

Quinta Práctica

Implementación de una Interfaz Gráfica de Usuario básica siguiendo el patrón Modelo-Vista-Controlador

Competencias específicas de la quinta práctica

- Conseguir una visión inicial de un entorno de desarrollo de interfaces gráficas de usuario en Java con Netbeans.
- Aprender un modo de trabajo en el desarrollo de interfaces gráficas de usuario que mantiene un acoplamiento reducido entre las clases que modelan la aplicación y las que modelan la interfaz.

Programación de la quinta práctica

Tiempo requerido: Tres sesiones (6 horas)

Planificación y objetivos:

Duración:

- Grupos del lunes, jueves y viernes: de la semana del 12 de diciembre a la del 16 de enero
- Grupos del martes y miércoles: de la semana del 19 de diciembre a la del 16 de enero

Objetivos:

- Familiarizarse con el desarrollo de interfaces gráficas de usuario en Java.
- Añadirle una interfaz gráfica de usuario a la aplicación Napakalaki según el patrón de diseño modelo-vista-controlador
- Esta práctica se realizará exclusivamente en Java.

Examen:

- **Lugar:** Las aulas se indicarán en la convocatoria.
- **Día:** Todos los grupos el **27 de enero**, la hora se indicará en la convocatoria.
- **Tipo examen:** Realizar modificaciones y/o añadidos sobre el propio código.

Descripción general de la práctica

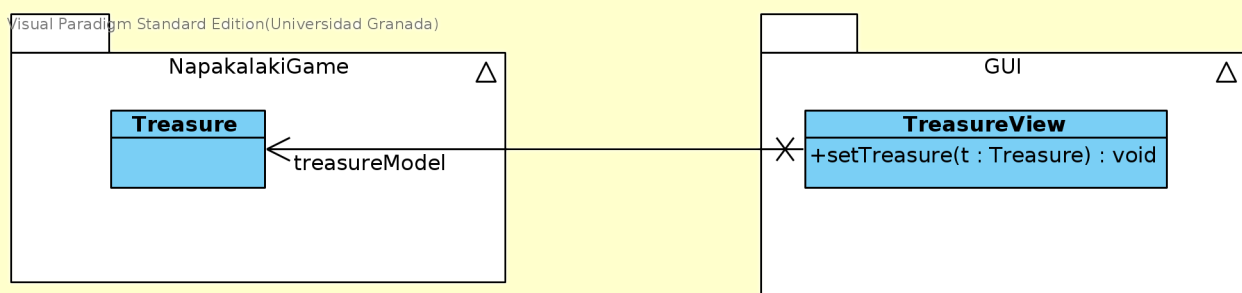
- **Importante:** No limitarse a seguir los pasos. Se debe entender lo que se está haciendo y preguntar todo lo que no se entienda.
- Se creará un conjunto de clases para añadirle una Interfaz Gráfica de Usuario (Graphics User Interface, GUI) al juego Napakalaki siguiendo el patrón modelo-vista-controlador.
- Las clases del modelo ya se realizaron en prácticas anteriores. Son las clases Napakalaki, Player, Monster, etc.
- Las clases de la vista serán una clase denominada NapakalakiView que heredarán de JFrame y las respectivas vistas para algunas de las demás clases: PlayerView, MonsterView, TreasureView, etc. que heredarán de JPanel.
- El código del controlador estará insertado en las respectivas clases de vista.

Resumen de las tareas que se realizarán

- Se creará en primer lugar una versión sin contenido de la ventana principal del juego, la vista asociada a la clase Napakalaki.
- Posteriormente se creará el cuadro de diálogo para introducir los nombres de los jugadores.
- A continuación, se irán creando el resto de vistas desde la más interna a la más externa, de modo que dado que la vista del jugador contendrá las vistas de sus tesoros, se hará la vista asociada a la clase Treasure antes que la vista asociada a la clase Player. Y así sucesivamente hasta la vista más externa que será la asociada a la clase Napakalaki.
- Se finalizará añadiendo el código que se ejecutará ante los diferentes eventos que se produzcan, como pulsar un botón o seleccionar un tesoro.

