Programación para Ingeniería Telemática

Curso 2018-19

Práctica 5: Genéricos e iteradores.

1. Objetivos de aprendizaje. Estructura de la práctica.

Objetivos

- Ser capaces de utilizar métodos genéricos.
- Ser capaces de utilizar listas genéricas y recorrerlas con iteradores.
- Ser capaces de utilizar factorías de objetos.
- Ser capaces de utilizar los eventos de un temporizador para actualizar el juego.
- Ser capaces de utilizar los eventos de teclado para controlar el movimiento del personaje principal.
- Presentar una estructura básica del juego que pueda ser utilizada para el desarrollo de la práctica evaluable.

En esta práctica abordaremos genéricos, listas, expresiones lambda y factorías de objetos. Parte del código definido en las prácticas anteriores tendrá que ser reescrito para aprovechar la capacidad expresiva de los nuevos conceptos aprendidos en teoría. En esta práctica se definirá además una estructura básica para la programación del juego.

Contenidos de la memoria

Esta memoria se estructura en cinco secciones con los siguientes contenidos:

Sección 1: Objetivos y contenido de la práctica.

Sección 2: Los paquetes `common`, `game` y `views`. En esta sección se presenta el código de partida que tendrá que usarse para realizar esta práctica en concreto. Este código se utilizará como base para la realización de la práctica evaluable.

Sección 3: Enunciado de los ejercicios a realizar. Estos ejercicios están pensados para facilitar la realización de la práctica entregable correspondiente al bloque de la asignatura dedicado a la programación orientada a objetos. Además, los ejemplos y ejercicios que se vayan mostrando en las prácticas y o bien son semejantes a los que pueden pedirse en los exámenes de la asignatura o bien constituyen un paso previo para ser capaces de realizarlos.

Antes de hacer los ejercicios es aconsejable tener una visión de conjunto, por ello **se recomienda leer detenidamente el boletín de prácticas y entender el código suministrado antes de abordar la resolución de los ejercicios.**

2. Los paquetes `common`, `game`, y `views`.

2.1. El paquete `common`

Sin cambios relevantes.

2.2. El paquete `game`

Se presentan nuevas clases e interfaces y nuevas implementaciones de clases ya existentes.

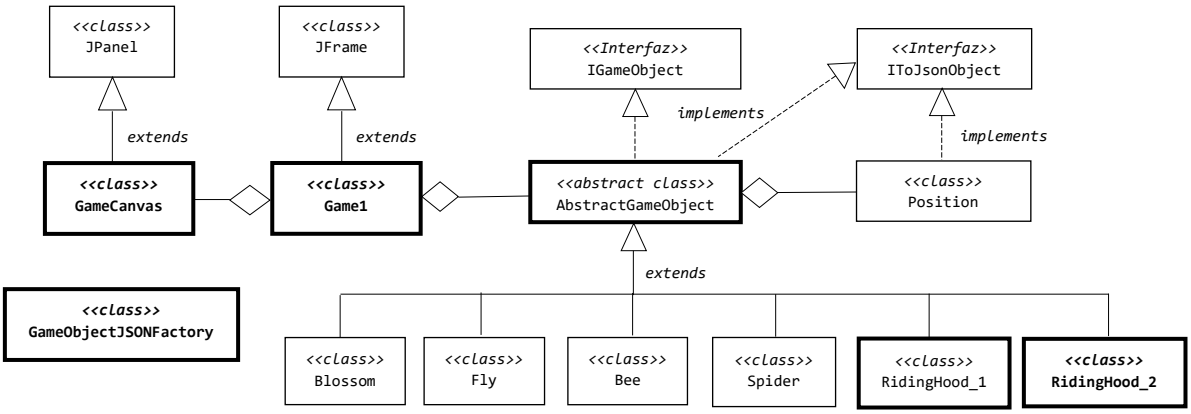


Figura 1: Tipos de datos

- Las clases **`RidingHood_2`** y **`Game_1`** proporcionan una plantilla para la realización del juego y serán brevemente descritas y comentadas en una sección aparte.
- Algunos métodos de **`AbstractGameObject`** se han modificado para hacerlos genéricos y utilizar estructuras dinámicas de datos en lugar de arrays. (Tabla 1).
- Las clases **`RidingHood_1`** ha sido modificada para que use las nuevas versiones de los métodos de `AbstractGameObject`.

