

Proyecto Final  
Robot recolector de residuos  
Arquitectura de Comportamientos y Algoritmos

Guillermo Campelo  
Juan Ignacio Goñi  
Diego Nul

11 de agosto de 2010

**Resumen**

**Palabras clave:** *Robot, recolector, residuos, robótica, comportamientos, visión, MR-2FA, L298, FR304, HX5010, GP2D120, BC327, SRF05, CNY70, motor, servo, telémetro, daisy chain, RS232, subsumption, Webots, OpenCV, detección, objetos*

# Índice

<b>1. Arquitectura de Comportamientos</b>	<b>4</b>
1.1. Introducción . . . . .	4
1.2. Investigaciones previas . . . . .	5
1.2.1. Desarrollo de comportamientos no triviales en robots reales : Robot recolector de basura - Stefano Nolfi [4] . . . . .	5
1.2.2. Arquitectura de control para un Robot Autónomo Móvil - Neves And Oliveira [3] . . . . .	6
1.2.3. Path Planning usando Algoritmos Genéticos - Salvatore Candido [1] . . . . .	7
1.2.4. Navegación predictiva de un robot autónomo - Foka And Trahanias [2] . . . . .	7
1.2.5. Algoritmo de navegación y evitamiento de obstáculos en un entorno desconocido - Clark Et. al [5] . . . . .	8
1.3. Arquitectura propuesta . . . . .	10
1.4. Comportamientos e implementaciones . . . . .	12
1.4.1. Wandering . . . . .	13
1.4.1.1. Detalle del comportamiento . . . . .	13
1.4.1.2. Implementación del comportamiento . . . . .	13
1.4.2. Enfocar Basura . . . . .	15
1.4.2.1. Detalle del comportamiento . . . . .	15
1.4.2.2. Implementación del comportamiento . . . . .	17
1.4.3. Ir a Basura . . . . .	19
1.4.3.1. Detalle del comportamiento . . . . .	19
1.4.3.2. Implementación del comportamiento . . . . .	19
1.4.4. Recolectar Basura . . . . .	21
1.4.4.1. Detalle del comportamiento . . . . .	22
1.4.4.2. Implementación del comportamiento . . . . .	22
1.4.5. Ir a zona de descarga de basura . . . . .	22
1.4.5.1. Buscar línea . . . . .	23
1.4.5.1.1. Detalle del comportamiento . . . . .	23
1.4.5.1.2. Implementación del comportamiento . . . . .	23
1.4.5.2. Entrar a línea . . . . .	25
1.4.5.2.1. Detalle del comportamiento . . . . .	25
1.4.5.2.2. Implementación del comportamiento . . . . .	25
1.4.5.3. Seguir línea . . . . .	27
1.4.5.3.1. Detalle del comportamiento . . . . .	27
1.4.5.3.2. Implementación del comportamiento . . . . .	27
1.4.6. Descargar Basura . . . . .	28
1.4.6.1. Detalle del comportamiento . . . . .	28
1.4.6.2. Implementación del comportamiento . . . . .	28
1.4.7. Ir a base de recarga de batería . . . . .	28
1.4.8. Cargar Batería . . . . .	28
1.4.8.1. Detalle del comportamiento . . . . .	28
1.4.8.2. Implementación del comportamiento . . . . .	29
1.4.9. Evitar Obstáculos . . . . .	29
1.4.9.1. Detalle del comportamiento . . . . .	29
1.4.9.2. Implementación del comportamiento . . . . .	29

1.4.10.	Salir de situaciones no deseadas . . . . .	30
1.4.10.1.	Detalle del comportamiento . . . . .	30
1.4.10.2.	Implementación del comportamiento . . . . .	31
1.5.	Odometría . . . . .	32
1.5.1.	Problemas con la odometría . . . . .	33
1.6.	Interfaces con hardware y módulo de reconocimiento de objetos .	35
1.6.1.	Interfaz con hardware . . . . .	35
1.6.2.	Interfaz con módulo de reconocimiento de objetos usando Visión . . . . .	35
1.7.	Resultados obtenidos . . . . .	37
1.7.1.	Distribución y progreso de comportamientos . . . . .	38
1.7.2.	Tiempos en <i>caa</i> y <i>cai</i> de comportamientos . . . . .	41
1.7.3.	Transiciones entre comportamientos . . . . .	42
1.7.4.	Arena recorrida . . . . .	43
1.8.	Conclusión . . . . .	45
<b>A.</b>	<b>Implementación de arquitectura de software del controlador</b>	<b>48</b>
A.1.	Comportamientos . . . . .	50
A.2.	Dispositivos del robot . . . . .	52
A.3.	Reconocimiento de objetos . . . . .	54
<b>B.</b>	<b>Implementación del protocolo en PC</b>	<b>56</b>
B.1.	Packet Server . . . . .	56
B.2.	Packets . . . . .	56
B.3.	Handlers . . . . .	57
B.4.	Qué hacer en caso que se modifique el protocolo . . . . .	58

# 1. Arquitectura de Comportamientos

## 1.1. Introducción

Como mencionamos en las secciones anteriores, el objetivo de éste trabajo es desarrollar un robot autónomo y reactivo que recolecte basura en un entorno estructurado pero dinámico, debido a que la arena (lugar donde se mueve el robot) es transitado por personas y está al aire libre.

En esta sección de *Arquitecturas de Comportamientos* detallamos las acciones que debería llevar a cabo el robot y cómo organizar las mismas para cumplir la meta de recolectar basura y ser autónomo, es decir, decidir por sí mismo las acciones a realizar en cada momento y ser capaz de mantenerse cargado para poder continuar recolectando.

En la sección 1.2 analizamos papers relacionados con este desarrollo, desde la forma de organizar los comportamientos, la definición y composición de los mismos, hasta su implementación. En la sección 1.3 detallamos la arquitectura elegida para llevar a cabo y organizar los comportamientos elegidos, y las ventajas y desventajas de usar la misma. En la sección 1.4 detallamos uno por uno los comportamientos que elegimos para que posea el robot, y sus correspondientes implementaciones en *pseudo-código*. En la sección 1.5 explicamos que es la odometría, para qué sirve y qué ventajas y desventajas tiene usarla. También detallamos el test utilizado para mejorar la eficiencia de la misma. En la sección 1.6 describimos cómo estructuramos el controlador del robot para que podamos ir desarrollando y probando el mismo con un simulador y minimizar el trabajo al pasarlo a el robot físico. Los resultados de performance y eficiencia del controlador desarrollado los obtuvimos de la simulación, y los mostramos en la sección 1.7. Finalmente, en la sección 1.8 sacamos conclusiones de los resultados obtenidos y las dificultades encontradas a lo largo del desarrollo de éste proyecto final.

## 1.2. Investigaciones previas

A continuación presentamos trabajos realizados por otros autores relacionados en alguna forma con el nuestro. Primero damos una breve descripción del trabajo del otro autor y luego lo comparamos con nuestro trabajo, analizando similitudes y diferencias entre ambos.

### 1.2.1. Desarrollo de comportamientos no triviales en robots reales : Robot recolector de basura - Stefano Nolfi [4]

En este paper se muestra el uso de un Khepera con el módulo Gripper en una arena delimitada por paredes para la recolección de "basura". Dicho robot se puede ver en la figura 1. La base tiene una fuente de luz asociada y el objetivo

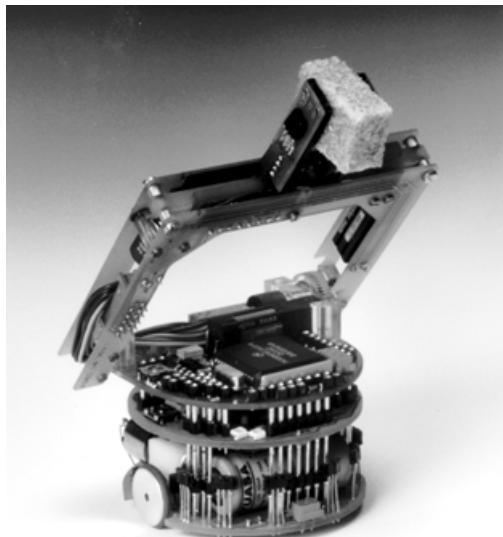


Figura 1: Robot Khepera con el módulo Gripper

del robot es llevar pequeños objetos a un lugar de depósito. Se utiliza *aprendizaje por refuerzo* para asociar las velocidades angulares de las ruedas a los diferentes tamaños de las "basuras". Para obtener el comportamiento deseado se hace uso de *algoritmos genéticos* y *redes neuronales*. Una vez obtenido el mismo mediante simulación en Webots, se lo probó en el Khepera físico. Comparando el trabajo de Stefano Nolfi con el nuestro podemos realizar las siguientes comparaciones:

- Existencia de un depósito: Ambos tienen un depósito en el cual dejar la basura recogida y una forma de identificarla: una fuente de luz usada por Nolfi y en nuestro trabajo, al llegar al final de una determinada línea de la arena.
- Autonomía: Ambos son autónomos en el sentido que no son manejados por un ente externo. En cuanto a la recarga de la batería, Nolfi no explicita

si es asistida o no; en nuestro caso, el robot se encarga de ir a la base de recarga una vez detectada la falta de energía.

- Método de Recolección: En el trabajo de Nolfi se utiliza el módulo “Gripper” agregado al Khepera. Este módulo simula un brazo con dos dedos. De ésta forma, la recolección se realiza juntándolos y la liberación se realiza separándolos. Nuestro trabajo se basa más en la actividad humana que en el comportamiento humano para realizar la recolección, ya que podría compararse con un recolector de basura que primero guarda lo encontrado usando una pala en su tacho y luego lo descarga en un depósito de mayor tamaño.
- Método de aprendizaje: En nuestro trabajo no utilizamos aprendizaje por refuerzo o algoritmos evolutivos. No es el caso de Nolfi, que utiliza ambas técnicas para evolucionar el comportamiento deseado.
- Robot Utilizado: Como mencionamos anteriormente, Nolfi utilizó un Khepera en la simulación. En nuestro trabajo utilizamos un E-puck para la simulación, una versión nueva del Khepera, pero sin el módulo Gripper.



Figura 2: Robot E-puck

### 1.2.2. Arquitectura de control para un Robot Autónomo Móvil - Neves And Oliveira [3]

Neves y Oliveira describen en su trabajo una arquitectura de control basada en comportamientos para un robot móvil en un ambiente dinámico, utilizando muchos aspectos de la arquitectura *Subsumption* (Ver sección 1.3) propuesta por *Brooks* y que usamos al realizar éste proyecto.

La arquitectura propuesta en el paper, o *Control System Architecture*, se basa

también en la teoría de *The Society of Mind*, escrita por Minsky, donde el sistema es visto como una sociedad de agentes, cada uno con una competencia particular y que colaboran entre ellos para ayudar a la sociedad a alcanzar su meta. La arquitectura está compuesta por tres niveles: un nivel reflexivo, uno reactivo, y otro cognitivo, aumentando la complejidad al igual que el orden en que fueron presentados.

El nivel reflexivo incluye aquellos comportamientos innatos, es decir, actúan directamente como estímulo-respuesta. El segundo nivel, el reactivo, está compuesto por agentes que responden rápidamente a los estímulos ya que requieren poco nivel de procesamiento. Finalmente, en el nivel cognitivo, se encuentran los agentes encargados de guiar y administrar los comportamientos reactivos de forma tal que el robot muestre un comportamiento orientado.

Aunque la arquitectura explicada es similar a la utilizada en nuestro trabajo, no utilizamos ésta organización de los comportamientos en capas. Sin embargo, podemos divisar que algunos de los comportamientos que propusimos corresponden a la primera capa de reflexión, como por ejemplo el evitamiento de obstáculos y otros podrían incluirse en la segunda capa de comportamientos reactivos, como sería deambular. Finalmente en la última capa estaría el reconocimiento de objetos debido a su gran demanda de procesamiento.

#### **1.2.3. Path Planning usando Algoritmos Genéticos - Salvatore Cандido [1]**

En este paper se describe la utilización de algoritmos genéticos para resolver el problema de Path Planning. Éste consiste en armar un plan, una secuencia de acciones de forma tal que a partir de un punto de origen se llegue al punto de destino, siguiendo ese plan. Debido a la componente dinámica de nuestro trabajo, siempre tratamos de mantener los comportamientos del robot lo más reactivos posibles, por lo que armar un plan no sería la mejor opción a utilizar. Por el contrario, el uso de algoritmos genéticos puede ser una herramienta que se podría llegar a usar en una futura continuación de nuestro trabajo y que no utilizamos por el tiempo que nos hubiese demandado.

#### **1.2.4. Navegación predictiva de un robot autónomo - Foka And Trahanias [2]**

La navegación predictiva nace como una posible solución al problema de un robot navegando en un ambiente con muchas personas y obstáculos, tal como lo es el ambiente en el cual navegará nuestro robot.

La forma en que es implementada por los autores es mediante un *POMDP*, es decir, un proceso de decisión de markov parcialmente observable. Cabe destacar estas dos últimas palabras, ya que indican la naturaleza del ambiente: hay incertidumbre o falta de información acerca de ciertas variables del entorno. En este caso, se utiliza el *POMDP* para manejar tanto la navegación del robot como el evitamiento de obstáculos, un punto en que se diferencia de lo que propusimos nosotros, que es tratar el *wandering* de forma separada del comportamiento de evitamiento de obstáculos. Ésto, en parte, se debió a causa de la arquitectura utilizada ya que desde ese punto de vista, ambos comportamientos tienen niveles de jerarquía muy diferentes como se puede ver en la figura 6.

### 1.2.5. Algoritmo de navegación y evitamiento de obstáculos en un entorno desconocido - Clark Et. al [5]

En este paper se presentan dos algoritmos complementarios para la navegación en ese tipo de entornos.

El primero consiste en la navegación y un mapeo del entorno que garantiza una cobertura completa de una arena cuyas ubicaciones de la paredes no se conocen *a priori*. Consiste básicamente en un seguimiento de las paredes complementado por una variación de *flood filling* para asegurarse la cobertura completa de la arena. El algoritmo completo se puede apreciar en la figura 3.

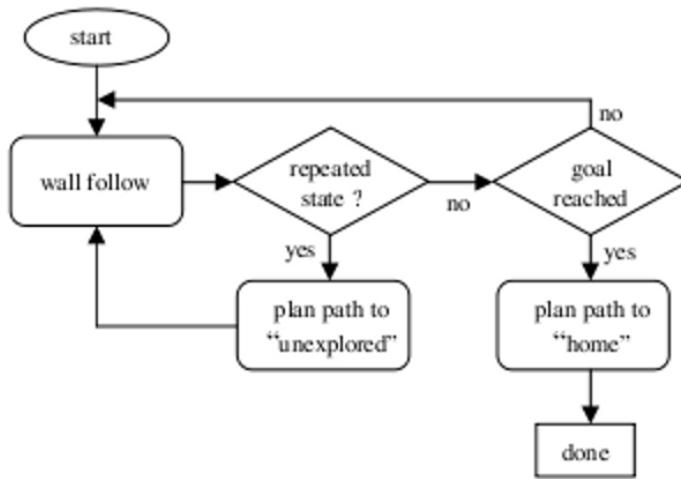


Figura 3: Algoritmo propuesto por Clark Et. al

Un autómata con aprendizaje estocástico es el algoritmo que complementa al primero, y su objetivo es el evitamiento de obstáculos. Para ésto, se utiliza un mecanismo de recompensa/castigo de forma tal que se adapten las probabilidades de las acciones a tomar.

En relación con nuestro trabajo, hay ciertas similitudes y diferencias, enumeradas a continuación:

- Ambos presentan un mecanismo de “seguimiento de”, en nuestro caso lo hacemos con líneas y basuras, en el paper se utiliza con paredes. Sin embargo, se utilizan para objetivos diferentes. Nosotros lo usamos para dirigirnos a la base ya que tratamos de mantener el conocimiento que el robot tiene sobre el mundo lo más acotado posible, o en el caso de las basuras, para recolectarlas. En el paper se utiliza el mecanismo de seguir las líneas para obtener un modelo del mundo, algo que puede llegar a servir mucho en ambientes no tan dinámicos como el nuestro, razón por la cual no elegimos implementarlo.
- También coincidimos con la existencia de un método para evitar obstáculos. En el paper se implementa como un autómata que va aprendiendo según los premios o castigos que recibe. En nuestro caso el comportamiento

to es puramente reactivo y reacciona en base a los valores de los sensores de distancia y no tiene memoria.

### 1.3. Arquitectura propuesta

Una arquitectura basada en comportamientos define la forma en que los mismos son especificados, desde su granularidad (qué tan complejo o simple es un comportamiento), la base para su especificación, el tipo de respuesta y la forma en que se coordinan.

La arquitectura que elegimos para desarrollar nuestro trabajo es *Subsumption*, desarrollada por Rodney Brooks a mediados de 1980. Esta arquitectura está basada en comportamientos puramente reactivos, rompiendo así con el esquema que estaba de moda en la época de *sensar-planear-actuar*. Algunos de los principios propuestos que tuvimos en cuenta durante el desarrollo de nuestro trabajo son:

- Un comportamiento complejo no es necesariamente el producto de un complejo sistema de control,
- El mundo es el mejor modelo de él mismo,
- La simplicidad es una virtud,
- Los sistemas deben ser construidos incrementalmente.

Cada comportamiento es un par estímulo-respuesta. Tal como mostramos en la figura 4, cada estímulo o respuesta puede ser inhibido o suprimida por otros comportamientos activos. Además cada comportamiento recibe una señal de reset, que lo devuelve a su estado original. El nombre *Subsumption* proviene

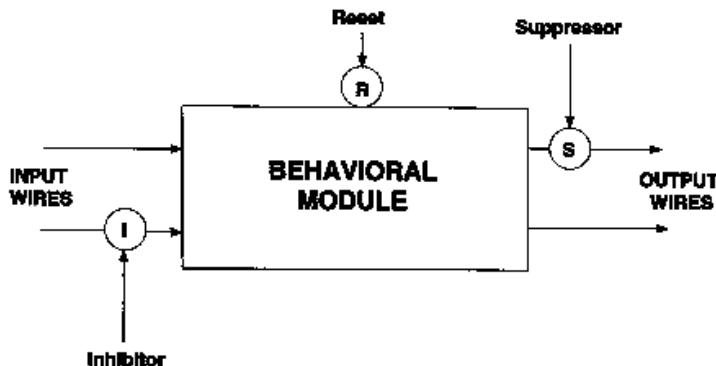


Figura 4: Esquema de comportamiento

de la forma en que los comportamientos son coordinados entre sí. Hay una jerarquía donde los comportamientos de la arquitectura tienen mayor o menor prioridad según su posición. Los comportamientos de los niveles inferiores no tienen conocimiento de los comportamientos de las capas superiores. Gracias a ésto, se puede plantear un diseño incremental, brindando flexibilidad, adaptación y paralelismo al desarrollo e implementación de los comportamientos.

La idea a seguir es que el mundo sea el principal medio de comunicación entre los comportamientos. Ésto se debe a que la respuesta de un comportamiento ante un estímulo resulta en un cambio en el mundo y, por lo tanto, en la relación del

robot con el mismo. De ésta manera, el robot en su próximo paso sensará otro estado del mundo.

