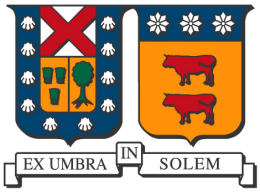


Documentación

Tarea 3: Robot en Laberinto como Objeto de Software en C++

Programación Orientada a Objetos - ELO 329
Departamento de Electrónica

Valparaíso, 31 de Junio del 2018

Integrantes	**Rol**
Matías Contreras	201321034-1
Damian Quiroz	201321056-2
Francisco Frez	201321007-4
Profesor	Agustín González
Ayudantes	Jesús Márquez
	Pilar Arancibia

Índice

1. Objetivos

En esta tarea se inicia con la programación en C++, lenguaje híbrido (presenta a C como base y comparten algunas características) de programación orientada a objetos mediante el desarrollo de un robot al interior de un laberinto, tal que este debe buscar la manera de salir previa identificación de la salida por el usuario.

Dado el desafío presentado, se busca llevar a cabo la solución mediante desarrollo interativo e incremental, considerando los siguientes objetivos:

- Reconocer clases y relaciones entre ellas en lenguaje C++.
- Ejercitar la entrada y salida de datos en C++.
- Ejercitar el formato .csv y su importación a una planilla electrónica.
- Ejercitar la preparación y entrega de resultados de software (creación de makefiles, readme, documentación, manejo de repositorio -GIT).
- Familiarización con una metodología de desarrollo iterativa e incremental.

Se destaca que este lenguaje de programación separa en dos archivos respecto de Java, tal que se genera un archivo de cabecera ("clase.h") donde se encuentra la **definición** de la clase (atributos y prototipos de métodos) y otro archivo ("clase.cpp") donde está la **implementación** de cada método del archivo de cabecera.

2. Descripción General

Se pide modelar un robot que ingrese a un laberinto y busque la manera de salir de él por un lugar distinto al de ingreso. Se considera un robot muy simple que puede desplazarse sobre un plano y reconocer las paredes mediante sensores de proximidad (se incluyen tres: dos laterales y uno en la dirección del movimiento).

Este robot se puede mover en todas direcciones, tal que se espera que sus sensores sean capaces de evitar colisiones al moverse por el interior del laberinto que será modelado como segmentos de líneas paralelas y perpendiculares entre sí con una entrada y una salida como mínimo (pueden incluirse más entradas y salidas). Esta mazmorra es cargada como archivo *pbm* en una de las etapas de la presente tarea.

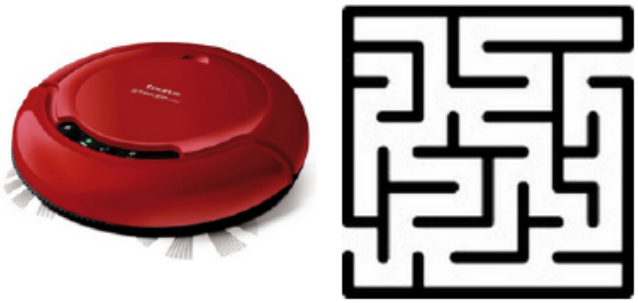


Figura 1: Idea de Robot (izquierda) y Laberinto (derecha) a implementar.

3. Desarrollo por Etapas

Se aplica una metodología "iterativa" e "incremental" para el desarrollo de esta solución, obteniendo en cada etapa un subconjunto del requerimiento final.

3.1. Etapa 1: Robot Movible

Esta etapa pide implementar un robot con capacidad de moverse en base a un patrón definido e irá registrando su posición mientras se mueve.

3.1.1. MyWord.h / MyWorld.cpp

Clases que generan el mundo simulado y define la interacción entre las diferentes clases, constructores y métodos. Su método principal es **simulate(double delta_t, double endTime)** que se encarga de iniciar el tiempo de la simulación, en donde ciertos elementos (p.e. el robot) son capaces de calcular su próximo estado.

3.1.2. Robot.h / Robot.cpp

Clases que definen los atributos del robot a utilizar, implementando los métodos necesarios para obtener su posición **getPosition()** y descripción **getDescription()**, además de definir una serie de movimientos: giro a la derecha **turnRight()**, giro a la izquierda **turnLeft()** y avance por el plano **moveDelta_t()**.

