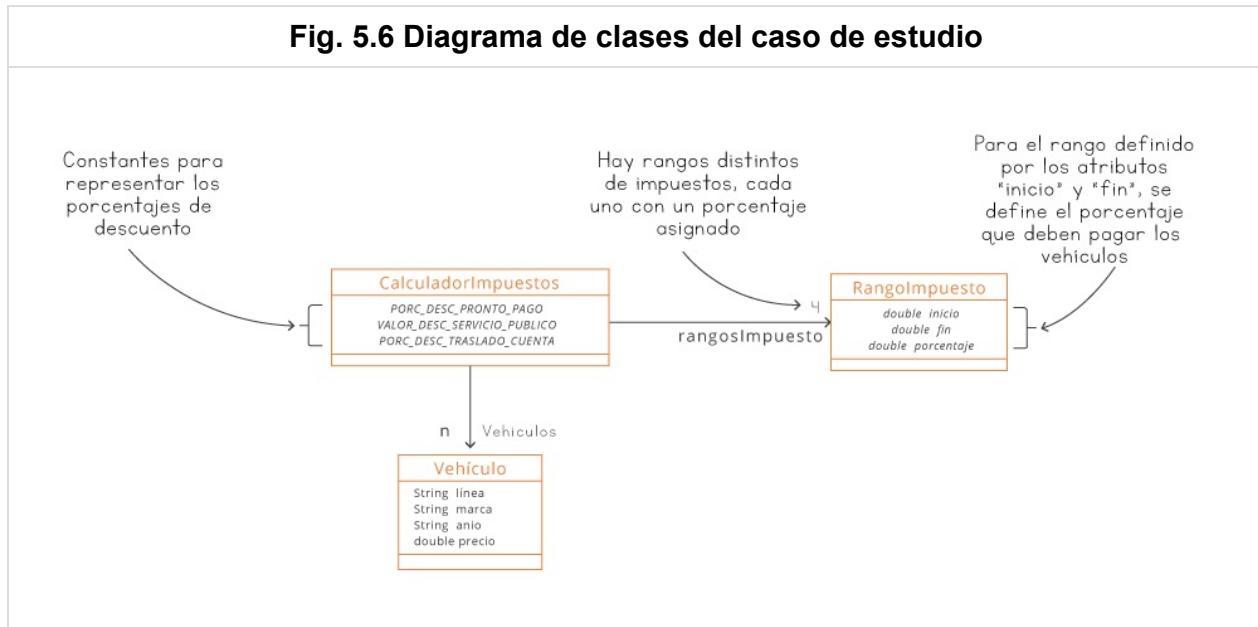
Nombre	
Resumen	
Entradas	
Resultado	

3.2. Comprensión del Mundo del Problema

En el modelo conceptual aparecen tres entidades con la estructura que se muestra en la [figura 5.6](#). Dichas entidades son:

1. El calculador de impuestos ([clase](#) CalculadorImpuestos).
2. El vehículo ([clase](#)Vehiculo).
3. Un rango de precios al que se le asocia un porcentaje de impuestos ([clase](#) RangolImpuesto).

Fig. 5.6 Diagrama de clases del caso de estudio



Tarea 2

Objetivo: Entender la estructura y las entidades del modelo conceptual del caso de estudio.

Llea de nuevo el enunciado del problema y estudie el diagrama de clases de UML que aparece en la [figura 5.6](#). Para cada **clase** describa las constantes, los atributos y las asociaciones que aparecen en el diagrama.

Clase **CalculadorImpuestos**:

Constantes:

Asociaciones:



Clase Vehículo:

Atributos:



Clase RangoImpuesto:

Atributos:



3.3. Definición de los Contratos

Describimos a continuación los contratos de los principales métodos de la [clase CalculadorImpuestos](#). Estos son los métodos que invocaremos desde la interfaz para pedir los servicios que solicite el usuario, pasándoles como parámetros la información que éste ingrese.

[Método constructor:](#)

```
/**  
 * Crea un calculador de impuestos, cargando la información de dos archivos.<br>  
 * <b>post:</b> Se inicializaron los arreglos de vehículos y rangos.<br>  
 *           Se cargaron los datos correctamente a partir de los archivos.<br>  
 * @throws Exception Si hay algún error al tratar de leer los archivos.  
 */  
public CalculadorImpuestos( ) throws Exception  
{ ... }
```

Leyendo este [contrato](#) podemos deducir tres cosas: el constructor sabe dónde encontrar sus archivos para leerlos (no es nuestro problema definir su localización), el contenido de dichos archivos debe ser correcto (el [método](#) no va a hacer ninguna [verificación interna](#)) y si hay algún error físico de lectura de los archivos, va a lanzar una [excepción](#) que deberá atrapar quien llame al constructor.

En algún punto de la [interfaz de usuario](#), con la instrucción que aparece a continuación, vamos a construir un [objeto](#) que representará el modelo del mundo. Dicha instrucción debe encontrarse dentro de un try-catch que nos permita atrapar la [excepción](#) que puede generarse.

```
CalculadorImpuestos calculador = new CalculadorImpuestos( );
```

Calcular pago de impuesto:

```
/**  
 * Calcula el pago de impuesto que debe hacer el vehículo actual.  
 * <b>pre:</b> Las listas de rangos y vehículos están inicializadas.  
 * @param pDescProntoPago Indica si aplica el descuento por pronto pago.  
 * @param pDescServicioPublico Indica si aplica el descuento por servicio público.  
 * @param pDescTrasladoCuenta Indica si aplica el descuento por traslado de cuenta.  
 * @return Valor por pagar de acuerdo con las características del vehículo y los  
 * descuentos que se pueden aplicar. Si no encuentra un rango para el modelo devuelve 0  
 * .  
 */  
double calcularPago( boolean pDescProntoPago, boolean pDescServicioPublico, boolean pD  
escTrasladoCuenta )  
{ ... }
```

Este caso está orientado a la construcción de la interfaz del programa, por lo que suponemos que ya se han implementado el modelo conceptual y las pruebas unitarias del mismo.

En las próximas secciones vamos a estudiar:

1. Cómo se organizan los elementos gráficos de la [interfaz de usuario](#) en clases Java.
2. Cómo se asignan las responsabilidades.
3. Cómo se maneja la interacción con el usuario.

Todo esto se ilustrará con el programa del caso de estudio, el cual construiremos paso a paso, dando respuesta a los tres puntos planteados anteriormente.

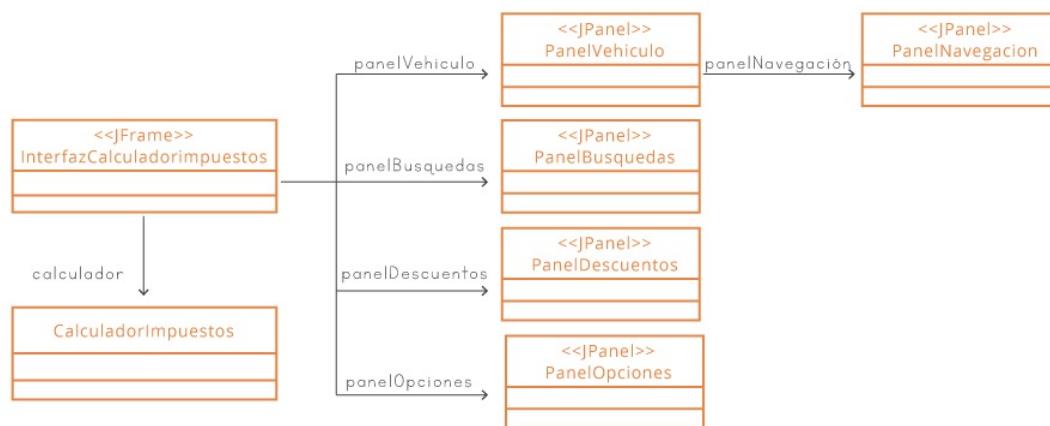
4. Construcción de Interfaces Gráficas

En este nivel vamos a estudiar una manera de construir interfaces de usuario para problemas pequeños. El [diseño](#) gráfico de estas interfaces incluye una [ventana](#) en la que aparece un formulario sencillo, el cual cuenta con algunos campos de edición y algunos botones para activar los requerimientos funcionales. Muchos de los elementos que se necesitan para crear una interfaz un poco más completa están por fuera del tema de este libro. El objetivo es estudiar únicamente lo indispensable para hacer una [interfaz de usuario](#) elemental. En particular, quedan por fuera todos los elementos de visualización e interacción que permiten manejar grupos de valores, de manera que, en algunos de los casos de estudio del libro, la interacción puede parecer un poco artificial.

Existen herramientas que permiten crear parcialmente el código en Java de la interfaz a partir de una descripción de la misma que se crea usando un editor gráfico. Pero en este nivel vamos a construir manualmente todos los elementos, puesto que es el único medio que tenemos de explorar a fondo la [arquitectura](#) del programa.

La buena noticia es que en la [interfaz de usuario](#) vamos a trabajar usando los mismos elementos e ideas que hemos utilizado hasta ahora. Allí vamos a encontrar clases, métodos, asociaciones, instrucciones iterativas, etc., y los vamos a expresar por medio de los mismos formalismos que hemos venido utilizando. El diagrama de clases de la interfaz, por ejemplo, se expresará en UML. La diferencia es que en lugar de trabajar con las entidades del mundo del problema, vamos a trabajar con las entidades del mundo gráfico y de interacción. En vez de tener conceptos como estudiante, tienda y banco, vamos a tener entidades como [ventana](#), botón, campo de texto, etc. Por lo demás, es aplicar lo que ya hemos aprendido a un mundo con otro tipo de elementos.

Nuestra estrategia de [diseño](#) consiste en identificar los elementos de la interfaz que tienen un propósito común y, para cada grupo de elementos, crear una [clase](#). Si revisamos el [diseño](#) gráfico de la interfaz en la [figura 5.2](#) podemos ver que ésta estará compuesta de seis clases principales: una que represente la [ventana](#) principal, otra que represente la zona de información del vehículo, una zona para la navegación, una zona para las búsquedas, una zona de descuentos y la última para la zona de las opciones. En la [figura 5.7](#) aparece el diagrama de clases de la [interfaz de usuario](#) del caso de estudio. En esa figura se muestra la [asociación](#) que existe entre una [clase](#) de la interfaz (la [clase](#) InterfazImpuestosCarro) y una [clase](#) del mundo del problema (la [clase](#) CalculadorImpuestos). Es usando dicha [asociación](#) que vamos a hacer las llamadas hacia el modelo del mundo.

Fig. 5.7 Diagrama de clases de la interfaz de usuario para el caso de estudio

El objetivo de este nivel es explicar la manera de construir el diagrama de clases de la [interfaz de usuario](#) y, posteriormente, implementarlo en Java usando swing y awt.

Hay dos aspectos prácticos que debemos tratar antes de seguir adelante: el primero es que las clases del [framework swing](#) están en el [paquete javax.swing](#) y en el [paquete java.awt](#). Esto significa que estos paquetes o alguno de sus subpaquetes deben ser importados cada vez que se quiera incluir un elemento gráfico. El segundo aspecto es que algunas de las clases de swing han ido cambiando según la versión del lenguaje. Lo que aparece en este libro vale para las versiones posteriores a Java 5. En todo caso, la adaptación a las versiones anteriores es trivial, y en la mayoría de los casos se reduce a una simple transformación en las llamadas de los métodos.

Comencemos con la tarea 3, en la que el lector debe tratar de identificar las entidades del mundo de la interfaz.

Tarea 3

Objetivo: Identificar intuitivamente las entidades, los atributos, las asociaciones y los métodos que forman parte de una interfaz gráfica.

1. Enumere al menos 8 elementos gráficos que pueden aparecer en una interfaz gráfica cualquiera (piense en elementos como [ventana](#), botón, [etiqueta](#), etc.).
2. Dibuje el diagrama de clases relacionando los elementos antes identificados por medio de asociaciones.
3. Complete la descripción de cada [clase](#) con los principales atributos que debería tener.
4. Agregue al diagrama de clases las signaturas de los métodos que reflejen las principales responsabilidades de cada uno de ellos.

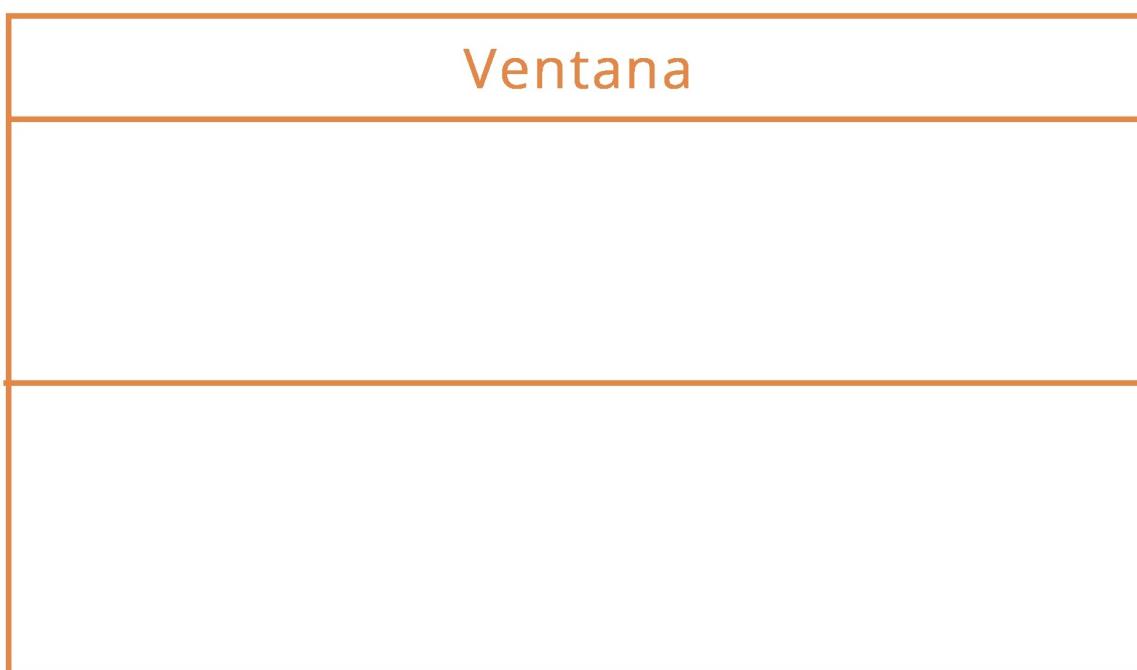
Identifique al menos 8 elementos de una [interfaz de usuario](#). Piense en los elementos que forman parte de la interfaz de un programa:

Para la [clase](#) que representa la [ventana](#) principal del programa, trate de identificar sus atributos. Guíese por las características que debe tener.

Dibuje el diagrama de clases con los elementos de una [interfaz de usuario](#) cualquiera:



Puede usar el siguiente diagrama como guía:

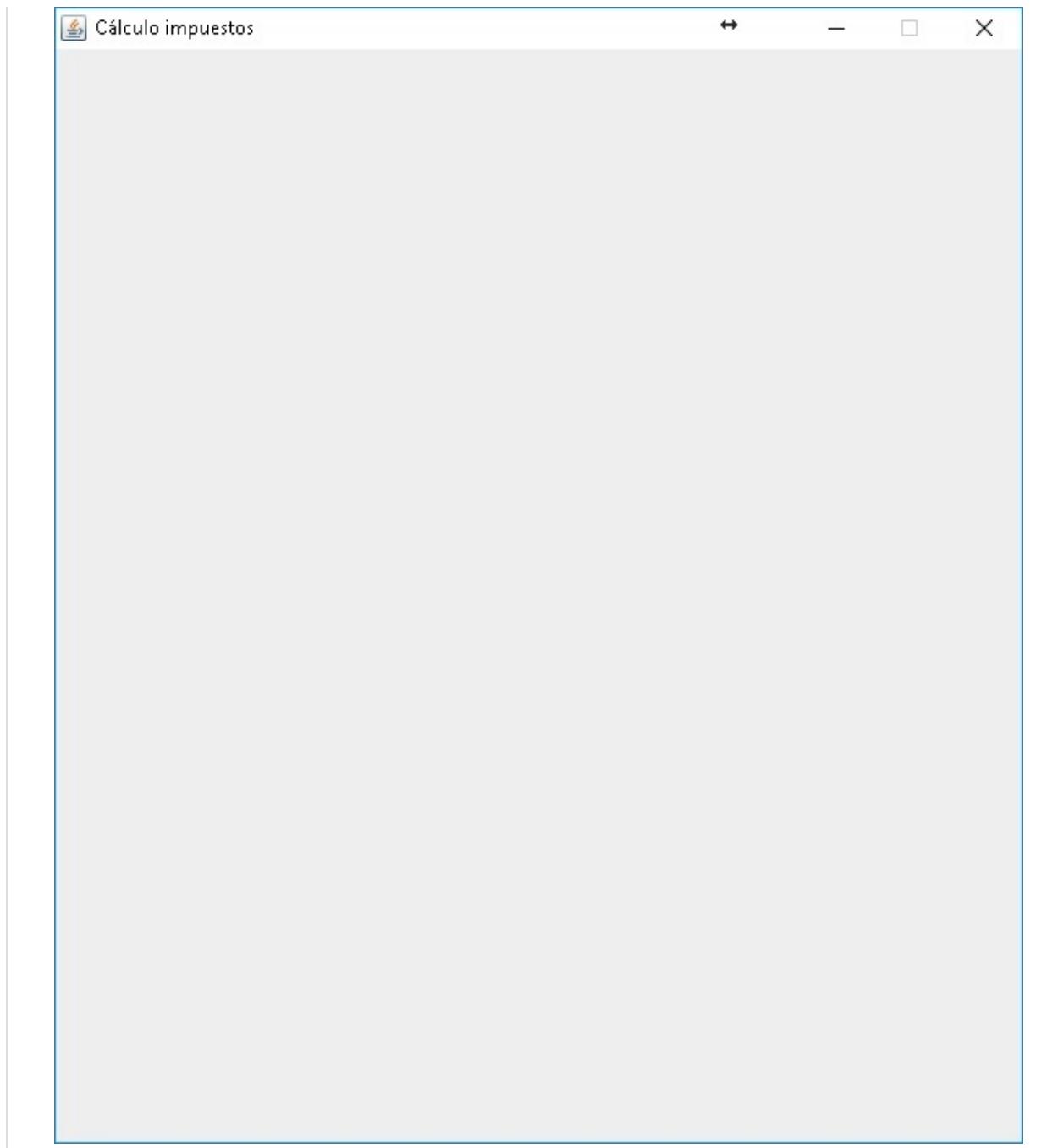


5. Elementos Gráficos Estructurales

5.1. Creación de la Ventana Principal

La [ventana](#) principal de la [interfaz de usuario](#) es la encargada de contener todos los elementos de visualización e interacción, por medio de los cuales el usuario va a utilizar el programa. Su única función es servir como marco para los demás elementos de la interfaz. Típicamente, la [ventana](#) tiene en la parte superior derecha los controles para cerrar el programa, minimizar y cambiar de tamaño. Cuenta también con una zona para presentar un título, como se ilustra en la [figura 5.8](#).

