

Alpaca Emblem

Tarea 3: Hasta la vista

Profesor: Alexandre Bergel

Auxiliares: Juan-Pablo Silva
Ignacio Slater

Semestre: Primavera 2019

Resumen

Para esta entrega ya tienen la mayor parte del código necesario para poder construir un juego.

Lo último que queda por implementar es la **vista**. La vista es la interfaz gráfica con la que interactúa directamente el usuario. Este elemento no tiene absolutamente nada de conocimiento del comportamiento del juego y sólo debe preocuparse de capturar las acciones del usuario e informárselas al controlador (por ejemplo, lo único que necesita saber la vista es que el usuario presionó un botón A y le notifica eso al controlador, pero no tiene conocimiento de qué acción se ejecutará en el juego al ocurrir esto).

Junto con este documento se le entregará una interfaz básica que puede extender para desarrollar lo que resta del juego, agregando todo lo que falte para que cumpla con los requisitos. El código entregado **puede** ser modificado tanto como se quiera mientras se mantengan buenas metodologías de diseño.

1. Requisitos

Al igual que en la entrega anterior se parte del supuesto de que su tarea anterior estaba correctamente implementada, por lo que, en caso de tener alguno, se le solicita que arregle los errores cometidos.

Lo que se le solicita para completar el proyecto es crear una interfaz gráfica para el juego, además de hacer las modificaciones necesarias al modelo y al controlador para integrar el nuevo componente.

A continuación se explican en detalle las funcionalidades que debe agregar y las modificaciones que tendrá que realizar.

1.1. Cálculo de distancias

Como habrá notado en su tarea anterior, la manera en la que se calculan las distancias es ineficiente y toma demasiado tiempo en ejecutarse como para que sea factible jugar una partida.

Lo que causa la ineficiencia es que el algoritmo utilizado actualmente para calcular los caminos más cortos (*Dijkstra*) asume que no se tiene nada de información sobre el grafo en el que opera y por lo tanto su tiempo de ejecución es $O(|E| + |V| \log |V|)$, lo que en el caso de un mapa de tamaño $n \times n$ sería $O(n^2 \log(n))$. Entonces, en un mapa de 7×7 celdas para encontrar el camino más corto entre dos celdas se requerirían **a lo menos** 137 operaciones.

Sin embargo, para el caso del juego, se tiene bastante información de cómo está compuesto el grafo. Se sabe que la distancia entre todo par de nodos conectados entre sí será siempre 1, y además se tiene que ningún nodo tendrá más de cuatro vecinos. Lo que se buscará entonces es modificar el algoritmo que calcula las distancias para que aproveche esta información.

De ahora en adelante nos referiremos al algoritmo de cálculo de distancias como *A*.

1.1.1. Limitar el rango de búsqueda

La primera optimización que se hará es reducir el área de búsqueda de caminos.

Como el cálculo de distancias se hace para comprobar que las acciones de una unidad caigan dentro de su rango es fácil ver que cualquier distancia mayor al rango no cumplirá con ser la distancia más corta dentro del alcance de la unidad, y por lo tanto pueden ser descartados.

Entonces, siendo r el rango de acción de la unidad, lo que se hará será tomar detener la recursión del algoritmo si ésta es más larga que el rango deseado. Considerando esto, el grafo de búsqueda siempre será más pequeño que uno de dimensiones $(2r + 1) \times (2r + 1)$ donde la unidad estará en el nodo central del grafo. Además, como se sabe que la distancia entre nodos vecinos es a lo más 1, se puede saber si una celda no forma parte del subgrafo (y por lo tanto está fuera del rango de la unidad) solamente sabiendo sus coordenadas.

Con la optimización anterior se tiene que el nuevo tiempo de ejecución del algoritmo A será:

$$O(A) = \begin{cases} O(r^2 \log(r)) & \text{si la celda de destino está dentro del subgrafo} \\ O(1) & \text{sino} \end{cases}$$

Lo anterior siempre será menor o igual al tiempo original.

