Curso 2019 / 2020

Tutorial:

Añadido de una Interfaz Gráfica de Usuario (GUI) a una aplicación

Información previa

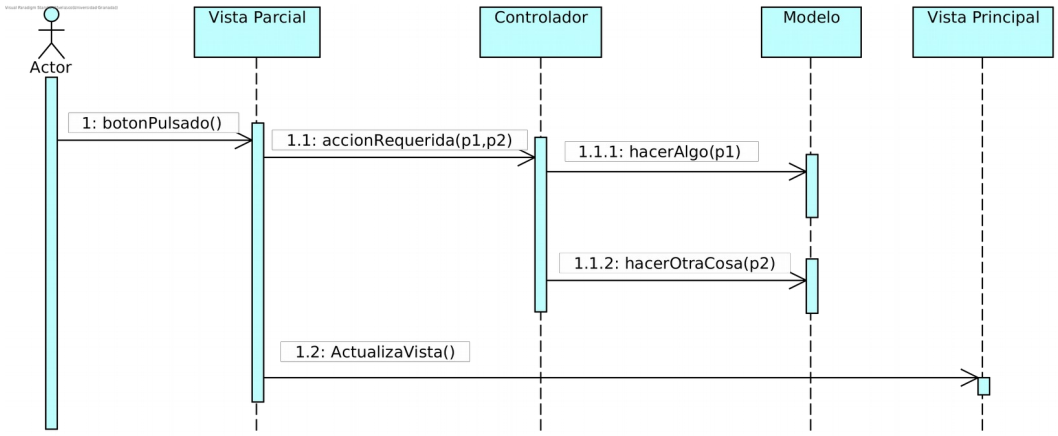
Una vez implementadas todas las clases que modelan, representan y gestionan los datos implicados en una aplicación, es necesario establecer un mecanismo de comunicación con el usuario para que haga uso de las funcionalidades proporcionadas por la aplicación. Mediante este tutorial vamos a abordar la incorporación de una interfaz gráfica como mecanismo de comunicación con el usuario.

Patrón Modelo-Vista-Controlador

En teoría ya has aprendido sobre este patrón, no obstante, aquí tienes un pequeño resumen adaptado a cómo se han hecho las cosas en Deep-Space.

- Modelo: Clases que modelan la aplicación, se accede al modelo a través de su fachada
  - Capa *ToUI: Clases que permiten acceder (solo lectura) a los objetos de la aplicación.
- Vista: Clases encargadas de interactuar con el usuario.
  - Le muestran información al usuario, presentada de una forma amigable.
  - Contiene una vista principal, la ventana principal de la aplicación, y otras vistas parciales para mostrar información de diversos objetos.
  - Escucha y atiende los eventos que realiza el usuario, escribir un dato en un campo, pulsar un botón, seleccionar un elemento, etc. dando respuesta a dichas acciones del usuario. En muchas ocasiones, el procesamiento realizado como respuesta a un requerimiento del usuario supone **pedirle al modelo, a través del controlador,** que realice todas las acciones que sean necesarias para dar respuesta a dicho requerimiento del usuario. Dicho procesamiento suele finalizar con una actualización de la vista, para que refleje el nuevo estado del modelo.
- Controlador: El encargado de la comunicación entre Modelo y Vista.
  - Ante un requerimiento del usuario, recibe un mensaje de la vista y envía los mensajes que correspondan al modelo para que se realice la acción requerida. En muchas ocasiones, una acción del usuario implica realizar varias acciones sobre el modelo.

La relación entre usuario, modelo, vista y controlador se ve gráficamente en el siguiente diagrama.



Elementos para crear la GUI, ventanas, contenedores y componentes

Para aprender en profundidad sobre cómo crear la parte visual de la GUI con Netbeans y Swing, nada mejor que dirigirse a la fuente original:
https://docs.oracle.com/javase/tutorial/uiswing/index.html

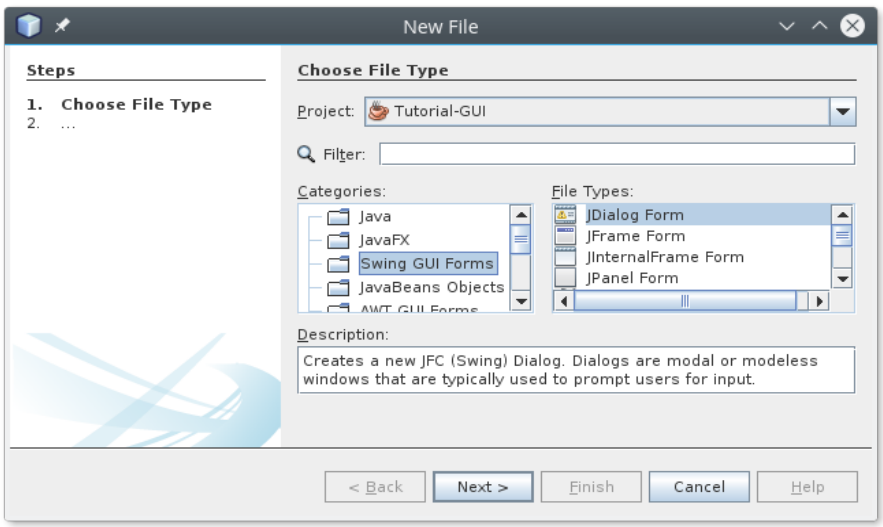
y a la documentación sobre las diversas clases que conforman Swing:
https://docs.oracle.com/javase/8/docs/api/javax/swing/package-frame.html

No obstante, se va a comentar algunas cosas básicas para empezar.

