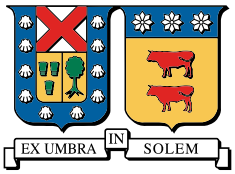

ELO329:POO

Tarea I: Documentación

Profesor: Agustín Gonzalez

Alumnos:
Leonardo Solis 201104505-k
Diego Riquelme 201303041-6
Gabriel Rudloff 201303044-0

13 de abril de 2018

Introducción

Esta tarea consiste en modelar un robot que ingresa en un laberinto y busca la manera de salir de el. Este robot contará con tres sensores que le permitirán saber si hay murallas a sus costados y frente a él. Tambien tendrá la facultad de moverse en cualquier dirección a una velocidad determinada. Luego con la información de los sensores, deberá ejecutar cierto algoritmo que le permita resolver el laberinto.

Para lograr este objetivo se utilizó una metodología de desarrollo incremental. Esta se dividió en cinco etapas que cumplen diferentes objetivos de avance.

La primera Etapa, consiste es modelar un robot que tuviera la capacidad de moverse según algún patrón definido. Este recorrido debe ser registrado para luego ser mostrado.

La segunda Etapa, tenía por objetivo la lectura y escritura del laberinto. Debía tomar un archivo de datos binarios que tenia representado el laberinto, para luego reescribirlo pero noventa grados rotado.

La siguiente Etapa, añadía los sensores de distancia asociados al robot. Se creó una clase que los caracterizaba. Estos tienen un rango de funcionamiento y no tienen movimiento propio. El robot debe implementar 3 sensores, para que cada movimiento vayan tomando lectura y le aporten información de las paredes cercanas.

Posteriormente se definió una cuarta etapa que tenia por rol crear una clase *Pilot*, responsable del movimiento de los sensores y del robot, de manera que se muevan juntos de la manera adecuada. Ademas contará con el algoritmo de toma de decisiones responsable de llevar el robot a la salida.

La quinta etapa, lleva por meta lograr implementar una clase que extienda de *Pilot*, y que ejecute un algoritmo diferente.

A continuación se describirán alguna de las de las clases y sus métodos que requieran más explicación que la del código mismo, que fueron programadas para el cumplimiento de las etapas.

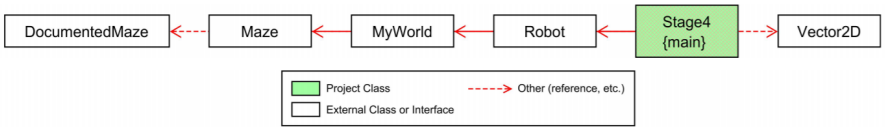

Figura 1: Diagrama de Clases

Pilot

Esta clase está definida en Robot, sus métodos refieren al movimiento del robot y sus sensores.

setCourse(*delta_t*)

Se utilizan *if statements* para designar la estrategia que llevara al robot pegado a la muralla derecha. La variable *lookingForRightWall* es de valor booleano, y comienza preguntando si el sensor derecho (lado que seguirá la pared) esta activado, en caso de no estarlo, se activara en *True*, luego si se encontrase la pared derecha, tomaría el valor de *False*. Sin embargo si no encuentra la pared derecha seguirá andando hacia adelante.

DistanceSensor

Esta clase también definida dentro de Robot, tiene el rol de describir el sensor de distancia, ademas de describir los métodos que lo hacen moverse junto con el robot. Lleva por entrada la dirección del sensor y el rango (alcanze) de este.

senseWall()

El método senseWall(), tiene por rol verificar si en la dirección que apunta el sensor, detecta una pared dentro de su rango.
Esta verificación la logra gracias al método *isThere_a_wall* el cual compara con el arreglo del laberinto. En caso de que la coordenada preguntada no se encuentre en el rango del arreglo, se asumirá que hay una pared. Esto para que el robot no salga del rango permitido.