El procedimiento básico para diseñar y desarrollar comportamientos para robots con esta arquitectura es sencillo:

1. Especificar cualitativamente la forma en que el robot responde al mundo, es decir, el comportamiento que realizará.
2. Descomponer la especificación como un conjunto de acciones disjuntas.
3. Determinar la granularidad del comportamiento, analizando en qué nivel de la jerarquía existente se encontrará y cuantas acciones disjuntas es necesario llevar a cabo para el cumplimiento de la tarea.

Un ejemplo de esta arquitectura se puede observar en la figura 5. En la misma hay 4 comportamientos: Homing, Pickup, Avoiding y Wandering. Las líneas que entran a cada comportamiento son los estímulos ante los cuales se activan y las salidas son las señales de respuesta correspondientes. La señal de respuesta de la arquitectura es la línea que sale por la derecha de la caja (Arquitectura) que contiene la relación entre los comportamientos. Puede verse como Homing inhibe la salida de Pickup ya que su salida entra al supresor (denotado con un círculo con una S dentro) de la salida de Pickup y por lo tanto, las salidas de los demás comportamientos, ya que su prioridad es mayor al resto. En el caso que no estén presentes los estímulos de Homing, Pickup y Avoiding, no hay inhibición en la salida de Wandering, y por lo tanto se lleva a cabo el comportamiento de Wandering.

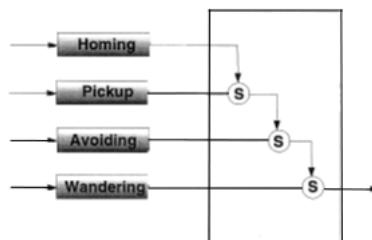


Figura 5: Ejemplo de la arquitectura Subsumption

## 1.4. Comportamientos e implementaciones

Una vez que decidimos utilizar la arquitectura explicada en la sección 1.3 para nuestro proyecto, tuvimos que analizar:

- Forma de implementación de la arquitectura en código,
- Comportamientos a realizar,
- Ordenes de inhibición y supresión entre los mismos,
- Forma de implementación de los mismos,
- Orden de implementación

Para implementar la arquitectura decidimos asignarle un *ID* numérico diferente a cada comportamiento. También tomamos la decisión de elegir como comportamiento activo en el instante  $t$ , aquel comportamiento que esté activo en ese instante y tenga mayor *ID*, suprimiendo así el resto de los comportamientos (con un *ID* menor) que podrían estar activos.

Como requerimiento de nuestro proyecto teníamos la realización de un robot autónomo que recolecte basura de su entorno dinámico pero estructuralmente estático. De aquí se infieren algunos de los comportamientos que debe tener el robot:

- *Recolectar basura* (1.4.4)
- *Recargar batería* (1.4.8): Por ser autónomo, debe poder ser capaz de recargarse solo para poder continuar con su actividad.
- *Wandering* (1.4.1): El robot, al ser autónomo, no es radio-controlado y debe poder recorrer el entorno por sí mismo.
- *Evitamiento de obstáculos* (1.4.9): Debido a la naturaleza dinámica del entorno, el robot debe ser capaz de navegar sin chocarse contra las paredes ni con las personas que circulan por el mismo.

El comportamiento de recolectar basura y el requerimiento de la autonomía llevan a su vez a la aparición de más comportamientos: *Descargar basura* (1.4.6) e *Ir hacia basura* (1.4.3).

En la figura 6 mostramos los comportamientos implementados y su orden de jerarquía. Se puede ver que hay más comportamientos de los detallados anteriormente ya que la forma en que implementamos los comportamientos básicos del robot requirió el desarrollo de otros auxiliares.

Primero implementamos *Wandering* debido, en una primera aproximación, a su sencillez. Luego implementamos *Evitamiento de obstáculos* para lograr que el robot pueda navegar sin problemas por el entorno. Como el comportamiento de recolectar e ir hacia la basura dependía del módulo de reconocimiento de objetos y el mismo estaba siendo desarrollado en paralelo, decidimos implementar el comportamiento de *Recargar batería* y *Descargar basura*. Una vez que tuvimos la primera implementación funcional del reconocimiento de objetos, procedimos a desarrollar *Ir hacia basura* y *Recolectar Basura*.

A continuación detallamos los comportamientos indicados en la figura 6, así como su implementación en *pseudo-código* y detalles tenidos en cuenta para su realización.

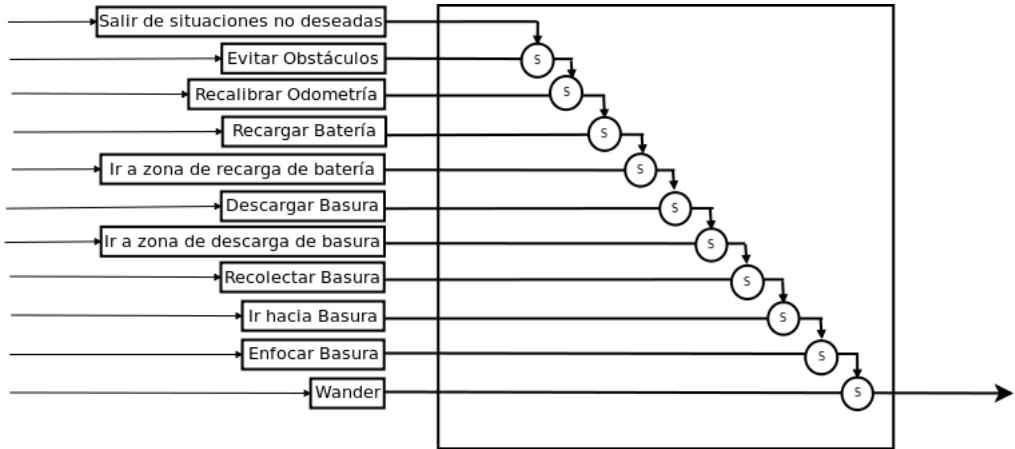


Figura 6: Arquitectura de comportamientos implementada

#### 1.4.1. Wandering

##### 1.4.1.1. Detalle del comportamiento

Por ser el comportamiento que menor jerarquía tiene (Ver figura 6), es el único comportamiento que está activo ante la ausencia de un estímulo, asegurandonos que siempre haya por lo menos un comportamiento activo.

En una primera aproximación de *Wandering*, sólo nos preocupamos por ir hacia adelante ya que eventualmente, el robot encuentra un obstáculo y realiza un giro cambiando su dirección.

Los resultados de la simulación nos indicaron que el robot no recorría ciertas zonas o las recorría después de un largo tiempo, lo que nos llevó a una segunda implementación. La misma lleva un seguimiento de los lugares que más recientemente recorrió el robot. Además, hicimos uso de la cámara para redefinir la idea de zona recorrida. Decimos que el robot recorrió cierta zona si:

- Estuvo físicamente en ella, o
- La zona fue alcanzada por la imagen que se ve en la cámara (Ver figura 8)

Ésto, en cierta forma, genera un modelo del mundo, un hecho que conflictúa con uno de los principios propuestos a seguir en la sección 1.3. Para minimizar el conflicto, decidimos mantener al mínimo la información almacenada para el funcionamiento del algoritmo, es decir, por cada zona de la arena sólo mantenemos el timestamp de la última vez que el robot la visitó.

##### 1.4.1.2. Implementación del comportamiento

La segunda implementación en *pseudo-código* es la siguiente:

```

por cada paso
    zona = pedir_zona_vista(camara)
    marcar_zona_como_vista(modelo_del_mundo, zona)
    ultima_zona_visitada = pedir_ultima_zona_visitada(modelo_del_mundo)

```

```

velocidades = calcular_velocidades_de_ruedas(ultima_zona_visitada)
poner_velocidades_en_ruedas(velocidades)

```

Para obtener la zona vista por la cámara, necesitamos de la altura  $C_h$  a la cual está ubicada la cámara en el robot, el campo de visión (de ahora en adelante *Field of View o FOV*) horizontal  $FOV_h$  o vertical  $FOV_v$  y el ángulo de inclinación de la cámara  $ac$ . Analizando la figura 7 podemos ver que:

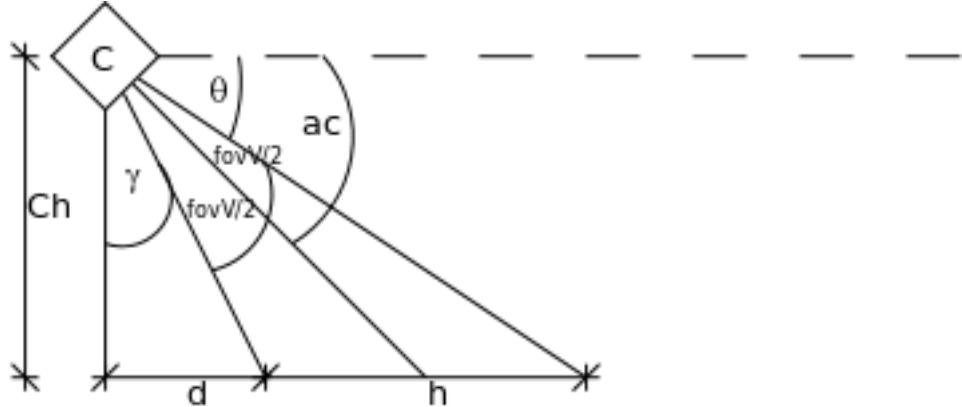


Figura 7: Diagrama de posición de la cámara

$$ac = \frac{FOV_v}{2} + \theta \quad (1)$$

$$\gamma + FOV_v + \theta = \frac{\pi}{2} \quad (2)$$

$$\tan(\gamma) = \frac{d}{C_h} \quad (3)$$

$$\tan(\gamma + FOV_v) = \frac{d + h}{C_h} \quad (4)$$

Organizando las ecuaciones, podemos deducir que:

$$\gamma = \frac{\pi}{2} + ac - \frac{FOV_v}{2} \quad (5)$$

$$d = \tan(\gamma) * C_h \quad (6)$$

$$d + h = \tan(\gamma + FOV_v) * C_h \quad (7)$$

Por lo que obtenemos el ángulo hasta el inicio de la imagen de la cámara  $\gamma$  y como consecuencia, la distancia  $d$  desde la posición de la cámara hasta el inicio de la imagen y  $d + h$ , la distancia desde la posición de la cámara hasta el final de la imagen. Usando estos datos, la posición del robot  $P$  y basándonos en la figura 8, podemos obtener los puntos  $A$ ,  $B$ ,  $C$  y  $D$  del trapezoide que determina la zona que ve la cámara.

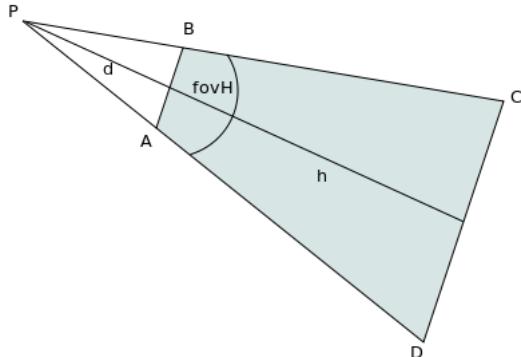


Figura 8: Zona vista por la cámara

#### 1.4.2. Enfocar Basura

Una vez que el módulo de detección de objetos reconoce algo como basura (ver sección ??), hay que elegir la forma en que se va a la basura. En la figura 9 mostramos una aproximación. Consiste en primero enfocar la basura de forma tal que la misma quede en el centro de la imagen de la cámara. De aquí surge el comportamiento *Enfocar Basura*.

Una segunda forma de lograr ésto (figura 10) no consiste en enfocar la basura, sino que se marca un arco hacia la misma. Ésto requiere que se seteen las velocidades correspondientes a las ruedas de forma tal que la trayectoria del robot describa dicho arco. Con esta alternativa no existiría el comportamiento que se está describiendo.

Dado que la primera aproximación es levemente más simple de implementar, y teniendo en cuenta el principio enunciado en la sección 1.3 “*La simplicidad es una virtud*”, elegimos implementarlo, a pesar de tener un posible inconveniente, como mostramos en la figura 11.

Cuando hay una basura en alguna esquina superior de la imagen de la cámara, y el robot gira sobre sí mismo para enfocarla, la basura puede llegar a perderse por el fondo de la imagen. Ésto se debe a que la distancia hacia dichas esquinas es mayor a la distancia hacia el centro del borde superior de la imagen (ver figura 8). Decimos que es un “possible” problema, ya que una vez que se pierde de vista la basura, el robot no enfocará más debido a que el estímulo desapareció, pero si luego se dirige hacia adelante (por la activación de algún otro comportamiento), la basura volverá a aparecer en la imagen, sin aprovechar la oportunidad de recogerla.

##### 1.4.2.1. Detalle del comportamiento

La primer aproximación se puede describir como:

- Si la basura está a la izquierda de la imagen, se debe girar hacia la izquierda
- Si la basura está a la derecha de la imagen, se debe girar hacia la derecha
- Si la basura está en el centro, ya está enfocada

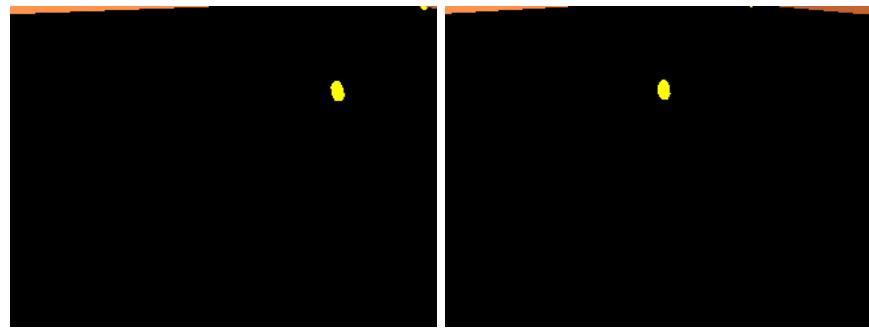


Figura 9: Primera aproximación de “Ir a la basura”. Inicialmente el objeto reconocido como basura se encuentra a un costado del eje vertical de la imagen. Luego el robot gira de forma tal que el objeto quede sobre el eje vertical.

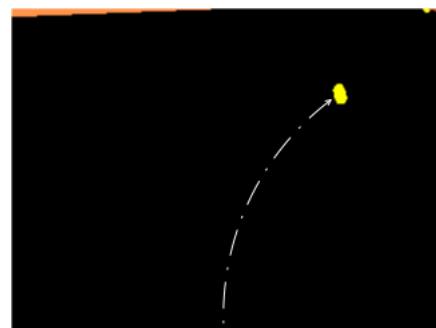


Figura 10: Segunda aproximación de “Ir a la basura”. Se traza un arco hacia el objetivo de forma tal que no sea necesario enfocar la basura.



Figura 11: Posible inconveniente con la primera aproximación de “Ir a la basura”. La basura se encuentra en una de las esquinas superiores de la imagen. El robot la enfoca, girando hacia la derecha y la basura desaparece de la imagen. En el caso que el robot se dirija hacia adelante, la basura volverá a aparecer en la imagen.

#### 1.4.2.2. Implementación del comportamiento

Para la implementación se sigue el siguiente *pseudo-código*:

```

por cada paso
    lista_de_basuras = obtener_lista_de_basuras(modulo_de_reconocimiento)
    basura_mas_cercana = elijo_basura_mas_cercana(lista_de_basuras)
    angulo_a_basura = obtener_angulo(basura_mas_cercana)
    velocidad = VELOCIDAD_BASE * (abs(angulo_a_basura) / (PI/2))
        + VELOCIDAD_BASE_MINIMA

    si angulo_a_basura < 0 entonces
        veloc_izq = -velocidad
        veloc_der = velocidad
    sino
        veloc_izq = velocidad
        veloc_der = -velocidad
    fin_si
    poner_velocidades_en_ruedas(veloc_izq, veloc_der)

```

Se puede ver que la velocidad de giro del robot es proporcional al módulo del ángulo que hay hacia la basura, logrando enfocar más rápido cuando el ángulo es mayor y tener mayor precisión cuando el ángulo es más chico, además de tener mayor rapidez de enfoque y precisión que si la velocidad de giro fuera constante.

Como observamos en el pseudo-código del cuadro 1.4.2.2, el algoritmo enfoca la basura que se encuentra más cerca del robot. Para ésto realiza una transformación en la cual obtiene la distancia al objeto según su coordenada ( $x, y$ ) en la imagen. En la figura 12 podemos apreciar mejor el área vista por la cámara, delimitada por el trapecio gris. Las líneas azules marcan los segmentos hacia las esquinas izquierda, derecha y centro inferior de la imagen. Las verdes indican lo mismo pero para la parte superior de la imagen. En rojo se pueden ver los triángulos formados hacia una basura a distancia  $db$ . Ésta, puede calcularse mediante la siguiente ecuación:

$$db = \sqrt{dx^2 + dy^2} \quad (8)$$

Dado que los valores de  $dx$  y  $dy$  no son conocidos, nos ayudamos con la figura 13. Descomponiendo la misma en la figura 14, podemos afirmar que:

$$dx = 2b \sin\left(\frac{\alpha}{2}\right) \quad (9)$$

donde  $b$ , a su vez, lo calculamos como:

$$b = \sqrt{dy^2 + Ch^2} \quad (10)$$

donde  $Ch$  representa la altura de la cámara y es un valor conocido.

Para el cálculo de  $dy$  tenemos que recurrir a la ecuación 7 donde reemplazamos  $FOV_v$  por  $\beta$  que es el ángulo vertical que forma la cámara con el objeto en cuestión. Éste se computa como  $\beta = kFOV_v$  donde  $k = \frac{dist_y}{C_{rh}}$  que se corresponden con la distancia en pixels sobre el eje  $y$  y la resolución vertical de

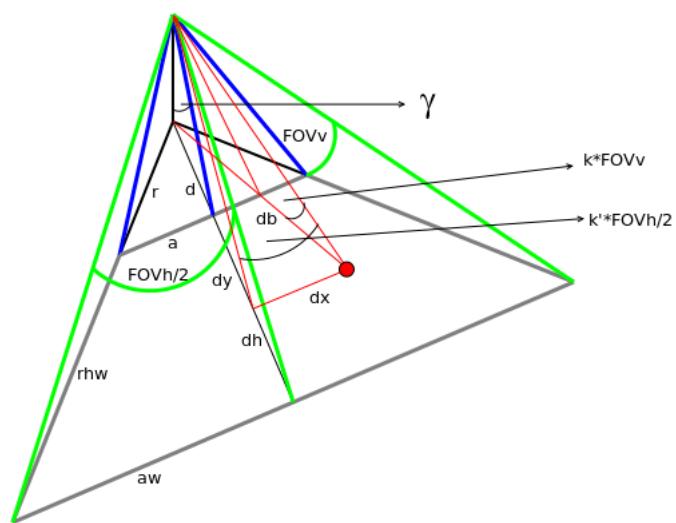


Figura 12: Área vista por la cámara y distancias y ángulos

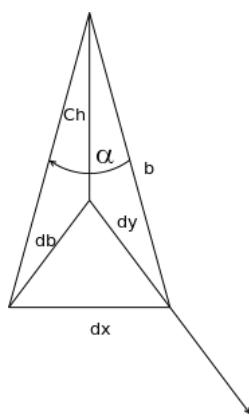


Figura 13: Caso particular de distancia hacia una basura a distancia  $db$

la imagen respectivamente. El cálculo de  $k$  puede apreciarse mejor en la figura 17.