Crear una clase de vista consiste en añadir al proyecto, dentro de un paquete específico para ello normalmente denominado GUI, una clase que herede de `JFrame`, `JPanel` o `JDialog`.

- `JFrame`: Se usará para la ventana principal de la aplicación. Solo se tendrá una clase que herede de `JFrame`.
- `JPanel`: Se usará para vistas parciales, muestran información de determinados objetos.
- `JDialog`: Se usará para crear cuadros de diálogo personalizados.

En la ventana de Netbeans que se abre cuando elegimos “File > New File …” se encuentran los tres tipos mencionados.



Una vez añadida la clase para una determinada vista, Netbeans ofrece un aspecto similar al de la imagen que se muestra a continuación.

- Pestañas “Source” y “Design”: Permiten acceder al código fuente de la clase o a su diseño gráfico. Cuando se está en la pestaña de diseño gráfico se muestra Netbeans como en imagen siguiente.
- Zona de diseño gráfico: Es donde se va componiendo el aspecto gráfico de la vista añadiendo elementos, mediante arrastrar y soltar, desde la paleta de contenedores y componentes.
- Paleta de contenedores y componentes: Los contenedores pueden incluir componentes y otros contenedores; los contenedores también son, a su vez, componentes. Los más usados suelen ser:
  - Componentes:
    - Label: De la clase `JLabel`, son etiquetas para mostrar información en una sola línea.
    - Text Field: De la clase `JTextField`, son campos para que el usuario escriba datos en una sola línea.
    - Text Area: De la clase `JTextArea`, son rectángulos que permiten mostrar (si se configura como no editable) información en varias líneas. Si se configura como editable, permite que el usuario escriba en ellos textos de varias líneas.
    - Button: De la clase `JButton`, son botones para que el usuario pueda dar órdenes al pulsar sobre ellos.
  - Contenedores (y al mismo tiempo componentes)
    - Panel: De la clase `JPanel`, son rectángulos que pueden mostrar una línea en su borde, e incluso un título, y sirven para agrupar otros componentes. Como se ha comentado, también es la clase de la que heredan las clases que van a representar vistas parciales.
    - Scroll Pane: De la clase `JScrollPane`, se usan para mostrar elementos de tamaño grande. Ya que cuando lo que muestra no cabe en las dimensiones que tiene, aparecen barras de scroll para que el usuario pueda visualizar todo su contenido.
      Es habitual usarlos conjuntamente con un Panel que va a mostrar un número variable de elementos. En ese caso, un Scroll Pane del tamaño deseado incluye un JPanel, teniendo este `JPanel` un `Flow Layout` (en breve se explicarán los layouts) que a su vez contendrá ese número variable de elementos. Cuando el número de elementos sea tal que no quepan todos en las dimensiones del Scroll Pane aparecerán las barras de scroll para poder visualizarlos todos.
- Estructura jerárquica (Navigator): Como se ha comentado, la vista puede incluir componentes y contenedores, éstos a su vez pueden incluir otros componentes y contenedores, estableciéndose entre los elementos una estructura jerárquica bajo el criterio “incluido en”. En esta zona de Netbeans se observa esta jerarquía y se pueden mover elementos para incluirlos en contenedores.
- Propiedades del elemento seleccionado: Muestra los atributos de dicho elemento permitiendo configurarlo al gusto: El color, el texto que muestra una etiqueta, etc.
- Proyectos, paquetes y clases: Esta zona es común tanto si se está en la pestaña de código (Source) como si se está en la pestaña de diseño gráfico (Design) y permite seleccionar y abrir los archivos del proyecto, además de añadir nuevas clases y paquetes.