3.1.3. Vector2D.h / Vector2D.cpp

Clases que poseen como atributos las variables x e y (cartesianas), implementando métodos correspondientes a un vector en dos dimensiones tales como: **operator+()** (suma), **operator*()** (producto escalar), **operator-()** (resta), **setTo()** (setea nueva posición), **getX()** (obtiene variable x) y **getY()** (obtiene variable y).

3.1.4. Stage1.cpp

Clase que contiene el método **main()**, donde se definen los parámetros y variables necesarias de inicio del programa. Está relacionada en forma directa con todas las demás clases descritas al requerir de ellas para iniciar.

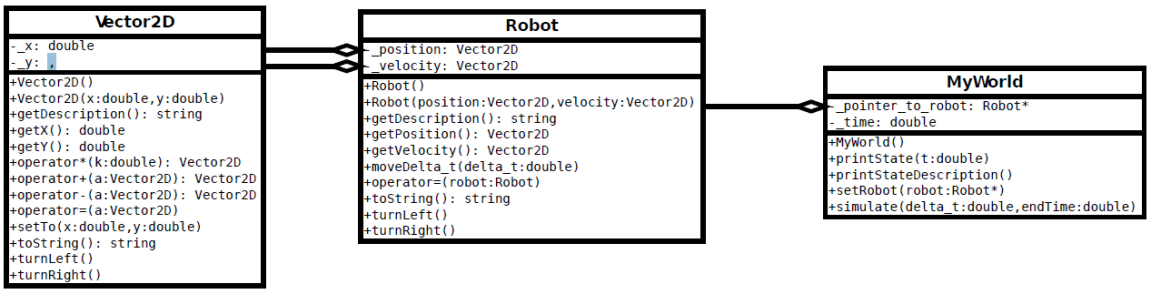


Figura 2: Diagrama UML Stage 1.

3.1.5. Resultados

Se ilustra en la Figura 3 el movimiento del robot en el plano cartesiano, tal que se nota que la dirección del **eje y** está cambiada respecto de la imagen dada como resultado esperado. Esto sucede debido al cambio de dirección realizado en los parámetros iniciales del archivo **"Stage1.cpp"** con tal de evitar futuros inconvenientes dada la forma en que se recorrerá el laberinto en la Stage3.

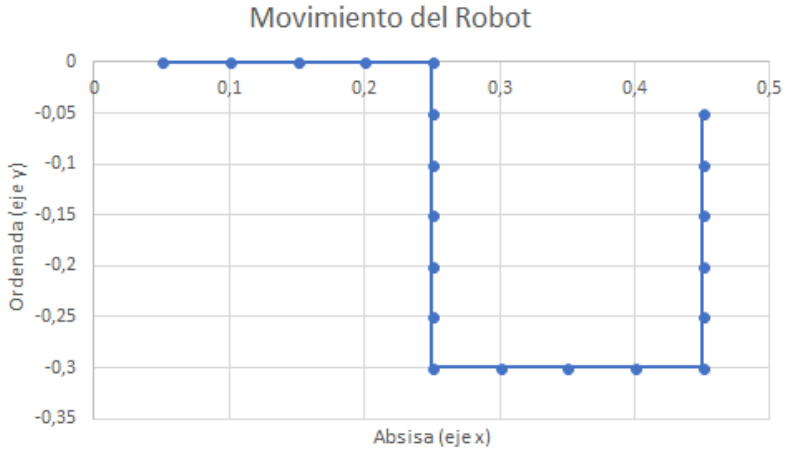


Figura 3: Movimiento del Robot en Coordenadas Cartesianas.

3.2. Etapa 2: Lectura y Escritura de Laberinto

Esta etapa es donde el programa recibe un archivo con el laberinto, leyendo dicho archivo y almacenándolo en una instancia de la clase "Maze", lo rota en 90^o a la izquierda y genera un archivo de salida ("maze_out.pbm") con el laberinto rotado. Se nota que no se requiere de las clases descritas en forma previa.

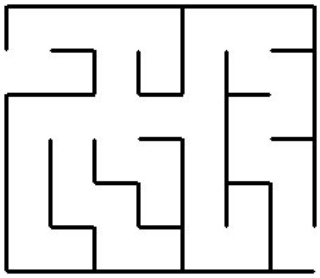


Figura 4: Idea de Robot (izquierda) y Laberinto (derecha) a implementar.