Fig. 5.8 Visualización de la [ventana](#) principal



Una [ventana](#) es el primer ejemplo de lo que se denomina un [contenedor gráfico](#). Al igual que con las estructuras contenedoras que manejábamos en el modelo del mundo, un [contenedor gráfico](#) está hecho para incluir dentro de él otros elementos gráficos más sencillos: es un medio para agrupar y estructurar componentes de visualización e interacción. De alguna manera, dentro de una [ventana](#), vamos a poder incluir las zonas de texto, los menús, los iconos, etc.

Una [ventana](#) es un [objeto](#) de una [clase](#) que se ha declarado de una manera particular ([clase](#) InterfazImpuestosCarro en el caso de estudio). Esta [ventana](#) principal va a contener la imagen con el título y cuatro de las zonas de trabajo que mencionamos antes y sus responsabilidades principales están relacionadas con la creación y organización visual de las zonas de trabajo.

La **clase** que representa la **ventana** principal (InterfazImpuestosCarro en nuestro ejemplo), al igual que cualquier **clase** del modelo del mundo, debe estar declarada en su propio **archivo** Java, siguiendo las mismas reglas definidas en los niveles anteriores. La única diferencia es que, como la **clase** pertenece a otro mundo distinto (el mundo gráfico), la vamos a situar en otro **paquete**. En el caso de estudio, por ejemplo, todas las clases de la interfaz van a estar en el **paquete** `uniandes.cupi2.impuestosCarro.interfaz`.

Para que la **ventana** principal tenga el comportamiento estándar de una **ventana**, como minimizarse, cerrarse o moverse cuando el usuario la arrastra, debemos indicar que nuestra **clase** es una extensión de una **clase** de un tipo particular llamado JFrame. Esta es una **clase** predefinida del **framework swing**, que tiene ya implementados los métodos para que la **ventana** se comporte de la manera esperada y no nos toca a nosotros, cada vez que hacemos una **ventana**, escribir el código para que se pueda mover, cerrar, etc.

Ejemplo 1

Objetivo: Presentar la manera de declarar en Java la **clase** que implementa la **ventana** de una **interfaz de usuario**.

En este ejemplo presentamos la declaración de la **clase** InterfazImpuestosCarro, la cual va a implementar la **ventana** de la interfaz para el caso de estudio. El código que se presenta en este ejemplo debe ir dentro del **archivo** InterfazImpuestosCarro.java, el cual se irá completando en los ejemplos de las secciones siguientes.

Al final del ejemplo estudiamos la representación de la **clase** en UML.

```
package uniandes.cupi2.impuestosCarro.interfaz;

import java.awt.*;
import javax.swing.*;

import uniandes.cupi2.impuestosCarro.mundo.*;

/**
 * Interfaz de cálculo de impuestos de vehículos
 */
public class InterfazImpuestosCarro extends JFrame
{
    ...
}
```

- La **clase** se declara dentro del **paquete** de las clases de la **interfaz de usuario**.
- Se importan las clases swing de los dos paquetes indicados (swing y awt).
- Se importan las clases del modelo del mundo. Debido a que están en un **paquete**

distinto, es indispensable especificar su posición.

- La **clase** se declara con la misma sintaxis de las clases del modelo del mundo. La única diferencia es que se agrega en la declaración el término `extends JFrame` para indicar que es una **ventana**.
- `JFrame` es la **clase** en swing que implementa las ventanas.



- En UML vamos a utilizar lo que se denominan estereotipos para representar las clases de la interfaz. Eso quiere decir que, en cada **clase**, se hace explícita la **clase** del **framework swing** que esa **clase** está extendiendo.
- Al extender la **clase** `JFrame`, tenemos derecho a utilizar dentro de nuestra **clase** todos sus métodos.

Las preguntas ahora son dos: ¿cómo hacemos para poner los elementos gráficos dentro de una **ventana**? y ¿cómo hacemos para modificar sus características? La respuesta a estas dos preguntas es la misma: tenemos un conjunto de métodos implementados en la **clase** `JFrame`, que podemos utilizar para cambiar el estado de la **ventana**. Con estos métodos, vamos a poder cambiar el título de la **ventana**, su tamaño o agregar en su interior otros componentes gráficos.

Algunos de los principales métodos que podemos usar con una **ventana** son los siguientes (la lista completa se puede encontrar en la documentación de la **clase** `JFrame`):

- **setSize(ancho, alto)**: este **método** permite cambiar el alto y el ancho de la **ventana**. Los valores de los parámetros se expresan en píxeles.
- **setResizable(cambiable)**: indica si el usuario puede o no cambiar el tamaño de la **ventana**.
- **setTitle(titulo)**: cambia el título que se muestra en la parte superior de la **ventana**.
- **setDefaultCloseOperation(EXIT_ON_CLOSE)**: indica que la aplicación debe terminar su ejecución en el momento en el que se cierre la **ventana**. `EXIT_ON_CLOSE` es una **constante** de la **clase**.
- **setVisible(esVisible)**: hace aparecer o desaparecer la **ventana** de la pantalla,

dependiendo del valor lógico que se le pase como [parámetro](#).

- **add(componente)**: permite agregar un [componente gráfico](#) a la [ventana](#). En la siguiente sección abordaremos el tema de cómo explicarle a la [ventana](#) la "posición" dentro de ella donde queremos añadir el componente.

La configuración de las características de la [ventana](#) (tamaño, zonas, etc.) se debe hacer en el [método](#) constructor de la [clase](#), tal como se muestra en el ejemplo 2. Lo único que nos falta en la [ventana](#) es agregar una [asociación](#) con las clases del modelo del mundo, de tal forma que sea posible traducir los eventos que genere el usuario en llamadas a los métodos que manipulan los objetos del mundo. Esto lo hacemos agregando una [asociación](#) en la [ventana](#) hacia uno o más objetos del mundo del problema.

Ejemplo 2

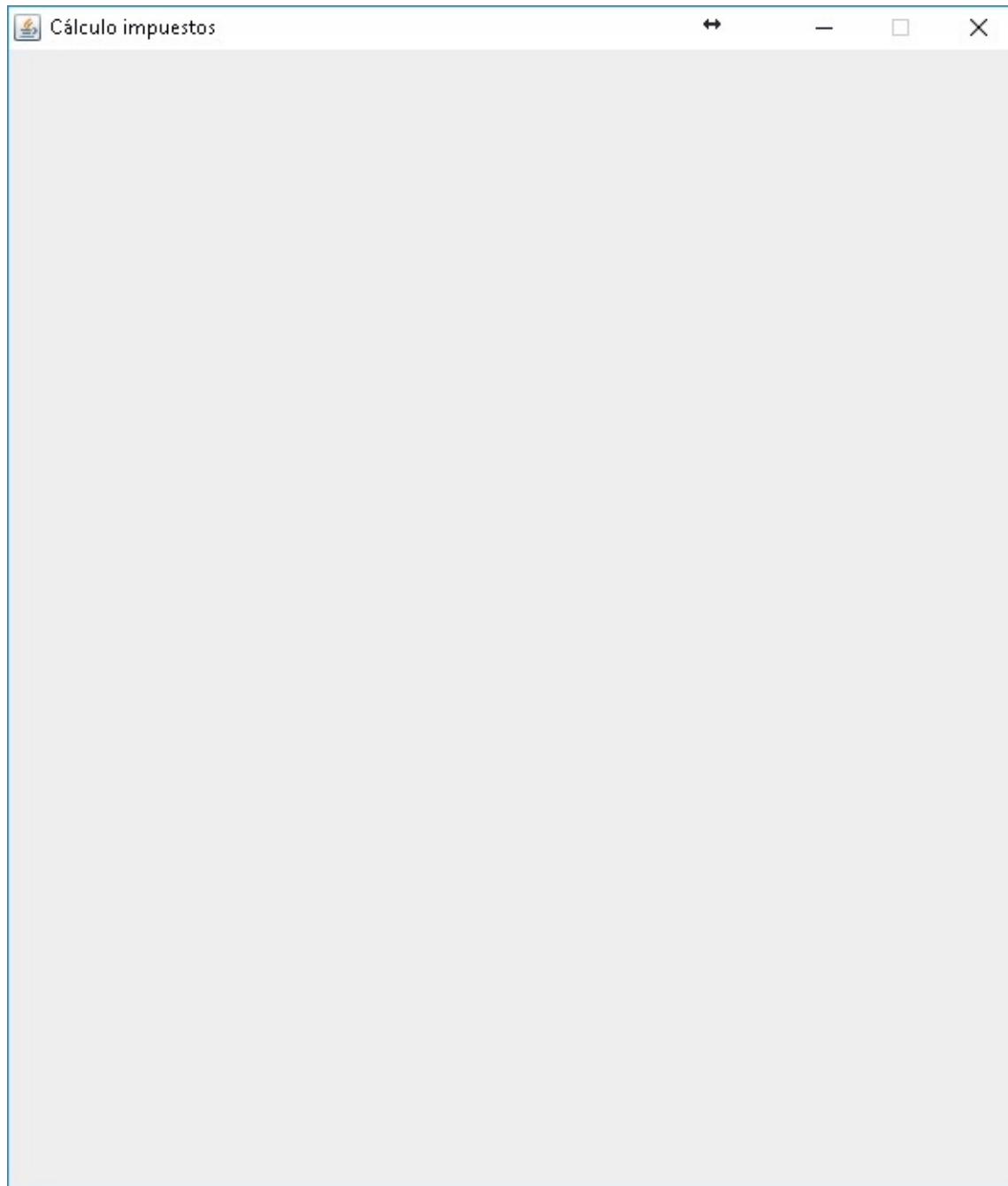
Objetivo: Mostrar la manera de definir la configuración básica de una [ventana](#).

En este ejemplo se muestra parte del [método](#) constructor de la [clase](#) que implementa la [ventana](#), lo mismo que la manera de declarar un [atributo](#) para representar la [asociación](#) con el modelo del mundo.

```
public class InterfazImpuestosCarro extends JFrame
{
    private CalculadorImpuestos calculador;

    public InterfazImpuestosCarro( )
    {
        setTitle( "Cálculo impuestos" );
        setSize( 600, 700 );
        setResizable( false );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        ...
    }
}
```

- En el [método](#) constructor de la [clase](#) de la [ventana](#) definimos su configuración: título, tamaño, evitamos que el usuario la cambie de tamaño y definimos que cuando el usuario cierre la [ventana](#) el programa debe terminar.
- La llamada de los métodos se hace como si fueran de nuestra propia [clase](#), puesto que pertenecen a la [clase](#) JFrame y nuestra [clase](#) la extiende.
- Si se incluye en la [clase](#) InterfazImpuestosCarro el constructor que aparece en este ejemplo y se ejecuta el programa, veremos aparecer en la pantalla la imagen que se muestra más abajo.



- La imagen corresponde a una [ventana](#) que tiene 600 píxeles de ancho y 700 píxeles de alto.
- Por ahora no agregamos los componentes internos de la [ventana](#), hasta que no tratemos el tema de distribución gráfica.
- Como [atributo](#) de la [ventana](#) definimos una [asociación](#) a un [objeto](#) de la [clase](#) CalculadorImpuestos. Ya veremos más adelante cómo se inicializa y cómo lo utilizamos para implementar los requerimientos funcionales.

5.2. Distribución Gráfica de los Elementos

El siguiente problema que debemos enfrentar en la construcción de la [ventana](#) es la distribución de los componentes gráficos que va a tener en su interior. Para manejar esto, Java incluye el concepto de [distribuidor gráfico](#) (layout), que es un [objeto](#) que se encarga de hacer esa tarea por nosotros. Lo que hacemos entonces en la [ventana](#), o en cualquier otro [contenedor gráfico](#) que tengamos, es crear y asociarle un [objeto](#) que se encargue de hacer este proceso; es decir que nosotros nos contentamos con agregar los componentes y dejamos que este [objeto](#) que creamos se encargue de situarlos en alguna parte de la [ventana](#).

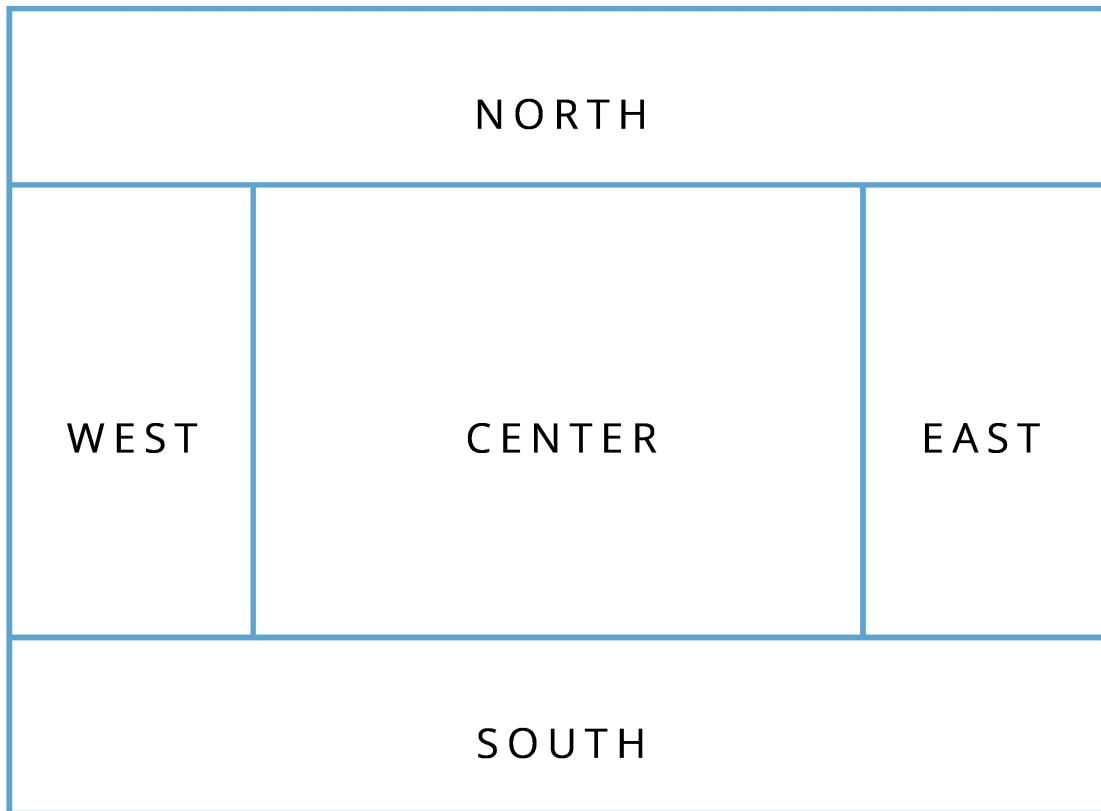
En el [framework swing](#) existe ya un conjunto de distribuidores gráficos listos para utilizar. En este nivel veremos dos de los más simples que existen, los cuales están implementados en las clases `BorderLayout` y `GridLayout`. Para asociar uno de estos distribuidores con cualquier [contenedor gráfico](#), se utiliza el [método](#) `setLayout()`, al cual se le debe pasar como [parámetro](#) una instancia de la [clase](#) que queremos que maneje la presentación gráfica de los elementos que contiene. En el caso de estudio, basta con agregarle al constructor de la [ventana](#) la siguiente llamada:

```
setLayout( new BorderLayout( ) );
```

- Si agregamos esta instrucción dentro del constructor de la [ventana](#) que vimos en el ejemplo 2, cada vez que agreguemos en ella un [componente gráfico](#), será la instancia de la [clase](#) `BorderLayout` que acabamos de crear la que se encargue de situar el nuevo elemento dentro de la [ventana](#).

5.2.1. Distribuidor en los Bordes

El [distribuidor en los bordes](#) (`BorderLayout`) divide el espacio del [contenedor gráfico](#) en cinco zonas, tal como muestra la [figura 5.9](#). Cada una de ellas se identifica con una [constante](#) definida dentro de la [clase](#) (`NORTH` , `CENTER` , `SOUTH` , `WEST` , `EAST`).

