

# Diseño del Sistema de Software del Microcontrolador del Robot Desmalezador

Versión 1.4.1

Laura Pomponio

11 de octubre de 2022

CONICET



---

C I F A S I S

# Índice general

<b>Índice general</b>	<b>1</b>
<b>1. Introducción</b>	<b>3</b>
1.1. Acerca de este documento	3
1.2. Lectura de este documento	3
<b>2. Descripción General</b>	<b>5</b>
2.1. Estilo arquitectónico y estructura general	5
2.2. Organización de los módulos lógicos	9
2.3. Especificación funcional del sistema	10
2.3.1. Designaciones y definiciones	10
2.3.2. Statecharts	11
<b>3. Estructura de Módulos</b>	<b>16</b>
<b>4. Estructura de Herencia</b>	<b>22</b>
4.1. Unidades de medida	22
4.2. Conectores	22
4.3. Datos, algoritmos, funciones y constantes	23
4.4. Dispositivos Físicos	23
4.5. Recolectores de señales físicas	24
4.6. Sensores y buffers	24
4.7. Sistemas de control	25
4.7.1. Sistema de control de rueda	25
4.7.2. Sistema de control de dirección	26
4.8. Controlador principal, órdenes, estados y modos de operación	26
4.9. Lectura y escritura de información	27
4.10. Comandos utilizados como manejadores de señales físicas	28
4.11. Construcción de objetos	28
<b>5. Diagramas</b>	<b>29</b>
5.1. Unidades de medida	29
5.2. Connectores	29
5.3. Datos, algoritmos, funciones y constantes	31
5.4. Dispositivos Físicos	36
5.5. Recolectores de señales físicas	38
5.6. Sensores y buffers	40
5.7. Sistemas de Control	42
5.7.1. Sistema de Control de Ruedas	42
5.7.2. Sistema de Control de Dirección	46
5.8. Controlador principal, órdenes, estados y modos de operación	49
5.9. Lectura y escritura de información	54
5.10. Comandos utilizados como manejadores de señales físicas	57
5.11. Construcción de objetos del sistema	58
5.12. Programa principal	63
<b>6. Interfaces de Módulos</b>	<b>64</b>
6.1. Unidades de medida	64
6.2. Conectores	65
6.3. Datos, algoritmos, funciones y constantes	67
6.4. Dispositivos físicos	72
6.5. Recolectores de señales físicas	74

6.6.	Sensores y buffers	75
6.7.	Sistemas de control	77
6.7.1.	Sistema de control de rueda	77
6.7.2.	Sistema de control de dirección	82
6.8.	Controlador principal, órdenes, estados y modos de operación	85
6.9.	Lectura y escritura de información	92
6.10.	Comandos utilizados como manejadores de señales físicas	94
6.11.	Construcción de objetos	95
6.12.	Programa principal	98
<b>7.</b>	<b>Guía de Módulos</b>	<b>99</b>
7.1.	EXTERNALCOMMUNICATION	99
7.1.1.	CONNECTORS	99
7.1.2.	READERSWRITERS	104
7.2.	PRINCIPALCONTROLLER	107
7.2.1.	Serializable	107
7.2.2.	MainController	108
7.2.3.	ControlSystemPool	108
7.2.4.	ControllerTimeOut	110
7.2.5.	OPERATIONMODES	110
7.2.6.	STATES	112
7.3.	CONTROLSYSTEMS	118
7.3.1.	Pipe	118
7.3.2.	DIRCONTROLSYSTEM	119
7.3.3.	WHEELCONTROLSYSTEMS	124
7.4.	PHYSICALDEVICES	133
7.4.1.	Collector	133
7.4.2.	TimeCollector	133
7.4.3.	DecoCollector	133
7.4.4.	Command	134
7.4.5.	PassiveSensor	134
7.4.6.	ACTIVE	134
7.4.7.	PASSIVE	136
7.5.	CALCULATIONS	137
7.5.1.	MEASUREMENTUNITS	137
7.5.2.	INFORMATION	139
7.5.3.	OPERATIONS	144
7.6.	MCUCONSTRUCTION	147
7.6.1.	MCDirector	147
7.6.2.	MainControllerBuilder	147
7.6.3.	MCBuilder	148
7.6.4.	CSPCONSTRUCTION	150
7.7.	Main	155
<b>8.</b>	<b>Patrones de Diseño</b>	<b>157</b>
	<b>Bibliografía</b>	<b>171</b>

# Capítulo 1

## Introducción

### 1.1. Acerca de este documento

El presente documento establece el diseño del programa que se ejecutará en el microcontrolador (MCU) del robot desmalezador, de acuerdo al correspondiente documento de requerimientos [Pom22a].

Este trabajo contó con la colaboración del Dr. Maximiliano Cristiá, quien ofició de guía en cuanto a ciertas decisiones de diseño y la documentación.

### 1.2. Lectura de este documento

El diseño de software descripto en este documento comprende una descripción general que es presentada en el siguiente capítulo, la cual incluye el estilo arquitectónico utilizado y su descripción, la organización del software en términos de módulos lógicos y una especificación funcional en Statecharts, describiendo el comportamiento principal del sistema.

En los siguientes capítulos, el diseño estará dado por las siguientes cinco descripciones.

- Diagramas (o figuras).
- Estructura de herencia.
- Interfaces de módulos.
- Estructura de módulos.
- Guía de módulos.
- Descripción de patrones.

En las mencionadas descripciones aparecen los símbolos que se detallan a continuación; siendo estos, enlaces que permiten recorrer el documento de un modo no secuencial.

- MI** : enlace a la interfaz del módulo. El nombre de un módulo coloreado como **NombreModulo** también es un enlace a su interfaz.
- MG** : enlace a la guía del módulo. El nombre de un módulo coloreado como **NombreModulo** también es un enlace a la guía del mismo.
- F** : enlace a la figura o diagrama en la cual está presente el módulo.
- PD** : enlace a la descripción del patrón de diseño utilizado que involucra al módulo.

En algunos casos, los módulos están presentes en la utilización de más de un patrón de diseño. En tales casos aparecerán secuencias de enlaces como **PD1 PD2 PD3**, etc.

Por cuestiones de legibilidad, cuando se mencionan los métodos de un módulo en alguna descripción, el nombre de dicho método aparece subrayado: metodoA. En el caso en que el método no pertenezca al módulo que se está describiendo, el método estará precedido por ‘::’ y por el nombre del módulo al que pertenece. Por ejemplo, Modulo1::metodoA.

Aquellos **módulos concretos** que hereden la interfaz de una clase abstracta, solo presentarán los métodos que implementan. Todos aquellos omitidos no son implementados en el módulo.

Los **módulos abstractos**, sean padres o herederos e implementen o no algún método, presentarán todos los métodos que definen la interfaz.

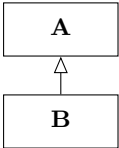
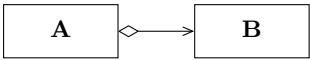
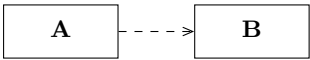
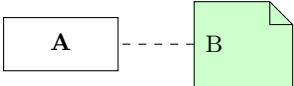
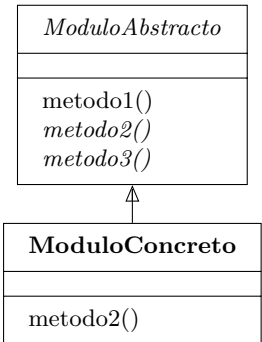
Los **constructores** de módulos concretos que no requieran argumentos ni una descripción más detallada de su funcionalidad, serán omitidos. Por el contrario, aquellos que lo requieran serán presentados con la correspondiente descripción de sus argumentos o funcionalidad.

En la sección Interfaces de Módulos, aparece en algunos casos la sentencia **private**, incluyendo algunos métodos o variables privadas. La introducción de la misma es simplemente a modo descriptivo y orientativo; a fin de contribuir a una mayor comprensión del rol del módulo en el sistema.

En la Estructura de Módulos, los **módulos físicos** son aquellos que implican algún tipo de implementación. Esto es, los **módulos abstractos** y los **módulos concretos**.

Por el contrario, los **módulos lógicos** de la Estructura de Módulos, son aquellos que agrupan los módulos que deben ser implementados, de acuerdo a un cierto criterio; y por tanto, no cuentan con una implementación. Los nombres de estos módulos lógicos son escritos con mayúsculas de este modo **MODULOLOGICO**.

En los diagramas (o figuras) se utilizan los siguientes símbolos.

Significado de los símbolos	
	B hereda de A (herencia de interfaces)
	A está compuesto por B. En el extremo de la flecha pueden aparecer los siguientes símbolos: *, n; o ninguno. En el primer caso indica que en la composición hay <b>0 o más elementos</b> , el segundo caso significa que hay <b>al menos un elemento</b> y en el último caso, la flecha sin símbolos adicionales indica que la composición tiene <b>exactamente un elemento</b> .
	A crea uno o más elemento de tipo B.
	B es una nota o descripción acerca de A.
	Los módulos <i>abstractos</i> presentan sus nombres en letra <i>itálica</i> , como así también los nombres de los métodos que no implementan. Aquellos métodos que son implementados, se presentan en letra <b>normal</b> . Por ejemplo, <i>ModuloAbstracto</i> no implementa <i>metodo2()</i> ni <i>metodo3()</i> pero sí implementa <b>metodo1()</b> . Los módulos <b>concretos</b> que hereden métodos de un padre, solo presentarán los métodos que implementen y estos serán mostrados en letra <b>normal</b> . Por ej, <b>ModuloConcreto</b> hereda toda la interfaz del padre pero solo implementa <b>metodo2()</b> .

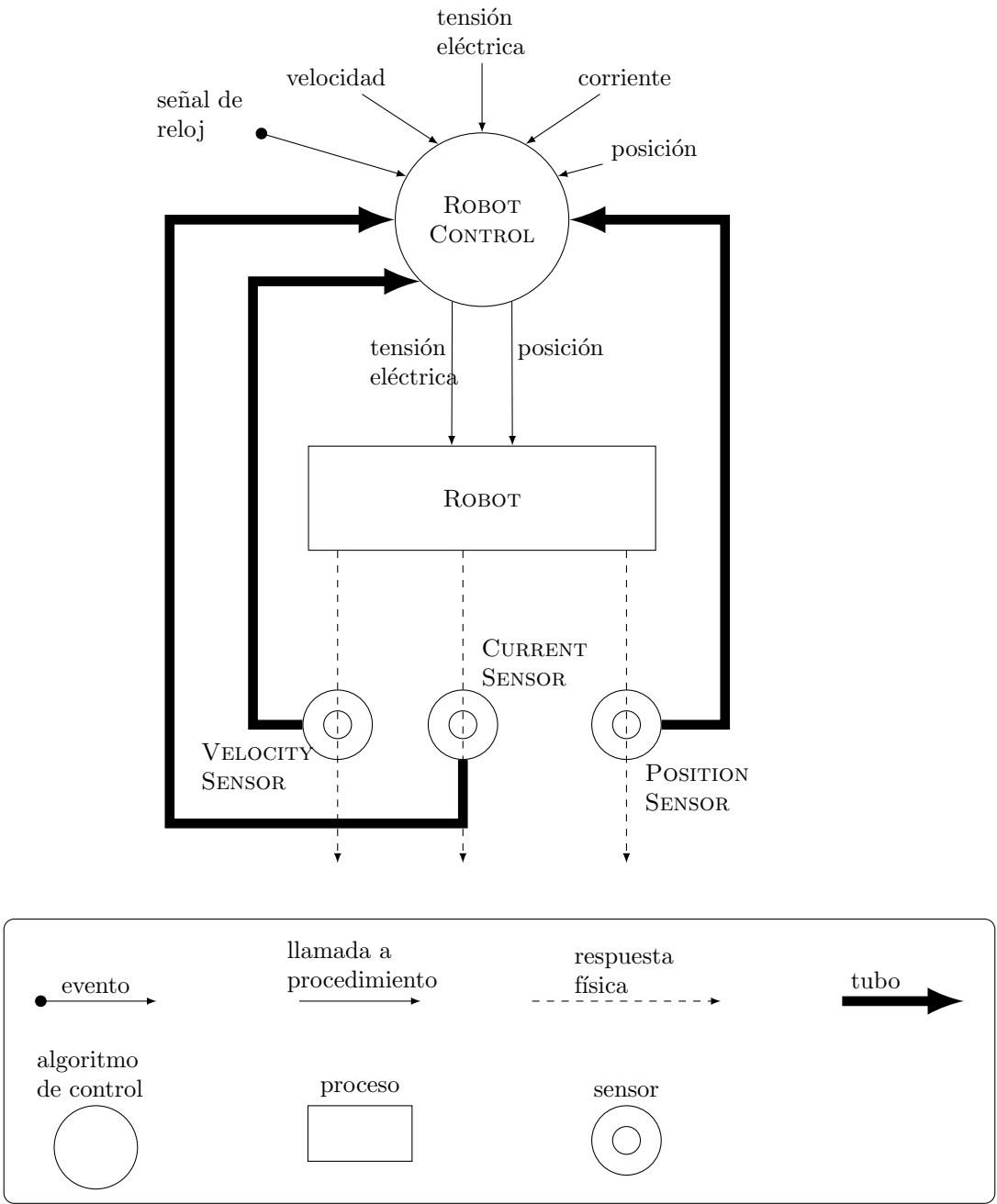
Finalmente, el documento [Pom22b] es un recurso útil para la comprensión del presente trabajo, ya que oficia como una guía que permite relacionar los requerimientos funcionales con las porciones del diseño que cumplan con estos, una vez implementado el software.

# Capítulo 2

## Descripción General

### 2.1. Estilo arquitectónico y estructura general

Figura 2.1: Diagrama canónico para el control del movimiento del robot



El diseño del sistema está basado en el estilo arquitectónico *Control de Procesos* [SG96, Cri06]. Figura 2.1 muestra el **diagrama canónico** del sistema de control diseñado.

El control se llevará a cabo en **ciclo cerrado** (*closed-loop*) de **retroalimentación hacia atrás** (*feedback*).

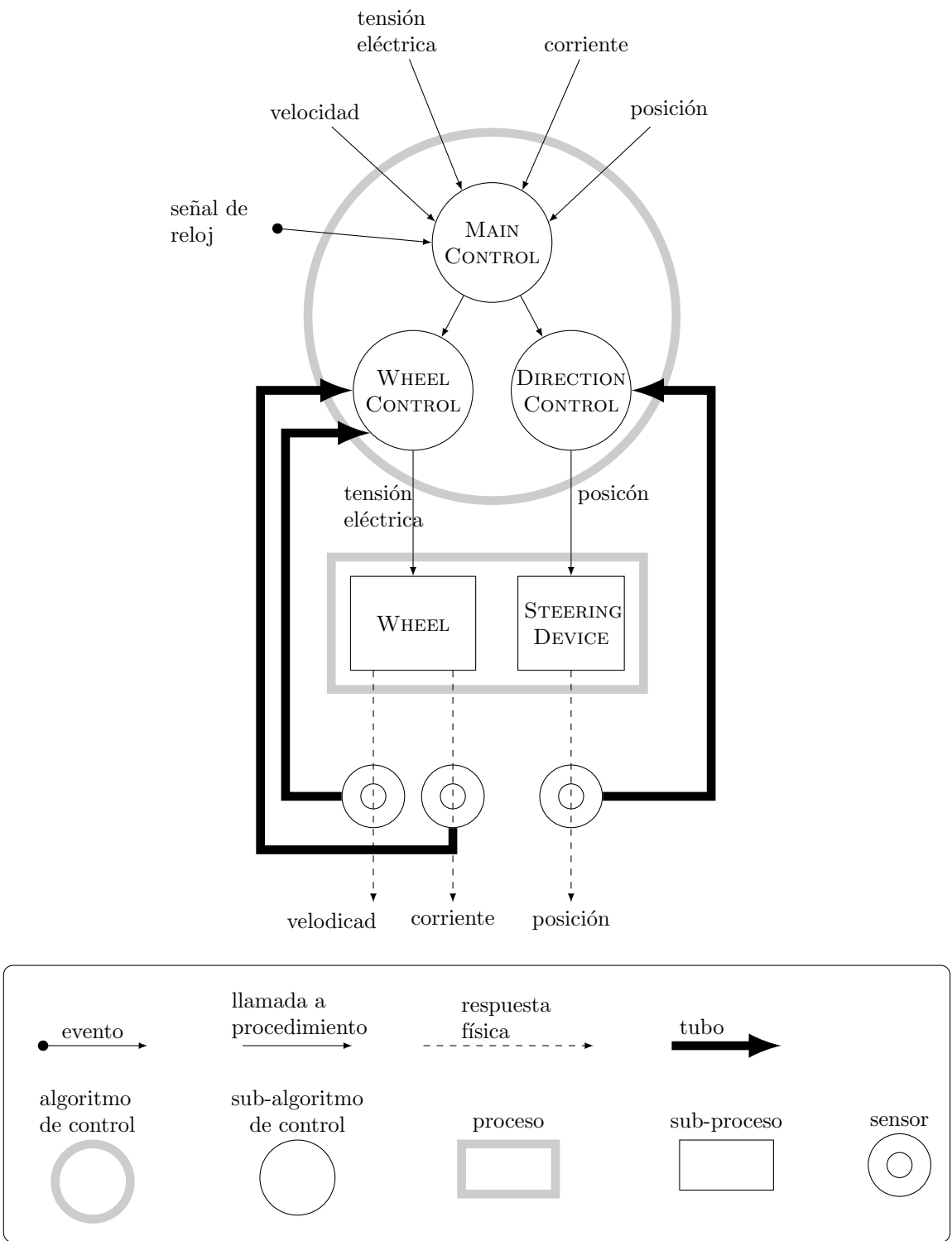
Las **variables manipuladas** (o variables del proceso) son la **tensión** a aplicar a las ruedas y la **posición** del dispositivo de giro.

Las **variables controladas** (o variables medidas) son la **velocidad** y la **corriente** de las ruedas; y la **posición** del dispositivo de giro.

Las **variables de setpoint** (valor deseado) son **velocidad**, **corriente** y **tensión** de las ruedas; y **posición** del dispositivo de giro.

Como Figura 2.1 muestra, las interrupciones de un reloj, darán inicio a un ciclo de control del robot. Esto es, el sistema obtendrá los valores deseados en el movimiento del robot, obtendrá los valores establecidos por los sensores y manipulará la tensión de las ruedas y la posición de giro para lograr los valores esperados.

Figura 2.2: Diagrama canónico de granularidad fina.



Cada elemento en el diagrama canónico de Figura 2.1 representa una porción del sistema que puede ser observada en más detalle en la Figura 2.2. Esta última muestra que el control del robot **ROBOT CONTROL** está constituido por tres tipos de sub-controles: **WHEEL CONTROL** que controla una ruedas, **DIRECCION CONTROL** que controla el dispositivo de dirección y **MAIN CONTROL** que coordina y sincroniza los dos tipos de subcontroles mencionados. Obsérvese que, en realidad el robot tiene cuatro ruedas, por tanto tendrá cuatro subsistemas de ruedas que controlar. Sin embargo, como son cuatro elementos del mismo tipo, en la representación introducimos solo uno.

Por su parte, el proceso **ROBOT** está formado por dos tipos de sub-procesos: **WHEEL** que constituye el comportamiento de una rueda y **STEERING DEVICE** que constituye el comportamiento del dispositivo de dirección.

Figura 2.1 muestra el diagrama canónico del sistema en una granularidad gruesa, mientras que Figura 2.2 lo muestra en una granularidad fina. Los elementos de la última figura agrupan diferentes porciones del sistema, de este modo, cada elemento puede ser desplegado en uno o más módulos.

Figura 2.3 presenta una visión aún más detallada, a la que se observa en Figura 2.2, desplegando los módulos que constituyen cada elemento en el diagrama canónico y graficando el funcionamiento general del sistema. En la figura, los nombres de los módulos son enlaces a los diagramas en donde los módulos aparecen.

Se describe entonces, el funcionamiento que Figura 2.3 intenta ilustrar.

El sistema contará con un temporizador **Timer** principal y uno secundario. El primero será utilizado para marcar los ciclos de control que llevará a cabo el controlador principal **MainController**, el segundo será utilizado para medir constantemente la corriente de las ruedas y para hacer girar la dirección cuando esto sea requerido.

**PC** y **CR** son la computadora y el control remoto, elementos externos al sistema que se ha diseñado. Tanto la **PC** como el **CR** enviarán órdenes al **MCU** escribiendo en los conectores **ConnectPCtoMCU** y **ConnectBufferToMCU**. Dichas órdenes serán leídas por el controlador principal **MainController** a través de llamadas a métodos de los correspondientes lectores, **SerialReader** y **BufferReader**.

El sistema contará con cuatro sistemas de control **WheelSystem**, uno para cada rueda, y un sistema de control de dirección **DirSystem**.

**WheelSystem** estará constituido por: un controlador **WheelController**; la rueda **Wheel** a controlar; el sensor de corriente de la rueda **CntSensor**, y los módulos **ReadCnt** y **ValueCollector** responsables de medir y promediar la corriente de la misma; el sensor Hall de la rueda **ActiveSensor** y los módulos **VelSensor**, **SensorCollector** y **CountSignal**, responsables de contabilizar las señales provenientes del sensor Hall.

Ante cada interrupción física del sensor **ActiveSensor**, se invocará el comando **CountSignal**. Este llamará a métodos del recolector **SensorCollector** para registrar la interrupción. Por su parte, **VelSensor** invocará métodos del recolector, cuando el controlador principal lo indique, para obtener información sobre las interrupciones.

A su vez, ante cada interrupción física del temporizador secundario, se invocará el comando **ReadCnt**. Este llamará a métodos del recolector **ValueCollector** para medir y registrar la corriente de la rueda. Por su parte **CntSensor** invocará métodos del recolector, cuando el controlador principal lo indique, para obtener la corriente medida.

El sistema de control de dirección **DirSystem** estará conformado por un controlador **DirController**, el dispositivo de dirección **SteeringDevice** a controlar, el sensor de dirección **DirSensor** y el comando **DirCtrlTimeOut** que permitirá llevar a cabo el giro del dispositivo.

El temporizador secundario **SecondTimer**, marcará interrupciones cada  $1,5ms$ . Ante cada una de estas interrupciones se ejecutarán dos comandos: **ReadCnt** mencionado más arriba y **DirCtrlTimeOut**. Este último indicará al sensor de dirección **DirSensor** que escriba su valor en el correspondiente conector **Pipe** y al controlador **DirController** que gire el dispositivo. Dependiendo de en qué estado se encuentre este controlador, omitirá o no la orden de giro. Solo responderá ante la mencionada orden si su estado es **DTurning**, y en tal caso, enviará al dispositivo de dirección (si corresponde) un pulso de giro.

Por su parte, el temporizador principal **FirstTimer** emitirá interrupciones físicas cada  $100ms$ , a fin de dar inicio a un ciclo de control de todo el sistema. Ante cada interrupción se ejecutará el comando **ControllerTimeOut**, el cual le indicará al controlador principal **MainController** que debe iniciar un nuevo ciclo de control. Cuando esto ocurre, el controlador solicita la lectura de las órdenes provenientes de la **PC** y del **CR**, a los lectores **SerialReader** y **BufferReader**. Luego, lleva a cabo el control indicando primero a los sensores **VelSensor**, **CntSensor** y **DirSensor**, que escriban sus mediciones en los correspondientes conectores **Pipe**. Después, indica a los controladores **WheelController** y **DirController** de cada subsistema, que deben iniciar sus controles. Ambos controladores actúan sobre sus dispositivos, tanto ruedas como el dispositivo de dirección, invocando comandos y otros módulos intermedios. Finalmente, el controlador principal envía a la **PC** la información correspondiente respecto de los controles; esto lo hace, mediante el escribiente **SerialWriter**.



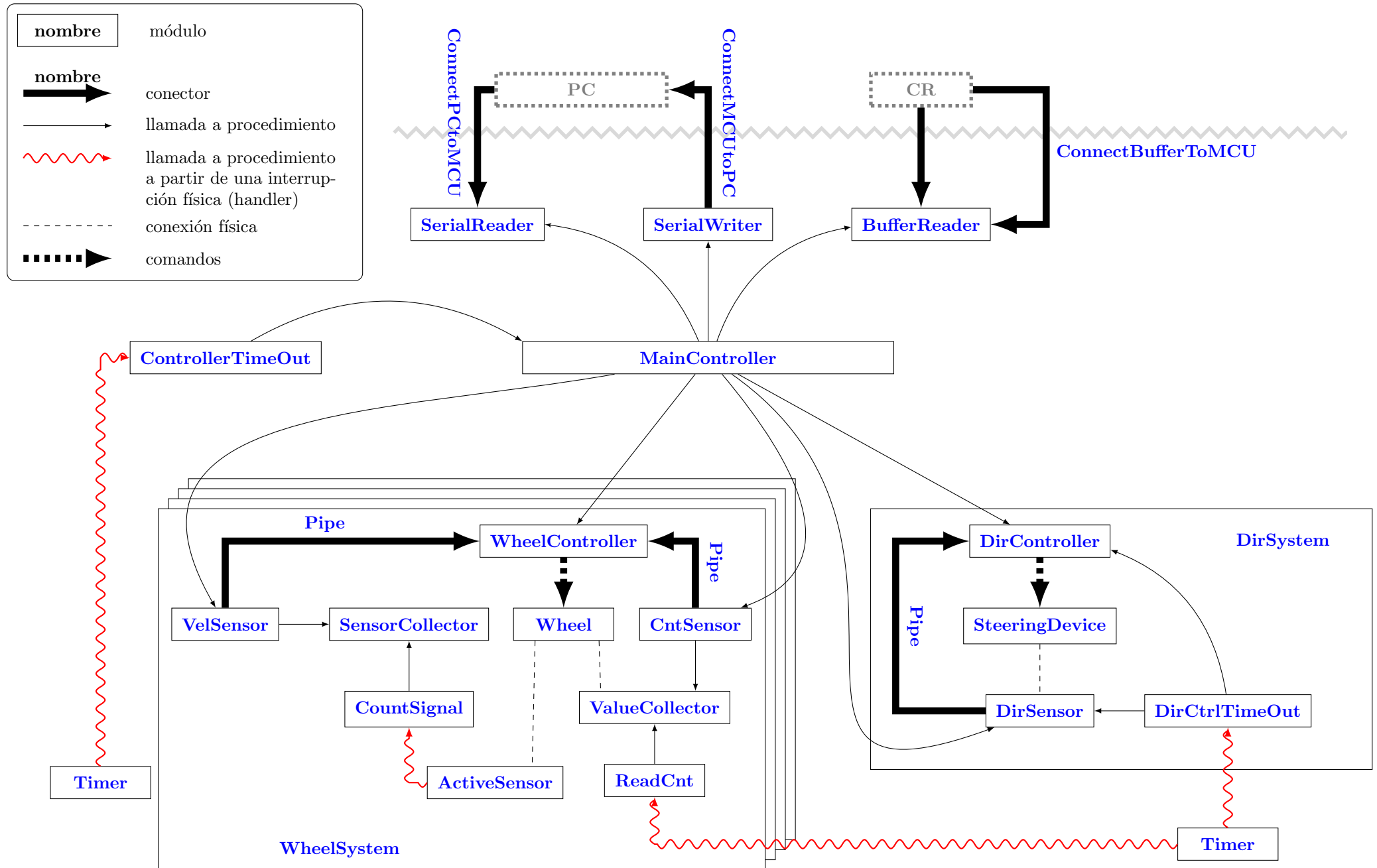


Figura 2.3: Diagrama general del sistema del MCU

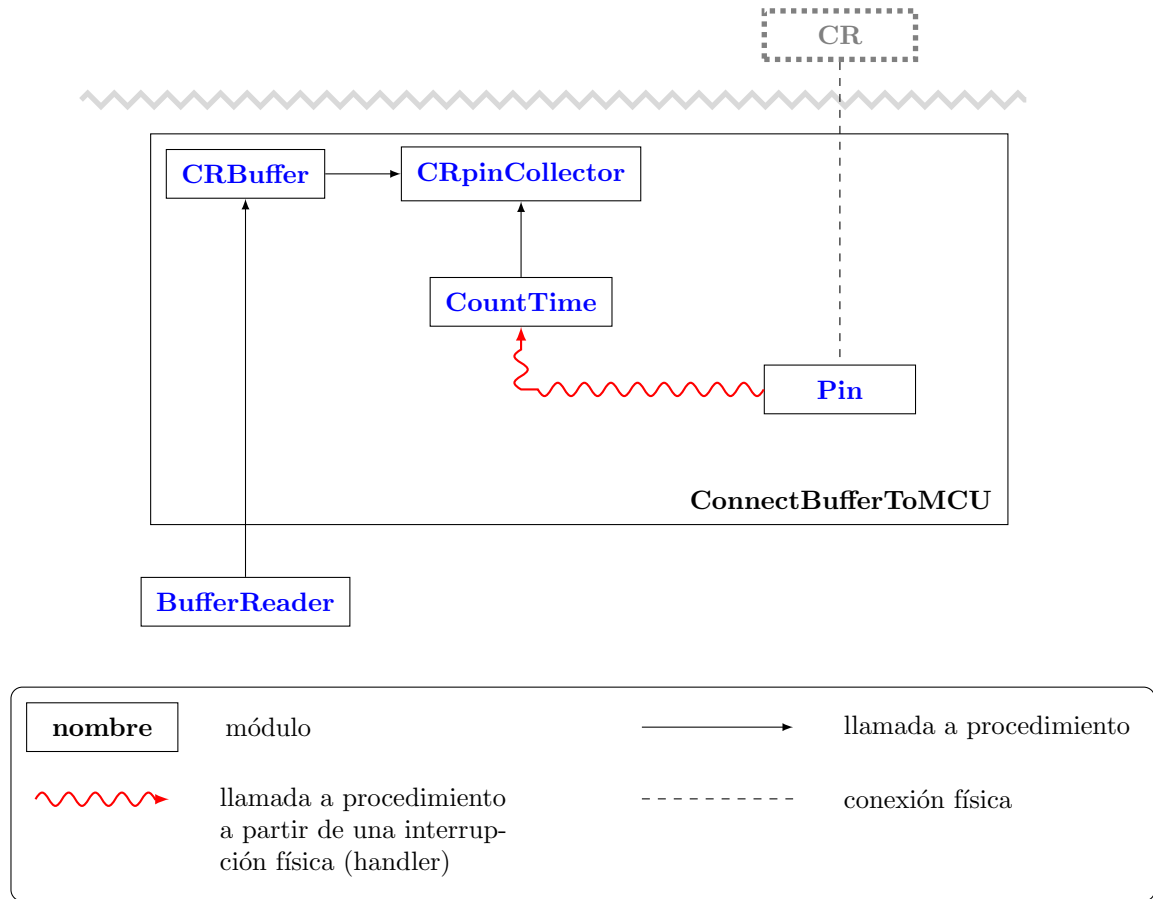


Figura 2.4: Componente conector **ConnectBufferToMCU**

El **CR** enviará las órdenes de velocidad y de dirección por medio de dos pines receptores. De este modo, el conector **ConnectBufferToMCU** que vincula el **CR** con el lector **BufferReader**, será un componente complejo constituido por varios módulos; los cuales se ilustran en Figura 2.4. Cuando el gatillo del **CR** es presionado, el pin asociado emitirá una interrupción física en el sistema. Ante dicha interrupción el sistema invocará el comando **CountTime** que tomará registro de dicha interrupción invocando al recolector de señales **CRpinCollector**. Cuando el lector **BufferReader** solicite el valor registrado al buffer **CRBuffer**, este último pedirá los valores correspondientes al recolector, hará ciertos cálculos y devolverá el valor de lectura. Este mecanismo de envío de una orden de velocidad, se replica para el caso del pin de dirección. Por tal motivo, dado que se trata de elementos del mismo tipo, en la figura aparece solo un pin, un comando y un recolector.

La descripciones previas son informales y orientativas, a fin de contar con una visión general del comportamiento del sistema.

## 2.2. Organización de los módulos lógicos

Figura 2.5: Organización de los subsistemas (módulos lógicos) del MCU

EXTERNALCOMMUNICATION	CALCULATIONS	MCUCONSTRUCTION
PRINCIPALCONTROLLER		
CONTROLSYSTEMS		
PHYSICALDEVICES		

El sistema está organizado en los módulos lógicos ilustrados en Figura 2.5. Algunos de estos módulos están ubicados en capas, indicando que hay una organización en la cual algunos módulos implementan porciones del sistema de más bajo nivel de abstracción, que otros.