Para el cómputo de la ecuación 9 también es necesario el valor de  $\alpha$  el cual computamos como  $\alpha = k' \frac{FOV_h}{2}$  donde en este caso  $k' = \frac{dist_x}{0.5C_{rh}}$  y  $FOV_h$  se corresponden con la distancia en pixeles sobre el eje  $x$  y el field of view horizontal de la cámara imagen. Habiendo obtenido estos valores, estamos en condiciones de calcular tanto la ecuación 10 como 9 y finalmente, utilizando estos resultados, reemplazamos en 8 y obtenemos la distancia hacia el objeto. En la figura 15 podemos observar los arcos de distancia hacia cada punto de la imagen.

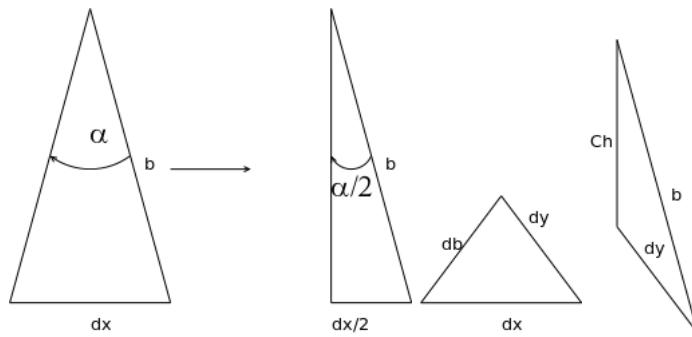


Figura 14: Descomposición de figura 13

### 1.4.3. Ir a Basura

Luego de la elección de la forma en que se resuelve el problema de encontrar una basura (ver figura 1.4.2), el comportamiento de ir a basura resulta trivial, ya que luego de ser enfocada, la basura está delante del robot y lo basta con ir hacia adelante.

#### 1.4.3.1. Detalle del comportamiento

El estímulo necesario para que este comportamiento esté presente, está dado por dos condiciones:

1. El método de reconocimiento de objetos encontró una basura
2. La basura se encuentra en un entorno del centro del eje vertical de la imagen obtenida de la cámara.

Para decidir si una basura se encuentra en un entorno del eje vertical, utilizamos un umbral  $\delta_f$  que determina el ángulo de abertura. Si una basura se encuentra en la zona, entonces se la toma como enfocada. En la figura 16 se pueden ver 2 valores distintos para el umbral.

#### 1.4.3.2. Implementación del comportamiento

por cada paso

```
distanzia = obtener_distancia_a_basura(modulo_de_reconocimiento)
coeff = (distanzia - DIST_MIN)/(DIST_MAX - DIST_MIN)
```

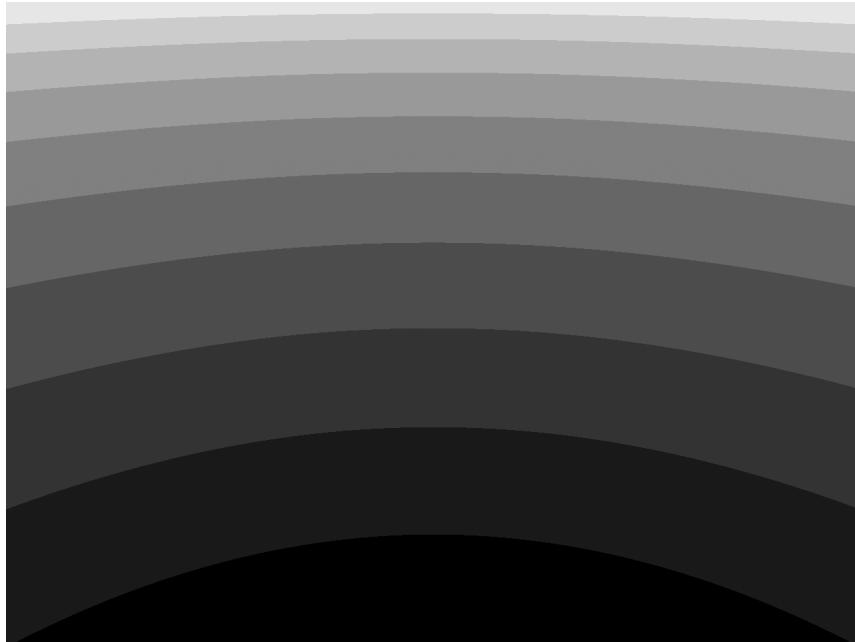


Figura 15: Distancias a pixeles en imagen según la variación de color. Los arcos marcan distancias iguales. Cuanto más clara la zona, más lejana está. Dentro de cada zona delimitada por dos arcos hay una variación de un color más claro a uno oscuro a medida que se sube en la imagen. Cuanto más claro, más cercano.

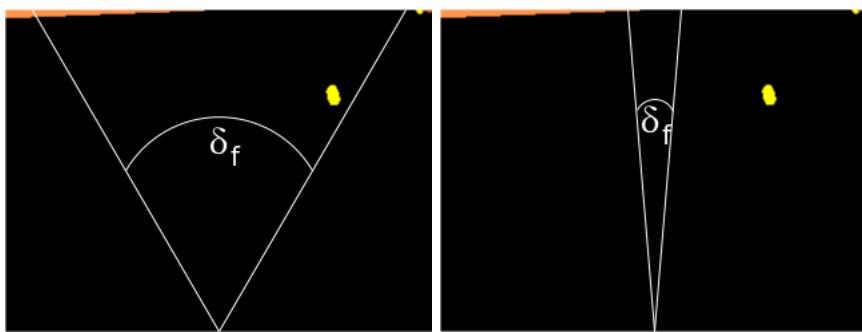


Figura 16: Distintos  $\delta_f$

```

veloc_der = VELOCIDAD_MIN*(1 - coeff) + coeff*VELOCIDAD_MAX
veloc_izq = veloc_der
poner_velocidades_en_ruedas(veloc_izq, veloc_der)

```

Al igual que en la implementación del comportamiento 1.4.2.2, las velocidades que se le otorgan a las ruedas dependen de la distancia hacia la basura, de forma tal que si una basura está muy lejos, la velocidad sea mayor y a medida que se va acercando, vaya disminuyendo linealmente.

Dado que la basura se encuentra en un entorno del eje Y en la imagen (Ver figura 17b) cometemos un error muy pequeño al estimar la distancia a la basura como si estuviera sobre el mismo (asumiendo que la coordenada X de la basura en la imagen es 0).

Como mostramos en las figuras 17 y 7, el ángulo vertical hacia el punto más alto de la imagen  $(0, C_{rh})$  es  $\gamma + FOV_v$  y hacia el más bajo,  $(0, 0)$ , el ángulo es  $\gamma$ . De las ecuaciones (6) y (7) sabemos las distancias a los puntos  $(0, 0)$  y  $(0, C_{rh})$  son  $(DIST\_MIN)$  y  $(DIST\_MAX)$  respectivamente. Entonces, para obtener la distancia  $dy$  hacia el punto  $(0, y)$  basta con calcular:

$$y_{angle} = FOV_v * \frac{y}{h} \quad (11)$$

$$dy = \tan(\gamma + y_{angle}) * C_h \quad (12)$$

donde  $y_{angle}$  es la proporción de  $FOV_v$  hacia  $y$ .

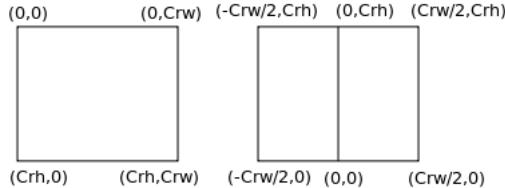


Figura 17: Conversión de coordenadas de imagen de cámara. Dado un pixel  $(y, x)$ , su nueva coordenada está dada por  $(x - Crw/2, Crh - y)$

#### 1.4.4. Recolectar Basura

Una vez que el robot llegó a estar posicionado para recolectar la basura deseada, el comportamiento de *recolectar basura* se activa. El mecanismo para lograr ésto fue cambiando a lo largo del desarrollo del proyecto.

En un principio pensamos en usar una rampa interna dentro del robot, de forma tal que la basura suba esa rampa para luego caer en un depósito interno. Por problemas con la simulación de este procedimiento, buscamos otro mecanismo.

El mecanismo que elegimos para usar en la simulación tiene estas componentes:

- El robot tiene un servo en su parte posterior (debajo de la cámara)
- Dos paredes delimitan el espacio a lo largo de la dirección que une el centro del robot con el servo anteriormente mencionado.

#### 1.4.4.1. Detalle del comportamiento

La activación de *recolectar basura* depende de 3 condiciones:

- Las dos condiciones impuestas para *ir a basura*
- La distancia a la basura elegida para ser recolectada debe ser menor a un umbral.

Aquí se puede observar que tan importantes son los comportamientos anteriores para que el robot logre recolectar la basura.

Es importante la elección del umbral: un valor muy chico, cercano a DIST\_MIN, puede llevar a que el comportamiento no se active porque la basura ya no está más en la imagen. Por otro lado, un valor muy grande, cercano a DIST\_MAX, causaría que el robot se disponga a recolectar una basura que está muy lejos y podría llegar a moverse por cuestiones propias del ambiente, llevando así a una innecesaria activación del comportamiento.

#### 1.4.4.2. Implementación del comportamiento

El *pseudo-código* de este comportamiento es el siguiente:

```
distanzia = obtener_distancia_a_basura(modulo_de_reconocimiento)
levantar(servo_delantero)
recorrer_distancia(distanzia)
cerrar(servo_delantero)
```

La distancia hacia la basura es obtenida de la misma forma que en la sección 1.4.3.2. Levantar y cerrar el servo consiste en setear su posición en  $\frac{\pi}{2}$  y 0 respectivamente. Para calcular la distancia recorrida utilizamos la distancia entre la posición del robot en el instante  $t$ ,  $P_r(t)$  y el instante  $t + 1$ ,  $P_r(t + 1)$ , ambas dadas por la odometría (Ver sección 1.5). Las diferentes etapas de la recolección de una basura las detallamos en la figura 18.

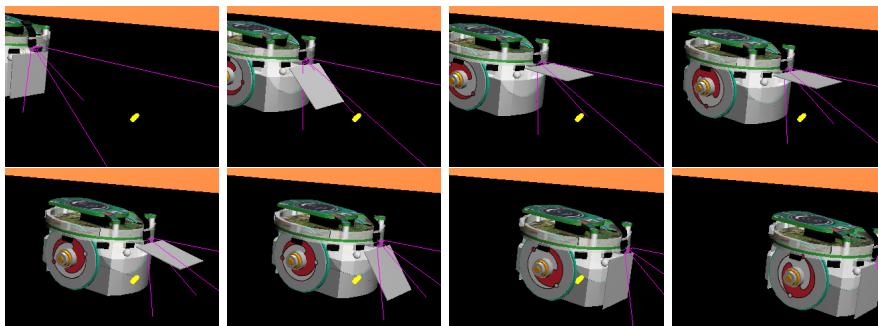


Figura 18: Etapas de recolección de basura

#### 1.4.5. Ir a zona de descarga de basura

Como la capacidad del depósito interno de basura del robot tiene un límite, surge como necesidad que el robot sea capaz de ir hacia una zona donde descargará la basura que contiene. Entonces, al llenarse el depósito surge el estímulo

necesario para la activación de este comportamiento.

Para ir hacia dicha zona decidimos imponerle una condición al entorno donde el robot actuará. Esta condición consiste en poner líneas de forma tal que si el robot la sigue, lo lleve a la zona de descarga.

Por lo tanto para dirigirse a la zona de descarga de basura, es necesario:

- Buscar la línea e ir a la misma
- Entrar a la línea de forma tal que el robot y la línea queden alineados
- Seguir la línea

Siguiendo con la idea que es mejor descomponer un comportamiento complejo en otros más simples, decidimos separar el comportamiento de *Ir a zona de descarga de basura* en 3 comportamientos más simples: *Buscar línea*, *Entrar a la línea* y finalmente *Seguir la línea*.

#### 1.4.5.1. Buscar línea

Inicialmente habíamos dispuesto las líneas de forma tal que sigan los límites de la arena. Entonces para buscar la línea tuvimos que calcular para cada una cuál es la distancia hacia la misma, luego elegir ir a la que menor distancia se encontraba. Ésta implementación, además de ser costosa, tenía un problema: a veces sucedía que al ir hacia una línea, la distancia hacia otra pasaba a ser más corta y ésta última podía estar más lejos de la zona de recarga que la primera.

Luego repensamos el problema y nos dimos cuenta que el objetivo era llegar a la zona de descarga, por lo que decidimos dejar sólo 2 las líneas que se encuentran a la misma distancia, como se puede apreciar en la figura 19. La zona de descarga se ubica cerca de donde se encuentra el cilindro de color verde.

Para distinguir si el robot está en una línea o no, utilizamos sensores de piso dispuestos como se muestra en la figura 20. Los sensores se encuentran a una distancia  $Fsd$  del centro del robot sobre el eje  $Y$  y aquellos de los costados a una distancia  $Fss$  del eje  $Y$ . La distancia desde el sensor del medio hacia el centro del robot es  $a$ , mientras que la distancia hacia un sensor del costado es  $Fsl = \sqrt{Fsd^2 + Fss^2}$ .

##### 1.4.5.1.1. Detalle del comportamiento

La decisión de dejar sólo dos líneas, acompañada por la elección de la ubicación de la zona de descarga, nos facilitó la composición de éste comportamiento.

Como mostramos en la figura 19, para ir a la línea se debemos girar hasta tener un ángulo de  $\frac{\pi}{2}$  y luego ir hacia adelante.

##### 1.4.5.1.2. Implementación del comportamiento

El *pseudo-código* de este comportamiento es sencillo, ya que no requiere cálculos extras:

```
por cada paso
    angulo_actual = obtener_angulo_actual(odometria)
    si esta_en_un_entorno_de(angulo_actual, PI/2) entonces
        si angulo_actual > PI/2 && angulo_actual < 3PI/2 entonces
```

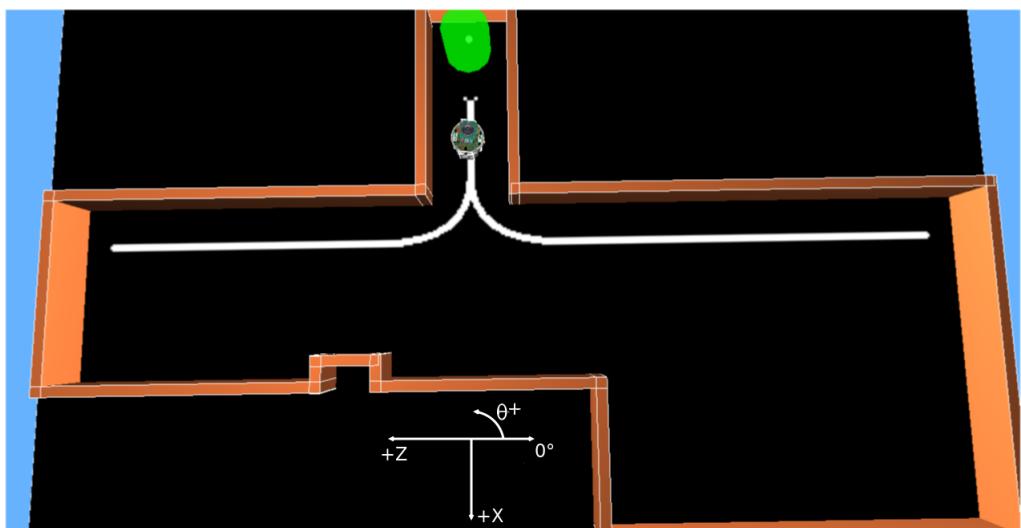


Figura 19: Arena de simulación y ejes de coordenadas

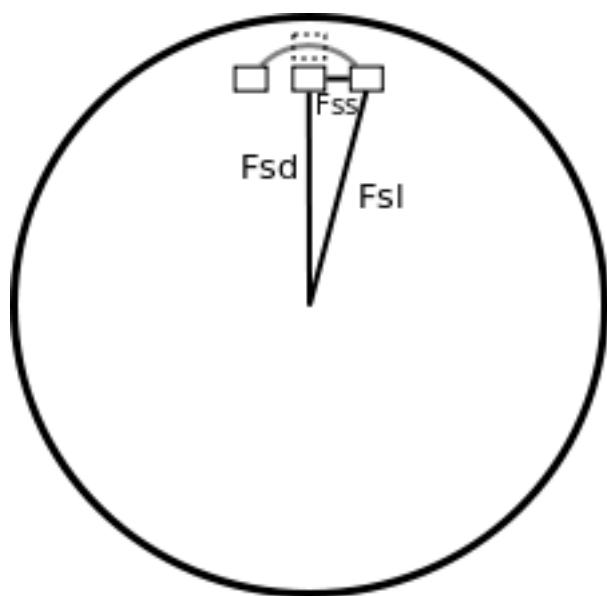


Figura 20: Disposición de sensores de piso

```

        giro_para_la_derecha
    sino
        giro_para_la_izquierda
    fin si
sino
    voy_hacia_linea
fin si

```

Hicimos la verificación de que el ángulo esté entre  $\frac{\pi}{2}$  y  $\frac{3\pi}{2}$  para evitar girar más de  $\pi$ , girando a la izquierda o a la derecha dependiendo cual sea el caso. Para ver esto más claramente, veamos que sucediese si no estuviera:

```

si esta_en_un_entorno_de(angulo_actual,PI/2) entonces
    giro_para_la_izquierda
sino

```

En el caso que el angulo actual sea  $\pi$ , el robot giraría un total de  $\frac{3\pi}{2}$  hasta llegar al ángulo destino  $\frac{\pi}{2}$ , cuando en realidad girando para el sentido contrario sólo tendría que girar  $\frac{\pi}{2}$ .

Se puede observar en el código que usamos datos calculados por la odometría, en este caso, la orientación actual del robot. El lector se podría preguntar: “Si la odometría tiene la posición actual del robot, ¿porqué no se usaron los datos de la misma para ir hacia la base?”. En principio esto requiere que el robot tenga conocimiento acerca de la ubicación de la base. Por otro lado, se hubiera tenido que utilizar algún algoritmo de *Path Planning* para realizar el recorrido, algo que no concuerda con la arquitectura elegida. Es importante destacar el uso de datos calculados por la odometría porque si la misma llega a tener un error grande, puede llevar a una activación errónea de comportamientos. Decimos que la odometría tiene un error grande cuando, debido a esa magnitud, el robot no toma las decisiones correctas debido al error en la información en la cual se basó. En la sección 1.5.1 se puede ver la incidencia de un error grande de la odometría en el comportamiento emergente del robot.

#### 1.4.5.2. Entrar a línea

##### 1.4.5.2.1. Detalle del comportamiento

Para seguir la línea es necesario que primero el robot esté posicionado sobre ella y alineado con la misma. También se necesita que la dirección del robot sea la que lo lleve hacia la zona de descarga. Una vez en la línea, el robot deberá girar dependiendo de que lado se encuentre la misma (Ver figura 19).