Esquema general de las clases de la vista

Para cada clase en el modelo que tenga una vista en la interfaz gráfica de usuario se implementará el diagrama que se muestra a continuación. El ejemplo se muestra con la clase Treasure. Se hará de manera equivalente con el resto de clases del modelo que tengan una vista.



El método set no solo se encarga de actualizar el atributo de referencia. También tiene la responsabilidad de actualizar los componentes gráficos de la vista. Véase la siguiente plantilla:

```

public void setTreasure (Treasure aTreasure) {
    // se actualiza el atributo de referencia
    treasureModel = aTreasure;
    // se actualiza la vista del tesoro, por ejemplo
    this.name.setText (treasureModel.getName());
    // la siguiente instrucción hace que los cambios en la vista sean efectivos
    repaint();
}

```

Enlaces interesantes

Si no se está familiarizado con el diseño y creación de interfaces gráficas de usuario de manera interactiva con Netbeans, se recomienda leer el siguiente enlace:

<https://netbeans.org/kb/docs/java/quickstart-gui.html>

En PRADO tenéis diversos ejemplos sobre interfaces gráficas de usuario en Java.

Incorporación de un dado con representación gráfica

A) Incorporar un dado gráfico al juego

1. Crear un nuevo paquete, denominado GUI, que contendrá las clases definidas para la vista.
2. Eliminar el archivo Dice.java que tenías.
3. Añadir en el paquete GUI el fichero Dice.java que se ha proporcionado con esta práctica.
4. Habrá que añadir `import GUI.Dice;` en aquellos archivos donde se usa esta clase. Se pueden seguir las indicaciones de errores que proporciona Netbeans en los márgenes, para localizar dichas referencias y añadir los imports.

Creación de la estructura básica de la aplicación

B) Crear un paquete para las clases de la vista y crear la ventana principal del juego (sin contenido)

1. Crear un nuevo paquete, denominado **GUI**, que contendrá las clases definidas para la vista.
2. Crear la clase `NapakalakiView` y enlazarla con la clase `Napakalaki`, según los siguientes pasos:
 - a) Añadir un nuevo archivo al proyecto de la categoría **Swing GUI Forms** y de tipo **JFrame Form** que se llame **NapakalakiView** y pertenezca al paquete **GUI**.

La ventana de tipo JFrame se usa como ventana principal de una aplicación.

- b) Seleccionar la pestaña *Source* (situada en la parte superior izquierda del editor de texto) y hacer los siguientes cambios:
 1. Añadir un atributo de la clase **Napakalaki**, denominado **napakalakiModel**. Añadir también un método **setNapakalaki** para darle valor a dicho atributo.
 2. Ir al final del fichero y sustituir la función **main** que Netbeans ha incluido por este método:

```
public void showView() {  
    this.setVisible(true);  
}
```

- c) Añadir una clase al paquete `NapakalakiGame` que contenga un método **main**, es decir, `public static void main (String[] args)` y que sea el principal de la aplicación. En dicho **main**, declarar y definir las variables que sean necesarias de modo que:

1. Se tenga una variable de la clase **Napakalaki**, denominada **game** que referencie a su única instancia.
2. Se tenga una variable de tipo **NapakalakiView**, denominada **napakalakiView** que referencie al objeto de la clase **NapakalakiView** que se construya. Puede construirse en esta misma línea.
3. Se cree la única instancia del dado gráfico, asociado a la ventana principal del juego.

```
Dice.createInstance (napakalakiView);
```

4. Se enlace el modelo a la vista:

```
napakalakiView.setNapakalaki (game).
```

5. Se muestre la ventana principal de la aplicación:

```
napakalakiView.setVisible(true);
```

- d) Ejecutar la aplicación, debería abrirse la aplicación en una ventana, aunque obviamente, la ventana no tendrá contenido ya que aún no se le ha dado ninguno.