Esta organización no debe confundirse con el estilo arquitectónico de *Sistemas en Capas* [SG96, Cri06], ya que no respeta un invariante del destilo en cuanto a la visibilidad de un estrato respecto a otro. En particular, PRINCIPALCONTROLLER conoce módulos de EXTERNALCOMMUNICATION. Sin embargo, la figura es orientativa y su objetivo es tener en cuenta que ciertas porciones del sistema refieren a elementos de más bajo nivel de abstracción que otros.

Los nombres de cada módulo lógico en la figura son enlaces a la Estructura de Módulos.

## 2.3. Especificación funcional del sistema

En esta sección se presenta una especificación funcional del sistema, realizada en Statecharts.

### 2.3.1. Designaciones y definiciones

- Pulso de reloj que le indica al controlador principal que debe iniciar el control  $\approx tick$
- Llega un carácter, que no indica el fin del mensaje, desde la PC a la conexión en serie  $\approx char$
- Llega un carácter, que indica fin de mensaje (“\n”), desde la PC a la conexión en serie  $\approx endMsg$
- La secuencia de caracteres enviados desde la PC a la conexión en serie es leída por el sistema  $\approx readS$ .
- Una orden proveniente del CR es escrita en el buffer asociado a éste  $\approx writeCR$
- Una orden proveniente del CR es leída por el sistema, del buffer correspondiente  $\approx readB$
- El sistema cambia su modo de funcionamiento a modo PC  $\approx modePC$
- El sistema cambia su modo de funcionamiento a modo CR  $\approx modeCR$
- El sistema interpreta la secuencia de caracteres proveniente de la PC y determina si hay un cambio de modo  $\approx readPCMsg$
- El sistema interpreta la secuencia de caracteres proveniente de la PC y determina los *setpoints* establecidos por esta  $\approx readPCOrd$
- El sistema interpreta la orden proveniente del CR y determina los *setpoints* establecidos por esta  $\approx readCROrd$
- $readOrder \stackrel{\text{def}}{=} (start(readPCOrd); stopped(readPCOrd)) \vee (start(readCROrd); stopped(readCROrd))$   
Descripción: el sistema interpreta una orden (secuencia de caracteres) proveniente del CR o de la PC.
- El sistema no pudo leer un mensaje proveniente del CR o la PC  $\approx noMsg$
- Cantidad máxima de ciclos de reloj que el sistema esperará por una orden  $\approx MAX$
- Los sensores realizan sus mediciones; en particular, los sensores de velocidad de cada rueda y el sensor de posición del dispositivo de dirección  $\approx sensorsWrite$
- El sistema detecta que las dos ruedas delanteras tienen velocidad nula  $\approx twoWNull$
- El sistema detecta que solo una de las ruedas delanteras tiene velocidad nula  $\approx oneWNull$
- El sistema detecta que ninguna de las ruedas delanteras tiene velocidad nula  $\approx twoWNotNull$
- Los controladores de ruedas y dispositivo de dirección, realizan sus controles (comparar, calcular y setear)  $\approx controllersControl$
- $controlRobot \stackrel{\text{def}}{=} sensorsWrite; (twoWNull \vee oneWNull \vee twoWNotNull); controllersControl$   
Descripción: el sistema lleva a cabo el control del robot (medir, comparar, calcular y setear).
- El sistema envía a la PC, los resultados de las mediciones y de los cálculos realizados  $\approx sendMsg$
- El sistema detiene las ruedas del robot  $\approx stopRobot$
- El sensor de posición del dispositivo de dirección realiza su medición  $\approx dSensorSignal$
- Se enciende el dispositivo de dirección  $\approx on$
- Se apaga el dispositivo de dirección  $\approx off$

- Estado del mensaje recibido en cada ciclo de control, indicando si hay o no una orden de detener el robot  $\approx$  Order
- Estado del error de giro (diferencia entre el *setpoint* de giro y la posición del dispositivo de dirección) respecto al giro mínimo, que determina que se debe girar  $\approx$  TurnError
  - Estado de error de giro, mayor o igual al giro mínimo  $\approx E \geq \text{MinTurn}$
  - Estado de error de giro, menor al giro mínimo  $\approx E < \text{MinTurn}$
- Estado de velocidad (nula o no) de la rueda delantera izquierda  $\approx$  WV1
- Estado de velocidad (nula o no) de la rueda delantera derecha  $\approx$  WV2
- Estado de error de posición (diferencia entre el *setpoint* de giro y la posición del dispositivo de dirección) respecto al valor mínimo, que determina que se alcanzó la posición deseada  $\approx$  PosError
  - Estado en que el dispositivo de dirección aún no alcanzó la posición deseada  $\approx EP > \text{Min}$
  - Estado en el que el dispositivo de dirección alcanzó la posición deseada  $\approx EP \leq \text{Min}$

### 2.3.2. Statecharts

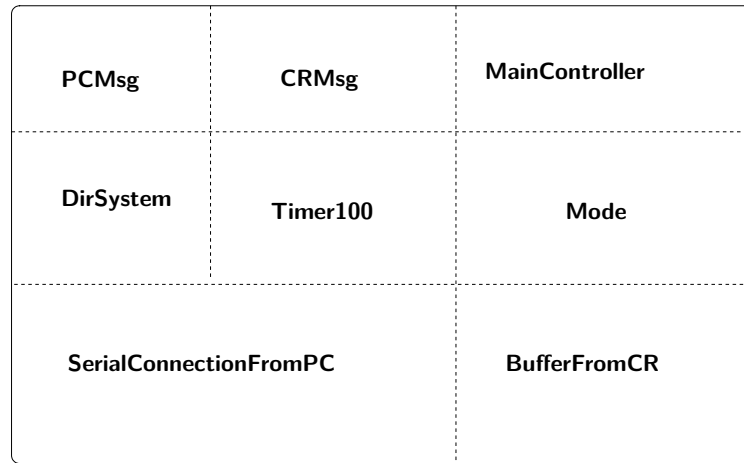


Figura 2.6: Robot - Nivel 0

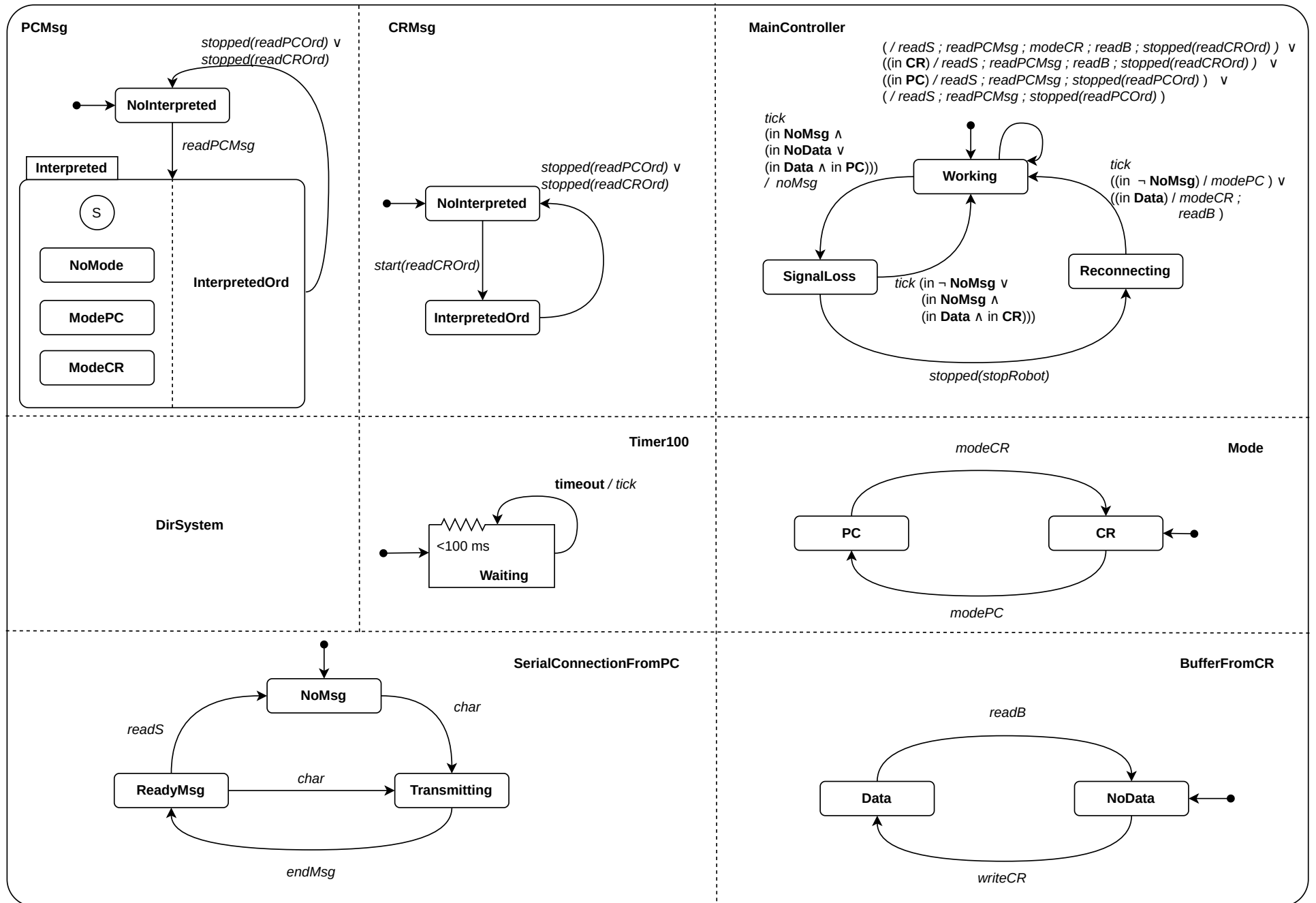


Figura 2.7: Robot - Nivel 1

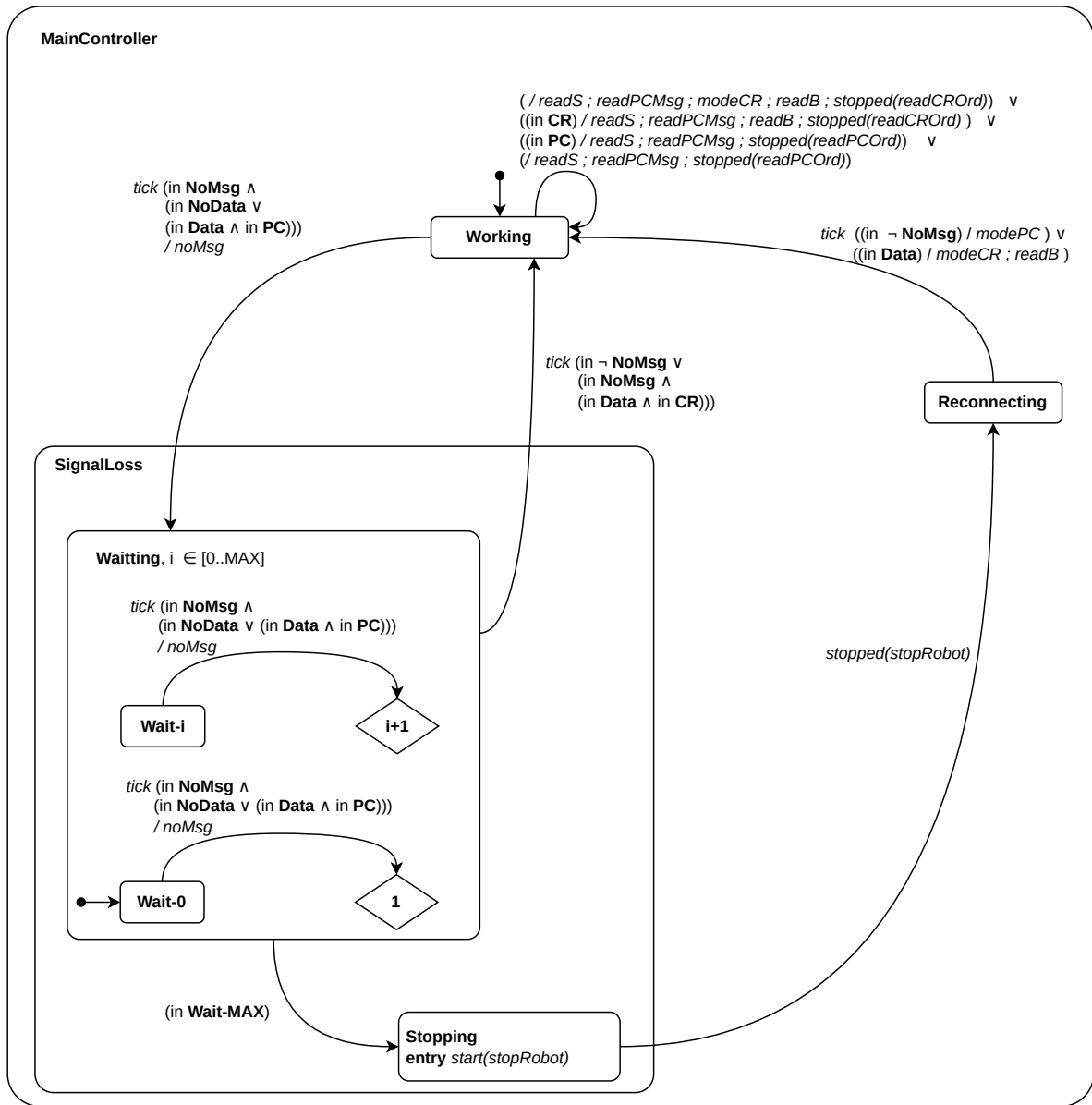


Figura 2.8: Controlador principal del robot - Nivel 2

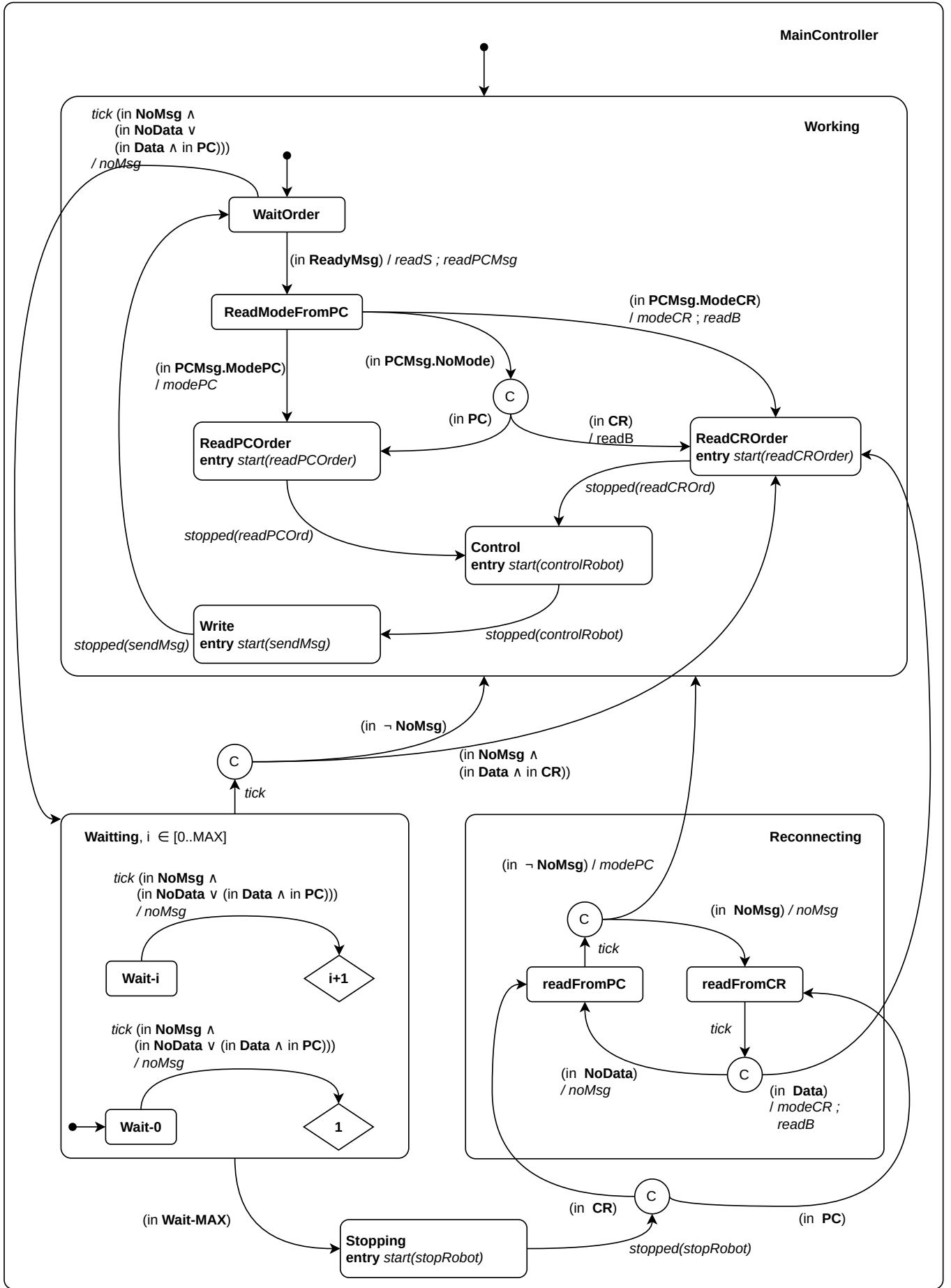


Figura 2.9: Controlador principal del robot - Nivel 3

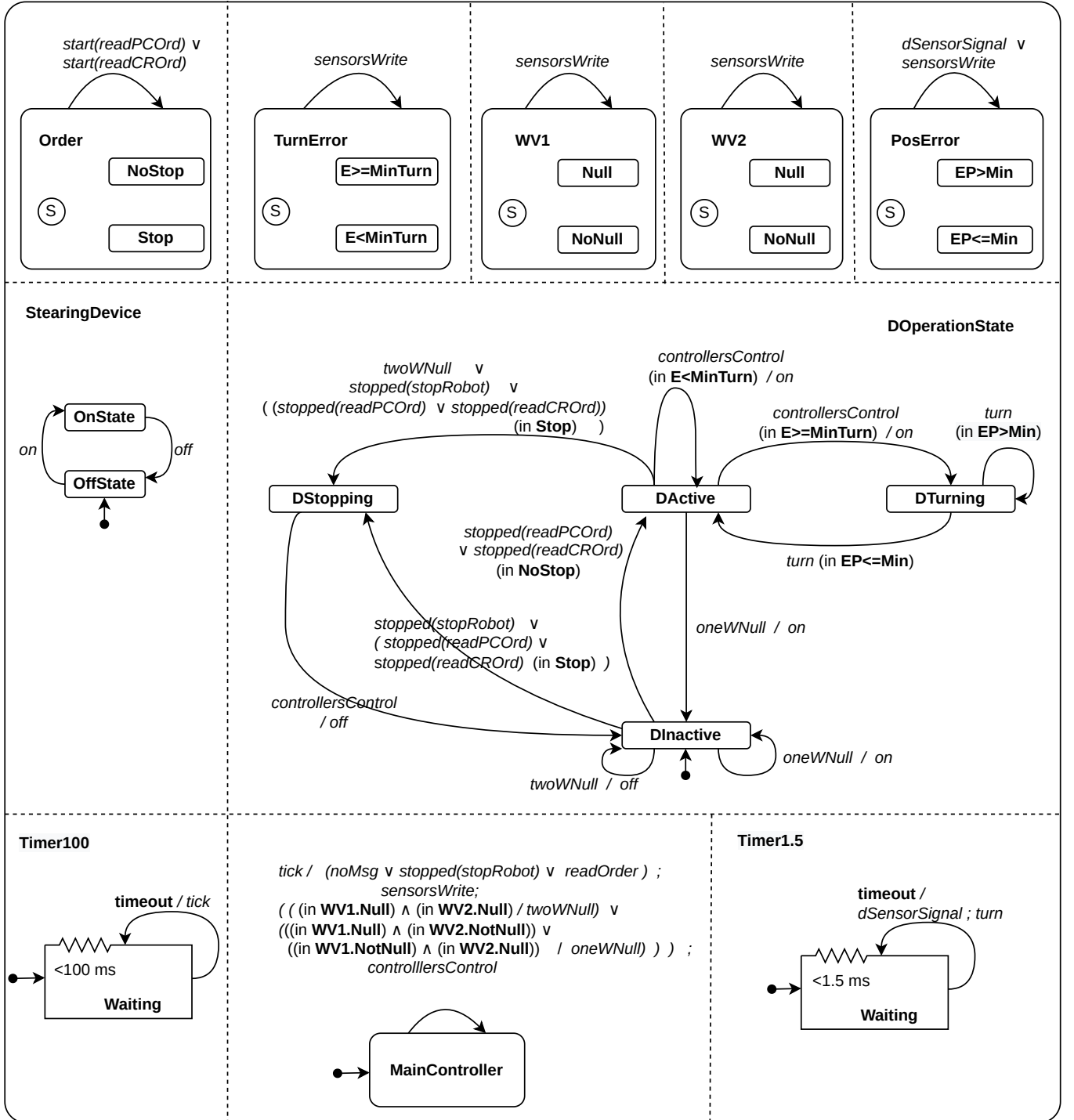
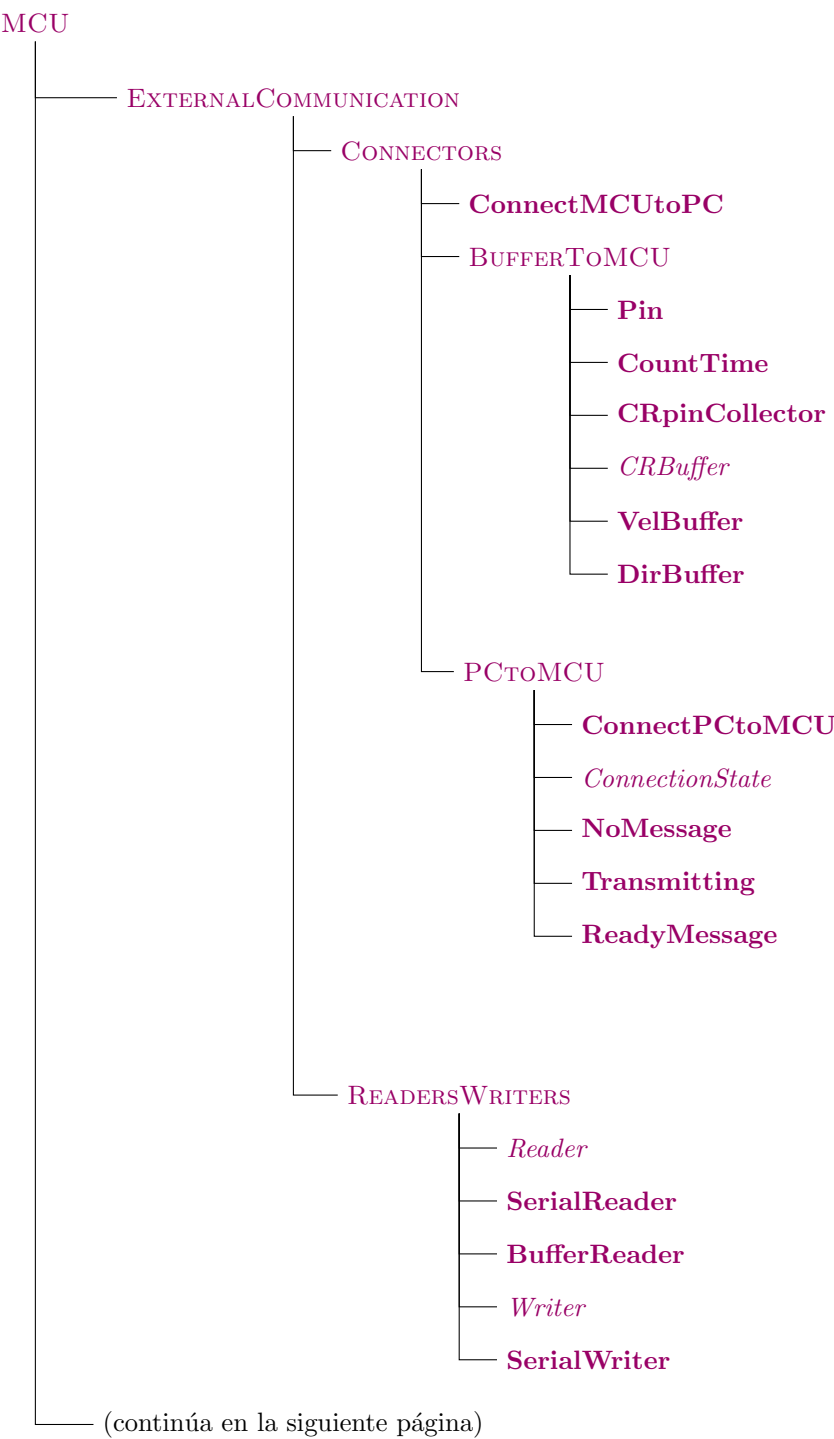


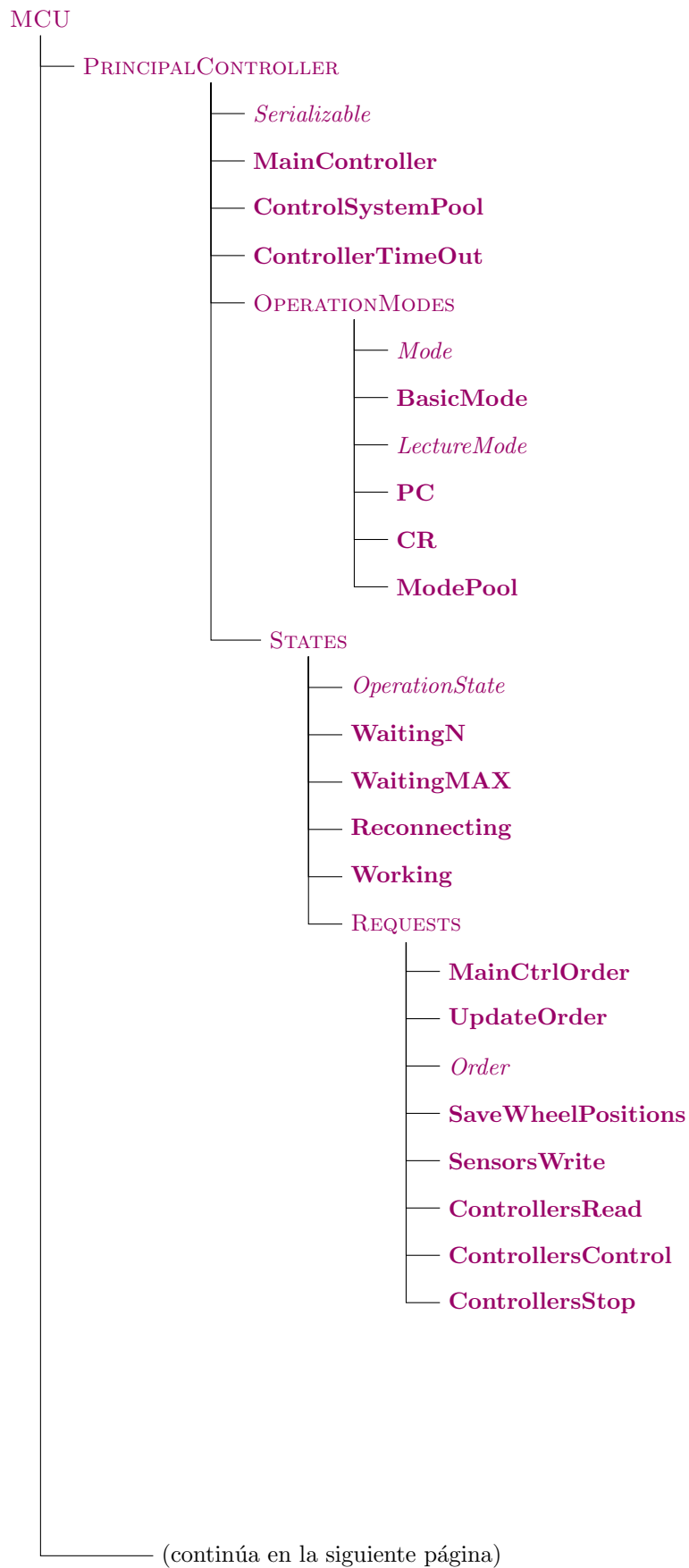
Figura 2.10: Sistema de dirección DirSystem - Nivel 1