Fig. 5.9 Distribuidor en los bordes (BorderLayout)

Si asociamos este distribuidor con un [contenedor gráfico](#), cuando agreguemos un elemento deberemos pasar como [parámetro](#) la zona que queremos que éste ocupe. Por ejemplo, si quisiéramos situar un [componente gráfico](#) llamado `panelVehiculo` en la zona norte de la [ventana](#) del caso de estudio, deberíamos agregar en el constructor de la [clase](#) la siguiente instrucción:

```
add( panelVehiculo, BorderLayout.NORTH );
```

- Con esta instrucción agregamos un [componente gráfico](#) llamado `panelVehiculo` en la zona norte de un [contenedor gráfico](#) que tiene asociado un [distribuidor en los bordes](#).
- Fíjese cómo se referencia la [constante](#) `NORTH` de la [clase](#) `BorderLayout`.

Es importante resaltar que este distribuidor utiliza el tamaño definido por cada uno de los componentes que va a albergar (cada uno tiene un ancho y un alto en pixeles) para reservarles espacio en el [contenedor gráfico](#), y asigna todo el espacio sobrante para el componente que se encuentre en la zona del centro. Nosotros usaremos este [distribuidor gráfico](#) para construir la [interfaz de usuario](#) del caso de estudio, de manera que esto último lo veremos en detalle más adelante.

5.2.2. Distribuidor en Malla

Para usar el [distribuidor en malla](#), se debe indicar en su constructor el número de filas y de columnas que va a tener, las cuales van a establecer las zonas en las que estará dividido el [contenedor gráfico](#), tal como se muestra en la [figura 5.10](#) para un distribuidor de 4 filas y 3 columnas.

Fig. 5.10 Distribuidor en malla (GridLayout) con orden de llenado

fila 1	1	2	3
fila 2	4	5	6
fila 3	7	8	9
fila 4	10	11	12

- Además de definir una estructura en filas y columnas, el [distribuidor en malla](#) define un orden de llenado.
- La primera zona que se va a ocupar es la que se encuentra en la primera columna de la primera fila (arriba a la izquierda de la [ventana](#)).
- Los componentes deben agregarse secuencialmente, siguiendo el orden de llenado del distribuidor.

Para asociar un distribuidor con un [componente gráfico](#) se utiliza la siguiente instrucción, siguiendo con el ejemplo de 4 filas y 3 columnas:

```
setLayout( new GridLayout( 4, 3 ) );
```

- Si esta instrucción se coloca en el constructor de un [contenedor gráfico](#), todos los

elementos que se le agreguen ocuparán en orden cada una de las 12 zonas en las que está dividido.

A diferencia del [distribuidor en los bordes](#), cuando se utiliza un [distribuidor en malla](#) no es necesario definir la posición que va a ocupar el componente que se va a incluir, porque estas posiciones son asignadas en orden de llegada: se llena primero toda la fila 1, luego la fila 2 y así sucesivamente. Este distribuidor ignora el tamaño definido para cada componente, ya que hace una distribución uniforme del espacio. En la próxima sección veremos un ejemplo de uso de este [distribuidor gráfico](#).

5.3. Divisiones y Paneles

Dentro de la [ventana](#) principal aparecen las divisiones (o paneles), encargadas de agrupar los elementos gráficos por contenido y uso, de tal manera que sea sencillo para el usuario localizarlos y usarlos. Esta manera de estructurar la visualización del programa es muy importante, puesto que de ella depende en gran medida lo fácil e intuitivo que resulte utilizarlo. En la interfaz del caso de estudio (figura 5.2), por ejemplo, tenemos cuatro divisiones dentro de la [ventana](#): en la primera van los datos del vehículo, en la segunda las opciones del búsqueda, en la tercera los descuentos y, en la cuarta, los botones con las opciones.

Cada división se implementa como una [clase](#) aparte en el modelo (en nuestro caso, con las clases PanelDescuentos, PanelBusquedas, PanelResultados y PanelVehiculo) y, al igual que la [ventana](#), cada una de ellas es un [contenedor gráfico](#) al cual hay que asociarle su propio distribuidor (layout) y al cual se le pueden agregar en su interior otros componentes gráficos. En el caso del [panel Vehículo](#), se puede observar que contiene adicionalmente el PanelNavegacion. En el constructor de la [ventana](#) se debe crear una instancia de cada una de las divisiones o paneles y luego agregarlas a la [ventana](#). Este proceso se ilustra en el ejemplo 3.

Ejemplo 3

Objetivo: Mostrar la manera de agregar paneles a una [ventana](#).

En este ejemplo se muestra el [método](#) constructor de la [clase](#) InterfazImpuestosCarro, en donde se crean las instancias de los cuatro paneles y luego se agregan a la [ventana](#) en una de las zonas del [distribuidor en los bordes](#). Aquí se debe suponer que las clases que implementan cada una de las divisiones ya fueron creadas y sus nombres son: PanelBusqueda, PanelDescuentos, PanelOpciones y PanelVehiculo. Note que las asociaciones con los paneles se declaran como cualquier otra [asociación](#) en Java.

```

public class InterfazImpuestosCarro extends JFrame
{
    private CalculadorImpuestos calculador;

    private PanelVehiculo panelVehiculo;
    private PanelBusquedas panelBusquedas;
    private PanelDescuentos panelDescuentos;
    private PanelOpciones panelOpciones;

    public InterfazImpuestosCarro( )
    {
        setTitle( "Cálculo impuestos" );
        setSize( 290, 300 );
        setResizable( false );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        setLayout( new BorderLayout( ) );

        JPanel centro = new JPanel( );
        centro.setLayout( new BorderLayout( ) );
        add( centro, BorderLayout.CENTER );

        panelVehiculo = new PanelVehiculo( this );
        centro.add( panelVehiculo, BorderLayout.CENTER );

        panelBusquedas= new PanelBusquedas( this );
        centro.add( panelBusquedas, BorderLayout.SOUTH );

        JPanel sur = new JPanel( );
        sur.setLayout( new BorderLayout( ) );
        add( sur, BorderLayout.SOUTH );

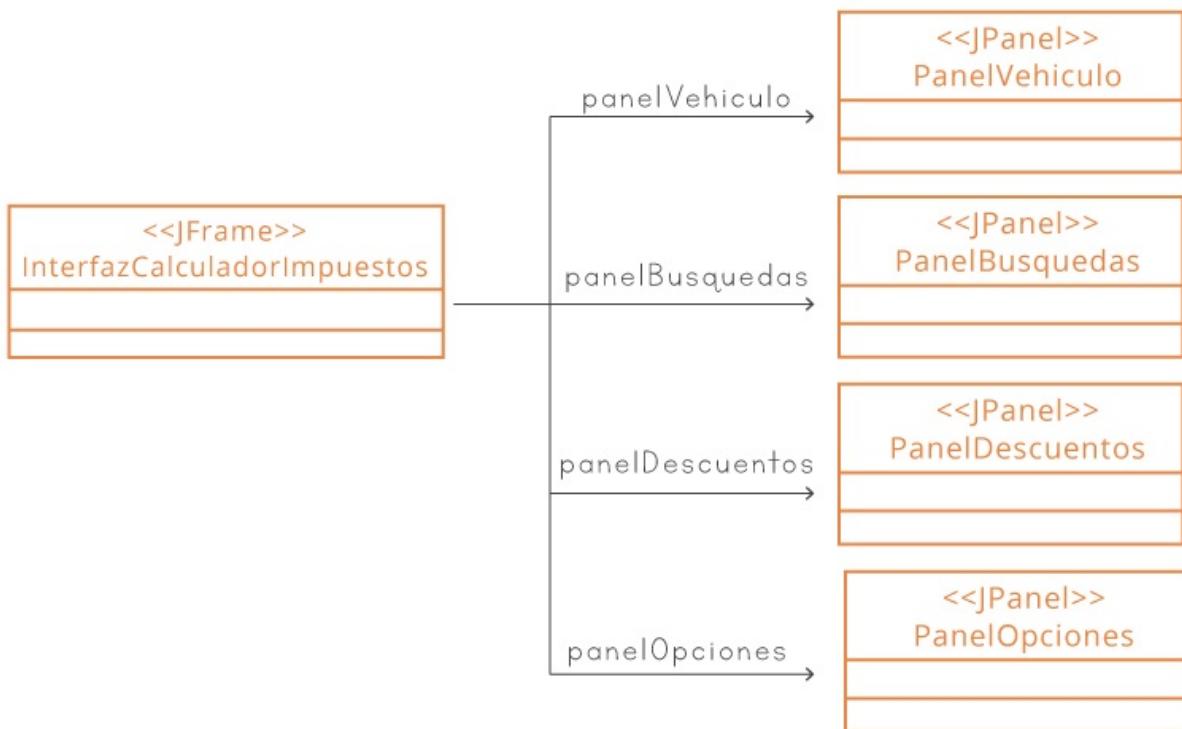
        panelDescuentos = new PanelDescuentos( );
        sur.add( panelDescuentos, BorderLayout.CENTER );

        panelOpciones = new PanelOpciones( this );
        sur.add( panelOpciones, BorderLayout.SOUTH );
    }
}

```

- En la [ventana](#) se declara un [atributo](#) por cada una de las divisiones o paneles.
- En el constructor se asocia con la [ventana](#) un [distribuidor en los bordes](#).
- Esta es una versión provisional del constructores, que después cambiaremos levemente a medida que vayamos conociendo nuevos elementos.
- Se crea una instancia de cada una de las divisiones y se agrega en una posición de las definidas en el [distribuidor en los bordes](#).
- Por el momento no agregaremos nada en el norte, debido a que este espacio se reservará para la imagen con el título de la aplicación.

- Debido a que hay más paneles que divisiones disponibles, se crean dos paneles auxiliares llamados centro y sur.
- El **panel** con la información del vehículo va en el centro del **panel** centro.
- El **panel** para las búsquedas va en el sur del **panel** centro.
- El **panel** con la información de los descuentos va en el centro del **panel** sur.
- El **panel** con las opciones va en el sur del **panel** sur.
- Las zonas este y oeste quedan sin ningún componente en ellas, por lo que el distribuidor no les asigna ningún espacio en la **ventana**.



- Con el **método** constructor definido hasta el momento, hemos creado las asociaciones que se muestran en el diagrama de clases de la figura.
- Se omite la **asociación** hacia el modelo del mundo para concentrarnos únicamente en los elementos gráficos.

Para la construcción de las clases que representan las divisiones, se sigue un proceso muy similar al que seguimos con la **ventana**, ya que todas comparten el hecho de ser contenedores gráficos. Ahora, en lugar de la **clase** JFrame, que representa las ventanas en swing, vamos a utilizar la **clase** JPanel, que representa las divisiones o paneles.

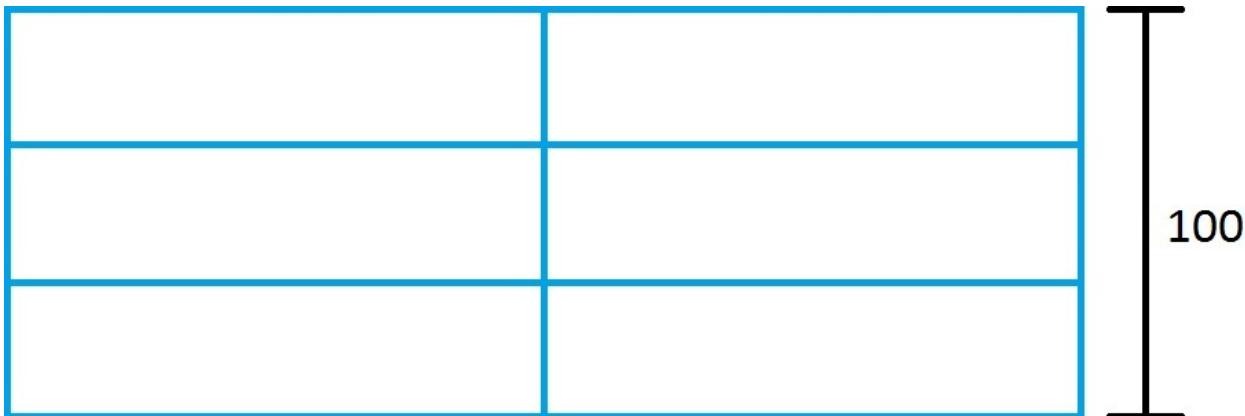
Una diferencia importante es que ahora usamos el [método setPreferredSize\(dimension\)](#) para definir el tamaño de las divisiones (en el ejemplo 4 se explica su utilización en más detalle). Esta información es facultativa; el [distribuidor gráfico](#) decide si hace uso de ella, si sólo la utiliza parcialmente o si sencillamente la ignora.

Ejemplo 4

Objetivo: Mostrar la manera de declarar los paneles de una [ventana](#).

En este ejemplo se muestra la declaración en Java de las clases que implementan los paneles de la [ventana](#) principal de la [interfaz de usuario](#) en el caso de estudio.

```
public class PanelBusquedas extends JPanel
{
    public PanelBusquedas( )
    {
        setLayout( new GridLayout(3,2) );
        setPreferredSize( new Dimension( 0, 100 ) );
    }
}
```

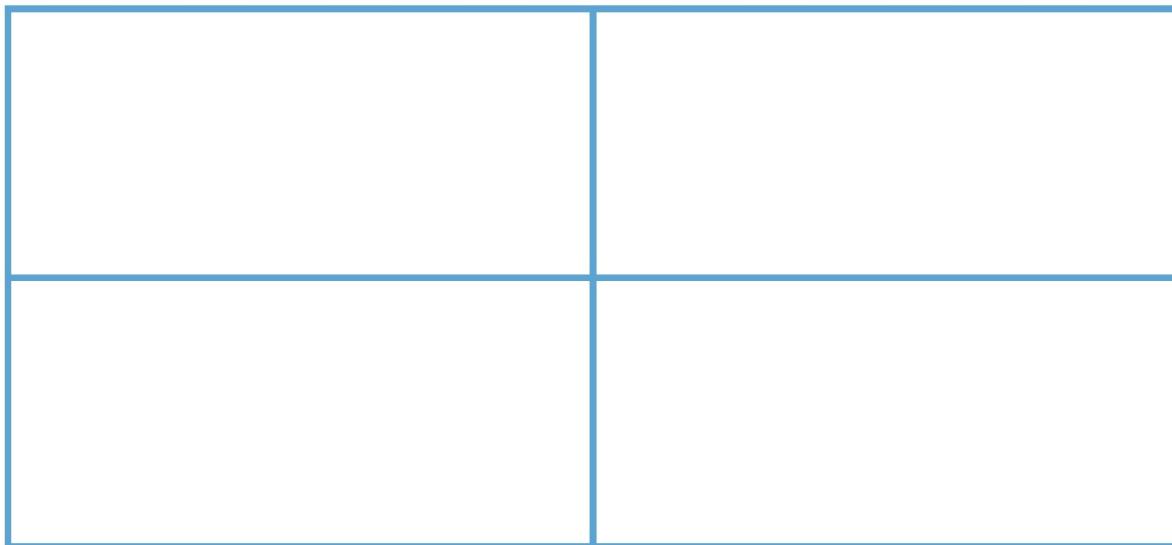


El [panel](#) con las opciones de búsqueda estará dividido en 6 zonas (3 filas y 2 columnas en cada una).

En el momento de definir la dimensión del [panel](#) es importante declarar el alto que queremos que tenga (100 píxeles), puesto que este valor es utilizado por el [distribuidor en los bordes](#) para reservarle espacio al [panel](#), y ése es el distribuidor que usamos en la [ventana](#) principal para situar este [panel](#) en la [ventana](#). Fíjese cómo se utiliza la [clase Dimension](#) para definir el tamaño del [panel](#).

Al definir la dimensión del [panel](#), pasamos 0 píxeles como ancho. Allí habríamos podido escribir cualquier valor, porque de todas maneras el distribuidor lo ignorará y le asignará como ancho todo el espacio disponible en la [ventana](#), descontando el espacio necesario para los componentes del este y del oeste.

```
public class PanelDescuentos extends JPanel
{
    public PanelDescuentos( )
    {
        setLayout( new GridLayout( 2,2 ) );
    }
}
```



El **panel** con la información de los descuentos estará dividido en cuatro zonas (dos filas y dos columnas en cada una). Aquí no es importante definir la dimensión del **panel**, porque el distribuidor de la **ventana** en la cual va a estar situado le asignará todo el espacio disponible después de haber colocado los otros paneles.

```
public class PanelOpciones JPanel
{
    public PanelOpciones ( )
    {
        setLayout( new GridLayout( 1, 3 ) );
    }
}
```



El **panel** con las opciones del programa estará dividido en 3 zonas(una fila y tres columnas). Aquí tampoco se debe definir la dimensión del **panel**.

```

public class PanelVehiculo( ) extends JPanel
{
    public class PanelVehiculo( )
    {
        setLayout( new BorderLayout( ) );

        JPanel informacion = new JPanel( );
        informacion.setLayout( new GridLayout( 4, 2, 10, 5 ) );
        add( informacion, BorderLayout.CENTER );

        PanelNavegacion panelNavegacion = new PanelNavegacion( );
        add( panelNavegacion, BorderLayout.SOUTH );
    }
}

```

El **panel** con la información del vehículo es más complejo que los paneles anteriores. Por esta razón, se deben utilizar nuevamente paneles auxiliares que permitan ajustar todos los elementos en esa distribución. Como podemos ver en la imagen anterior, se divide el **panel** en 3 zonas, una con la imagen del vehículo, otra con los datos del vehículo y otra con las opciones de navegación.