##### 1.4.5.2.2. Implementación del comportamiento

```

angulo_final = obtener_angulo_final(obtener_linea(odometria))
tita = atan(dist_sens_piso_X, dist_sens_piso_Y)
si (esta_en_la_linea(sensor_piso(DERECHA))) entonces
    tita = -tita
fin si
si (esta_en_la_linea(sensor_piso(MEDIO))) entonces
    tita = 0
fin si
si (tita != 0) entonces

```

```

        girar(tita)
fin si
distancia_a_recorrer = dist_sens_piso_Y;
si (tita != 0) entonces
    distancia_a_recorrer = sqrt(dist_sens_piso_X^2 + dist_sens_piso_Y^2)
fin si
recorrer(distancia_a_recorrer)
angulo_actual = obtener_angulo(odometria)
girar(normalizar(angulo_actual - angulo_final))

```

El primer giro del robot es para lograr que el segmento que une el sensor de piso del medio con el centro del robot quede ortogonal al segmento de línea. Como mostramos en la figura 21, en el caso (a) y (b) el robot girará de forma tal que llegue un caso parecido al que mostramos en la figura 22 ya posiblemente el sensor del medio no estará en la línea. Notar que si el sensor del medio está en la línea, como es el caso de la figura 21c, entonces el robot no gira, cualquiera sea el estado de los sensores de los costados. El ángulo que debe girar está dado por  $\theta = \arctan\left(\frac{F_{ss}}{F_{sd}}\right)$  (Ver figura 20) si debe girar hacia la izquierda, o  $-\theta$  en el otro caso.

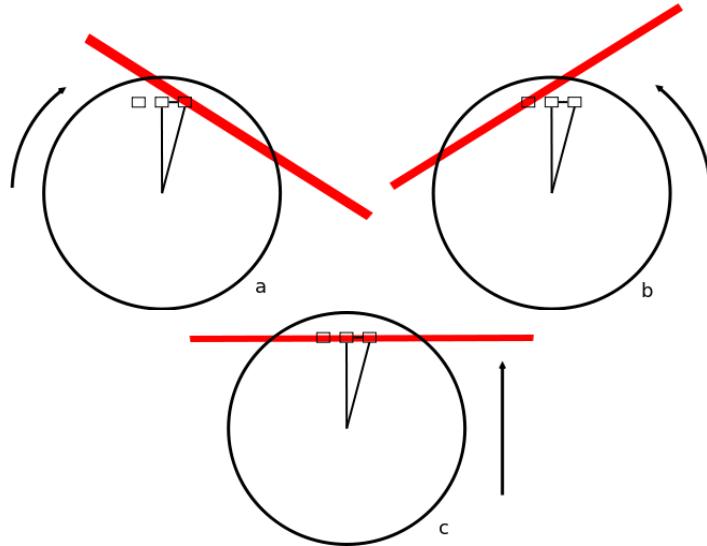


Figura 21: Posibles estados iniciales de “Entrar a la línea”. En el caso (a) el robot debe girar hacia la derecha. En el caso (b), el robot debe girar hacia la izquierda. En el caso (c) el robot no debe girar

Luego del posible giro para lograr que el sensor de piso del medio quede sobre la línea, se recorre una distancia de  $F_{sd}$  en el caso que el robot no haya girado anteriormente o  $F_{sl}$  en el caso que sí lo haya hecho. El motivo de realizar este trayecto es que el centro del robot quede sobre la línea, como muestra la figura 23. De ésta forma sólo queda girar nuevamente hacia el ángulo que se quiera. Si la línea es la izquierda entonces el robot deberá girar hasta que su orientación sea 0 o  $\pi$  en el caso que la línea sea la derecha.

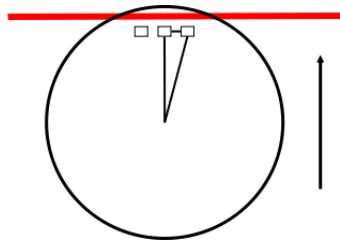


Figura 22: Posible estado final luego del primer giro del comportamiento “Entrar a la línea”

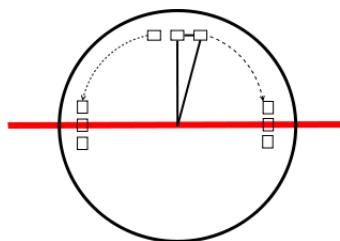


Figura 23: Centro del robot sobre la línea y posibles giros

#### 1.4.5.3. Seguir línea

Una vez que el robot está posicionado y con el sensor del medio sobre la línea, sólo basta con seguirla para llegar hacia la zona deseada. Este comportamiento es sencillo de desarrollar.

##### 1.4.5.3.1. Detalle del comportamiento

Para lograr que el robot siga la línea debemos tratar de mantener sólo el sensor del medio sobre la línea. Entonces, basta con analizar que se debe hacer en los siguientes casos:

1. El sensor de la derecha está sobre la línea
2. El sensor de la izquierda está sobre la línea

En el primer caso se debe lograr sacar el sensor de la derecha de la línea describiendo un pequeño arco hacia ese lado. El segundo caso es análogo: para sacar el sensor de la izquierda de la línea se debe seguir una trayectoria con un pequeño ángulo hacia la izquierda.

##### 1.4.5.3.2. Implementación del comportamiento

El *pseudo-código* es simple:

```
por cada paso
    veloc_izq = veloc_der = VELOC_SEGUIR_LINEA
    si (esta_en_la_linea(sensor_piso(IZQUIERDA))) entonces
        veloc_izq *= (1 - FACTOR_DE_GIRO)
```

```

    veloc_der *= (1 + FACTOR_DE_GIRO)
fin si
si (esta_en_la_linea(sensor_piso(DERECHA))) entonces
    veloc_izq *= (1 + FACTOR_DE_GIRO)
    veloc_der *= (1 - FACTOR_DE_GIRO)
fin si
poner_velocidades_en_ruedas(veloc_izq, veloc_der)

```

#### 1.4.6. Descargar Basura

Una vez que el robot logró llegar a la zona de descarga de basura, debe descargarla. Para hacer ésto decidimos ubicar un servo en la parte trasera del robot, con la misma idea del servo en el frente utilizado para recolectar.

##### 1.4.6.1. Detalle del comportamiento

Para descargar la basura el robot debe posicionarse de forma tal que la compuerta de descarga quede adyacente a la zona. Dado que para llegar a la misma el robot siguió la línea, al finalizar va a estar orientado con un ángulo en un entorno de  $\frac{\pi}{2}$  mirando la zona de descarga. Como el servo de descarga se encuentra en la parte trasera, deberá realizar un giro de aproximadamente  $\pi$  para luego poder descargar la basura.

##### 1.4.6.2. Implementación del comportamiento

```

posicionarse()
levantar(servo_trasero)
recorrerdistancia(ANCHO_ROBOT)
cerrar(servo_trasero)

```

#### 1.4.7. Ir a base de recarga de batería

Ayudados por la elección que tomamos de poner la zona de descarga de basura muy cercana a la zona donde se recarga la batería, decidimos utilizar la misma estrategia de seguir la línea para llegar hacia la misma.

La diferencia entre ambos casos es el estímulo ante el cual se activan. En el caso de ir a la zona de descarga, el estímulo proviene del sensor del depósito interno de basura que indica que el mismo está lleno. En el caso de ir a la base de recarga de batería, el estímulo para la activación depende de los valores de dos sensores de batería que indican batería baja:

- El sensor de la batería del robot, de donde se alimentan los sensores, actuadores y motores.
- El sensor de la computadora que corre el controlador.

#### 1.4.8. Cargar Batería

##### 1.4.8.1. Detalle del comportamiento

Una vez que el robot logró llegar a la zona de recarga de batería, debe recargarse. Para ésto, debe posicionarse para lograr ubicarse el cargador y esperar

hasta que la batería alcance un valor de carga tal que le permita seguir buscando basura sin problemas de batería.

#### 1.4.8.2. Implementación del comportamiento

```

posicionarse()
mientras(bateria_no_llena(BATERIA_ROBOT)
          o bateria_no_llena(BATERIA_PC)) hacer

    esperar(time_step)
fin mientras

```

#### 1.4.9. Evitar Obstáculos

*Evitar obstáculos* es uno de los comportamientos con mayor jerarquía en la arquitectura que elegimos. Su nivel se debe a la importancia que tiene en un ambiente estructurado pero dinámico como es el nuestro. El objetivo del robot es facilitar una tarea sin entorpecer el tránsito de personas.

##### 1.4.9.1. Detalle del comportamiento

Para lograr tener conocimiento sobre la proximidad de un obstáculo, utilizamos sensores de proximidad explicados en [??](#). Dependiendo de la proximidad indicada por un determinado sensor, el robot deberá alejarse de ese lado para evitar un posible choque. La activación del comportamiento dependerá entonces del valor sensado tal que la distancia hacia un obstáculo lleve al robot a una colisión o le imposibilite moverse.

##### 1.4.9.2. Implementación del comportamiento

La implementación de este comportamiento la hicimos utilizando una red neuronal sin capas ocultas (ver figura 24) usando los sensores de distancia como entradas y 2 neuronas de salida indicando los valores a ser seteados en los motores de las ruedas.

Debemos notar que hay conexiones tanto inhibitorias como excitatorias. Como es de esperarse, ambos sensores traseros (4 y 5) exitan a ambos motores. Distinto es el caso de los sensores del lado izquierdo (5, 6 y 7) que exitan el motor ubicado de su lado e inhiben el motor del lado opuesto, de forma tal que el robot gire para el lado opuesto de la ubicación de los sensores. La misma idea se sigue con los sensores (0, 1 y 2) ubicados en el costado derecho del robot.

Luego de entrenar la red, obtuvimos los siguientes valores para los pesos de la misma:

Rueda	$W_0$	$W_1$	$W_2$	$W_3$	$W_4$	$W_5$	$W_6$	$W_7$
Izquierda	-0.9	-0.85	-0.2	0.6	0.5	0.35	0.8	0.6
Derecha	0.9	0.85	0.2	0.6	0.5	-0.35	-0.8	-0.6

Cuadro 1: Asignación de pesos para evitar obstáculos

Se puede ver que los pesos que influyen en el motor de una rueda influyen

con igual fuerza pero distinto signo en el motor de la rueda contraria.

Este comportamiento, en *pseudo-código* puede verse como:

```
por cada paso
    veloc_izq = suma(coeficiente(RUEDA_IQZ, SENSOR_I) * VALOR(SENSOR_I))
    veloc_izq *= FACTOR_DE_INCIDENCIA
    veloc_der = suma(coeficiente(RUEDA_DER, SENSOR_I) * VALOR(SENSOR_I))
    veloc_der *= FACTOR_DE_INCIDENCIA
    poner_velocidades_en_ruedas(veloc_izq, veloc_der)
```

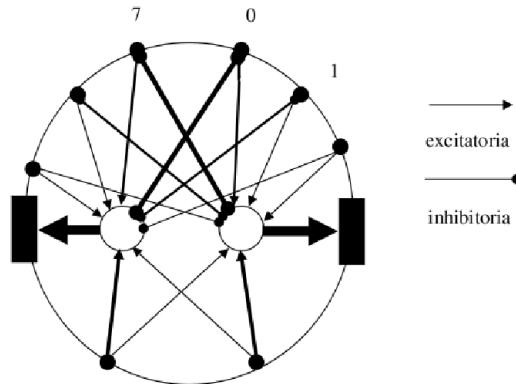


Figura 24: Red neuronal entre los sensores de distancia y los motores

#### 1.4.10. Salir de situaciones no deseadas

*Salir de situaciones no deseadas* surgió como un comportamiento para ayudar al objetivo de crear un robot autónomo. Con las sucesivas simulaciones observamos que había situaciones donde corría peligro la autonomía del robot. Un ejemplo de estas situaciones es el caso donde dos comportamientos se “activan mutuamente”.

Para ver esto más claramente, supongamos dos comportamientos *A* y *B* con nivel en la jerarquía  $N(A)$  y  $N(B)$ , siendo  $N(A) > N(B)$ . Si la respuesta a un estímulo de *A* lleva a la activación de *B* y a la desactivación de *A* y luego la respuesta de *B* lleva a la activación de *A*, podría llegar a entrarse en un ciclo si es que esta situación se da por un tiempo prolongado o indeterminado. El mayor peligro para la autonomía ocurre cuando *A* es el comportamiento de *evitar obstáculos* y *B* es el comportamiento de *ir a zona de recarga de batería* ya que el robot podría terminar quedándose sin batería en ese ciclo.

##### 1.4.10.1. Detalle del comportamiento

Las situaciones no deseadas se dan la mayoría de los casos cuando un comportamiento hace girar al robot hacia un lado y el otro comportamiento hacia el lado contrario, aproximadamente en la misma magnitud. Ésto quiere decir que la posición del robot se mantiene alrededor de un punto por un período prolongado de tiempo, por lo que decidimos tomar este hecho como estímulo para la activación de este comportamiento.

La respuesta del comportamiento es, girar un ángulo que cambie la dirección del robot y además que la suma de ese ángulo repetidas veces no sea periódica o lo sea luego de una gran cantidad de giros como por ejemplo, un valor de  $\frac{\pi}{4} + 0.1$ . Esta última condición se pide por el siguiente escenario:

- Los comportamientos *A* y *B* se “activan mutuamente” cuando la orientación del robot es 0.
- Los comportamientos *C* y *D* se “activan mutuamente” cuando la orientación del robot es  $\pi$ .
- La respuesta de *salir de situaciones no deseadas* es girar un ángulo  $\pi$ .

#### 1.4.10.2. Implementación del comportamiento

```
angulo_actual = obtener_angulo_actual(odometria)
nuevo_angulo = angulo_actual + ANGULO_A_SUMAR
girar(nuevo_angulo)
```

## 1.5. Odometría

Para saber donde se encuentra el robot en cierto momento utilizamos odometría. Esta técnica se basa en la medición del encoder en cuentas realizadas por cada motor para obtener el desplazamiento realizado por la rueda asociada.

A diferencia de los métodos de posicionamiento absoluto, la odometría da una estimación del desplazamiento *local* a la ubicación anterior del robot, por lo cual *un error en una estimación se propaga* hacia las siguientes estimaciones.

Para calcular la posición  $P_n$  en el instante de tiempo n y la orientación  $O_n$  en base a la posición  $P_{n-1}$  en el instante (n-1) y la correspondiente orientación  $O_{n-1}$ , usamos las siguientes fórmulas:

$$d_l = \frac{(e_l(n) - e_l(n-1)) * R_l}{EncRes} \quad (13)$$

$$d_r = \frac{(e_r(n) - e_r(n-1)) * R_r}{EncRes} \quad (14)$$

$$lc = \frac{d_r + d_l}{2} \quad (15)$$

$$P_n = P_{n-1} + (lc * \cos O_{n-1}, lc * \sin O_{n-1}) \quad (16)$$

$$O_n = O_{n-1} + \frac{d_r - d_l}{dbw} \quad (17)$$

donde  $d_l$  y  $d_r$  son las distancias recorridas por las ruedas izquierda y derecha respectivamente. Ambas son calculadas teniendo en cuenta los valores anteriores y actuales de los encoders  $e_i(n)$ , el radio de la rueda  $R_i$ , la resolución del encoder  $EncRes$  y es la distancia entre ruedas  $dbw$ .

Los errores que influyen en el cálculo de la odometría pueden ser de dos tipos:

- Sistemáticos: Son aquellos que pueden ser corregidos o tenidos en cuenta para disminuir el error.
- No sistemáticos: Son aquellos que pueden intentarse corregir pero *no* eliminar.

Decidimos tener en cuenta dos errores sistemáticos para disminuir el error en la odometría :

- Incertezza sobre la distancia entre las ruedas ( $dbw$ )
- Ruedas con radios diferentes ( $R_l$  y  $R_r$ )

Tuvimos en cuenta estos errores dado que son los que más contribuyen al error acumulado a lo largo del trayecto del robot. Para corregir estas fuentes de error, utilizamos el *test del camino bidireccional describiendo un cuadrado* (UMBmark)<sup>1</sup>. Dado que en el momento de realizar el test no teníamos un robot físico, decidimos realizar el test sobre un e-puck simulado en Webots. Así fuimos obteniendo los valores de corrección para los diámetros de las ruedas  $c_i$  y la corrección sobre la distancia entre las ruedas  $c_{dbw}$  hasta que el error sistemático máximo dejara de disminuir. En la figura 25 mostramos cómo disminuye el error a lo largo de las iteraciones.

---

<sup>1</sup><http://www-personal.umich.edu/~johannb/Papers/umbmark.pdf>

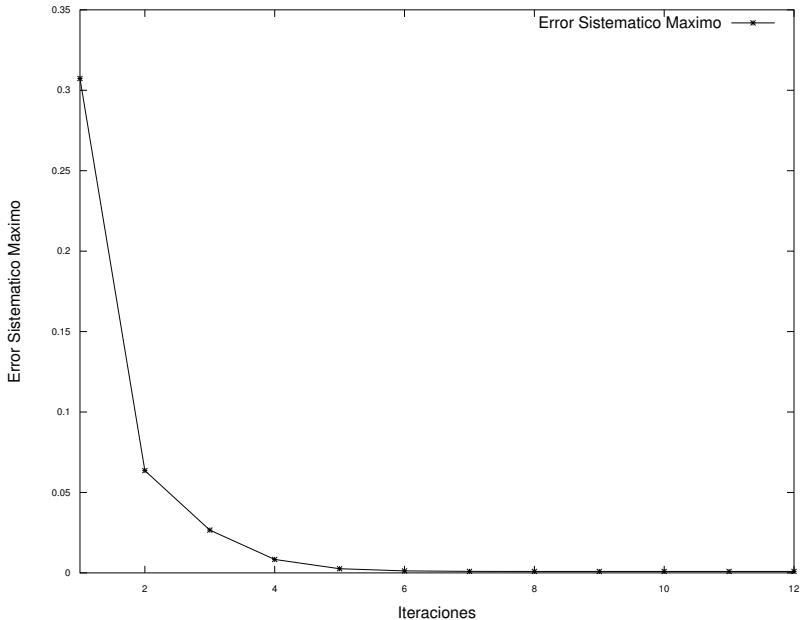


Figura 25: Error Sistemático Máximo a lo largo de las iteraciones

En la última iteración obtuvimos los coeficientes de corrección  $c_l = 0.99998703$ ,  $c_r = 1.00001297$  y  $c_{dbw} = 1.092171094$  para ruedas con radio  $R_l = R_r = 0.205$ , una distancia entre ruedas de 0.052 y una resolución de encoder  $EncRes = 159.23$ .

### 1.5.1. Problemas con la odometría

En las simulaciones previas a la presentada en la sección 1.7 observamos que el comportamiento emergente del robot no era el esperado. Viendo más en detalle, el robot tomaba decisiones erróneas en los comportamientos que utilizaban la odometría. Después de una extensiva revisión de código, llegamos a la conclusión que el mismo no tenía problemas.

Fue entonces que decidimos agregar un GPS (con error mínimo y que luego eliminamos) al e-puck en la simulación con Webots, con el objetivo de obtener un gráfico que indique el error en la odometría a lo largo de la simulación. El resultado esperado era que la odometría inicie con un error cercano a 0 y vaya creciendo constantemente a medida que avanzaba la simulación. El resultado no fue el esperado pero si iluminativo: el crecimiento era en partes constantes, pero había momentos en los cuales crecía abruptamente.