Tabla 1: Métodos estáticos de `AbstractGameObject`

Método	**Descripción**
`static double` **`distance`**`(` `  Position p1, Position p2)`	Obtiene la distancia entre dos posiciones.
`static double` **`getDistance`**`(` `  IGameObject jObj1, IGameObject jObj2)`	Obtiene la distancia entre dos elementos del juego.
`static` **`<T extends IGameObject>`** `IGameObject` **`getClosest`**`(` `  Position p,` **`ArrayList<T>`** `jObjs)`	Obtiene el elemento del juego contenido en `jObjs` más cercano a la posición p
`static` **`<T extends IGameObject>`** `IGameObject` **`getClosest`**`(` `  IGameObject jObj,` **`ArrayList<T>`** `jObjs)`	Obtiene el elemento del juego contenido en `jObjs` más cercano al elemento `jObj`.

- La clase **`GameObjectJSONFactory`** proporciona un método para obtener un objeto a partir de su representación JSON. Las factorías permiten eliminar una gran cantidad de código condicional del resto del programa, que de esta forma queda más compacto y fácil de entender. Además independizamos el código del juego frente a la adición de nuevos tipos de `IGameObject`.

- La clase **GameCanvas** proporciona un panel para representar gráficamente el juego que puede usarse en cualquier ventana. `GameCanvas` es una versión ligeramente modificada de la clase interna `Canvas` utilizada en las clases `GameEditor_#`. En la tabla 2 se resumen sus métodos públicos. Hay dos aspectos de su implementación que merecen ser comentados:

  `GameCanvas` utiliza una lista de datos que proporciona un acceso concurrente seguro a sus elementos.

  ```
  ConcurrentLinkedQueue<IGameObject> gObjects = new ConcurrentLinkedQueue<>();
  ```

  En la implementación del juego vamos a almacenar los objetos del juego en una única lista que será manipulada tanto por la lógica del juego, que se ejecuta de acuerdo a los ticks de un reloj, como por el panel en el que se representan las vistas de dichos elementos, que se ejecuta de forma independiente de la lógica del juego. Es decir, tenemos una lista que es accedida "al mismo tiempo" (concurrentemente[1]) por dos procesos diferentes: la lógica del juego y los gráficos del juego. La lógica del juego puede añadir y eliminar elementos en cualquier momento, pudiéndose dar situaciones en las que se intenta representar un elemento del juego que ya no existe. Para evitar estas situaciones se usan listas sincronizadas que mantienen la coherencia de los datos ante accesos concurrentes.

  `GameCanvas` utiliza el método factoría `views.AbstractGameView.getView` para obtener las vistas de los objetos que debe representar.

Tabla 2: Métodos de `GameCanvas`

Constructor	**Descripción**
`GameCanvas()`	Construye un panel cuadrado con una cuadrícula de 20x20 cuadros de 20 pixeles de lado.
`GameCanvas``(int canvasEdge, int squareEdge)`	Construye un panel de `canvasEdge` pixeles de lado y ajusta él número de cuadrados de la cuadrícula de acuerdo con el tamaño del cuadrado, `squareEdge`.
Método	**Descripción**
`void` **`setSquareEdge`**`(int squareEdge)`	Modifica el tamaño del cuadrado de la cuadrícula. Como el tamaño del panel permanece constante varía el número de cuadros de la cuadrícula.
`void` **`drawObjects`**`(ConcurrentLinkedQueue<IGameObject> gObjects)`	Indica al panel la lista de objetos que deben representarse y *solicita* que se representen. Los objetos no se representan de forma inmediata, sino cuando lo *decide* la máquina virtual de Java.
`void` **`refresh`**`()`	Solicita que se represente la lista de objetos del juego que maneja al panel (se usa la lista pasada en una anterior llamada a `drawObjects`).
`void` **`setViewsFamily`**`(IViewFactory viewFactory)`	Fija la factoría de vistas a utilizar para representar los objetos del juego. Por defecto se utiliza una `views.boxes.BoxesFactory`
`void` **`paintComponent`**`(Graphics g)`	Dibuja los gráficos del juego. Aunque es pública nunca se debe llamar directamente, sino a través de `drawObjects` y `refresh`.

