Universidad Politécnica de Cartagena

Programación para Ingeniería Telemática

PRÁCTICA 2. Parte I:

Diseño OO, serialización, E/S y conectividad con base de datos.

Contenido

Objetivos de aprendizaje

(1) Repasar conceptos básicos de la POO: interfaces, clases, clases abstractas, herencia, composición, etc.
(2) Presentar el diseño básico de la aplicación que tendrán que desarrollar los alumnos en esta y en sucesivas prácticas (5 o 6 sesiones de prácticas):
(3) Ser capaces de serializar objetos y guardarlos en ficheros de texto y en una base de datos.
(4) Ser capaces de cargar datos en la aplicación desde ficheros o bases de datos.

Recursos proporcionados

En el aula virtual podéis encontrar el fichero comprimido git_2014_15.zip que contiene un proyecto Java de Eclipse con el siguiente contenido:

- Paquete `p1:` ejemplos de conexión a una base de datos.
- Paquete `p2.basic`: Tipos de datos básicos.
- 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.
- Librerías:
  - `mysql-connector-java-5.1.13-bin.jar`: conexión a base de datos.
  - `org.json-20120521.jar` : codificación y decodificación de objetos en JSON

También el aula virtual podréis encontrar el fichero `er_gusano.sql`, Implementación parcial de una base de datos para guardar los datos de la aplicación.

Ejercicios a realizar.

1. Completar la implementación de las clases `Poison`, `Bug` y `Fruit`.
2. Implementar la clase `SimpleGame`, excepto el método `saveToDB`.
3. Implementar el método `saveToDB` de `SimpleGame`.

Antes de realizar los ejercicios lea cuidadosamente todo el boletín de prácticas.

Descripción general de la aplicación.

La aplicación consiste en un juego cuyo desarrollo es un híbrido entre el gusano (*snake*) y el comecocos. Los detalles de funcionamiento se explicarán en boletines de prácticas posteriores, pero en esencia, y para lo que hace falta por ahora, puede resumirse así (figura 1):

1. El jugador mueve una serpiente por un tablero de juego mediante las flechas del teclado. El tablero es una cuadrícula y cada elemento de juego ocupa exactamente un cuadro.
2. La serpiente se compone de eslabones, inicialmente uno solo, la cabeza.
3. Cuando la serpiente choca contra los bordes del tablero, contra un obstáculo o contra sí misma muere y el juego acaba.
4. A medida que pasa el tiempo la velocidad del juego aumenta y crece la longitud de la serpiente (se la añaden eslabones) con lo que es más fácil que choque consigo misma.
5. Además de obstáculos, el tablero contiene comida y veneno. Si la serpiente pasa por una comida se suma su valor y se le quitan tantos eslabones como vale la comida, si pasa por un veneno se le restan tantos puntos y se le añaden tantos eslabones como valga el veneno.
6. De vez en cuando salen unos bichos que se mueven por el tablero. Si alguno de ellos toca a la serpiente, excepto su cabeza, el juego acaba.
7. Si la serpiente se come a un bicho se le quitan todos sus eslabones, excepto la cabeza, y se le suman tantos puntos como valga el bicho.

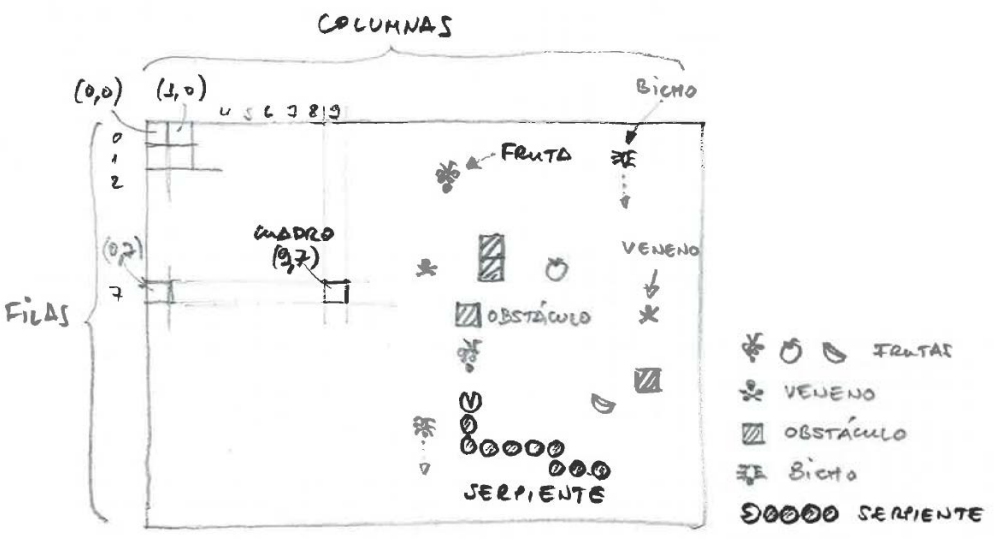

Figura 1: Esquema del juego.