Decidimos entonces simular nuevamente y ver que comportamiento estaba activo en dichos momentos. Nos sorprendió ver que en esos momentos el comportamiento activo era *Evitar obstáculos* ya que era uno de los más simples y de los que más teníamos confianza que no haya un error de lógica de código. Nuevamente revisamos el código, sin encontrar problemas.

Ésto nos llevó a pensar qué relación había entre *Evitar obstáculos* y la odometría. El comportamiento setea velocidades en las ruedas y la odometría utiliza los valores de los encoders de las ruedas. Revisando las ecuaciones 13 y 14 nos dimos cuenta de lo siguiente: dado que las correcciones  $c_l$  y  $c_r$  de los radios  $R_l$  y  $R_r$  de las ruedas minimizan el error pero no lo eliminan, el error de las ecuaciones aumenta proporcionalmente a la diferencia entre  $e_i(n)$  y  $e_i(n - 1)$ . Y ésto sucedía cuando se activaba el evitamiento de obstáculos ya que daba velocidades a las ruedas de un orden de magnitud mayor a los que tenían anteriormente.

Finalmente, teníamos que buscar una solución a éste problema, ya sabiendo la verdadera causa del mismo. Pensamos en dos opciones:

1. Disminuir las velocidades que *Evitar obstáculos* asignada a las ruedas, dividiéndolas por una cte.
2. Hacer que cada vez que se setee una velocidad, se haga un cálculo con la velocidad seteada anteriormente, de forma tal que no haya saltos de órdenes de magnitud.

Por el tiempo que disponíamos en ese momento y por simplicidad y rapidez de implementación, elegimos la primer opción. Cabe aclarar que la segunda opción es más abarcativa y robusta, ya que sirve para cualquier comportamiento. Sin embargo, es invasiva en parte ya que puede que el desarrollador quiera, por algún motivo, que haya ese tipo de variaciones y no va a poder lograrlo.

## **1.6. Interfaces con hardware y módulo de reconocimiento de objetos**

Decidimos hacer que el controlador encargado de realizar los comportamientos sea independiente de la forma con la que se implemente el hardware y el reconocimiento de los objetos. Para ésto definimos interfaces de forma tal que el controlador las mismas y tanto el hardware como el módulo de reconocimiento de objetos puedan cambiar su implementación pero brindando siempre la información que necesita el controlador para poder realizar los comportamientos. Esta decisión nos posibilitó realizar un controlador que sea capaz de ser ejecutado tanto en Webots, donde se hizo el desarrollo, como en la base real del robot. Cabe aclarar que el traspaso de la simulación a la realidad no es instantánea, ya que hay que calibrar los dispositivos y realizar nuevamente la odometría, entre otras cosas, pero el trabajo demandado es mucho menor ya que una corrección en la lógica de los comportamientos se puede observar tanto en un simulador como en la realidad.

### **1.6.1. Interfaz con hardware**

La interfaz con el hardware se basa en tener clases encargadas de obtener los valores de los sensores o del accionar de los actuadores.

En la implementación de la interfaz que corrimos en Webots, hicimos llamadas al controlador del simulador, que utiliza sensores y actuadores simulados.

En la implementación que se comunica con el robot físico, las llamadas las hicimos a un servidor encargado de enviar y recibir paquetes del protocolo descripto en la sección ?? a través del puerto serial.

De esta forma es cuestión de decidir que implementación se usa en base a si se quiere correr en un simulador como Webots o en el robot físico. Si quisieramos usar otro simulador u otro tipo de hardware, sólo haría falta que implementemos la interfaz que cumpla con lo establecido e indicarle a la capa de comportamientos que la utilice para realizar su ejecución.

### **1.6.2. Interfaz con módulo de reconocimiento de objetos usando Visión**

Como el controlador de comportamientos es **cliente** del módulo de reconocimiento, necesita conocer en el instante de tiempo  $t$ , que objetos están siendo reconocidos. Para esto el módulo debe tener una fuente de información, en nuestro caso, una cámara usada en Visión.

Desde el punto de vista anatómico, ésto se puede ver como los ojos (cámara), la parte del cerebro encargada de analizar el estímulo recibido (imágenes) por los mismos (módulo de reconocimiento) y la parte del cerebro encargada de analizar los estímulos recibidos y realizar las acciones (controlador de comportamientos). Si hubiésemos usado algún tipo de conjunto de sensores táctiles para reconocer objetos (mano), en vez de visión, el módulo de reconocimiento de objetos bien podría analizar la información que ellos proveen e informar al controlador sobre su análisis.

Esta analogía puede llevar a pensar que los sensores de distancia u otros dispositivos que usamos en este desarrollo bien podrían formar parte de otro módulo, y no se estaría equivocado. La razón por la cual el módulo de visión está sepa-

rado y el resto de los sensores no, es que el procesamiento de una secuencia de imágenes es muy complejo y demandante, tal como detallamos en la sección ??.

## 1.7. Resultados obtenidos

En esta sección describimos los parámetros que utilizamos en la simulación para obtener los resultados sobre los diferentes comportamientos. Luego presentamos dichos resultados y sacamos conclusiones sobre los mismos.

A lo largo del proceso de desarrollo fuimos realizando diferentes simulaciones, de las cuales observamos defectos en las implementaciones de los comportamientos así como retardos y detalles que podíamos mejorar en algunos y que fuimos corrigiendo.

La arena utilizada en dichas simulaciones la mostramos en la figura 26. A lo largo de la misma hay ubicados 15 objetos que simulan ser basuras, un área de recarga de batería y un área de descarga de basura, marcadas por el cilindro naranja. Los valores de los parámetros que utilizamos los detallamos en el cuadro 2. Los ángulos los expresamos en radianes y las distancias las expresamos en metros, así como también los radios, correcciones y separaciones. La resolución de la cámara la expresamos en pixeles, la de la grilla en unidades y los tiempos están en milisegundos.

Parámetro	Descripción del parámetro	Valor
$P_r(0)$	Posición Inicial del robot	(-0.295,-0.4)
$O_r(0)$	Orientación Inicial del robot	0
$R_r$	Radio del robot	0.026
$dbw$	Distancia entre ruedas	0.052
$c_{dbw}$	Corrección de la distancia entre ruedas	1.09217109
$R_l$	Radio de la rueda izquierda	0.0205
$R_r$	Radio de la rueda derecha	0.0205
$c_l$	Corrección del radio de la rueda izquierda	0.99998702
$c_r$	Corrección del radio de la rueda derecha	1.00001297
$EncRes$	Resolución del encoder	159.23
$Cd$	Distancia de la cámara al centro del robot	0.0355
$Ch$ ó $C_h$	Altura de la cámara	0.038
$FOV_h$	Field of view horizontal	1.1
$ac$	Ángulo de la cámara	-0.5
$C_{rw}$	Ancho de la imagen obtenida de la cámara	640
$C_{rh}$	Alto de la imagen obtenida de la cámara	480
$A_w$	Ancho del arena	2
$A_h$	Alto del arena	1.2
$A_{rw}$	Ancho de la grilla	100
$A_{rh}$	Alto de la grilla	38
$Fsd$	Dist. s.de piso del medio al centro del robot	0.03
$Fss$	Separación entre sensores de piso	0.01
$\delta_f$	Ángulo umbral de basura enfocada	0.1
$T_s$	Time Step	32

Cuadro 2: Parámetros utilizados para las simulaciones

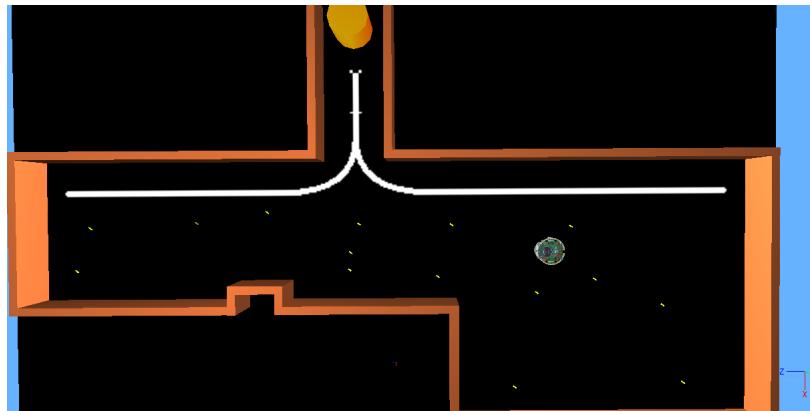


Figura 26: Arena que utilizamos para las simulaciones

#### 1.7.1. Distribución de tiempos y progreso de comportamientos en la simulación

En la figura 27 mostramos la distribución de time steps en la simulación de cada comportamiento. Podemos ver que *Recargar batería* y *Descargar basura* en conjunto están activos el 54 % de la simulación. Los comportamientos involucrados en la recolección de basura(*Enfocar basura*, *Ir a basura* y *Recolectar basura*) abarcan el 10.5 % del tiempo total de simulación. El 35.5 % restante está repartido en los comportamientos encargados de una navegación libre de obstáculos y situaciones peligrosas (*Wandering*, *Evitar obstáculos* y *Salir de situaciones no deseadas*). Todos los comportamientos relacionados con la basura(desde que se detecta hasta que se descarga) abarcan un 43 % del tiempo total de simulación. También se puede observar que el comportamiento que mayor tiempo está activo a lo largo de la simulación es *Descargar basura* y el que menor tiempo está activo es *Salir de situaciones no deseadas*.

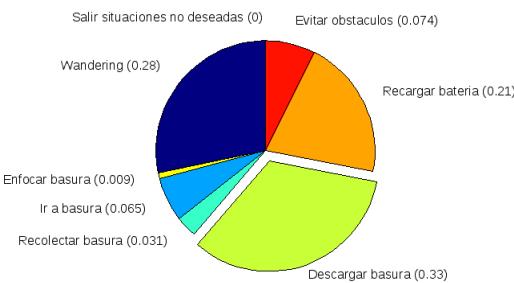


Figura 27: Distribución de los tiempos de los comportamientos en la simulación

En la figura 28 mostramos el progreso a lo largo de la simulación de todos los comportamientos. Observamos que en el comienzo ( $0 < t < 10000$ )<sup>2</sup> la mayoría

<sup>2</sup>t es el número de time step de la simulación

de los comportamientos está activo aproximadamente la misma cantidad de time steps, salvando el caso de *Descargar Basura*. En el período ( $40000 < t < 50000$ ), el comportamiento *Wandering* alcanza a estar activo la misma cantidad de steps que *Ir a basura*. Podemos ver también que la forma en que evolucionan *Enfocar basura*, *Ir a basura* y *Recolectar basura* a lo largo de la simulación es muy similar, salvando la cantidad de steps que están activos. La evolución de *Recargar batería* es periódica: inicia con una meseta de time steps en los cuales no está activo y luego tiene una pendiente pronunciada. Diferente es el progreso de *Evitamiento de obstáculos* ya que sus mesetas son las menos prolongadas y sus pendientes son poco abruptas.

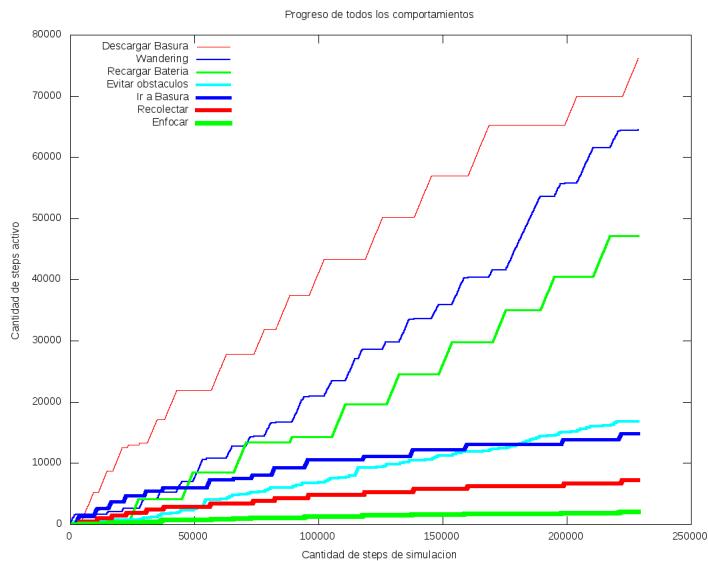


Figura 28: Evolución de los comportamientos a lo largo de la simulación

En la figura 29 mostramos más en detalle el comportamiento vital para nuestro proyecto, *Recolectar basura*. Podemos ver que en el período ( $0 < t < 50000$ ) acumula aproximadamente la misma cantidad de steps hechos que en el período ( $50000 < t < 150000$ ). El progreso describe mesetas cuyas duraciones se van prolongando a medida que se avanza la simulación y las pendientes tienen aproximadamente la misma magnitud a lo largo de la misma.

Para que pudieramos ver más en detalle lo que sucedió con los comportamientos, obtuvimos la cantidad de veces que cada uno se activó (Ver cuadro 3). En el mismo observamos que el comportamiento que más veces se activó fue *Evitar obstáculos*, seguido por *Enfocar basura* y *Wandering*. En un orden de magnitud menor están *Ir a descargar basura* e *Ir a recargar batería*. Finalmente, *Recolectar basura* se activó 15 veces y *Salir de situaciones indeseadas* nunca.

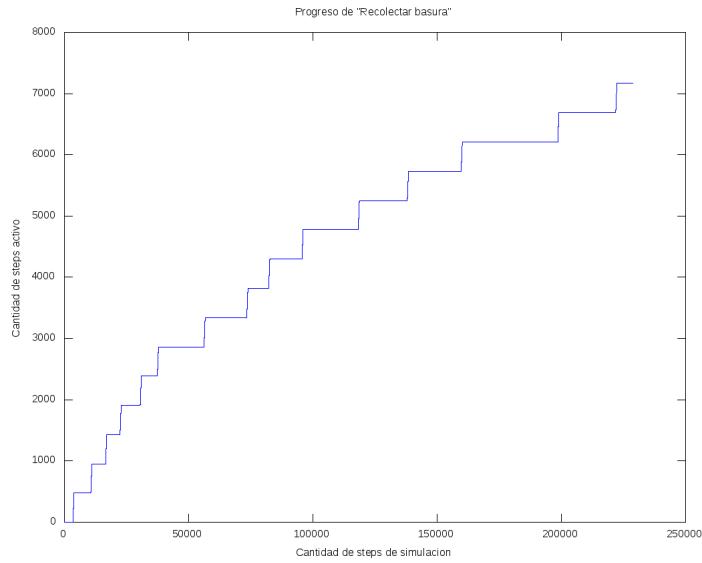


Figura 29: Progreso del comportamiento *Recolectar Basura*

Comportamiento	# total de steps activo	# veces que se activó
Wandering	64476	1330
Enfocar basura	2057	1384
Ir a basura	14832	1218
Recolectar basura	7166	15
Ir a descargar basura	76160	321
Ir a recargar batería	47188	265
Evitar obstáculos	16843	1507
Salir situaciones indeseadas	0	0

Cuadro 3: Resultados sobre steps en los cuales los comportamientos están activos

### 1.7.2. Tiempos en que los comportamientos están continuamente activos e inactivos

En los cuadros 4 y 5 mostramos cálculos sobre los steps que cada comportamiento estuvo *caa*<sup>3</sup> y *cai*<sup>4</sup>, respectivamente.

Podemos ver que *Recolectar basura* es en promedio el mayor en ambos cuadros y que tiene el menor desvío estándar en *caa*. *Wandering* es en promedio el que menor tiempo está *cai*, con el segundo menor desvío. Sigue lo mismo con *Evitar obstáculos*, segundo en promedio en estar *cai* y con el menor desvío. Ambos, además, tienen los mínimos máximos en estar *cai*, con una diferencia de un orden de magnitud con respecto al resto de los comportamientos. *Ir a descargar basura* e *Ir a recargar batería* son de los que mayor promedio y desvío tienen tanto en estar *caa* como en estar *cai*. Los comportamientos de *Enfocar basura* y *Ir a basura* tienen un promedio y desvío estándar de los más bajos en estar *caa*.

Comportamiento	Promedio	Desvío Estándar	# máximo
Wandering	48.44	225.07	2212
Enfocar basura	1.48	3.27	67
Ir a basura	12.17	58.44	1222
Recolectar basura	477.73	0.70	478
Ir a descargar basura	237.25	840.49	6012
Ir a recargar batería	178.06	647.39	3950
Evitar obstáculos	11.17	54.72	1386
Salir situaciones indeseadas	-	-	-

Cuadro 4: Resultados sobre steps que los comportamientos están continuamente activos

Comportamiento	Promedio	Desvío Estándar	# máximo
Wandering	123.4	754.4	8351
Enfocar basura	163.7	1766	38000
Ir a basura	175.5	1811	38040
Recolectar basura	13850	9479	38400
Ir a descargar basura	473.8	2689	30520
Ir a recargar batería	682.5	3322	23270
Evitar obstáculos	140.5	724.3	9485
Salir situaciones indeseadas	-	-	-

Cuadro 5: Resultados sobre steps que los comportamientos están continuamente inactivos

<sup>3</sup>caa: continuamente activo. Por continuamente activo se entiende que estuvo activo en el time step  $t - 1$  y en el  $t$

<sup>4</sup>cai: continuamente inactivo. Por continuamente inactivo se entiende que no estuvo activo en el timestep  $t - 1$  ni en el  $t$

### 1.7.3. Transiciones hacia y desde un comportamiento a otro

En el cuadro 6 mostramos las transiciones entre los comportamientos. Éstos están abreviados y presentados en el mismo orden que en la sección 1.4. El valor en la posición  $(i,j) = t_{ij}$ , siendo  $i$  y  $j$  el número de fila y columna respectivamente, se lee como: “desde el comportamiento  $i$  se pasó  $t_{ij}$  veces al comportamiento  $j$ ”. También se puede leer como: “el comportamiento  $j$  se activó  $t_{ij}$  veces siendo el comportamiento  $i$  el que anteriormente estaba activo”. Consideramos que hay una transición cuando  $i \neq j$ , por lo que  $t_{ii} = 0$ . Una transición de  $i$  a  $j$  con  $i > j$  puede ser porque el estímulo necesario para la activación del comportamiento  $i$  ya no está presente. Una transición de  $i$  a  $j$  con  $i < j$  puede ser porque el estímulo necesario para la activación del comportamiento  $j$  pasó a ser sensado.

El comportamiento que más veces “activó” a otro comportamiento es *Enfocar basura*, activando a *Ir a basura*. Observamos también que el caso contrario es la 2da transición más hecha y ambos valores están relativamente cercanos. El caso que los valores de  $t_{ij}$  y  $t_{ji}$  estén cercanos también sucede con *Evitar obstáculos* y el resto de los comportamientos, aunque en estos casos la diferencia es mucho menor.