[1] Se verá una introducción a la Programación Concurrente en el último bloque de la asignatura.

2.3. El paquete `views`

El paquete `views` ha sido ampliado y reorganizado para facilitar la adición de familias de vistas. Se ha definido un paquete diferente para cada familia de vistas que contiene sus propias vistas para los objetos del juego y una clase factoría para obtenerlos (figura 2). En el paquete raíz `views` se ha definido la interfaz `IViewFactory` que deben implementar todas las factorías de vistas y la clase `AbstractGameView` se ha ampliado con un método factoría que toma como argumento el objeto del juego para el que hay que proporcionar la vista, su tamaño y la factoría de vistas que queremos utilizar.

```
public static IAWTGameView getView(
                IGameObject gObj,
                int length,
                IViewFactory factory) throws Exception
{
        return factory.getView(gObj, length);
}
```

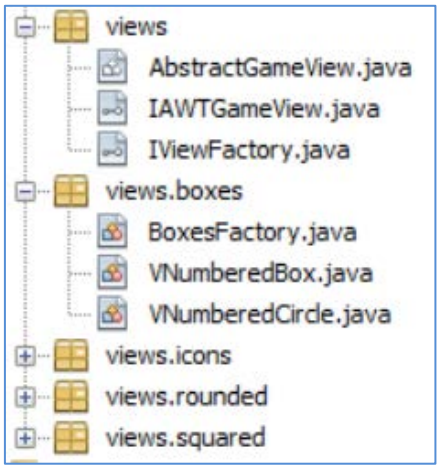


Figura 2: Estructura del paquete `views`.

2.4. Las clases `game.RidingHood_2` y `game.Game_1`

Las clases **`RidingHood_2`** y **`Game_1`** proporcionan una plantilla para la realización del juego. A partir de esta plantilla podemos ir añadiendo nuevas características al juego.

La clase **`RidingHood_2`** está pensada para poder ser controlada desde el teclado, para lo cual proporciona 4 métodos que sirven para fijar su dirección de movimiento: `moveRigth`, `moveLeft`, `MoveUp` and `moveDown`. Estos métodos no cambian la posición del objeto, sino que establecen cómo se cambiará dicha posición cuando se invoque el método `moveToNextPosition`.

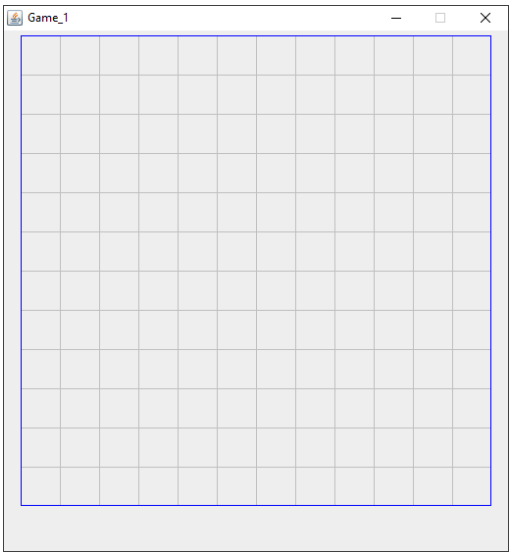
La clase **`Game_1`** es una ventana (`JFrame`) que contiene tanto los elementos del juego como los componentes gráficos en los que se representa. Más adelante se muestra su estructura y se comentan las acciones realizadas en cada método.

El funcionamiento de `Game_1` (figuras 3a a 3d) es el siguiente:

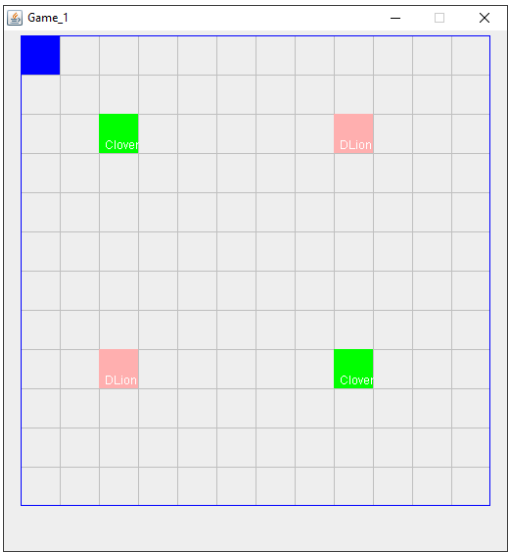
- Al arrancar el programa aparece una cuadrícula vacía (figura 3a).
- Al pulsar una vez la barra espaciadora aparecen los elementos del juego (figura 3b). El personaje principal (cuadrado azul) aparece en la esquina superior izquierda y cuatro