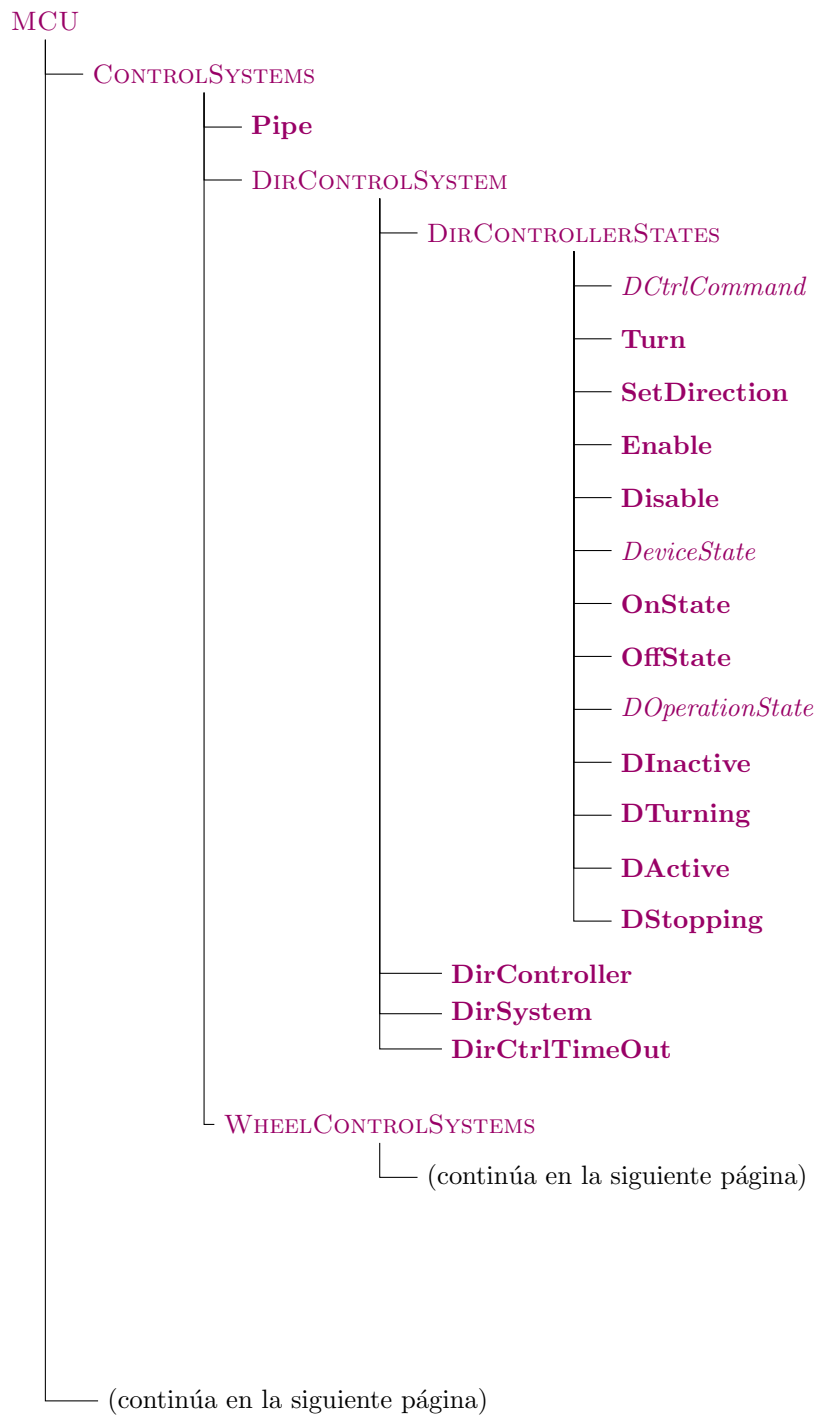


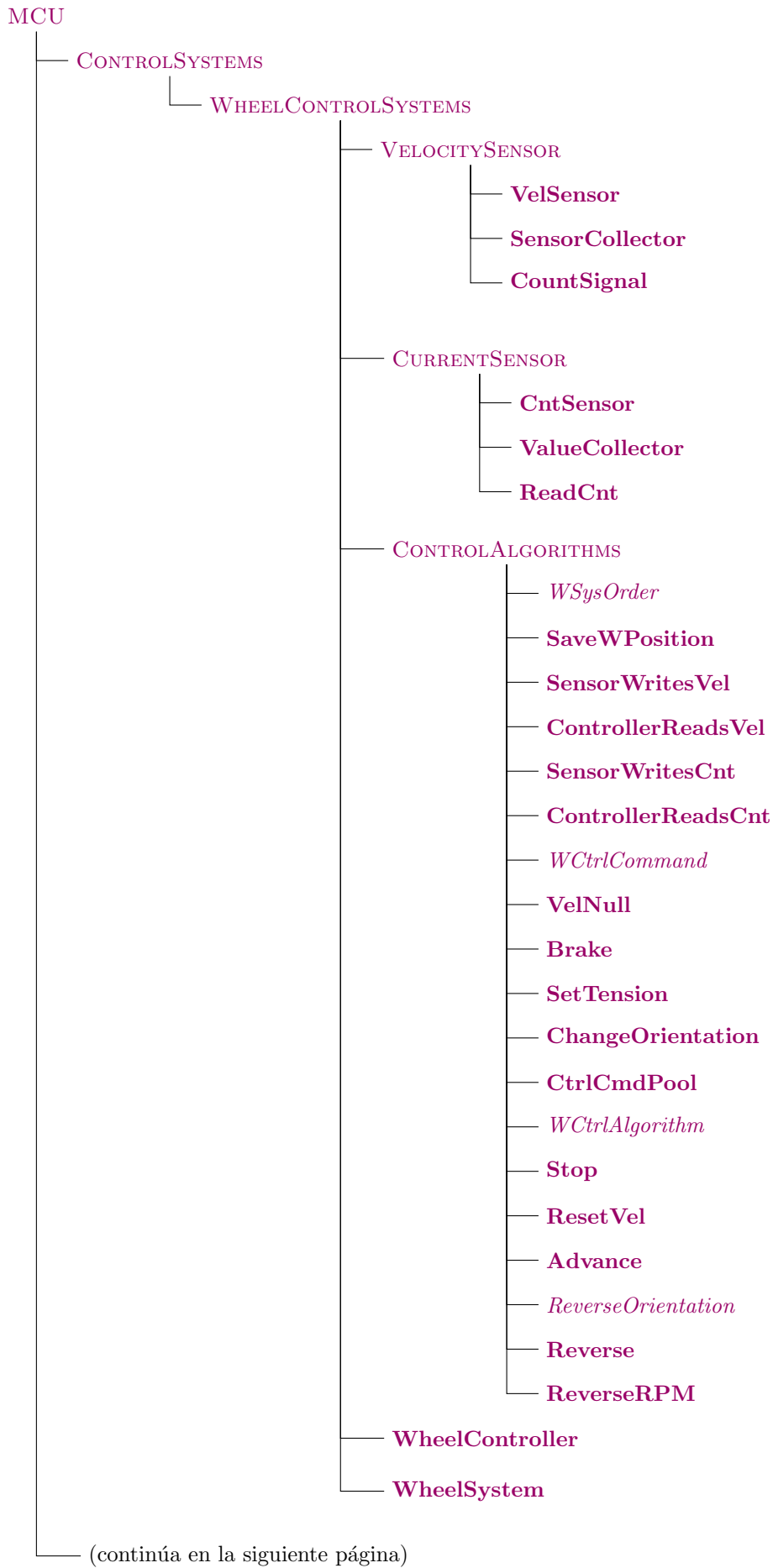
# Capítulo 3

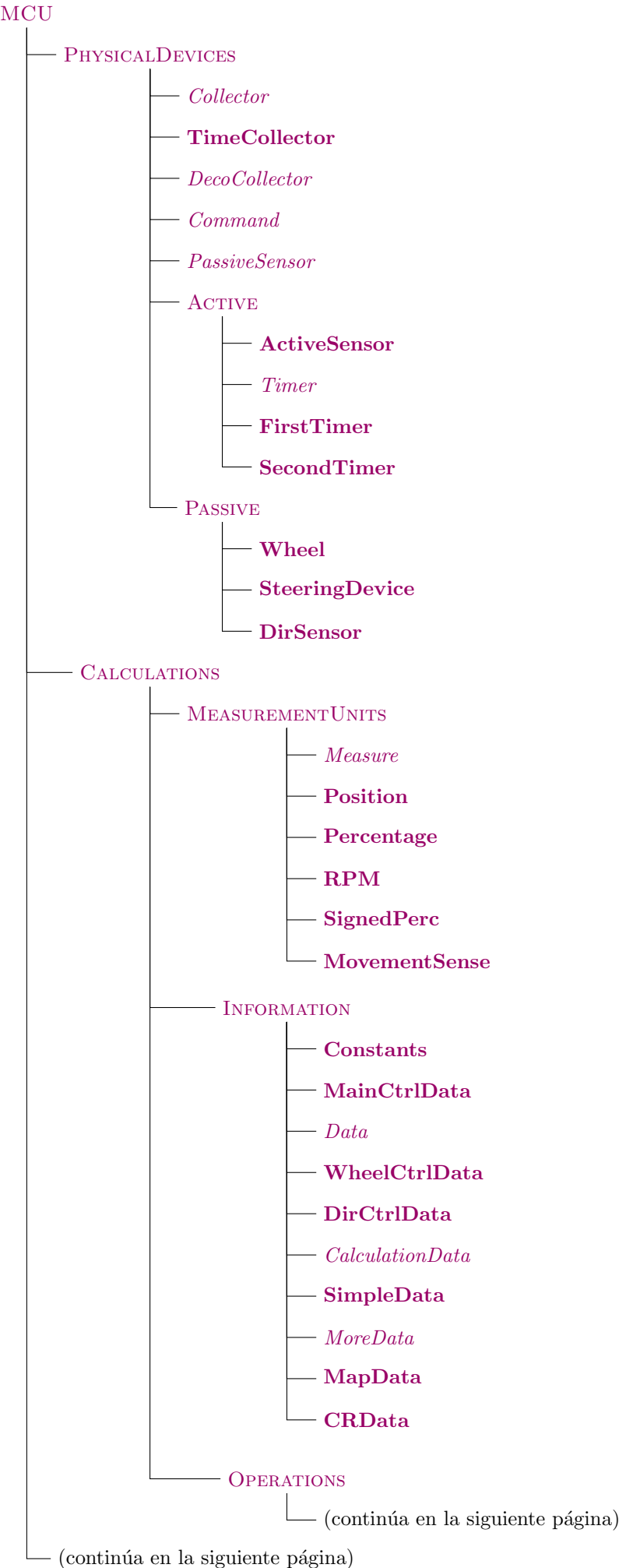
## Estructura de Módulos

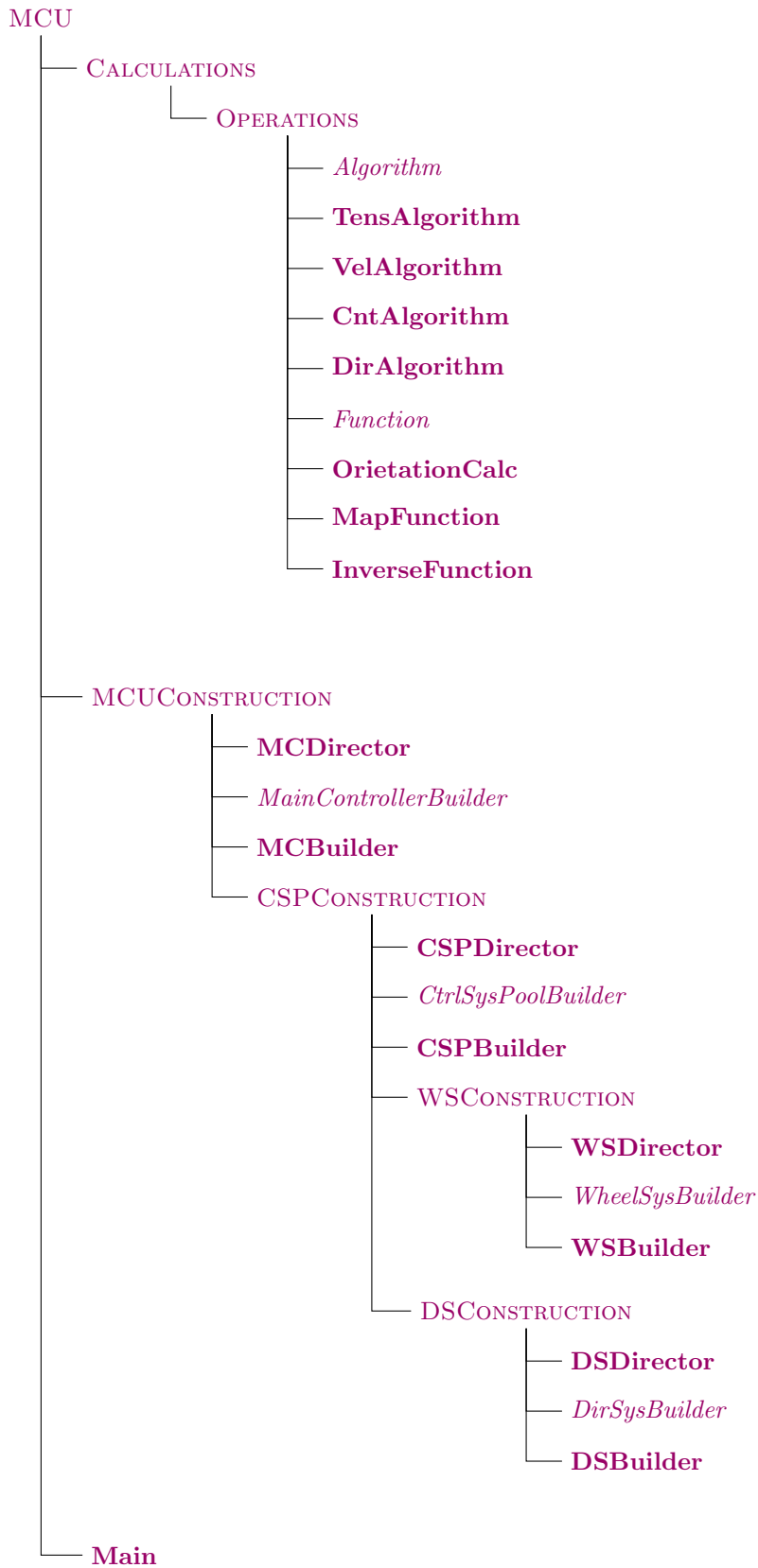












# Capítulo 4

## Estructura de Herencia

### Descripción

En esta estructura se presenta la relación *hereda-de* entre los módulos. Cabe aclarar que en este diseño cuando se habla de herencia, se está refiriendo a herencia de interfaces.

En esta sección están incluidos todos los módulos del sistema, incluso aquellos que no heredan de ningún módulo. De este modo, hacemos explícita la ausencia de herencia de éstos.

Los nombres de los módulos son un vínculo a la Especificación de Interfaces. El símbolo [F](#) es un vínculo al diagrama que los ilustra.

### 4.1. Unidades de medida

<i>Measure</i>	
—	<b>Position</b>
—	<b>Percentage</b>
—	<b>RPM</b>
—	<b>SignedPerc</b>
—	<b>MovementSense</b>
	<a href="#">F</a>

### 4.2. Conectores

<b>Pipe</b>	<a href="#">F</a>
<b>ConnectMCUtoPC</b>	<a href="#">F</a>
<b>ConnectPCtoMCU</b>	<a href="#">F</a>
<i>ConnectionState</i>	
—	<b>NoMessage</b>
—	<b>ReadyMessage</b>
—	<b>Transmitting</b>
	<a href="#">F</a>

---

### 4.3. Datos, algoritmos, funciones y constantes

**MainCtrlData** F

---



**Constants** F

---



### 4.4. Dispositivos Físicos

**Wheel** F

---

**SteeringDevice** F

---



<i>Timer</i> <ul style="list-style-type: none"> <li><b>FirstTimer</b></li> <li><b>SecondTimer</b></li> </ul>	F
--	---

<b>Pin</b>	F
------------	---

#### 4.5. Recolectores de señales físicas

<i>Collector</i> <ul style="list-style-type: none"> <li><b>TimeCollector</b></li> <li><i>DecoCollector</i> <ul style="list-style-type: none"> <li><b>CRpinCollector</b></li> <li><b>SensorCollector</b></li> </ul> </li> </ul>	F
--	---

<b>ValueCollector</b>	F
-----------------------	---

#### 4.6. Sensores y buffers

<i>PassiveSensor</i> <ul style="list-style-type: none"> <li><b>VelSensor</b></li> <li><b>CntSensor</b></li> <li><b>DirSensor</b></li> </ul>	F
---	---

<b>ActiveSensor</b>	F
---------------------	---

<i>CRBuffer</i> <ul style="list-style-type: none"> <li><b>VelBuffer</b></li> <li><b>DirBuffer</b></li> </ul>	F
--	---

4.7. Sistemas de control

4.7.1. Sistema de control de rueda

<div><div><i>WSysOrder</i></div><div><div></div><div>SaveWPosition</div><div>SensorWritesVel</div><div>ControllerReadsVel</div><div>SensorWritesCnt</div><div>ControllerReadsCnt</div></div></div>	F
<div><div><i>WCtrlCommand</i></div><div><div></div><div>VeNull</div><div>Brake</div><div>SetTension</div><div>ChangeOrientation</div></div></div>	F
<div><div>CtrlCmdPool</div></div>	F
<div><div><i>WCtrlAlgorithm</i></div><div><div></div><div>Stop</div><div>ResetVel</div><div>Advance</div><div>ReverseOrientation<div><div></div><div>Reverse</div><div>ReverseRPM</div></div></div></div></div>	F
<div><div>WheelController</div></div>	F
<div><div>WheelSystem</div></div>	F

4.7.2. Sistema de control de dirección

<div><i>DCtrlCommand</i><ul style="list-style-type: none"><li>Turn</li><li>SetDirection</li><li>Enable</li><li>Disable</li></ul></div>	F
<div><i>DeviceState</i><ul style="list-style-type: none"><li>OnState</li><li>OffState</li></ul></div>	F
<div><i>DOperationState</i><ul style="list-style-type: none"><li>DInactive</li><li>DTurning</li><li>DActive</li><li>DStopping</li></ul></div>	F
<div>DirController</div>	F
<div>DirSystem</div>	F

4.8. Controlador principal, órdenes, estados y modos de operación

<div><i>Mode</i><ul style="list-style-type: none"><li>BasicMode</li><li>LectureMode<ul style="list-style-type: none"><li>PC</li><li>CR</li></ul></li></ul></div>	F
<div>ModePool</div>	F

*Order*

- **SaveWheelPositions**
- **SensorsWrite**
- **ControllersRead**
- **ControllersControl**
- **ControllersStop**

F

**UpdateOrder**

F

**MainCtrlOrder**

F

*OperationState*

- **WaitingN**
- **WaitingMAX**
- **Reconnecting**
- **Working**

F

*Serializable*

- **MainController**
- **ControlSystemPool**

F

## 4.9. Lectura y escritura de información

*Reader*

- **SerialReader**
- **BufferReader**

F

*Writer*

- **SerialWriter**

F

4.10. Comandos utilizados como manejadores de señales físicas

<i>Command</i> <ul style="list-style-type: none"><li>DirCtrlTimeOut</li><li>CountTime</li><li>CountSignal</li><li>ReadCnt</li><li>ControllerTimeOut</li></ul>	F
---	---

---

4.11. Construcción de objetos

WSDirector	F
------------	---

---

<i>WheelSysBuilder</i> <ul style="list-style-type: none"><li>WSBuilder</li></ul>	F
--	---

---

DSDirector	F
------------	---

---

<i>DirSysBuilder</i> <ul style="list-style-type: none"><li>DSBuilder</li></ul>	F
--	---

---

CSPDirector	F
-------------	---

---

<i>CtrlSysPoolBuilder</i> <ul style="list-style-type: none"><li>CSPBuilder</li></ul>	F
--	---

---

MCDirector	F
------------	---

---

<i>MainControllerBuilder</i> <ul style="list-style-type: none"><li>MCBuilder</li></ul>	F
--	---

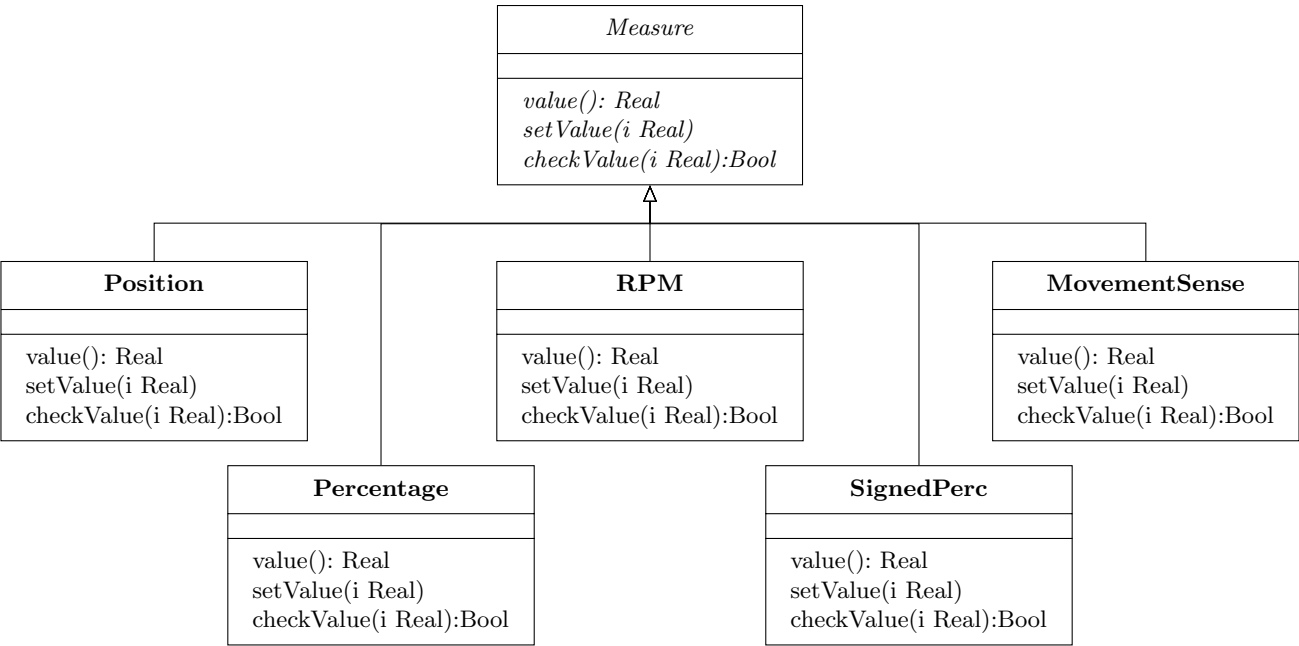
---

# Capítulo 5

## Diagramas

### 5.1. Unidades de medida

Figura 5.1: Unidades de medidas  
*Measure Position Percentage RPM SignedPerc MovementSense*



### 5.2. Conectores

Figura 5.2: Tubo. Conector interno de los sistemas de control  
*Pipe*

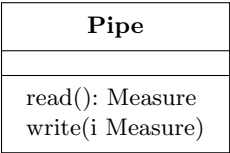


Figura 5.3: Conector hacia el exterior del sistema. Conecta el MCU a la PC  
*ConnectMCUtoPC*

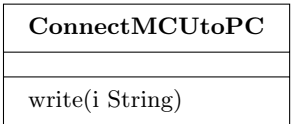
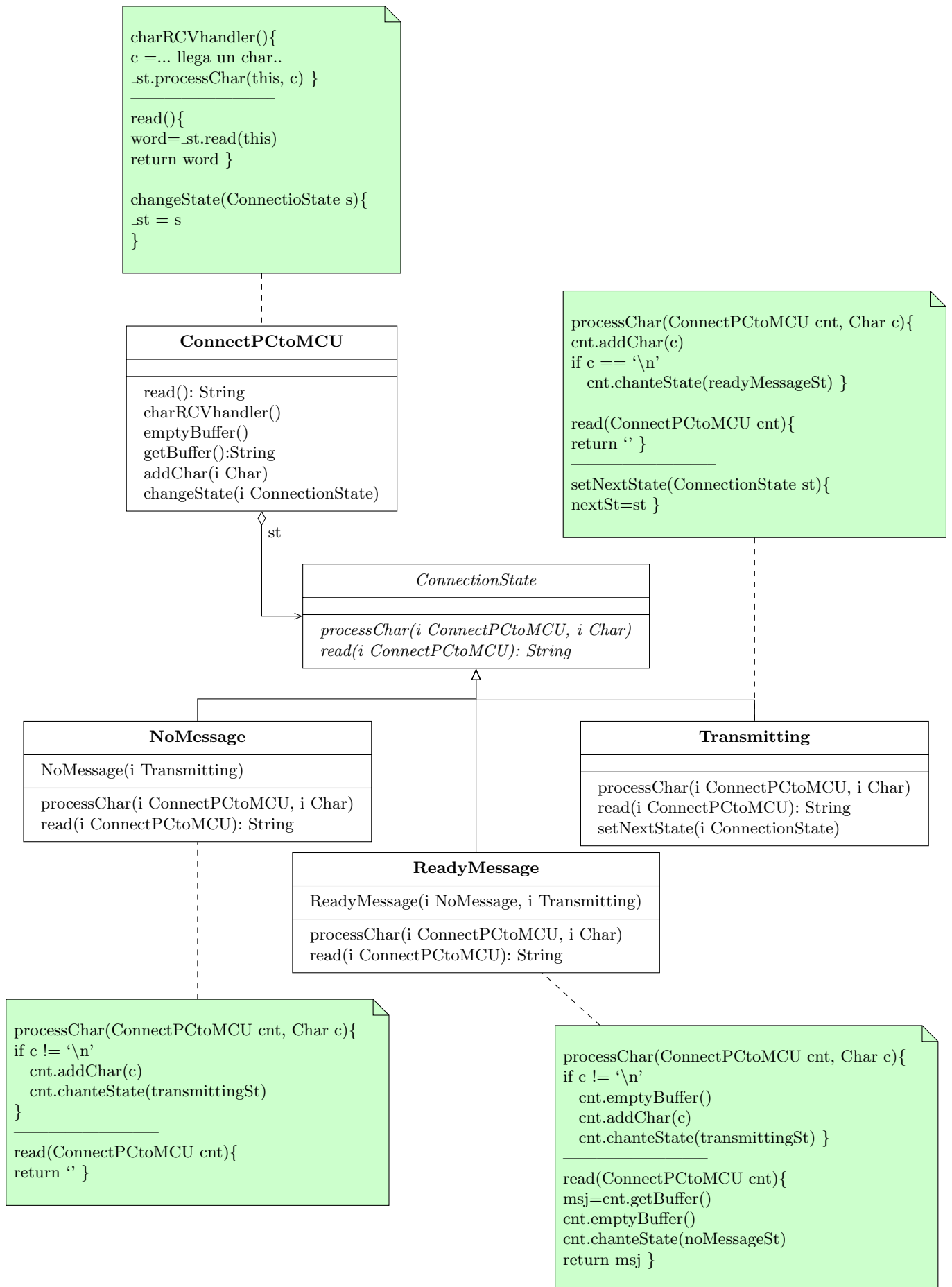


Figura 5.4: Conector desde el exterior del sistema. Conecta la PC al MCU. Patrón Estado  
**ConnectPCtoMCU** *ConnectionState* *NoMessage* *ReadyMessage* *Transmitting*



### 5.3. Datos, algoritmos, funciones y constantes

Figura 5.5: Datos del controlador principal  
*MainCtrlData*

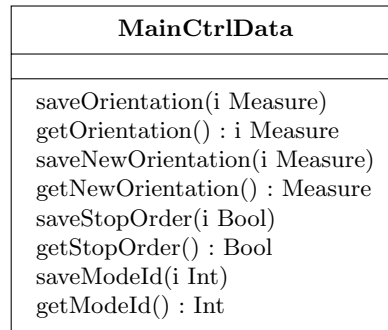


Figura 5.6: Datos del control de rueda y del control de dirección  
*Data WheelCtrlData DirCtrlData*

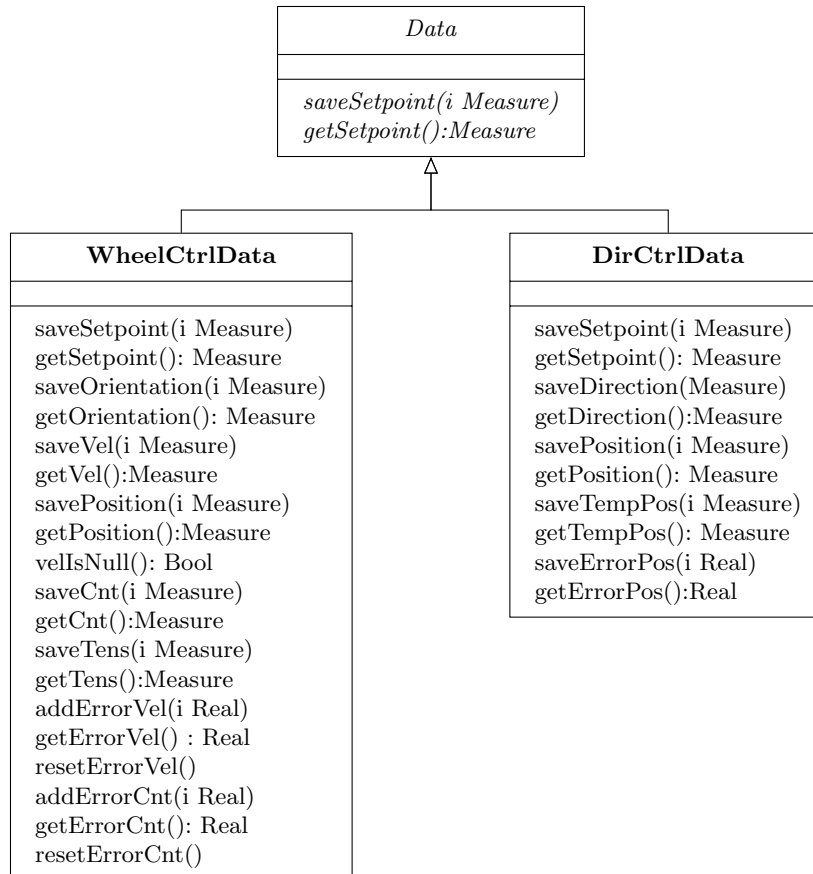




Figura 5.7: Algoritmos de control de ruedas y de dirección. Patrón Estrategia  
*Algorithm TensAlgorithm VelAlgorithm CntAlgorithm DirAlgorithm*

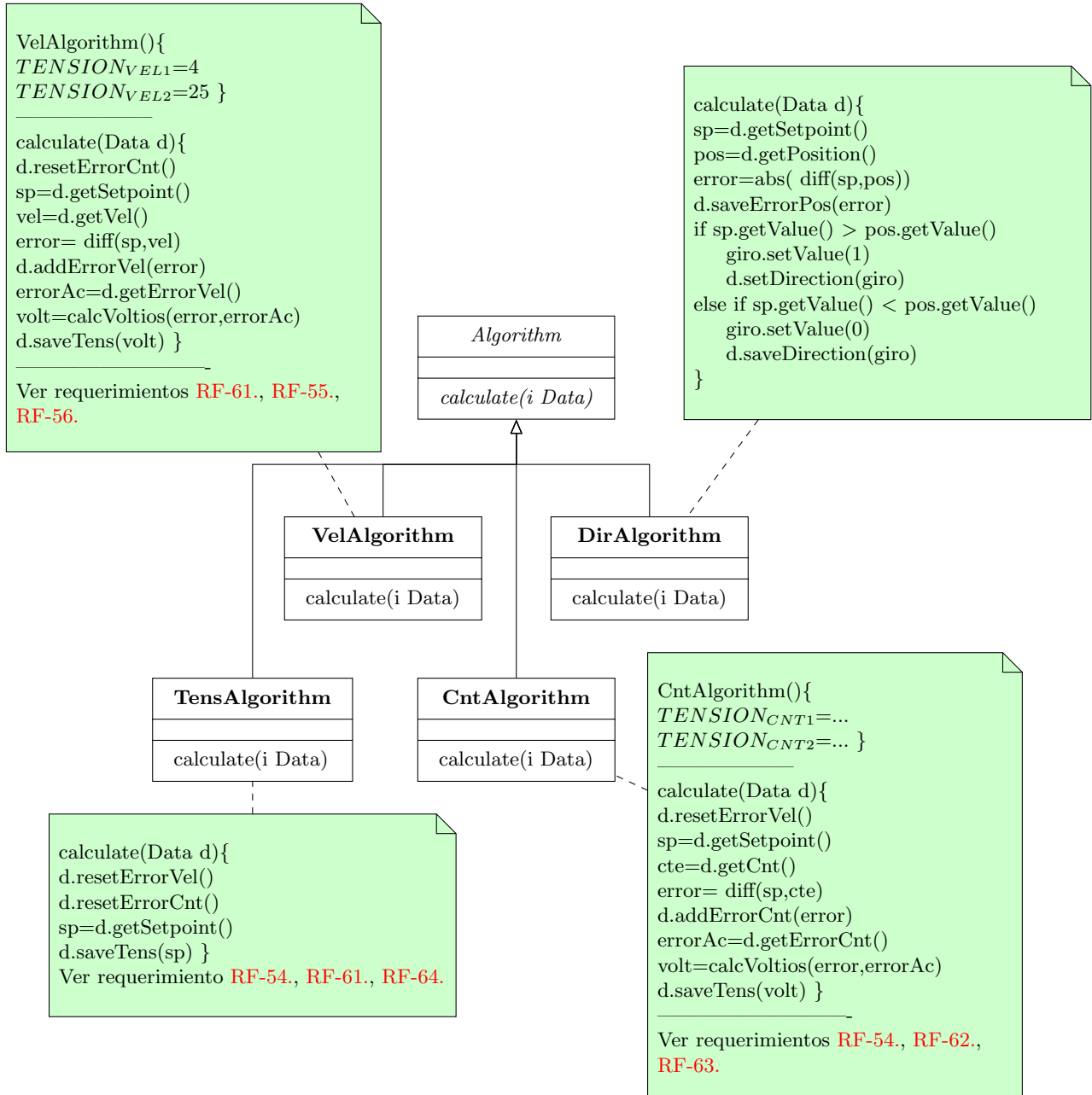


Figura 5.8: Constantes del sistema. Esta clase es un singleton  
**Constants**

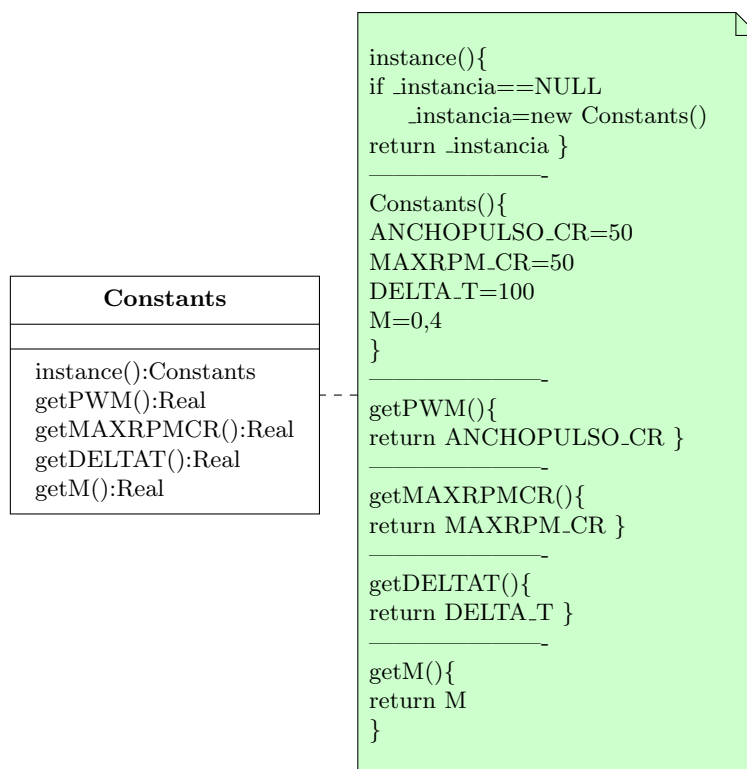


Figura 5.9: Argumentos de funciones que llevarán a cabo cálculos requeridos por el sistema.  
*CalculationData SimpleData MoreData MapData CRData*. Patrón Decorador

