Universidad Politécnica de Cartagena

Programación para Ingeniería Telemática.

PRÁCTICA 2. Parte 2:

Interfaz Gráfica de Usuario. Manejo básico de eventos.

Contenido

Objetivos de aprendizaje

(1) Uso extensivo de librerías gráficas de Java.

(2) Manejadores de eventos asociados a los elementos de la interfaz de usuario. Programación dirigida por eventos e inversión de control.

Recursos proporcionados

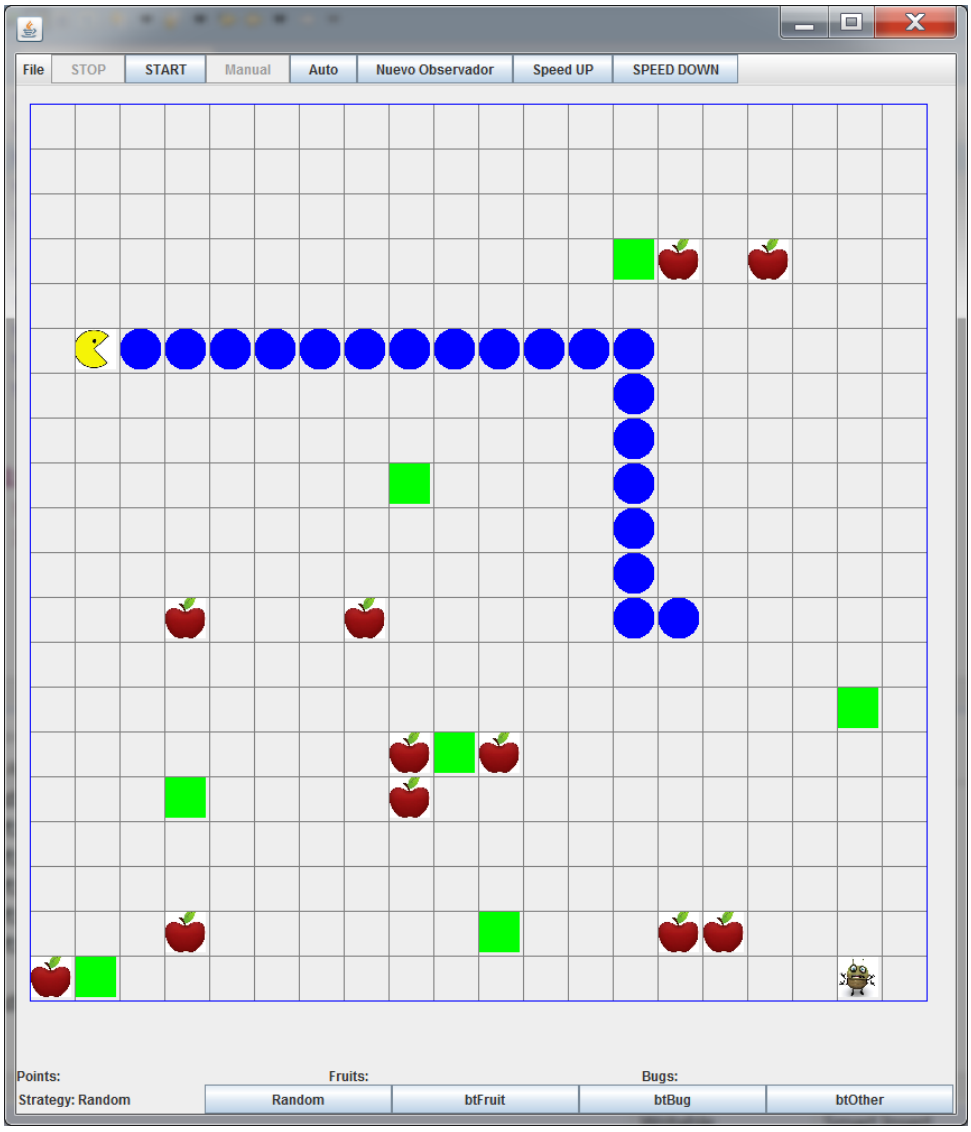
En el aula virtual podéis encontrar el fichero comprimido git_2014_15_c.zip que contiene un proyecto Java de Eclipse con el siguiente contenido:

- Paquete `p2.basic`: Tipos de datos básicos. Se han añadido nuevos tipos de datos
  - `IAutonomousObject` que modela un objeto de juego con comportamiento autónomo.
  - `IGameObservable` e `IGameObserver` que modela una relación entre el juego y sus observadores.
- Paquete `p2.model_impl`: implementaciones básicas de los tipos definidos en `p2.basic`. A veces la implementación es parcial y se pide a los alumnos que la completen. Otras veces, los alumnos tendrán que implementar los tipos completamente. Se añaden varias implementaciones:
  - `BugAutonomous0` y `nakeSAutonomous0` que proporcionan una plantilla para implementar bichos y serpientes autónomas.
  - `Game0` que proporcionan una plantilla para la implementación del juego. Algunos de sus métodos se dan completamente implementados.
- Paquete `p2.view_impl`: implementaciones básicas de las vistas de los objetos del juego. Se añade una nueva implementación:
  - `GamePanel` que proporciona un panel donde representar el juego.