El juego se va actualizando con un tic de reloj cuyo periodo puede ajustarse. Además, el juego tiene dos modos de funcionamiento:

- Manual: el jugador maneja su personaje desde el teclado (con las teclas de flecha).
- Automático: el jugador define un método de movimiento en la serpiente que es llamado por el juego en cada tic de reloj. Este modo asume que desde tal método puede accederse el estado del tablero (posiciones de vidas, comidas y bichos). A partir de esta información la serpiente le indica al juego la dirección en la que debe moverla en el siguiente tic de reloj. En cada tic de reloj cada personaje (serpiente o bicho) solo es desplazado una fila o una columna.

Modelado de los elementos de la aplicación: interfaces e implementaciones básicas.

Todas los tipos de datos están definidos en el paquete `p2.basic`. El paquete `p2.model_impl` incluye una serie de implementaciones básicas, que los alumnos deberán completar como se indicará en esta práctica y en otras sucesivas. **La información detallada de los métodos está definida en comentarios sobre el código**. No se explican todos los tipos de datos, sólo los necesarios para hacer la parte I de la práctica.

Todos los tipos de datos modelados por interfaces comienzan por **`I`** mayúscula, los modelados por clases abstractas comienzan con **`A`** mayúscula. Los que no empiezan por **`I`** ni por **`A`** se modelan mediante clases.

Modelado de tipos serializables utilizando JSON. Tipo de datos `IJSONizable`.

Serialización:

La serialización de un objeto es un proceso de transformación mediante el cual la información que almacena el objeto se especifica en un formato que facilita tanto su almacenamiento en un fichero o en una base de datos como su transmisión a través de un enlace de comunicaciones. El proceso es reversible, es decir, es posible reconstruir el objeto (deserializarlo) a partir de la información generada en el proceso de serialización. Los dos formatos más utilizados para serializar los objetos son XML y JSON (*JavaScript Object Notation*). JSON es notablemente más sencillo que XML y por ello ha sido elegido para la realización de la práctica.

> Pueden encontrar toda la información que necesitan sobre JSON (y bastante más) a partir de http://www.w3schools.com/json/json_syntax.asp

El formato JSON es sintácticamente idéntico al código que se utiliza para crear objetos en JavaScript. Con JSON los objetos se convierten en texto plano (legible para un ser humano), manteniendo la estructura jerárquica de contención de los objetos. Los datos JSON consisten en pares nombre/valor:

```
"firstName":"John"
```

El nombre va entre comillas y el valor puede ser un número, un string, un booleano, un array u otro objeto. Los objetos JSON se escriben entre llaves que contienen múltiples pares nombre/valor separados por comas.

```
{"firstName":"John", "lastName":"Doe"}
```

Los arrays JSON se escriben entre corchetes y pueden contener valores u objetos, como en el ejemplo siguiente:

```
"employees":[ {"firstName":"John", "lastName":"Doe"},
              {"firstName":"Anna", "lastName":"Smith"},
              {"firstName":"Peter", "lastName":"Jones"} ]
```

La interfaz `IJSONizable`:

Todos los objetos del juego que pueden ser guardados en un fichero implementan esta interfaz que define una etiqueta y dos métodos. En todos los datos serializados deberá existir el par nombre/valor:

“obj_type” = class_name

El nombre `obj_type` está definido en la constante `TypeLabel`. El valor *`class_name`* es el nombre de la clase a la que pertenece el objeto serializado. Este campo será utilizado para saber (o comprobar) la clase del objeto a reconstruir (deserializar). Los dos métodos que define la interfaz son:

- `JSONObject toJSONObject()`:Devuelve una representación del objeto en formato JSON.
- `String serialize()`:Devuelve el objeto JSON en formato texto

`JSONObject` es un tipo definido en la librería `org.json` mediante el cual vamos a poder crear y manipular los objetos JSON. La representación textual del objeto JSON la obtenemos mediante el método `serialize`. En las clases que se ofrecen ya implementadas hay abundantes ejemplos del uso de esta librería para crear y manipular objetos JSON.

Hay bastantes librerías que permiten serializar y desrealizar objetos JSON. `org.json` se ha elegido por su sencillez.