3.2.1. Maze.h / Maze.cpp

Clases que implementan los métodos principales **Maze()** (obtiene el largo y ancho del laberinto, genera una matriz auxiliar y guarda los valores en ella), **rotate()** (se encarga de rotar el laberinto) y **write()** (genera el archivo de salida con el laberinto rotado) que se encargan de rotar el laberinto dado como argumento.

3.2.2. Stage2.cpp

Clase que contiene el método **main()**, donde se definen los parámetros y variables necesarias de inicio del programa. Además, se entrega como argumento el archivo "maze_in.pbm" para ser rotado en la clase **Maze()**.

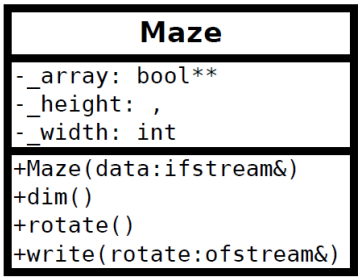


Figura 5: Diagrama UML Stage 2.

3.2.3. Resultados

Se observa en la Figura 6 en forma clara la rotación de 90^o a la izquierda del laberinto. Se denota que se hizo un tanto compleja la implementación y uso de arreglos multidimensionales en C++ (se utilizaron punteros dobles), ejercitando y leyendo más de este tópico en específico para desarrollar esta etapa.

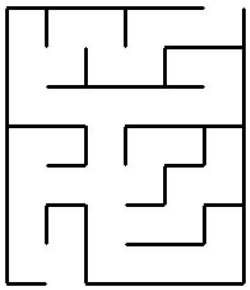


Figura 6: Imagen del Laberinto Rotado (maze_out.pbm).

3.3. Etapa 3: Robot con Sensores de Proximidad

Esta etapa implementa sensores como objetos mediante las clases "DistanceSensor.h" y "DistanceSensor.cpp", los cuales van empotrados al robot y miden la distancia en forma constante para detectar alguna pared en una dirección dada. Dicho sensor tiene un alcance finito y no posee movimiento propio, moviéndose en conjunto con el robot en la dirección que este determine según su algoritmo de movimiento.

Se destaca que se incluyen las clases descritas en forma previa, modificando cada una de manera conveniente con tal de lograr el objetivo planteado.

3.3.1. DistanceSensor.h / DistanceSensor.cpp

Clases nuevas que definen la estructura y los parámetros de los sensores a implementar. Además de los constructores necesarios, implementa los métodos **turnLeft()** (permite girar a la izquierda), **turnRight()** (permite girar a la derecha) y **senseWall()** (sensado de murallas en las cercanías de algún sensor).

3.3.2. Maze.h / Maze.cpp

Clases que permiten obtener la data del laberinto desde un archivo de extensión "pbm". Agrega los métodos **getArray()** (retorna el arreglo que tiene la data del laberinto), **getWidth()** (retorna el ancho del laberinto), **getHeight()** (retorna el alto del laberinto), **isThere_a_wall()** (identifica las murallas cercanas a cada sensor del robot) y **traceRoute()** (escribe la ruta que sigue el robot).

3.3.3. MyWorld.h / MyWorld.cpp

Clases que generan el mundo simulado donde se agregan los diferentes elementos de software para interactuar entre ellos. Respecto de la Stage1, se agregan los métodos **setRobot()** (incluye robot en el mundo), **setMaze()** (incluye laberinto en el mundo) y **isThere_a_wall()** (permite detectar las murallas cercanas).

3.3.4. Robot.h / Robot.cpp

Clases modificadas para agregar 3 sensores al robot (adelante, izquierda y derecha) que van empotrados a este. Además, modifica el constructor **Robot()** (inicializa nuevas variables) junto con los métodos **turnLeft()** y **turnRight()** para permitir que

los sensores giren en conjunto con el robot. Como último punto, agrega el método **moveRobot()** que define el algoritmo de movimiento del robot.

3.3.5. Vector2D.h / Vector2D.cpp

Clases que permiten el uso de un vector de dos dimensiones (plano XY). Respecto a la Stage1, agrega el método **getUnitary()** que calcula y retorna el vector unitario dadas las coordenadas (x,y) de algún vector que lo instancie.

3.3.6. Stage3.cpp

Clase que contiene el método **main()**, donde se definen los parámetros y variables necesarias de inicio del programa. Se inician los objetos de software y sus características propias para que la simulación funcione según lo esperado.