A continuación puede ver la implementación de dicha optimización.

```
public double distanceTo(final Location otherNode, int range) {
    return shortestPathTo(otherNode, range, new HashSet<>());
}

private double shortestPathTo(final Location otherNode, int range,
    final Set<Location> visited) {
    if (otherNode.equals(this)) {
        return 0;
    }
    visited.add(this);
    double distance = Double.POSITIVE_INFINITY;
    if (range != 0 && range >= minDistanceTo(otherNode)) {
        for (Location node :
            neighbours) {
            if (!visited.contains(node)) {
                iterations++;
                distance = Math.min(distance,
                    1 + node.shortestPathTo(otherNode, --range, new HashSet<>(visited)));
            }
        }
    }
    return distance;
}

private int minDistanceTo(final Location otherLocation) {
    return Math.abs(this.row - otherLocation.row) + Math
        .abs(this.column - otherLocation.column);
}
```

Note que para limitar la profundidad de la recursión se le pasa un parámetro adicional

1.1.2. Uniform Cost Search

Si bien la optimización anterior mejora en gran parte el rendimiento, sigue existiendo el caso en que las búsquedas en el subgrafo tomen tiempo $O(r^2 \log(r))$, y si r es cercano a n se cae en el mismo problema original.

Para enfrentar esto se debe optimizar *Dijkstra*. Como se mencionó antes, el algoritmo que utilizamos hasta el momento se basa en que no se tiene nada de información sobre el grafo, eso se puede ver en el primer paso de *Dijkstra* en el que se define la distancia hacia todos los nodos como infinito.

Para el caso del juego se sabe que el mapa es conexo por lo que ningún nodo estará a distancia infinita de otro, e incluso se puede saber cuál sería la distancia máxima que podría separar a dos nodos.¹

UFC se diferencia de *Dijkstra* en que el orden en que revisa los nodos no es cualquiera. La idea general es siempre recorrer primero el camino más corto desde el nodo inicial.

Algorithm 1 *Uniform Cost Search*

Require: *nodoInicial*, *nodoFinal*

Ensure: *caminoMnimo* \equiv el camino más corto de *nodoInicial* hasta *nodoFinal*

Se crea el conjunto de nodos por visitar como una cola de prioridad

Se agrega *nodoInicial* a la cola con prioridad 0

while La cola no esté vacía \wedge No se ha alcanzado *nodoFinal* **do**

candidato \leftarrow El camino con **menor prioridad** de la cola

if El camino termina en *nodoFinal* **then**

caminoMnimo \leftarrow *candidato*

else

 Inserta todos los caminos que se pueden alcanzar desde el último nodo de *candidato* a la cola con su distancia desde *nodoInicial* como prioridad

end if

end while

En el siguiente enlace se muestra una explicación alternativa del algoritmo en el que se va formando un árbol con los caminos que ha recorrido (de esta forma, luego se puede obtener el camino más corto utilizando *BFS*): <https://youtu.be/dRMvK76xQJI>. Dicha implementación es equivalente a utilizar una cola de prioridad pero más difícil de implementar (razón por la que se suele utilizar una cola).

También es más fácil evaluar la eficiencia del algoritmo en el caso de utilizar un árbol, puesto que la complejidad de *BFS* es conocida e igual a $O(|V| + |E|)$. Como en el caso del juego se tiene que $|V| < |E|$, se puede expresar la eficiencia del algoritmo en el peor caso como $O(|E|)$.

Ahora, aplicando *UFC* y la optimización de la sección anterior se puede concluir que:

$$O(A) = \begin{cases} O(r^2) & \text{si la celda de destino está dentro del subgrafo} \\ O(1) & \text{sino} \end{cases}$$

A continuación se presentará una implementación del algoritmo anterior junto con la optimización realizada en la sección anterior. Pero antes debe introducirse el concepto de camino.

Caminos Un camino se definirá como una secuencia (L_0, L_1, \dots, L_k) de ubicaciones conectadas entre si. En un camino siempre se cumplirá que:

- Todo par de ubicaciones (L_i, L_{i+1}) , estarán conectadas por un arco de largo 1.

¹Se podría aprovechar esta información para optimizar aún más la búsqueda de caminos con un algoritmo llamado A^* .

- Ninguna ubicación puede aparecer más de una vez en el camino.