> Pueden encontrar toda la información que necesitan sobre `org.json` (y bastante más) a partir de http://www.json.org/java/

Modelado las posiciones de los elementos del juego en el tablero. Tipo de datos `Coordinate`.

La clase `Coordinate`, que implementa `IJSONizable`, modela la posición de los cuadros de la cuadrícula del tablero, indicando la columna y la fila en la que se encuentra dicho cuadro (figura 2). La definición de este tipo de datos podría haberse evitado dada la simplicidad del juego, pero nos proporciona ciertas ventajas de cara al aprendizaje de la programación OO y es un punto de partida sencillo para explicar la serialización de objetos utilizando JSON.

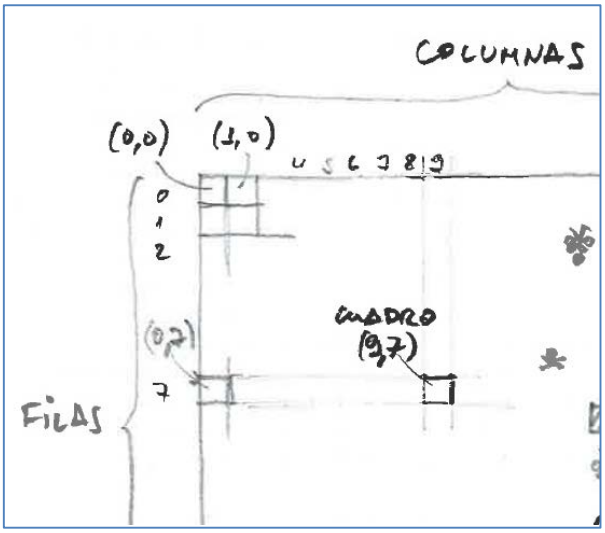

Figura 2: Cuadrícula del juego.

Las coordenadas pueden consultarse (métodos `getColumn` y `getRow`), y modificarse (`setColumn` y `SetRow`), ya que (algunos) elementos del juego pueden cambiar de ubicación.

Merece la pena echar un vistazo al código del método `toJSONObject` (figura 3) y al del constructor de una coordenada a partir de un objeto JSON (figura 4). Este constructor, que debe estar presente en todas las clases que implementen `JSONizable`, construye una coordenada a partir de una representación del objeto en formato JSON.

Obsérvese (figura 3) como se crea un objeto JSON vacío y después se le van añadiendo campos mediante los métodos `put`. `put` permite añadir cualquier cosa (un número, cadena o valor booleano, un *string*, otro objeto JSON o un *array*). Los métodos `put` están convenientemente sobrecargados, de forma que no tenemos que preocuparnos por el tipo de los datos que estamos serializando. En ejemplos sucesivos veremos casos más complicados.

Obsérvese también (figura 4) cómo recuperamos un objeto `Coordinate` a través de los métodos `get`*X*, donde la *X* representa un tipo de datos. En este caso debemos saber a priori el tipo de los datos asociado a una etiqueta (nombre) determinada para llamar al método `get` conveniente. En el ejemplo se llama a `getString` para obtener el nombre de la clase del objeto y a `getInt` para obtener la fila y la columna.

La clase `Coordinate` incluye un método `main` donde puede verse el resultado de serializar y deserializar coordenadas.

```
public JSONObject toJSONObject() {
        // Se crea objeto JSON vacío.
        JSONObject jObj = new JSONObject();

        try {
        // Se le van añadiendo los campos del objeto.
        // Primero: una indicación del tipo de datos.
                jObj.put(IJSONizable.TypeLabel, Coordinate.TypeName);

        // Después el resto de los datos.
                jObj.put(Coordinate.ColumnLabel, this.column);
                jObj.put(Coordinate.RowLabel, this.row);
        } catch (JSONException e) {
                e.printStackTrace();
        }
        return jObj;
}
```

Figura 3: Serialización de una coordenada: Implementación de `toJSONObject` en la clase `Coordinate`.

```
/**
 * Crea un objeto Coordinate a partir de su representación JSON
 * @param jsonObj representación del objeto serializado.
 * @throws JSONException Se lanza si el objeto no representa a una coordenada.
 * @throws ParamException Se lanza si alguno de los campos del JSONObject no se corresponde
 * con los que se experan en una coordenada o alguno de los campos está ausente.
 */
public Coordinate(JSONObject jsonObj) throws JSONException, ParamException {
    try{
        if (!jsonObj.getString(IJSONizable.TypeLabel).equals(TypeName)){
            throw new ParamException("Constructor.jSONObj is malformed");
        }
        this.row     = jsonObj.getInt(Coordinate.RowLabel);
        this.column  = jsonObj.getInt(Coordinate.ColumnLabel);
    }
    catch(JSONException je){
        // Mensajes a consola …
        throw je;
    }
}
```

Figura 4: Construcción de un objeto `Coordinate` a partir del `JSONObject` que lo representa.