Maze

rotate()

En el constructor de maze se genera un arreglo de tamaño [hight][width], el objetivo de este método es reemplazar el arreglo original con un arreglo que sea el original girado en $+90\,\text{deg}$ (sentido antihorario). Para este objetivo se genera un arreglo auxiliar de tamaño [width][hight], luego para cada par h, w posible dentro del arreglo se debe asignar $array\_aux[w][h] = array[h][width - 1 - w]$; Podríamos describir esto como que se transpone la matriz, es decir en cada columna nueva se coloca lo que anteriormente era una fila, pero ademas cada una de estas filas se invierte. Luego se le asigna al arreglo original el arreglo auxiliar creado. Este método solo se encuentra en maze en la etapa 2 ya que solo en este es necesario.

write(PrintStream out)

Este método imprime para cada posición de $array$ un "1"si es "truez un "0"si es "false". Esto se logra fácilmente recorriendo con dos for anidados cada posición del arreglo y luego con un if consultar el valor de este para imprimir "1.º "0".

isThere_a_wall(int x, int y)

El objetivo de este método es saber si existe pared en el lugar que se testea, junto a esto se agrega en la etapa 4 un if con el objetivo de evitar problemas al momento de llegar al borde del mapa, para esto se eligió que el comportamiento sea ver como pared si esta testeando para algún valor fuera del rango de $array$.

markPoint(Vector2D p)

Este método setea la posición dada por el vector de entrada como true, este es para marcar el camino del robot. Los valores de posición se castean a int, ya que estos son double, pero necesitamos un int como indice del arreglo.

read(Scanner sc)

Es el mismo código que previamente se tenia dentro del constructor principal de Maze, este método se agrega en la etapa 4 para permitir la implementación de DocumentedMaze que extiende a Maze, esta nueva clase en su constructor lee una linea previo a leer los datos del laberinto, por lo que es necesario que haga eso primero y luego se pasa el *Scanner* a *read*() para que siga leyendo el archivo.

Maze(Maze m)

A este constructor se le llama copy constructor y se agrega en la etapa 4. Uno no puede directamente asignarle a un arreglo otro arreglo, ya que esto lo que hace es hacer que ambos apunten al mismo arreglo, por esto es necesario recorrer el arreglo y elemento por elemento asignar al nuevo arreglo. Pero existe un comando perteneciente a la librería de java, que es System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length), que copia arreglos completos (1 dimensionales) por lo que basta iterar en una dimensión y copiar cada uno de los arreglos.

MyPilot

MyPilot(boolean u)

Mismo constructor de Pilot, pero con entrada booleana u, para que ante el caso de que se elija usar la alternativa 2, es decir que u sea $true$, entonces se setea $lookingFor$ para que ante el caso de que no este al lado de una pared en su lado izquierdo se avance hasta encontrarla.

setCourse2(double delta_t)

Análogo a setCourse() de Pilot, pero con la lógica de seguir la pared por la izquierda.

Problemas

Uno de los problemas que se tuvo fue que debido al rango de los sensores, eventualmente tomarían coordenadas negativas para algunos valores, o bien fuera del rango definido por el tamaño del archivo. Esto arrojaba error, ya que dentro del método para consultar si en el rango del sensor había o no una muralla, se debía consultar un arreglo en las coordenadas en cuestión y verificar si era un 1 (muralla), sin embargo al no estar definido el arreglo para elementos fuera del tamaño del archivo arrojaba error.
Esto se solucionó poniendo un *if statement* que consultara si la coordenada a consultar se ubicaba dentro del rango del arreglo, si no, entonces retornaba *True*, significando que en ese lugar había una pared. En otras palabras se asumió que por los limites del laberinto hay paredes, esto para evitar que el robot se

"caiga".

Otro problema se tuvo fue de que si la velocidad del robot era suficientemente grande, se saltaba paredes, esto se debía a que al consultar el estado de los sensores y no detectar nada frente a él; el robot, de avanzar lo haría a la velocidad indicada, terminando, luego de la iteración cierta cantidad de pixeles adelante. Ahora si la velocidad fuera muy grande, eso podría ser al otro lado de una pared, o bien en una pared. Debido a esto, es que se eligió de criterio no elegir velocidades mayores a una magnitud de *sensorRange*[pixeles/iteración]. Esto es coherente, debido a que si en el rango del sensor no detecta una pared, que avance esa distancia como máximo.

Un inconveniente que se tuvo al desarrollar el código fue que no puede asignar un arreglo a otro, ya que lo que en realidad se esta haciendo es pasar la misma dirección de memoria de donde apunta, y no una nueva dirección de memoria con el mismo valor dentro *i.e,* un nuevo arreglo con los mismos valores. Esto se solucionó utilizando un método de la librería de Java llamado *System.arraycopy*, que lo que hace es crear un nuevo arreglo y asignar elemento por elemento los valores de modo de obtener ahora la copia del arreglo.