Ejercicios a realizar.

Contenidos a realizar	Individual		Pareja	
	Req	Valor	Req	Valor
1. Persistencia				
a) Salvar a fichero/Cargar desde fichero (se recomienda uso de JFileChooser).	**x**	**1.0**	**x**	**0.5**
b) Salvar a DB/Cargar desde DB. Implica también diseñar la BD.		**2.0**		**1.5**
2. Modo Manual Sencillo.	**x**	**4.0**	**x**	**3.0**
a) Sólo la cabeza de la serpiente sin eslabones. b) Sólo obstáculos y fruta. c) Se detectan límites de tablero y obstáculos. d) Indicadores de puntos y frutas. e) Independencia de vistas y modelos.				
3. Modo Manual Extendido (asume que se ha realizado completo el modo manual sencillo)		**2.0**		
a) Serpiente con eslabones que aparecen con el paso del tiempo y desparecen en función de la fruta que se come la serpiente. b) Aparecen bichos que se mueven por el tablero. Si la serpiente se los come se le suman puntos y se le quitan eslabones, si es el bicho quien come a la serpiente se acaba el juego. c) Indicador de bichos comidos. d) Control de velocidad de juego: desde la gui y con el paso del tiempo la velocidad se incrementa.				
4. Modo Automático (I). Asume modo manual sencillo.		**2.0**	**x**	**1.5**
a) La serpiente se mueve sola. b) Botón de selección de modo manual a automático. c) Botón de parada y reanudación del juego.				
5. Modo Automático Extendido.		**1..4**		**1..3**
a) Se asumen elementos de modo manual extendido. b) Al menos dos formas de movimiento que puedan seleccionarse desde la pantalla (patrón estrategia).				
6. Ventanas observadoras del juego.		**1.0**		**0.75**
a) Se pueden arrancar ventanas en las que se visualiza simultáneamente el desarrollo del juego (patrón observador). b) Botón de arranque de nueva pantalla. c) La pantalla observadora puede cerrarse correctamente sin que se produzcan errores.				
7. Calidad de la GUI (Se asume una calidad mínima como la de los ejemplos)		**3.0**		**2.0**
a) Mejor disposición de los elementos en la ventana: b) Uso de diferentes fondos y tipos de letra. c) Variedad de las vistas de los elementos del juego y optimización del uso de la memoria. d) Implementación de vistas geométricas (no basadas en imágenes cargadas de un archivo). e) Uso de animaciones. f) Efectos de sonido. g) Ventanas emergentes al terminar el juego (p.e.: para indicar que el juego ha terminado o para introducir el nombre del jugador). h) Se admiten otras que se le ocurran al alumno.				

Ejemplo de GUI para el modo automático extendido.



Descripción de las clases e interfaces.

Este apartado completa las descripciones hechas en los dos anteriores boletines correspondientes a la práctica 2.

Constantes del juego. Tipo de datos `p2.basic.IGameConstants`.

En esta clase se definen las constantes que representan las teclas de flecha, la dirección de movimiento de los objetos del juego y los caracteres que se utilizan para codificar cada uno de los tipos de los elementos del juego. Por ejemplo:

```
// Keys for finding the views.
    public static final char Free          = ' ';
    public static final char Obstacle      = 'o';

// …

// Definition of keys
    public static final int UpKey =   38;

// …

// Definition of directions.
    public static final int Up    = 1;
```

Representación gráfica del juego. Clase `p2.basic.GamePanel`.

La clase `GamePanel`, que se proporciona completamente implementada, es un panel en el que se dibujan los elementos del juego.

Mantiene un diccionario de vistas (`tViews`).

```
// Dictionary that associate keys with views.
private Hashtable<Character, IView> tViews = new Hashtable<>();
```

Las claves de acceso son los códigos definidos en `IGameConstants`. Por ejemplo:

```
// Se crea una vista a partir de un fichero
vGrape = new VImage("food", "resources/apple2.jpg");


// Se guarda la vista en el diccionario de vistas.
tViews.put(IGameConstants.Fruit, vGrape);


// Para acceder a esta vista.
IView vi = tViews.get(IGameConstants.Fruit);
```

El método de actualización del panel es

```
/**
 * Updates the game grid and repaint the component accordingly.
 * It is invoked by the game when the game state changes.
 * @param board representation of the game objects in the board.
 */
public void updateGame(char [][] board)
```

Donde el argumento es una matriz de caracteres que indica que tipo de objeto hay en cada posición. Cada posición de `board` contiene un valor definido en `IGameConstants`. Esta es toda la información que necesita el panel para representar los elementos del juego en las posiciones que ocupan.