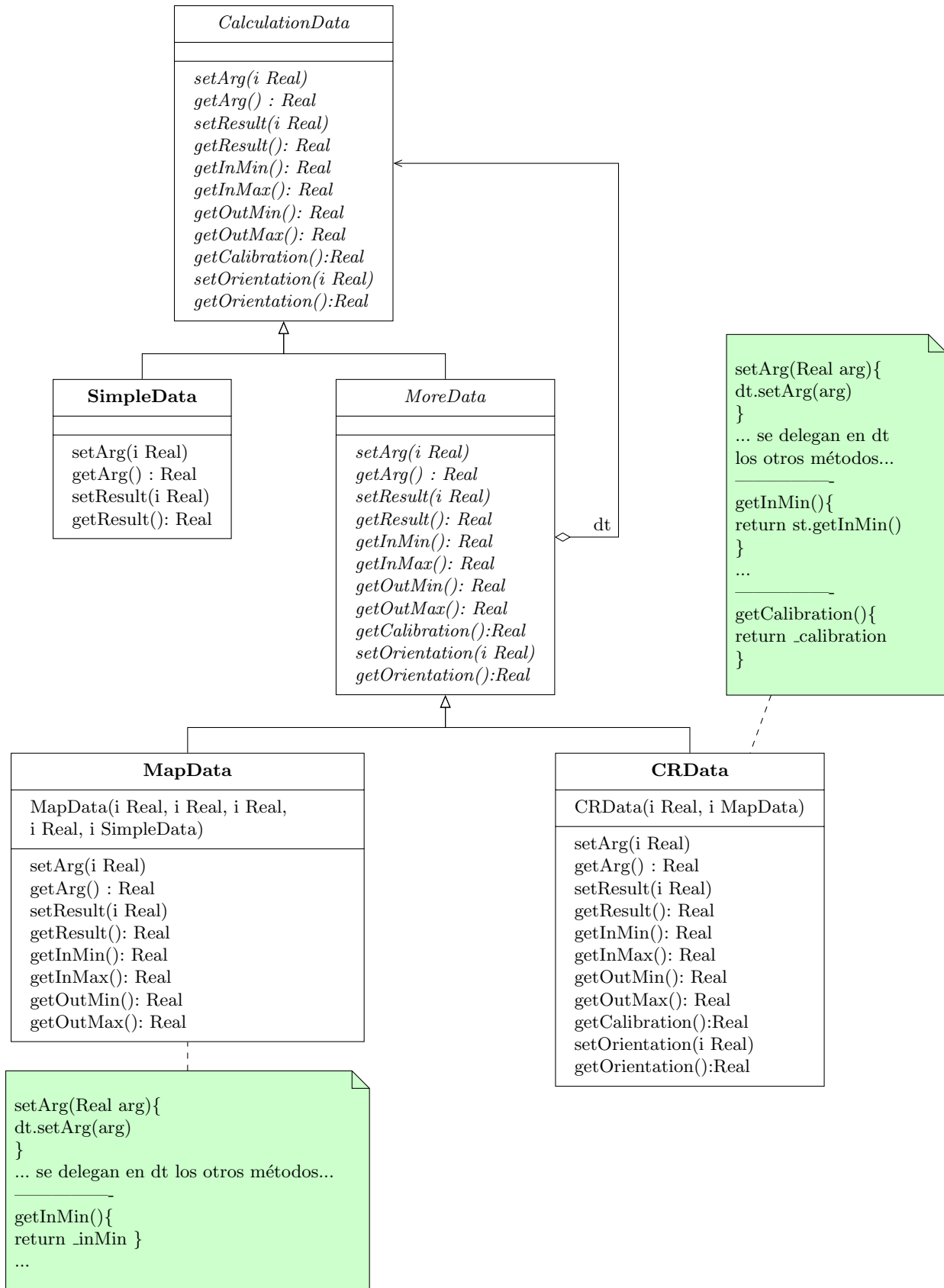
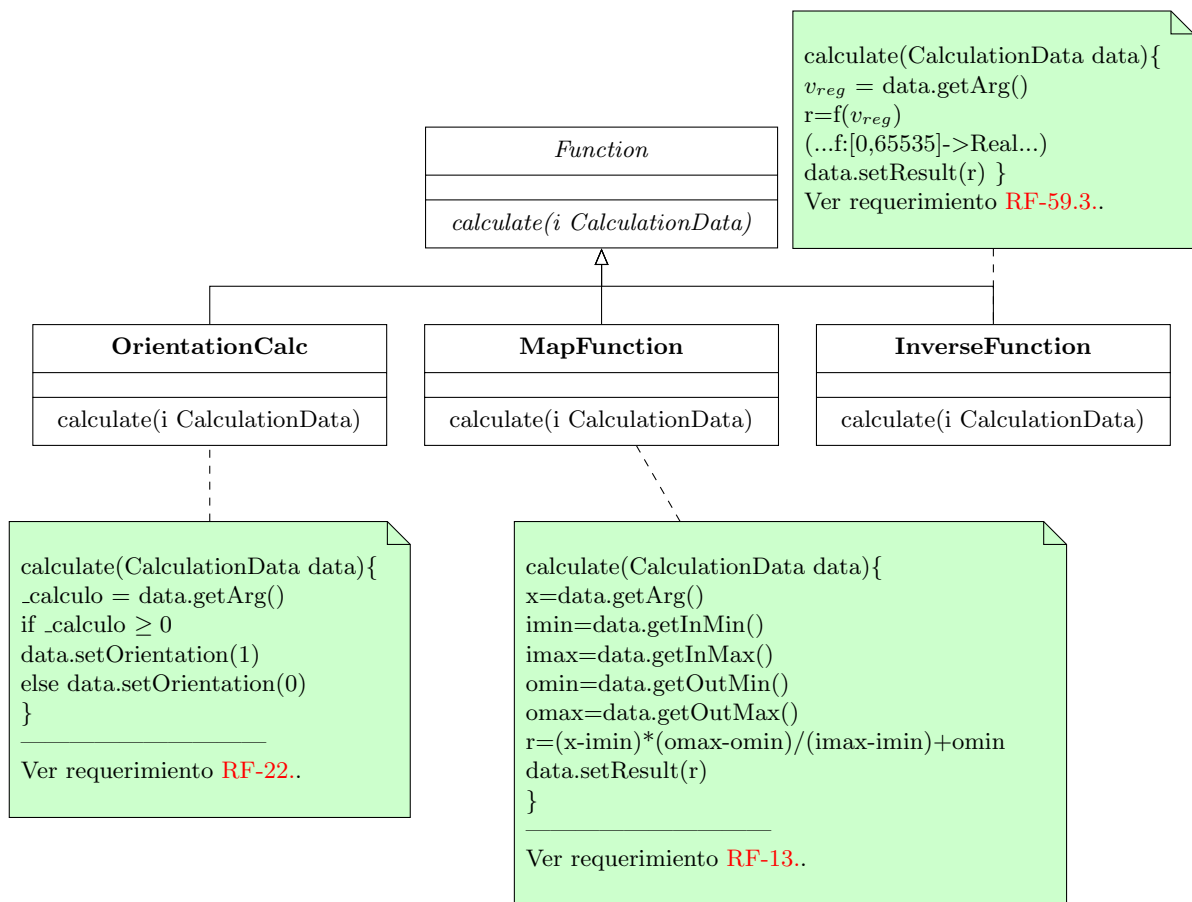


Figura 5.10: Funciones. Patrón Estrategia  
*Function* *OrientationCalc* *MapFunction* *InverseFunction*



5.4. Dispositivos Físicos

Figura 5.11: Rueda  
Wheel

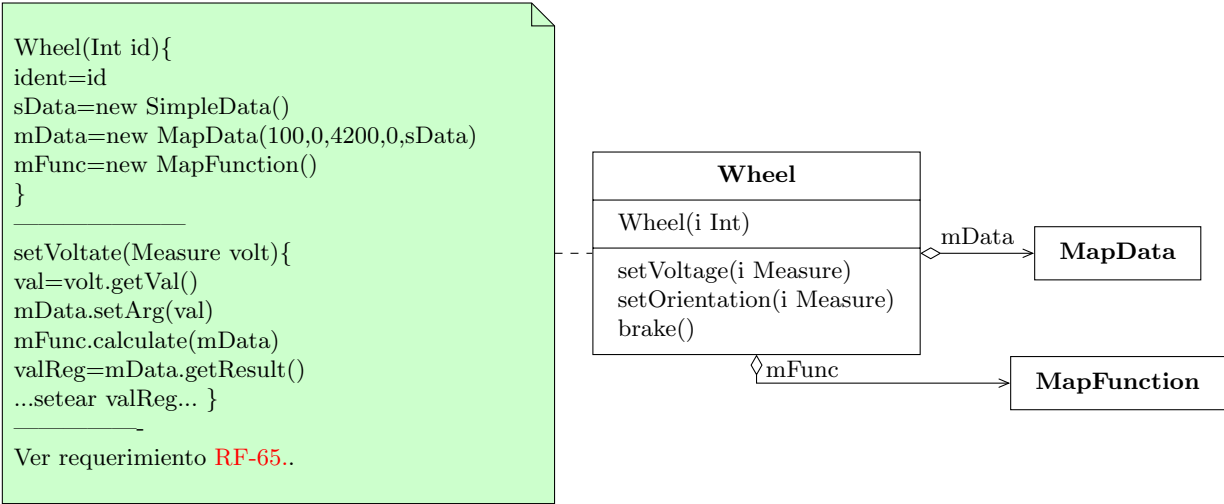


Figura 5.12: Dispositivo de dirección  
SteeringDevice

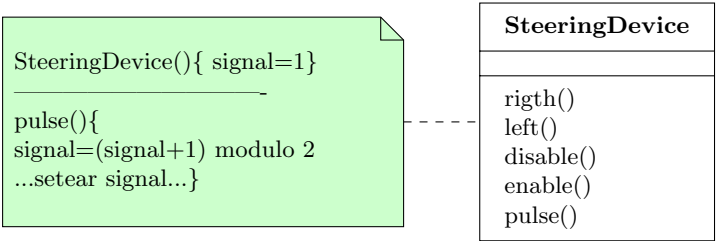


Figura 5.13: Temporizadores  
*Timer FirstTimer SecondTimer*

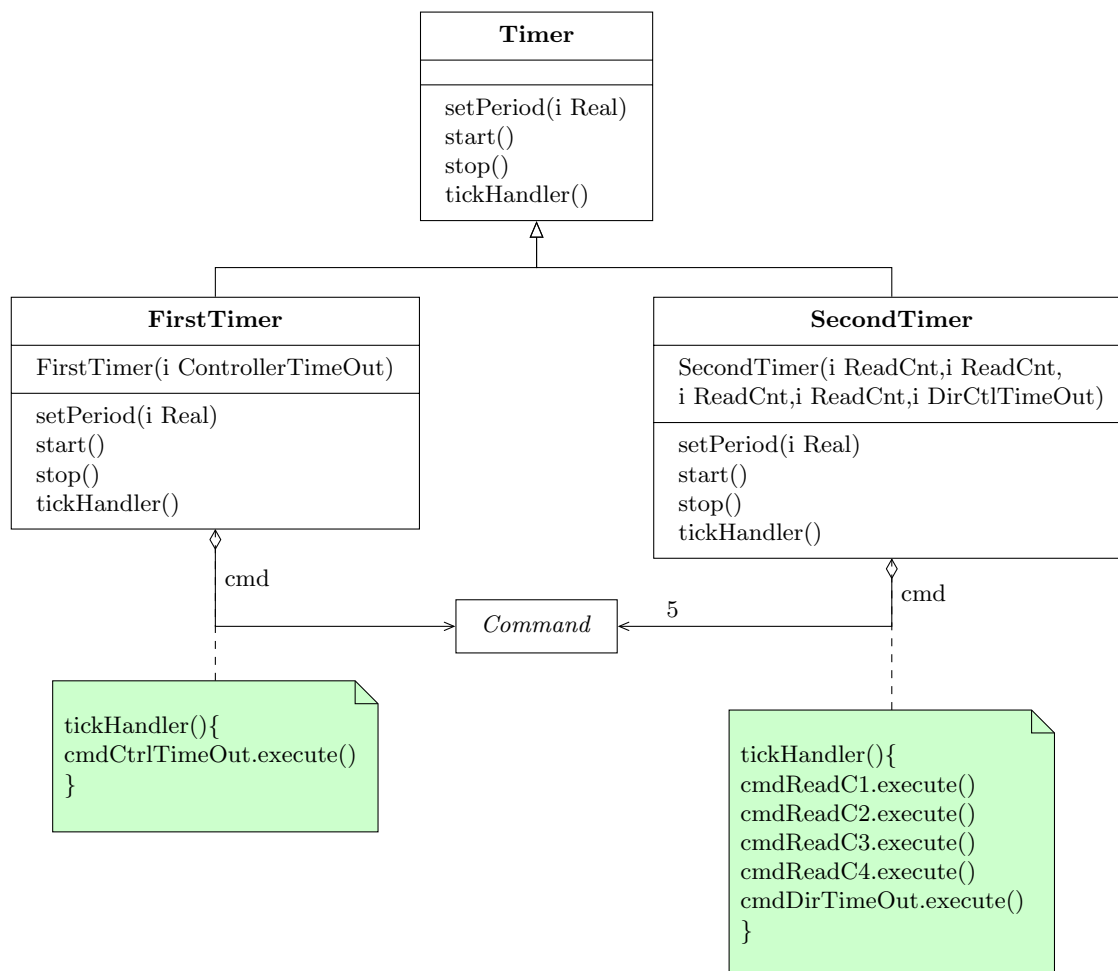
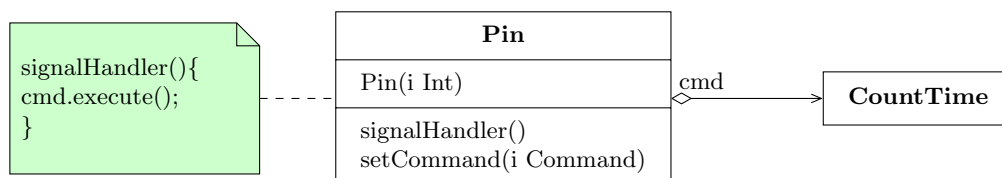


Figura 5.14: Pin receptor de las señales provenientes del CR  
*Pin*



## 5.5. Recolectores de señales físicas

Figura 5.15: Recolectores de instantes de tiempo. Patrón Decorador  
*Collector* *TimeCollector* *DecoCollector* *CRpinCollector* *SensorCollector*

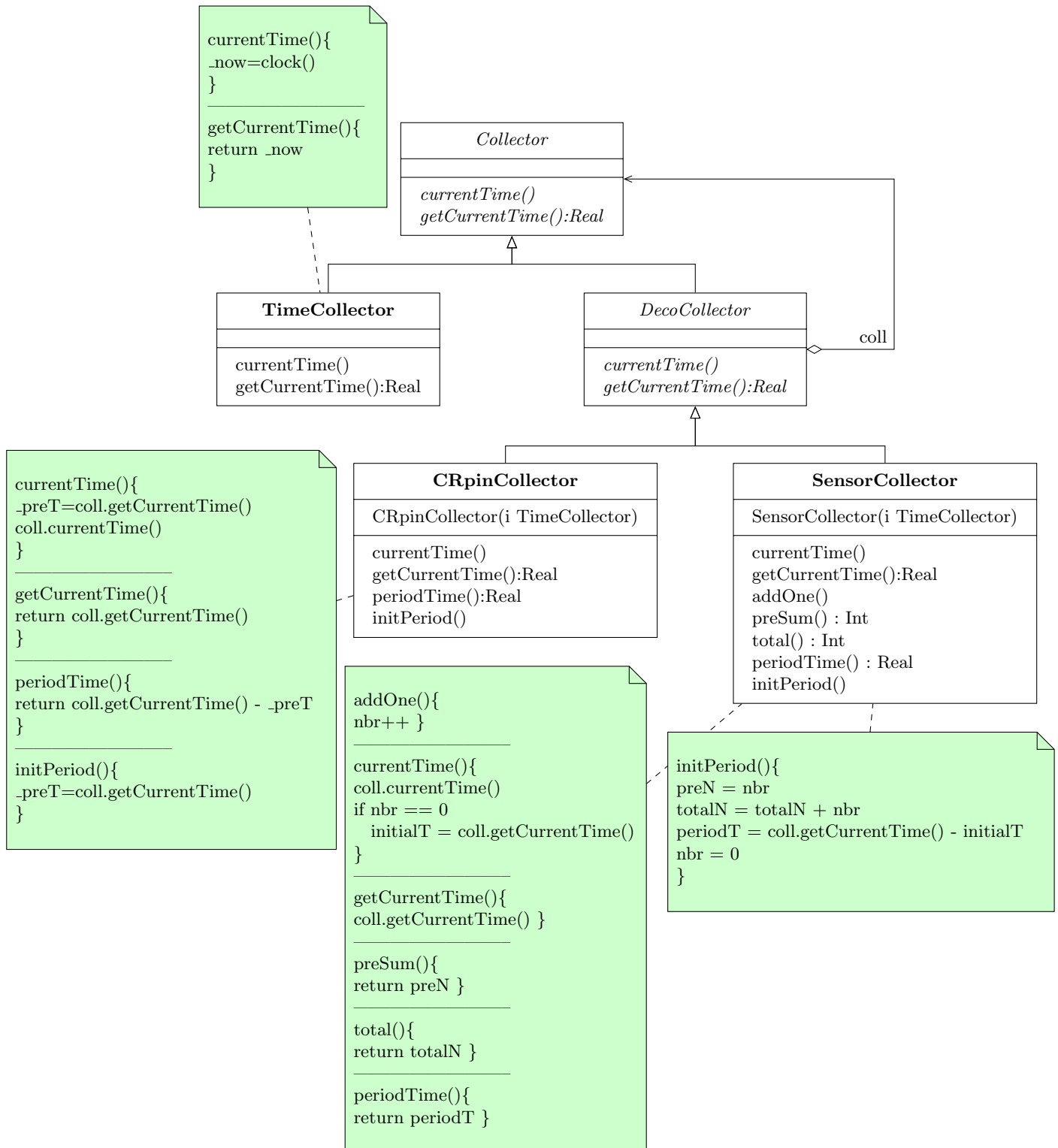
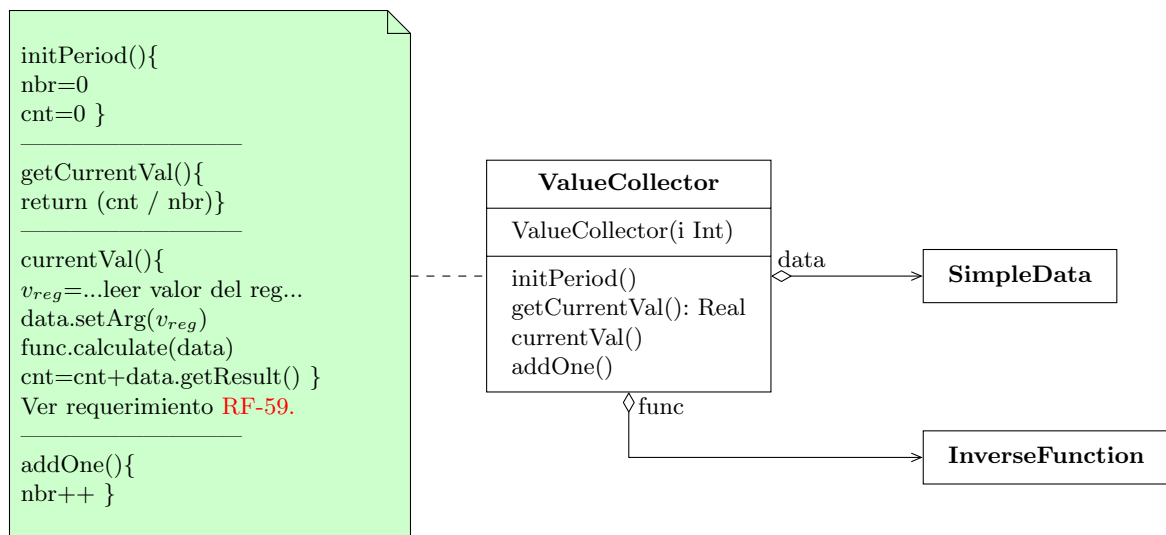


Figura 5.16: Recolector de valores utilizado para la lectura del sensor de corriente.

**ValueCollector**





5.6. Sensores y buffers

Figura 5.17: Sensores Pasivos  
*PassiveSensor VelSensor CntSensor DirSensor*

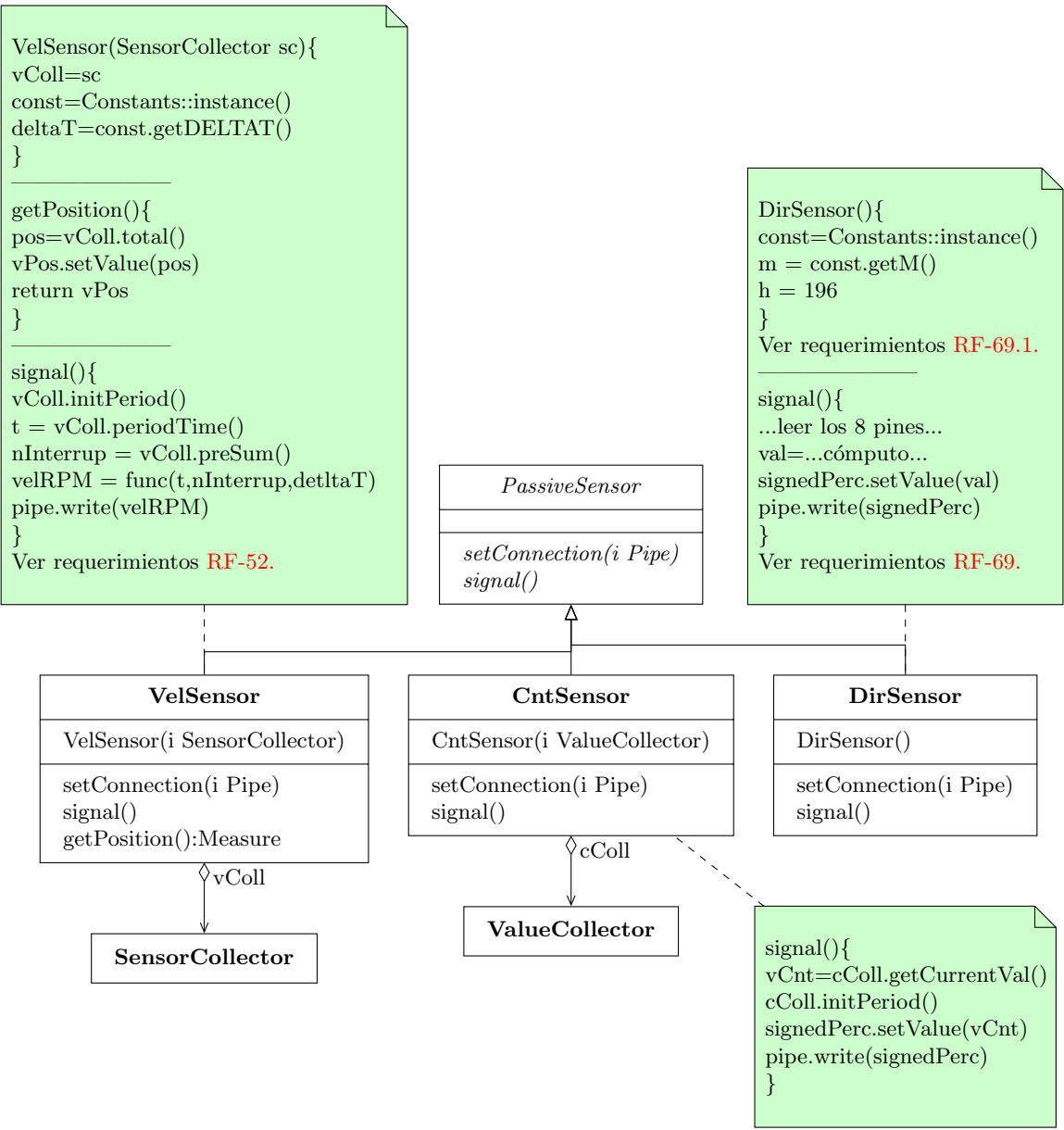


Figura 5.18: Sensor Hall  
*ActiveSensor*

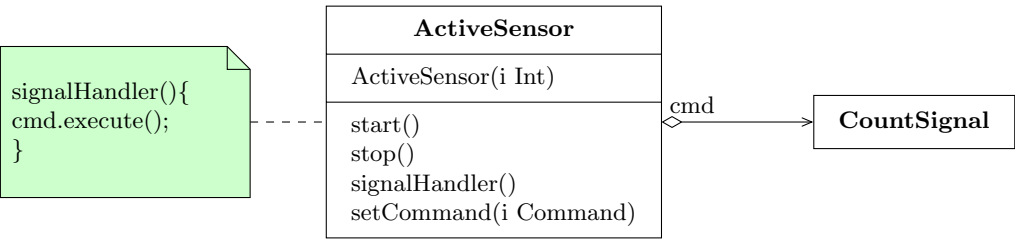
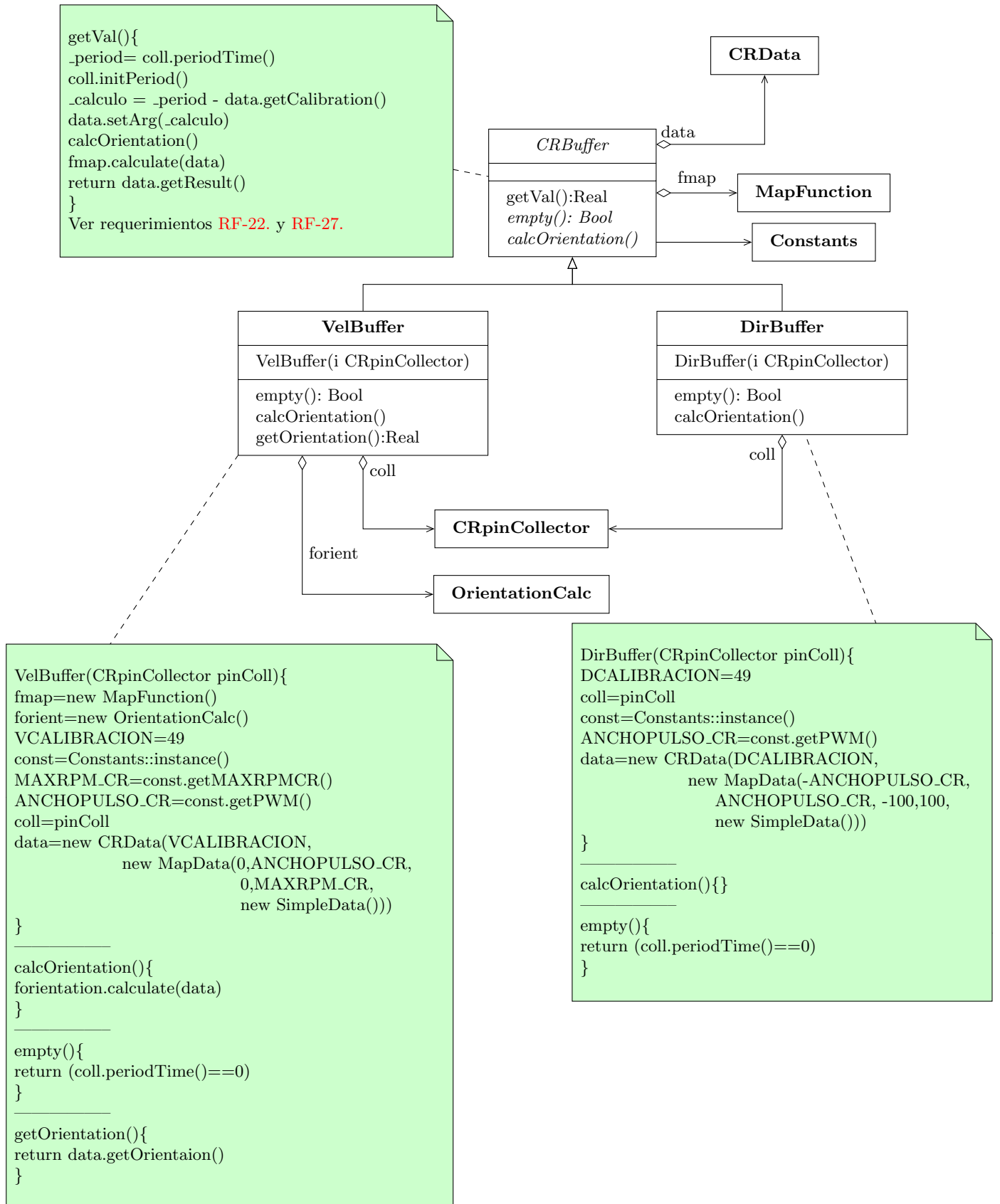


Figura 5.19: Buffers del CR. Patrón Método plantilla  
*CRBuffer* *VelBuffer* *DirBuffer*



## 5.7. Sistemas de Control

### 5.7.1. Sistema de Control de Ruedas

Figura 5.20: Órdenes a llevar a cabo sobre un sistema de control de ruedas. En particular: registrar la posición de las ruedas, que los sensores escriban en el pipe y que el controlador lea de los pipes. Patrón Estrategia  
*WSysOrder* *SaveWPosition* *SensorWritesVel* *ControllerReadsVel* *SensorWritesCnt* *ControllerReadsCnt*

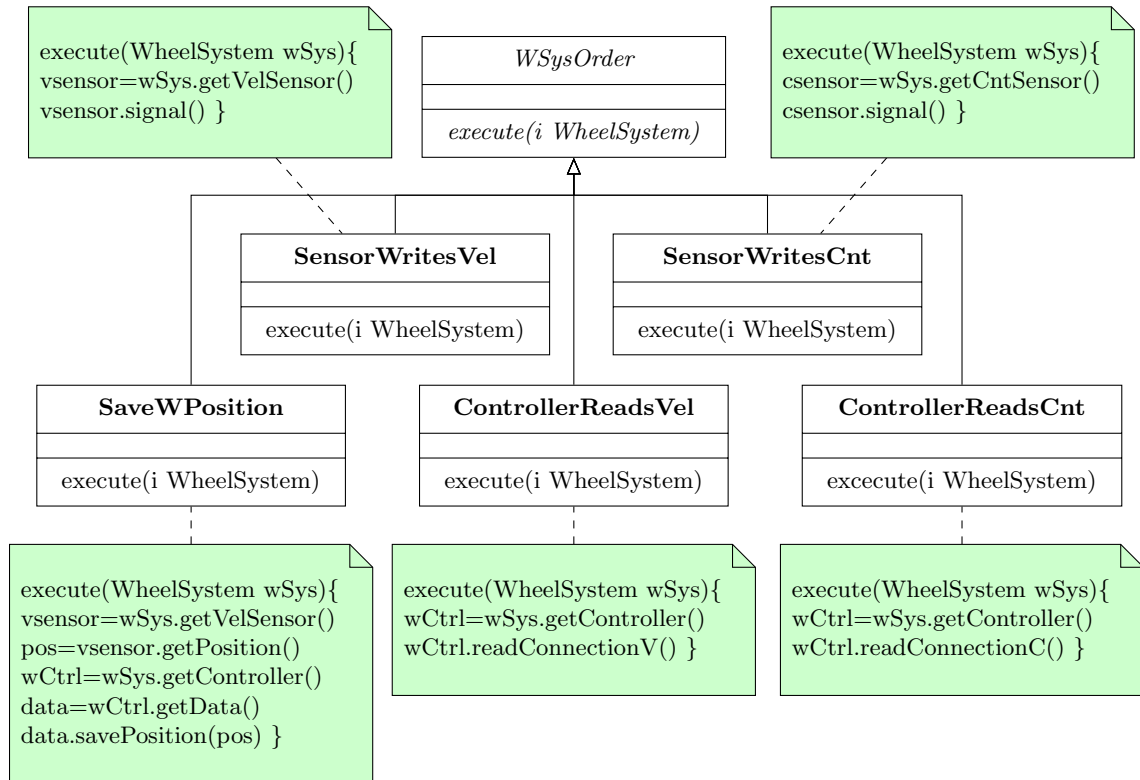


Figura 5.21: Comandos que serán utilizados dentro del algoritmo de control de ruedas dependiendo si hay o no cambio de sentido o si se quiere detener la rueda. Patrón Orden  
*WCtrlCommand VelNull Brake SetTension ChangeOrientation*