- Se usa un BorderLayout para la distribución general del **panel**.
- Por ahora no agregaremos nada en el oeste, debido a que este espacio se reservará para la imagen del vehículo.
- Se crea un **panel** auxiliar para llamado **informacion** para poner la información del vehículo, y se ubica en el centro del **panel**.
- El **panel** **informacion** usa un GridLayout, con 4 filas y 2 columnas. Se agregan 3 parámetros adicionales al constructor del **distribuidor gráfico**, para indicar los espacios, en pixeles, entre cada uno de sus zonas. En este caso, se deja un espacio de 10 pixeles entre las columnas y un espacio de 5 pixeles entre las filas.



- Para el **panel** de navegación, vamos a crear una nueva **clase**, llamada **PanelNavegacion**, debido a que tiene funciones diferentes que veremos más adelante.
- Las zonas este y norte quedan sin ningún componente en ellas, por lo que el distribuidor no les asigna ningún espacio en la **ventana**.

```
public class PanelNavegacion extends JPanel
{
    public PanelNavegacion( )
    {
        setLayout( new GridLayout( 1, 4 ) );
    }
}
```

El **panel** con las opciones de navegación del programa estará dividido en 4 zonas(una fila y 4 columnas).



Con esta **clase** completamos seis clases en el **paquete** de la interfaz: una para la **ventana**, cuatro para los paneles en los cuales la **ventana** está dividida y una para el **panel** de navegación.

La **clase** **JPanel** dispone de una amplia variedad de métodos para manejar sus propiedades. Si quiere modificar el color, por ejemplo, pruebe alguna de las siguientes instrucciones dentro del respectivo **método** constructor. En la documentación de la **clase** encontrará la lista de servicios que ofrece dicha **clase**.

```
setForeground( Color.RED );
setBackground( Color.WHITE );
```

Para facilitar la identificación de las divisiones dentro de la **ventana**, tenemos el concepto de borde, que se maneja como un **objeto** que se asocia con el **panel**. La creación de los bordes se hace de manera de la misma manera como se crean otros objetos (se utiliza el **método**

`new`) y la [asociación](#) con el [panel](#) se realiza de la manera que se muestra en el ejemplo 5.

Ejemplo 5

Objetivo: Mostrar la manera de crear un borde en un [panel](#).

En este ejemplo se muestra la creación de los bordes para las tres divisiones del programa del caso de estudio. De todos los tipos de borde disponibles en swing, vamos a utilizar el borde con título, el cual permite que, además de marcar las divisiones, podamos asociar una cadena de caracteres que indique el contenido de cada uno de los paneles.

A continuación se presentan las instrucciones que se deben agregar a los métodos constructores de los paneles para asociarles los bordes necesarios. Al final se muestra la imagen de la interfaz que se ha construido hasta el momento.

```
public PanelVehiculo( )
{
    ...
    TitledBorder border = new TitledBorder( "Datos del vehículo" );
    border.setTitleColor( Color.BLUE );
    setBorder( border );
}

public PanelBusquedas( )
{
    ...
    TitledBorder border = new TitledBorder( "Búsquedas" );
    border.setTitleColor( Color.BLUE );
    setBorder( border );
}

public PanelDescuentos( )
{
    ...
    TitledBorder border = new TitledBorder( "Descuentos" );
    border.setTitleColor( Color.BLUE );
    setBorder( border );
}

public PanelOpciones( )
{
    ...
    TitledBorder border = new TitledBorder( "Opciones" );
    border.setTitleColor( Color.BLUE );
    setBorder( border );
}
```



Es conveniente utilizar alguna convención clara para nombrar las clases de los componentes gráficos. En este libro las clases que implementan las divisiones de las ventanas comenzarán por la cadena "[Panel](#)", seguidas de una descripción de su contenido. Con esta convención podemos fácilmente localizar las clases involucradas en algún aspecto de la interfaz.

Como lo vimos anteriormente, cuando en una [ventana](#) necesitamos cuatro o más divisiones en sentido vertical y horizontal y queremos utilizar el distribuidor en bordes, lo único que debemos hacer es agregar divisiones adicionales dentro de uno de los paneles, aprovechando que éstos son contenedores gráficos y pueden contener en su interior cualquier tipo de [componente gráfico](#).

En este punto ya se tienen los conceptos indispensables para comenzar a utilizar los entrenadores de construcción de interfaces de usuario, uno de los cuales permite manipular interactivamente los distribuidores gráficos vistos en este capítulo.

5.4. Etiquetas y Zonas de Texto

Una vez que hemos terminado de estructurar las divisiones y hemos asociado con cada una de ellas un [distribuidor gráfico](#), podemos comenzar a agregar los elementos gráficos y de interacción. Vamos a empezar por dos de los componentes gráficos más simples, que permiten una comunicación básica con el usuario: las etiquetas y las zonas de texto.

Las etiquetas permiten agregar un texto corto o imágenes como parte de la interfaz, la mayor parte de las veces con el fin de explicar algún elemento de interacción, por ejemplo una [zona de texto](#). Las etiquetas (labels) son objetos de la [clase JLabel](#) en swing, que se crean pasando en el constructor el texto que deben contener. Dicha [clase](#) cuenta con diversos métodos, entre los cuales tenemos los siguientes:

- **setText(etiqueta):** permite cambiar el texto de la [etiqueta](#).
- **setForeground(color):** permite cambiar el color de la [etiqueta](#). Como color se puede pasar cualquiera de las constantes de la [clase Color](#) (`BLACK` , `RED` , `GREEN` , `BLUE` , etc.), o definir un nuevo color utilizando los tres índices ROJO-VERDE-AZUL del estándar RGB.

Para agregar una [etiqueta](#) a un [panel](#), se siguen cuatro pasos:

1. Declarar en el [panel](#) un [atributo](#) de la [clase JLabel](#).
2. Agregar la instrucción de creación de la [etiqueta](#) (`new`).
3. Utilizar los métodos de la [clase](#) para configurar los detalles de visualización deseados.
4. Utilizar la instrucción `add` del [panel](#) para agregarla en la zona que le corresponda.

Estos cuatro pasos son los mismos para cualquier [componente gráfico](#) que se quiera incorporar a una división. En el ejemplo 6 aparece el código necesario para crear todas las etiquetas de la interfaz del caso de estudio.

También es importante definir una convención de nombres para los atributos, que permita distinguir el tipo de [componente gráfico](#) al que corresponde. Nuestra convención es que el nombre de los atributos que representan las etiquetas comienza por la cadena "lab", mientras que aquellos que representan una [zona de texto](#) comienzan por "txt".

Las zonas de texto (objetos de la [clase JTextField](#)) cumplen dos funciones en una interfaz. Por una parte, permiten al usuario ingresar la información correspondiente a las entradas de los requerimientos funcionales (por ejemplo, la marca del vehículo) y, por otra, obtener las

respuestas calculadas por el programa (por ejemplo, el monto que se debe pagar por impuestos). Los siguientes métodos permiten configurar y manipular las zonas de texto:

- **getText()**: retorna la cadena de caracteres ingresada por el usuario dentro de la [zona de texto](#). Independientemente de si el usuario ingresó un valor numérico o una secuencia de letras, todo lo que el usuario ingresa se maneja y retorna como una cadena de caracteres. Más adelante veremos cómo convertirla a un número cuando así lo necesitemos.
- **setText(texto)**: presenta en la [zona de texto](#) la cadena que se pasa como [parámetro](#). Este [método](#) se usa frecuentemente para mostrar los resultados de un cálculo hecho por el programa.
- **setEditable(editable)**: indica si el contenido de la [zona de texto](#) puede ser modificado por el usuario. En el caso de las zonas de texto utilizadas para mostrar resultados, es común impedir que el usuario modifique el valor allí contenido.
- **setForeground(color)**: define el color de los caracteres que aparecen en la [zona de texto](#). De la misma manera que con las etiquetas, aquí se pueden usar las constantes de la [clase Color](#) o crear otro color distinto.
- **setBackground(color)**: define el color del fondo de la [zona de texto](#).

Ejemplo 6

Objetivo: Mostrar la manera de agregar componentes gráficos simples a un [panel](#).

Este ejemplo muestra la manera de añadir los componentes gráficos al primer [panel](#) de la interfaz del caso de estudio. Inicialmente, se presenta el contenido final esperado. Luego se muestran las instrucciones que se deben agregar al [método](#) constructor del primer [panel](#) para lograr el objetivo. Lo único que no se agrega en este momento es el [panel](#) con los botones de navegación, que es tema de una sección posterior.



```

public class PanelVehiculo extends JPanel
{
    //-----
    // Atributos
    //-----

    private JTextField txtMarca;
    private JTextField txtLinea;
    private JTextField txtModelo;
    private JTextField txtValor;
    private JLabel labImagen;
}

```

Paso 1: se declara un **atributo** en la **clase** por cada **componente gráfico** cuyo valor cambiará después de creado, que se quiera incluir en el **panel**.

Si vemos la imagen anterior, podemos ver que tendremos 4 etiquetas (Marca, Línea, Modelo y Valor) con texto, 4 zonas de texto asociadas y , una **etiqueta** con la imagen del vehículo. Como el valor de las etiquetas nunca cambia, no la agregamos como **atributo**.

Es conveniente asociar parejas de nombres para indicar que los componentes están relacionados entre sí. Por ejemplo los nombres `txtMarca` y `labMarca` indican que se trata de dos componentes relacionados con el mismo concepto (la marca del vehículo).

```

public PanelVehiculo( )
{
    ...
    labImagen = new JLabel( );
    JLabel labMarca = new JLabel( "Marca" );
    txtMarca = new JTextField( );

    JLabel labLinea = new JLabel( "Línea" );
    txtLinea = new JTextField( );

    JLabel labModelo = new JLabel( "Modelo" );
    txtModelo = new JTextField( );

    JLabel labValor = new JLabel( "Valor" );
    txtValor = new JTextField( );
    ...
}

```

Note que las campos de texto y la **etiqueta** con la imagen todavía no tienen ningún valor, porque este depende del vehículo que se quiera visualizar.

Paso 2: en el constructor del **panel** se crean los objetos que representan cada uno de los componentes gráficos.

En los constructores de algunos elementos gráficos es posible configurar algunas de las características que queremos que tenga. Estas instrucciones se escriben después de las instrucciones de definición del [distribuidor gráfico](#) y del borde.

```
txtValor.setEditable( false );
txtValor.setForeground( Color.BLUE );
txtValor.setBackground( Color.WHITE );
```

En este caso, se indica el campo de texto `txtValor` no puede ser editado por el usuario, que el color del texto de la [etiqueta](#) es azul y el color de fondo blanco.

Paso 3: utilizando los métodos de cada [clase](#) se configura el componente.

Aquí sólo van las características que no hayan podido ser definidas en la creación del [objeto](#).

```
add( labImagen, BorderLayout.WEST );

informacion.add( labMarca );
informacion.add( txtMarca );
informacion.add( labLinea );
informacion.add( txtLinea );
informacion.add( labModelo );
informacion.add( txtModelo );
informacion.add( labValor );
informacion.add( txtValor );
}
```

Paso 4: se añaden al [panel](#) los componentes gráficos creados. La imagen del vehículo se agrega en el oeste del [panel](#), que es la zona que se había reservado para esto. El resto de las etiquetas y los campos de texto se agregan en el [panel](#) de información, se teniendo cuidado de agregarlos en el orden utilizado por el [distribuidor gráfico](#) (de izquierda a derecha y de arriba a abajo).

5.5. Formateo de Datos Numéricos

En algunas ocasiones, es importante formatear de manera adecuada los valores numéricos en el momento de presentárselos al usuario. Si el valor de los impuestos del vehículo actual es 1615500,120023883, debemos buscar la manera de que en la [zona de texto](#) aparezca algo del estilo "\$ 1.615.500,00". Esto se logra con el código que se presenta a continuación, en el cual suponemos que en la [variable](#) precio, de tipo real, está el valor que queremos presentar y que la [zona de texto](#) en donde debe aparecer se llama `txtValor`:

```
DecimalFormat df = (DecimalFormat) NumberFormat.getInstance( );
df.applyPattern( "$ ###.###,##" );
String strPrecio= df.format( precio );
txtValor.setText( strPrecio );
```

- DecimalFormat es una [clase](#) que hace este tipo de formateo. Se encuentra en el [paquete](#) java.text.
- En la primera línea se obtiene una instancia de dicha [clase](#).
- En la segunda línea se define el formato que queremos utilizar. Marcamos con # los espacios ocupados por los dígitos que forman parte del número.
- En la tercera línea aplicamos el formato al valor que se encuentra en la [variable](#) llamada "precio".
- En la última línea colocamos la respuesta en la [zona de texto](#) llamada txtValor, utilizando el [método](#) setText().

5.6. Selección de Opciones

El [framework swing](#) provee un [componente gráfico](#) que permite al usuario seleccionar o no una opción. En el caso de estudio lo utilizamos para que el usuario seleccione los descuentos a los que tiene derecho. Con estos controles el usuario sólo puede decir "sí" o "no". El manejo de estos componentes gráficos sigue las mismas reglas explicadas en la sección anterior, tal como se muestra en el ejemplo 7.

Estos componentes son manejados por la [clase](#) JCheckBox, cuyos principales métodos son los siguientes:

- **isSelected():** retorna un valor lógico que indica si el usuario seleccionó la opción (verdadero si la opción fue escogida y falso en caso contrario).
- **setSelected(seleccionado):** marca como seleccionado o no el control, dependiendo del valor lógico del [parámetro](#).

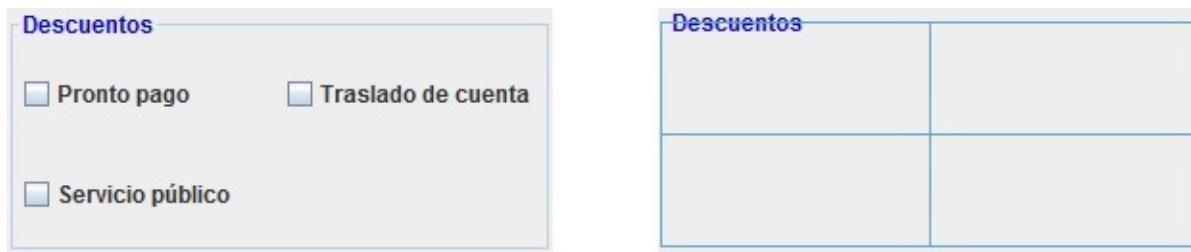
Por convención utilizaremos el prefijo "cb" para los nombres de los atributos que representen este tipo de componentes gráficos (JCheckBox).

Ejemplo 7

Objetivo: Mostrar el manejo de los componentes de selección de opciones.

Este ejemplo muestra el manejo del componente JCheckBox en el contexto del caso de estudio. Vamos a utilizar tres objetos de esa [clase](#) en el segundo de los paneles, para que el usuario pueda escoger los descuentos a los que tiene derecho. Comenzamos mostrando

la imagen esperada en la interfaz y el [distribuidor gráfico](#) instalado sobre la división, de manera que sea claro el orden en el que los componentes se deben agregar.



```
public class PanelDescuentos extends JPanel
{
    private JCheckBox cbPPago;
    private JCheckBox cbSPublico;
    private JCheckBox cbTCuenta;
    ...
}
```

- Declaración como atributos de los tres componentes gráficos de selección de opciones.

```
public PanelDescuentos( )
{
    ...
    cbPPago = new JCheckBox( "Pronto pago" );
    cbSPublico = new JCheckBox( "Servicio público" );
    cbTCuenta = new JCheckBox( "Traslado de cuenta" );

    add( cbPPago );
    add( cbTCuenta );
    add( cbSPublico );
}
```

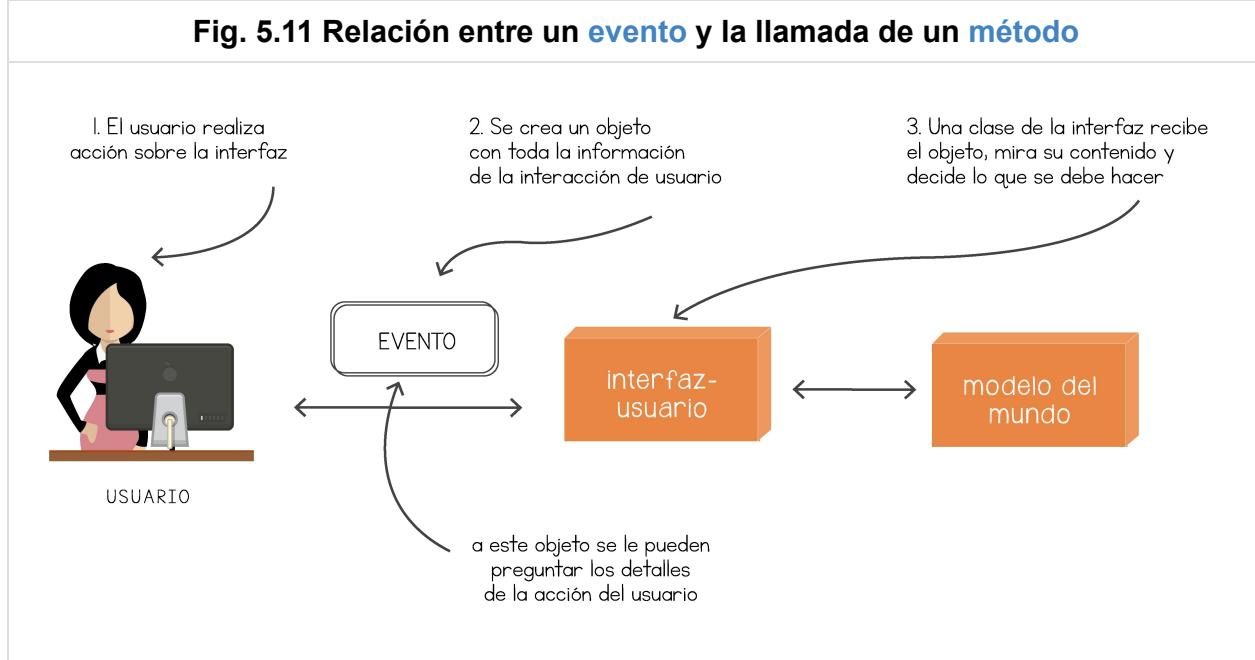
En el constructor de la [clase](#) se crean inicialmente los objetos, pasando como [parámetro](#) el nombre que se debe asociar con cada opción.