Creación del cuadro de diálogo para leer los nombres de los jugadores

C) Crear un cuadro de diálogo para leer los nombres de los jugadores

1. Añadir un nuevo archivo de la categoría **Swing GUI Forms** y de tipo **JDialog Form** que se llame **PlayerNamesCapture** y pertenezca al paquete **GUI**.

Las ventanas de tipo JDialog se usan como ventanas auxiliares de una aplicación para mostrar información o leer información del usuario.

2. Añadir a la clase **PlayerNamesCapture** un atributo de tipo **ArrayList<String>**, denominado **names**, que alojará los nombres de los jugadores. Inicializar dicho atributo.
3. Añadir al constructor que Netbeans ha creado para dicha clase el código necesario para que se quede como se muestra a continuación. De esta manera, la aplicación finaliza si el usuario cierra esta ventana. Habrá que añadir algunos import's. Netbeans los sugiere.

```
public PlayerNamesCapture(JFrame parent, boolean modal) {  
    super(parent, modal);  
    initComponents();  
    this.addWindowListener (new WindowAdapter() {  
        @Override  
        public void windowClosing (WindowEvent e) {  
            System.exit(0);  
        }  
    });  
}
```

4. Sustituir el método **main** que Netbeans ha creado en la clase **PlayerNamesCapture** por este otro método. El método muestra el cuadro de diálogo y se queda esperando mientras el usuario interactúa con él. Cuando recupera el control, devuelve los nombres que se encuentran en el atributo **names**.

```
public ArrayList<String> getNames() {  
    this.setVisible(true);  
    return names;  
}
```

5. Ir a la pestaña Design y diseñar el cuadro de diálogo que contenga 3 **JLabels** Player 1, Player 2, Player 3 junto con sus respectivos 3 **JTextField** para que el usuario pueda escribir los nombres de los jugadores. Completar el diseño añadiendo un botón etiquetado como "Cancel" y otro etiquetado como "Play". Darle nombres significativos a las variables asociadas a los componentes que se van creando para que sea fácil referenciarlas en el código.
6. Asociar código a los botones "Cancel" y "Play"

- a) Al hacer doble-clic sobre un componente se crea el método asociado a dicho componente (si no existía ya) y se abre en el editor para que se pueda añadir el código asociado al evento hacer clic.
- b) El código asociado al botón "Cancel" solo será cerrar la aplicación: `System.exit(0)`.
- c) El código asociado al botón "Play" deberá rellenar el `ArrayList` `names` con los nombres que se han escrito en la interfaz de usuario y cerrar el cuadro de diálogo devolviendo el control.

```
names.add (name1.getText());  
//Se supone que name1 es el nombre que se le ha dado al JTextField para  
el nombre del primer jugador  
// Similar para los otros dos nombres  
this.dispose();
```

D) Comenzar el juego leyendo los nombres

1. Editar de nuevo el método **main** de **NapakalakiGame**.
2. Añadir una variable de tipo **ArrayList<String>** denominada **names**.
3. Añadir una variable de tipo **PlayerNamesCapture** denominada **namesCapture** que referencia a un objeto que se cree de dicha clase. Al constructor se le deben pasar dos parámetros:
 - a) El objeto de la clase **NapakalakiView** que se ha creado previamente. Ya que este cuadro de diálogo depende de dicha ventana principal.
 - b) El valor **true**. De este modo, la ejecución de la aplicación no continúa hasta que este cuadro de diálogo se haya cerrado.
4. Leer los nombres de los jugadores: **names = namesCapture.getNames()**
5. Iniciar el juego: **game.initGame(names)**
6. Visualizar la ventana principal (esto ya se tenía hecho: **napakalakiView.setVisible(true)**). Esta instrucción debe ser la última.
7. Ejecutar la aplicación, se abrirá el cuadro de diálogo para leer los nombres de los jugadores. Tras introducirlos y hacer clic en **Play**, se abrirá la ventana gráfica del dado. Tras detenerlo y hacer clic en **OK** se abrirá la ventana de la aplicación, que está todavía sin contenido.