Con la definición anterior se puede crear una clase `Path` como sigue:

```
// model.map.Path
public class Path implements Comparable<Path> {

    private final Location last;
    private final int length;
    private final List<Location> locations;

    public Path(Path prevPath, Location nextLocation) {
        locations = new LinkedList<>(
            prevPath != null ? prevPath.locations : Collections.emptyList());
        locations.add(nextLocation);
        last = nextLocation;
        length = locations.size() - 1;
    }
}
```

El código anterior presenta la estructura básica de un camino:

- `last` una referencia al final del camino.
- `length` es el largo del camino (note que el largo se define en base al número de arcos).
- `locations` es la lista de celdas que componen el camino.

Los caminos se irán construyendo agregando un nodo a la vez, así que el primer parámetro del constructor es el camino al que se va a agregar la nueva ubicación. En caso de que se quiera crear un camino que contenga solamente una ubicación, entonces se le debe entregar al constructor un camino nulo como parámetro (es importante notar que podría definirse una clase `NullPath` para evitar hacer la comparación `prevPath != null`, pero dado que esta comparación sucede en sólo una línea del código no tiene sentido crear una clase nueva).

Ahora, la clase `Path` implementa la interfaz `Comparable`, esto significa que la clase puede compararse con otras para determinar cuál es mayor. Dicha comparación se hace mediante el método `compareTo(Object)`, este método retorna un entero que puede ser comparado con otro. Las colas de prioridad en *Java* utilizan esto para determinar la prioridad entre objetos.

En este caso, la prioridad se definirá por el largo del camino, por lo que el método se puede definir como:

```
// model.map.Path
@Override
public int compareTo(Path another) {
    return length;
}
```

Por último, se necesitará una manera de comprobar las ubicaciones que se pueden alcanzar desde el final del camino (evitando las ubicaciones que ya están contenidas en el camino para no caer en un *loop* infinito) y una forma de comprobar si el camino termina en el nodo de destino. Esto se puede ver en el siguiente segmento:

```

// model.map.Path
public boolean endsIn(final Location location) {
    return last == location;
}

public List<Location> reachableLocations() {
    return last.getNeighbours().stream()
        .filter(location -> !locations.contains(location))
        .collect(Collectors.toCollection(LinkedList::new));
}

```

Shortest path Una vez que se tienen definidos los caminos se puede implementar el cálculo de caminos optimizado como:

```

// model.map.Location
public double distanceTo(final Location otherNode, int range) {
    return shortestPathTo(otherNode, range);
}

private double shortestPathTo(final Location otherNode, int range) {
    if (otherNode.equals(this)) {
        return 0;
    }
    Path shortestPath = null;
    Queue<Path> toVisit = new PriorityQueue<>();
    toVisit.add(new Path(null, this));
    while (!toVisit.isEmpty() && shortestPath == null) {
        final Path candidate = toVisit.poll();
        final int candidatePriority = candidate.getLength();
        if (candidate.endsIn(otherNode)) {
            shortestPath = candidate;
        } else {
            for (final Location neighbour : candidate.reachableLocations()) {
                if (candidatePriority < range) {
                    toVisit.add(new Path(candidate, neighbour));
                }
            }
        }
    }
    return shortestPath == null ? Double.POSITIVE_INFINITY : shortestPath.getLength();
}

```

Note que además esta versión optimizada ya no necesita de recursión y por lo tanto de un parámetro con el conjunto de nodos visitados, ni del método auxiliar `minDistanceTo(Location)` definido en la sección anterior.

1.1.3. Distancias frecuentes

La última optimización que se tendrá en consideración, y que **deberá implementar por su cuenta** es la de no calcular dos veces la misma distancia.

Además, dada la naturaleza del juego hay celdas que serán más comunmente accedidas que otras y algunas que nunca serán accedidas. Por esto, no tiene sentido que se calcule la distancia a las celdas a menos que sea necesario, por lo que las distancias entre dos ubicaciones deberá ser computada *on-demand*.

Para lo anterior note lo siguiente, sea $S = (e_0, e_1, \dots, e_k)$ el camino más corto entre las celdas v_i y v_j , al ser el camino más corto en particular contiene todos los caminos más cortos entre todo par de nodos por los que pasa el camino. Teniendo esto en cuenta, al momento de calcular la distancia más corta entre dos nodos ya se calcularon las distancias más cortas para todos los nodos del camino, por lo que no debieran calcularse de nuevo.