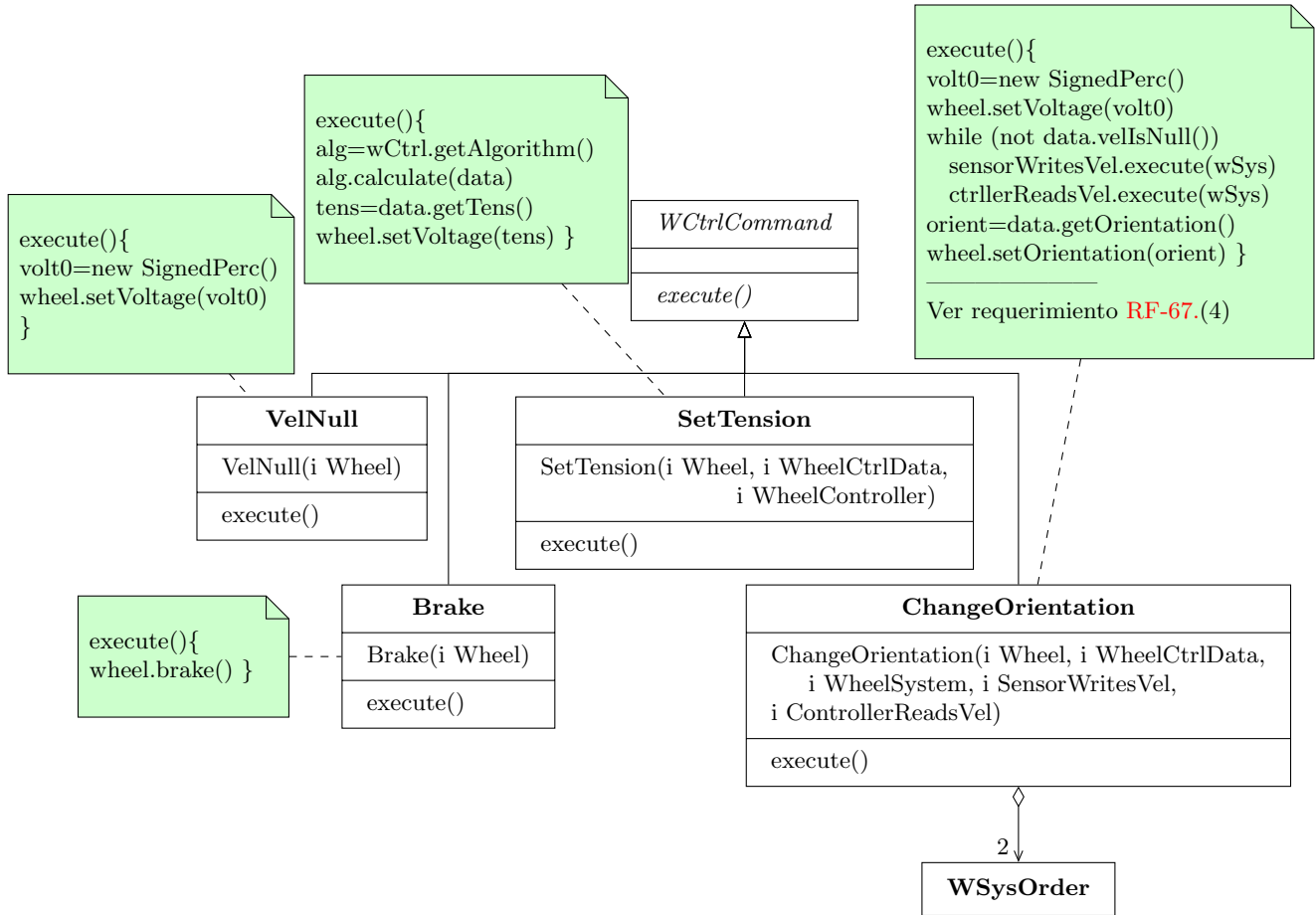


Figura 5.22: Conjunto de comandos que pueden ser utilizados por el algoritmo de control de ruedas  
*CtrlCmdPool*

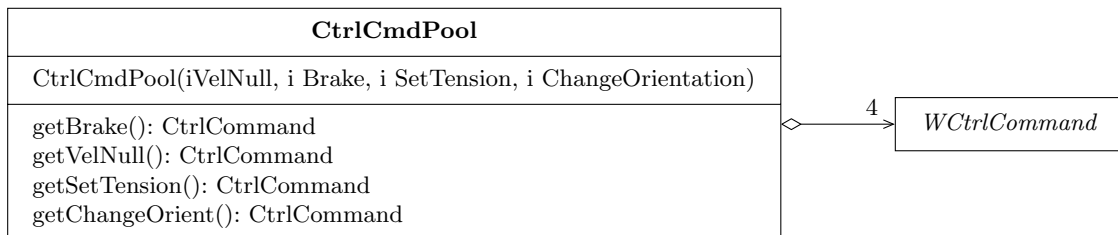
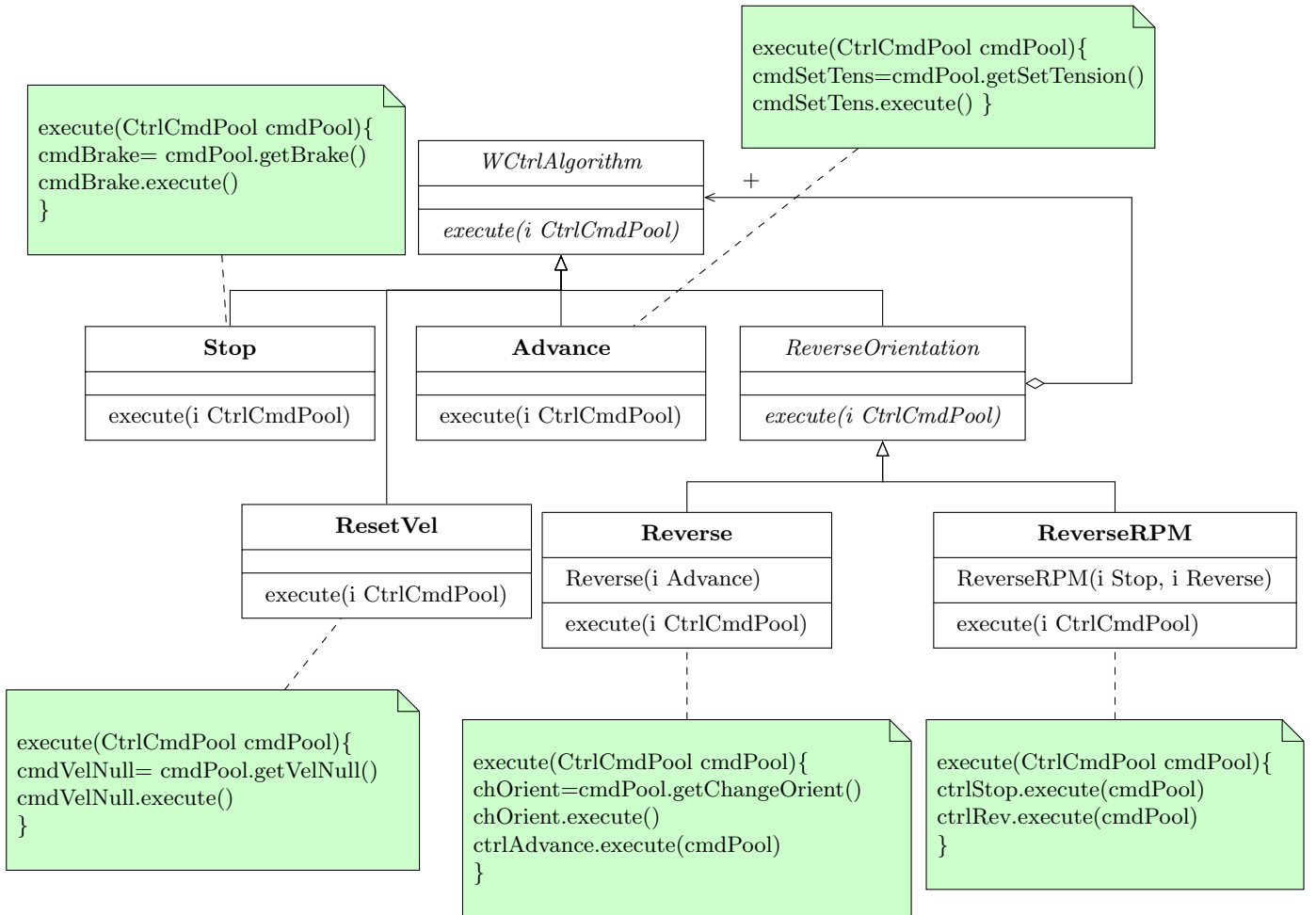
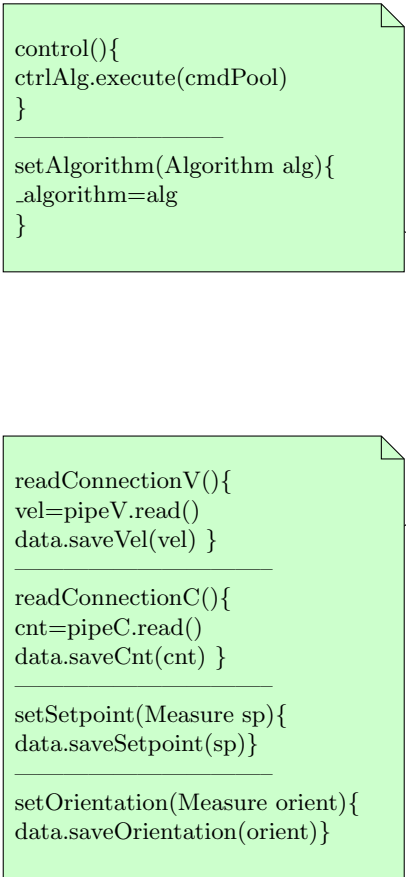


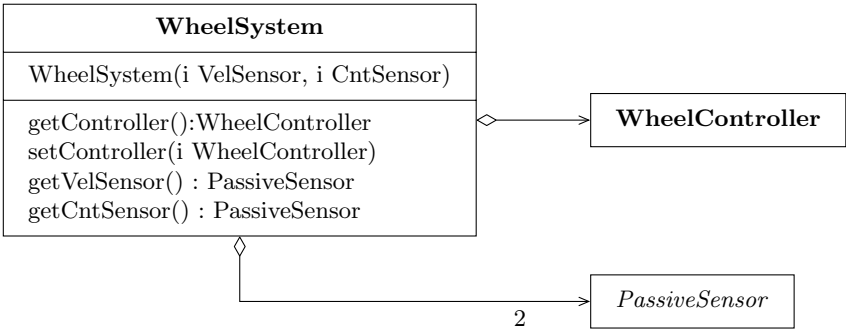
Figura 5.23: Algoritmos de control de rueda. Patrón Decorador. Patrón Estrategia  
*WCtrlAlgorithm Stop ResetVel Advance ReverseOrientation. Reverse ReverseRPM*



## WheelController



# WheelSystem



### 5.7.2. Sistema de Control de Dirección

Figura 5.26: Comandos sobre el dispositivo de dirección. Patrón Orden  
*DCtrlCommand* *Turn* *SetDirection* *Enable* *Disable*

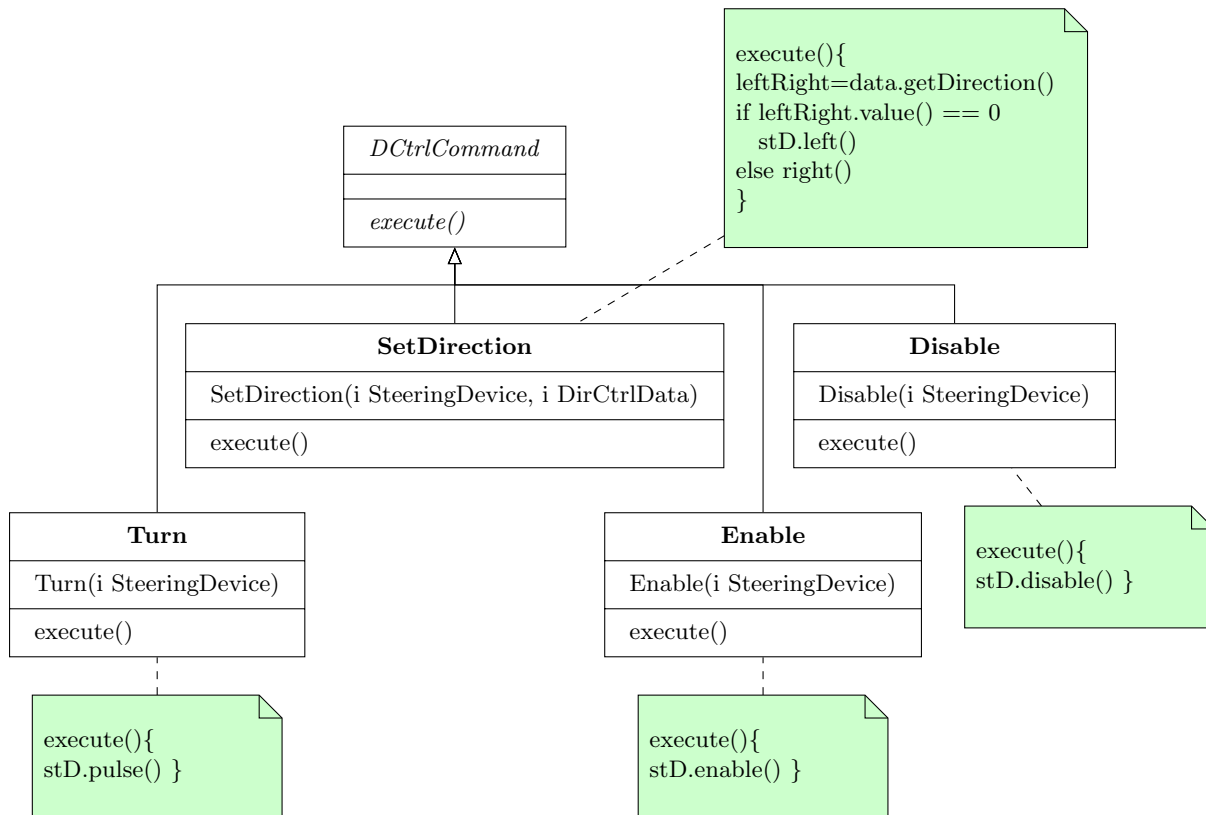


Figura 5.27: Estados del dispositivo de dirección que serán un estado del DirController. Patrón Estado  
*DeviceState* *OnState* *OffState*

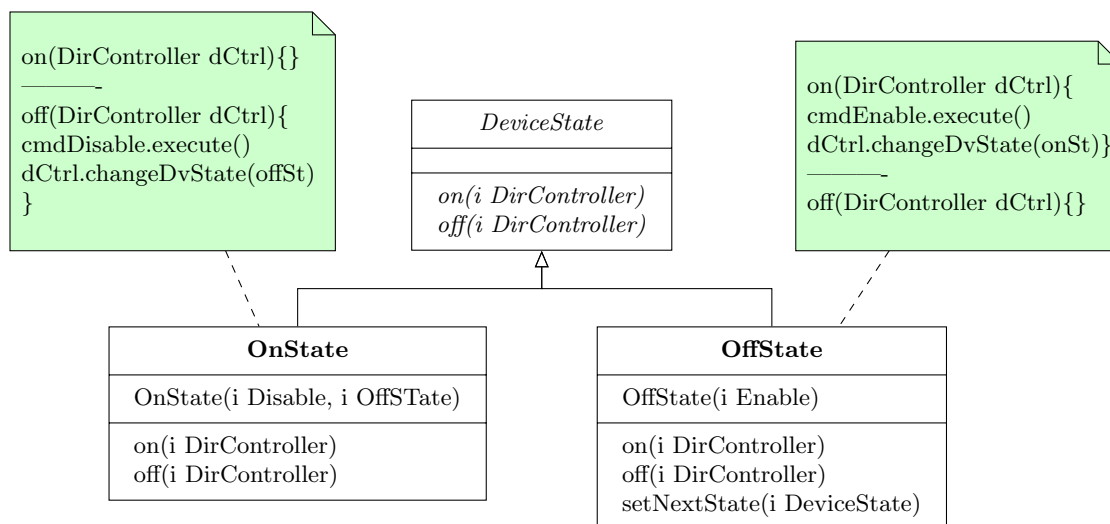


Figura 5.28: Estados de operación del control de dirección. Patrón Estado  
*DOperationState* *DInactive* *DTurning* *DActive* *DStopping*

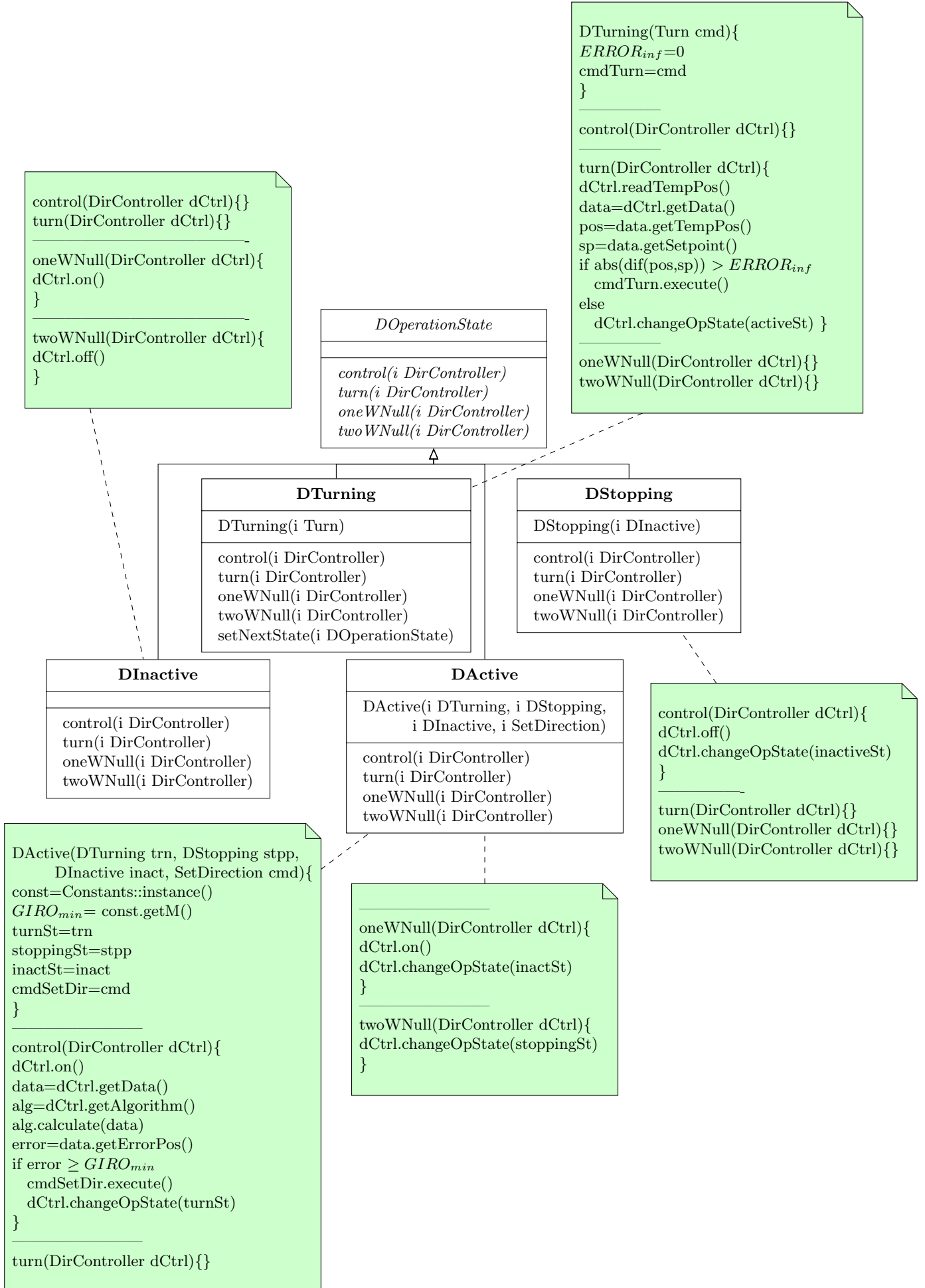




Figura 5.29: Controlador de dirección  
**DirController**

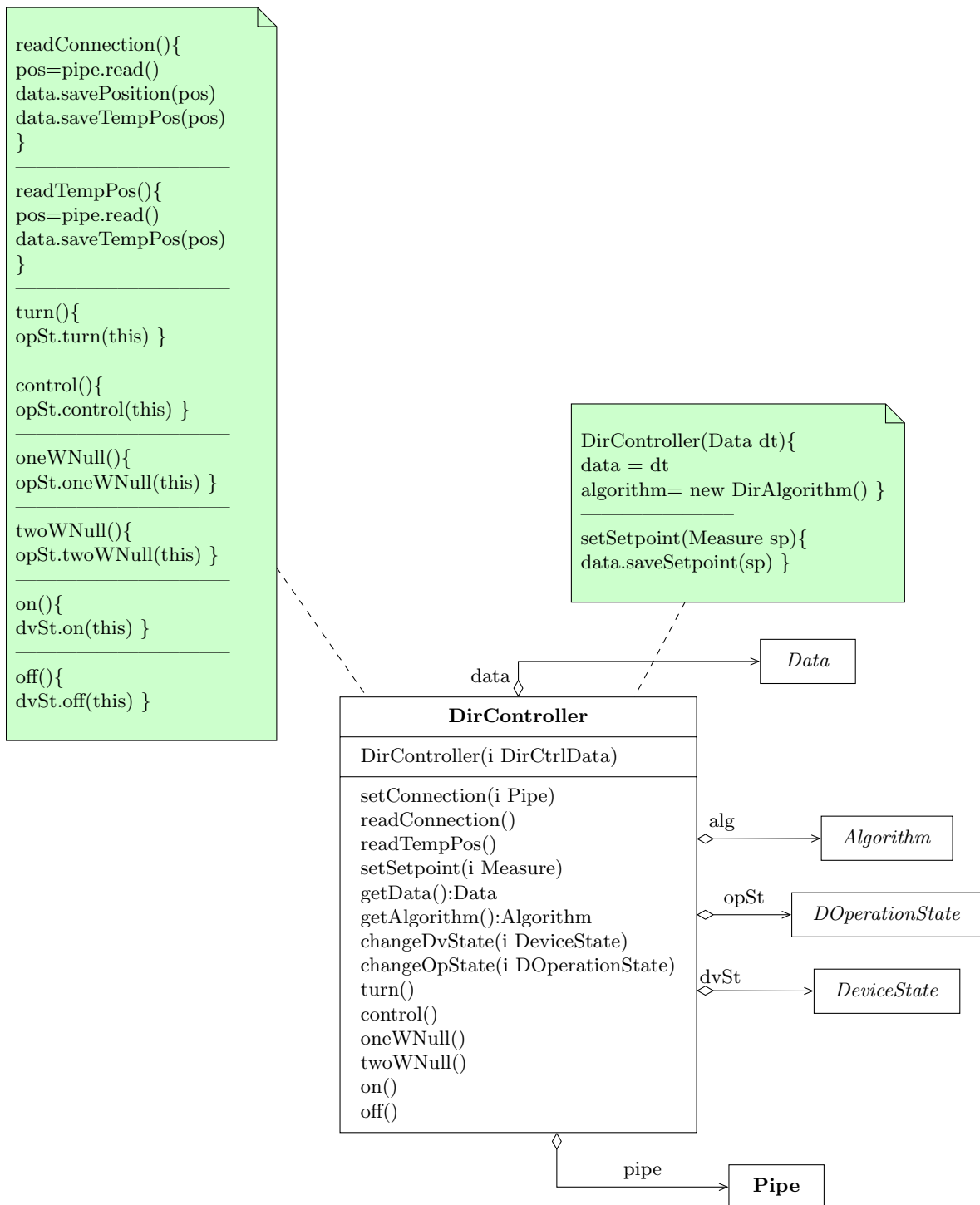
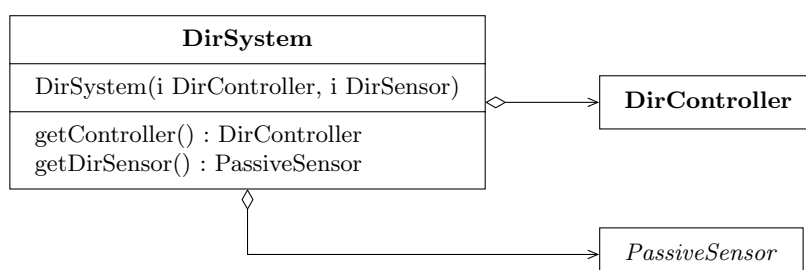


Figura 5.30: Sistema de control de dirección  
**DirSystem**



# 5.8. Controlador principal, órdenes, estados y modos de operación

Figura 5.31: Modos de operación: PC o CR. Patrón Estado y Patrón Decorador  
*Mode BasicMode LectureMode PC CR*

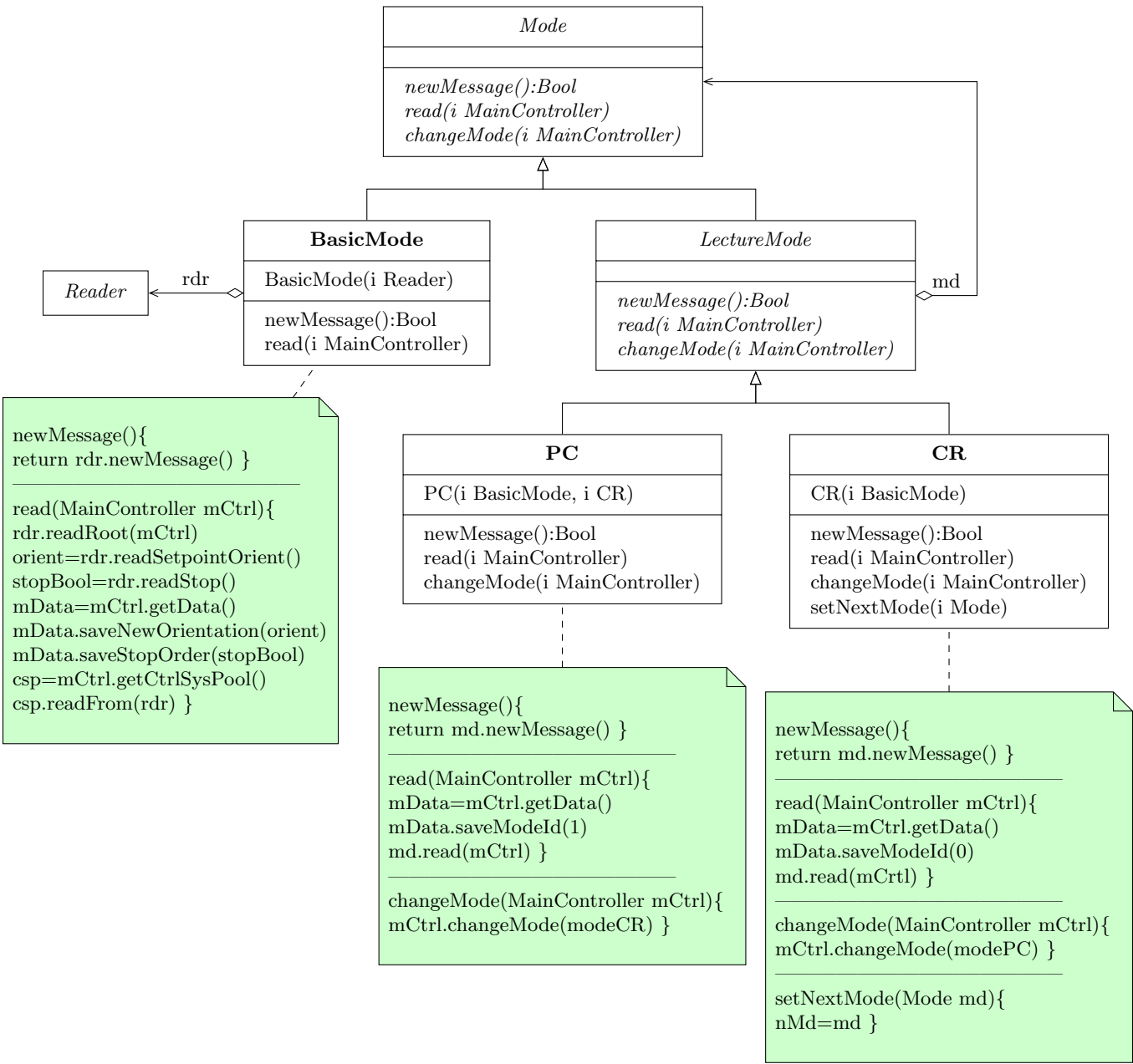


Figura 5.32: Este contenedor se utiliza para la construcción del MainController, es decir es utilizado por MCBuilder.  
*ModePool*

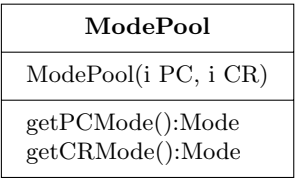


Figura 5.33: Órdenes que debe llevar a cabo el controlador principal sobre los sistemas de control.