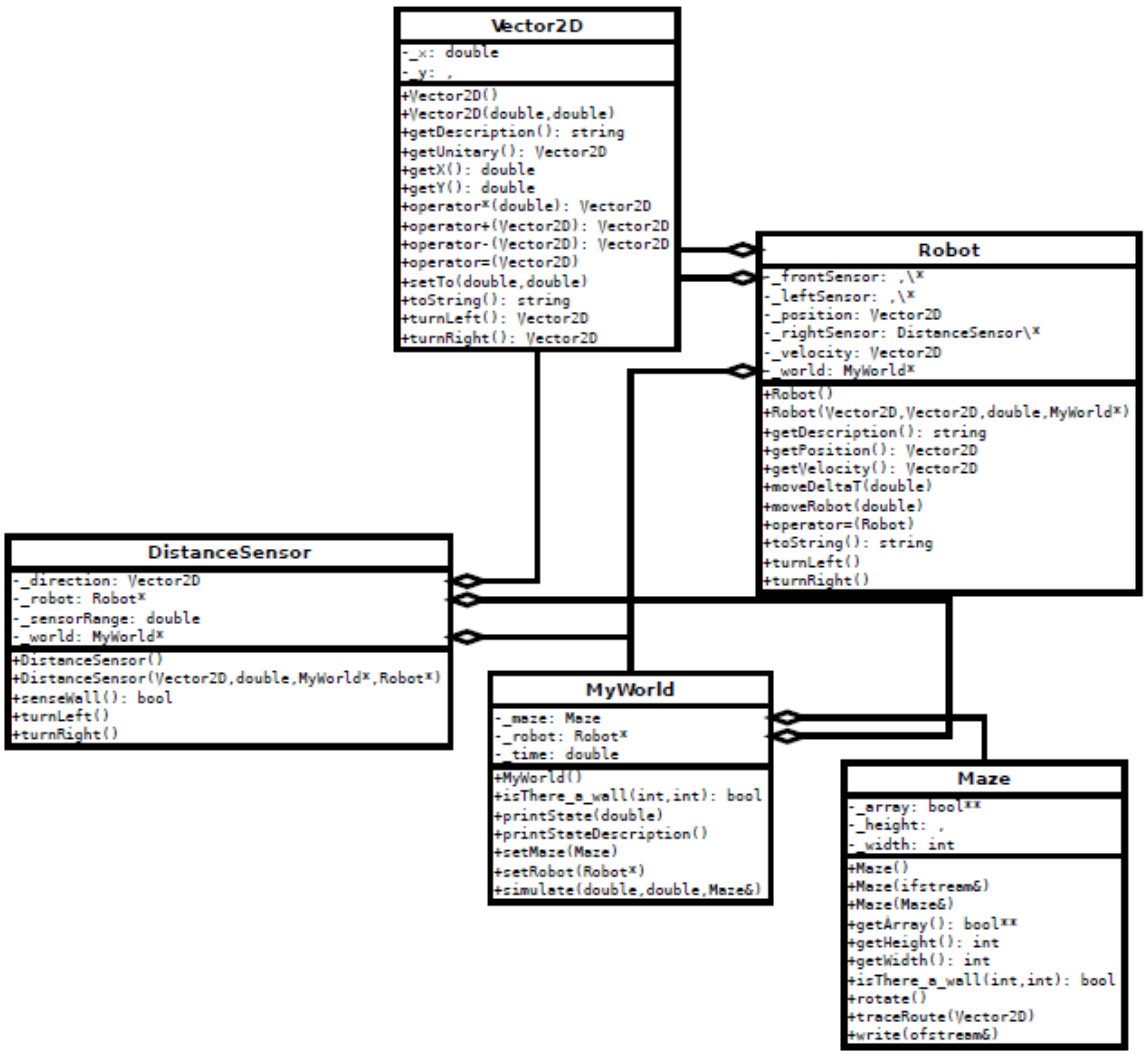


Figura 7: Diagrama UML Stage 3.

3.3.7. Resultados

Se aprecia en la Figura 8 que el robot es capaz de moverse por el laberinto en base al algoritmo definido, tal que sigue un parámetro de movimiento de **giro a la derecha** cada vez que el sensor delantero se encuentra frente a una muralla. Dado este hecho, es notorio que el robot equivoca los movimientos lógicos en relación a la bísqueda de la salida del laberinto (poner atención a la línea del recorrido).