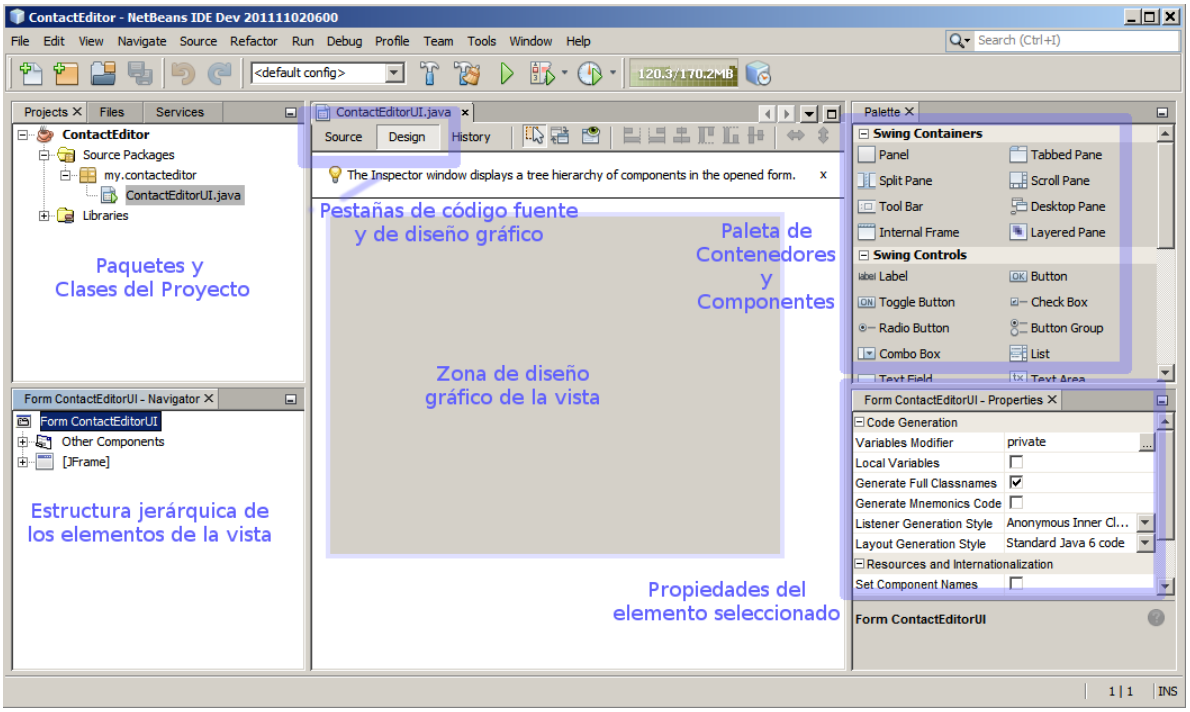


Imagen original sin rótulos: https://netbeans.org/images_www/articles/80/java/quickstart-gui/01_gb_ui.png

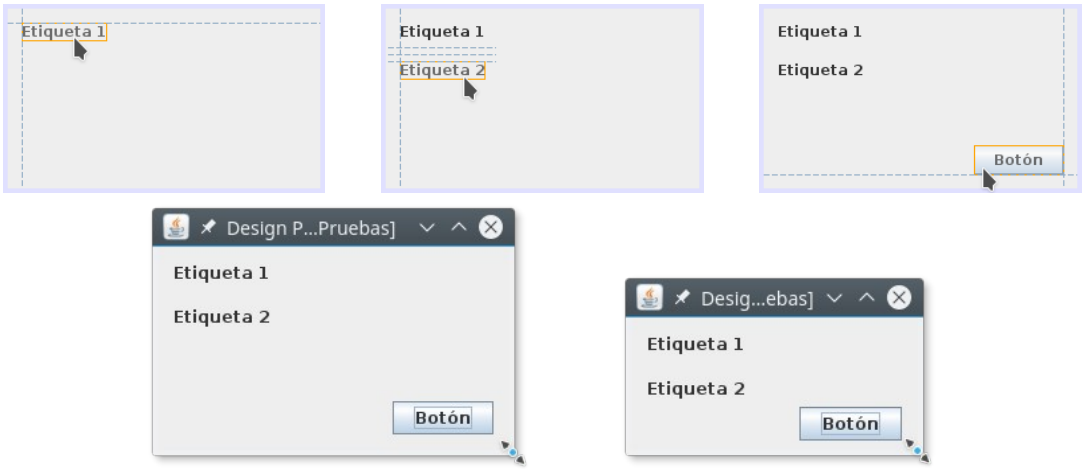
Cada vez que se añada un componente a la vista, **se le debe dar un nombre significativo** para que, en el código, cada elemento sea fácilmente identificable. Ello puede hacerse haciendo clic con el botón derecho sobre el elemento (en el Navigator o en la zona de diseño gráfico) y eligiendo "Change Variable Name ..." en el correspondiente menú contextual.

Para manejar los diferentes componentes de la GUI en el código fuente es imprescindible conocer la clase que están instanciando, especialmente los métodos de dicha clase, consultores y modificadores. Recuerda, todo ello lo tienes en:
https://docs.oracle.com/javase/8/docs/api/javax/swing/package-frame.html

Layouts

Un layout son las reglas que definen cómo se distribuyen los componentes en un contenedor. Por ejemplo, el Null Layout permite indicar en que posición exacta, medida en píxeles con respecto a la esquina superior izquierda del contenedor, se posiciona cada elemento.

Un layout muy usado, por su versatilidad, es el denominado Free Design. Cada elemento se coloca con relación a una o varias referencias. Esas referencias pueden ser otros elementos o los bordes de un contenedor. En las imágenes siguientes se ve cómo la etiqueta "Etiqueta 1" se está situando con respecto al borde superior e izquierdo de la ventana (observar los gizmos -ayudantes visuales- de líneas de guiones que ayudan a la colocación), la etiqueta "Etiqueta 2" se sitúa a tres *espacios* en vertical de "Etiqueta 1" y alineada con su extremo izquierdo. El cambio el botón se está situando tomando como referencias los bordes derecho e inferior de la ventana. Si en tiempo de ejecución el usuario modifica el tamaño de la ventana, las etiquetas se van a mantener en la esquina superior izquierda (su referencia) mientras que el botón se va a mantener en la esquina inferior derecha (su referencia).



Otro layout muy usado, especialmente cuando se desea añadir, en tiempo de ejecución, un número variable de elementos a un panel, es el Flow Layout, también denominado layout de flujo en este documento. Es muy cómodo porque, en código, solo hay que usar el método `contenedor.add(componente)` sin necesidad de dar coordenadas ni referencias de ningún tipo. Los elementos se van situando uno al lado del otro en el panel. Por defecto los elementos quedan centrados en el panel aunque esto puede modificarse en las propiedades del layout.

Puede verse un ejemplo en la imagen siguiente. El panel “Mis cheques” tiene un Flow Layout, y se le han añadido en tiempo de ejecución 3 vistas de cheques.