Objetos del juego autónomos. Tipo de datos `IAutonomousObject` y sus implementaciones (`BugAutonomous0` y `SnakeAutonomous0`).

Los objetos que se pueden mover de forma autónoma deben implementar la interfaz `IAutnomousObject` que define los dos métodos que se describen más abajo. Uno determina la dirección en la que quiere moverse el objeto y el otro la posición que quiere ocupar en el siguiente tick del juego. En realidad bastaría con uno de ellos, ya que el juego puede gestionar los movimientos solicitados por los objetos tanto sabiendo en qué dirección quieren moverse como sabiendo la posición que quieren ocupar en el próximo tick. Según se elija una u otra se simplifica la implementación del juego o de los objetos autónomos. En el esqueleto del juego que se proporciona se asume el uso de `getNextCoordinate.`

```
/**
 * Determine the direction in which the object want to be moved.
 * It does not cause the motion of the object, because
 * only the game can change the object position
 * @param board representation of the game objects in the game board.
 * @return direction in which the object want to be moved.
 */
public int getDirection(char[][] board);

/**
 * Determine the position in which the object want to be located.
 * It does not cause the motion of the object, because
 * only the game can change the object position
 * @param board representation of the game objects in the game board.
 * @return position in which the object want to be located.
 */
public Coordinate getNextCoordinate(char[][] board);
```

Las dos clases que implementan estas interfaces son `BugAutonomous0` y `SnakeAutonomous0`, que deberán ser implementadas por los alumnos, `BugAutonomous0` en cualquier caso (salvo modo manual sencillo) y `SnakeAutonomous0` en los modos automáticos.

El juego de partida. Clase Game0.

Se proporciona un esqueleto del juego del que el alumno puede, si lo estima oportuno partir, para realizar la práctica. En el cuadro de abajo se presenta un resumen con los elementos más relevantes del código. Observe que:

