Proyecto

SmartFarm - PARTE 1 (10 %)

Objetivos:
Practicar de manera más autónoma los conceptos trabajados en la asignatura hasta el momento.

Herramientas que vamos a utilizar:

- Herramienta de diseño StarUML
- Entorno de desarrollo Eclipse

Entregas Se realizarán dos entregas en esta primera parte del proyecto[1]. En cada entrega se nombrará el fichero comprimido con el siguiente formato: *apellido1_apellido2_nombre*. Recordad que la entrega es individual y se subirá a eGela en las fechas indicadas.

- *Entrega1:* Diseño con StarUML de la aplicación. Se debe entregar:
  - El fichero **.uml** (apellido1_apellido2_nombre_SmartFarming.uml)
  - La imagen exportada con el diagrama de clases del proyecto en .**jpg** o.**jpeg** (apellido1_apellido2_nombre_SmartFarming.jpg).

  Fecha límite de entrega: miércoles 31 de marzo a las 23:55

- *Entrega2:*
  - Proyecto exportado (apellido1_apellido2_nombre_parte1.zip) de esta primera parte completa, tal y como se indica en las tareas a realizar (*se podrá volver a entregar también el diagrama de clases en formato uml y jpg, si éste ha sufrido cambios significativos con respecto a la entrega anterior*).
  - Documento con los siguientes apartados:
    - Información de los cambios en el UML actual con respecto al presentado en la 1ª entrega (si es que procede).
    - Explicaciones de aquellas partes de la entrega de las que se duda de su correcto funcionamiento o que directamente no funcionan correctamente
    - Informe de todo lo realizado y los problemas que se han tenido en su desarrollo.
    - Estudio de casos de prueba considerados para la elaboración de los Junit para los que no se hace el estudio.
    - Captura de una ejecución completa

  Fecha límite de entrega: domingo 25 de abril a las 23:55

¡¡ATENCIÓN!!, para que se corrija la práctica se deben respetar todos los nombres indicados (ficheros, proyectos, clases etc.). Además, el proyecto no podrá tener errores de compilación y su ejecución no terminará por una excepción.

[1] Los alumnos en evaluación GLOBAL podrán realizar una única entrega el 25 de abril con todo lo indicado en las dos entregas.

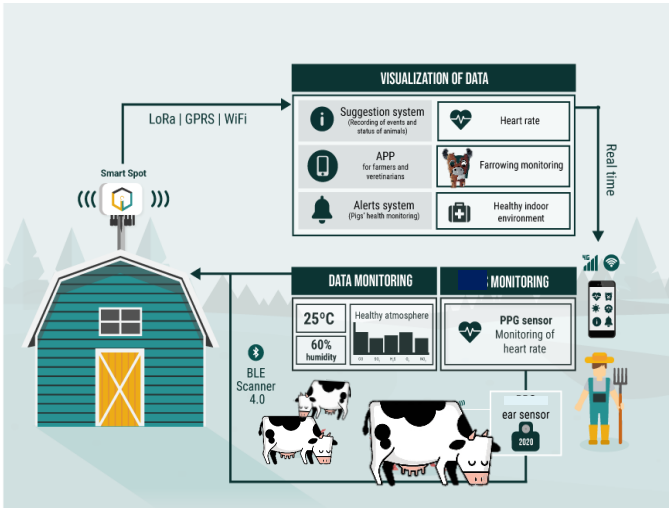
Visión general de la aplicación

En este proyecto vamos a diseñar, desarrollar y probar una aplicación para gestionar **una granja inteligente**. En la granja tendremos animales que estarán monitorizados mediante sensores.

La monitorización de la salud animal en granjas contribuye a la mejora de la eficiencia de la producción y el confort de los animales, así como a la calidad del producto que llega al consumidor o consumidora final.

Gracias a los sensores que incorporan los animales de granja, se pueden controlar la temperatura, la frecuencia cardíaca y los movimientos de cada animal, parámetros básicos para saber si la cabeza de ganado está enferma, incluso antes de que muestre síntomas evidentes. Es decir, la medición continua de estos parámetros aporta información detallada sobre el estado de los animales de la granja con el objetivo de detectar de forma temprana enfermedades o posibles bajas.