Diseño e Implementación de Vistas**E) Diseñar e implementar la vista asociada a la clase *Treasure***

1. Añadir un archivo dentro del paquete **GUI** de tipo **JPanel Form** denominado **TreasureView**.

Los componentes de tipo **JPanel** se usan para agrupar otros componentes. En esta aplicación resultan útiles para representar las vistas de algunas de las clases del modelo. Un **JPanel** debe estar incluido, directa o indirectamente, en un **JFrame** para visualizarlo.

2. Realizar el diseño gráfico de la vista del tesoro, incluyendo su nombre, bonus, y tipo de tesoro. **Importante:** Recordar darle nombres significativos a los componentes que se usen, para que luego sea fácil referenciarlos en el código.
3. Intenta, en general, realizar diseños gráficos de las vistas que no ocupen mucho espacio. Todo el juego, es decir, la información del monstruo actual, la información del jugador actual incluyendo sus tesoros, tanto visibles como ocultos, así como diversos botones para realizar las distintas acciones del juego como combatir, equipar tesoros, etc. deben caber en la pantalla del ordenador.
4. Añadir un atributo de la clase **Treasure** denominado **treasureModel** que referenciará al tesoro al que representa esta vista.
5. Añadir un método público **setTreasure** que reciba como parámetro un **Treasure** y actualice los componentes que forman parte del diseño de su vista. Es decir, tal como se indica en la plantilla dada al principio del guion. Obviamente, no solo se debe actualizar el nombre, sino todos sus atributos.

F) Diseñar e implementar la vista asociada a la clase Player

1. Añadir un archivo dentro del paquete **GUI** de tipo **JPanel Form** denominado **PlayerView**.
2. Añadirle una variable de instancia de tipo **Player**, llamada **playerModel**, que referenciará al jugador que representa esta vista.
3. Realizar el diseño gráfico de la vista del jugador, incluyendo su nombre, nivel, etc. Si el jugador es sectario, se puede indicar con un **JLabel**, o incluso indicar cuántos sectarios hay en la partida en ese momento.
4. Incluir como parte del diseño gráfico de la vista del jugador dos **JPanel**, denominados **visibleTreasures** y **hiddenTreasures** que servirán para alojar las vistas de los tesoros visibles y ocultos de dicho jugador. Ponerle un borde para que sea distinguible y establecer el layout de dichos **JPanel** como **FlowLayout**. De esta forma, cuando se añadan mediante código, vistas de tesoros a dichos paneles, irán apareciendo uno al lado del otro.

Para ello, al hacer clic con el botón derecho del ratón en el **JPanel** se accede a su menú contextual donde:

1. Se puede acceder a las propiedades del componente para cambiarle el borde.
 2. Se puede establecer el Layout de dicho contenedor.
5. Añadir e implementar su correspondiente método **setPlayer**.
 6. Dado que el número de tesoros que tiene un jugador no se sabe en esta fase de diseño, sino que va a depender de cada momento de la partida, no se puede hacer un diseño estático del contenido de los paneles que van a contener las vistas de los tesoros. Es necesario añadir un método **fillTreasurePanel** que rellene los **JPanel** de los tesoros ocultos o visibles con las listas de los tesoros que en cada momento tenga el jugador actual:

```
private void fillTreasurePanel (JPanel aPanel, ArrayList<Treasure> aList) {  
    // Se elimina la información antigua  
    aPanel.removeAll();  
    // Se recorre la lista de tesoros construyendo y añadiendo sus vistas  
    // al panel  
    for (Treasure t : aList) {  
        TreasureView aTreasureView = new TreasureView();  
        aTreasureView.setTreasure (t);  
        aTreasureView.setVisible (true);  
        aPanel.add (aTreasureView);  
    }  
    // Se fuerza la actualización visual del panel  
    aPanel.repaint();  
    aPanel.revalidate();  
}
```

7. En el método **setPlayer**, junto con la actualización de sus diversos JLabels, se actualizarán los tesoros del jugador

```
public void setPlayer (Player p) {  
    playerModel = p;  
    // Incluir instrucciones para actualizar su nombre, nivel, etc.  
    // A continuación se actualizan sus tesoros  
    fillTreasurePanel (visibleTreasures, playerModel.getVisibleTreasures());  
    fillTreasurePanel (hiddenTreasures, playerModel.getHiddenTreasures());  
    repaint();  
    revalidate();  
}
```

8. Añadir botones para las acciones, “Steal Treasure”, “Make Visible”, “Discard Treasures” y “Discard All Treasures”; aunque por ahora no tendrán código asociado. El código asociado a dichos botones se explica en la sección J.