1. Se ofrecen las estructuras de datos del juego. Pueden cambiarlas si lo estiman oportuno.
2. Algunos de los métodos se dan completamente implementados.
3. La dinámica del juego queda determinada en el manejador del Timer.
   Debe entender la secuencia de acciones que se llevan a cabo en este método. El alumno puede cambiar si quiere la implementación de este manejador, pero entonces es posible que deba definir métodos nuevos para gestionar y actualizar el movimiento de los objetos de juego.
4. En el cuadro de abajo no se incluye todo el código de Game0. Debe inspeccionar el código proporcionado.

```
public class Game_0 extends JFrame implements IGame, KeyListener, ActionListener {

    // Board size ..............................................
    int nRows = 20, nCols = 20, edge = 40;

    // Timer ...................................................
    int ticks;
    Timer timer;
    int tick = 200;

    // Modes of operation ......................................
    // Auto mode active: game calls snake algorithm to obtain next coordinate.
    // Auto mode inactive: snake is commanded with the arrow keys.
    boolean autoModeActive;
    boolean isStopped;

    // Next coordinate for snake. Update in key listener in manual mode.
    int currentDirection = IGameConstants.Right;

    // Game objects.....................................................
    ConcurrentLinkedQueue<IGameObject> objects = new ConcurrentLinkedQueue<>();
    SnakeAutonomous aSnake;

    // Declaration of GUI Elements.....................................................

    // …


    public Game_0(int rows, int cols) throws IOException {

        nRows = rows; nCols = cols;

        // … Creation and addition of gui elements …

        addKeyListener(this);
        this.setFocusable(true);

        // Fijamos tamaño de la ventana.
        setSize (60 + edge * nRows, 200 + edge * nCols);

        // …. Otras inicializaciones …

        // Creación de objetos del juego
        fillRandom(1);
        aSnake = new SnakeAutonomous("link", "link", 0, new Coordinate(0,0));
        objects.add(aSnake);

        // Create timer.
         timer = new Timer(tick, this);
    }

    // BUILD MENUS.
    private void buildMenuBar(){ … }

    // BUILD INFO PANEL
    private JPanel buildInfoPanel(){ …    }


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    // GAME LOGIC.

    /**
     * Obtiene la coordenada en la que estaría el objeto si se moviera en una dirección determinada.
     * NO provoca que el objeto cambie su posición
     * @param obj objeto del juego
     * @param direction dirección de movimiento
     * @return coordenada en la que estaría el objeto si se moviera en la dirección determinada
     */
    public Coordinate getNextCoordinate(IGameObject obj, int direction){

         // Se proporciona completamente implementado …

    }
```

```java
/**
* Test if the argument can be moved to the target position: test board limits, obstacles, etc.
* NO provoca que el objeto cambie su posición.
* @param obj object to move
* @param target position where the object is to be moved.
* @return true if the motion is possible.
*/
public boolean testObjectPosition(IGameObject obj, Coordinate target){
        return true;
}

/**
 * MOVE obj to the target position.
 * PROVOCA QUE EL OBJETO CAMBIE SU POSICIÓN.
 * Utiliza testObjectPosition para comprobar si puede realizar el movimiento.
 * @param obj game object
 * @param target postition where obj is to be placed
 * @return true if the motion has been done.
 */
public boolean updateObjectPosition(IGameObject obj, Coordinate target) {
        return true;
}

/**
 * Process object position and update the game consequently.
 * Por ejemplo, si obj es la serpiente y está en una posición en la que hay una fruta,
 * la serpiente se come la fruta.
 *
 * @param obj object to be processed.
 * @return true if game can continue.
 */
public boolean processObjectPosition(IGameObject obj) {
        return true;
}

/**
 * Generate obstacles and fruits randomly and add them to the instance variable objects that
 * stores all game objects.
 * As greater the level as much the obstacles
 * @param level level of difficulty
 */
public void fillRandom(int level){

        // Se proporciona completamente implementado …
}

/**
 * Gets all the game objects located at a given coordinate.
 * @param coord coordinate
 * @return game objects located at the coordinate
 */
public ArrayList<IGameObject> getObjectsAt(Coordinate coord) {

        // Se proporciona completamente implementado …
}


/**
 * Returns a char representation of the board.
 * @return char representation of the board.
 */
public char [][] toCharMatrix(){
        // Se proporciona completamente implementado …
}

// Actions to be taken when the snake eats a fruit.
private void processFruit(Fruit f){

}

// Actions to be taken when the snake eats a bug.
private void processBug(Bug b){
}

// Actions to be taken to add a “bicho” to the game.
private BugAutonomous spawnBug(){   }
```

```
    /*****************************************************************************************
     * MANEJADORES DE ENTRADAS DE TECLADO: cambio de dirección en modo manual.
     */
        @Override
        public void keyPressed(KeyEvent arg0) {       }

        @Override
        public void keyReleased(KeyEvent arg0) {
                if (!autoModeActive) {
                        int keyCode = arg0.getKeyCode();
                        switch(keyCode){
                        case IGameConstants.UpKey: currentDirection = IGameConstants.Up; break;
                        case IGameConstants.DownKey:currentDirection = IGameConstants.Down; break;
                        case IGameConstants.LeftKey:currentDirection = IGameConstants.Left; break;
                        case IGameConstants.RightKey:currentDirection = IGameConstants.Right; break
                        default:break;
                        }
                }
        }

        @Override
        public void keyTyped(KeyEvent arg0) {}

    /*****************************************************************************************
     * MANEJADOR DE TEMPORIZADOR: avance del juego.
     */

        public void actionPerformed(ActionEvent arg0) {

                // Recuperar foco. Necesario para que lleguen eventos de teclado.
                requestFocus();

                // Añadir link cada 10 ticks de reloj.
                if (ticks%10 == 0){
                        SnakeLink lnk = aSnake.addLink();
                        objects.add(lnk);
                }

                // Añadir bug cada 50 ticks de reloj.
                if (ticks%50 == 0){
                        objects.add(spawnBug());
                }

                // Incrementar velocidad y reiniciar cuenta de
                // ticks cada 100 ticks.
                if (ticks%100 == 0){
                        tick -= (tick > 50)?50:0;
                        timer.setDelay(tick);
                        ticks = 0;
                }

                // Incrementar ticks.
                ticks++;

                // Update bugs
                ArrayList<BugAutonomous> bugs = getBug();
                for (BugAutonomous bug : bugs){
                        Coordinate next = bug.getNextCoordinate(toCharMatrix());
                        updateObjectPosition(bug, next);
                        processObjectPosition(bug);
                }
                // Update snake
                Coordinate nextSnakeCoord = null;
                if (autoModeActive) {
                        nextSnakeCoord = aSnake.getNextCoordinate(toCharMatrix());
                }
                else{
                        nextSnakeCoord = getNextCoordinate(aSnake, currentDirection);
                }
                updateObjectPosition(aSnake, nextSnakeCoord);
                processObjectPosition(aSnake);

                // Actualizar gráficos del juego.
                panelJuego.updateGame(toCharMatrix());
        }
}
```