Luego se agregan los objetos al [panel](#), siguiendo el orden pedido por el distribuidor (por filas, de arriba hacia abajo y de izquierda a derecha).

6. Elementos de Interacción

Existen muchos mecanismos de interacción mediante los cuales el usuario puede expresar sus órdenes a la interfaz. Desde hacer clic en algún punto de la [ventana](#), hasta arrastrar un ícono de una zona a otra de un [panel](#). Todas estas acciones del usuario son convertidas en eventos en Java y son manipuladas mediante objetos. Esto quiere decir que cada vez que el usuario hace algo sobre la [ventana](#) del programa, esta acción se convierte en un [objeto](#) (llamado un [evento](#)) que contiene toda la información para describir lo que el usuario hizo. De esta manera, podemos tomar dicho [objeto](#), estudiar su contenido y hacer que el programa reaccione como se supone debe hacerlo, de acuerdo con la acción del usuario. Por ejemplo, si en el [evento](#) aparece que el usuario oprimió un botón, debemos ejecutar la respectiva reacción, que puede incluir cambiar o consultar algo en el modelo del mundo. La [figura 5.11](#) ilustra la idea anterior.

Fig. 5.11 Relación entre un [evento](#) y la llamada de un [método](#)



En este libro únicamente estudiamos la interacción usando botones, posiblemente el mecanismo más simple que existe para que el usuario exprese sus órdenes. Dichos botones son componentes gráficos que pertenecen a la [clase JButton](#). Estos componentes se declaran y agregan a los paneles como cualquier otro, tal como se muestra en el ejemplo 8.

Ejemplo 8

Objetivo: Mostrar la manera de agregar botones a un [panel](#).

En este ejemplo se presenta la manera de declarar y agregar los botones [panel](#) del caso de estudio usado para las búsquedas. Una vez instalado como aparece en el ejemplo, los botones se pueden oprimir, pero no reaccionan de ninguna manera.

Ese es el tema que sigue: ¿cómo asociar una reacción con un [evento](#) de un botón? En la imagen que se presenta a continuación aparece la visualización esperada del [panel](#).



```

public class PanelBusquedas extends JPanel
{
    //-----
    // Atributos
    //-----
    private JTextField txtLinea;

    /**
     * Campo de texto para la marca.
     */
    private JTextField txtMarca;
    private JButton btnBuscarLinea;
    private JButton btnBuscarMarca;
    private JButton btnBuscarCaro;

    //-----
    // Constructor
    //-----
    public PanelBusquedas ( )
    {
        ...

        txtLinea = new JTextField( );
        add( txtLinea );

        btnBuscarLinea = new JButton( BUSCAR_POR_LINEA );
        add( btnBuscarLinea );

        txtMarca = new JTextField( );
        add( txtMarca );

        btnBuscarMarca = new JButton( BUSCAR_POR_MARCA );
        add( btnBuscarMarca );

        add( new JLabel( ) );

        btnBuscarCaro = new JButton( BUSCAR_MAS_CARO );
        add( btnBuscarCaro );
    }
}

```

- Se declara un **atributo** por cada componente botón: dos **zona de texto** para ingresar lo que se desea buscar y tres botones.
- El prefijo utilizado para los botones en este ejemplo es "btn".
- Las instrucciones que aquí se muestran deben venir después de aquellas que asocian el **distribuidor gráfico** y el borde.
- Se crean los objetos que implementan los componentes gráficos y se inicializan.
- Al crear un botón, se define la **etiqueta** que va a aparecer sobre él.
- Fíjese que agregamos una **etiqueta "vacía"** para obtener la visualización deseada.

Hay tres pasos que se deben seguir para decidir la manera de manejar un **evento** con un botón de la interfaz, los cuales se explican a continuación:

- Decidir el nombre del **evento**. A los eventos de los botones se les asocia un nombre por medio del cual se van a poder identificar más adelante. El nombre es una cadena de caracteres y es muy conveniente definir dicha cadena como una **constante**. Para el caso de estudio, los nombres de los eventos se asocian de la siguiente manera con los dos botones:

```
public class PanelBusquedas extends JPanel
{
    //-----
    // Constantes
    //-----
    public final static String BUSCAR_POR_LINEA = "Buscar por línea";
    public final static String BUSCAR_POR_MARCA = "Buscar por marca";
    public final static String BUSCAR_MAS_CARO = "Buscar vehículo más Caro";

    public PanelBusquedas ( )
    {
        ...
        btnBuscarLinea.setActionCommand( BUSCAR_POR_LINEA );
        btnBuscarMarca.setActionCommand( BUSCAR_POR_MARCA );
        btnBuscarCaro.setActionCommand( BUSCAR_MAS_CARO );
    }
}
```

- Implementar el **método** que va a atender el **evento**. Para atender el **evento**, el **panel** que contiene el botón debe agregar una declaración en el encabezado de la **clase** (`implements ActionListener`) e implementar un **método** especial llamado `actionPerformed`, que recibe como **parámetro** el **evento** ocurrido en el **panel**. Dicho **evento** es un **objeto** de la **clase** `ActionEvent`. Estos puntos se ilustran en el siguiente código, en el cual se muestra también la manera de obtener el nombre del **evento** ocurrido, a partir del **objeto** que lo representa.

```

public class PanelBusquedas extends JPanel implements ActionListener
{
    public void actionPerformed( ActionEvent pEvento )
    {
        String comando = evento.getActionCommand( );

        if( comando.equals( BUSCAR_MAS_CARO ) )
        {
            // Reacción al evento de BUSCAR_MAS_CARO
        }
        else if( comando.equals( BUSCAR_POR_LINEA ) )
        {
            // Reacción al evento de BUSCAR_POR_LINEA
        }
        else if( comando.equals( BUSCAR_POR_MARCA ) )
        {
            // Reacción al evento de BUSCAR_POR_MARCA
        }
    }
}

```

- La **clase** del **panel** debe incluir en su encabezado la declaración **implements ActionListener**.
- Esa misma **clase** debe implementar un **método** con la **signatura** planteada en el ejemplo.
- Con el **método** **getActionCommand** podemos saber el nombre del **evento** ocurrido.

Cada vez que el usuario oprime un botón en un **panel**, se ejecuta su **método actionPerformed**. El contenido exacto de dicho **método** se estudiará en una sección posterior, puesto que hay decisiones de nivel de **arquitectura** que todavía no hemos tomado. Pero a grandes rasgos se puede decir que ese **método** debe utilizar el nombre del **evento** ocurrido para decidir la acción que debe tomar.

- Declarar que el **panel** es el responsable de atender los eventos de sus botones. Para esto se utiliza el **método addMouseListener**, pasando como referencia el **panel**. Puesto que esto se debe ejecutar en el constructor del mismo **panel**, utilizamos la **variable** **this** que provee el lenguaje Java para hacer referencia al **objeto** que está ejecutando un **método**. De esta manera podemos decir dentro del constructor del **panel** que quien va a atender los eventos del botón es el **panel** mismo. El código es el siguiente:

```

public class PanelBusquedas extends JPanel implements ActionListener
{
    public PanelResultados( )
    {
        ...
        btnBuscarLinea.addActionListener( this );
        btnBuscarMarca.addActionListener( this );
        btnBuscarCaro.addActionListener( this );
    }
}

```

- Con el **método** `addActionListener`, el botón declara que es el **panel** quien va a atender sus eventos.
- La **variable** `this` siempre referencia al **objeto** que está ejecutando un **método**.

Con eso completamos el manejo de eventos relacionados con los botones y sólo queda pendiente el cuerpo exacto del **método** que atiende los eventos.

A continuación mostramos el contenido completo de la **clase** `PanelBusquedas` , para dar una visión global de su contenido:

```

public class PanelBusquedas extends JPanel implements ActionListener
{
    // -----
    // Constantes
    // -----
    public final static String BUSCAR_POR_LINEA = "Buscar por línea";
    public final static String BUSCAR_POR_MARCA = "Buscar por marca";
    public final static String BUSCAR_MAS_CARO = "Buscar vehículo más Caro";

    // -----
    // Atributos de la interfaz
    // -----
    private JTextField txtLinea;
    private JTextField txtMarca;
    private JButton btnBuscarLinea;
    private JButton btnBuscarMarca;
    private JButton btnBuscarCaro;

    // -----
    // Constructores
    // -----
    public PanelBusquedas( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        setLayout( new GridLayout( 3, 2 ) );
        TitledBorder border = new TitledBorder( "Búsquedas" );

```

```
border.setTitleColor( Color.BLUE );
setBorder( border );
setBorder( border );

txtLinea = new JTextField( );
add( txtLinea );

btnBuscarLinea = new JButton( BUSCAR_POR_LINEA );
btnBuscarLinea.addActionListener( this );
btnBuscarLinea.setActionCommand( BUSCAR_POR_LINEA );
add( btnBuscarLinea );

txtMarca = new JTextField( );
add( txtMarca );

btnBuscarMarca = new JButton( BUSCAR_POR_MARCA );
btnBuscarMarca.addActionListener( this );
btnBuscarMarca.setActionCommand( BUSCAR_POR_MARCA );
add( btnBuscarMarca );

add( new JLabel( ) );

btnBuscarCaro = new JButton( BUSCAR_MAS_CARO );
btnBuscarCaro.addActionListener( this );
btnBuscarCaro.setActionCommand( BUSCAR_MAS_CARO );
add( btnBuscarCaro );

}

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

    if( comando.equals( BUSCAR_MAS_CARO ) )
    {
        // Por definir

    }
    else if( comando.equals( BUSCAR_POR_LINEA ) )
    {
        // Por definir

    }
    else if( comando.equals( BUSCAR_POR_MARCA ) )
    {
        // Por definir
    }
}
```

Si una [clase](#) incluye la declaración implements ActionListener y no implementa el [método actionPerformed](#) (o si lo implementa con otra [signatura](#)), se obtiene el siguiente error de compilación:

| Class must implement the inherited abstract method ActionListener.
| actionPerformed(ActionEvent)

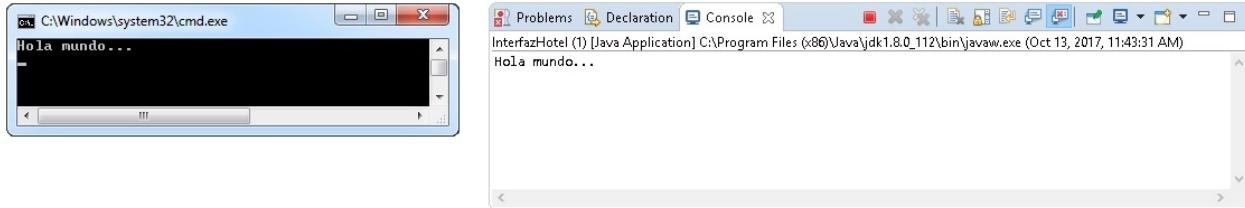
7. Mensajes al Usuario y Lectura Simple de Datos

7.1. Mensajes en la Consola

Para presentar un mensaje en la [ventana](#) de comandos del sistema operativo, se utiliza la instrucción `System.out.println(cadena)`. Es poco usual enviarle mensajes al usuario a esa [ventana](#), pero en algunos casos (errores fatales, por ejemplo), esto es indispensable.

Si el programa se está ejecutando en un [ambiente de desarrollo](#) como Eclipse, los mensajes aparecerán en una [ventana](#) especial llamada consola.

```
System.out.println( "Hola mundo..." );
```



Se puede utilizar esta instrucción, en cualquier [clase](#) de la interfaz, para enviar un mensaje al usuario a la [ventana](#) de comandos del sistema operativo.

Si durante la ejecución de un programa se lanza una [excepción](#) que no es atrapada por ninguna [clase](#) de la interfaz, la acción por defecto es generar una secuencia de mensajes en la [ventana](#) de comandos, con información relativa al error.

7.2. Mensajes en una Ventana

El [paquete](#) swing incluye una [clase](#) JOptionPane que, entre sus múltiples usos, tiene un [método](#) para enviarle mensajes al usuario en una pequeña [ventana](#) emergente. Esto es muy útil en caso de error en las entradas del usuario o con el fin de mostrarle un resultado puntual de una consulta. La sintaxis de uso es la que se muestra en el ejemplo 9.

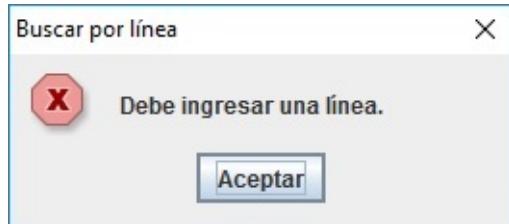
Ejemplo 9

Objetivo: Mostrar la manera de presentar un mensaje a un usuario, usando una [ventana](#) simple de diálogo.

Este ejemplo muestra la manera de enviarle mensajes al usuario, abriendo una nueva [ventana](#) y esperando hasta que el usuario oprima el botón para continuar. En cada imagen aparece la [ventana](#) que se va a mostrar al usuario y, debajo, la instrucción que ordena hacerlo. Esta instrucción debe ir dentro de un [método](#) de un [panel](#).

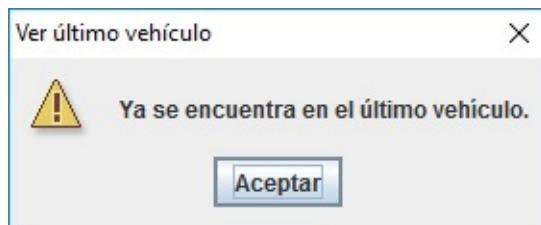
Mensaje de error:

```
JOptionPane.showMessageDialog( principal, "Debe ingresar una línea.", "Buscar por línea", JOptionPane.ERROR_MESSAGE );
```



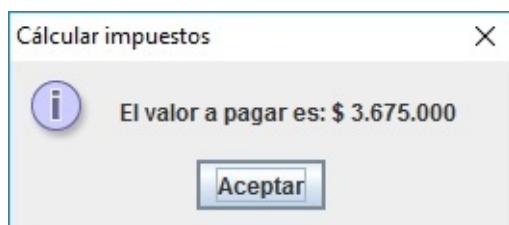
Mensaje de advertencia:

```
JOptionPane.showMessageDialog( this , "Ya se encuentra en el Último vehículo.", "Ver último vehículo" , JOptionPane.WARNING_MESSAGE );
```



Mensaje de información:

```
JOptionPane.showMessageDialog( this , "El valor a pagar es: $3.675.000" , "Cálculo de Impuestos" , JOptionPane.INFORMATION_MESSAGE );
```



7.3. Pedir Información al Usuario

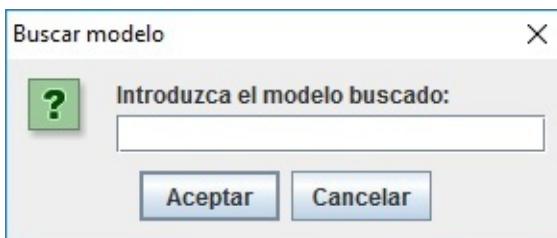
Cuando la información que se necesita como entrada de un [requerimiento funcional](#) es muy sencilla (un nombre o un valor numérico), se puede utilizar un [método de la clase JOptionPane](#) que abre una [ventana](#) de diálogo y luego retorna la cadena tecleada por el usuario. Su uso se ilustra en el ejemplo 10. Si la información que se necesita de parte del usuario es más compleja, se debe utilizar un cuadro de diálogo más elaborado, en el cual irían los componentes gráficos necesarios para recoger la información.

Ejemplo 10

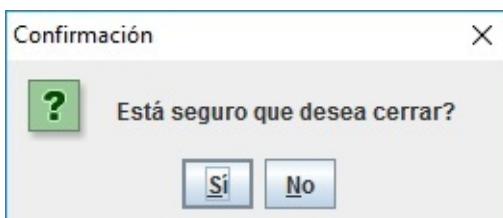
Objetivo: Mostrar la manera de pedir información simple al usuario.

Este ejemplo muestra la manera de pedir al usuario que teclee alguna información en una nueva [ventana](#) de diálogo. Al igual que en el ejemplo anterior, en cada imagen aparece la visualización de la [ventana](#) y, debajo, el código en Java que la presenta y que recupera el valor tecleado. Esta instrucción debe ir dentro de un [método](#) de un [panel](#) o [ventana](#).

```
String strModelo = JOptionPane.showInputDialog( this , "Introduzca el modelo buscado:"  
, "Buscar modelo", JOptionPane.QUESTION_MESSAGE );  
  
if( strModelo != null )  
{  
    // el usuario tecleó algo  
}
```



```
int resp = JOptionPane.showConfirmDialog( this , "Está seguro que desea cerrar?" , "Co  
nfiración" , JOptionPane.YES_NO_OPTION );  
  
if( resp == JOptionPane.YES_OPTION )  
{  
    // el usuario seleccionó Sí  
}
```



7.4. Validación y formateo de datos

Cuando el usuario ingresa alguna información, la interfaz tiene muchas veces la **responsabilidad** de convertirla al formato y al tipo adecuados para poder manipularla (por ejemplo, convertir una cadena en una **variable** de tipo entero o pasar una cadena a minúsculas). De la misma manera, si el usuario tecleó un contenido que no corresponde a lo esperado (ingresó una letra cuando se esperaba un número), la interfaz debe advertir al usuario de su error. Vamos entonces por partes para ver cómo manejar cada uno de los casos.