blossoms (cuadrados rosas y verdes) distribuidos por el tablero. También se arranca el *timer*, pero el personaje no se mueve hasta que no se pulsa una flecha.

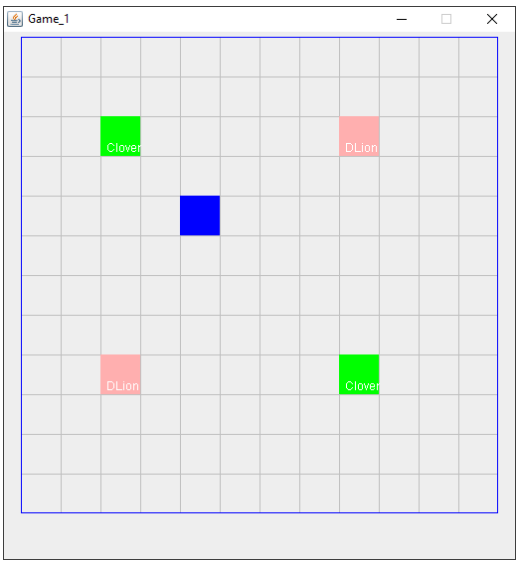
- El rectángulo azul (Caperucita) comienza a moverse cuando pulsamos una tecla de flecha, en la dirección que indica esa flecha. Podemos cambiar la dirección pulsando otra flecha y detener el *timer* pulsando de nuevo la barra espaciadora (figura 3c).
- Si pasamos por encima de un *blossom*, Caperucita deja de verse mientras coinciden la posición del *blossom* y de Caperucita (los *blossoms* se dibujan después que Caperucita), pero no ocurre nada (figura 3d).



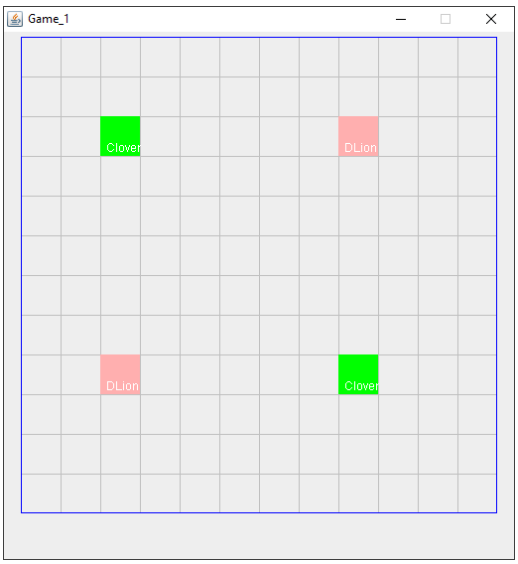
a) Al arrancar Game_1 aparece una cuadrícula vacía.



b) Al pulsar la barra espaciadora, aparecen los elementos del juego y se arranca el *timer*, pero no sucede nada porque el personaje no sabe hacia dónde moverse.



c) Podemos movernos por el tablero pulsando las teclas de fecha.



d) Si pasamos por encima de un *blossom* desaparecemos momentáneamente.