Podemos ver que hay transiciones que sólo se dan 1 sola vez. Éstas son:

- *Wandering* a *Recolectar basura*,
- *Enfocar basura* a *Descargar basura* e
- *Ir a basura* a *Recargar batería*

	W	EB	IAB	RB	DB	R	EO
W	0	303	176	1	12	5	833
EB	238	0	1036	2	1	0	107
IAB	229	967	0	12	2	1	7
RB	12	3	0	0	0	0	0
DB	14	0	0	0	0	2	305
R	9	0	0	0	1	0	255
EO	828	111	6	0	305	257	0

Cuadro 6: Transiciones entre comportamientos

Mostramos también las transiciones en forma de porcentajes en los cuadros 7 y 8. En el primero detallamos las transiciones desde  $i$  hacia  $j$  y en el segundo, transiciones de  $j$  hacia  $i$ . Se puede ver que las transiciones que más sucedieron fueron desde *Descargar basura* y *Recargar batería* hacia *Evitar obstáculos*, con un porcentaje mayor al 95 %, y que el segundo también es la causa principal de la activación de los primeros, con porcentajes de 95 % y 97 % respectivamente.

En segundo lugar se ubican las transiciones desde *Enfocar basura* hacia *Ir a basura* y el recíproco, con valores cercanos al 75 %. Algo parecido sucede en el segundo cuadro.

El 80 % de las veces que estuvo activo *Recolectar basura* luego se activó *Wandering* y el 80 % de las veces que se activó el primero fue por parte de *Ir a basura*.

	W	EB	IAB	RB	DB	R	EO
W	0 %	0.227 %	0.132 %	0.001 %	0.009 %	0.003 %	0.626 %
EB	0.172 %	0 %	0.748 %	0.002 %	0.001 %	0 %	0.077 %
IAB	0.188 %	0.794 %	0 %	0.01 %	0.002 %	0.001 %	0.006 %
RB	0.8 %	0.2 %	0 %	0 %	0 %	0 %	0 %
DB	0.043 %	0 %	0 %	0 %	0 %	0.006 %	0.951 %
R	0.034 %	0 %	0 %	0 %	0.004 %	0 %	0.962 %
EO	0.55 %	0.074 %	0.004 %	0 %	0.202 %	0.170 %	0 %

Cuadro 7: Transiciones desde  $i$  hacia  $j$  en porcentajes

	W	EB	IAB	RB	DB	R	EO
W	0 %	0.179 %	0.172 %	0.009 %	0.011 %	0.007 %	0.623 %
EB	0.219 %	0 %	0.699 %	0.002 %	0 %	0 %	0.08 %
IAB	0.145 %	0.851 %	0 %	0 %	0 %	0 %	0.004 %
RB	0.066 %	0.134 %	0.8 %	0 %	0 %	0 %	0 %
DB	0.037 %	0.003 %	0.006 %	0 %	0 %	0.003 %	0.95 %
R	0.019 %	0 %	0.003 %	0 %	0.008 %	0 %	0.97 %
EO	0.553 %	0.071 %	0.004 %	0 %	0.202 %	0.170 %	0 %

Cuadro 8: Transiciones desde  $j$  hacia  $i$  en porcentajes

#### 1.7.4. Progreso de recorrido de la arena a lo largo de la simulación

En la figura 30 mostramos el progreso del área recorrida por el robot a lo largo de la simulación. El arena estaba dividida en  $A_{rw}$ .  $A_{rh} = 3800$  slots. Como es de esperarse, la suma de cantidad de slots vistos y slots restantes en un instante de tiempo  $t$  es  $cte = 3800$ . El tiempo en que se llegó a recorrer todo el arena fue  $t_{fv} = 95241$ . Dado que la duración de la simulación fue de  $t_{fs} = 228721$ , el arena se recorrió en el 0.42 % del tiempo total de simulación.

Se puede ver que al principio de la simulación ( $0 < t < 17000$ ) hay pendientes abruptas, con mesetas de pocos steps. Luego las pendientes tienen menor valor y las mesetas son más prolongadas, salvando el caso del intervalo ( $35000 < t < 40000$ ).

En el cuadro 9 vemos más en detalle el progreso del porcentaje de arena cubierta. Tomamos 10 muestras uniformes en el período ( $0 < t < t_{fv}$ ) y obtuvimos el porcentaje cubierto en esa muestra, así como también el acumulado la misma. Podemos ver que ya en la 2da muestra,  $0 < t < 0.042t_{fs}$ , se alcanza a recorrer el 50 % del total del arena. Después los incrementos son más pequeños en comparación a los primeros, salvando el caso de la cuarta y última muestra. En la primer muestra el robot recorrió aproximadamente el 25 % de la arena, en la segunda el 50 % y en la cuarta el 75 %.

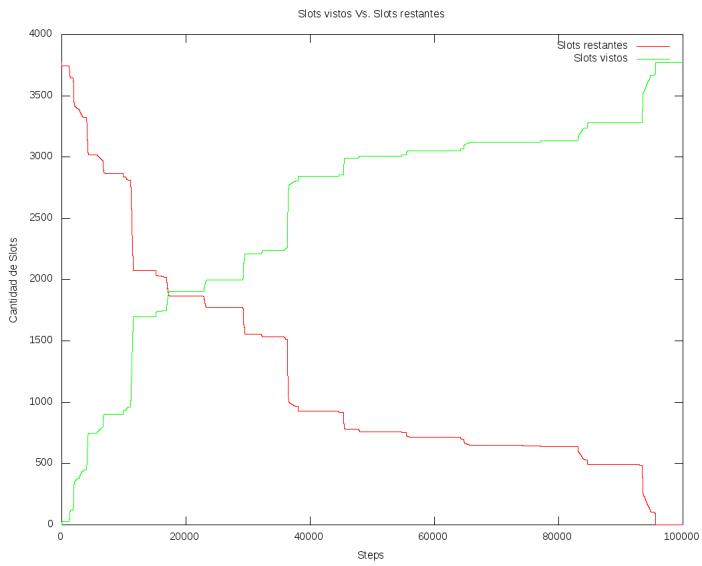


Figura 30: Área visualizada y área no visualizada a lo largo de la simulación

Nº muestra	(%)	(%) Acumulado
1	0.246	0.246
2	0.258	0.505
3	0.081	0.586
4	0.167	0.754
5	0.044	0.798
6	0.011	0.809
7	0.017	0.827
8	0.003	0.830
9	0.038	0.869
10	0.130	1

Cuadro 9: Porcentaje de arena cubierta

## 1.8. Conclusión

En ésta sección sacamos conclusiones sobre los resultados obtenidos en la sección 1.7, así como dar posibles explicaciones a los mismos.

Dado que en la simulación había 15 basuras, era esperado que *Recolectar basura* se activase la misma cantidad de veces. En promedio, el robot recolectó  $\frac{t_{fs}}{15 \text{ basuras}} = \frac{1.52 \times 10^5 \text{ Ts}}{\text{basura}}$ , es decir, 1 basura cada 8 minutos. El mayor tiempo que pasó entre una recolección y otra fue de  $38400 \text{ Ts} = 20m28s$ .

Los comportamientos íntimamente relacionados, *Enfocar basura* e *Ir a basura* se activaron más de 15 veces. Ambos comportamientos tuvieron un nivel de activación parecido a la cantidad de steps que estuvieron activos, provocando un bajo promedio de steps *caa*. Ésto pudo haber sido causado debido al umbral  $\delta_f$  en el cual se decide si una basura está enfocada o no (ver figura 16). Un valor muy grande del mismo, figura (a), cubre más porcentaje de la imagen, por lo que va a llevar a indicar que la basura se enfoca más rápido, disminuyendo el tiempo en que el comportamiento está activo. Un valor chico (figura (b)), hará que la mayor cantidad de veces que se detecte una basura, la misma no esté enfocada y al cubrir menos porción de la imagen, aumentará el promedio de time steps en los cuales tiene que enfocar. Éste caso es más sensible a la situación en la que la basura se mueve, ya que va a ocasionar que se tenga que enfocar nuevamente. El primer caso es menos sensible al movimiento de la basura, pero tiene el siguiente defecto: cuando la basura se sienta como enfocada, la misma *NO* está frente al robot, y el comportamiento de *Ir a basura* sólo va hacia adelante, suponiendo que la basura se encuentra frente al robot, ocasionando que la basura pase a no estar enfocada y una nueva activación de *Enfocar basura*. En los cuadros de transiciones 7 y 8 podemos ver que ambos comportamientos son los mayores proveedores de transiciones de dichos estados al otro.

Como ya mencionamos, *Ir a descargar basura* e *Ir a recargar batería* en conjunto abarcan el 54 % del tiempo total de simulación. Además son de los que mayor promedio están *caa*, lo que nos lleva a pensar que una activación incorrecta de los mismos es indeseable, ya que son de los que más jerarquía tienen y por lo tanto inhibirían innecesariamente y por un tiempo relativamente prolongado a otros. Una activación incorrecta puede estar dada por:

- Mal funcionamiento de los sensores
- Haber juntado un objeto que no era basura
- Una falla en la lógica de activación de comportamientos

El gran desvío estándar de ambos en *caa* se debe a que su acción es llevar al robot hacia las líneas, y la distancia y el recorrido por las mismas varía según la posición del robot cuando se hizo presente el estímulo correspondiente.

El promedio en estar *cai* de *Ir a descargar basura* depende directamente de la cantidad de basuras que haya en la arena y la efectividad del módulo de reconocimiento. En el caso que no haya basuras en la arena, lo ideal es que el comportamiento nunca se active, aunque ésto depende de las posibles activaciones incorrectas. El gran desvío estándar en estar *cai* se debe a que al principio de la simulación había más basuras para recolectar, disminuyendo el tiempo entre activaciones, y llegando al final de la simulación había menos basuras, disminuyendo la probabilidad que el robot las encuentre y aumentando el tiempo entre activaciones.

Los valores de *Wandering* dependen totalmente de las activaciones de los otros comportamientos, ya que está activo si y sólo si ningún otro está activo. En un arena con 15 basuras, estuvo activo un 28 % del tiempo de simulación. Teniendo en cuenta que los comportamientos relacionados con las basuras abarcaron un 43 % de  $t_{fs}$ , es de esperarse que en una simulación sin basuras el protagonismo de *Wandering* sea el mayor de todos superando el 70 %. Su bajo promedio y desvío estándar en estar *caí* puede estar dado por situaciones en que está activo, un comportamiento pasa a estar activo por un período corto de tiempo(por ejemplo, *Evitar obstáculos*, que causa el 62 % de activaciones) hasta que desaparece el estímulo que lo activó y el robot vuelve a deambular. En la mayor parte de los casos entre una activación y otra de *Wandering* no hay muchos comportamientos que se activan.

*Evitar obstáculos* es un comportamiento que tiene ciertas cualidades. Una de ellas se puede ver en los cuadros de transiciones. Las transiciones hacia él y desde él se dan en proporciones muy parecidas, esto quiere decir que  $t_{7j}^1 \approx t_{7j}^2$ , siendo  $t^1$  y  $t^2$  transiciones de los cuadros 7 y 8 respectivamente.

Ésto da la siguiente idea: hay un comportamiento activo *A* y sus respuestas están llevando al robot a una posible colisión. En ese momento se activa *Evitar obstáculos*, detiene *A*, saca al robot de la posible colisión y vuelve a activar a *A*. Cabe aclarar que en realidad no lo activa a *A*, sino que desaparece el estímulo de *Evitar obstáculos* y *A* toma el control nuevamente. Su promedio y desvío estandar de *caa* son bajos y se ven reflejados en la figura 28. En la misma figura se puede observar el efecto de tener un promedio y desvió estándar bajos, reflejados en la duración muy corta de las mesetas. El poder de detener a otros comportamientos está dado por su nivel en la arquitectura. Es importante resaltar que este beneficio viene solo por el hecho de usar *Subsumption*, y en el caso de haber utilizado otra arquitectura, nos hubiese demandado trabajo.

El comportamiento de *Salir de situaciones no deseadas* no se activó nunca. Éste hecho es muy importante: indica que el resto de los comportamientos nunca se “activaron mutuamente” y por lo tanto no hubo situaciones que puedan llevar al robot a quedarse sin carga en la batería, arriesgando su autonomía. El hecho que no se active nunca el comportamiento puede llevar a pensar que no es necesario tenerlo. Éste pensamiento es totalmente erróneo ya que uno de nuestros objetivos era que la autonomía del robot no corra riesgo bajo ningún concepto. Cabe aclarar que hay situaciones que escapan al alcance de este proyecto, por ejemplo, que se rompa una rueda.

El funcionamiento de *Salir de situaciones no deseadas* lo probamos en diferentes simulaciones anteriores a la que presentamos y el resultado fue el esperado.

En la sección 1.7 vimos que algunas transiciones sucedían sólo una vez. A continuación detallamos posibles causas de las mismas:

1. *Wandering a Recolectar basura,*
2. *Enfocar basura a Descargar basura e*
3. *Ir a basura a Recargar batería*

El primer caso puede estar dado por el siguiente escenario:

- El robot esquivó un obstáculo.
- En el time step que desapareció ese estímulo, había una basura en la imagen detectada por la cámara.

- En dicho time step, el sistema de reconocimiento no la detectó, activando *Wandering*.
- En el siguiente time step, el sistema de reconocimiento detecta la basura, y además se encuentra enfocada y a una distancia en la cual puede activarse el comportamiento de *Recolectar basura*.

El segundo caso puede estar dado por el siguiente escenario:

- El robot recolectó una basura.
- En el time step que se desactivó la recolección, había otra basura en la imagen detectada por la cámara y por el sistema de reconocimiento que era necesario enfocar para que sea recolectada, pero la basura recolectada no había llegado a activar el sensor del depósito de basura. Estó llevó a la activación de *Enfocar basura*.
- En el siguiente time step, la basura recolectada activa el sensor del depósito y por ende, activa el comportamiento *Descargar basura*.

El tercer caso es más sencillo de explicar:

- En el time step  $t$  el robot se dirigía hacia una basura.
- En el time step  $t + 1$ , el valor de la batería hizo que se active el comportamiento de *Recargar batería*. Ésto es posible debido a que el último tiene mayor nivel en la arquitectura que el primero.

Otro dato curioso es que el 80 % de las veces que estuvo activo *Recolectar basura* luego se activó *Wandering* y el restante a *Enfocar basura*. Teniendo un almacenamiento máximo de 1 basura, es de esperarse que luego de recolectar, se active *Descargar basura*. Ésto no sucede ya que, como explicamos en el escenario del segundo caso, hay un período de time steps en los cuales la basura recolectada está yendo hacia el sensor del depósito, permitiendo la activación de otros comportamientos.

Finalmente, el robot recorrió toda la arena en el 42 % del tiempo total de simulación. Éste porcentaje disminuye o aumenta dependiendo de si aumenta el tiempo de simulación o disminuye. Es decir, lo importante no es el porcentaje del tiempo total, sino el tiempo en que recorrió el área en su totalidad. Dicho tiempo fue de  $95241Ts = 50m47s$ . Durante éste tiempo el robot no sólo recorrió un poco menos de  $200m^2$ (la escala es 10:1), sino que recolectó 9 basuras de las 15 que había en la arena y fue a descargarlas(uno de los comportamientos que más tiempo lleva). Además, se recargó 3 veces en ese período, como dijimos anteriormente, es otro de los comportamientos que más tiempo demanda. Otro factor importante para destacar es que el 75 % del área se recorrió en un cuarto del tiempo en cuestión, es decir, en aproximadamente  $13mins$ . Por todo ésto, concluimos que el recorrido de la arena es muy eficiente.

## A. Implementación de arquitectura de software del controlador

La implementación de la arquitectura de software del controlador la hicimos en C++, tal como explicamos en la sección 1.6. La relación explicada en dicha sección entre el robot (con sus dispositivos), los comportamientos y el sistema de reconocimiento se puede ver más claramente en la figura 31, unidos por la clase *GarbageCleaner*, que implementa la arquitectura *Subsumption*. En la misma también está el main-loop. Mostramos en la figura 32 que *GarbageCleaner* posee comportamientos, así como también un robot y el mismo posee dispositivos que pueden ser sensores o actuadores, tal como se ve en la figura 33. También posee el módulo de reconocimiento de objetos, mostrado en la figura 34. La implementación de *GarbageCleaner* es simple, ya que debe:

- Obtener los sensores y actuadores del robot.
- Inicializar el módulo de reconocimiento de basuras.
- Instanciar los comportamientos, brindándoles los sensores, actuadores y eventualmente el módulo de reconocimiento, necesarios para saber si su estímulo está activo y para poder interactuar con el entorno.
- Correr el main-loop, utilizando la arquitectura de comportamientos elegida (*Subsumption*), haciendo actuar al comportamiento cuyo estímulo esté activo en ese instante de tiempo y tenga mayor nivel en la arquitectura.

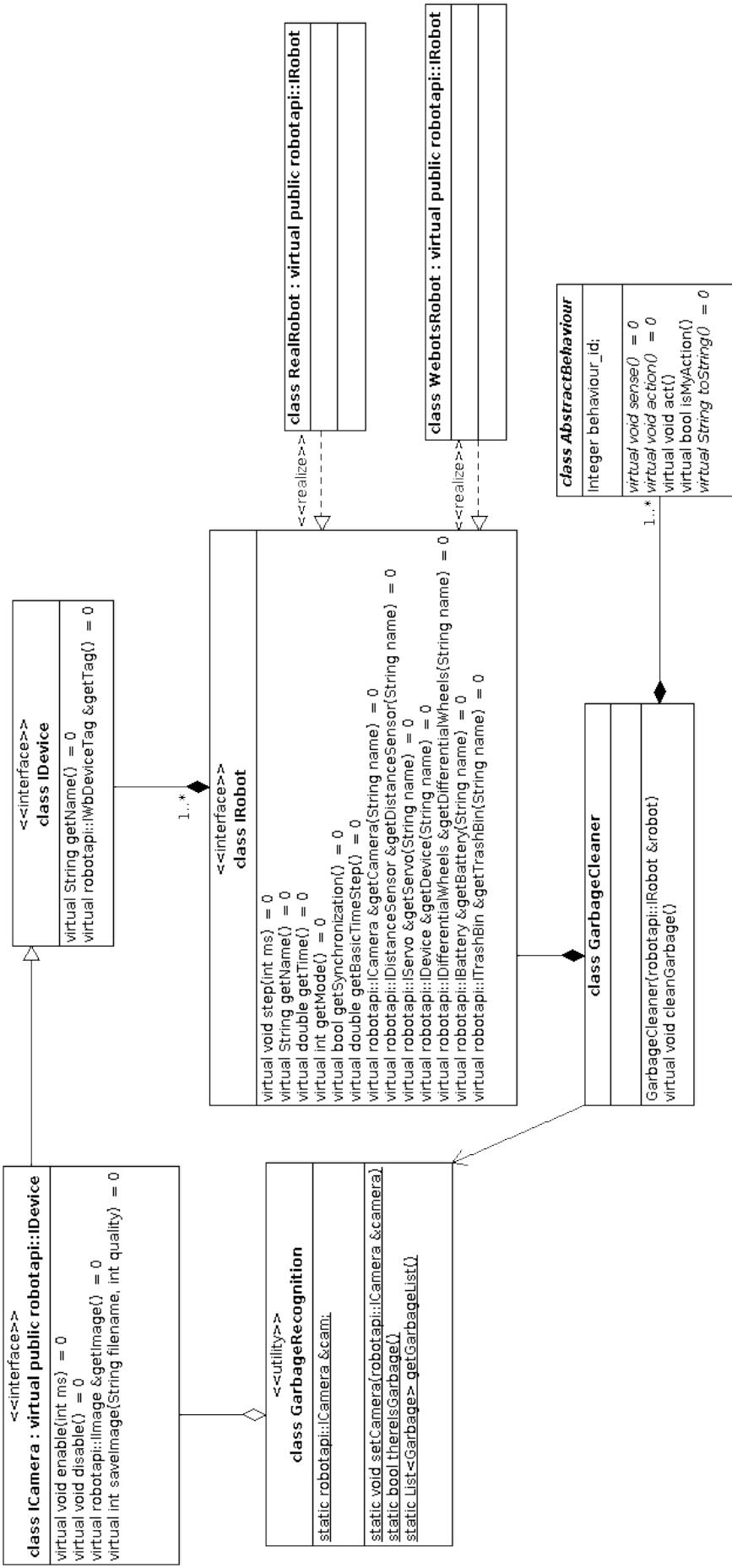


Figura 31: Diagrama de clases de la arquitectura de software. Relación entre las 3 grandes componentes del proyecto

## A.1. Comportamientos

Como ya mencionamos anteriormente, en la figura 32 mostramos la relación de *GarbageCleaner* con los comportamientos. Podemos ver que más precisamente posee *AbstractBehaviours*. Ésto se debe a que la única información que necesita para coordinarlos es saber si están activos (método *sense*) y luego poder indicarles que den su respuesta (método *act*).