Para convertir una cadena de caracteres (que sólo contenga dígitos) en un número, se utiliza el **método** de la **clase** Integer llamado parseInt, usando la sintaxis que se muestra a continuación. Dicho **método** lanza una **excepción** cuando la cadena que se pasa como **parámetro** no se puede convertir en un valor entero. En la sección de recuperación de la **excepción** (sección catch) podría incluirse un mensaje al usuario.

```
String strModelo = JOptionPane.showInputDialog( this , "Introduzca el modelo buscado:" ,
, "Buscar por modelo", JOptionPane.QUESTION_MESSAGE );

if( strModelo != null )
{
    try
    {
        int nModelo = Integer.parseInt( strModelo );
    }
    catch( Exception e )
    {
        JOptionPane.showMessageDialog( principal, "Debe ingresar un valor numérico.",
"Buscar por modelo", JOptionPane.ERROR_MESSAGE );
    }
}
```

- Suponga que queremos convertir el modelo del vehículo que ingresó el usuario en el valor entero correspondiente (si ingresó la cadena "2016", queremos obtener el valor entero 2016).
- Lo primero que hacemos es tomar la cadena de caracteres ingresada por el usuario en el JInputDialog.
- Luego intentamos convertir dicha cadena en el valor entero correspondiente.
- En este ejemplo, si se produce una **excepción**, le presentamos un mensaje al usuario indicándolo .
- Este esquema de conversión es típico de las interfaces gráficas, puesto que no estamos seguros del **tipo de datos** de lo que ingresó el usuario y, en algunos casos, es conveniente verificarlo antes de continuar.

La **clase** String, por su parte, nos ofrece los siguientes métodos para transformar la cadena tecleada por el usuario:

- **toLowerCase()**: convierte todos los elementos de una cadena de caracteres a minúsculas.
- **toUpperCase()**: convierte todos los elementos de una cadena de caracteres a mayúsculas.
- **trim()**: elimina todos los caracteres en blanco del comienzo y el final de la cadena.

En la siguiente tabla se muestran algunos ejemplos del uso de los métodos anteriores:

String ejemplo = " La Casa ";	// valor inicial de la cadena
String minusculas = ejemplo.toLowerCase();	minusculas.equals(" la casa ")
String mayusculas = ejemplo.toUpperCase();	mayusculas.equals(" LA CASA ")
String sinBlancos = ejemplo.trim();	sinBlancos.equals("La Casa")

8. Arquitectura y Distribución de Responsabilidades

8.1. ¿Por dónde Comienza la Ejecución de un Programa?

Un [método](#) que no hemos mencionado hasta ahora y que, sin embargo, es el punto por donde comienza siempre la ejecución de un programa, es el [método main\(\)](#). Este [método](#) se implementa en la [clase](#) de la [ventana](#) principal del programa y tiene la sintaxis que se muestra a continuación. Su principal tarea es crear una instancia de la [ventana](#) y hacerla visible en la pantalla.

```
-----  
// Programa principal  
-----  
public static void main( String[] args )  
{  
    try  
    {  
        InterfazImpuestosCarro vent = new InterfazImpuestosCarro( );  
        vent.setVisible( true );  
    }  
    catch( Exception e )  
    {  
        JOptionPane.showMessageDialog( null, e.getMessage(), "Calculador impuestos", JOptionPane.ERROR_MESSAGE );  
    }  
}
```

- Este [método](#) debe ir en la [clase](#) que implementa la [ventana](#) principal. Su objetivo es establecer la manera de comenzar la ejecución del programa, creando una instancia de la [ventana](#) y haciéndola visible.
- La [signatura](#) del [método](#) debe ser idéntica a la que aparece en el ejemplo.

8.2. ¿Quién Crea el Modelo del Mundo?

La [responsabilidad](#) de crear el modelo del mundo (los objetos que lo van a representar) es de la interfaz. En la [arquitectura](#) que presentamos en este libro, nosotros asignamos esta [responsabilidad](#) al constructor de la [ventana](#) principal. Allí se deben realizar todas las

acciones necesarias para que los objetos del modelo del mundo (uno o varios) sean creados, inicializados y almacenados en atributos de dicha [clase](#). A continuación se muestra, para el caso de estudio, la creación del [objeto](#) que representa el calculador de impuestos.

```
public class InterfazImpuestosCarro extends JFrame
{
    //-----
    // Atributos
    //-----
    private CalculadorImpuestos calculador;

    private PanelVehiculo panelVehiculo;
    private PanelDescuentos panelDescuentos;
    private PanelOpciones panelOpciones;
    private PanelBusquedas panelConsultas;

    //-----
    // Constructor
    //-----
    public InterfazImpuestosCarro( ) throws Exception
    {
        calculador = new CalculadorImpuestos( );
        ...
    }
}
```

- En este caso la creación es simple, pues el constructor del calculador de impuestos tiene la [responsabilidad](#) de abrir los archivos con la información y crear los objetos necesarios para representarla.
- Definimos un [atributo](#) de la [clase](#) CalculadorImpuestos, y allí guardamos la [asociación](#) que nos va a permitir "hablar" con el modelo del mundo (ver diagrama de clases).
- En el constructor dejamos pasar las excepciones generadas en la construcción del modelo del mundo. Dejamos al programa principal la [responsabilidad](#) de atraparlas y enviarle el mensaje respectivo al usuario.
- No poder construir el modelo del mundo (p.ej. no poder abrir los archivos con los valores que utiliza la calculadora) lo consideramos un error fatal, y por esa razón no existe ninguna manera de recuperarse.
- Se puede ver la [ventana](#) principal como la [clase](#) que va a coordinar el trabajo entre los paneles y el modelo del mundo.

8.3. ¿Qué Métodos Debe Tener un Panel?

Una pregunta que debemos responder en este punto es cuáles son los métodos que debe tener un **panel**, ya que hasta este momento sólo tenemos un **método** constructor y un **método** para atender los eventos. La respuesta es que los paneles tienen dos grandes responsabilidades además de las ya estudiadas:

1. Proveer los métodos indispensables para permitir el acceso a la información tecleada por el usuario. Considere la **interfaz de usuario** del caso de estudio, en la cual en el primer **panel** está la información del vehículo. Puesto que el tercer **panel** va a necesitar esta información para poder calcular los impuestos, es **responsabilidad** del **panel** que tiene la información proveer un conjunto de métodos que garantice que aquellos que requieran la información puedan tener acceso a ella. Eso no quiere decir que haya que construir un **método** por cada **zona de texto**. Lo que quiere decir es que se debe establecer qué información se necesita manejar desde fuera del **panel** y crear los métodos respectivos. Lo que quiere decir es que se debe establecer qué información se necesita manejar desde fuera del **panel** y crear los métodos respectivos. El programador debe decidir si estos métodos son responsables de hacer las conversiones o si esta labor se deja a aquellos que van a utilizar la información. En el ejemplo del **panel** para hacer las búsquedas de vehículos, tendremos dos métodos de acceso a la información, los cuales no hacen ningún tipo de conversión: `darMarca()` y `darLinea()`. El ejemplo 11 ilustra esta **responsabilidad**.
2. Proveer los métodos para refrescar la información presentada en el **panel**. Si en un **panel** se presenta información que depende del estado del modelo del mundo, debemos implementar los servicios necesarios para poder actualizarla. Por ejemplo, en el **panel** de información del vehículo, debemos tener un **método** que pueda modificar la información del vehículo actual. Estos métodos se denominan de **refresco** y su objetivo es permitir actualizar el contenido de los componentes gráficos del **panel**. Los ejemplo 11 y 12 ilustran esta **responsabilidad**.

Ejemplo 11

Objetivo: Mostrar los métodos que debe implementar un **panel**, para prestar servicios a los demás elementos de la interfaz.

Este ejemplo muestra los métodos de acceso a la información y de refresco para la **clase** que implementa el **panel** que permite hacer búsquedas de vehículos.

```
public class PanelVehiculo extends JPanel implements ActionListener
{
    //-----
    // Métodos de refresco
    //-----

    public void refrescar()
    {
        txtLinea.setText("");
        txtMarca.setText("");
    }

    //-----
    // Métodos de acceso a la información
    //-----
    public String darLinea()
    {
        return txtLinea.getText();
    }

    public String darMarca()
    {
        return txtMarca.getText();
    }
}
```

La **clase** tiene un **método** de refresco que permite eliminar la información de las etiquetas. De esta manera, utilizando el **método** `setText()`, se elimina el texto que el usuario escribió en los campos de texto

Ejemplo 12

Objetivo: Mostrar un **método** de refresco de la información de un **panel**.

Este ejemplo muestra un **método** de refresco para la **clase** que implementa el **panel** con la información del vehículo.

```

public class PanelVehiculo extends JPanel implements ActionListener
{
    //-----
    // Métodos de refresco
    //-----
    public void actualizar( String pMarca, String pLinea, String pAnio, double pPrecio
    , String pRutaImagen)
    {
        txtMarca.setText( pMarca );
        txtLinea.setText( pLinea );
        txtModelo.setText( pAnio );
        DecimalFormat df = ( DecimalFormat )NumberFormat.getInstance( );
        df.applyPattern( "$ ###,###.##" );
        txtValor.setText( df.format( pPrecio ) );
        labImagen.setIcon( new ImageIcon( new ImageIcon( "./data/imagenes/" + pRutaIma
gen ).getImage( ).getScaledInstance( 280, 170, Image.SCALE_DEFAULT ) ) );
    }
}

```

La **clase** tiene un **método** de refresco que permite cambiar la información del vehículo. De esta manera, utilizando el **método** `setText()` actualiza la información de los campos de texto y con el **método** `setIcon()` cambia la imagen mostrada en la **etiqueta** `labImagen`.

8.4. ¿Quién se Encarga de Hacer Qué?

Si recapitulamos lo que llevamos hasta este momento, podemos decir que ya sabemos:

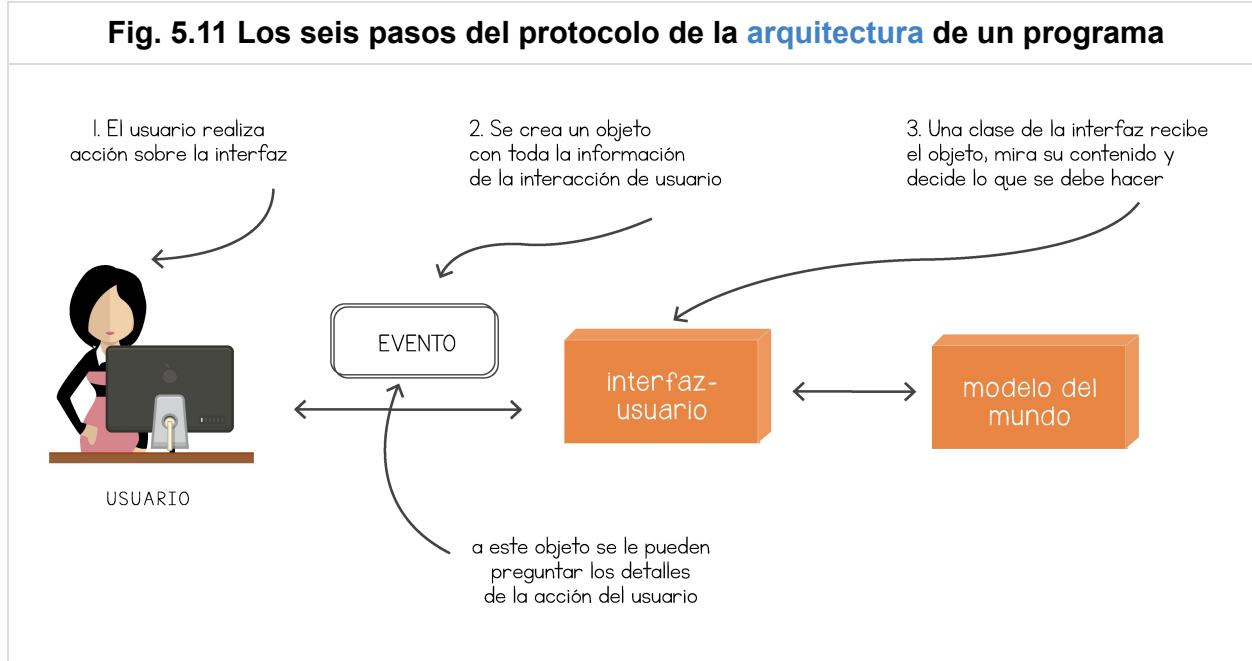
- Crear la **ventana** de la interfaz, con sus paneles y sus componentes gráficos.
- Obtener de los componentes gráficos la información suministrada por el usuario.
- Asociar un nombre con el **evento** que genera cada botón y escribir el **método** que lo atiende.
- Convertir la información que teclea el usuario a otros tipos de datos.
- Presentar al usuario mensajes con información simple.
- Escribir el **método** que inicia la ejecución del programa.
- Crear el modelo del mundo en el constructor de la **ventana** y guardar una **asociación** hacia él.
- Escribir en los paneles los métodos de servicio (refresco y acceso a la información).

Lo único que nos falta en este momento es definir la manera de utilizar todo lo anterior para implementar los requerimientos funcionales. Para esto, debemos definir las responsabilidades y compromisos de cada uno de los participantes, de manera que siempre sepamos quién debe hacer qué, y en qué orden. A esto lo denominaremos el **protocolo** de la **arquitectura**. Sobre este punto debemos decir que hay muchas soluciones posibles y que la **arquitectura** que utilizamos a lo largo de este libro es sólo una manera de estructurar y

repartir las responsabilidades. Tiene la ventaja de facilitar la localización de cada uno de los componentes del programa, aumentando su claridad y simplificando su mantenimiento, dos puntos fundamentales a la hora de escribir un [programa de computador](#).

La [arquitectura](#) que usamos se basa en la idea de que los requerimientos funcionales se implementan en la [ventana](#) principal (un [método](#) por requerimiento) y que es allí donde se coordinan todas las acciones, tanto de los elementos que se encuentran en los paneles como de los elementos del modelo del mundo. En la [figura 5.11](#) aparece el protocolo con los seis pasos básicos para reaccionar a un [evento](#) generado por el usuario.

Fig. 5.11 Los seis pasos del protocolo de la arquitectura de un programa



Veamos ahora paso por paso el protocolo, para explicar la figura anterior. Los números asociados con las flechas indican el orden en el que las acciones se llevan a cabo.

Paso 1: el usuario genera un [evento](#) oprimiendo un botón en uno de los paneles de la interfaz. Dicho [evento](#) se convierte en un [objeto](#) que lleva toda la información relacionada con la acción del usuario.

- Debe reaccionar el [panel](#) que contiene el botón.



Paso 2: el [panel](#) reacciona al [evento](#) con su [método](#) `actionPerformed`, el cual debe solicitar a la [ventana](#) principal que ejecute el [requerimiento funcional](#) pedido por el usuario.

- El [panel](#) debe pasarle toda la información que tiene en su interior y que se necesita como entrada del [requerimiento funcional](#).
- Si hay necesidad de convertir la información ingresada por el usuario a un tipo específico de datos, es [responsabilidad](#) del [panel](#) hacerlo.
- Un [requerimiento funcional](#) se implementa como un [método](#) en la [ventana](#).



Paso 3: la [ventana](#) principal completa la información necesaria para poder cumplir con el [requerimiento funcional](#), pidiéndola a los demás paneles.

- Puesto que el [método](#) que implementa el [requerimiento funcional](#) es responsable de que se cumplan las precondiciones de los métodos del modelo del mundo, en este punto debe hacer todas las verificaciones necesarias y, en caso de que surja un problema, puede cancelar la reacción y notificar al usuario de lo sucedido.
- Para realizar este paso, desde el [método](#) que implementa el [requerimiento funcional](#) se invocan los métodos de acceso a la información de los demás paneles.



Paso 4: se pide al modelo del mundo que haga una modificación (basada en los valores ingresados por el usuario) o que calcule algún valor.

- Se utiliza en este paso la [asociación](#) (o las asociaciones) que tiene la interfaz hacia el modelo del mundo, para invocar el o los métodos que van a ayudar a implementar el [requerimiento funcional](#). Cualquier [excepción](#) lanzada por los métodos del modelo del mundo debería ser atrapada en este punto.
- Si sólo se está pidiendo al modelo del mundo que calcule un valor (por ejemplo, calcular el avalúo del vehículo), al final de este paso ya se tiene toda la información necesaria para iniciar el [proceso de refresco](#).
- Si se pidió una modificación del modelo del mundo, se debe ejecutar el paso 5.



Paso 5: si en el paso anterior se pidió una modificación al modelo del mundo, se llaman aquí los métodos que retornan los nuevos valores que se deben presentar.

Para saber qué métodos invocar, se debe establecer qué partes de la información mostrada al usuario deben ser recalculadas.



Paso 6: se pide a todos los paneles que tienen información que pudo haber cambiado que actualicen sus valores.

Para eso se utilizan los métodos de refresco implementados por los paneles.

Hay muchos modelos distintos para mantener la información de la interfaz sincronizada con el estado del modelo del mundo. El que se plantea aquí puede ser muy ineficiente en problemas grandes, por lo que insistimos en que esta [arquitectura](#) sólo debe ser utilizada en problemas pequeños.



8.5. ¿Cómo Hacer que los Paneles Conozcan la Ventana?