Figura 3: Funcionamiento de Game_1

```
public class Game_1 extends JFrame implements KeyListener, ActionListener {

    // KeyBoard identifiers and last key pressed …
    int lastKey = DOWN_KEY;

    // Game Panel and game info panels
    GameCanvas canvas;
    // …

    // Timer
    Timer timer;
    int tick = 200;

    // Game Variables
    ConcurrentLinkedQueue<IGameObject> gObjs = new ConcurrentLinkedQueue<>();
    RidingHood_2 ridingHood = new RidingHood_2(new Position(0,0), 1, 1);
    int screenCounter = 0;

    // INITIALIZATIONS ------------------------------------------------------------------
    public Game_1() throws Exception{

       super("Game_1");

       // Game Initializations.
       gObjs.add(ridingHood);
       loadNewBoard(0);

       // Window initializations.
       canvas = new GameCanvas(CANVAS_WIDTH, boxSize);
       // …

       addKeyListener(this);
       this.setFocusable(true);
       timer = new Timer(tick, this);
    }

    // KEYBOARD EVENTS ------------------------------------------------------------------
    @Override
    public void keyPressed(KeyEvent ke) {
        lastKey = ke.getKeyCode();
        if (lastKey == SPACE_KEY){
            if (timer.isRunning())timer.stop();
            else timer.start();
        }
    }

    // TIMER EVENTS ---------------------------------------------------------------------
    @Override
    public void actionPerformed(ActionEvent ae) {

        // Actions on Caperucita
        setDirection(lastKey);

        // Moving Caperucita
        ridingHood.moveToNextPosition();

        // Check if Caperucita is in board limits
        setInLimits();

        // Logic to change to a new screen.
        if (processCell() == 1){
            screenCounter++;
            ridingHood.incLifes(1);
            loadNewBoard(screenCounter);
        };
        // Updating graphics and labels
        canvas.drawObjects(gObjs);
    }
```

- Se utiliza una lista para almacenar todos los elementos del juego.
- Se maneja el personaje principal a través de la referencia `ridingHood`.
- Se lleva la cuenta de las pantallas del juego

- Se arranca y para el temporizador con la barra espaciadora.
- Se maneja el personaje principal a través de las teclas de flecha. La última tecla pulsada se almacena en la variable de instancia `lastKey`

Se invoca en cada tick de reloj.

1. Se realizan acciones externas sobre los personajes, en este caso sobre Caperucita (ver método setDirection).
2. Se mueven los personajes (en este caso sólo Caperucita).
3. Se comprueba si los movimientos (en general las acciones realizadas por los personajes) es posible y si no se deshacen (véase método `setInLimits`).
4. Se procesa el tablero (véase `processCell`) y si se dan las condiciones se actualiza el juego con un nuevo tablero (loadNewBoard).
5. Finalmente se actualizan los gráficos del juego.

```
    // ACCIONES SOBRE PERSONAJES ------------------------------------------------------
    // Fija la dirección de caperucita.
    // Caperucita se moverá en esa dirección cuando se invoque su método moveToNextPosition.
    private void setDirection(int lastKey){
        switch (lastKey) {
            case UP_KEY:  ridingHood.moveUp();
                break;
            case DOWN_KEY: ridingHood.moveDown();
                break;
            case RIGTH_KEY: ridingHood.moveRigth();
                break;
            case LEFT_KEY: ridingHood.moveLeft();
                break;
        }
    }

    // COMPROBAR ACCIONES DE LOS PERSONAJES -------------------------------------------
    // Comprueba que Caperucita no se sale del tablero.
    // En caso contrario corrige su posición
    private void setInLimits(){

        int lastBox = (CANVAS_WIDTH/boxSize) - 1;

        if (ridingHood.getPosition().getX() < 0){
            ridingHood.position.x = 0;
        }
        else if ( ridingHood.getPosition().getX() > lastBox ){
            ridingHood.position.x = lastBox;
        }

        if (ridingHood.getPosition().getY() < 0){
            ridingHood.position.y = 0;
        }
        else if (ridingHood.getPosition().getY() > lastBox){
            ridingHood.position.y = lastBox;
        }
    }

    // PROCESAR TABLERO  --------------------------------------------------------------
    // Procesa la celda en la que se encuentra caperucita.
    private int processCell(){
        return 0;
    }


    // CARGAR NUEVO TABLERO  ----------------------------------------------------------
    Carga un nuevo tablero
    */
    private void loadNewBoard(int counter){
        switch(counter){
            case 0:
            default:
              gObjs.add(new Blossom(new Position(2,2), 10, 10));
              gObjs.add(new Blossom(new Position(2,8), 4, 10));
              gObjs.add(new Blossom(new Position(8,8), 10, 10));
              gObjs.add(new Blossom(new Position(8,2), 4, 10));
        }
    }

    public static void main(String [] args) throws Exception{
       Game_1 gui = new Game_1();
    }
}
```

En este caso sólo se define el método que fija la dirección de movimiento de Caperucita.