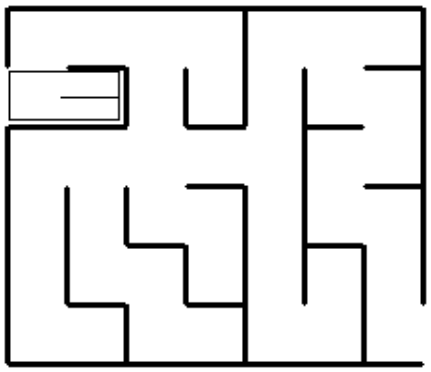


Figura 8: Recorrido del Robot por el Laberinto.

3.4. Etapa 4: Robot con Sensores y Navegante

Esta etapa define la clase "Pilot" que posee referencias a los tres sensores del robot (adelante, derecha e izquierda) y un método para fijar el curso del robot, donde la lógica a seguir es recorrer el laberinto pegado a la pared derecha. Además, se utiliza la clase "DocumentedMaze" que hereda de la clase "Maze" y permite obtener los datos desde el archivo pbm que se utilizan en la simulación.

Se destaca que esta etapa incluye las clases descritas en forma previa dado el desarrollo iterativo e incremental planteado en esta tarea.

3.4.1. DocumentedMaze.h / DocumentedMaze.cpp

Clases nuevas que se encargan de extraer la información desde el archivo pbm mediante el método **DocumentedMaze()** (heredado de "Maze"). En base a los datos obtenidos, se crean los métodos **setEntrance()** (setea posición de entrada), **setDirection()** (setea dirección de entrada), **setExit()** (setea salida del laberinto) y **setExitRadius()** (radio de salida del laberinto), además de los métodos **getVariable()** necesarios para obtener cada una de las variables desde otras clases.

3.4.2. Pilot.h / Pilot.cpp

Clases nuevas que definen el algoritmo de movimiento al interior del laberinto (pegado a la derecha). Se define el método **setCourse()** para implementar la secuencia de instrucciones necesarias que cumplan con el movimiento dado.

3.4.3. Robot.h / Robot.cpp

Clases modificadas respecto de la "Stage3" para instanciar el piloto. Se agrega el método **moveRobot()** que mueve un robot utilizando el algoritmo diseñado y setea el curso seguido para ser escrito en un archivo de salida de extensión pbm.

3.4.4. MyWorld.h / MyWorld.cpp

Clases modificadas respecto de la "Stage3" para agregar el término de la simulación al robot salir del laberinto. Se modifican los métodos **simulate()** (define condición de término) y **theFinal()** ("true" cuando se detecta la salida).

3.4.5. Stage4.cpp

Clase que contiene el método **main()**, donde se definen los parámetros y variables necesarias de inicio del programa. Se inician los objetos de software y sus características propias para que la simulación funcione según lo esperado.