Cuadros de diálogo

Se pueden diseñar y usar cuadros de diálogo ya sea para solicitarle nuevos datos al usuario, hacerle una pregunta de respuesta SI/NO, o simplemente para mostrar información.

Si solo se desea mostrar información, esperando un OK cuando el usuario la ha leído o se desea hacer una pregunta de respuesta simple SI/NO, se puede usar la clase de Java `JOptionPane`, la cual tiene diversos métodos de clase con los que se puede diseñar un cuadro de diálogo para este tipo de usos. Por ejemplo:

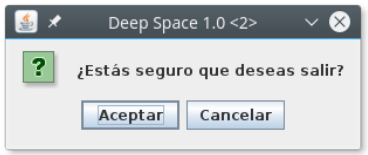
- Solo deseamos mostrar un mensaje esperando un ‘OK’

```
JOptionPane.showMessageDialog (
   vistaPrincipal,
   "Has PERDIDO el combate.\nCumple tu castigo.",
   "Deep Space 1.0",
   JOptionPane.INFORMATION_MESSAGE);
```

Lo cual mostraría

- Se desea preguntar algo esperando un ‘Aceptar’ o ‘Cancelar’

```
JOptionPane.showConfirmDialog(
   vistaPrincipal,
   "¿Estás seguro que deseas salir?",
   "Deep Space 1.0",
   JOptionPane.OK_CANCEL_OPTION);
```

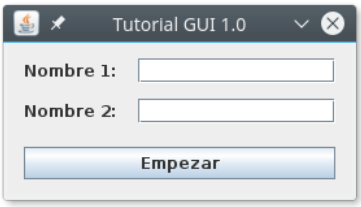


Lo cual mostraría

Y cuyo valor devuelto se puede comparar con `JOptionPane.OK_OPTION` o `JOptionPane.CANCEL_OPTION`

Consultar el resto de posibilidades que ofrece esta clase en la documentación del lenguaje.

Si se desea recopilar más información que una respuesta sencilla, se debe diseñar un cuadro de diálogo personalizado. Por ejemplo, suponer que se desean leer un par de nombres con un cuadro de diálogo. El diseño gráfico de esa ventana podría ser como el que se muestra a continuación.



Para ello se crea una clase que derive de la clase JDialog. El constructor recibe la ventana de la que deriva, que suele ser la ventana principal. Se implementa llamando a super, inicializando los atributos del objeto y ejecutando initComponents() para que se cree el diseño gráfico que se haya hecho. En este caso el diseño gráfico es un par de etiquetas (“Nombre 1” y “Nombre 2”), un par de Text Field para escribir los nombres y un botón ‘Empezar’. También se incluye código para indicar qué quiere que se haga si el usuario cierra la ventana del cuadro de diálogo. Por ejemplo:

```
public NamesCapture (JFrame parent) {
      super(parent, true);
// El parámetro  true  indica que la aplicación se quedará esperando
//    a que se cierre el cuadro de diálogo
// Se inicializa el atributo que alojará los nombres escritos por el usuario
      names = new ArrayList<>();
      initComponents();
      this.addWindowListener (new WindowAdapter() {
            @Override
            public void windowClosing (WindowEvent e) {
// Aquí se programa lo que se desea hacer si el usuario cierra la ventana.
            }
      });
}
```

El método asociado al botón ‘Empezar’, que el usuario pulsará cuando haya finalizado de escribir los nombres, actualizará el atributo names y cerrará el cuadro de diálogo.

```
      names.add (jtName1.getText());
```

```
        names.add (jtName2.getText());
        dispose();
```

Por último, el método `ArrayList<String> getNames()`, el método que es llamado para obtener los nombres leídos, tendría solo 2 instrucciones: mostrar el cuadro de diálogo y devolver los nombres.

```
        this.setVisible (true);
// En este punto la ejecución de este método está detenida hasta
//   que se ejecute el método dispose() en el código asociado al botón "Empezar"
        return names;
```

¿Cómo se usaría este cuadro de diálogo para leer un nombre? Fácil. Se crea una instancia del cuadro de diálogo y se usa el consultor para obtener la lectura que se desea.

```
        NamesCapture namesCaptureDialog = new NamesCapture (this);
        ArrayList<String> names = namesCaptureDialog.getNames();
```

Procedimiento para el añadido de una GUI a una aplicación

Con toda esta información previa que se ha visto ya se puede incorporar una GUI a una aplicación. No obstante, se ha entregado un ejemplo de una aplicación muy básica para tenerla como referencia. La aplicación consiste en un par de personas que van obteniendo cheques bancarios y se los van gastando. Muy naïve, pero suficiente para entender los conceptos fundamentales y practicar el procedimiento.

Puedes practicar borrando (o moviendo a otro sitio) el contenido de la carpeta GUI para hacerlo tú mismo según se indica a continuación.

Como siempre, es importante ir paso a paso y probando y depurando cada pequeño avance que se vaya haciendo.

Estructura de la aplicación

En el diagrama de clases adjunto, tutorialGUI-DC.pdf, se muestra el diseño estructural de una aplicación de ejemplo que contempla el modelo con su capa *ToUI, el controlador, y la vista junto con las clases del paquete Swing de las cuales derivan las diferentes clases de la vista.

Se observa que se ha introducido una interfaz Java, esta interfaz declara el comportamiento que debe implementar la vista principal (ventana principal) de la aplicación. De esta forma, se puede cambiar una vista por otra fácilmente.