Con lo anterior, se tendrá que el nuevo orden de ejecución del algoritmo será:

$$O(A) = \begin{cases} O(r^2) & \text{si el camino es más corto que el rango y no ha sido calculado} \\ O(1) & \text{sino} \end{cases}$$

1.2. Flujo del juego

Llamaremos *flujo* a la **secuencia de acciones que se puedan realizan durante la ejecución de la aplicación**, puede pensar en ella como los pasos que se deben hacer para que la aplicación pase de un estado A a uno B .

Para facilitar la explicación, el flujo se separará en 2 conjuntos.

1.2.1. Configuración de la partida

En esta parte del flujo es en la que se definen todas las condiciones previas a jugar una partida. Entre ellas: establecer la cantidad de jugadores y unidades, definir las dimensiones del mapa, seleccionar las unidades de cada jugador y todo lo que sea necesario para poder jugar.

La figura 1 plantea una **posible** implementación de esta etapa.

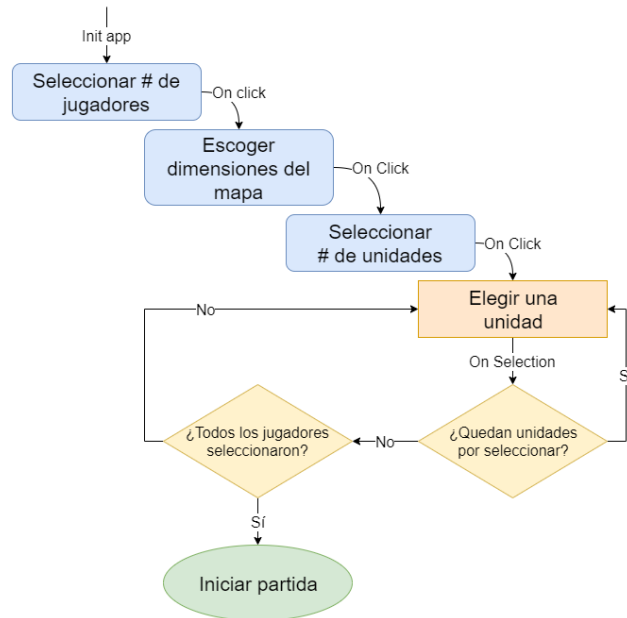


Figura 1: Diagrama de flujo de la configuración de la partida

Note que en el diagrama no se incluye la selección de items. Puede omitir esta etapa del flujo en su implementación para simplificar su trabajo predefiniendo los items con los que comenzará cada tipo de unidad.

1.2.2. Desarrollo de la partida

El flujo de esta fase queda como tarea propuesta.

Esta fase es la que representa la partida en sí, y debe por lo tanto definir el flujo principal del juego. Para esto tenga en cuenta qué acciones deben realizarse antes que otras y qué acciones representarían un cambio de estado.

A continuación se muestran algunas secuencias de acciones que deben poder realizarse en su implementación del flujo.

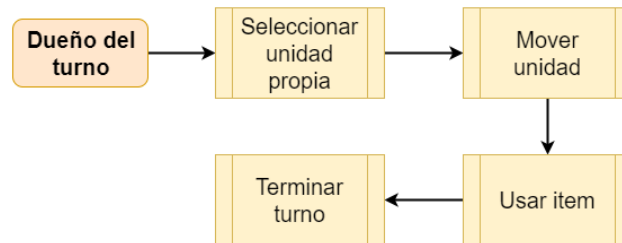


Figura 2: Ejemplo de selección de unidad y uso de un objeto

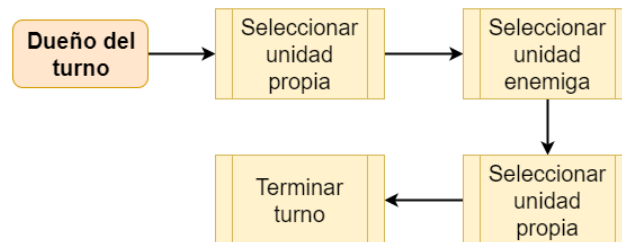


Figura 3: Ejemplo de selección de múltiples unidades en el mismo turno

Los ejemplos anteriores en ningún caso cubren todas las acciones posibles del jugador, y están solamente para fines ilustrativos.