Modelado de los elementos del juego. Tipo de datos `IGameObject`.

La interfaz `IGameObject`, que extiende `IJSONizable`, modela los elementos del juego. Dispone de métodos para (1) obtener su identificador (`getId`) y su descripción (`getDescripcion`); (2) obtener y fijar su valor (`getValue` y `setValue`); y (3) obtener y establecer su posición en el tablero (`getCoordinate` y `setCoordinate`).

Todos los elementos del juego tienen asociados un identificador único y una descripción, que se fijan cuando se crea el elemento y no pueden ser cambiados (**no** hay métodos `setId` o `setDescription`). Se pueden cambiar el valor y la posición del elemento, si bien se pueden fijar límites al movimiento de los elementos por el tablero. Si se sobrepasan estos límites el método `setCoordinate` lanza excepción.

Obsérvese que todos los elementos del juego pueden ser serializados (`IGameObject` extiende `IJONSonizable`).

Modelado del personaje del juego. Tipo de datos `IGameCharacter`.

La interfaz `IGameCharacter`, que extiende `IGameObject`, modela al personaje del juego. `IGameCaracter` no añade ningún método. En su estado actual, sirve para marcar a un determinado elemento del juego como al personaje del juego.

Modelado del juego. Tipo de datos `IGame`.

La interfaz `IGame`, que extiende `IJSONizable`, modela una partida del juego. Dispone de métodos para (1) obtener el identificador de la partida (`getId`) del dueño de la partida (`getOwnerId()`); (2) obtener el personaje y resto de elementos de la partida (`getCharacter` y `getItems`); (3) obtener el elemento que hay en un lugar determinado del tablero (`getItem`); (4) obtener el tamaño en filas y columnas del tablero (`getNumberOfRows`, `getNumberOfColumns`); (5) imprimir el tablero en consola (`printBoard`); y (6) salvar la partida (`saveToDB`, `saveToFile`).

Obsérvese que aunque se habla de tablero, no hay una clase específica para modelar dicho tablero. El estado del tablero se conoce a través de las posiciones en las que se encuentran los elementos del juego y no hace falta más.

Obsérvese que el juego pueden ser serializado (`IGameObject` extiende `IJONSonizable`) y guardado en un fichero en formato texto. También se considera que puede ser guardado en una base de datos. Se proporciona más adelante un ejemplo (clase `TestDB`).

Implementación de los métodos comunes a todos los elementos del juego: clase abstracta `AGameObject`.

La clase abstracta `AGameObject` proporciona una implementación básica de todos los métodos de `IGameObject` con la excepción de `setCoordinate`. La mayor parte de estas implementaciones son triviales (variables de instancia y métodos de acceso). Como en el caso de `Coordinate` vamos a fijarnos en el método de serialización y en el constructor de deserialización (figuras 5 y 6).

La novedad respecto de `Coordinate` es que entre las variables de instancia de `AGameObject` se incluye un objeto, precisamente una instancia de `Coordinate`. Obsérvese cómo se serializa (figura 5), incluyéndolo en el objeto JSON mediante la sentencia:

```
jObj.put(AGameObject.PositionLabel, this.pos.toJSONObject());
```

Utilizamos un método `put` de `JSONObject` que toma como argumento la versión serializada de la coordenada, que obtenemos mediante `this.pos.toJSONObject()`.

Obsérvese cómo se recupera esta información en el constructor (figura 6) a través de la sentencia:

```
this.pos =
    new Coordinate(jsonObj.getJSONObject(AGameObject.PositionLabel));
```

En esta sentencia llamamos al constructor de `Coordinate` que toma como argumento un `JSONObject`, que a su vez obtenemos mediante una llamada a `getJSONObject` sobre el objeto `JSONObject` que toma como argumento el constructor. Recuérdese que un objeto JSON puede contener a otros objetos JSON (estructura jerárquica).

```
public JSONObject toJSONObject() {
        // Se crea objeto JSON vacío.
        JSONObject jObj = new JSONObject();

        try {
        // Se le van añadiendo los campos del objeto.
        // Primero: una indicación del tipo de datos.
                jObj.put(IJSONizable.TypeLabel, Coordinate.TypeName);

        // Después el resto de los datos.
                jObj.put(AGameObject.IdLabel, this.id);
                jObj.put(AGameObject.NameLabel, this.name);
                jObj.put(AGameObject.ValueLabel, this.value);
                jObj.put(AGameObject.PositionLabel, this.pos.toJSONObject());
        } catch (JSONException e) {
                e.printStackTrace();
        }
        return jObj;
}
```