Proceso a seguir

Añadir un nuevo paquete al proyecto, denominado View, dentro del cual se creará la interfaz java, denominada View, en principio solo con estos métodos:

```
        public void updateView ();
        public void showView ();
```

Añadir el paquete View.GUI al proyecto. En dicho paquete, crear una clase, MainWindow, de tipo JFrame. Modificar el archivo fuente para que la clase, además, implemente la interfaz View.

Esta clase será singleton, ya que para una determinada vista de la aplicación, solo va a existir una ventana principal. Ejemplo:

```
public class MainWindow extends JFrame implements View {
      private static MainWindow instance = null;
      public static MainWindow getInstance () {
            if (instance == null) {
                  instance = new MainWindow();
            }
            return instance;
      }
      private MainWindow () { . . . }
      . . .
}
```

Es importante instanciar el objeto único de la clase en el momento de la primera consulta y no en el momento de la declaración (como se muestra en el ejemplo). Con la estructura que se está presentando, se pueden tener diferentes vistas para una aplicación; el hacerlo de esta manera hace que solamente la vista que se vaya a usar sea la que se construya. El resto de vistas posibles no se construyen y no consumen recursos.

Netbeans le añade un método main a las clases de tipo JFrame y JDialog que se crean, estos métodos main se pueden eliminar, se usará el propio main que se implemente, más adelante.

En el constructor por defecto que ha creado Netbeans, se le añade unas instrucciones para que la aplicación finalice cuando se cierra la ventana principal de la aplicación.

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
addWindowListener(new java.awt.event.WindowAdapter() {
    @Override
    public void windowClosing(java.awt.event.WindowEvent e) {
        Controller.getInstance().finish(0);
    }
});
```

Lo siguiente será implementar los métodos que se heredan de la interfaz View.

```
public void updateView();
```

Por ahora se deja en blanco.

```
public void showView();
```

Este método sólo hace visible la vista, `this.setVisible(true);`

Añadir el paquete Controller al proyecto y en él, crear la clase Controller, con atributos privados para referenciar a la clase fachada del modelo y a la ventana principal de la vista (nótese que el tipo del atributo view es la interfaz que se ha creado previamente).

La clase Controller también será singleton. Una aplicación solo va a tener un controlador.

Como métodos de la clase Controller, en principio se hará solamente:

- El constructor, privado, no realiza nada.
- Un método `public void setModelView` que recibirá 2 parámetros: uno del tipo Model (la clase fachada del modelo) y otro de tipo View (la interfaz java definida anteriormente). Dicho método inicializará los atributos `model` y `view` respectivamente.
- El método `public void start ();` que realizará lo necesario para poner en marcha la aplicación. En este ejemplo, se necesita un `ArrayList<String>` con los nombres de las personas que intervienen en la aplicación para inicializarla.

Para la lectura de los nombres, como es una interacción con el usuario, se hará desde las clases de la vista. Para ello se necesita:

- Añadir al paquete `View.GUI` una clase que herede de `JDialog` para leer dichos nombres, tal como se ha visto previamente en este documento. Se llama `NamesCapture` en el ejemplo.
- Añadir a la interfaz `View` una declaración de método para realizar dicha lectura, por ejemplo, `public ArrayList<String> getNames();`
- Implementar dicho método en la clase `MainWindow` (que implementa la interfaz `View`), por ejemplo,

```
@Override
public ArrayList<String> getNames() {
    NamesCapture namesCapt = new NamesCapture(this);
    return namesCapt.getNames();
}
```

Así, el método start() de la clase Controller queda así

```
public void start() {
  model.init(view.getNames());
  view.updateView();
  view.showView();
}
```

Se completa la clase Controller, por ahora, con el método de paquete `void finish (int i);` que en principio cerrará la aplicación. `System.exit(i);`

Programa principal

Crear un paquete nuevo para alojar a la clase del programa principal. Y crear en esta clase el correspondiente main. Recordar que el método main debe ser de clase y con visibilidad pública.

Con todo lo hecho hasta ahora, el programa principal queda reducido a pocas líneas, se crea un modelo, se obtiene la instancia única de la vista, se obtiene la instancia única del controlador, se *conectan* estos elementos y se le pide al controlador que arranque la aplicación.

