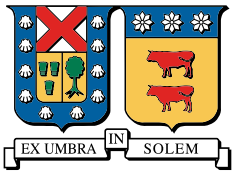

ELO329:POO

Tarea III: Documentación

Profesor: Agustín Gonzalez

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

31 de Julio de 2018

Introducción

Esta tarea consiste en modelar un robot que ingresa en un laberinto y busca la manera de salir de él, tal como se hizo en la primera tarea del ramo. Este robot contará con tres sensores que le permitirán saber si hay murallas a sus costados y frente a él. También 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. La diferencia principal ahora, es que se utiliza el lenguaje de programación *C++*, el cuál tiene bastantes aspectos que lo hacen diferente a *Java*.

Para lograr este objetivo se utilizó una metodología de desarrollo incremental, tal como se hizo en las dos tareas anteriores. 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, rotación 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. Además 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 algunas de las clases y sus métodos que requieran más explicación que la del código mismo, que fueron programadas para el cumplimiento de las diferentes etapas.

Diagrama UML etapa final

A continuación se presenta el diagrama correspondiente a la etapa final

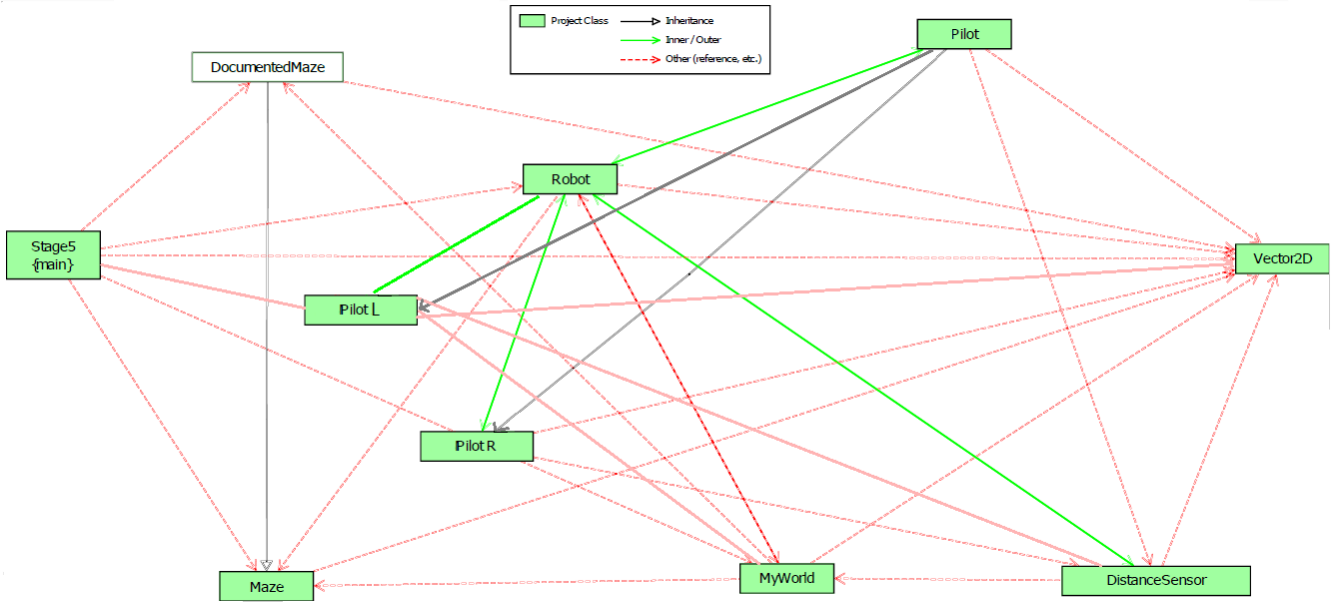

Figura 1: Diagrama de Clases

A continuación se realizará una breve explicación de las clases utilizadas.

Relación entre las clases

En esta sección se explicara la relación de las clases en la etapa final, ya que las etapas anteriores son versiones incompletas de esta.
El *stage5* es el *Main* del programa, donde además de crear el Robot, se le entrega un Maze. Esto se agrega a la instancia de MyWorld, la cual luego se simula a través del método *simulate()*.

Robot contiene dos clases internas. Estas son Pilot y MyPilot la cual especifican el algoritmo de solución del laberinto, el cual es generado gracias a la clase DocumentedMaze la cual extiende de Maze.

Cabe aclarar que clases como Robot y Maze hacen uso de vectores con instancias de la clase Vector2D.