De acuerdo con el protocolo antes mencionado, todos los paneles que tengan botones (llamados paneles activos) deben tener una **asociación** hacia la **ventana** principal, de manera que sea posible ejecutar los métodos que implementan los requerimientos funcionales. Esto hace que los constructores de los paneles que tienen botones deban modificar un poco su estructura, tal como se muestra en el ejemplo 12.

```
public class PanelBusquedas extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelBusquedas( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

- Modificación de la **clase** que implementa el **panel** de búsquedas, para incluir una **asociación** a la **ventana** principal.
- En el **atributo** llamado "principal" almacenamos la referencia a la **ventana** principal, recibida como **parámetro**.

```
public class PanelOpciones extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelOpciones ( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

- Modificación de la **clase** que implementa el **panel** de búsquedas para incluir una **asociación** a la **ventana** principal.
- En el constructor de la **ventana**, cuando se cree este **panel**, se debe pasar como **parámetro** una referencia a la **ventana** de la interfaz.

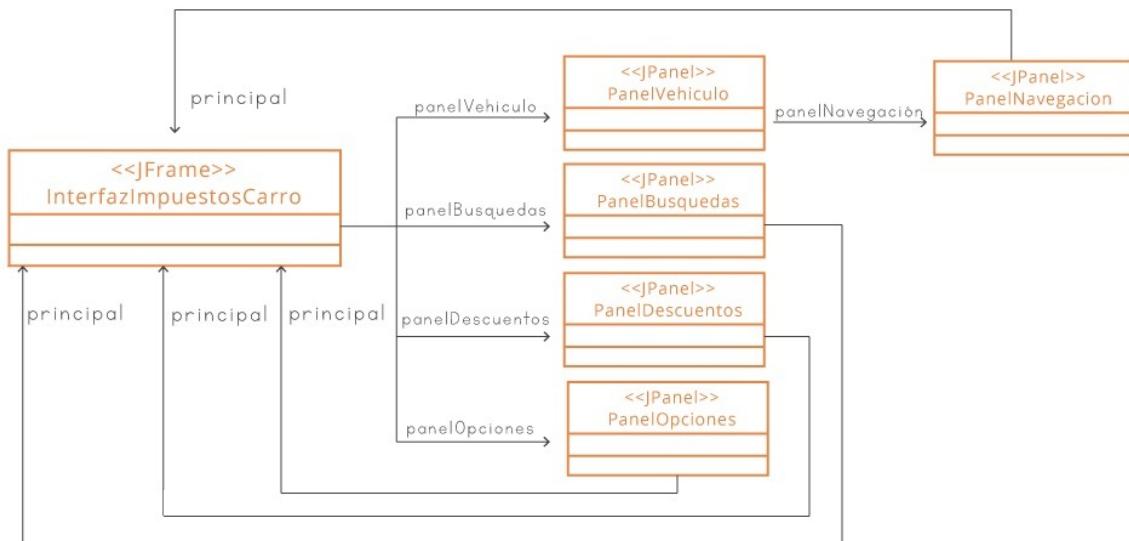
```
public class PanelNavegacion extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelNavegacion ( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

- Modificación de la **clase** que implementa el **panel** de navegación para incluir una **asociación** a la **ventana** principal.
- En el constructor de **PanelVehiculo**, cuando se cree este **panel**, se debe pasar como **parámetro** una referencia a la **ventana** de la interfaz.

```
public class PanelVehiculo extends JPanel implements ActionListener
{
    public PanelVehiculo( InterfazImpuestosCarro pPrincipal )
    {
        add( new PanelNavegacion( pPrincipal ), BorderLayout.SOUTH );
        ...
    }
}
```

- Modificación de la **clase** que implementa **panel** con la información del vehículo, para incluir una **asociación** a la **ventana** principal.
- En este caso no se crea el **atributo** llamado "principal" porque la referencia a la **ventana** principal sólo se utiliza para crear el **panel** de navegación.



8.6. ¿Cómo Implementar los Requerimientos Funcionales?

Lo único que nos hace falta ahora es implementar los métodos de los requerimientos funcionales. Estos métodos deben formar parte de la [clase](#) de la [ventana](#) principal de la interfaz, y tienen como objetivo coordinar los paneles y el modelo del mundo para lograr lo pedido por el cliente. En el ejemplo 13 se muestra la estructura de dichos métodos.

Ejemplo 13

Objetivo: Ilustrar la construcción de los métodos que implementan los requerimientos funcionales.

En este ejemplo se muestran los dos métodos de la [clase](#) `InterfazImpuestosCarro` que implementan los requerimientos funcionales del caso de estudio.

```

public void buscarPorLinea( String pLinea )
{
    // 1
    Vehiculo respuesta = calculador.buscarVehiculoPorLinea( pLinea );
    if( respuesta == null )
    {
        // 2
        JOptionPane.showMessageDialog( this, "No se encontró ningún vehículo de esta
línea", "Buscar por línea", JOptionPane.ERROR_MESSAGE );
    }
    else
    {
        // 3
        panelVehiculo.actualizar( respuesta.darMarca( ), respuesta.darLinea( ), respu
esta.darAnio( ), respuesta.darPrecio( ), respuesta.darRutaImagen( ) );
    }
}

```

- Método de la **ventana** principal que atiende el **requerimiento funcional** de mostrar el vehículo con la línea dada.
- En el paso 1 se le pide al modelo del mundo que busque el vehículo con la línea dada.
- Si no se encontró un vehículo de la línea dada (`respuesta == null`), se muestra un mensaje al usuario indicándolo.
- En caso contrario, se actualiza la información del PanelVehiculo con la del vehículo encontrado, usando el **método** `actualizar` de este **panel**.

```

public void calcularImpuestos( )
{
    // 1
    boolean descProntoPago = panelDescuentos.hayDescuentoProntoPago( );
    boolean descServicioPublico = panelDescuentos.hayDescuentoServicioPublico( );
    boolean descTrasladoCuenta = panelDescuentos.hayDescuentoTrasladoCuenta( );

    // 2
    double pago = calculador.calcularPago( descProntoPago, descServicioPublico, desc
TrasladoCuenta );

    // 3
    DecimalFormat df = ( DecimalFormat )NumberFormat.getInstance( );
    df.applyPattern( "$ ###,###.##" );

    // 4
    JOptionPane.showMessageDialog( this, "El valor a pagar es: " + df.format( pago )
, "Cálcular impuestos", JOptionPane.INFORMATION_MESSAGE );
}

```

- Método de la **ventana** principal que atiende el **requerimiento funcional** de calcular el valor que se debe pagar de impuestos.

- En el paso 1 se pide toda la información de los descuentos que se requiere para calcular el pago.
- En el paso 2 se pide al modelo del mundo que calcule el valor que se debe pagar de impuestos.
- En el paso 3 se crea el formato en el cual se desea visualizar la información.
- En el paso 4 se muestra un mensaje al usuario con el valor que se debe pagar por los impuestos del vehículo actual.

9. Ejecución de un Programa en Java

Para ejecutar un programa en Java es necesario especificar desde la [ventana](#) de comandos del sistema operativo el nombre del [archivo](#) jar que contiene el código compilado del programa y el nombre completo de la [clase](#) principal por la cual debe comenzar la ejecución (la [clase](#) que tiene el [método main](#)). (Si el programa no está empaquetado en un [archivo](#) jar, hay que dar solamente el nombre de la [clase](#) principal.) Para el caso de estudio, el comando de ejecución es el siguiente (en una sola línea):