```
Model model = new Model();
View view = MainWindow.getInstance();
Controller controller = Controller.getInstance();
controller.setModelView (model,view);
controller.start();
```

En este punto, ya se puede ejecutar la aplicación. Se abrirá el cuadro de diálogo para la lectura de

nombres y posteriormente la ventana principal de la aplicación. Esta ventana no mostrará nada aún. Se puede finalizar la aplicación cerrando la ventana.

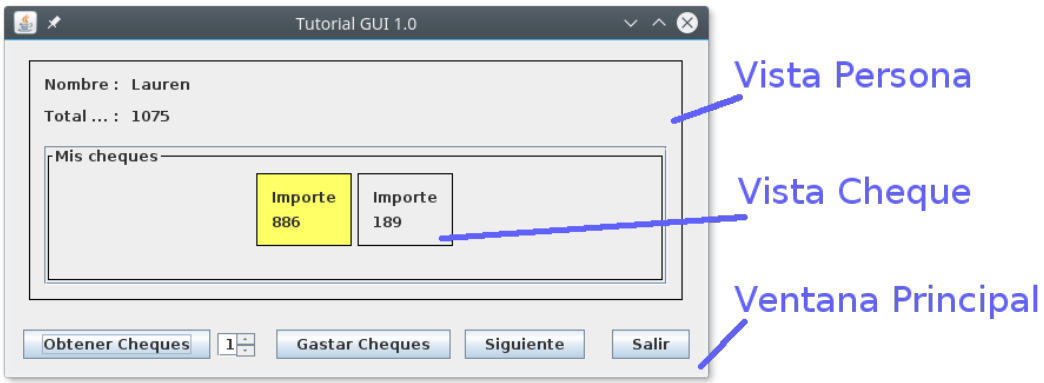
Diseño e implementación de vistas

Se pasa a realizar el diseño visual de la aplicación. Para ello, se deja temporalmente el ordenador y en una hoja de papel se realiza el diseño gráfico de todo, qué vistas se van a hacer para cada cosa, con qué contenido, con qué distribución, etc. Creednos, es mucho más rápido hacer esta fase en papel.

A pesar de que el espacio de un monitor da mucho de sí, en tus primeros diseños de interfaces gráficas de usuario es normal que se te quede pequeño. Intenta distribuir bien los distintos elementos y no dejar demasiado espacio en blanco entre ellos.

Una vez se tiene claro el diseño gráfico de todo, se pasa a implementarlo en el ordenador.

En el diseño realizado habrá vistas que contienen a su vez a otras vistas. En el ejemplo mostrado, la vista de la aplicación (ventana principal) contiene la vista de una persona, que a su vez contiene vistas de cheques.



Se comienza implementando la vista más interior de todas, luego se irán implementando las vistas intermedias hasta que se termine con la vista más exterior de todas, la ventana principal.

Cada vista (salvo la ventana principal) se crea añadiendo una nueva clase al proyecto, en el paquete View.GUI, de tipo JPanel. Se diseña de la manera habitual, arrastrando y soltando los diferentes componentes desde la paleta de componentes al JPanel que se está diseñando. Recuerda ponerle nombres significativos a los elementos que se añaden.

Se le añade un método set* que se encargará de actualizar la información que muestra la vista a partir del objeto *ToUI que recibe como parámetro. Por ejemplo, en la clase de la vista de un cheque existe un método `void setBankCheck (BankCheckToUI bc)` cuya implementación actualiza la etiqueta JLabel que muestra el importe de dicho cheque concreto.

```
amount.setText (Integer.toString (bc.getAmount());
```

Se finaliza siempre con la orden `repaint();` para que la actualización se haga efectiva.

Si la actualización de la vista no solo ha sido cambiar unas etiquetas, sino que ha supuesto añadir o

eliminar componentes, además de `repaint();` hay que ejecutar el método `revalidate();`

Por ejemplo, la vista de la persona requiere hacer `revalidate()` además de `repaint()` ya que cuando se actualiza la información de una persona, puede tener más o menos elementos que antes (ha podido adquirir o gastar cheques).

Preparar la ventana principal para poder probar las vistas que se van haciendo

Como hemos recomendado, conviene ir probando y depurando cada nueva implementación que se va haciendo. Veamos cómo preparar la ventana principal para que al ejecutar la aplicación se puedan probar las vistas que se van haciendo.

Se añade a la vista principal un `JPanel` desde la paleta de componentes, y se le llama `panelPruebas`. Se le pone el layout de flujo.

En la clase fachada del modelo se le añade un método público que cree un objeto de la clase que se quiere probar.

En ejemplo que se está siguiendo, se añadiría en la clase Model un método `public BankCheck dameUnChequePrueba()` que haga precisamente eso.
Estos métodos para pruebas luego se pueden borrar.

Tambien se añadiría un método similar en el controlador, `public BankCheckToUI dameUnChequePrueba()`, solo que éste devuelve una versión *ToUI del objeto.

Finalmente, el método updateView de la clase de la ventana principal se programa para crear un objeto de la vista que queremos probar y mostrarlo. En el ejemplo que se está siguiendo sería algo como esto:

```
BankCheckToUI chequePrueba = Controller.getInstance().dameUnChequePrueba();
BankCheckView vistaChequePrueba = new BankCheckView();
vistaChequePrueba.setBankCheck (chequePrueba);
panelPruebas.add(vistaChequePrueba);
repaint();
revalidate();
```

Ahora, si ejecutas la aplicación, se debe ver la vista del cheque que se está probando.

Inclusión de vistas interiores en vistas exteriores

En muchas ocasiones una vista (contenedora) incluye otra (contenida). En el ejemplo de los cheques la vista principal incluye a la vista de la persona. También, la vista persona incluye vistas de sus cheques.

En el caso de que la vistas contenidas sean fijas en cuanto a cantidad, por ejemplo la vista principal de la aplicación de ejemplo incluye una vista de persona y siempre una, se puede dejar el trabajo hecho en tiempo de diseño. A la vista contenedora se le añade un `JPanel` con las dimensiones adecuadas y en el lugar que se desee que será el panel contenedor de la vista contenida. Se le debe poner el **layout de flujo** (Flow Layout). Posteriormente, en el código fuente, en el constructor y tras la llamada al método `initComponents()` se crea una instancia de la vista contenida y se añade al panel contenedor