Para agregar un comportamiento, es necesario que extienda de *AbstractBehaviour* e implemente los métodos abstractos de la clase: *sense*, *action* y *toString*. En el primero el comportamiento debe indicar si está activo o no en base a la información de los sensores que utilice. En el segundo se debe implementar la respuesta del comportamiento en el caso que esté activo. El último método provee una descripción del comportamiento implementado. Finalmente, hay que instanciarlo y agregarlo a la lista de comportamientos que posee *GarbageCleaner*.

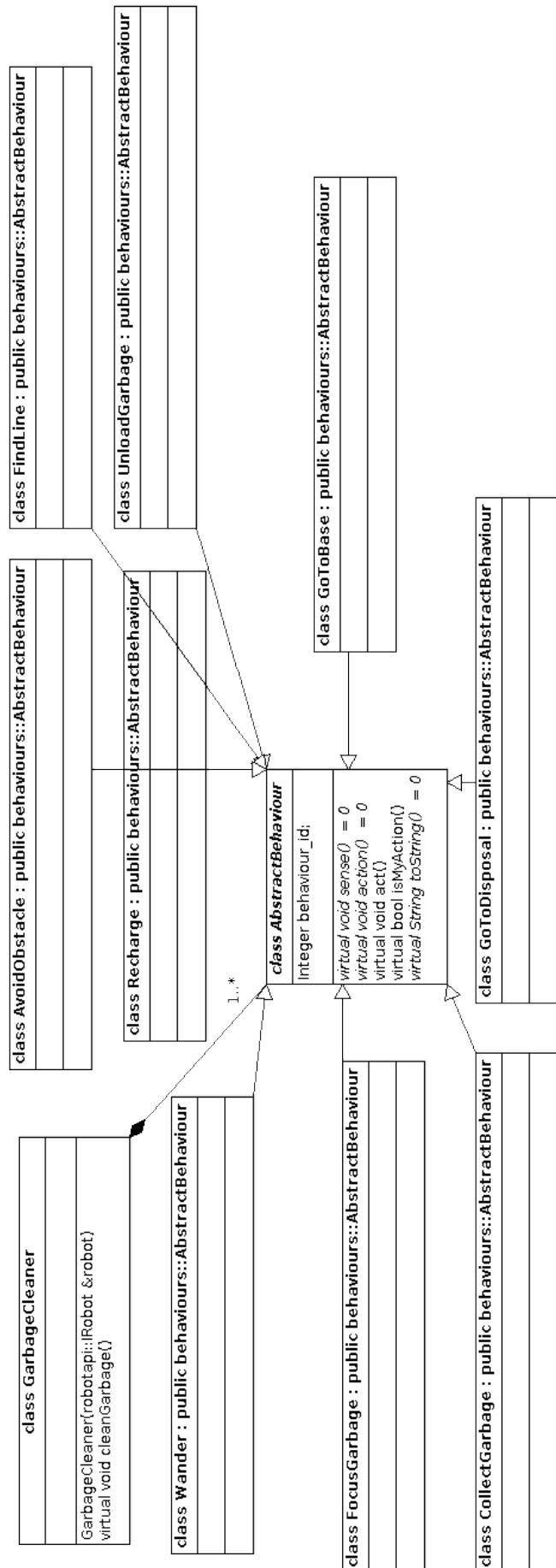


Figura 32: Diagrama de clases de la arquitectura de software. Paquete de comportamientos

## A.2. Dispositivos del robot

Para que los comportamientos sensen y actúen con el entorno, deben disponer de dispositivos que les permitan hacerlo. Éstos últimos forman parte del *IRobot* que le es entregado a *GarbageCleaner* para que instancie los comportamientos. Como se puede observar, la interfaz que representa al robot provee métodos de obtención de sensores como *getDistanceSensor* y de actuadores, como *getDifferentialWheels*. Para el desarrollo de la arquitectura nos basamos fuertemente en la api que provee Webots para interactuar con su robot simulado.

Para minimizar el acoplamiento entre *IRobot* y *GarbageCleaner*, hicimos que los dispositivos sean pedidos por un nombre que los representa. En la implementación de la interfaz para la simulación con Webots (*WebotsRobot*), la mayor cantidad de llamadas a éstos métodos son simples wrappers de una invocación al dispositivo virtual del programa de simulación. La implementación para el robot real, *RealRobot*, posee un mapa que relaciona los nombres de los dispositivos con instancias de implementaciones de los mismos. Éstas implementaciones, como por ejemplo *RealDistanceSensor*, tienen los handlers necesarios para poder enviar y recibir información de los sensores y actuadores de los cuales son responsables. En el caso de *RealDifferentialWheels*, el mismo posee dos handlers, dado que cada handler es capaz de interactuar con una placa con determinado groupid y boardid, y cada motor posee una placa que lo controla.

En el caso que se quiera agregar un sensor nuevo a la api, como por ejemplo un GPS, es necesario crear la interfaz que represente el mismo, con métodos de obtención de valores de los sensores o seteo de valores para los actuadores. Luego de crear la interfaz, es necesario agregar un método a la interfaz de *IRobot* para poder obtener una instancia del dispositivo en cuestión. Después es necesario implementar éste método en las clases *WebotsRobot* y *RealRobot*, dependiendo si se va a querer simular con Webots, correr en la realidad o ambos. Finalmente, se deben realizar las implementaciones correspondientes de la interfaz creada para el dispositivo.

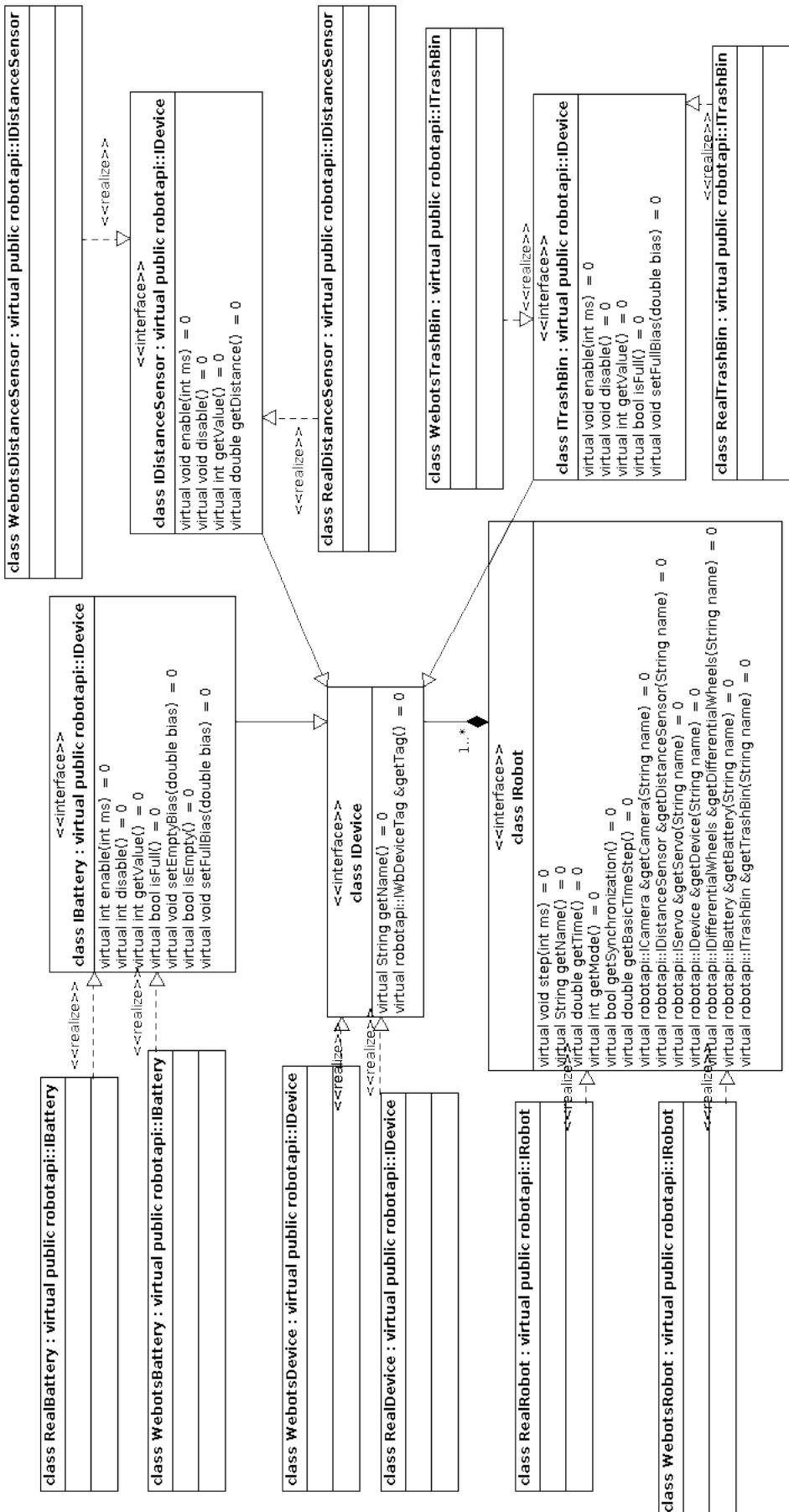


Figura 33: Diagrama de clases de la arquitectura de software. Paquete de api de 1 robot

### A.3. Reconocimiento de objetos

La figura 34 exhibe la relación entre el módulo de reconocimiento de objetos y el módulo de comportamientos. El módulo de visión queda encapsulado dentro de la clase de *GarbageRecognition*, permitiendo que el módulo de comportamientos funcione en forma independiente a éste. La creación de una interface para la cámara nos permite alternar entre el entorno de webots y el mundo real. El comportamiento de las clases *WebotsCamera* y *RealCamera* es el mismo pero sus implementaciones difieren en que, en el caso de la primera, se pide la imagen a webots mientras que en la segunda se toma una imagen del mundo real. Por motivos similares utilizamos una interface para la representación de las imágenes. Esto se debe a que OpenCv y Webots codifican las imágenes de maneras distintas y tuvimos que implementar sus respectivas clases.

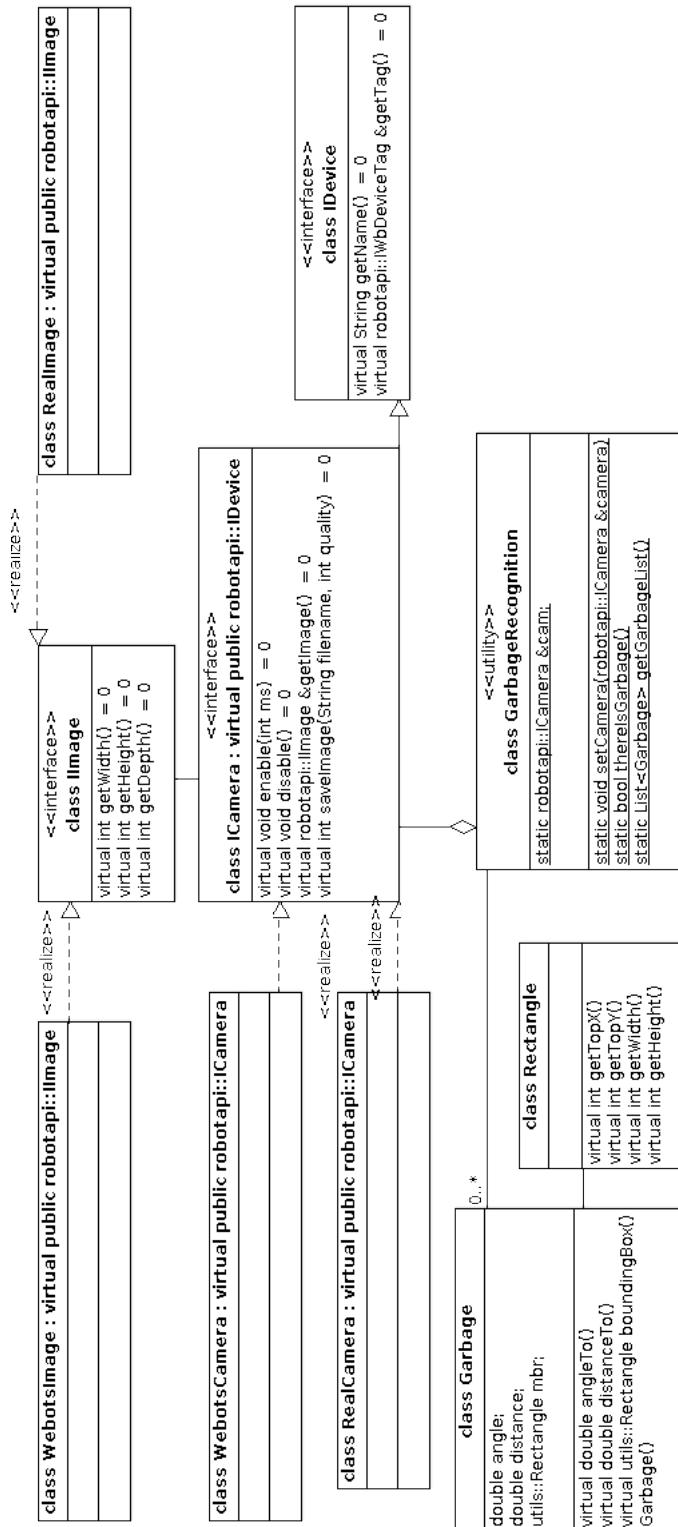


Figura 34: Diagrama de clases de la arquitectura de software. Paquete de reconocimiento de basuras

## B. Implementación del protocolo de comunicación del lado de la PC

### B.1. Packet Server

La implementación del protocolo en la PC la hicimos en C++, al igual que el controlador y el módulo de reconocimiento de basuras. Para la misma implementamos los paquetes descriptos en el protocolo y un servidor de envío y recepción de los mismos a través del puerto serial. También implementamos lo que llamamos handlers, quienes son los responsables de convertir los comandos de la api del robot en paquetes y enviarlos al servidor, así como también de recibir los paquetes de respuesta que le incumben y proveerlos a la api de una forma que ésta los entienda. Mostramos el diagrama de ésta relación en la figura 35.

Como mencionamos anteriormente, hay un servidor que denominamos packet server encargado del envío y recepción de paquetes. El servidor sólo se encarga de mandar una serie de bytes y de recibir los mismos, sin inspeccionar de qué tipo de paquete se trata. Provee dos métodos para realizar el envío y recepción:

- sendPacket: Su función es recibir un paquete y encollarlo para que el servidor lo envíe. Como el servidor es un thread diferente al de la api, no se interrumpe la recepción o envío de paquetes.
- registerHandler: Registra un handler para un paquete enviado desde una placa con grupo *groupid* e id de placa *boardid*. Cuando llega un paquete de dicha placa, el servidor invoca el método handlePacket() del handler registrado.

La idea del método handlePacket es que se limite sólo a guardar los nuevos valores recibidos, ya que durante la invocación del mismo el server no recibe ni envía paquetes. La clase packet provee funciones de utilidad para cualquier tipo de paquete, tales como cálculo y verificación de CRC, seteo y obtención de valores de los campos del paquete, entre otras.

### B.2. Packets

Como indicamos en la figura 36, group packet extiende a packet, agregando métodos de utilidad para los comandos que todos los grupos son capaces de recibir. Lo mismo sucede con board packet, extendiendo de group packet ya que es más específico que este último.

En las figuras 37 y 38 mostramos los paquetes que extienden de board packet, específicos para cada sensor o actuador que pueda haber en una placa. Como se puede observar, cada paquete específico tiene funciones que permiten obtener y setear información propia del sensor/actuador en cuestión, respetando el protocolo que propusimos.

Por ejemplo, BatteryPacket permite setear los thresholds o umbrales a los cuales informará que la batería se encuentra cargada o en estado crítico, respectivamente. Además permite obtener el valor de carga de la batería.

A continuación damos un ejemplo de uso de un paquete. Supongamos que se quiere saber la carga de la batería. Para ésto instanciamos un BatteryPacket con el id de grupo y placa correspondientes. El mismo se encarga del seteo de

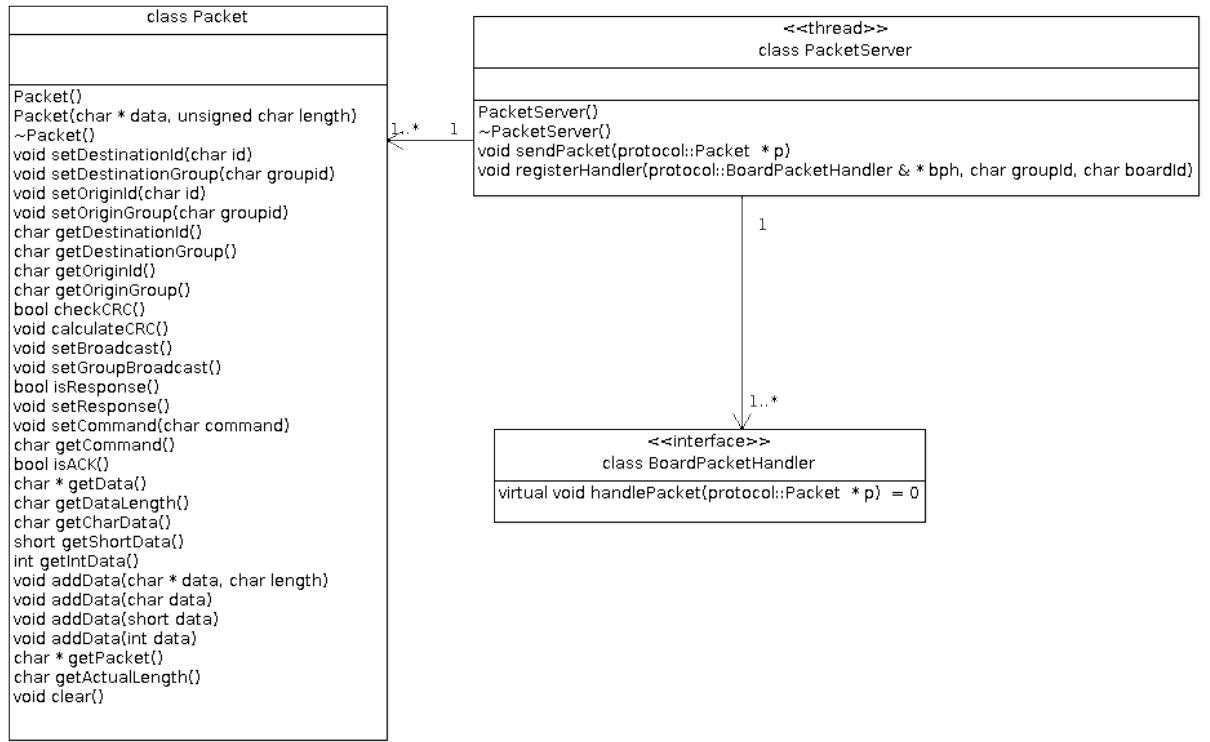


Figura 35: Diagrama de clases. Estructura general del servidor de paquetes e interface para los handlers.

los campos de grupo y placa de origen y destino, tamaño del paquete y comando. Le indicamos que queremos el valor de carga con `senseBattery()`, que pone el comando correspondiente en el campo del paquete. Una vez que tenemos el paquete armado, llamamos a la función `sendPacket()` del packet server. Suponiendo que registramos un handler para ese id de grupo y de placa, cuando el server reciba la respuesta de la placa va a llamar al método `handlePacket()` del handler. En la función, instanciamos un `BatteryPacket` y llamamos a la función `analysePacket()` con el paquete que recibimos. Finalmente, nos queda invocar al método `getBatteryValue()` de este paquete.