Note que el jugador puede terminar su turno en cualquier momento.

Tiene libertad al momento de definir cómo será el flujo del juego, pero deben poder realizarse todas las funcionalidades solicitadas en las tareas anteriores. Incluya en el *readme* los diagramas de flujo necesarios para mostrar claramente los estados del juego o esta sección **no será corregida**.

1.3. Interfaz gráfica

Lo último que se le solicitará para esta entrega es que implemente una interfaz gráfica simple para el juego utilizando *Swing*.² Para simplificar su trabajo dispondrá de una interfaz gráfica simple³ con todas las componentes que podría necesitar para crear una interfaz simple.

²Puede encontrar la especificación completa de la API de *Swing* en <https://docs.oracle.com/javase/7/docs/api/index.html>.

³Puede asumir que el código entregado sigue buenas prácticas de diseño.

Usted tendrá total libertad respecto a como diseñar la interfaz gráfica con las mismas restricciones que se definieron para el flujo.

Se le recomienda encarecidamente que no utilice los métodos de *Swing* para dibujar figuras, esto debido a que será una complicación adicional para su implementación. En vez de eso, puede utilizar *sprites* para representar los elementos del juego.

La figura 4 muestra un ejemplo de una posible interfaz gráfica en la que puede inspirarse para su implementación. Tenga en cuenta que el resultado esperado puede ser mucho más simple que el mostrado en el ejemplo.



Figura 4: Ejemplo de interfaz gráfica

Aparte de crear la interfaz gráfica, debe conectar la interfaz desarrollada con el controlador del juego. Puede ver un ejemplo simple de esto en <https://github.com/islaterm/el-rey-de-los-patronos>.

1.3.1. Código base

A continuación se explicarán las clases incluidas en el código base entregado.

Ventana principal Este es el punto de entrada de la aplicación. Aquí se define la ventana en la que estará contenido el juego, sus dimensiones, y el contenido inicial que se mostrará.

```
public class AlpacaEmblem extends JFrame {  
    private JPanel mainPanel;
```



```

private AlpacaEmblem() throws HeadlessException {
    mainPanel = new ViewsContainer(this);
    setupFrame();
}

public static void main(String[] args) {
    EventQueue.invokeLater(AlpacaEmblem::run);
}

private static void run() {
    AlpacaEmblem game = new AlpacaEmblem();
    game.setVisible(true);
}

private void setupFrame() {
    this.setTitle("Alpaca Emblem");
    this.add(mainPanel, BorderLayout.CENTER);
    this.setSize(720, 720);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}

```

La clase `JFrame` representa una ventana, al hacer que `AlpacaEmblem` la extienda se define esta clase como una ventana y por lo tanto como una aplicación. El método `setupFrame()` se encarga de hacer las configuraciones iniciales de la aplicación.

La instrucción `this.add(mainPanel, BorderLayout.CENTER)` añade un componente a la ventana y define la manera en que se alineará dentro de ésta, el detalle de los *layouts* de *Swing* los puede ver en <https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>.

La línea `this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` indica lo que hará el programa al cerrarse (en este caso detener la ejecución de la aplicación).

En el constructor se define `mainPanel` como una instancia de `ViewsContainer`. La implementación de esta clase se explicará más adelante.

Finalmente, el método `main` hace una llamada a `EventQueue.invokeLater(AlpacaEmblem::run)`, esto hará que la aplicación sea ejecutable. La notación `AlpacaEmblem::run` es una manera de pasarle una referencia estática al método `run` a `EventQueue`.

Views container Esta clase representa un contenedor para todas las vistas del juego y definirá que es lo que se mostrará en la ventana en todo momento.

```

public class ViewsContainer extends JPanel {
    private final JFrame game;

    public ViewsContainer(final JFrame game) {
        super(new CardLayout());
        this.game = game;
        setupViews();
    }

    private void setupViews() {

```

```

        this.add(new LandingView(this));
        this.add(new GameView());
    }
    public void changeFrameDimensions(final int width, final int height) {
        game.setSize(width, height);
    }

    public void nextView() {
        ((CardLayout) super.getLayout()).next(this);
    }
}

```

El constructor de la clase recibe una referencia a la ventana, esto es para que conozca el contexto en el que se encuentra y pueda enviar mensajes a éste. Esto se usa, por ejemplo, en el método `changeFrameDimensions(int, int)`.