```
personView = new PersonView();
panelContenedorPersonView.add (personView);
```

La referencia a la vista contenida es un atributo privado de la clase de la vista contenedora.

En cuanto a la actualización, debe transmitirse jerárquicamente *hacia abajo*. Es decir, el método set* encargado de actualizar la información de la vista contenedora deberá llamar al método set* de las vistas contenidas para que éstas también se actualicen.

En otros casos la vista contenedora puede tener un número variable, incluso 0, de vistas contenidas. En el ejemplo de los cheques la vista de la persona incluye un número variable de vistas de cheques. Además, según va creciendo el número vistas contenidas puede que no quepan en el panel previsto para contenerlas y se desee disponer de barras de scroll para poder visualizarlas todas.

En esta ocasión hay que añadir un `JScrollPane` que proporcionará la funcionalidad de scroll. Dentro se añadirá un `JPanel` que alojará a las vistas contenidas que se vayan añadiendo. Este `JPanel` debe tener **layout de flujo**. Debe quedar una estructura como la que se muestra en la siguiente imagen extraída del Navigator de Netbeans.

El método encargado de actualizar la vista contenedora será el encargado de crear y añadir al `JPanel` las vistas contenidas a partir de la colección de objetos referenciados que nos devuelva el consultor correspondiente. Por ejemplo,

```
panelChecks.removeAll(); // se borra la visualización anterior
ArrayList<BankCheckToUI> checks = aPerson.getBankChecks();
BankCheckView checkView;
for (BankCheckToUI c : checks) {
      checkView = new BankCheckView();
      checkView.setBankCheck(c);
      panelChecks.add(checkView);
}
repaint();
revalidate(); // recordar, necesario ya que se añaden o eliminan elementos
```

Recordar, después de cada nueva vista que se implemente hacer lo necesario para poder probarla, añadiendo métodos temporalmente al modelo, al controlador y modificado `updateView()` de la ventana principal.

La última vista que se implementa es la vista principal, la más exterior de todas. En este caso, la implementación que se haga de `updateView()` será ya la definitiva.

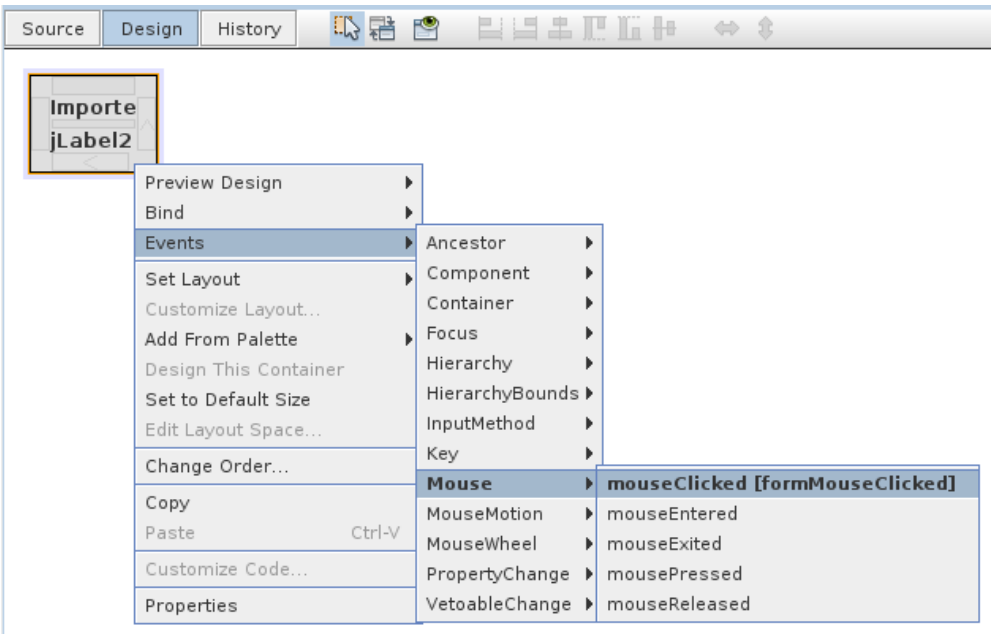
Selección de vistas contenidas en un JPanel para realizar algo con ellas

A veces, en un panel que contiene un número variable de vistas, el usuario puede necesitar seleccionar algunas de ellas para realizar una operación. En el ejemplo, se eligen cheques concretos para poder gastarlos. Veamos cómo implementar esa funcionalidad de selección. Implica realizar varias tareas: se debe anotar que una vista está seleccionada; darle una realimentación al usuario para que sepa qué vistas están seleccionadas y cuáles no; y crear un consultor para que desde el código se pueda saber qué vistas concretas ha seleccionado el usuario.

La anotación de si una vista está seleccionada o no se realiza con un atributo privado de tipo boolean que se añade a la clase de esa vista, inicializado a false ya que por defecto ninguna vista está seleccionada (`private boolean selected = false;`). También es necesario añadir un consultor sobre este atributo (`boolean isSelected()`).

Para darle realimentación al usuario de qué vistas estás seleccionadas se puede cambiar algún atributo visual, por ejemplo el color de fondo de la vista. Una forma sencilla de hacer eso es ponerle, actuando sobre las propiedades del JPanel de la vista, un color de fondo fijo, el que se desee para mostrar que la vista está selecionada. Y después mostrarlo o no, haciendo que el fondo sea opaco (se vería dicho color) o no. Se realiza con el método `setOpaque (boolean onOff)`.

Cuando el usuario desea seleccionar una vista hará clic sobre ella, al igual que cuando desea deseleccionar una vista que ya estaba seleccionada. Se debe programar el método asociado a ese evento. Para ello, mostramos en Netbeans la pestaña de diseño asociada a dicha clase y hacemos clic con el botón derecho del ratón sobre el fondo de la vista. En el menú contextual buscamos la opción `mouseClicked` (ver imagen sigiente).



Ello añade el método correspondiente y permite programarlo. Su código sería cambiar el estado del atributo selected, cambiar el estado de opacidad del fondo de la vista y, por supuesto, llamar a `repaint()`. Es decir,

```
        selected = !selected;
        setOpaque (selected);
        repaint();
```

Es necesario, en la vista contenedora (la vista de la persona), añadir un consultor para saber que vistas contenidas están seleccionadas. Cómo se puede imaginar, no es más que un bucle sobre todas las vistas para consultar, una a una, cuáles están seleccionadas.