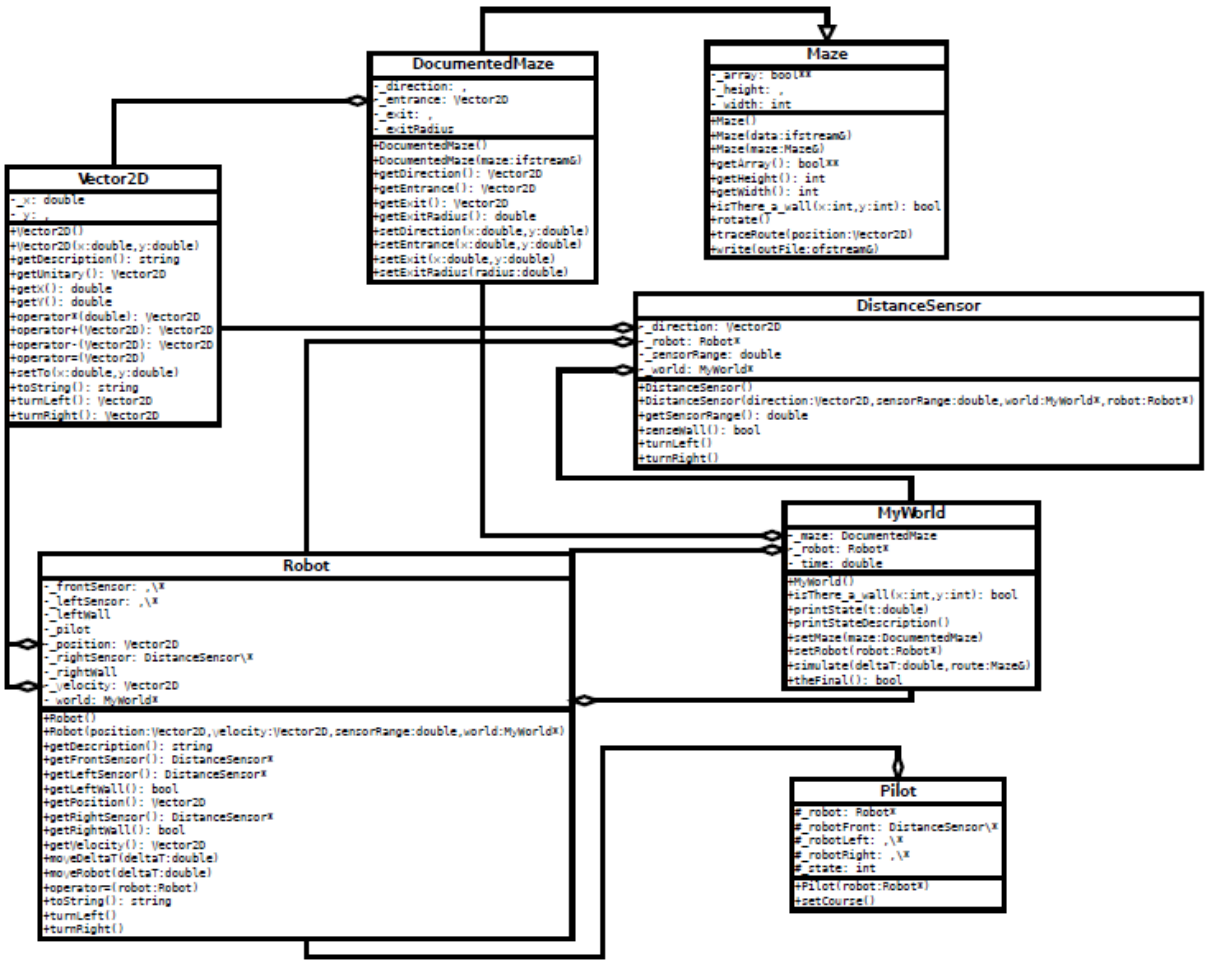


Figura 9: Diagrama UML Stage 4.

3.4.6. Resultados

Se observa en la Figura 10 el recorrido llevado a cabo por el robot al interior del laberinto, tal que cumple con la indicación de moverse pegado a la pared derecha. De esta manera, se concluye que el algoritmo diseñado funciona en forma óptima (note que el robot recorre más cerca la pared en ciertos sectores por efecto del cálculo de distancia realizado por los sensores frente a ciertos movimientos).

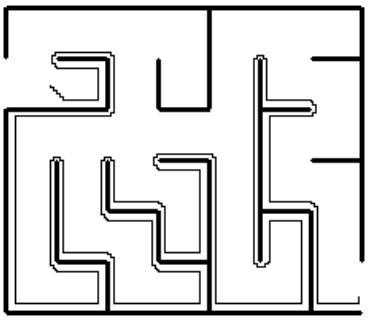


Figura 10: Robot sale del Laberinto usando Algoritmo Pegado a la Derecha.

3.5. Etapa 5: Corridas con 2 Estrategias de Navegación

Esta etapa genera la clase "MyPilot" que hereda de la clase "Pilot" para ocupar las cualidades de herencia para redefinir el método "setCourse()" y definir otra estrategia de navegación (algoritmo pegado a la pared izquierda).

Se destaca que en "Stage5" se generan dos robots, uno para cada estrategia de navegación para el mismo laberinto: un piloto se apega a la derecha y otro se apega a la izquierda (note que se incluyen las clases antes descritas).

3.5.1. MyPilot.h / MyPilot.cpp

Clases nuevas que heredan de "Pilot" y definen la nueva estrategia de navegación **setCourseLeft()** del laberinto (algoritmo pegado a la izquierda).

3.5.2. Robot.h / Robot.cpp

Clases modificadas respecto de la "Stage4" para instanciar los dos pilots disponibles a ocupar. Se agregan los métodos **moveRobot_right()** y **moveRobot_left()** que mueven un robot utilizando un algoritmo determinado (right para algoritmo pegado a pared derecha y left para algoritmo pegado a pared izquierda).