Patrón Método Plantilla

*Order SaveWheelPositions SensorsWrite ControllersRead ControllersControl ControllersStop*

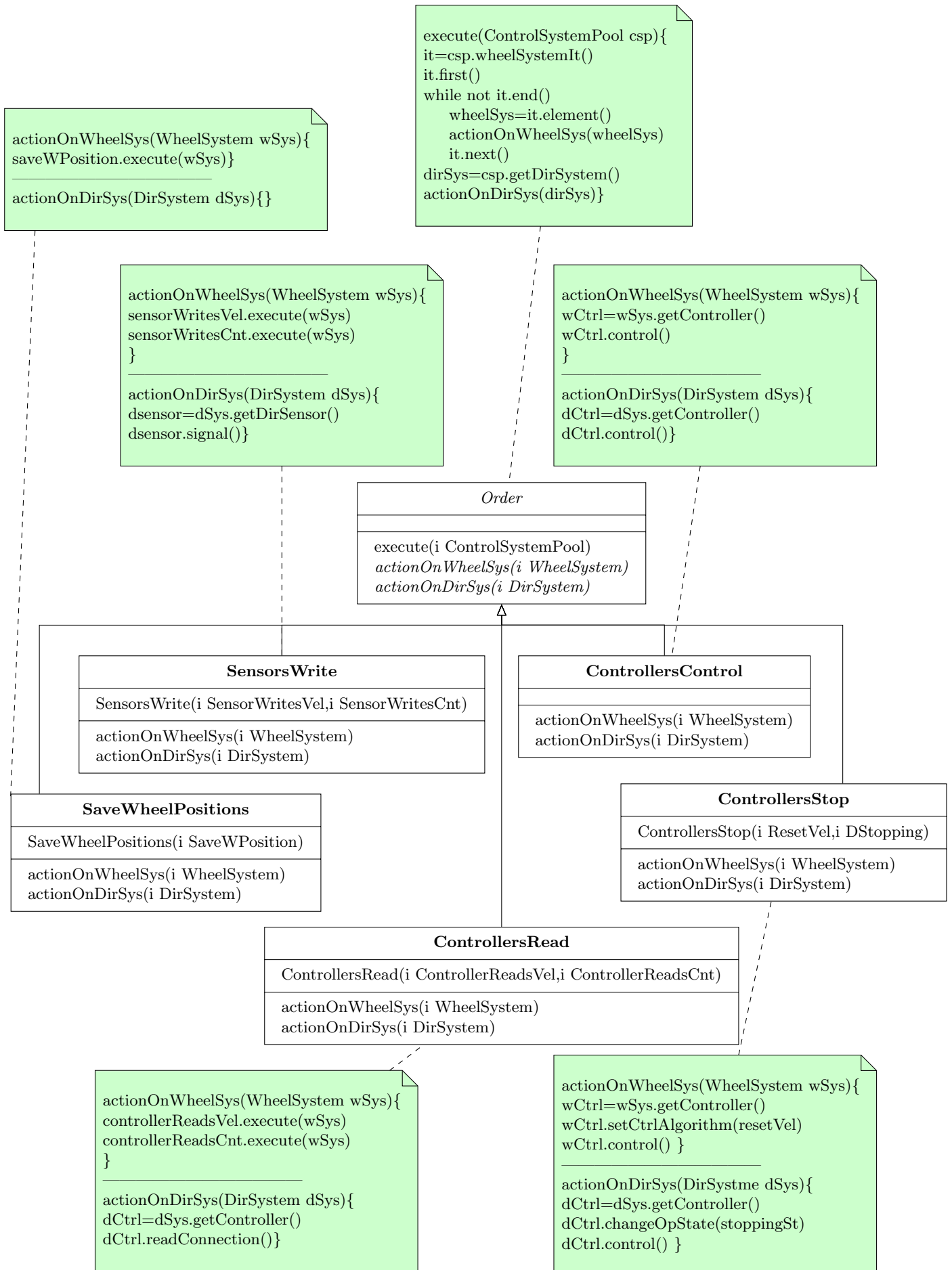


Figura 5.34: Esta orden actualiza el estado (moviéndose o detenido) del sistema de dirección.  
**UpdateOrder**

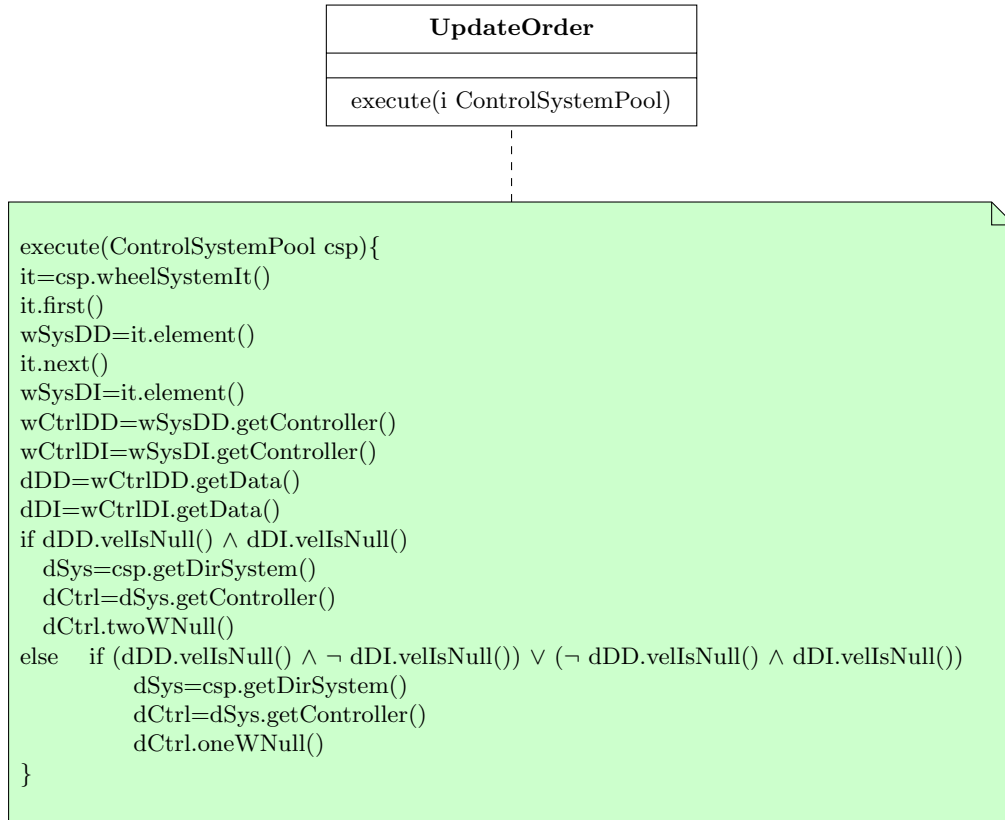


Figura 5.35: Orden sobre el controlador principal que se llevan a cabo en los diferentes estados de operación de este  
**MainCtrlOrder**

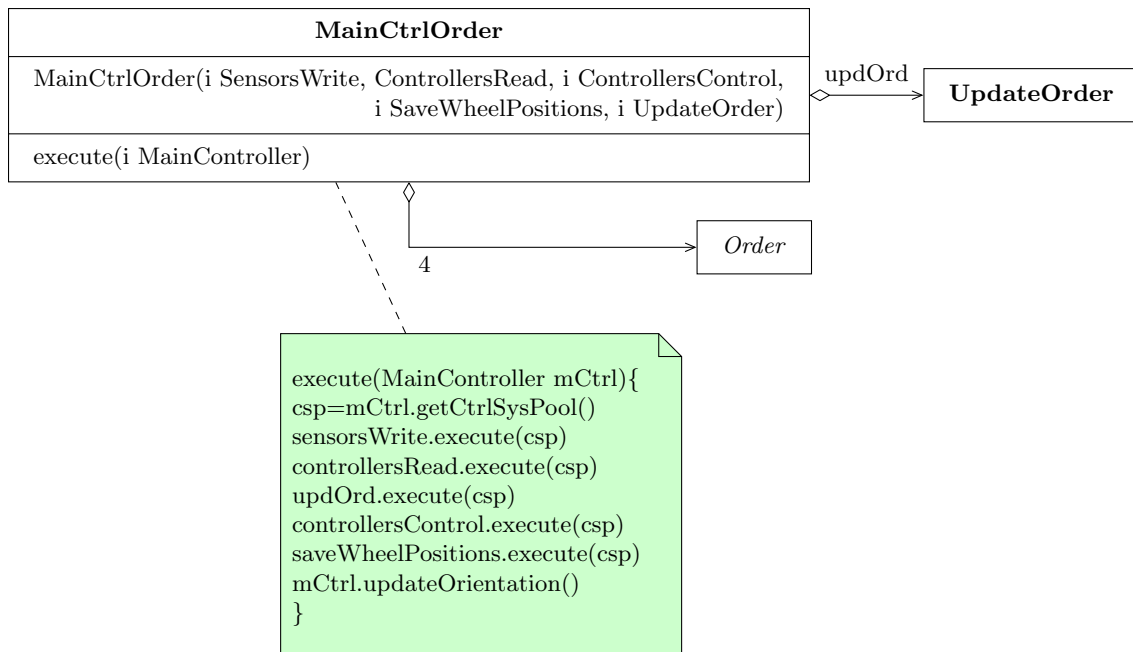


Figura 5.36: Estados de operación del controlador principal. Patrón Estado. Patrón Método Plantilla  
*OperationState* *WaitingN* *WaitingMAX* *Reconnecting* *Working*

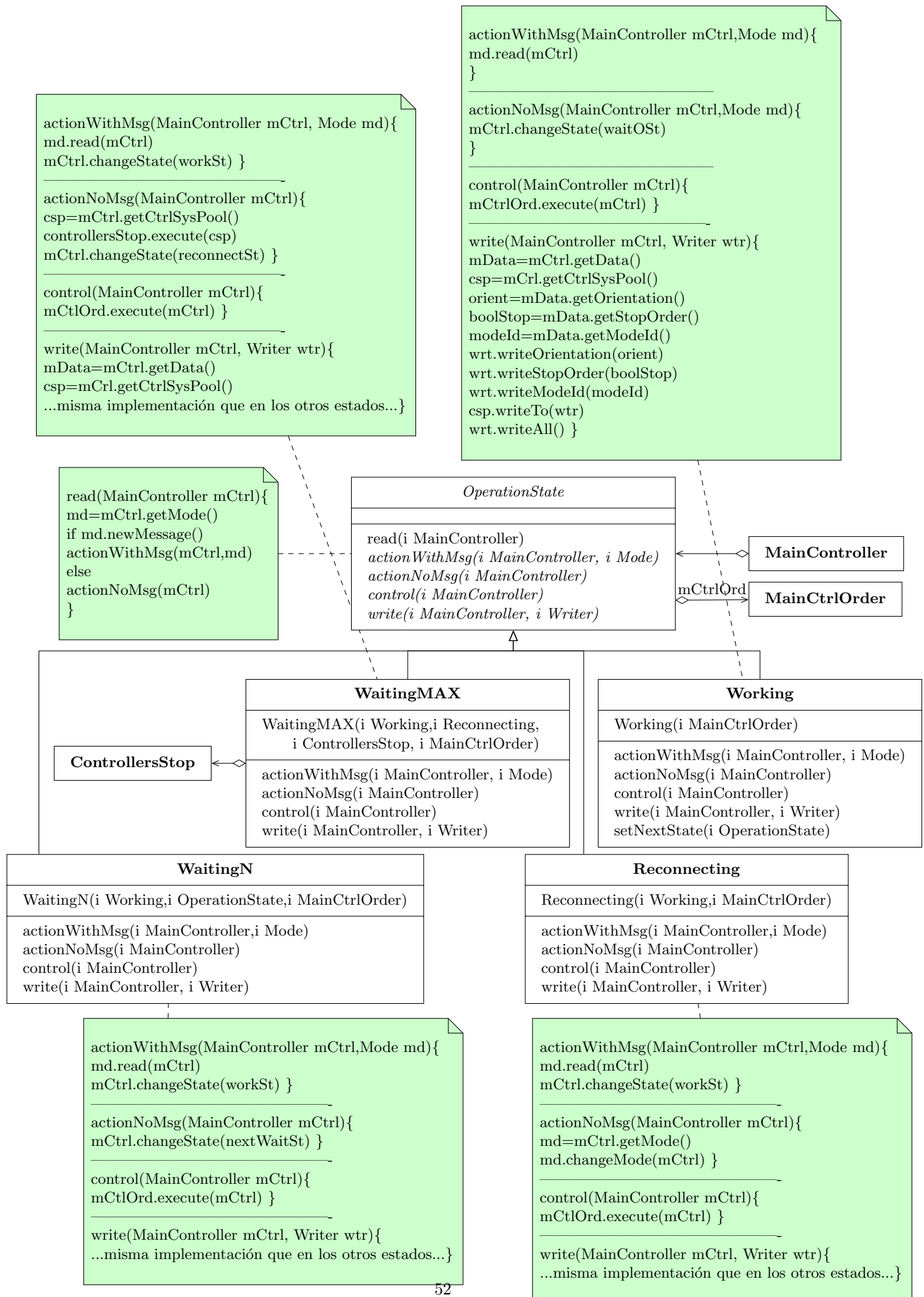
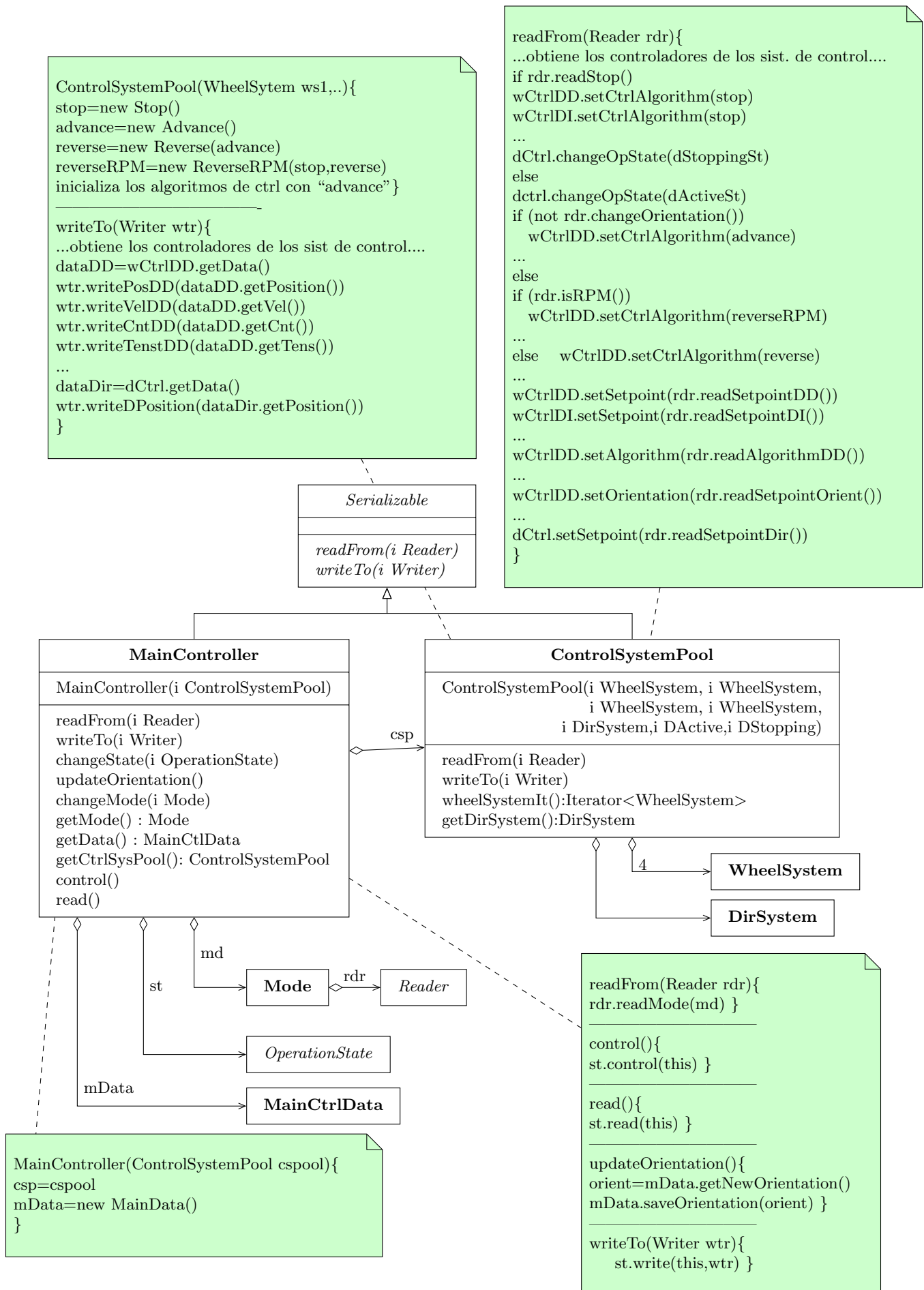


Figura 5.37: Controlador principal que controla los sistemas de control de rueda y dirección. Patrón Serializador  
*Serializable MainController ControlSystemPool*



## 5.9. Lectura y escritura de información

Figura 5.38: Módulo de escritura de la información hacia de la PC. Patrón Serializador  
*Writer SerialWriter*

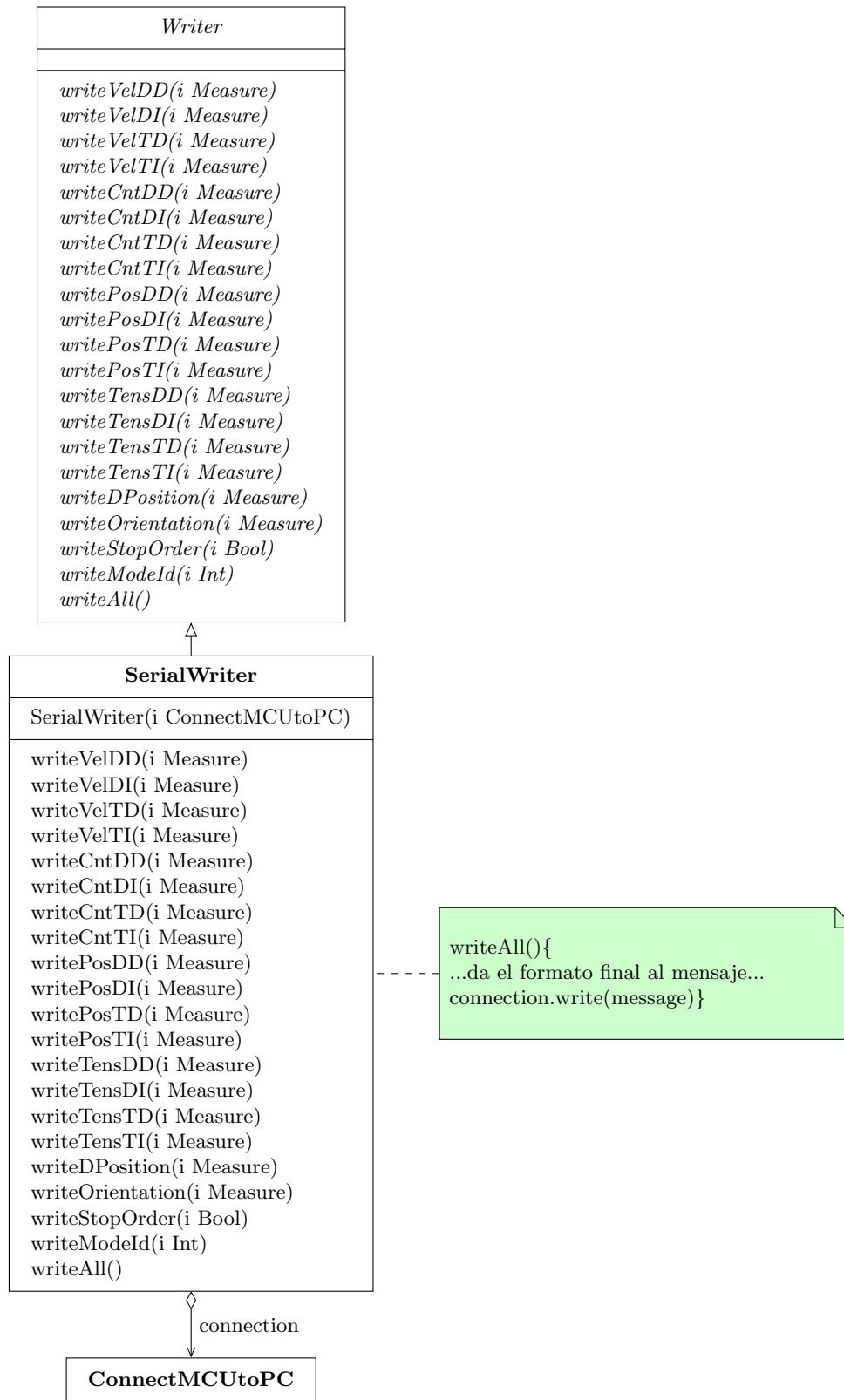


Figura 5.39: Módulos lectores de la información proveniente de la PC y del CR. Patrón Serializador

*Reader SerialReader BufferReader*

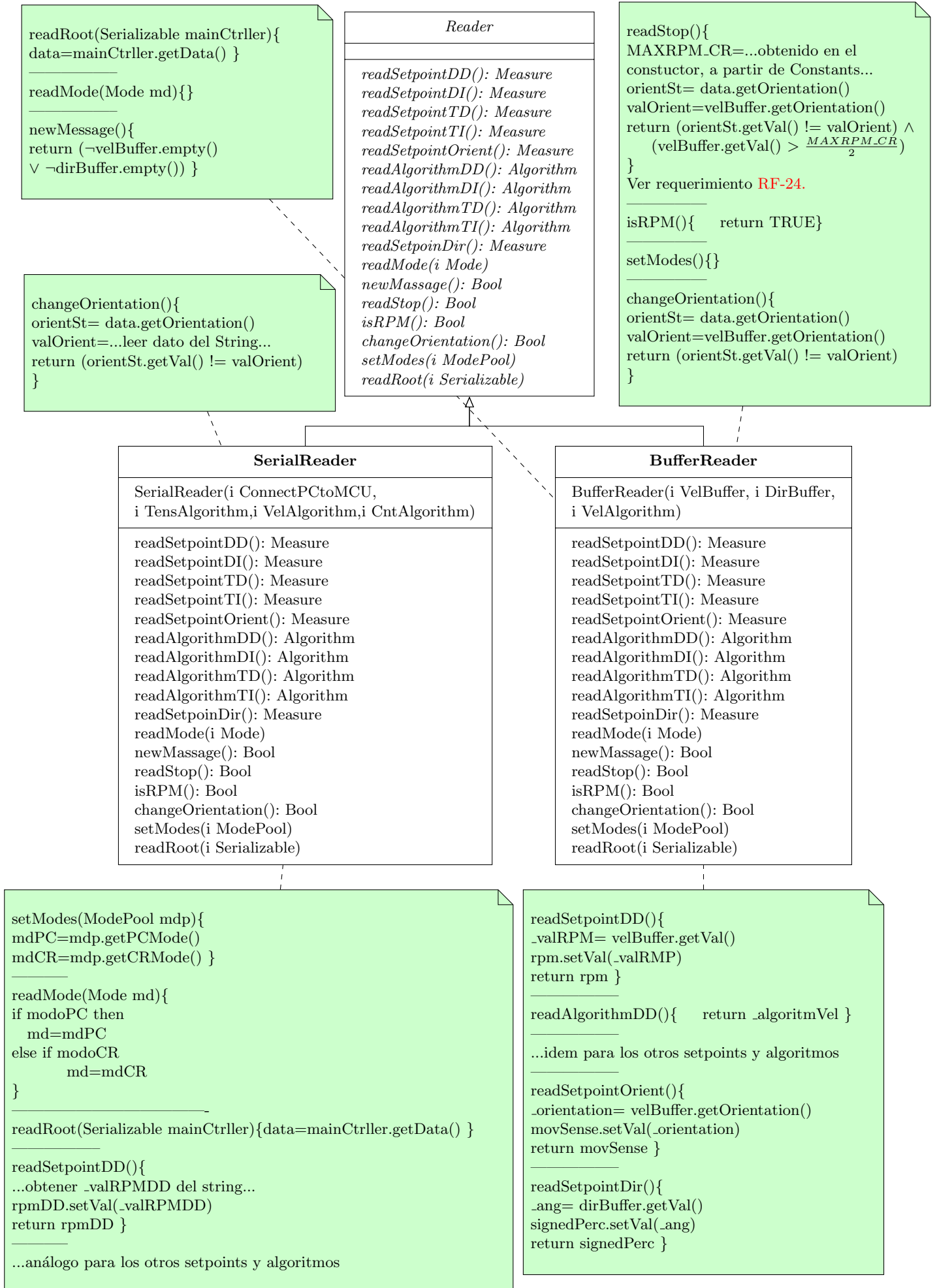
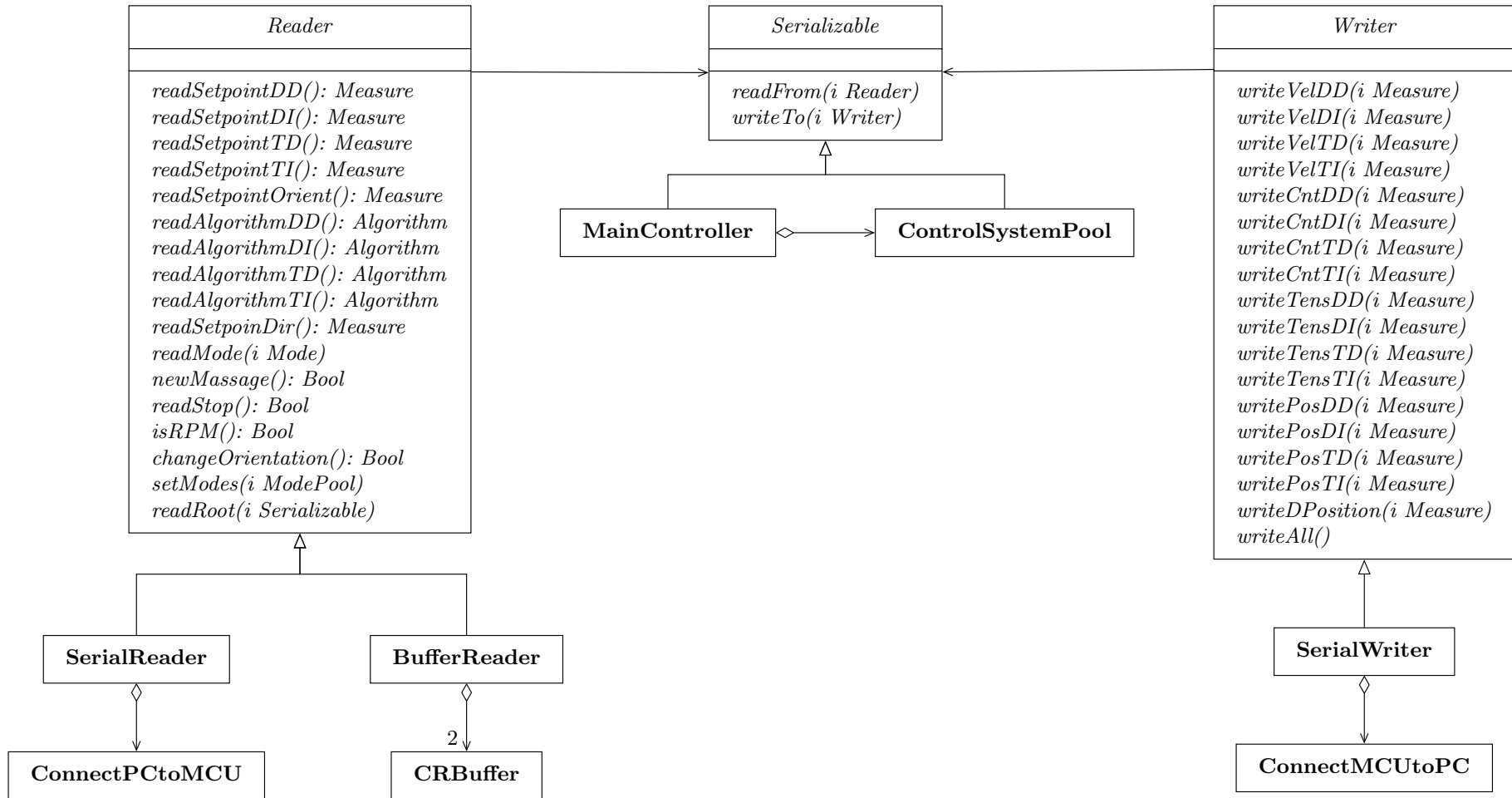


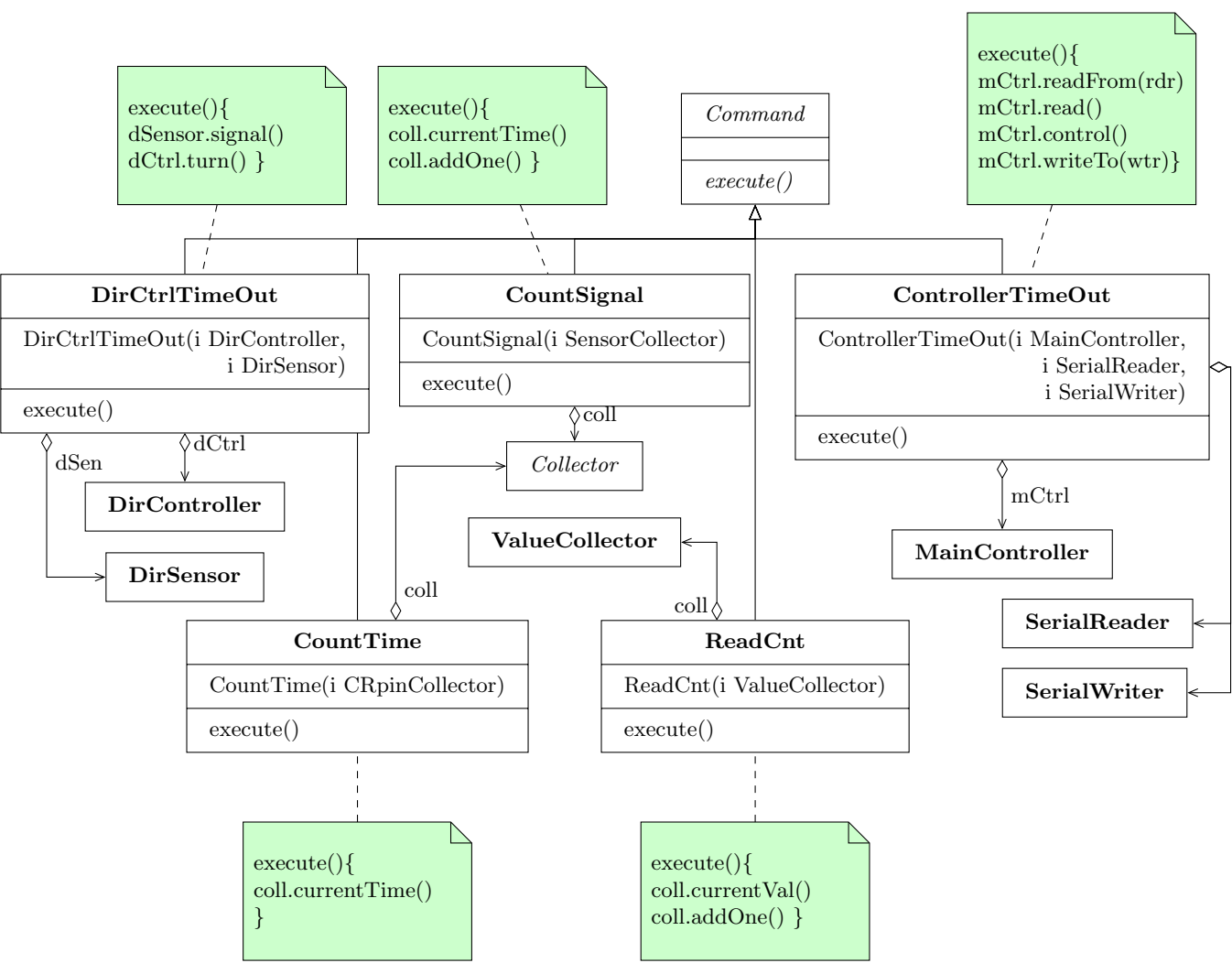


Figura 5.40: Lectores y escritores del sistema. Patrón Serializador. Ver Figuras 5.37, 5.38, 5.39  
*Serializable* *MainController* *ControlSystemPool* *Reader* *SerialReader* *BufferReader* *ConnectPCtoMCU* *CRBuffer* *Writer* *SerialWriter* *ConnectMCUtoPC*



# 5.10. Comandos utilizados como manejadores de señales físicas

Figura 5.41: Comandos utilizados en el patrón Orden para sustituir el *callback*.  
*Command DirCtrlTimeOut CountTime CountSignal ReadCnt ControllerTimeOut*



## 5.11. Construcción de objetos del sistema

Figura 5.42: Constructor de sistema de control de ruedas. Patrón Constructor. Ver también Figura 5.46  
*WSDirector WheelSysBuilder WSBuilder*



Figura 5.43: Constructor de sistema de dirección. Patrón Constructor. Ver también Figura 5.46  
*DSDirector DirSysBuilder DSBuilder*

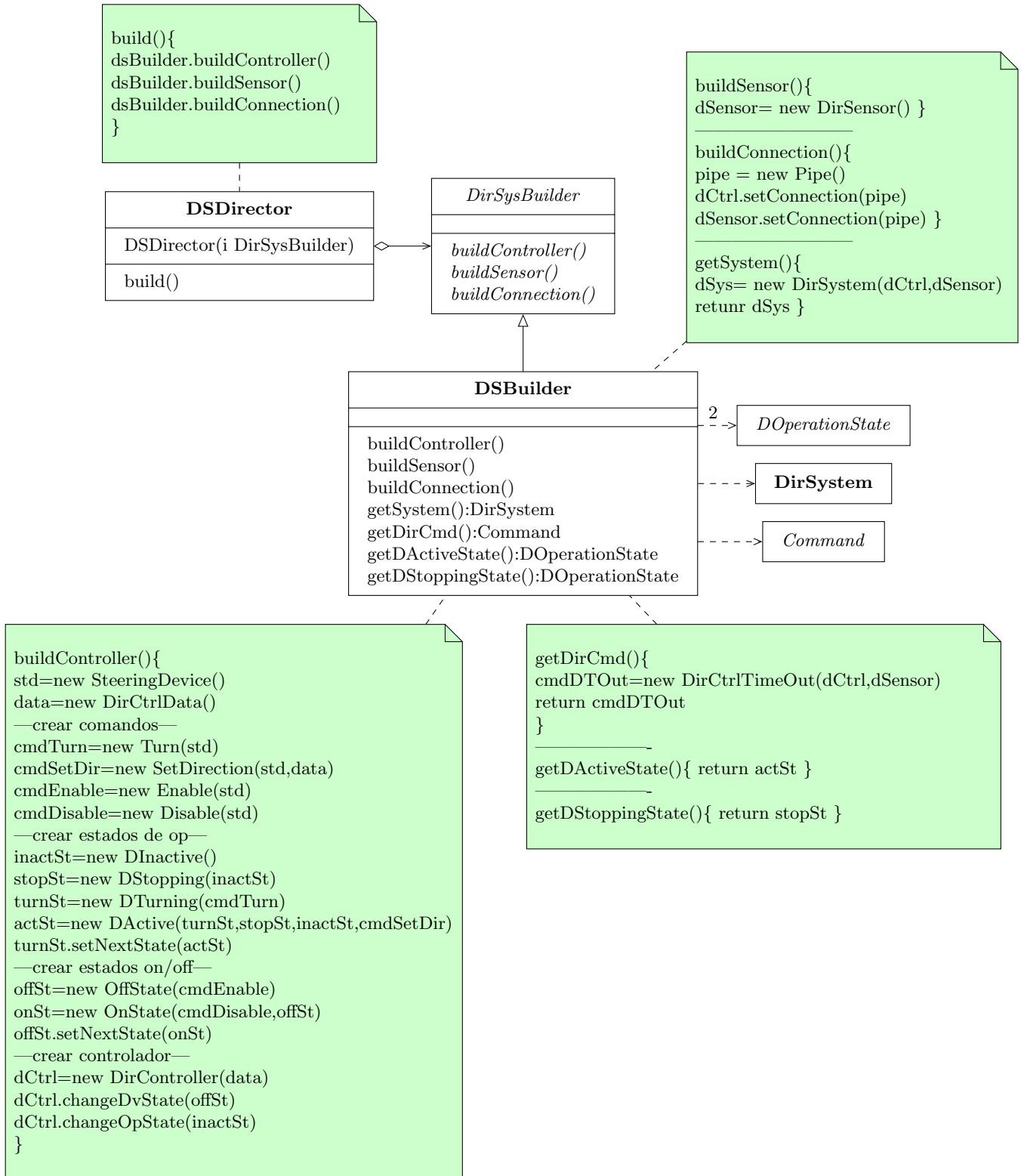


Figura 5.44: Constructor del conjunto de sistemas de control. Patrón Constructor. Ver también Figura 5.46  
*CSPDirector CtrlSysPoolBuilder CSPBuilder*