### B.3. Handlers

En las figuras 39 y 40 mostramos los handlers implementados para que las implementaciones de los sensores de la api del robot pudieran manejar los mismos, sin tener conocimiento sobre el protocolo. Hay un handler por default, `DefaultBoardPacketHandler`, para el caso en que no haya registrado un handler para determinado id de grupo y placa. La función `handlePacket` del mismo se encarga de imprimir por salida estándar cada campo del paquete en formato hexadecimal.

Se puede observar que la cantidad de métodos en los paquetes y en los handlers no es la misma, y en algunos casos, hay métodos que en el otro no están.

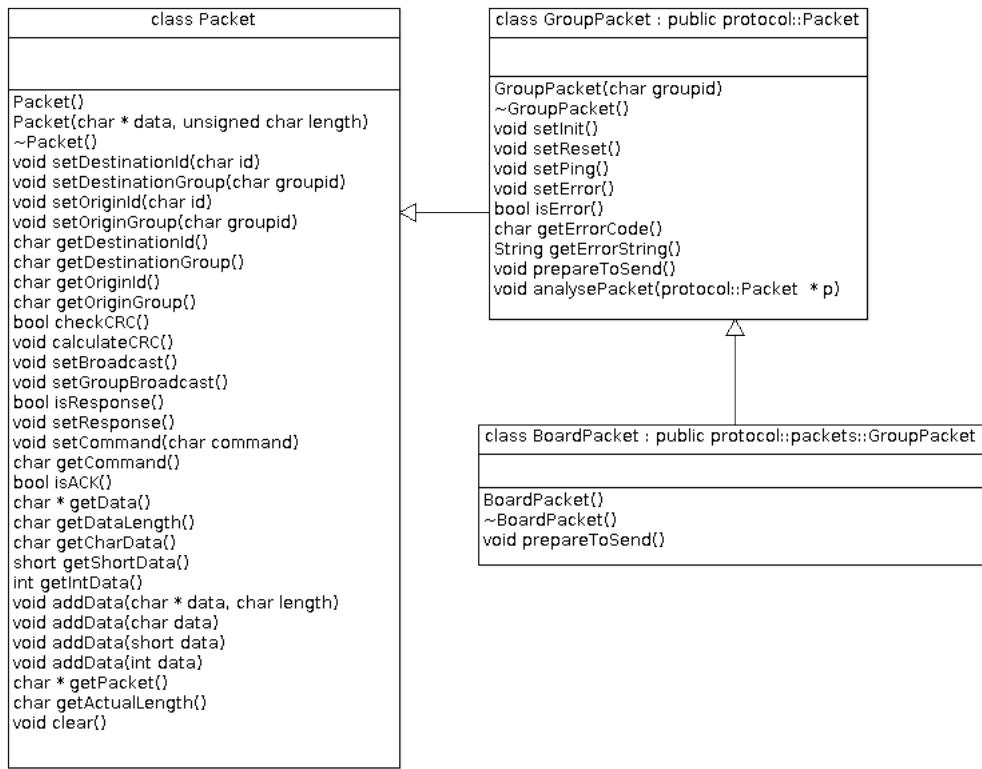


Figura 36: Diagrama de clases. Estructura general de los paquetes.

Ésto se debe a que los handlers son intermediarios entre el protocolo y la api del controlador encargado de la ejecución de los comportamientos. En algunos casos, el protocolo brinda más opciones de las que realmente utiliza el controlador para funcionar.

#### B.4. Qué hacer en caso que se modifique el protocolo

En el caso que haya una modificación en el protocolo, se deberán actualizar las clases afectadas por los cambios, y eventualmente, los handlers correspondientes.

Supongamos el caso en el que se agrega un campo de 1 byte luego del tamaño total del paquete. Éste cambio, en principio, sólo afecta la clase Packet. En el caso que el campo indique algo sobre el grupo o placa, también afectará a group packet o board packet, respectivamente. En el caso que se amplíe el set de comandos de un determinado sensor o actuador, es necesario modificar el packet y handler asociados a los mismos. Lo importante de ésto es que los cambios están localizados. Es decir, un cambio en el protocolo llega como mucho hasta los handlers, aislando a la api del controlador de posibles cambios.

## Referencias

- [1] Salvatore Candido. Autonomous robot path planning using a genetic algorithm.
- [2] Amalia F. Foka and Panos E. Trahanias. Predictive autonomous robot navigation. In *In Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 490–495, 2002.
- [3] Maria C. Neves and R. S. Tome. A control architecture for an autonomous mobile robot. In *Proceedings of Autonomous Agents*. ACM, 1997.
- [4] Stefano Nolfi. Evolving non-trivial behaviors on real robots: A garbage collecting robot. *Robotics and Autonomous Systems*.
- [5] K. Wedeward R. Clark, A. El-Osery and S. Bruder. A navigation and obstacle avoidance algorithm for mobile robots operating in unknown, maze-type environments. December 2004.

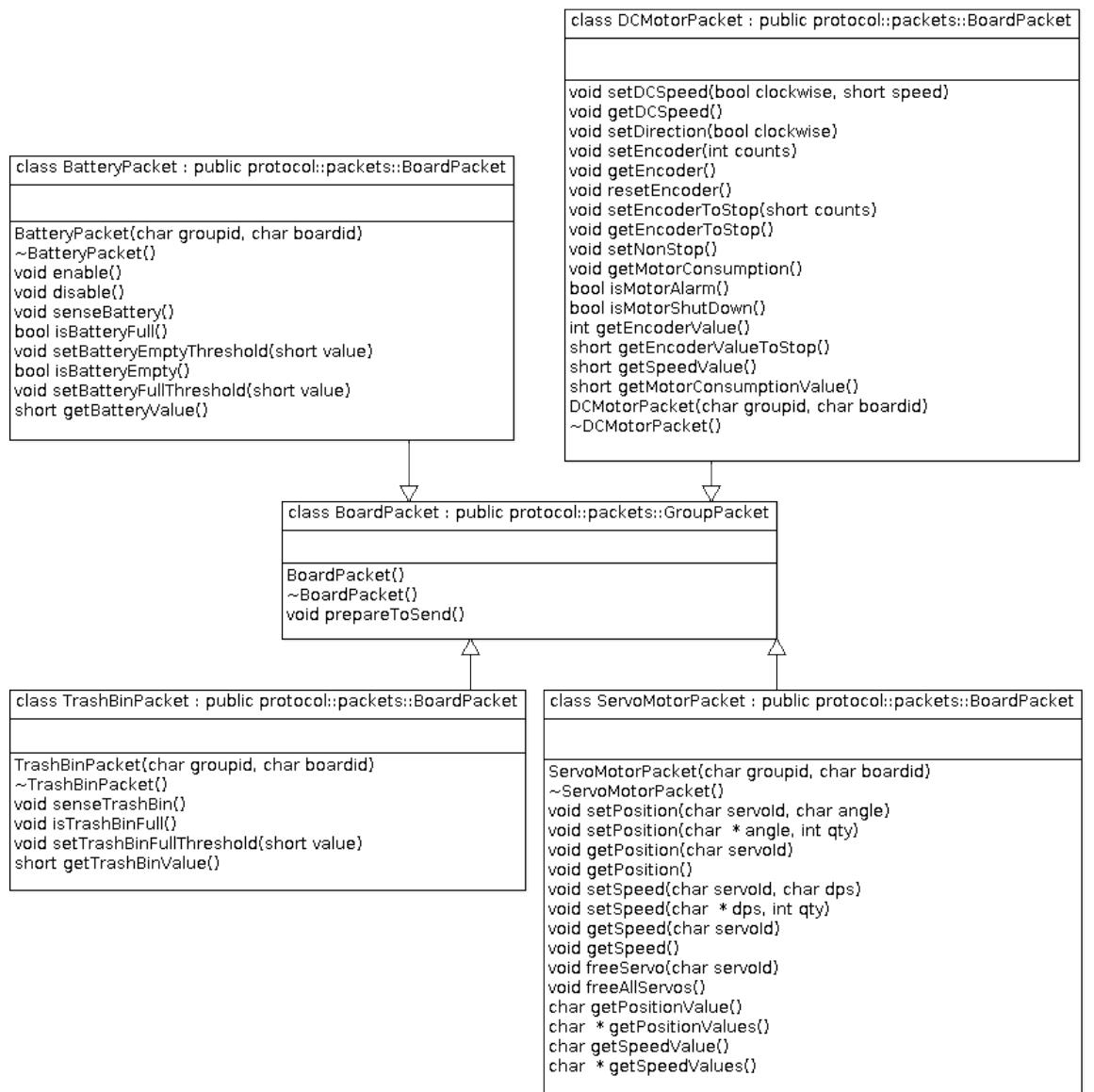


Figura 37: Diagrama de clases. Estructura de los paquetes de motor, recipiente de basura, servos y batería.

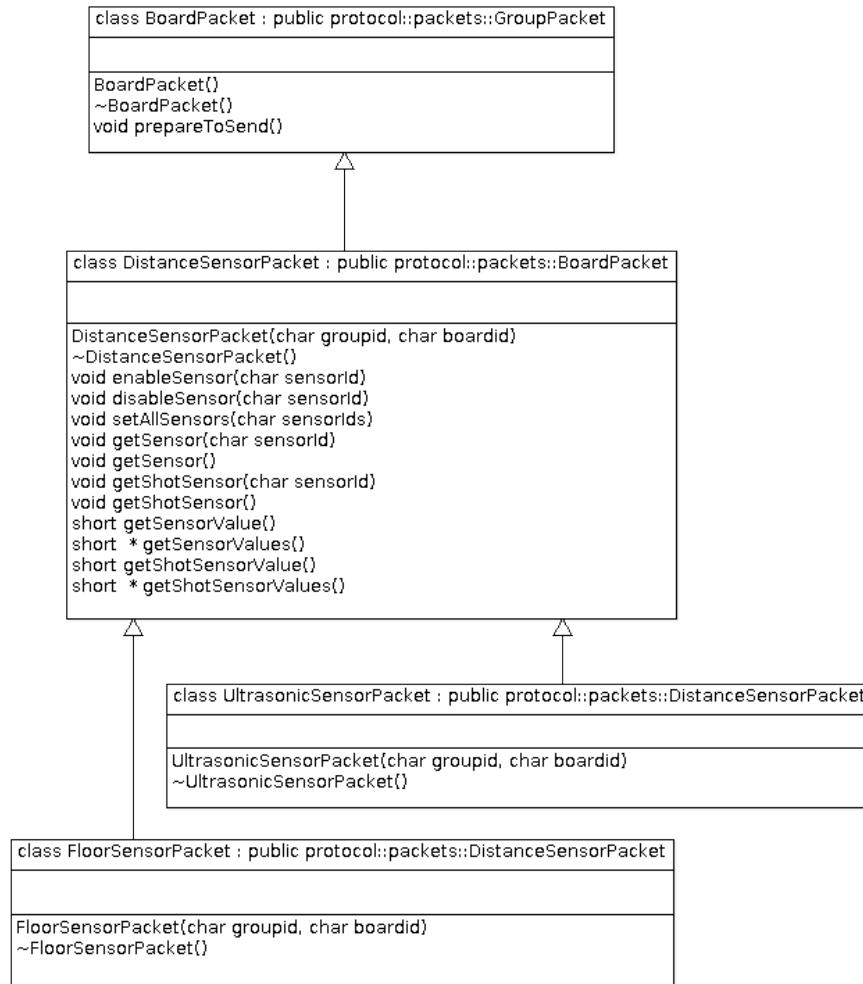


Figura 38: Diagrama de clases. Estructura general de los paquetes para sensores de distancia.

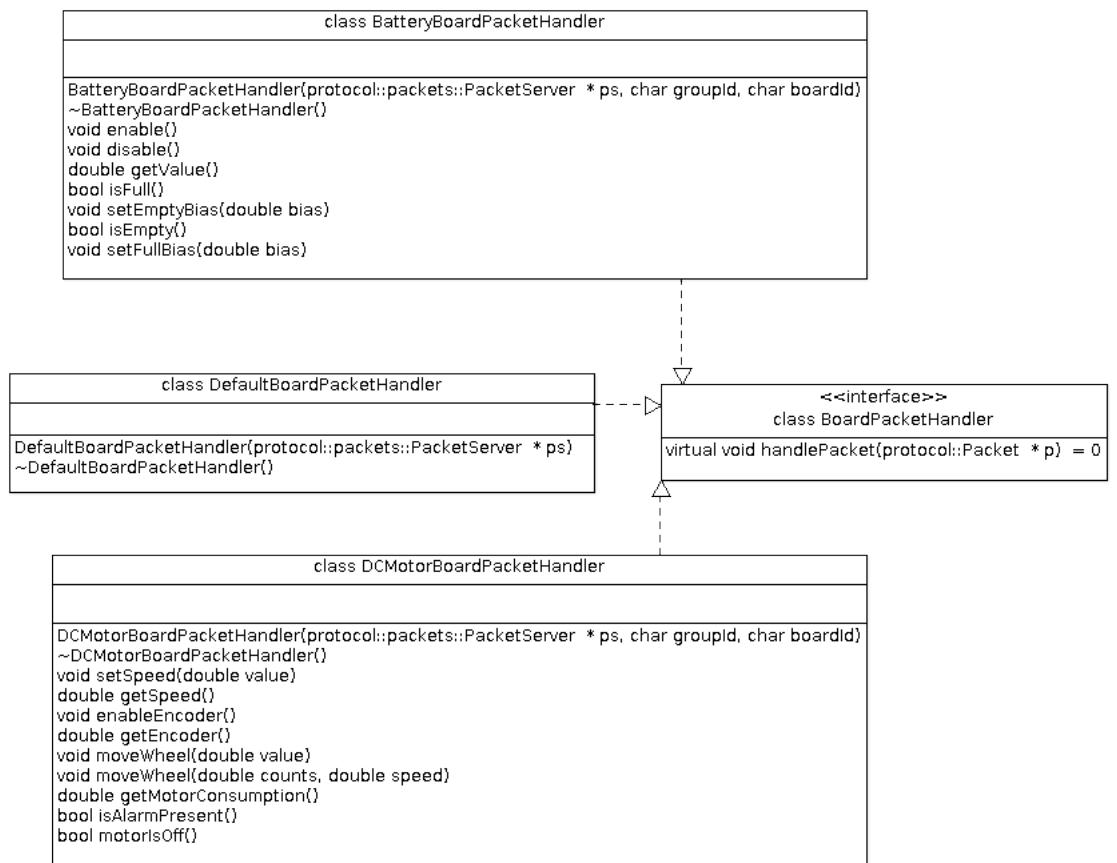


Figura 39: Diagrama de clases. Handlers de paquetes default y para las placas de motor y batería.

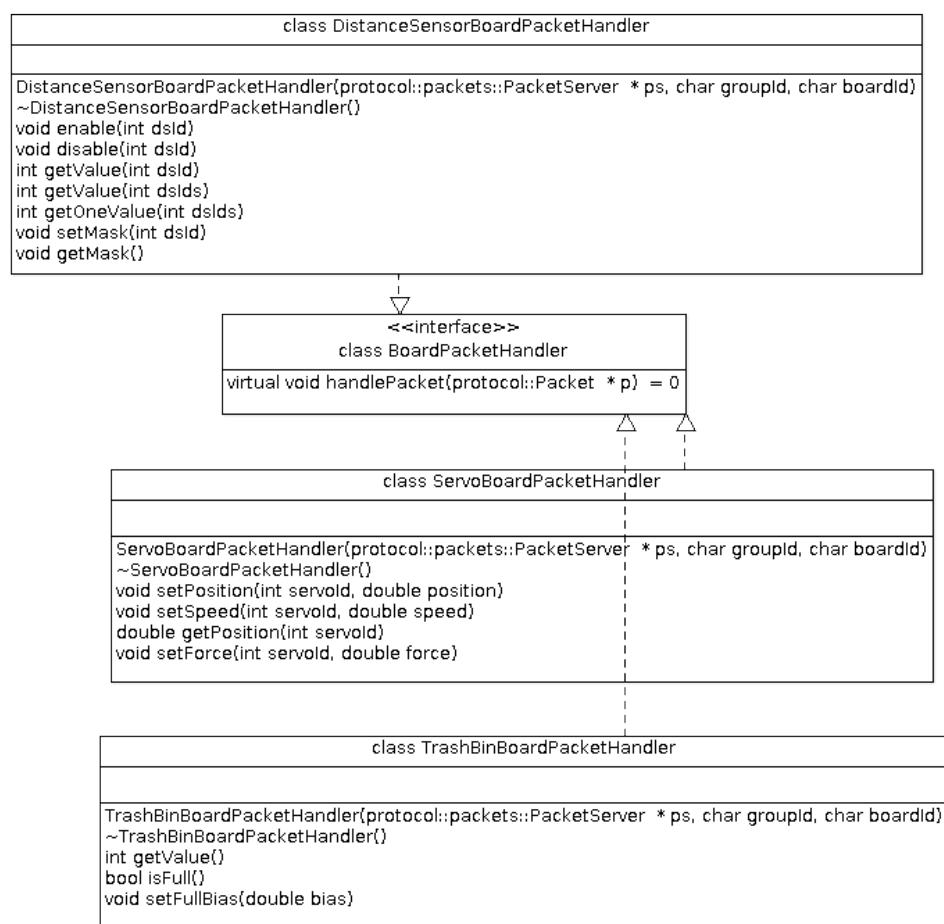


Figura 40: Diagrama de clases. Handlers de paquetes para las placas de sensores de distancia, servos y recipiente de residuos.