3.5.3. MyWorld.h / MyWorld.cpp

Clases modificadas respecto de la "Stage4" para agregar dos robots al mundo simulado. Se modifican los métodos **simulate()** (permite simulación de dos robots en simultáneo), **setRobot()** (setea la posición inicial de ambos robots) y **theFinal()** (término de simulación en base al robot que salga primero del laberinto).

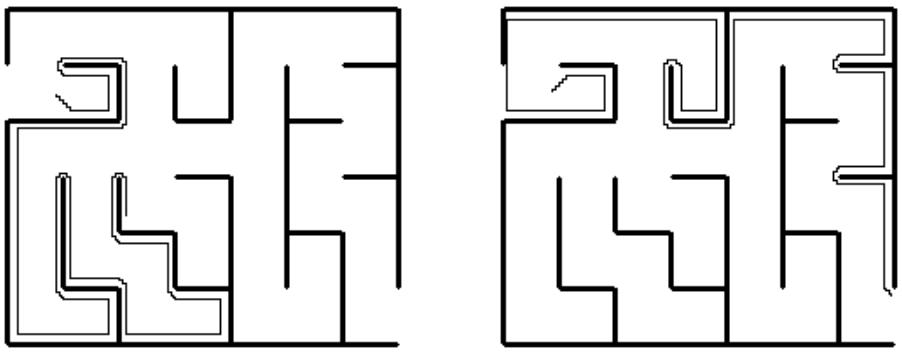
3.5.4. Stage5.cpp

Clase que contiene el método **main()**, donde se definen los parámetros y variables necesarias de inicio del programa. Se inician los objetos de software y sus características propias para que la simulación funcione según lo esperado.

3.5.5. Resultados

Se destaca en la Figura 11 la ruta seguida por los robots al interior del laberinto, tal que en la Figura 11a se aprecia el algoritmo de recorrido pegado a la derecha (note que este robot no alcanza a salir del laberinto antes de que otro robot salga) y en la Figura 11b se ve el algoritmo de recorrido a la izquierda (este finaliza el recorrido al interior del laberinto primero, marcando el término de la simulación).

Se denota que ambos robots parte de la misma posición inicial, por lo que se cumple el hecho de robots transparentes (no se molestan entre sí).



(a) Pegado a la Derecha. (b) Pegado a la Izquierda.

Figura 11: Algoritmos Diseñados para Recorrer el Laberinto.