Además de lo anterior, el constructor llama al constructor de su superclase pasándole un `CardLayout` como parámetro. Este *layout* permite tener varias vistas dentro de un panel e ir cambiando entre ellas. En este caso las vistas serán `LandingView` y `GameView`.

Landing view Ésta será la primera vista con la que se encontrará el usuario. En este caso, la vista se compone de una imagen de fondo y un botón que permite pasar a la vista siguiente.

```

// gui.views.LandingView
public class LandingView extends ImagePanel {

    private static final String BACKGROUND_IMAGE = "resources/landing_page.jpg";
    private final ViewsContainer container;
    private JButton changeViewBtn;

    public LandingView(final ViewsContainer container) {
        super(BACKGROUND_IMAGE);
        this.container = container;
        this.setLayout(null);
        setupChangeViewButton();
        this.add(changeViewBtn);
    }

    private void setupChangeViewButton() {
        changeViewBtn = new JButton("Change view");
        changeViewBtn.addActionListener(e -> {
            LandingView.this.changeContainerDimensions(1280, 720);
            container.nextView();
        });
        changeViewBtn.setBounds(panelBackground.getWidth(null) / 2 - 160,
            panelBackground.getHeight(null) / 2 - 20, 320, 40);
    }

    private void changeContainerDimensions(final int width, final int height) {
        container.changeFrameDimensions(width, height);
    }
}

```

```
}
}
```

La clase se define como una extensión de `ImagePanel` que es una subclase de `JPanel`, debido a su simpleza se omitirá la implementación de esta clase, pero forma parte del código base que se le entrega.

La instrucción `this.setLayout(null)` del constructor indica que este panel no tendrá ninguna organización en particular. Para este ejemplo, tanto la imagen de fondo como el botón son colocados en posiciones específicas, por lo que no necesitan de un *layout* específico.

El método `setupChangeViewButton()` define un botón, el texto que contiene, su posición con la llamada a `setBounds(...)` y lo que realizará al momento de ser presionado.

En el contexto de *Swing* un botón es un *Observer* que espera ser notificado de que el usuario hizo click sobre él. Para definir lo que sucede cuando se presiona el botón se usa el método `addActionListener(ActionEvent -> void)`, la notación utilizada significa que el método recibe una función anónima como parámetro, dicha función aceptará un evento `e` y realizará una acción.

Existen 2 maneras de definir la acción a realizar, la que se utiliza en el código base es:

```
// gui.views.LandingView
changeViewBtn.addActionListener(e -> {
    LandingView.this.changeContainerDimensions(1280, 720);
    container.nextView();
});
```

La otra forma de hacerlo es extraer la función anónima a un método concreto dentro de la clase, esto puede ser útil si la acción que se realizará al recibir el evento es compleja.⁴ El ejemplo anterior escrito de la manera alternativa sería:

```
// gui.views.LandingView
private void setupChangeViewButton() {
    changeViewBtn = new JButton("Change view");
    changeViewBtn.addActionListener(this::actionPerformed);
    changeViewBtn.setBounds(panelBackground.getWidth(null) / 2 - 160,
        panelBackground.getHeight(null) / 2 - 20, 320, 40);
}

private void actionPerformed(ActionEvent e) {
    LandingView.this.changeContainerDimensions(1280, 720);
    container.nextView();
}
```

En este caso lo que sucederá al momento de presionar al botón será cambiar las dimensiones de la ventana y pasar a la siguiente vista.

Game view El código de esta vista es relativamente simple, pero se explicará ya que cumple un rol fundamental en la aplicación. Ésta es la vista donde se muestra toda la información del juego.