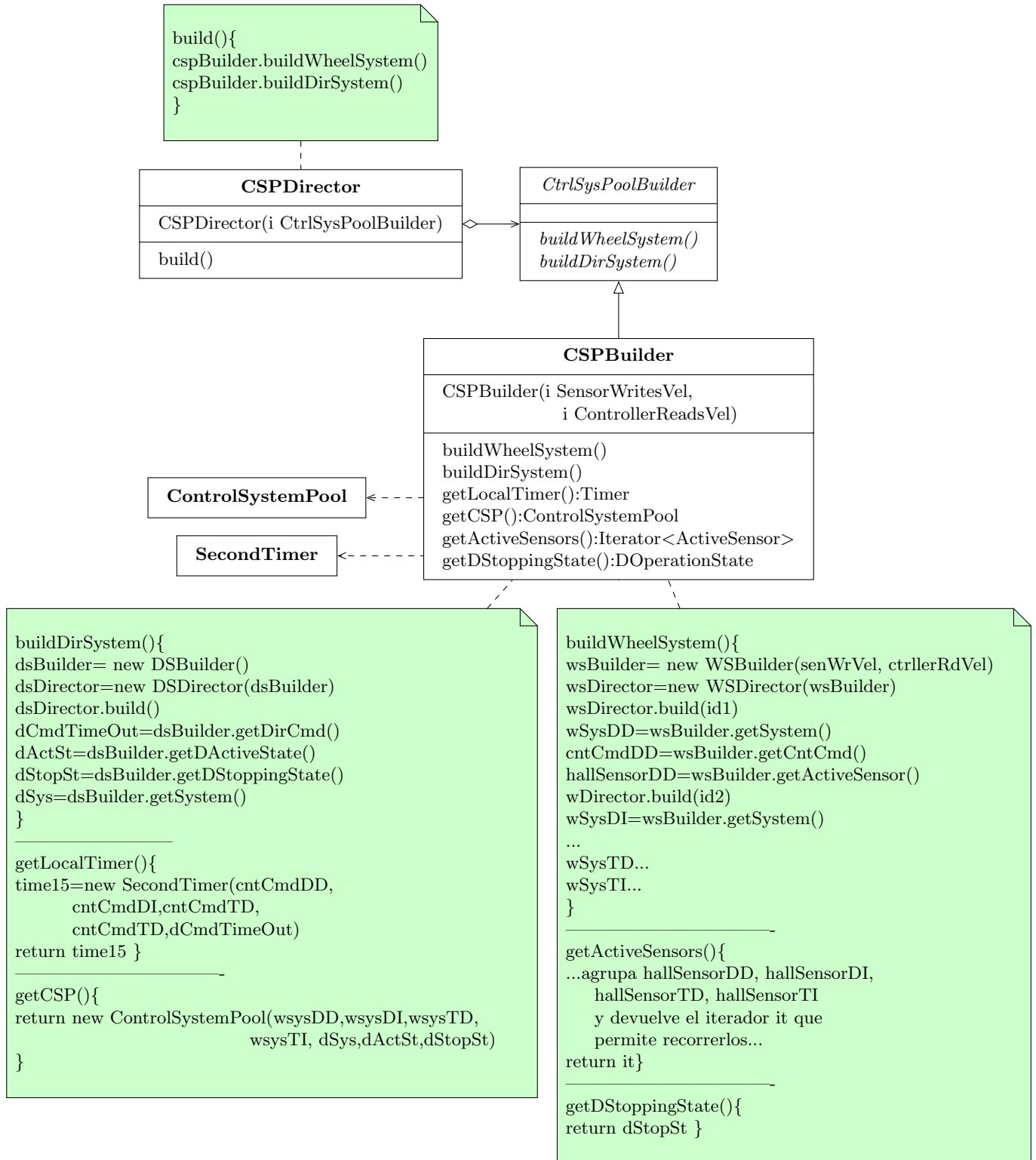
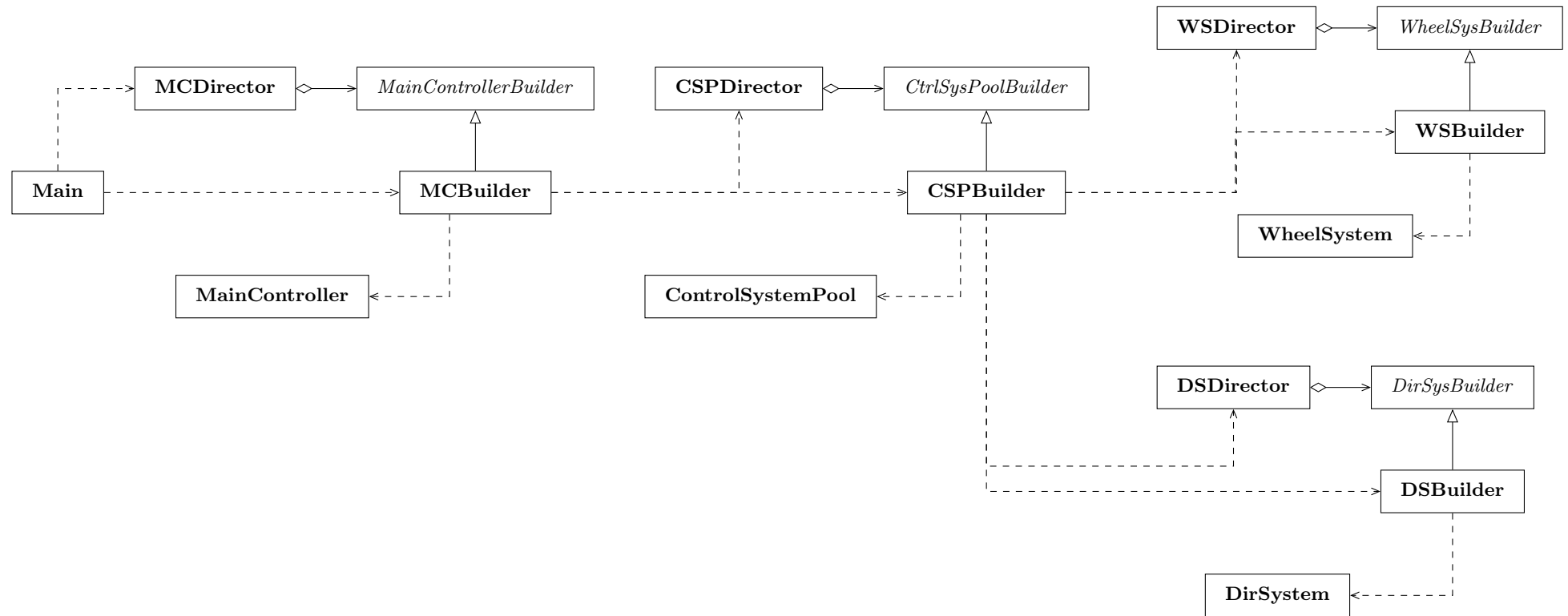


Figura 5.45: Constructor de sistema de control principal. Patrón Constructor. Ver también Figura 5.46  
*MCDirector MainControllerBuilder MCBUILDER*



Figura 5.46: Constructores de los sistemas de control. Ver Figuras [5.42](#), [5.43](#), [5.44](#), [5.45](#)



## 5.12. Programa principal

Figura 5.47: Programa principal

Main

Main

```
main(){
    -----pin de CR de velocidad-----
    vtColl= new TimeCollector()
    vPinColl=new CRpinCollector(vtColl)
    vCmdCountT=new CountTime(vPinColl)
    vpin=new Pin(id1)
    vpin.setCommand(vCmdCountT)
    vBuffer=new CRBuffer(vPinColl)
    -----pin de CR de dirección-----
    dtColl= new TimeCollector()
    dPinColl=new CRpinCollector(dtColl)
    dCmdCountT=new CountTime(dPinColl)
    dpin=new Pin(id2)
    dpin.setCommand(dCmdCountT)
    dBuffer=new CRBuffer(dPinColl)
    -----Escribiente-----
    mcuToPc=new ConnectMCUtoPC
    serialWr=new SerialWriter(mcuToPC)
    -----Lectores-----
    tensAlg=new TensAlgorithm()
    velAlg=new VelAlgorithm()
    cntAlg=new CntAlgorithm()
    bufferRdr=new BufferedReader(vBuffer,dBuffer,velAlg)
    transmittingSt=new Transmitting()
    noMessageSt=new NoMessage(transmittingSt)
    readyMessageSt=new ReadyMessage(noMessageSt,transmittingSt)
    transmittingSt.setNextState(readyMessageSt)
    pcToMCU=new ConnectPCtoMCU()
    pcToMCU.changeState(noMessageSt)
    serialRdr=new SerialReader(pcToMCU,tensAlg,velAlg,cntAlg)
    -----modos de operación-----
    bMdCR=new BasicMode(bufferRdr)
    cr=new CR(bMdCR)
    bMdPC=new BasicMode(serialRdr)
    pc=new PC(bMdPC, cr)
    cr.setNextMode(pc)
    mdPool=new ModePool(pc,cr)
    serialRdr.setModes(mdPool)
    -----construir el MainController-----
    mcBuilder= new MCBuilder(mdPool)
    mcDirector= new MCDirector(mcBuilder)
    mcDirector.build()
    mCtrl= mcBuilder.getMC()
    -----construir el temporizadores-----
    cmdTimeOut100=new ControllerTimeOut(mCtrl,serialRdr,serialWr)
    const=Constants::instance()
    deltaT=const.getDELTAT()
    t100= new FirstTimer(cmdTimeOut100)
    t100.setPeriod(deltaT)
    t15=mcBuilder.getLocalTimer()
    t15.setPeriod(1,5)
    -----iniciar el sistema-----
    itHallSensors=mcBuilder.getActiveSensors()
    itHallSensors.first()
    while not itHallSensors.end()
        hallSensor=itHallSensors.getElement()
        hallSensor.start()
        itHallSensors.next()
    t15.start()
    t100.start()
    while(1){...espera interrupciones...} }
```



## Capítulo 6

# Interfaces de Módulos

En las siguientes descripciones se asume, por defecto, que en aquellos módulos que sean abstractos, los métodos definidos en su interfaz no son implementados. En los casos particulares en los que un módulo sea abstracto pero alguno de sus métodos sea implementado, tal método será indica en los comentarios del módulo.

### 6.1. Unidades de medida

		MG	DP	F
<b>Module</b>	<b>Measure</b>			
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real):Bool			
<b>comments</b>	Módulo abstracto que provee la interfaz para determinar y acceder, al valor de una unidad de medida.			

		MG	DP	F
<b>Module</b>	<b>Position inherits from Measure</b>			
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real) : Bool			
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz y que implementa la representación de una posición. En particular, se utilizará para representar la posición de una rueda que será un valor entero no negativo.			

		MG	DP	F
<b>Module</b>	<b>Percentage inherits from Measure</b>			
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real) : Bool			
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz y que implementa la representación de un valor porcentual. En particular, valores de <b>tensión</b> y valores de <b>corriente</b> provenientes de la PC.			

MG

DP

F

<b>Module</b>	<b>RPM inherits from Measure</b>
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real) : Bool
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz y que implementa la representación de valores de <b>velocidad</b> en RPM.

MG

DP

F

<b>Module</b>	<b>SignedPerc inherits from Measure</b>
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real) : Bool
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz y que implementa la representación de valores porcentuales signados. Esto es, valores en el intervalo $[-100, 100]$ . En particular, valores de <b>corriente</b> medida y <b>posición</b> del dispositivo de dirección.

MG

DP

F

<b>Module</b>	<b>MovementSense inherits from Measure</b>
<b>exportsproc</b>	value(): Real setValue(i Real) checkValue(i Real) : Bool
<b>comments</b>	Este es un módulo concreto que define todos los métodos de la interfaz e implementa un sentido de movimiento. Esto puede representar una dirección, izquierda/derecha; o bien una orientación de movimiento, adelante/atrás.

## 6.2. Conectores

MG

DP

F

<b>Module</b>	<b>Pipe</b>
<b>exportsproc</b>	read(): Measure write(i Measure)
<b>comments</b>	Este es un módulo concreto que implementa un tubo de conexión.

MG

DP

F

<b>Module</b>	<b>ConnectMCUtoPC</b>
<b>exportsproc</b>	write(i String)
<b>comments</b>	Este es un módulo concreto que implementa un conector entre el MCU y la PC, a fin de que el primero le envíe información al segundo, a través de una comunicación serie.

<b>Module</b>	<b>ConnectPCtoMCU</b>	MG	DP1	DP2	F
<b>exportsproc</b>	read(i String) charRCVHandler() emptyBuffer() getBuffer(): String addChar(i Char) changeState(i ConnectionState)				
<b>private</b>	st : ConnectionState				
<b>comments</b>	Este es un módulo concreto que implementa un conector entre el MCU y la PC, a fin de que el primero pueda obtener información del segundo, a través de una comunicación serie. Este conector mantiene un estado de conexión, del cual depende su comportamiento.				

Module	ConnectionState	MG	DP	F
exportsproc	processChar(i ConnectPCtoMCU, i Char) read(i ConnectPCtoMCU): String			
comments	Módulo abstracto que provee la interfaz para procesar la información proveniente de la PC. Representa el estado de la conexión entre la PC y el MCU.			

Module	NoMessage inherits from <span>ConnectionState</span>	<span>MG</span>	<span>DP</span>	<span>F</span>
exportsproc	NoMessage(i Transmitting) processChar(i ConnectPCtoMCU, i Char) read(i ConnectPCtoMCU): String			
comments	Módulo concreto que implementa el estado de la conexión, entre la PC y el MCU, en el que no hay mensaje.			

Module	Transmitting inherits from <span>ConnectionState</span>	MG	DP	F
exportsproc	processChar(i ConnectPCtoMCU, i Char) read(i ConnectPCtoMCU): String setNextState(i ConnectionState)			
comments	Módulo concreto que implementa el estado de la conexión, entre la PC y el MCU, en el cual se está transmitiendo un mensaje.			

<b>Module</b>	<b>ReadyMessage inherits from</b> <i>ConnectionState</i>
<b>exportsproc</b>	ReadyMessage(i NoMessage, i Transmitting) processChar(i ConnectPCtoMCU, i Char) read(i ConnectPCtoMCU): String
<b>comments</b>	Módulo concreto que implementa el estado de la conexión, entre la PC y el MCU, en el cual un mensaje ha sido transmitido y está listo para ser leído por el MCU.

### 6.3. Datos, algoritmos, funciones y constantes

<b>Module</b>	<b>MainCtrlData</b>
<b>exportsproc</b>	saveOrientation(i Measure) getOrientation: Measure saveNewOrientation(i Measure) getNewOrientation(): Measure saveStopOrder(i Bool) getStopOrder() : Bool saveModeId(i Int) getModeId() : Int
<b>comments</b>	Este es un módulo concreto que implementa información del controlador principal. En particular, si el robot se desplaza hacia adelante o hacia atrás, si habrá un cambio de esta orientación, si hay una orden de parada y cuál es el modo de funcionamiento.

<b>Module</b>	<b>Data</b>
<b>exportsproc</b>	saveSetpoint(i Measure) getSetpoint(): Measure
<b>comments</b>	Este es un módulo es abstracto. Provee una interfaz para guardar y acceder a los setpoints establecidos por las órdenes provenientes desde el exterior del sistema.

<b>Module</b>	<b>WheelCtrlData inherits from Data</b>
<b>exportsproc</b>	saveSetpoint(i Measure) getSetpoint() : Measure saveOrientation(i Measure) getOrientation() : Measure saveVel(i Measure) getVel() : Measure savePosition(i Measure) getPosition() : Measure velIsNull() : Bool saveCnt(i Measure) getCnt() : Measure saveTens(i Measure) getTens() : Measure addErrorVel(i Real) getErrorVel(i Real) resetErrorVel() addErrorCnt(i Real) getErrorCnt(i Real) resetErrorCnt()
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz. Permite guardar y acceder a la información necesaria para el funcionamiento sistemas de control de ruedas.

<b>Module</b>	<b>DirCtrlData inherits from Data</b>
<b>exportsproc</b>	saveSetpoint(i Measure) getSetpoint() : Measure saveDirection(i Measure) getDirection() : Measure savePosition(i Measure) getPosition() : Measure saveTempPos(i Measure) getTempPos() : Measure saveErrorPos(i Real) getErrorPos() : Real
<b>comments</b>	Módulo concreto que define todos los métodos de la interfaz. Permite guardar y acceder a la información necesaria para el funcionamiento del sistema de control de dirección.

<b>Module</b>	<b>Algorithm</b>
<b>exportsproc</b>	calculate(i Data)
<b>comments</b>	Este es un módulo abstracto que provee la interfaz para llevar a cabo un cálculo.

MG

DP

F

<b>Module</b>	<b>TensAlgorithm inherits from Algorithm</b>
<b>exportsproc</b>	calculate(i Data)
<b>comments</b>	Módulo concreto que implementa el algoritmo de control de rueda, el cual a partir de cierta orden de tensión, establece la tensión a aplicar a una rueda.

MG

DP

F

<b>Module</b>	<b>VelAlgorithm inherits from Algorithm</b>
<b>exportsproc</b>	calculate(i Data)
<b>comments</b>	Módulo concreto que implementa el algoritmo de control de rueda, el cual a partir de cierta orden de velocidad calcula la tensión a aplicar a la rueda.

MG

DP

F

<b>Module</b>	<b>CntAlgorithm inherits from Algorithm</b>
<b>exportsproc</b>	calculate(i Data)
<b>comments</b>	Módulo concreto que implementa el algoritmo de control de rueda, el cual a partir de cierto valor de corriente calcula la tensión a aplicar a la rueda.

MG

DP

F

<b>Module</b>	<b>DirAlgorithm inherits from Algorithm</b>
<b>exportsproc</b>	calculate(i Data)
<b>comments</b>	Módulo concreto que implementa el algoritmo de control de dirección el cual determina a partir de una posición deseada, la dirección de giro (derecha/izquierda).

MG

DP

F

<b>Module</b>	<b>Constants</b>
<b>exportsproc</b>	instance():Constants getPWM():Real getMAXRPMCR():Real getDELTAT():Real getM():Real
<b>private</b>	Constants() _instancia : Constants
<b>comments</b>	Este es un módulo concreto que implementa los métodos necesarios para acceder a valores constantes utilizados por diversos módulos. Este módulo es un singleton, por tanto mantendrá internamente un elemento <b>Constants</b> y su constructor no será público para los otros módulos.

MG

DP

F

**Module** **CalculationData**

**exportsproc**

```

setArg(i Real)
getArg(): Real
setResult(i Real)
getResult(): Real
getInMin(): Real
getInMax(): Real
getOutMin(): Real
getOutMax(): Real
getCalibration(): Real
setOrientation(i Real)
getOrientation(): Real

```

**comments** Este es un módulo abstracto que provee la interfaz para establecer y acceder a los datos, que serán argumento de un cómputo y a aquellos que serán el resultado de este.

MG

DP

F

**Module** **SimpleData inherits from CalculationData**

```

setArg(i Real)
getArg(): Real
setResult(i Real)
getResult(): Real

```

**comments** Módulo concreto que implementa los métodos que permiten guardar y recuperar el valor de un argumento simple y el resultado de aplicar una función a este.

MG

DP

F

**Module** **MoreData inherits from CalculationData**

```

setArg(i Real)
getArg(): Real
setResult(i Real)
getResult(): Real
getInMin(): Real
getInMax(): Real
getOutMin(): Real
getOutMax(): Real
getCalibration(): Real
setOrientation(i Real)
getOrientation(): Real

```

**private** dt : CalculationData

**comments** Módulo abstracto que contiene un elemento **CalculationData** y provee una interfaz común a módulos con más responsabilidades.

<b>Module</b>	<b>MapData inherits from MoreData</b> MapData(i Real,i Real,i Real,i Real,i SimpleData) setArg(i Real) getArg(): Real setResult(i Real) getResult(): Real getInMin(): Real getInMax(): Real getOutMin(): Real geOutMax(): Real
<b>private</b>	dt : CalculationData
<b>comments</b>	Módulo concreto que mantiene los valores requeridos por la función de mapeo <b>MapFunction</b> para llevar a delante el cálculo, permitiendo el acceso a ellos.

<b>Module</b>	<b>CRData inherits from MoreData</b> CRData(i Real,i MapData) getInMin(): Real getInMax(): Real getOutMin(): Real geOutMax(): Real setArg(i Real) getArg(): Real setResult(i Real) getResult(): Real getCalibration(): Real setOrientation(i Real) getOrientation(): Real
<b>private</b>	dt : CalculationData
<b>comments</b>	Módulo concreto que mantiene los valores requeridos por las funciones <b>MapFunction</b> y <b>OrientationCalc</b> para llevar a delante los cálculos, los cuales son requeridos en la recepción de órdenes provenientes del CR.

<b>Module</b>	<b>Function</b>
<b>exportsproc</b>	calculate(i CalculationData)
<b>comments</b>	Este es un módulo abstracto que provee la interfaz para llevar a cabo un cálculo.

<b>Module</b>	<b>OrientationCalc inherits from Function</b>
<b>exportsproc</b>	calculate(i CalculationData)
<b>comments</b>	Este es un módulo concreto que implementa la interfaz que permite calcular la orientación deseada del robot a partir de una orden proveniente del CR (ver <b>RF-22.</b> ).



MG

DP

F

<b>Module</b>	<b>MapFunction inherits from Function</b>
<b>exportsproc</b>	calculate(i CalculationData)
<b>comments</b>	Este es un módulo concreto que implementa la función de mapeo (ver <b>RF-13.</b> ) utilizada por distintos módulos del sistema.

MG

DP

F

<b>Module</b>	<b>InverseFunction inherits from Function</b>
<b>exportsproc</b>	calculate(i CalculationData)
<b>comments</b>	Este es un módulo concreto que implementa la función que transforma el valor de un registro (entero de 16 bits) en un valor $\mathbb{R}$ . Esta función es requerida por <b>ValueCollector</b> que recolecta las lecturas de la corriente de una rueda. (ver <b>RF-59.3.</b> ).

## 6.4. Dispositivos físicos

MG

DP1

DP2

F

<b>Module</b>	<b>Wheel</b>
<b>exportsproc</b>	Wheel(i Int) setTension(i Measure) setOrientation(i Measure) brake()
<b>comments</b>	Este es un módulo concreto que implementa los métodos necesarios para enviar señales a una rueda.

MG

DP

F

<b>Module</b>	<b>SteeringDevice</b>
<b>exportsproc</b>	right() left() disable() enable() pulse()
<b>comments</b>	Este es un módulo concreto que implementa los métodos necesarios para enviar señales al dispositivo de dirección.

MG

DP

F

<b>Module</b>	<b>Timer</b>
<b>exportsproc</b>	setPeriod(i Real) start() stop() tickHandler()
<b>comments</b>	Este es un módulo abstracto que provee la interfaz de un temporizador.

MG

DP

F

<b>Module</b>	<b>FirstTimer inherits from Timer</b>
<b>exportsproc</b>	FirstTimer(i ControllerTimeOut) setPeriod(i Real) start() stop() tickHandler()
<b>private</b>	cmd : ControllerTimeOut
<b>comments</b>	Este es un módulo concreto que implementa el temporizador principal del sistema. El método <u>tickHandler</u> será el manejador que responderá ante una interrupción física de un pulso de reloj, invocando el comando <b>ControllerTimeOut</b> .

MG

DP1

DP2

DP3

F

<b>Module</b>	<b>SecondTimer inherits from Timer</b>
<b>exportsproc</b>	SecondTimer(i ReadCnt,i ReadCnt,i ReadCnt,i ReadCnt,i DirCtrlTimeOut) setPeriod(i Real) start() stop() tickHandler()
<b>private</b>	cmdReadC1 : ReadCnt cmdReadC2 : ReadCnt cmdReadC3 : ReadCnt cmdReadC4 : ReadCnt cmdDirTimeOut : DirCtrlTimeOut
<b>comments</b>	Este es un módulo concreto que implementa el temporizador secundario del sistema. El método <u>tickHandler</u> será el manejador que responderá ante una interrupción física de un pulso de reloj, invocando los comandos de lectura de corriente de cada rueda <b>ReadCnt</b> y el comando <b>DirCtrlTimeOut</b> que indica al sistema de dirección una marca de reloj.

MG

DP

F

Module	Pin
<b>exportsproc</b>	Pin(i Int) signalHandler() setComand(i Command)
<b>comments</b>	Este es un módulo concreto que implementa un pin de conexión entre el receptor del CR y el MCU. En particular, implementará el pin que recibirá órdenes de velocidad y aquel que recibirá órdenes de dirección, desde el CR.

## 6.5. Recolectores de señales físicas

MG

DP

F

Module	Collector
<b>exportsproc</b>	currentTime() getCurrentTime():Real
<b>comments</b>	Este es un módulo abstracto que provee la interfaz para registrar y obtener un instante de tiempo.

MG

DP

F

Module	TimeCollector inherits from Collector
<b>exportsproc</b>	currentTime() getCurrentTime():Real
<b>comments</b>	Este es un módulo concreto e implementa los métodos que permiten registrar y obtener un instante de tiempo.

MG

DP

F

Module	DecoCollector inherits from Collector
<b>exportsproc</b>	currentTime() getCurrentTime():Real
<b>comments</b>	Este es un módulo abstracto que no implementa ninguno de sus métodos, pero que contiene un elemento <b>Collector</b> y provee una interfaz común a otros módulos con responsabilidades adicionales.

		MG	DP1	DP2	F
<b>Module</b>	<b>CRpinCollector inherits from DecoCollector</b>				
<b>exportsproc</b>	CRpinCollecto(i TimeCollector) currentTime() getCurrentTime():Real periodTime():Real initPeriod()				
<b>comments</b>	Este es un módulo concreto que se encarga, ante cada interrupción de un pin del CR, de recolectar los instantes en los que esto ocurre y llevar a cabo ciertos cálculos.				

		MG	DP1	DP2	F
<b>Module</b>	<b>SensorCollector inherits from DecoCollector</b>				
<b>exportsproc</b>	SensorCollecto(i TimeCollector) currentTime() getCurrentTime():Real addOne() preSum():Int total():Int periodTime():Real initPeriod()				
<b>comments</b>	Este es un módulo concreto que se encarga, ante cada interrupción de un sensor Hall, de recolectar los instantes en los que esto ocurre y llevar a cabo ciertos cálculos.				

		MG	DP1	DP2	F
<b>Module</b>	<b>ValueCollector</b>				
<b>exportsproc</b>	ValueCollector(i Int) initPeriod() getCurrentVal():Real currentVal() addOne()				
<b>comments</b>	Este es un módulo concreto que se encarga de obtener las distintas lecturas del valor de corriente de una rueda y llevar a cabo ciertos cálculos.				

## 6.6. Sensores y buffers

		MG	DP	F
Module	PassiveSensor			
exportsproc	setConnection(i Pipe) signal()			
comments	Este es un módulo abstracto que define la interfaz de un sensor pasivo; esto es, uno que no genera interrupciones físicas.			

MG

DP

F

<b>Module</b>	<b>VelSensor inherits from <i>PassiveSensor</i></b>
<b>exportsproc</b>	VelSensor(i SensorCollector) setConnection(i Pipe) signal() getPosition():Measure
<b>private</b>	pipe : Pipe vColl : SensorCollector vPos : Position
<b>comments</b>	Este es un módulo concreto que implementa un sensor de velocidad de una rueda.

MG

DP

F

<b>Module</b>	<b>CntSensor inherits from <i>PassiveSensor</i></b>
<b>exportsproc</b>	CntSensor(i Int) setConnection(i Pipe) signal()
<b>private</b>	pipe : Pipe
<b>comments</b>	Este es un módulo concreto que implementa un sensor de corriente.

MG

DP

F

<b>Module</b>	<b>DirSensor inherits from <i>PassiveSensor</i></b>
<b>exportsproc</b>	setConnection(i Pipe) signal()
<b>private</b>	pipe : Pipe
<b>comments</b>	Este es un módulo concreto que implementa un sensor del dispositivo de dirección.

MG

DP1

DP2

F

<b>Module</b>	<b>ActiveSensor</b>
<b>exportsproc</b>	ActiveSensor(i Int) start() stop() signalHandler() setCommand(i Command)
<b>private</b>	cmd : Command
<b>comments</b>	Este es un módulo concreto que implementa un sensor Hall. El método <u>signalHandler</u> manejará cada interrupción física proveniente del mencionado sensor.

<b>Module</b>	<b>CRBuffer</b>	MG	DP1	DP2	F
<b>exportsproc</b>	getVal(): Real empty():Bool calcOrientation()				
<b>comments</b>	Este es un módulo abstracto que define la interfaz de un buffer receptor de una orden del CR. Solo el método <u>getVal</u> es implementado.				

<b>Module</b>	<b>VelBuffer inherits from CRBuffer</b>	MG	DP1	DP2	F
<b>exportsproc</b>	empty():Bool calcOrientation() getOrientation():Real				
<b>comments</b>	Este es un módulo concreto que implementa el buffer receptor de las órdenes de velocidad provenientes del CR.				

<b>Module</b>	<b>DirBuffer inherits from CRBuffer</b>	MG	DP1	DP2	F
<b>exportsproc</b>	empty():Bool calcOrientation()				
<b>comments</b>	Este es un módulo concreto que implementa el buffer receptor de las órdenes de dirección provenientes del CR.				

## 6.7. Sistemas de control

### 6.7.1. Sistema de control de rueda

Module	WSysOrder	MG	DP	F
exportsproc	execute(i WheelSystem)			
comments	Este es un módulo abstracto que define la interfaz de una orden a llevar a cabo sobre un sistema de control de rueda.			

Module	SaveWPosition inherits from WSysOrder	MG	DP	F
exportsproc	execute(i WheelSystem)			
comments	Este es un módulo concreto que implementa una orden sobre un sistema de control de rueda. En particular, registra la posición de la rueda en un momento determinado.			

MG

DP1

DP2

F

<b>Module</b>	<b>SensorWritesVel inherits from WSysOrder</b>
<b>exportsproc</b>	execute(i WheelSystem)
<b>comments</b>	Este es un módulo concreto que implementa una orden sobre un sistema de control de rueda. En particular, indica al sensor de velocidad que emita el valor medido.

MG

DP1

DP2

F

<b>Module</b>	<b>ControllerReadsVel inherits from WSysOrder</b>
<b>exportsproc</b>	execute(i WheelSystem)
<b>comments</b>	Este es un módulo concreto que implementa una orden sobre un sistema de control de rueda. En particular, indica al controlador que obtenga la velocidad medida.

MG

DP

F

<b>Module</b>	<b>SensorWritesCnt inherits from WSysOrder</b>
<b>exportsproc</b>	execute(i WheelSystem)
<b>comments</b>	Este es un módulo concreto que implementa una orden sobre un sistema de control de rueda. En particular, indica al sensor de corriente que emita el valor medido.

MG

DP

F

<b>Module</b>	<b>ControllerReadsCnt inherits from WSysOrder</b>
<b>exportsproc</b>	execute(i WheelSystem)
<b>comments</b>	Este es un módulo concreto que implementa una orden sobre un sistema de control de rueda. En particular, indica al controlador que obtenga la corriente medida.

MG

DP

F

<b>Module</b>	<b>WCtrlCommand</b>
<b>exportsproc</b>	execute()
<b>comments</b>	Este es un módulo abstracto que define la interfaz que ejecuta un comando, que tendrá efecto sobre el comportamiento de una rueda.

MG

DP

F

<b>Module</b>	<b>VelNull</b>
<b>exportsproc</b>	VelNull(i Wheel) execute()
<b>comments</b>	Este es un módulo concreto que implementa el comando que provee tensión nula a una rueda.

MG

DP

F

**Module****Brake****exportsproc**

```
Brake(i Wheel)
execute()
```

**comments**

Este es un módulo concreto que implementa el comando que envía la señal de frenado a una rueda.