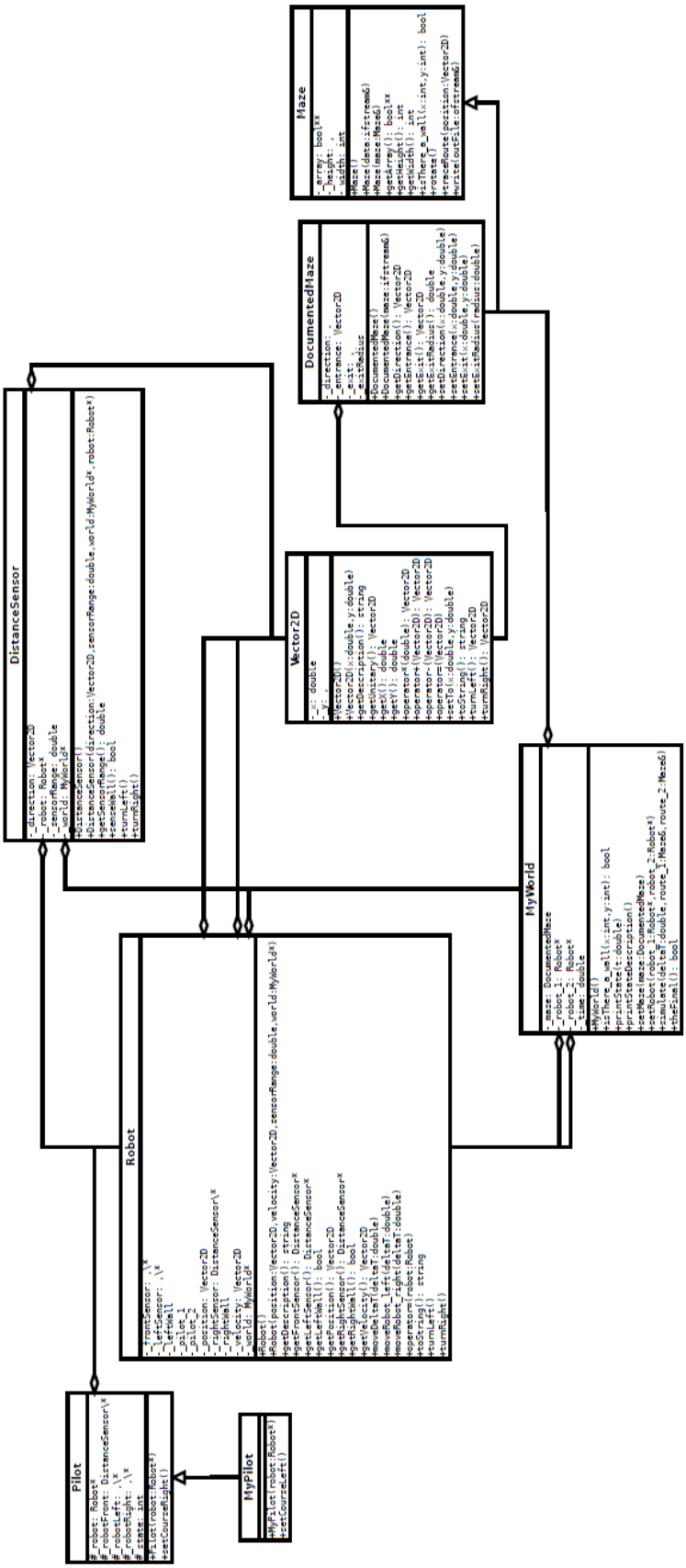


Figura 12: Diagrama UML Stage 5.

4. Problemas Presentados

4.1. Dependencia de Clases

Se descubrió la dependencia que se tiene entre los constructores, métodos, instancias y variables de cada clase, tal que no se pueden instanciar variables de una clase dada que no esté compilada en forma previa. Este problema limita la dependencia libre que se da en el lenguaje Java, tal que se tenía una mayor libertad.

Dado este suceso, un integrante generó una Stage3 diferente a la desarrollada en la tarea 1; mientras que otros dos integrantes implementaron clases, constructores, métodos e instancias similares a las dichas mediante el uso de punteros, siendo ambas soluciones completamente funcionales y cumpliendo los objetivos.

4.2. Uso de Punteros vs Paso por Referencia

Se denota que en la implementación de las Stages iniciales se utilizaron punteros para la definición de objetos tales como **Robot* _robot** en el caso de la Stage1. Este hecho produce que se apunte en memoria a la dirección de los objetos, sin posibilidad de cambiarlos y causando problemas para liberar espacio en memoria.

De esta forma, se cambiaron las definiciones de objetos mediante punteros utilizando **paso por referencia**, tal que se pasa la posición de memoria donde se encuentra el objeto, por lo que cada método puede saber su valor y modificarlo de cualquier manera, evitando conflictos por direccionamiento o acceso al estado.

4.3. Compilación en Aragorn

Una vez finalizada cada una de las etapa de la tarea, se fueron probando los archivos creados en Aragorn con ayuda de los archivos "Makefile" generados, esto con la finalidad de evitar problemas en la revisión de la tarea.

Dado este hecho, se presentaron problemas con la compilación de la Stage4 y Stage5, donde un integrante del equipo presentó un funcionamiento anómalo en Aragorn (robot corriendo en círculos) al ocupar los archivos del repositorio local respecto de los otros dos integrantes, tal que todos ocuparon los mismos archivos en la misma plataforma, presentando uno respuestas diferentes e inesperadas.

Este inconveniente no se pudo solucionar, explicitando esta problemática y esperando que a los ayudantes del ramo no les ocurra algo similar.

5. Conclusiones y Comentarios

Se concluye en la presente tarea que el lenguaje C++ resulta útil en el desarrollo de aplicaciones que requieran de la programación orientada a objetos, mas es un tanto complejo por temas de sintaxis, uso de punteros, dependencia de clases, entre otros; tal que esta es fundamental a la hora de simular soluciones de problemas reales en entornos de software que tengan la abstracción suficiente del mundo.

Se destaca que el manejo de conceptos tales como herencia, casteo y visibilidad fueron fundamentales para determinar las relaciones entre las diferentes clases, además de definir ciertas dependencias para la correcta implementación de cada etapa, tal que se ahorra la implementación reiterativa de métodos e instancias.

Se denota la importancia del desarrollo iterativo e incremental planteado para el desarrollo de esta tarea, táctica de gran utilidad para comprender que hace cada clase, método e instancia necesaria en el logro de la solución final.