A continuación el código:

⁴Debe intentar que las acciones que se realizan sean lo más simples posibles.

```
// gui.views.game.GameView
public class GameView extends JSplitPane {

    public GameView() {
        super(HORIZONTAL_SPLIT, new FieldPane(), new InfoPane());
        this.setDividerLocation(1280 * 3 / 4);
        this.setEnabled(false);
    }
}
```

La clase `GameView` se define como una subclase de `JSplitPane`, esto significa que esta vista estará dividida en dos o más paneles (en este ejemplo se divide en 2).

El primer parámetro del constructor de `JSplitPane` es la orientación de la división de la vista, y los parámetros siguientes son los paneles que compondrán las áreas. En caso de necesitar crear más subdivisiones, entonces alguna de las vistas de la división debe heredar también de `JSplitPane`.

Es importante notar que por defecto la división que se hará de la ventana será al centro y se podrá modificar su tamaño, las últimas 2 líneas del constructor se encargan de evitar eso.

Game view Esta clase representa el panel derecho del *game view*, en ésta se ejemplifican varias herramientas que podrán serle útiles al momento de desarrollar su aplicación.

Para simplificar la explicación se omitirán los segmentos de código que no se consideren fundamentales. A continuación el código:

```
// gui.views.game.InfoPane
private void setupLayout() {
    this.setLayout(new BoxLayout(this, BoxLayout.PAGE_AXIS));
    Border margin = new EmptyBorder(10, 10, 10, 10);
    this.setBorder(new CompoundBorder(this.getBorder(), margin));
}
```

El método `setupLayout()` define la organización de los elementos que se mostrarán en el panel y define un borde de 10 píxeles para que los elementos estén separados del borde del panel.

```
// gui.views.game.InfoPane
private void addLabel(String text, int verticalSpace, String fontName, int fontSize,
    int fontStyle) {
    JLabel label = new JLabel(text);
    label.setFont(new Font(fontName, fontStyle, fontSize));
    this.add(label);
    if (verticalSpace >= 0) {
        this.add(Box.createRigidArea(new Dimension(0, verticalSpace)));
    }
}
```

El código anterior agrega una nueva etiqueta de texto al panel, con las propiedades del *font* que se desee utilizar y, además, se define un espacio en blanco que habrá luego del texto (esto con `this.add(Box.createRigidArea(new Dimension(0, verticalSpace)))`)

Adicionalmente, el panel tiene una lista elementos seleccionables que, en este caso, cambiarán el color de fondo del panel. Esto se logra con los siguientes métodos:

```
// gui.views.game.InfoPane
private void setupColorPicker() {
    JList<String> colorList = new JList<>(new String[]{
        "Red",
        "Green",
        "Blue",
        "Gray",
        "White"
    });
    colorList.getInputMap().put(KeyStroke.getKeyStroke(Keys.DOWN), "none");
    colorList.setSelectedIndex(0);
    colorList.setLayout(new BoxLayout(colorList, BoxLayout.PAGE_AXIS));
    removeBindings(colorList, Keys.DOWN, Keys.LEFT, Keys.RIGHT, Keys.UP);
    this.add(colorList);

    JButton pickColorBtn = new JButton("Change color");
    pickColorBtn.addActionListener(e -> {
        paneColor = COLORS[colorList.getSelectedIndex()];
        repaint();
    });
    this.add(pickColorBtn);
}

@Override
protected void paintComponent(final Graphics graphics) {
    graphics.setColor(paneColor);
    graphics.fillRect(0, 0, this.getWidth(), this.getHeight());
}
```

Los métodos `repaint()` y `paintComponent(Graphics)` son heredados de `JPanel` y son los que se encargan de dibujar sobre el lienzo del panel. Lo único que necesita saber sobre estos métodos es que al llamar a `repaint()` se enviará el mensaje `paintComponent(Graphics)` y se refrezcará el panel para mostrar la nueva información. En cualquier momento en el que deba actualizar algo en la pantalla deberá usar estos métodos.