<u>Tareas a realizar</u>:



Para poder realizar cada una de las tareas señaladas, deberás prestar atención al diseño propuesto de la aplicación (en el apartado A Diseño de la aplicación). Como se ha indicado anteriormente serán dos los entregables:

Tarea 1- Diseña con UML (Entrega 1)

Tarea 1. Diseña con StarUML la aplicación mediante un diagrama de clases para las clases[2] que hayan surgido de los requerimientos expresados en el diseño. No es necesario que aparezcan las clases excepción definidas en el diseño, pero sí debe aparecer la información pertinente de los métodos que pueden elevar excepciones. Una vez completado todo el diagrama obtén el esqueleto Java de la aplicación.

Tareas 2 a 5- Implementa, documenta y prueba la aplicación (Entrega 2)

Tarea 2. Implementa y documenta el diseño realizado. Recuerda que puedes partir del esqueleto Java creado directamente por StarUML. No olvides que todo debe ir documentado mediante JavaDoc, y **se debe generar la documentación**.

Tarea 3. (opcional recomendable) Como ayuda conviene ir creando para cada clase implementada de la aplicación, otra clase que permita crear pequeñas pruebas de funcionamiento. Incluye todas las clases de pruebas en una carpeta **`packcheck`**.

Tarea 4. Diseña e implementa[3] con JUnit en una carpeta **`packtest`**, todas las pruebas necesarias mediante estudio de caja negra (clases de equivalencia y casos límite si son necesarios) para las clases y métodos expresados en el apartado B de Tareas de Evaluación JUnit

Tarea 5. Finalmente, implementa la clase **`FarmSimulator`** en el paquete **`packsimulator`** para que en su método **`main`** realice las tareas indicadas en el apartado A de diseño.

[2] No es necesario representar las excepciones

3 Quizás para hacer correctamente las pruebas y poder volver a estados conocidos, sea necesario añadir alguna operación más en alguna clase no considerada inicialmente.

A. Diseño de la aplicación

En el diseño de la aplicación deben aparecer en el package `packfarm` las clases **`PhysiologicalValues`**, **`Sensor`**, **`FarmAnimal y Farm.`** La clase **`FarmSimulator`** pertenece al package `packsimulator`. Todas las clases deben ajustarse a la descripción y comportamiento que se describe a continuación. Debe estar sobrescrito en todas las clases el método `toString` con la semántica habitual y según se indique en cada clase. Además, el método `equals` se implementará siempre que se crea necesario (bien porque se vaya a usar directamente o a través de otros métodos que lo necesiten).

1. La clase **`PhysiologicalValues`** representa los parámetros fisiológicos de un animal.
   a) Los elementos para los que se guardan valores son: heart rate (int), temperature (double, grados Fahrenheit) y activity (int, es un valor entre 1 y 5 pero no lo comprobaremos).
   b) Tendrá una constructora con un parámetro por cada atributo.
   c) Tendrá getters de todos los atributos.
   d) El método `toString`: obtendrá un `String` con el valor de todos los atributos separados por un espacio en blanco y en el mismo orden de la descripción. Por ejemplo, "75 101,75 4" (siendo el valor 75 el del ritmo cardiaco, 101,75 la temperatura en grados ºF y 4 el valor de la actividad).

2. La clase **`Sensor`** representa a los sensores que se utilizan para monitorizar los animales de la granja.
   a) Está caracterizado sólo por un identificador (`String id`).
   b) Se creará con un identificador dado.
   c) El método `collectValues` tendrá como objetivo recoger y devolver los valores fisiológicos del animal en el momento actual. Para implementarlo será necesario usar el método privado `collect` (se da implementado y lo tenéis en el apartado C al final del documento). Es posible que la captura de valores mediante el método `collect` eleve la excepción `CollectErrorException`. Nuestro método `collectValues` seguirá elevando dicha excepción.
      - Define la excepción `CollectErrorException` con una única constructora con el parámetro `String`. Observad que el método `collect` eleva dicha excepción y hace uso de esa constructora. Define la excepción como clase independiente en el package `packexceptions`.
   d) El método `toString` devolverá un String con el valor del id.