Figura 5: Serialización de una coordenada: Implementación de `toJSONObject` en la clase `AGameObject`.

```
public AGameObject(JSONObject jsonObj) throws JSONException, ParamException {
        try{
            if (!jsonObj.getString(IJSONizable.TypeLabel).equals(this.getClass().getName())){
                System.out.println(this.getClass().getName() +
                                   ".Constructor: ParamException. jSONObj is malformed");
                throw new ParamException(this.getClass().getName() +
                                   ".Constructor.jSONObj is malformed");
            }
            this.value  = jsonObj.getInt(AGameObject.ValueLabel);
            this.id     = jsonObj.getString(AGameObject.IdLabel);
            this.name   = jsonObj.getString(AGameObject.NameLabel);
            this.pos    = new Coordinate(jsonObj.getJSONObject(AGameObject.PositionLabel));
        }
        catch(JSONException je){
        // Mensajes a consola …
            throw je;
        }
}
```

Figura 5: Construcción de un objeto `AGameObject` a partir de un `JSONObject` .

Implementación de los elementos del juego: clases Bug, Fruit, Poison, Obstacle y SnakeLink.

Las clases Bug, Fruit, Poison, Obstacle y SnakeLink representan respectivamente un bicho, una fruta (comida), un veneno, un obstáculo y un eslabón de la serpiente. Todas ellas extienden AGameObject. El único método que tienen que implementar necesariamente es setCoordinate.

De todas ellas, la única que se da completamente implementada es SnakeLink. El resto tendrán que implementarlas los alumnos.

Implementación del personaje del juego: clase Snake.

Las clase Snake (figura 7) extiende AGameObject (es un elemento más del juego y si hace falta puede tratarse como tal), pero además implementa la interfaz IGameCharacter por lo que puede tratarse como un tipo independiente. Por si fuera poco, está compuesta a su vez de elementos de juego (instancias de SnakeLink).

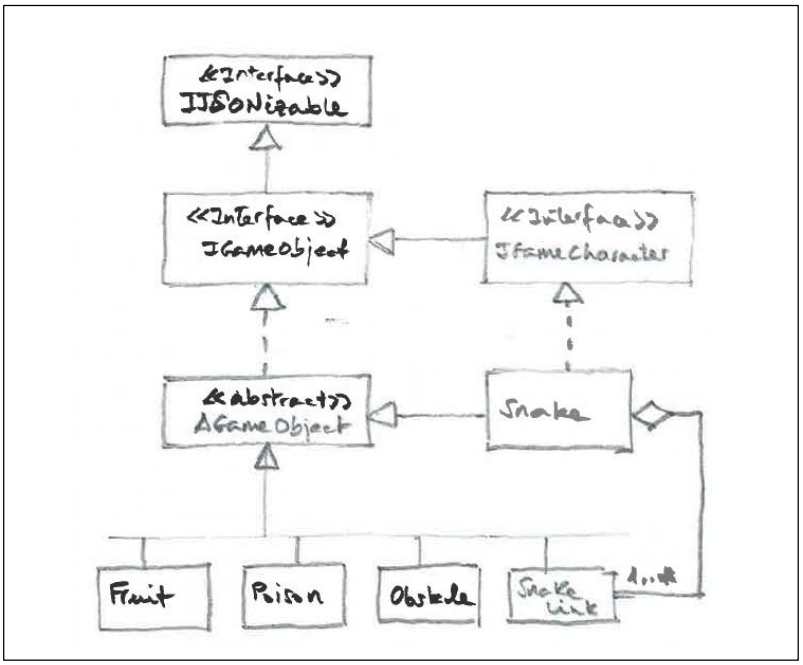

Figura 7: La estructura de clases e interfaces de la aplicación.

Un punto interesante de esta clase es que contiene una lista de objetos SnakeLink que se convierte en un array de objetos JSON al ser serializada (véase en el código el método toJSONObject). De manera similar, en el constructor de serialización hay que reconstruir la lista de eslabones (instancias de SnakeLink) a partir de un array de objetos JSON que representan la versión serializada de los objetos SnakeLink.

Volcado de datos en una base de datos: clase TestBD.

La clase TestBD contiene un ejemplo de guarda de un objeto Snake en la base de datos er_gusano suministrada en el aula virtual.