Pilot

Esta clase está definida en *Robot* como una clase anidada de carácter *private*. Sus métodos refieren al movimiento del robot y sus sensores. Lo que buscará realizar esta clase, es poder guiarse a través del sensor derecho para que el robot avance.

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 tipo *bool*, y comienza preguntando si el sensor derecho (lado que seguirá la pared) esta activado. En caso de no estarlo, se activará en *True*. Luego, si se encontrase la pared

derecha, *lookingForRightWall* tomaría el valor de *False*. Sin embargo, si no encuentra la pared derecha seguirá hacia adelante.

DistanceSensor

Esta clase también definida dentro de Robot como una clase anidada de carácter *private*, tiene el rol de describir el sensor de distancia además de describir los métodos que lo hacen moverse junto con el robot. Lleva por entrada la dirección del sensor, el rango (alcance) de éste y además un puntero tipo *Robot*, para que la clase tenga acceso a los atributos privados de la clase padre, que para este caso sería la clase *Robot*.

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 pero 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 además cada una de estas filas se invierte. Luego se le asigna al arreglo original el arreglo auxiliar creado. Luego, para liberar la memoria ocupada por el arreglo auxiliar, es necesario ocupar *delete*, con tal de eliminar dicho arreglo que sirvió para trasponer el arreglo original. Esto, a diferencia de Java, se hace de manera manual, liberando así la memoria para que ésta pueda ser utilizada. El metodo anteriormente mencionado sólo se encuentra en **maze** en la etapa 2, ya que sólo en esta etapa es necesario.

write(ofstream &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". Estas impresiones son hechas en un archivo de salida llamado ***maze_out.pbm***.

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 índice del arreglo.

read(ifstream& maze_in)

Es el mismo código que previamente se tenía dentro del constructor principal de *Maze*. Este método se agrega en la etapa 4 para permitir la implementación de *DocumentedMaze* que es una clase anidada de *Maze*. Esta nueva clase en su constructor lee una línea previo a leer los datos del laberinto, por lo que es necesario que haga eso primero y luego se pasa el *maze_in* a *read()* para que siga leyendo el archivo, como se hacía en la etapa 2.

Pilot - Stage 5

Robot(Vector2D pos,...,bool u)

En el constructor del robot, con la entrada tipo *bool u* se elige **PilotL** (u=true) o **PilotR** en caso de que se requiera que el robot avance tocando la pared izquierda o derecha, respectivamente. Para eso, la clase **Pilot** (clase anidada de **Robot**) se reescribe en ésta etapa, teniendo ahora el método abstracto *setCourse()*, la cuál será implementada de manera diferente para **PilotL** o para **PilotR**, de manera que se pueda realizar la estrategia pertinente.

Dificultades y soluciones

Dependencia circular

Este problema se generó ya que **Robot** tiene una instancia de **MyWorld** y **MyWorld** tiene una instancia de **Robot**, lo que genera una dependencia circular. Para resolver este problema, fueron necesarias dos cosas. Primero, saber que antes de definir la clase **Robot**, se define la clase **MyWorld** pero sin ningún prototipo ni nada, sólo para saber que la clase existe. Luego se ocupa un puntero a **MyWorld**, que es atributo de la clase Robot para así poder referirse a la clase **MyWorld**.

Creación de la Matriz

Para la creación de la matriz de tipo bool, en la etapa 2 se puede ver que ésta se creó con Vectores pero luego se siguió trabajando con punteros dobles. Esto se hizo así debido a que no se podía tener como atributo un arreglo doble cuya dimensión fuese indefinida. Las dos estrategias utilizadas dieron solución a este problema y finalmente se eligió seguir con la más conveniente. A diferencia de Java, C++ no permite la creación de arreglos cuya longitud sea indefinida.

Creación del Makefile

Para la creación del Makefile, se sigue una secuencia de pasos que es totalmente diferente a los pasos seguidos en Java. Para esto, fue necesario buscar la documentación pertinente para entender como se llevaba a cabo la generación del archivo makefile.

Lectura y escritura de archivos

Para poder leer y escribir un archivo con C++, fue entender los diferentes métodos de *ifstream* y de *ofstream*. Mediante estos métodos se implementó la lógica que puede realizar lo mismo que se hizo en

la tarea 1; eso sí, la diferencia era que los métodos son distintos a los de Java, por se aplicaron nuevas tácticas para dar con la manera correcta de leer un archivo externo y entregar un archivo.