Por último, la lista puede ser manejada utilizando los botones del teclado pero, en este caso, ese no es un comportamiento deseable ya que interferiría con el manejo del juego. Para desvincular una tecla de un componente esto se debe hacer explícitamente con el método que se muestra a continuación:

```
// gui.views.game.InfoPane
private void removeBindings(JComponent component, String... keys) {
    for (String key :
        keys) {
        component.getInputMap().put(KeyStroke.getKeyStroke(key), "none");
    }
}
```

1.4. Bonificaciones

Además del puntaje asignado por cumplir con los requisitos anteriores, puede recibir puntos adicionales (*que se sumarán a su nota total*) implementando lo siguiente:

- **Interfaz gráfica avanzada** (0.5 pts): Si su interfaz gráfica cumple más de los requisitos mínimos podrá recibir una bonificación de hasta 0.5 pts (esto quedará a criterio del ayudante que le revise).
- **Manejo de excepciones** (0.3 pts): Se otorgará puntaje por la correcta utilización de excepciones para manejar casos de borde en el juego. Recuerde que atrapar *runtime exceptions* es una mala práctica, así como arrojar un error de tipo `Exception` (esto último ya que no es un error lo suficientemente descriptivo). Bajo ninguna circunstancia una de estas excepciones debiera llegar al usuario.
- **Benchmarking** (0.3 pts): Cree *tests* que muestren que su implementación del cálculo de distancias efectivamente mejora la eficiencia del programa considerando la implementación original del algoritmo y las 2 optimizaciones presentadas en este documento además de la que usted implementó. Agregue al *readme* los resultados de dichos *tests*.

1.5. Requerimientos adicionales

Además de una implementación basada en las buenas prácticas y técnicas de diseño vistas en clases, usted debe considerar:

- **Cobertura:** Cree los *tests* unitarios, usando JUnit 5, que sean necesarios para tener al menos un coverage del 90 % de las líneas por paquete. Para esta entrega no debe testear las interfaces gráficas dado que no posee las herramientas necesarias para eso, así que debe asegurarse de que las interfaces gráficas estén en su propio paquete. El porcentaje de coverage se medirá respecto al resto de los paquetes. Todos los *tests* de su proyecto deben estar en el paquete `test`.
- **Javadoc:** Cada interfaz, clase pública y método público deben estar debidamente documentados siguiendo las convenciones de Javadoc⁵. En particular necesita `@author` y una pequeña descripción para su clase e interfaz, y `@param`, `@return` (si aplica) y una descripción para los métodos.
- **Resumen:** Debe entregar un archivo **pdf** que contenga su nombre, rut, usuario de Github, un link al repositorio de su tarea y un diagrama UML que muestre las clases, interfaces y métodos que usted definió en su proyecto. **No debe incluir los tests** en su diagrama.
- **Git:** Debe hacer uso de git para el versionamiento de su proyecto. Luego, esta historia de versionamiento debe ser subida a Github.
- **Readme:** Debe hacer un *readme* especificando los detalles de su implementación, los supuestos que realice y una breve explicación de cómo ejecutar el programa. Adicionalmente se le solicita dar una explicación general de la estructura que decidió utilizar, los patrones de diseño y la razón por la cual los utiliza.

Además debe considerar los requisitos que se especifican en el resumen del proyecto.

2. Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
 - **Funcionalidad (1.5 puntos):** Se analizará que su código provea la funcionalidad pedida. Para esto, se exigirá que testee las funcionalidades que implementó⁶. **Si una funcionalidad no se testea, no se podrá comprobar que funciona y, por lo tanto, NO SE ASIGNARÁ PUNTAJE por ella.**

⁵<http://www.oracle.com/technetwork/articles/java/index-137868.html>

- **Diseño (2.5 puntos):** Se analizará que su código provea la funcionalidad implementada utilizando un buen diseño.
- **Coverage (1.0 puntos):** Sus casos de prueba deben crear diversos escenarios y contar con un coverage de las líneas de al menos 90 % por paquete. No está de más decir que sus tests deben testear algo (es decir, no ser tests vacíos o sin asserts).
- **Javadoc (0.5 puntos):** Cada clase, interfaz y método público debe ser debidamente documentado. Se descontará por cada falta.
- **Resumen (0.5 puntos):** El resumen mencionado en la sección anterior. Se evaluará que haya incluido el diagrama UML de su proyecto. **En caso de no enviarse el resumen con el link a su repositorio su tarea no será corregida.**⁷

⁶Se le recomienda enfáticamente que piense en cuáles son los casos de borde de su implementación y en las fallas de las que podría aprovecharse un usuario malicioso.

⁷Porque no tenemos su código.