G) Proceder de manera similar con el resto de las vistas: **PrizeView**, **BadConsequenceView** y **MonsterView**.

1. Realizar el diseño gráfico de las distintas vistas comenzando por la más interna.
2. Implementar los correspondientes métodos set según la plantilla explicada al principio del guion.
3. Tener en cuenta que:
 - a) Para incluir una vista contenida (por ejemplo, **PrizeView**) en una vista contenedora (por ejemplo, **MonsterView**) se realiza arrastrando el fichero .java de la vista contenida sobre el diseño gráfico que se esté haciendo de la vista contenedora. Como si fuese un componente más.
 - b) Si al intentar lo que se indica en el apartado a) diera un error, compilar el proyecto (menú *Run*, opción *Clean and Build*) e intentarlo de nuevo.
 - c) Al implementar el método set de la vista contenedora habrá que llamar a los métodos set de las vistas contenidas.
4. Finalizar cada método set que actualiza una vista con la orden **repaint()** para que los cambios se hagan efectivos.
5. Hacer también una vista, denominada **PendingBadConsequenceView** que muestre solo los atributos que representan el mal rollo pendiente de cumplir. Añadir esta vista a la vista del jugador y modificar el método **setPlayer** para que se actualice también el pendingBadConsequence del jugador.

H) Finalizar la vista **NapakalakiView**

1. Realizar su diseño gráfico. Principalmente estará formada por dos vistas, la del jugador y la del monstruo, además de algunas etiquetas para mostrar el resultado del combate u otros mensajes, y los botones “Meet the Monster”, “Combat” y “Next turn”. El código asociado a dichos botones se explica en la sección J.
2. Completar el método setNapakalaki, actualizando los paneles de **currentMoster** y **currentPlayer**.

Procesamiento de las acciones del usuario

Ya se tiene la vista de las distintas clases que tienen una representación gráfica. Ahora se implementará el código que permita al usuario interactuar con la aplicación. En concreto:

1. Selección de tesoros. Necesario cada vez que el usuario quiera equiparse o descartarse de tesoros, así como en otras situaciones.
2. Pulsación de botones. Necesario para realizar las distintas acciones de la aplicación: combatir, equiparse de tesoros, avanzar de turno, etc.

I) Selección de tesoros

Para la selección de tesoros se añadirá un atributo booleano a la clase `TreasureView` con el que se representará si el tesoro está seleccionado por el usuario o no. Además, se añadirá código al método asociado al evento de hacer clic sobre el tesoro. Dicho código actualizará el atributo booleano antes mencionado y modificará algún atributo visual del tesoro, por ejemplo su color de fondo, para que el usuario tenga el feedback de los tesoros que tiene seleccionados en cada momento.

1. Abrir el fichero **TreasureView.java**
2. Añadir un atributo privado de tipo **boolean** denominado **selected** e inicializado a **false**.
3. Añadir un consultor para dicha variable, denominado **isSelected**.
4. Añadir un consultor, **getTreasure**, que devuelva el tesoro representado por esta vista, es decir, **treasureModel**.
5. Seleccionar la pestaña *Design*.
6. Hacer clic con el botón derecho del ratón sobre el fondo del `JPanel` de dicha vista. En el menú de contexto que se ha abierto elegir **Events**, a continuación **Mouse** y finalmente **mouseClicked**.
7. Se habrá creado el método que será llamado cuando el usuario haga clic sobre una vista de tesoro. El método se denomina **formMouseClicked**.
8. Incluir el código necesario para que si el tesoro ya estaba seleccionado (`selected==true`), deje de estarlo; y si no estaba seleccionado pase a estar seleccionado.
9. Incluir también algún cambio en el aspecto visual del tesoro (algún cambio de color) para que el usuario sepa qué tesoros tiene seleccionados en cada momento. Por ejemplo, si al diseñar el `JPanel` de los tesoros se le pone un color de fondo, se puede mostrar o no, dependiendo de si está seleccionado. Para ello se pueden usar los métodos `setBackground(Color)` y `setOpaque(boolean)`. Este último método, con el parámetro `true` hará visible el color de fondo, y con el parámetro `false` lo ocultará.
10. Concluir con la orden `repaint()` para que los cambios visuales se hagan efectivos.