3. La clase **`FarmAnimal`** representa cualquier animal de la granja. Todos están monitorizados, hasta que el animal vaya a abandonar la granja.

a) Esta clase se caracteriza por su `id` (`String`. El id será único y fijo para cada animal de la granja. El identificador será una constante), `age` (int), `weight` (double), `mySensor` (Sensor) y `myValues` (una lista de valores fisiológicos que se van recolectando durante la semana). Observad que la información se irá registrando poco a poco, por lo que se debe controlar cuál es la posición de registro en cada momento. La lista se implementará obligatoriamente con un Array[4] por ello podrás añadir algún atributo más para su gestión. Si hiciera falta se consideraría al id como discriminante de la igualdad de dos animales.

b) Tendremos dos constructoras. Una con un único parámetro id. Y la otra constructora con parámetros para el identificador, edad, peso y sensor. Ambas deben inicializar la estructura Array debidamente para su uso posterior.

c) Los métodos getters: `getId`, getAge, `getWeight` y `getMySensor`.

d) Los setters de todos los atributos (salvo el identificador y la lista)`.`

e) El método `toString` deberá obtener un `String` con el identificador, edad, peso e identificador del sensor (separados por un espacio en blanco). Por ejemplo, "BOV124 10 475 ID45" podría ser el `String` de un animal de la granja, siendo BOV124 el identificador del animal, 10 la edad, 475 los Kg e ID45 el identificador del sensor que lleva.

f) El método `register,` permite obtener, registrar y devolver los valores fisiológicos del animal en el momento actual.

- Si durante la recogida de valores del sensor se eleva la excepción `CollectErrorException,` el método `register` deberá tratarlo e insistir hasta conseguir dichos valores (está demostrado que falla un pequeño porcentaje de veces, pero siempre acaba por dar los valores).
- Además, este método deberá considerar que si el registro se solicita cuando ya está lleno el array (han pasado 7 días de registro), antes de registrar los nuevos datos:
  - Deberá volcar toda la información en la carpeta `data` (habrá que crearla explícitamente dentro del proyecto como un folder) en el archivo "`historicalValues.txt`"`.` Para implementarlo habrá que definir un método auxiliar `storeValuesInFile`, considerando las siguientes restricciones:
    - Se escribirá la información en modo **append**[5]. Es decir, se seguirá conservando la información que tuviera ya el archivo.
    - Primero escribirá una línea con la información del animal.
    - Después escribirá 7 líneas una para cada uno de los valores fisiológicos de cada día de la semana.
    - Para escribir cada línea hará uso del método `toString` pertinente.
  - Inicializará el Array de datos fisiológicos con un método auxiliar `initWeek`.

g) El método `avgTemperature,` permite consultar la temperatura media del animal durante los días ya almacenados`.` Si todavía no hay ningún registro hecho devolverá el valor 0.

4 No se acepta estructura `ArrayList`

5 `FileWriter(fileName, true)`

4. La clase **`Farm`** debe incluir toda la información necesaria para la monitorización de los animales de la granja. Lo representaremos como un patrón `Singleton`, ya que la información relativa a la monitorización debe estar centralizada y debe ser única.

a) La granja se caracteriza por el conjunto de animales monitorizados (`farmAnimalsSet`) y todos los sensores disponibles que hay en la granja (`sensorList`). Ambas características se implementarán con la estructura **`ArrayList`**.

b) El método `addFarmAnimal` permite incorporar un nuevo animal a la granja. Dados el identificador, edad y peso del animal, se le asigna el primer sensor disponible (eliminándolo de la lista) y se incorpora al conjunto de animales de la granja. Si no hay ningún Sensor disponible se lanzará la excepción `IndexOutOfBoundsException` (debe quedar declarada en el método).

c) Método `howManyAnimals`, que indica cuántos animales hay actualmente en la granja.

d) Método `addSensor`. Dado un sensor lo añade a la lista de sensores disponibles.

e) Método `howManySensors`, que indica cuántos sensores disponibles hay actualmente en la granja.

f) Método `register`, que solicita el registro de los valores fisiológicos de todos los animales de la granja.

g) Método `obtainPossiblySick`. Obtiene una lista de aquellos animales de la granja cuya temperatura media de esta semana es mayor que un valor **max** dado.

h) Método `obtainFarmAnimal`, que dado un identificador devuelve el `FarmAnimal` de ese identificador. El código siempre pertenecerá a un animal de la granja.

i) Método `removeFarmAnimal`. Dado el identificador de un animal lo da de baja en la granja y lo devuelve. Además, se deberá recuperar su sensor para su reutilización.

j) Método `obtainFarmAnimalOlder`, que dada una edad (int) devuelve un ArrayList con los identificadores de los animales de la granja mayores que esa edad.

k) Método `farmAnimalsDeparture` permitirá la salida de la granja de los animales que superen un peso dado, pero que no estén posiblemente enfermos con respecto a una temperatura **max** dada (no la deben superar). Se eliminan dichos animales de la granja, pero se recuperarán sus sensores. Se devolverá una lista con los identificadores de los animales que cumplen dicha restricción de peso.