En este caso sólo se define el método que comprueba que Caperucita está dentro de los límites del tablero y si no está corrige su posición.

En este caso sólo se define el método que procesa la celda en la que se ha colocado Caperucita.
La implementación está vacía: no ocurre nada

Se define el método que carga un nuevo tablero.
En este caso siempre el mismo.

3. Ejercicios a realizar

Ejercicio 1:

La clase `AbstractGameObject` ha sido sustituida por una nueva versión que usa métodos genéricos y listas genéricas. Sustituya en `TestObjects`:

- los arrays por `ArrayList`
- Los bucles `for` por bucles *for each*

De forma que desaparezcan los errores de compilación.

Ejercicio 2:

La clase `RidingHood_1` ha sido sustituida por una nueva versión que usa métodos genéricos y listas genéricas. Se proporciona la antigua versión de `TestRidingHood` Modifíquela sustituyendo donde lo crea conveniente los arrays por `ArrayList` y los bucles `for` por bucles *for each.*

Ejercicio 3:

Implemente una nueva versión de la clase GameEditor_3 (`GameEditor_4`) tal que:

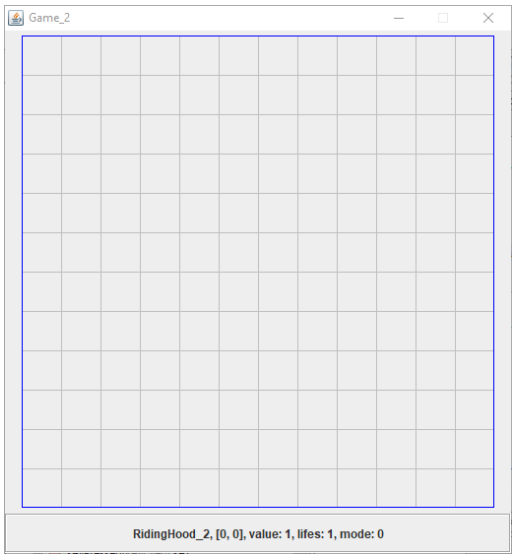
1) No utilice arrays, sino listas `ArrayList`.
2) Utilice la factoría `GameObjectsJSONFactory` para implementar el manejador que carga un fichero de objetos `JSON` (manejador asociado al menú ítem `itLoad`).

Ejercicio 4:

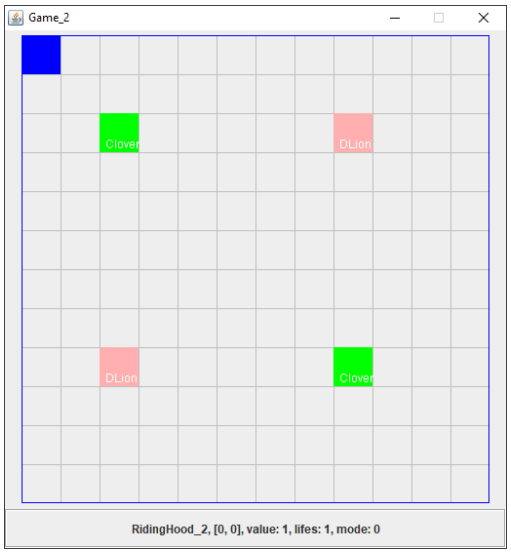
Vamos a implementar ahora una versión mínima del juego, Game_2, utilizando los nuevos elementos suministrados en el paquete `game` (GameCanvas, GameObjectsFactory, GameObjectViewFactory, Game_1). El código de partida será Game_1 (véase sección 2.4).

El funcionamiento y el código de `Game_1` se han explicado en la sección 2.4. Se pide que programe un nuevo juego Game_2 tal que (figura 4):