```
ArrayList<Integer> getSelectedChecks () {
    ArrayList<Integer> selectedChecks = new ArrayList<>();
    int i = 0;
    for (Component c : panelChecks.getComponents()) {
        if (((BankCheckView) c).isSelected()) {
            selectedChecks.add(i);
        }
        i++;
    }
    return selectedChecks;
}
```

Se devuelve un array de índices en vez de un array de objetos ya que los objetos que se están manejando son instancias de vista. El modelo desconoce la existencia de este tipo de objetos. Dándole al modelo la colección de índices, sí puede saber que objetos (en el modelo) son los que se han seleccionado; los que ocupen las mismas posiciones en su respectivo array en el modelo.

Programación de botones

La mayoría de las veces, las órdenes que da el usuario en una GUI las da pulsando botones.

Se añadirán botones en la GUI y se programará el método asociado al evento de pulsarlo. Para programar dicho método se hace doble clic sobre el botón a programar, en la pestaña de Netbeans de diseño.

La programación de un botón normalmente implica un algoritmo de 3 fases:

1. Recopilar información de la GUI, por ejemplo, qué cheques están seleccionados.
2. Enviarle un mensaje al controlador para que realice la acción requerida, por ejemplo, gastarlos.
3. Enviar un mensaje a la ventana principal para que se actualice la vista completa, con el nuevo estado en el que haya quedado el modelo tras la operación.

```
Controller.getInstance().spendBankChecks (personView.getSelectedChecks());
```
fase 2 *fase 1*
```
MainWindow.getInstance().updateView();
```
fase 3

En el controlador hay que añadir el correspondiente método para procesar dicha acción y es el que se encarga de comunicarse con el modelo para que la acción se lleve a cabo.

```
void spendBankChecks(ArrayList<Integer> selectedChecks) {
  // El modelo gasta los cheques de uno en uno
  // El controlador debe adaptarse a ese modo de proceder del modelo
  //    y enviarle un mensaje para cada cheque
  // El controlador debe conocer cómo actúa el modelo
  // El modelo gasta los cheques borrándolos de un ArrayList
  // Al borrar el elemento  i  en un ArrayList,
  //    los elementos j > i descienden una posición
  //    y dejan de estar referenciados por el mismo índice
  // Por ello el controlador recorre el bucle descendentemente
  //    Al borrar el elemento  i  de un ArrayList,
  //    los elementos j < i son referenciados por el mismo índice

    for (int i = selectedChecks.size() - 1; i >= 0; i--) {
        model.spendBankCheck(selectedChecks.get(i));
    }
}
```

Cuadros de diálogo

A veces, el controlador al procesar una orden necesita solicitar algún dato adicional, pedir confirmación de algo o avisar al usuario de alguna circunstancia. Necesita por tanto abrir un cuadro de diálogo para ello.

Los cuadros de diálogo es responsabilidad de la vista. Por cada cuadro de diálogo que se desee añadir se añadirá un método en la interfaz View y su correspondiente implementación en la vista principal que implemente la interfaz View. Cuando el controlador necesite comunicarse con el usuario lo hará enviándole el correspondiente mensaje a la vista para que abra el cuadro de diálogo que corresponda.

En el ejemplo de los cheques se ha usado esta técnica para salir de la aplicación. Se ha añadido a la interfaz `View` un método, `public boolean confirmExitMessage ()`, que se ha implementado en la vista principal abriendo un cuadro de diálogo con el típico mensaje solicitando confirmación sobre si realmente se desea salir; devolviendo `true` en caso afirmativo.

En el controlador se modifica el método `void finish (int i)` para terminar la aplicación solo si el usuario lo confirma.

En otras aplicaciones, este método `finish` comprobaría si parte del trabajo no se ha guardado para avisar al usuario de que se van a perder los últimos cambios dándole la opción de guardarlos antes de salir.

Habilitación y deshabilitación de elementos de la interfaz

No siempre se puede hacer de todo en la aplicación, hay que ir deshabilitando y habilitando botones u otros elementos dependiendo de las acciones que está permitido realizar en cada momento en la aplicación.

Es aconsejable que el propio modelo tenga implementado un sistema de estados de manera que se pueda consultar siempre en qué estado se encuentra la aplicación, y así saber qué acciones se

pueden realizar y cuales no.

El método `updateView()` de la vista puede consultar al modelo, a través del controlador, en qué estado se encuentra y usar esta información para habilitar/deshabilitar los elementos que correspondan con el método `setEnable(boolean onOff)`.

En la aplicación de ejemplo se deshabilita el botón de gastar cheques siempre que la persona no tenga al menos 1.000 euros en cheques.