5. Finalmente, la clase **`FarmSimulator`** llevará la gestión de la granja inteligente simulando en su método `main` el comportamiento de creación de la granja, de todos sus sensores, de los animales y el control de todo ello**.** Todas las excepciones que se puedan elevar se tratarán adecuadamente escribiendo un mensaje acorde al error. Recordad que el main no debe elevar ninguna excepción y que la captura de una excepción no debe impedir que se siga ejecutando el main.

a) Generar y obtener el objeto que representa la granja.

b) Carga la información de los sensores disponibles en la granja que están recogidos en el fichero "`availableSensors.txt`" y que deberá estar situado en la carpeta `data` del proyecto. Hay un identificador en cada línea. Crea un método privado `static loadSensors` para hacerlo y llámalo desde el main. Imprime después cuántos sensores disponibles hay en la granja.

c) Carga la información de la granja que está recogida en el fichero "`smartFarm.txt`" que deberá situarse en la carpeta `data` del proyecto. Dicha información se corresponde con la información de cada animal de la granja al principio de la semana. Para implementarlo crea el método privado `static loadFarmAnimals` y llámalo desde el main.
El fichero "`smartFarm.txt`" recoge en cada línea la información de un animal. Consideraremos que los ficheros son siempre correctos. Por ejemplo:

```
BOV128 2 620.62
BOV133 5 587.17
BOV138 17 549.06
BOV143 16 562.28
...
```

// BOV128- código del animal de la granja
// 2- edad
// `620.62`- kilos

d) Obtén la lista de los identificadores de los animales de edades superiores a 17 años. Imprime cuántos animales hay en la granja mayores de esa edad.

e) Solicita el registro de valores fisiológicos de todos los animales mayores de 17 años. Para implementarlo crea el método privado `static registerOlder` y llámalo desde el main.

f) Solicita 7 veces el registro de los valores fisiológicos de todos los animales de la granja.

g) Comprueba que todo funciona correctamente y que se ha guardado la información relativa a los registros de los animales de más de 17 años en el fichero "`historicalValues.txt`".

h) Imprime por pantalla cuántos sensores disponibles hay en la granja.

i) Añade un nuevo animal a la granja (con código "XXXX").

j) Obtén e imprime cuántos animales hay posiblemente enfermos con una temperatura superior a 102 ºF.

k) Obtén y elimina los animales que tengan más de 500 kilos pero que no tengan temperaturas superiores a 102.0 ºF. Imprime cuántos animales han abandonado la granja.

l) Imprime por pantalla cuántos sensores disponibles hay en la granja.

B. Tareas de Evaluación JUnit

Evalúa con JUnit las clases y métodos siguientes:

- Clase **`FarmAnimal`**:
  - método **`register`**. Casos
    - Todavía no hay ningún registro.
    - Cuando hay un registro
    - Hay varios registros (unos 4)
    - Último registro de la semana.
    - Registra uno cuando ya está lleno el registro de la semana
  - (opcional) método **`avgTemperature`**. Casos
    - Todavía no hay ningún registro.
    - Cuando hay un registro
    - Hay varios registros (unos 4)
    - Último registro de la semana (7 registros).
    - Registra uno más cuando ya está lleno el registro de la semana
- Clase **`Farm`**:
  - método **`addFarmAnimal`**. Determina cuáles son los casos necesarios a realizar.
  - (opcional) método **`removeFarmAnimal`**. Determina cuáles son los casos necesarios a realizar.

C. Implementación dada

```
/**
 * Simulates physiological values collection with potential failure
 * @return the physiological values collected
 * @throws CollectErrorException when not possible to return the values
 */
private PhysiologicalValues collect() throws CollectErrorException{
   Random r= new Random();
   int heartRate= 0;
   double temperature = 0.0;
   int activity= 0;
   // Simulates the possibility or not of values capture
   heartRate = 60+ r.nextInt(20);

   temperature = 101.5+r.nextDouble();
   temperature = Double.parseDouble(String.format("%.2f", temperature).
                                                          replace(',', '.'));
   activity = r.nextInt(5)+1;

   PhysiologicalValues phy=
                          new PhysiologicalValues(heartRate, temperature, activity);
   if (Math.random()>=0.9) throw new CollectErrorException(phy.toString());
   return phy;
}
```