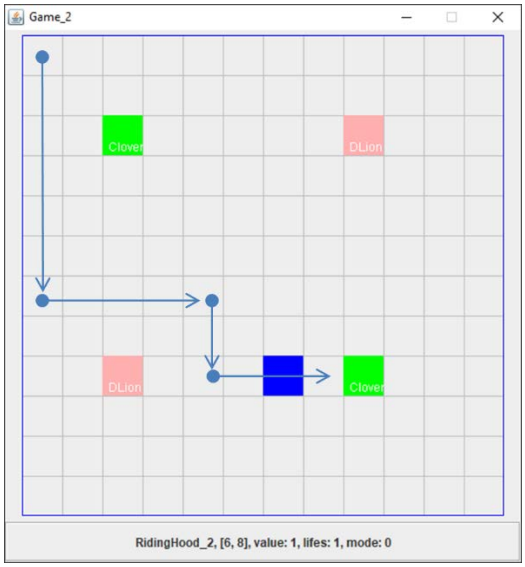
1) Muestre en un panel inferior el estado de Caperucita (puede utilizar `toString` como en la figura 4 o su representación en JSON).
2) Cuando Caperucita pase por encima de un *blossom*, el *blossom* debe desparecer del tablero y su valor sumarse al de Caperucita. Este comportamiento debe implementarlo en el método `processCell`.
3) Cuando no quede ningún *blossom* se cargue un nuevo tablero que haya sido guardado previamente en un fichero usando `GameEditor_4`. Para ello tendrá que ampliar el método `loadNewBoard`. Cada vez que se cargue un nuevo tablero incremente en una unidad la vida de Caperucita.



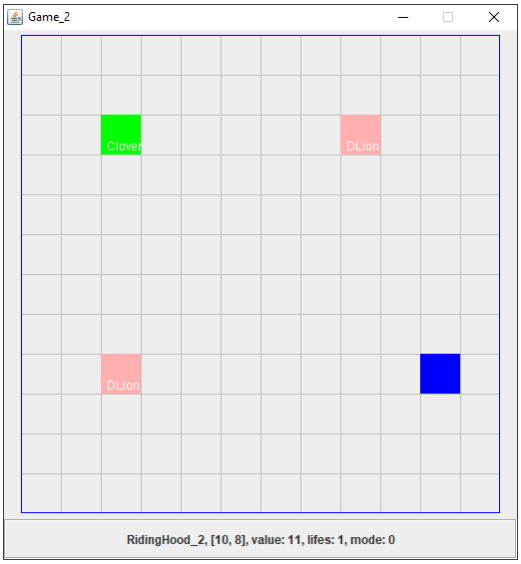
a) Ventana inicial de Game_2, con el tablero de juego en la parte superior y el estado de Caperucita (aun no representada en el gráfico) en la parte inferior.



b) Al pulsar la barra espaciadora se muestran los contenidos del tablero como en Game_1



c) Al pulsar las flechas nos movemos por el tablero (como en Game_1)



d) Pero a diferencia de Game_1, cuando pasamos por encima de un *Blossom* nos lo comemos, incrementando nuestro valor en el valor del *blossom*.

Ejercicio 5:

Vamos a implementar ahora otra versión mínima del juego, Game_3, partiendo de nuevo de Game_1 (véase sección 2.4).

El funcionamiento y el código de `Game_1` se han explicado en la sección 2.4. Se pide que programe un nuevo juego Game_3 tal que Caperucita no se mueva por teclado, sino que decida ella misma cómo moverse en función de la posición de los blossoms. Para realizar este ejercicio tendrá que:

1) Proporcionar una nueva implementación de Caperucita (clase RidingHood_3) tal que:
   - Proporciona un constructor que toma como argumento una referencia a los objetos del juego:

     ```
     public RidingHood_3(
            Position position,
            int value,
            int life,
            ConcurrentLinkedQueue<IGameObject> gObjs)
     ```

   - Proporciona una implementación de `moveToNextPosition` que se mueve hacia el *blossom* más cercano. Para programar este método se sugiere que (1) implemente un método que devuelva los *blossoms* contenidos en los objetos del juego, (2) obtenga aquel que está más próximo a Caperucita (mediante método `AbstractGameObject.getClosest`) y (3) se aproxime a él (como hace el método `approachTo` de `RidingHood_1`);
2) Cuando Caperucita pase por encima de un *blossom*, el *blossom* debe desparecer del tablero y su valor sumarse al de Caperucita. Este comportamiento debe implementarlo en el método `processCell` (exactamente igual que en el ejercicio anterior).
3) Cuando no quede ningún *blossom* genere un nuevo tablero con 4 blossoms en posiciones aleatorias. Cada vez que se cargue un nuevo tablero incremente en una unidad la vida de Caperucita.