J) Pulsación de botones

La implementación de cada pulsación sobre un botón supondrá normalmente 3 fases:

1. Recopilar información de la GUI, por ejemplo, qué tesoros están seleccionados.
2. Enviarle mensajes al modelo para que realice la acción requerida.
3. Llamar a métodos set de las vistas para que éstas se actualicen.

Se explica con el ejemplo de la acción de *equipamiento de tesoros de un jugador* (botón "Make Visible"). El resto de acciones se implementan de manera similar.

1. Abrir el fichero `PlayerView.java`
2. Dado que desde la vista del jugador es necesario enviarle mensajes al modelo `Napakalaki`, añadir en la clase **PlayerView** una variable de instancia privada denominada **napakalakiModel** de la clase **Napakalaki** y su método set correspondiente. Este atributo puede ser inicializado desde el método `setNapakalaki` de la clase `NapakalakiView`.

3. Añadir un método para obtener la lista de tesoros seleccionados:

```
private ArrayList<Treasure> getSelectedTreasures(JPanel aPanel) {
    // Se recorren los tesoros que contiene el panel,
    // almacenando en un vector aquellos que están seleccionados.
    // Finalmente se devuelve dicho vector.

    TreasureView tv;
    ArrayList<Treasure> output = new ArrayList();
    for (Component c : aPanel.getComponents()) {
        tv = (TreasureView) c;
        if ( tv.isSelected() )
            output.add ( tv.getTreasure() );
    }
    return output;
}
```

4. En la pestaña de diseño, hacer doble clic sobre el botón “Make Visible”. Se creará el método asociado a dicho botón situándose el cursor en él para implementarlo.

5. Realizar la implementación de dicho método según las tres fases descritas anteriormente:

a) Recopilar información de la GUI.

```
ArrayList<Treasure> selHidden = getSelectedTreasures (hiddenTreasures);
```

b) Enviar mensajes al modelo para que se desarrolle la acción.

```
napakalakiModel.makeTreasuresVisible (selHidden);
```

c) Actualizar la vista.

```
setPlayer (napakalakiModel.getCurrentPlayer());
```

Últimas consideraciones

Importante: Téngase en cuenta que no se puede realizar cualquier acción en cualquier momento. Por ejemplo, el jugador no se puede equipar tesoros después de haber conocido al monstruo. Para recordar qué se puede hacer en cada fase del turno de un jugador, se puede consultar las reglas del juego o la interfaz de texto que se usó en la práctica 3.

Por lo tanto, se tienen que ir habilitando o deshabilitando los diferentes botones según las acciones permitidas en cada momento. Estudiar en qué métodos puede hacerse la habilitación o deshabilitación de los distintos botones.

La habilitación/deshabilitación se realiza enviando el mensaje **setEnabled** al botón correspondiente. Dicho método tiene un parámetro de tipo **boolean**.

Al principio del turno de un jugador, el monstruo no puede mostrarse hasta que el jugador solicite conocerlo y combatir contra él. La vista del monstruo se puede ocultar y mostrar enviando el mensaje **setVisible** a la vista del monstruo. Dicho método tiene un parámetro de tipo **boolean**.

Si al ocultar la vista del monstruo en tiempo de ejecución “se desordenan” el resto de componentes de la vista **NapakalakiView**, se le puede poner el layout denominado **Null Layout** al **JFrame** de dicha vista para evitarlo.