```
java -classpath ./lib/impuestosCarro.jar uniandes.cupi2.impuestosCarro.interfaz.InterfaceImpuestosCarro
```

Si el computador no encuentra el [archivo](#) jar, o si dentro de éste no encuentra la [clase](#) que se le especificó en el comando de ejecución, aparece en la [ventana](#) de comandos del sistema operativo el error: *java.lang.NoClassDefFoundError*.

10. Hojas de Trabajo

10.1 Hoja de Trabajo Nº 1: Granja de traducciones

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

Enunciado. Lea detenidamente el siguiente enunciado sobre el cual se desarrollará la presente hoja de trabajo.

Se quiere crear una aplicación que ayude a aprender los nombres de los animales de la granja en inglés. Cada vez que se inicia una nueva jugada, aparece el nombre de un animal en inglés, y el usuario debe seleccionar la imagen del animal. Posteriormente, la aplicación muestra cuál era la respuesta correcta. Por cada respuesta correcta, el usuario obtiene 20 puntos.

Se espera que la aplicación permita: (1) iniciar una jugada, (2) seleccionar un animal, (3) visualizar la traducción correcta, (4) visualizar el puntaje del jugador.

La siguiente es la [interfaz de usuario](#) que se quiere construir, en la cual se identifican tres zonas:



Requerimientos funcionales. Identifique y especifique los cuatro requerimientos funcionales de la aplicación.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

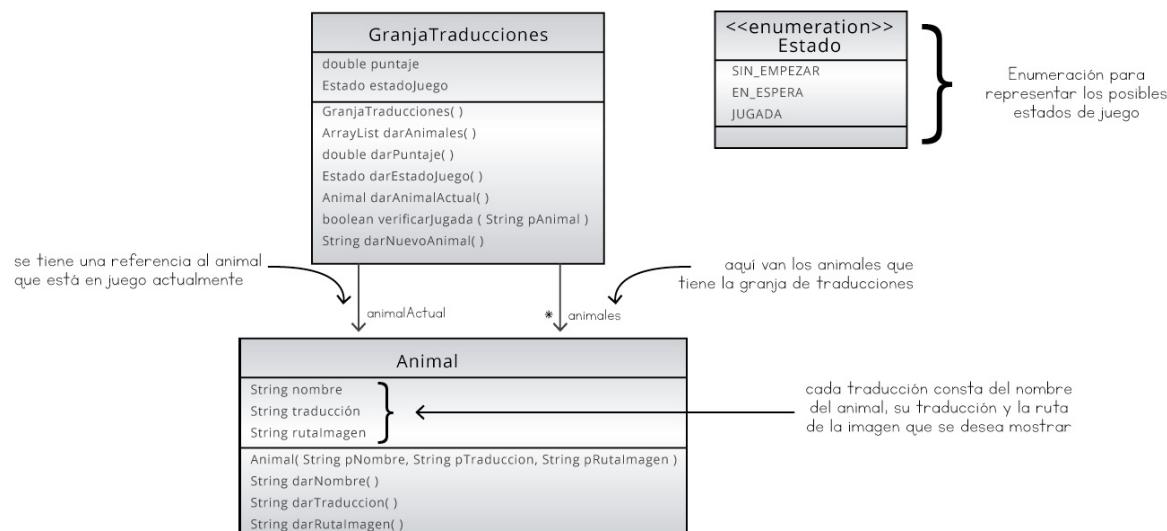
Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

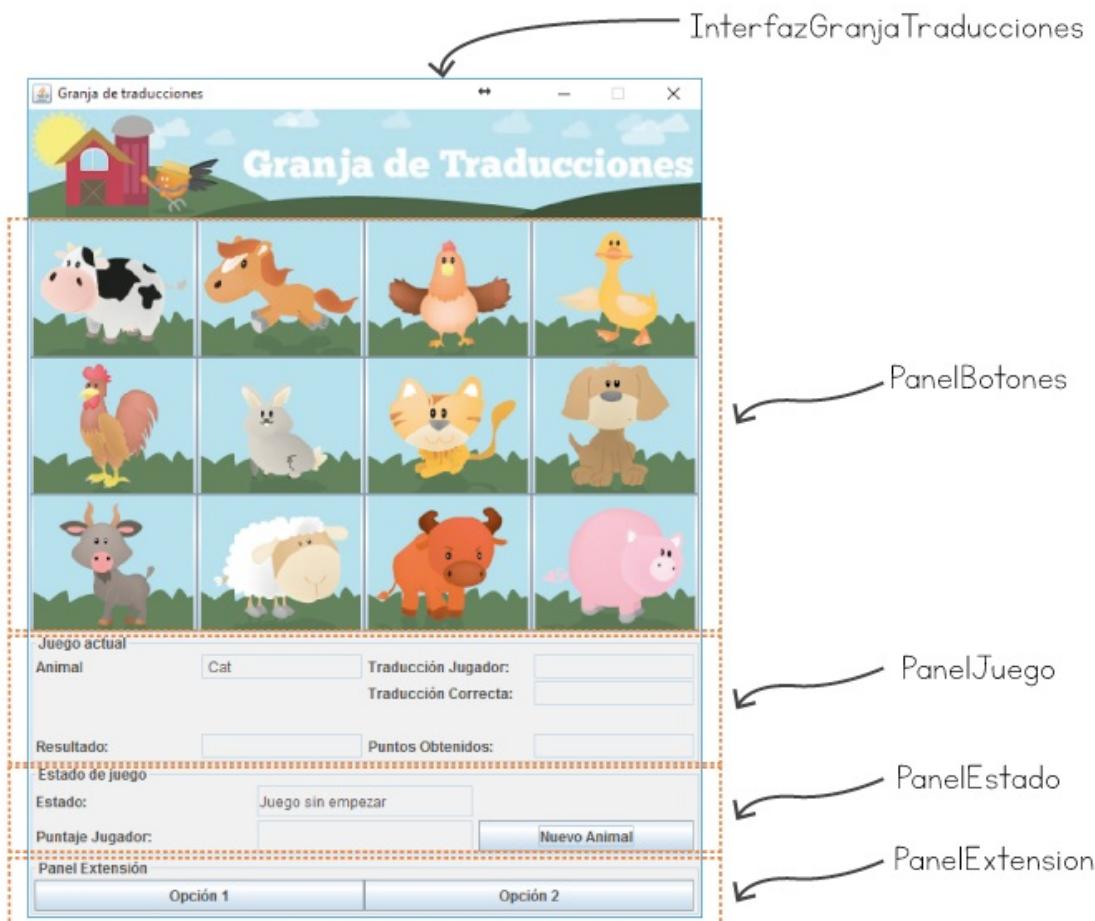
Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Estudie el diagrama de clases que implementa el modelo del mundo y los métodos de cada una de las clases.



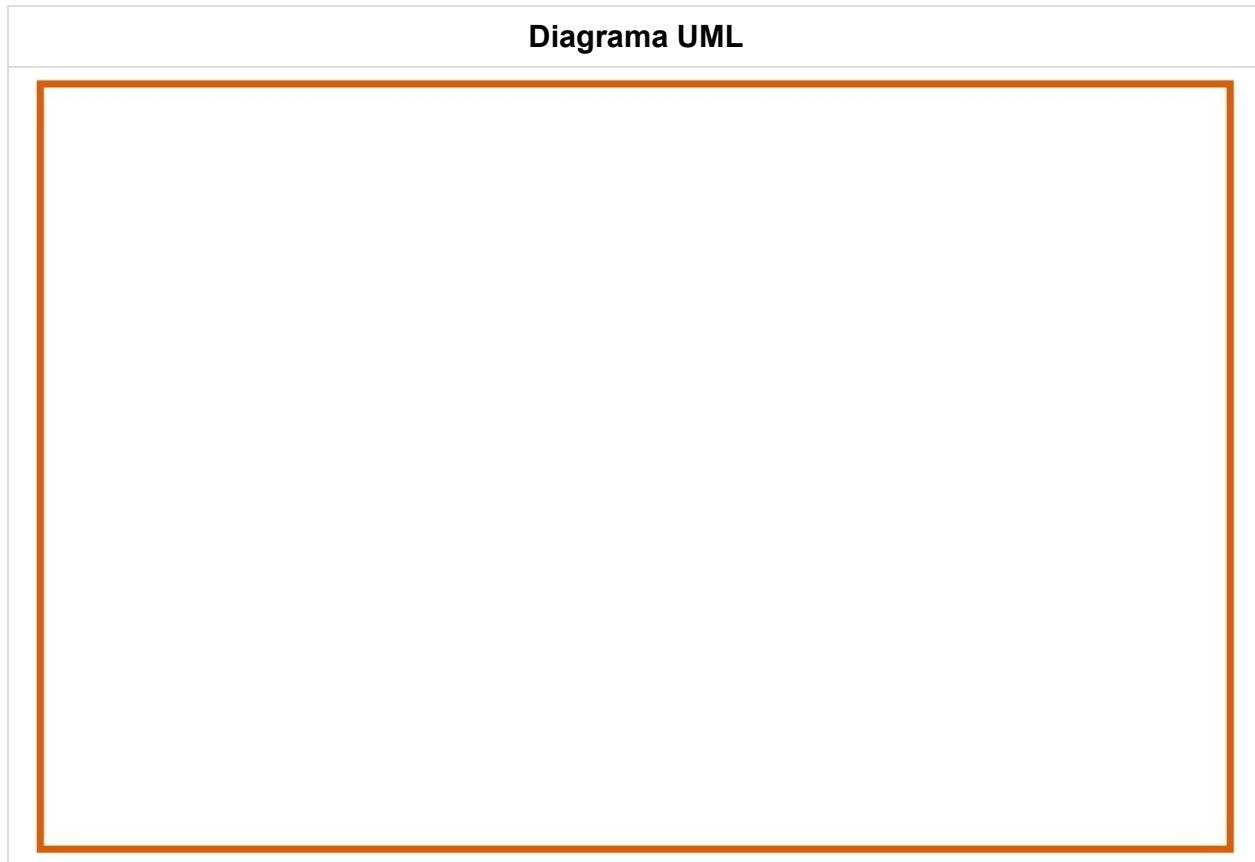
Nombre método	Descripción
GranjaTraducciones()	Crea una granja con sus traducciones.
Animal darAnimales()	Retorna la lista de animales de la granja.
double darPuntaje()	Retorna el puntaje del animal actual.
Estado darEstadoJuego()	Retorna el estado actual del juego.
Animal darAnimalActual()	Retorna el animal actual.
boolean verificarJugada(String pAnimal)	Verifica si la traducción del animal ingresado corresponde con la traducción del animal actual.
String darNuevoAnimal()	Retorna el nombre del nuevo animal seleccionado aleatoriamente.

Interfaz por construir. Observe la estructura de la interfaz que se desea construir y los nombres de las clases que se deben asociar con sus partes.



Arquitectura de la interfaz. Dibuje en UML el diagrama de las clases (sin atributos ni métodos) que conformarán la interfaz. Utilice los estereotipos para indicar si es un JFrame o JPanel. Dibuje también las clases del mundo con las que se relacionan.

Diagrama UML



Construcción de la interfaz. Siga los siguientes pasos para construir la interfaz dada.

1	Cree el paquete para las clases de la interfaz (uniandes.cupi2.granjaTraducciones.interfaz).
2	Cree la clase InterfazGranjaTraducciones como extensión de JFrame . Escriba el método main() , encargado de iniciar la ejecución del programa. Incluya los atributos para representar el modelo del mundo, la imagen del título y los paneles que lo conforman. Defina el tamaño de la ventana como 565x 700. Asocie con la ventana un distribuidor en los bordes . Cree cada uno de los paneles y añádalos adecuadamente a la ventana .
3	Cree la clase PanelBotones como una extensión de la clase JPanel que implementa ActionListener . Declare como atributo una contenedora de botones. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa y la lista de animales. Asocie con el panel un distribuidor en malla de 3 x 4. Cree todos los botones, asociando como comando el nombre del animal, y asignando la imagen asociada al animal. Escriba el esqueleto del método actionPerformed() .
4	Cree la clase PanelJuego como extensión de JPanel . Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Deshabilite la posibilidad de escribir en las zonas de texto. Asocie con el panel un distribuidor en malla de 4 x 4.

5	Cree la clase PanelEstado como una extensión de la clase JPanel que implementa ActionListener . Declare una constante para identificar el evento que va a generar el botón del panel . Declare los atributos para manejar los componentes graficos que se encuentran en su interior. Implemete un constructor que reciba como parámetro una referencia a la venatana del programa. Asocie con el panel un distribuidor en malla de 3 x 3. Escriba el esqueleto del método actionPerformed() .
6	Cree la clase PanelExtension como una extensión de la clase JPanel que implementa ActionListener . Declare una constante para identificar los eventos que van a generar los botones del panel . Declare los atributos para manejar los componentes graficos que se encuentran en su interior. Implemete un constructor que reciba como parámetro una referencia a la venatana del programa. Asocie con el panel un distribuidor en malla de 1 x 2. Escriba el esqueleto del método actionPerformed() .
7	En las clases de los paneles, escriba los métodos de refresco de la información. Incluya en los métodos de refresco el servicio de “borrar” el contenido de los campos una vez que se haya ejecutado una operación
8	En la clase InterfazGranjaTraducciones , escriba un método por cada uno de los requerimientos funcionales. Defina la signatura de manera que reciba como parámetro toda la información de la que dispone el panel que va a hacer la invocación.
9	Complete el método actionPerformed() en las clases PanelBotones , PanelEstado y PanelExtension , haciendo las llamadas respectivas a los métodos de la ventana principal que implementan los requerimientos funcionales.
10	Complete todos los detalles que falten en la interfaz, para obtener la visualización y el funcionamiento descritos en el enunciado. Pruebe cada una de las opciones del programa.

10.2 Hoja de Trabajo Nº 2: Examen

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

Enunciado. Lea detenidamente el siguiente enunciado sobre el cual se desarrollará la presente hoja de trabajo.

Se quiere construir una aplicación que permita simular un examen de geografía, donde se preguntan las capitales de diferentes países. El examen tiene 8 preguntas. De cada pregunta se muestra el número de la pregunta, el enunciado, la bandera del país cuya capital se está preguntando y las 4 posibles respuestas. Cuando se elige una respuesta, se muestra la respuesta seleccionada por el usuario, la respuesta correcta y los puntos obtenidos.

La aplicación carga la información de los países desde un [archivo](#), y selecciona aleatoriamente las preguntas del examen y sus posibles respuestas (el programa no implementa la forma de modificarlas). Esto quiere decir que cada vez que se inicia un nuevo examen, las preguntas serán diferentes.

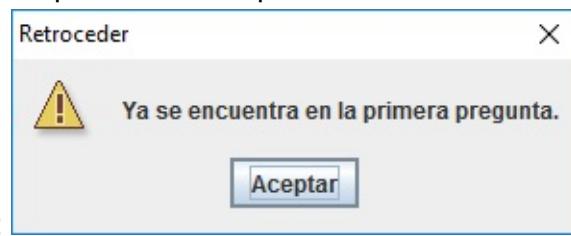
La aplicación debe permitir: navegar entre las preguntas del examen, visualizar la información de una pregunta, responder una pregunta, terminar un examen, iniciar un nuevo examen y visualizar el progreso de las preguntas.

La siguiente es la [interfaz de usuario](#) que se quiere construir, en la cual se identifican cuatro zonas:



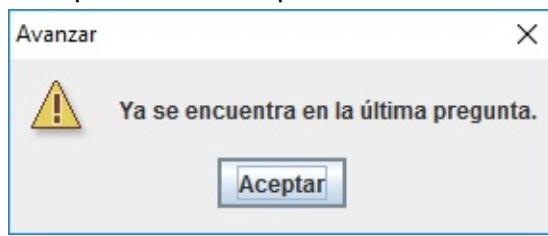
Los siguientes son los mensajes que hay que presentar al usuario, como resultado de su interacción con el programa:

Este mensaje aparece cuando el usuario oprime el botón para retroceder en la lista de



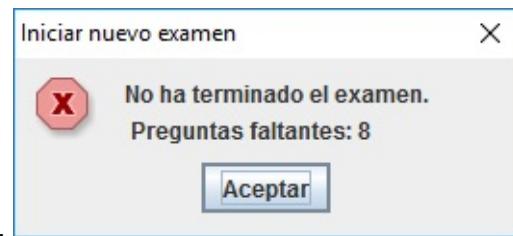
preguntas y está situado en el primero:

Este mensaje aparece cuando el usuario oprime el botón para avanzar en la lista de



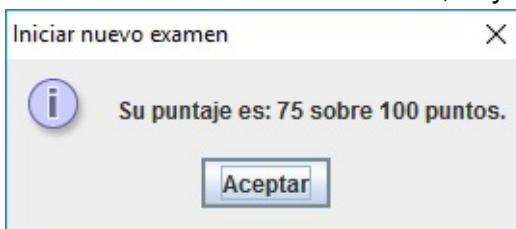
preguntas y está situado en el último:

Este mensaje aparece cuando el usuario intenta terminar el examen, pero no ha respondido



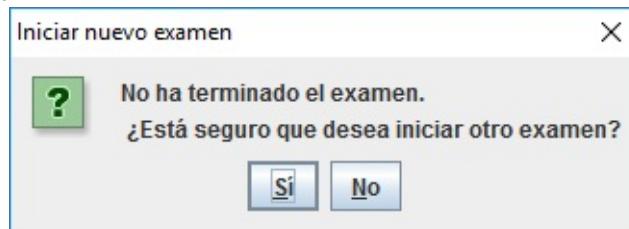
todas las preguntas.

Este mensaje aparece cuando el usuario termina un examen, cuyas respuestas fueron



respondidas en su totalidad.

Este mensaje de confirmación aparece cuando intenta iniciar un nuevo examen, pero no ha



respondido todas las preguntas.

Requerimientos funcionales. Identifique y especifique los requerimientos funcionales de la aplicación.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

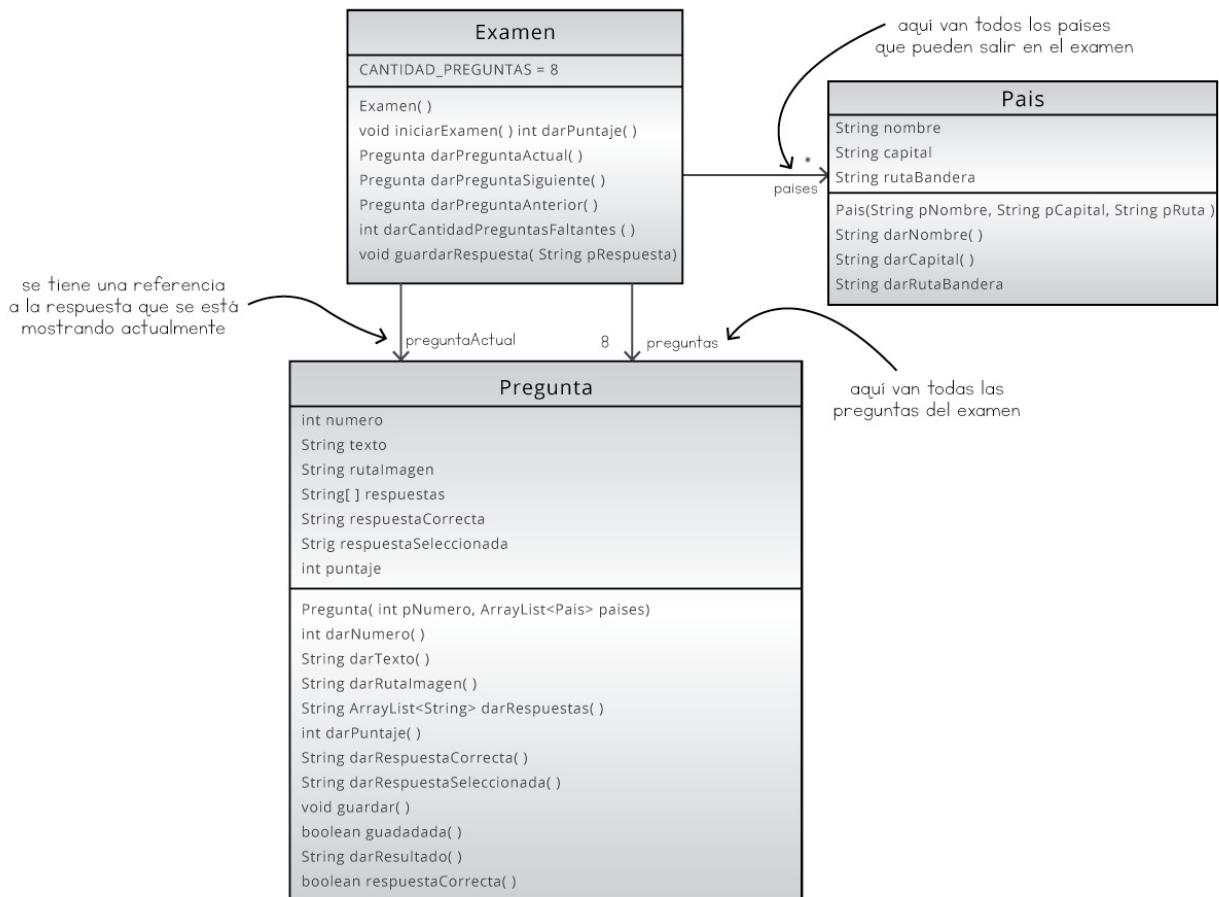
Requerimiento Funcional 5

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 6

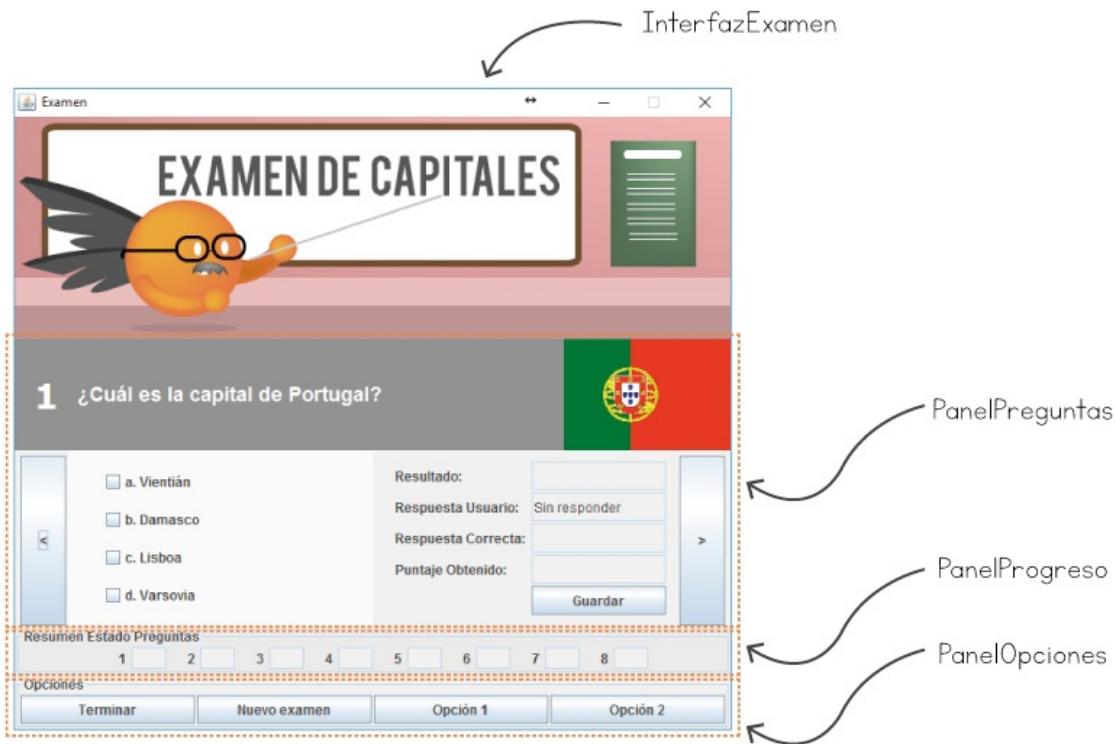
Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Estudie el diagrama de clases que implementa el modelo del mundo y los métodos de cada una de las clases.



Nombre método	Descripción
<code>Examen()</code>	Crea el examen, cargando la información de los países de un archivo . Si hay algún problema en el momento de leer el archivo , lanza una excepción .
<code>void iniciarExamen()</code>	Genera preguntas con sus respuestas seleccionando países de la lista aleatoriamente.
<code>int darPuntaje()</code>	Retorna el puntaje de la pregunta actual.
<code>Pregunta darPreguntaActual()</code>	Retorna la pregunta actual.
<code>Pregunta darPreguntaAnterior()</code>	Retorna la pregunta anterior y actualiza la nueva pregunta actual. Si ya se encuentra en la primera pregunta, lanza una excepción .
<code>Pregunta darPreguntaSigiente()</code>	Retorna la pregunta siguiente y actualiza la nueva pregunta actual. Si ya se encuentra en la última pregunta, lanza una excepción .
<code>int darCantidadPreguntasFaltantes()</code>	Retorna la cantidad de preguntas sin responder.
<code>void guardarRespuesta(String pRespuesta)</code>	Guarda la respuesta seleccionada por el usuario en la pregunta actual.

Interfaz por construir. Observe la estructura de la interfaz que se desea construir y los nombres de las clases que se deben asociar con sus partes.



Arquitectura de la interfaz. Dibuje en UML el diagrama de clases (sin atributos ni métodos) que conformarán la interfaz. Utilice los estereotipos para indicar si es un JFrame o JPanel. Dibuje también las clases del mundo con las que se relacionan.

Diagrama UML



Construcción de la interfaz. Siga los siguientes pasos para construir la interfaz dada.

1	Cree el paquete para las clases de la interfaz (uniandes.cupi2.sinonimos.interfaz).
2	Cree la clase InterfazExamen como extensión de JFrame . Escriba el método main() , encargado de iniciar la ejecución del programa. Incluya los atributos para representar el modelo del mundo, así como los elementos y los paneles que lo conforman. Defina el tamaño de la ventana como 400 x 180. Asocie con la ventana un distribuidor en los bordes . Cree cada uno de los paneles y añádalos adecuadamente a la ventana .
3	Cree la clase PanelPregunta como una extensión de la clase JPanel que implementa ActionListener . Declare las constantes para identificar los eventos que van a generar los botones del panel . Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa. Asocie con el panel un distribuidor en los bordes . Cree los paneles auxiliares necesarios para poder distribuir los elementos de la forma esperada. Tenga en cuenta que los elementos de tipo JCheckBox también deben llamar al método actionPerformed() , y por ende también deben tener un comando asociado. Escriba el esqueleto del método actionPerformed() .
4	Cree la clase PanelProgreso como una extensión de JPanel que implementa ActionListener . Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Deshabilite la posibilidad de escribir en las zonas de texto. Asocie con el panel un distribuidor en grilla. Escriba el esqueleto del método actionPerformed() . En dicho método no vamos a llamar ningún método de la ventana .
5	Cree la clase PanelOpciones como una extensión de la clase JPanel que implementa ActionListener . Declare las constantes para identificar los eventos de los botones. Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa. Asocie con el panel un distribuidor en malla . Escriba el esqueleto del método actionPerformed() .
6	En las clases de los tres paneles, escriba los métodos de refresco de la información y los métodos de acceso a la información.
7	En la clase InterfazExamen , escriba un método para implementar cada requerimiento funcional . Asegúrese de validar los datos y manejar las excepciones de manera que presente los mensajes descritos en el enunciado.
8	Complete el método actionPerformed() en las clases PanelPregunta y PanelOpciones , de modo que haga la llamada a los método de la ventana principal correspondientes. Recuerde que en este método también se deben ejecutar las acciones necesarias para que cuando un usuario seleccione un CheckBox , no haya ninguna otra casilla seleccionada.
9	Complete todos los detalles que faltan en la interfaz, para obtener la visualización y el funcionamiento descritos en el enunciado. Pruebe cada una de las opciones del programa.