MG

DP1

DP2

F

**Module****SetTension****exportsproc**

```
SetTension(i Wheel, i WheelCtrlData, i WheelController)
execute()
```

**comments**

Este es un módulo concreto que implementa el comando que envía la tensión correspondiente a una rueda.

MG

DP

F

**Module****ChangeOrientation****exportsproc**

```
ChangeOrientation(i Wheel, i WheelCtrlData, i WheelSystem, i SensorWritesVel,
i SensorsWritesCnt)
execute()
```

**comments**

Este es un módulo concreto que implementa el comando que lleva a cabo el cambio de orientación (adelante/atrás) de una rueda.

MG

DP

F

**Module****CtrlCmdPool****exportsproc**

```
getBreak():CtrlCommand
getVelNull():CtrlCommand
getSetTension():CtrlCommand
getChangeOrient():CtrlCommand
```

**comments**

Este es un módulo concreto que está constituido por cuatro comandos de control de rueda; y que provee la interfaz correspondiente para acceder a estos.

MG

DP1

DP2

F

**Module****WCtrlAlgorithm****exportsproc**

```
execute(i CtrlCmdPool)
```

**comments**

Este es un módulo abstracto que provee la interfaz para ejecutar un algoritmo el cual utiliza un conjunto de comandos, y que tendrá efecto sobre el comportamiento de una rueda.



MG DP1 DP2 DP3 F

**Module** Stop inherits from **WCtrlAlgorithm**

**exportsproc** execute(i CtrlCmdPool)

**comments** Este es un módulo concreto que implementa el algoritmo por medio del cual se frena una rueda.

MG DP1 DP2 DP3 F

**Module** ResetVel inherits from **WCtrlAlgorithm**

**exportsproc** execute(i CtrlCmdPool)

**comments** Este es un módulo concreto que implementa el algoritmo por medio del cual se le envía la tensión inicial a una rueda (en particular, tensión nula).

MG DP1 DP2 DP3 F

**Module** Advance inherits from **WCtrlAlgorithm**

**exportsproc** execute(i CtrlCmdPool)

**comments** Este es un módulo concreto que implementa el algoritmo por medio del cual se envía tensión a una rueda para que esta se mueva.

MG DP F

**Module** ReverseOrientation inherits from **WCtrlAlgorithm**

**exportsproc** execute(i CtrlCmdPool)

**private** wcmd:WCtrlCommand

**comments** Este es un módulo abstracto que provee una interfaz común a módulos con más responsabilidades. Estas responsabilidades implican no solo mover una rueda, sino también cambiar el sentido del movimiento (adelante/atrás). Los herederos de este módulo tendrán al menos un elemento **WCtrlAlgorithm**.

MG DP1 DP2 DP3 F

**Module** Reverse inherits from **ReverseOrientation**

**exportsproc** Reverse(i Advance)  
execute(i CtrlCmdPool)

**private** wcmd:WCtrlCommand

**comments** Este es un módulo concreto que implementa el algoritmo por medio del cual una rueda cambiará el sentido de movimiento (adelante/atrás) y avanzará. Este algoritmo será utilizado cuando alguno de los valores de referencia de alguna de las ruedas sea corriente o tensión. Ver requerimiento **RF-68**.

<b>Module</b>	<b>ReverseRPM inherits from <a href="#">ReverseOrientation</a></b>	<a href="#">MG</a>	<a href="#">DP1</a>	<a href="#">DP2</a>	<a href="#">F</a>
<b>exportsproc</b>	ReverseRPM(i Stop, i Reverse) execute(i CtrlCmdPool)				
<b>private</b>	wcmd:WCtrlCommand				
<b>comments</b>	Este es un módulo concreto que implementa el algoritmo por medio del cual una rueda cambiará el sentido de movimiento (adelante/atrás) y avanzará. Este algoritmo será utilizado cuando los valores de referencia de las cuatro ruedas sean RPM. Ver requerimiento <a href="#">RF-67</a> .				

<b>Module</b>	<b>WheelController</b>	<a href="#">MG</a>	<a href="#">DP1</a>	<a href="#">DP2</a>	<a href="#">F</a>
<b>exportsproc</b>	WheelController(i WheelCtrlData) setConnectionV(i Pipe) setConnectionC(i Pipe) readConnectionV() readConnectionC() setSetpoint(i Measure) setOrientation(i Measure) setAlgorithm(i Algorithm) getAlgorithm(): Algorithm setCtrlAlgorithm(i WCtrlAlgorithm) setCtrlCmdPool(i CtrlCmdPool) getData(): Data control()				
<b>private</b>	cmdPool : CtrlCmdPool data : Data alg : Algorithm ctrlAlg : WCtrlAlgorithm pipeV : Pipe pipeC : Pipe				
<b>comments</b>	Este es un módulo concreto que implementa el controlador de una rueda.				

<b>Module</b>	<b>WheelSystem</b>	<a href="#">MG</a>	<a href="#">DP1</a>	<a href="#">DP2</a>	<a href="#">F</a>
<b>exportsproc</b>	WheelSystem(i VelSensor, i CntSensor) setController(i WheelController) getController() : WheelController getVelSensor() : PassiveSensor getCntSensor() : PassiveSensor				
<b>comments</b>	Este es un módulo concreto que implementa un sistema de control de rueda como un conjunto de elementos y provee los métodos que permiten el acceso a estos.				

### 6.7.2. Sistema de control de dirección

<b>Module</b>	<b>DCtrlCommand</b>	MG	DP	F
<b>exportsproc</b>	execute()			
<b>comments</b>	Este es un módulo abstracto que provee la interfaz para ejecutar un comando que tendrá efectos sobre el dispositivo de dirección.			

<b>Module</b>	<b>Turn inherits from DCtrlCommand</b>	MG	DP	F
<b>exportsproc</b>	Turn(i SteeringDevice) execute()			
<b>comments</b>	Este es un módulo concreto que permite ejecutar el comando a través del cual se envían señales de pulso, para que el dispositivo de dirección lleve a cabo un paso de giro.			

<b>Module</b>	<b>SetDirection inherits from DCtrlCommand</b>	MG	DP	F
<b>exportsproc</b>	SetDirecction(i SteeringDevice, i DirCtrlData) execute()			
<b>comments</b>	Este es un módulo concreto que permite ejecutar el comando a través del cual se establece la dirección hacia donde debe girar (izq./der.) el dispositivo de dirección, de acuerdo a una orden dada.			

<b>Module</b>	<b>Enable inherits from DCtrlCommand</b>	MG	DP	F
<b>exportsproc</b>	Enable(i SteeringDevice) execute()			
<b>comments</b>	Este es un módulo concreto que permite ejecutar el comando a través del cual se enciende (o habilita) el dispositivo de dirección.			

<b>Module</b>	<b>Disable inherits from DCtrlCommand</b>	MG	DP	F
<b>exportsproc</b>	Disable(i SteeringDevice) execute()			
<b>comments</b>	Este es un módulo concreto que permite ejecutar el comando a través del cual se apaga (o deshabilita) el dispositivo de dirección.			

MG

DP

F

**Module****DeviceState****exportsproc**

```
on()
off()
```

**comments**

Este es un módulo abstracto que provee la interfaz de los estados de encendido y apagado del dispositivo de dirección. Provee los métodos necesarios para encender y apagar el dispositivo.

MG

DP1

DP2

F

**Module****OnState inherits from DeviceState****exportsproc**

```
OnState(i DCtrlCommand, i DeviceState)
on()
off()
```

**comments**

Este es un módulo concreto que implementa el estado de encendido del dispositivo de dirección.

MG

DP1

DP2

F

**Module****OffState inherits from DeviceState****exportsproc**

```
OffState(i DCtrlCommand)
on()
off()
setNexState(i DeviceState)
```

**comments**

Este es un módulo concreto que implementa el estado de apagado del dispositivo de dirección.

MG

DP

F

**Module****DOperationState****exportsproc**

```
control(i DirController)
turn(i DirController)
oneWNull(i DirController)
twoWNull(i DirController)
```

**comments**

Este es un módulo abstracto que provee la interfaz de los estados de operación del controlador del dispositivo de dirección, donde el controlador llevará a cabo el control o el giro.

MG

DP

F

<b>Module</b>	<b>DInactive inherits from DOperationState</b>
<b>exportsproc</b>	control(i DirController) turn(i DirController) oneWNull(i DirController) twoWNull(i DirController)
<b>comments</b>	Este es un módulo concreto que implementa cómo serán el control y el giro del dispositivo de dirección cuando el controlador esté inactivo.

MG

DP1

DP2

F

<b>Module</b>	<b>DTurning inherits from DOperationState</b>
<b>exportsproc</b>	DTurning(i Turn, i DActive) control(i DirController) turn(i DirController) oneWNull(i DirController) twoWNull(i DirController)
<b>comments</b>	Este es un módulo concreto que implementa cómo serán el control y el giro del dispositivo de dirección cuando el controlador esté girando el dispositivo.

MG

DP1

DP2

DP3

DP4

F

<b>Module</b>	<b>DActive inherits from DOperationState</b>
<b>exportsproc</b>	DActive(i DTurning, i DStopping, iDInactive, i SetDirection) control(i DirController) turn(i DirController) oneWNull(i DirController) twoWNull(i DirController)
<b>comments</b>	Este es un módulo concreto que implementa cómo serán el control y el giro del dispositivo de dirección cuando el controlador esté activo esperando una orden.

MG

DP1

DP2

F

<b>Module</b>	<b>DStopping inherits from DOperationState</b>
<b>exportsproc</b>	DStopping(i DInactive) control(i DirController) turn(i DirController) oneWNull(i DirController) twoWNull(i DirController)
<b>comments</b>	Este es un módulo concreto que implementa cómo serán el control y el giro del dispositivo de dirección cuando el controlador esté deteniendo el dispositivo.

Module	DirController
<b>exportsproc</b>	DirController(i Timer) setConnection(i Pipe) readConnection() readTempPos() setSetpoint(i Measure) getData() : Data getAlgorithm() : Algorithm changeDvState(i DeviceState) changeOpState(i DOperationState) turn() control() oneWNull() twoWNull() on() off()
<b>private</b>	data : Data alg : Algorithm opSt : DOperationState dvSt : DeviceState pipe : Pipe
<b>comments</b>	Este es un módulo concreto que implementa el controlador del dispositivo de dirección.

Module	DirSystem
<b>exportsproc</b>	DisSystem(i DirController,i DirSensor) getController() : DirController getDirSensor(): PassiveSensor
<b>private</b>	dCtrller : DirController dSensor : DirSensor
<b>comments</b>	Este es un módulo concreto que implementa un sistema de control de dirección como un conjunto de elementos, y provee los métodos para acceder a estos.

## 6.8. Controlador principal, órdenes, estados y modos de operación

Module	Mode
<b>exportsproc</b>	newMessage():Bool read(i MainController) changeMode(i MainController)
<b>comments</b>	Este es un módulo abstracto que provee la interfaz de los modos (PC/CR) de recepción de órdenes del sistema, de acuerdo a su procedencia.

MG

DP

F

<b>Module</b>	<b>BasicMode inherits from Mode</b>
<b>exportsproc</b>	BasicMode(i Reader) newMessage():Bool read(i MainController)
<b>comments</b>	Este es un módulo concreto que implementa un modo de recepción de órdenes básico, donde las mismas son obtenidas a través de un lector Reader.

MG

DP

F

<b>Module</b>	<b>LectureMode inherits from BasicMode</b>
<b>exportsproc</b>	newMessage():Bool read(i MainController) changeMode(i MainController)
<b>comments</b>	Este es un módulo abstracto y provee una interfaz común a módulos que extenderán sus responsabilidades, permitiendo un modo de obtención de órdenes provenientes de la PC o del CR.

MG

DP1

DP2

F

<b>Module</b>	<b>CR inherits from LectureMode</b>
<b>exportsproc</b>	CR(i BasicMode) newMessage():Bool read(i MainController) changeMode(i MainController) setNextState(i Mode)
<b>comments</b>	Este es un módulo concreto que implementa el estado del sistema en el cual las órdenes son leídas desde el CR.

MG

DP1

DP2

F

<b>Module</b>	<b>PC inherits from LectureMode</b>
<b>exportsproc</b>	PC(i BasicMode, i CR) newMessage():Bool read(i MainController) changeMode(i MainController)
<b>comments</b>	Este es un módulo concreto que implementa el estado del sistema en el cual las órdenes son leídas desde la PC.

MG

DP

F

**Module****ModePool****exportsproc**

```
ModePool(i PC, i CR)
getPCMode():Mode
getCRMode():Mode
```

**comments**

Este es un módulo concreto que agrupa los modos de lectura de órdenes con los que cuenta el sistema, y provee los métodos de acceso a estos.

MG

DP

F

**Module****Order****exportsproc**

```
execute(i ControlSystemPool)
actionOnWheelSys(i WheelSystem)
actionOnDirSys(i DirSystem)
```

**comments**

Este es un módulo abstracto que provee la interfaz para llevar a cabo órdenes sobre los sensores y los controladores del sistema. Solo implementa el método execute.

MG

DP1

DP2

F

**Module****SaveWheelPositions inherits from Order****exportsproc**

```
SaveWheelPositions(i SaveWPosition)
actionOnWheelSys(i WheelSystem)
actionOnDirSys(i DirSystem)
```

**private**

```
ordWP:SaveWPosition()
```

**comments**

Este es un módulo concreto que implementa la orden por medio de la cual se guardan las posiciones de las ruedas.

MG

DP1

DP2

F

**Module****SensorsWrite inherits from Order****exportsproc**

```
SensorsWrite(i SensorWritesVel, i SensorWritesCnt)
actionOnWheelSys(i WheelSystem)
actionOnDirSys(i DirSystem)
```

**private**

```
ordV:ControllerReadsVel()
ordC:ControllerReadsCnt()
```

**comments**

Este es un módulo concreto que implementa la orden para que los sensores escriban en los correspondientes **Pipe** los valores medidos.





<b>Module</b>	<b>OperationState</b>	MG	DP1	DP2	F
<b>exportsproc</b>	<pre>read(i MainController) actionWithMsg(i MainController, i Mode) actionNoMsg(i MainController) control(i MainController) write(i MainController, i Writer)</pre>				
<b>comments</b>	Este es un módulo abstracto que provee la interfaz de los estados de operación del controlador principal.				

<b>Module</b>	<b>WaitingN inherits from OperationState</b>	MG	DP1	DP2	DP3	F
<b>exportsproc</b>	<pre>WaitingN(i Working, i OperationState) actionWithMsg(i MainController, i Mode) actonNoMsg(i MainController) control(i MainController) write(i MainController, i Writer)</pre>					
<b>private</b>	mainCtrlOrder: MainCtrlOrder					
<b>comments</b>	Este es un módulo concreto que implementa el estado n-ésimo de espera, por una orden, del controlador principal.					

<b>Module</b>	<b>WaitingMAX inherits from OperationState</b>	MG	DP1	DP2	DP3	F
<b>exportsproc</b>	<pre>WaitingMAX(i Working, i Reconnecting, i ControllersStop) actionWithMsg(i MainController, i Mode) actonNoMsg(i MainController) control(i MainController) write(i MainController, i Writer)</pre>					
<b>private</b>	<pre>controllersStop: ControllersStop mainCtrlOrder: MainCtrlOrder</pre>					
<b>comments</b>	Este es un módulo concreto que implementa el último estado de espera, por una orden, del controlador principal.					

<b>Module</b>	<b>Reconnecting inherits from</b> <span>OperationState</span>	<span>MG</span>	<span>DP1</span>	<span>DP2</span>	<span>DP3</span>	<span>F</span>
<b>exportsproc</b>	Reconnecting(i Working) actionWithMsg(i MainController, i Mode) actonNoMsg(i MainController) control(i MainController) write(i MainController, i Writer)					
<b>private</b>	mainCtrlOrder: MainCtrlOrder					
<b>comments</b>	Este es un módulo concreto que implementa el estado en el cual el controlador principal intenta restablecer la conexión con la PC o el CR.					

<b>Module</b>	<b>Working inherits from</b> <span>OperationState</span>	<span>MG</span>	<span>DP1</span>	<span>DP2</span>	<span>DP3</span>	<span>F</span>
<b>exportsproc</b>	actionWithMsg(i MainController, i Mode) actonNoMsg(i MainController) control(i MainController) write(i MainController, i Writer)					
<b>private</b>	mainCtrlOrder: MainCtrlOrder					
<b>comments</b>	Este es un módulo concreto que implementa el estado en el cual el controlador principal está funcionando correctamente.					

Module	Serializable	MG	DP	F
exportsproc	readFrom(i Reader) writeTo(i Writer)			
comments	Este es un módulo abstracto que provee la interfaz para leer y escribir, a través de un módulo lector y uno escribiente, información desde o hacia un origen o destino de datos.			

		MG	DP1	DP2	DP3	DP4	F
<b>Module</b>	<b>MainController inherits from <i>Serializable</i></b>						
<b>exportsproc</b>	MainController(i ControlSystemPool) readFrom(i Reader) writeTo(i Writer) changeState(i OperationState) updateOrientation() changeMode(i Mode) getMode() : Mode getData() : MainCtrlData getCtrlSysPool() : ControlSystemPool control() read()						
<b>private</b>	csp : ControlSystemPool md : Mode st : OperationState mData : MainCtrlData						
<b>comments</b>	Este es un módulo concreto que implementa el controlador principal del sistema.						

		MG	DP1	DP2	F
Module	ControlSystemPool inherits from <span>Serializable</span>				
exportsproc	ControlSystemPool(i WheelSystem, i WheelSystem, i WheelSystem, i WheelSystem, i DirSystem, i DActive, i DStopping) readFrom(i Reader) writeTo(i Writer) wheelSystemIt() : Iterator<WheelSystem> getDirSystem() : DirSystem				
private	wSysDD : WheelSystem wSysDI : WheelSystem wSysTD : WheelSystem wSysTI : WheelSystem dSys : DirSystem				
comments	Este es un módulo concreto que agrupa los sistemas de control del MCU.				

## 6.9. Lectura y escritura de información

		MG	DP	F
Module	Reader			
exportsproc	readSetpointDD() : Measure readSetpointDI() : Measure readSetpointTD() : Measure readSetpointTI() : Measure readSetpointOrient() : Measure readAlgorithmDD() : Algorithm readAlgorithmDI() : Algorithm readAlgorithmTD() : Algorithm readAlgorithmTI() : Algorithm readSetpointDir() : Measure readMode(i Mode) newMessage() : Bool readStop() : Bool isRPM() : Bool changeOrientation() : Bool setModes(i ModePool) readRoot(i Serializable)			
comments	Este es un módulo abstracto que provee la interfaz para leer información de un origen de datos.			

		MG	DP	F
Module	SerialReader inherits from Reader			
exportsproc	SerialReader(i ConnectPCtoMCU,i TensAlgorithm,i VelAlgorithm, i CntAlgorithm) readSetpointDD() : Measure readSetpointDI() : Measure readSetpointTD() : Measure readSetpointTI() : Measure readSetpointOrient() : Measure readAlgorithmDD() : Algorithm readAlgorithmDI() : Algorithm readAlgorithmTD() : Algorithm readAlgorithmTI() : Algorithm readSetpointDir() : Measure readMode(i Mode) newMessage() : Bool readStop() : Bool isRPM() : Bool changeOrientation() : Bool setModes(i ModePool) readRoot(i Serializable)			
comments	Este es un módulo concreto que implementa un lector de la conexión serial entre la PC y el MCU.			

**Module****BufferReader inherits from Reader****exportsproc**

```

BufferReader(i VelBuffer,i DirBuffer,i VelAlgorithm)
readSetpointDD() : Measure
readSetpointDI() : Measure
readSetpointTD() : Measure
readSetpointTI() : Measure
readSetpointOrient() : Measure
readAlgorithmDD() : Algorithm
readAlgorithmDI() : Algorithm
readAlgorithmTD() : Algorithm
readAlgorithmTI() : Algorithm
readSetpointDir() : Measure
readMode(i Mode)
newMessage() : Bool
readStop() : Bool
isRPM() : Bool
changeOrientation() : Bool
setModes(i ModePool)
readRoot(i Serializable)

```

**comments**

Este es un módulo concreto que implementa un lector de la conexión entre el receptor de señales del CR y el MCU.

**Module****Writer****exportsproc**

```

writeVelDD(i Measure)
writeVelDI(i Measure)
writeVelTD(i Measure)
writeVelTI(i Measure)
writeCntDD(i Measure)
writeCntDI(i Measure)
writeCntTD(i Measure)
writeCntTI(i Measure)
writePosDD(i Measure)
writePosDI(i Measure)
writePosTD(i Measure)
writePosTI(i Measure)
writeTensDD(i Measure)
writeTensDT(i Measure)
writeTensTD(i Measure)
writeTensTI(i Measure)
writeDPosition(i Measure)
writeOrientation(i Measure)
writeStopOrder(i Bool)
writeModeId(i Int)
writeAll()

```

**comments**

Este es un módulo abstracto que provee la interfaz para escribir información en un destino de datos.

Module	SerialWriter inherits from Writer
exportsproc	SerialWriter(i ConnectMCUtoPC) writeVelDD(i Measure) writeVelDI(i Measure) writeVelTD(i Measure) writeVelTI(i Measure) writeCntDD(i Measure) writeCntDI(i Measure) writeCntTD(i Measure) writeCntTI(i Measure) writePosDD(i Measure) writePosDI(i Measure) writePosTD(i Measure) writePosTI(i Measure) writeTensDD(i Measure) writeTensDT(i Measure) writeTensTD(i Measure) writeTensTI(i Measure) writeDPosition(i Measure) writeOrientation(i Measure) writeStopOrder(i Bool) writeModeId(i Int) writeAll()
comments	Este es un módulo concreto de escritura que permite escribir en una conexión serial entre el MCU y la PC.

## 6.10. Comandos utilizados como manejadores de señales físicas

Module	Command
exportsproc	execute()
comments	Este es un módulo abstracto que provee la interfaz para ejecutar un comando.

Module	DirCtrlTimeOut inherits from Command
exportsproc	DirCtrlTimeOut(i DirController, i DirSensor) execute()
comments	Este es un módulo concreto el cual implementa un comando que permitirá que el dispositivo de dirección gire cuando corresponda, ante una interrupción del temporizador secundario SecondTimer.

MG

DP

F

<b>Module</b>	<b>CountTime inherits from Command</b>
<b>exportsproc</b>	CountSignal(i CRpinCollector) execute()
<b>comments</b>	Este es un módulo concreto el cual implementa el comando que registrará los intervalos de tiempo entre las interrupciones recibidas por un Pin del CR.

MG

DP

F

<b>Module</b>	<b>CountSignal inherits from Command</b>
<b>exportsproc</b>	CountSignal(i SensorCollector) execute()
<b>comments</b>	Este es un módulo concreto el cual implementa un comando que contabiliza las interrupciones físicas provenientes de un sensor Hall ActiveSensor.

MG

DP1

DP2

F

<b>Module</b>	<b>ReadCnt inherits from Command</b>
<b>exportsproc</b>	ReadCnt(i ValueCollector) execute()
<b>comments</b>	Este es un módulo concreto el cual implementa un comando que registra las distintas mediciones de corriente, que se llevarán a cabo ante la interrupción del temporizador secundario SecondTimer.

MG

DP

F

<b>Module</b>	<b>ControllerTimeOut inherits from Command</b>
<b>exportsproc</b>	ControllerTimeOut(i MainController, i SerialReader, i SerialWriter) execute()
<b>comments</b>	Este es un módulo concreto el cual implementa un comando que permitirá al controlador principal llevar a cabo las funciones correspondientes para el control de los subsistemas de control, ante una interrupción del temporizador principal FirstTimer.

## 6.11. Construcción de objetos

MG

DP

F

<b>Module</b>	<b>WSDirector</b>
<b>exportsproc</b>	WSDirector(i WheelSysBuilder) build(i Int)
<b>comments</b>	Este es un módulo concreto el cual es responsable de dirigir la construcción de los objetos que constituyen un sistema de control de ruedas.



MG

DP

F

**Module**                      **WheelSysBuilder**

**exportsproc**                buildController(i Int)  
                               buildCmdPool(i Int)  
                               buildSensors(i Int)  
                               buildConnections()

**comments**                    Este es un módulo abstracto el cual provee la interfaz que permite construir las distintas partes que constituyen un sistema de control de ruedas.

MG

DP1

DP2

DP3

DP4

F

**Module**                      **WSBuilder inherits from** WheelSysBuilder

**exportsproc**                buildController(i Int)  
                               buildCmdPool(i Int)  
                               buildSensors(i Int)  
                               buildConnections()  
                               getSystem() : WheelSystem  
                               getCntCmd() : Command  
                               getActiveSensor() : ActiveSensor

**comments**                    Este es un módulo concreto que permite construir las distintas partes que constituyen un sistema de control de ruedas.

MG

DP

F

**Module**                      **DSDirector**

**exportsproc**                DSDirector(i DirSysBuilder)  
                               build()

**comments**                    Este es un módulo concreto el cual es responsable de dirigir la construcción de los objetos que constituyen un sistema de control de dirección.

MG

DP

F

**Module**                      **DirSysBuilder**

**exportsproc**                buildController()  
                               buildSensor()  
                               buildConnection()

**comments**                    Este es un módulo abstracto el cual provee la interfaz que permite construir las distintas partes que constituyen un sistema de control de dirección.

<b>Module</b>	<b>DSBuilder inherits from DirSysBuilder</b>	MG	DP1	DP2	DP3	F
<b>exportsproc</b>	buildController() buildSensor() buildConnection() getSystem() : DirSystem getDirCmd() : Command getDActiveState() : DOperationState getDStoppingState() : DOperationState					
<b>comments</b>	Este es un módulo concreto que permite construir las distintas partes que constituyen un sistema de control de dirección.					

<b>Module</b>	<b>CSPDirector</b>	MG	DP	F
<b>exportsproc</b>	CSPDirector(i CtrlSysPoolBuilder) build()			
<b>comments</b>	Este es un módulo concreto el cual es responsable de dirigir la construcción del conjunto de sistemas de control.			

<b>Module</b>	<b>CtrlSysPoolBuilder</b>	MG	DP	F
<b>exportsproc</b>	buildWheelSystem() buildDirSystem()			
<b>comments</b>	Este es un módulo abstracto el cual provee la interfaz que permite construir las distintas partes que constituyen un conjunto de sistemas de control.			

<b>Module</b>	<b>CSPBuilder inherits from CtrlSysPoolBuilder</b>	MG	DP	F
<b>exportsproc</b>	buildWheelSystem() buildDirSystem() getLocalTimer() : Timer getCSP() : ControlSystemPool getActiveSensors() : Iterator<ActiveSensor> getDStoppingState() : DOperationState			
<b>private</b>	wsBuilder: WSBuilder wsDirector: WSDirector dsDBuilder: DSBuilder dsDirector: DSDirector			
<b>comments</b>	Este es un módulo concreto que permite construir las distintas partes que constituyen un conjunto de sistemas de control.			

MG

DP

F

<b>Module</b>	<b>MCDirector</b>
<b>exportsproc</b>	MCDirector(i MainControllerBuilder) build()
<b>comments</b>	Este es un módulo concreto el cual es responsable de dirigir la construcción del controlador principal.

MG

DP

F

<b>Module</b>	<b>MainControllerBuilder</b>
<b>exportsproc</b>	buildSensorOrders() buildCtrlSysPool() buildCtrllersOrders() buildMainOrder() buildOpStates()
<b>comments</b>	Este es un módulo abstracto el cual provee la interfaz que permite construir las distintas partes que constituyen al controlador principal.

MG

DP

F

<b>Module</b>	<b>MCBuilder inherits from MainControllerBuilder</b>
<b>exportsproc</b>	MCBuilder(i ModePool) buildSensorOrders() buildCtrlSysPool() buildCtrllersOrders() buildMainOrder() buildOpStates() getLocalTimer():Timer getMC():MainController getActiveSensors():Iterator <ActiveSensors>
<b>private</b>	cspBuilder: CSPBuilder cspDirector: CSPDirector
<b>comments</b>	Este es un módulo concreto que permite construir las distintas partes que constituyen el controlador principal.

## 6.12. Programa principal

MG

DP1

DP2

F

<b>Module</b>	<b>Main</b>
<b>exportsproc</b>	main()
<b>comments</b>	Este es un módulo que representa el programa principal del sistema.

# Capítulo 7

## Guía de Módulos

Esta guía describe la organización lógica de los módulos del sistema, qué implementa cada módulo concreto y qué oculta.

El criterio para dividir los módulos concretos y agruparlos en módulos lógicos se basó en la necesidad, que tendría un programador, de conocer los módulos que constituyen cierto comportamiento del sistema. De este modo, los módulos que están relacionados por contribuir a cierta funcionalidad, están agrupados.

Por otra parte, se hace hincapié en la herencia de interfaces y no en la herencia de implementación, a fin de mantener el principio de ocultamiento de información y abstracción. En consecuencia, en ciertos casos, algunos métodos declarados en una interfaz, tienen exactamente la misma implementación en todos los módulos herederos. Esto es una decisión de diseño, que prioriza la herencia de interfaces a la reutilización de código. Generalmente, dichas implementaciones son pequeñas. Sin embargo, hay algunos casos en los que módulos hijos, sí heredan implementación; pero generalmente son la implementación de un método plantilla como consecuencia de aplicar el patrón *Método Plantilla*.

La explicación de los métodos, en ocasiones, incluye una descripción de la funcionalidad en pseudocódigo. Dicha descripción es orientativa a fin de alcanzar una mejor comprensión de lo que debe hacer el método. Las descripciones en pseudocódigo, aún las que no se están presentes en esta guía, pueden también encontrarse en los diagramas o figuras del Capítulo 5; a los cuales puede accederse mediante el enlace [F](#) de cada módulo.

### MCU

---

*Función*

Módulo lógico que integra los subsistemas que constituyen el software del Microcontrolador del robot desmalezador.

### 7.1. EXTERNALCOMMUNICATION

---

*Función*

Módulo lógico que agrupa la porción del sistema que permitirá la comunicación entre el MCU con el exterior; en particular con la PC y el CR.

#### 7.1.1. CONNECTORS

---

*Función*

Módulo lógico que agrupa los conectores por medio de los cuales el MCU se comunicará con la PC y el CR.

##### 7.1.1.1. ConnectMCUtoPC

[MI](#) [DP](#) [F](#)

*Función*

Este es un módulo físico-concreto. Implementa un conector que permite una comunicación serie entre el MCU y la PC, en el cual el primero es el transmisor y el segundo es el receptor. Cuenta con un único método write que recibe el String a transmitir.

*Secreto*

Oculta la comunicación serie entre el MCU y la PC, llevada a cabo mediante un dispositivo UART integrado en el MCU.

### 7.1.1.2. BUFFERToMCU

#### Función

Módulo lógico que agrupa los módulos que permiten la comunicación entre el CR y el MCU.

#### 7.1.1.2.1 Pin

MI DP F

#### Función

Este es un módulo físico-concreto. Implementa un pin receptor de las señales provenientes del CR. El MCU contará con dos pines, uno que recibirá señales relativas a las órdenes de velocidad del CR y otro que recibirá señales relativas a la orden de giro. Las señales recibidas en el pin serán interrupciones físicas en el sistema que este módulo deberá manejar. El módulo define dos métodos en tu interfaz.

`signalHandler` será invocado ante cada interrupción física recibida en el pin. Este método solo ejecutará el comando correspondiente previamente establecido, en particular `CountTime::execute()`.

`setCommand`, recibirá como argumento un comando `Command`. Dicho comando será ejecutado ante una interrupción física recibida en el pin. Este método será invocado una única vez, con el comando `CountTime`, al momento de construir el objeto y determinar su comando.

#### Secreto

Oculta cómo es el manejo de una interrupción proveniente de un pin del CR.

#### 7.1.1.2.2 CountTime

MI DP F

#### Función

Este es un módulo físico-concreto el cual implementa un comando que actúa sobre un recolector de tiempo `CRpinCollector`. Cada vez que este comando es invocado, registra en el recolector mencionado el instante de tiempo en el que tuvo lugar dicha invocación. Los métodos son los siguientes.

`CountTime`, siendo el constructor, recibirá el recolector `CRpinCollector` sobre el cual el comando tendrá efecto.

`execute` registrará en el recolector el tiempo actual del sistema, invocando `CRpinCollector::currentTime()`.

#### Secreto

Oculta sobre qué objeto tiene efecto su ejecución. En particular, oculta cómo registra el instante de tiempo en el que el sistema recibió una interrupción física en un pin del CR.

#### 7.1.1.2.3 CRpinCollector

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa un recolector de tiempo, el cual es utilizado por el comando `CountTime` al ser ejecutado ante cada interrupción física proveniente de un pin del CR. Este recolector registrará los instantes de las dos últimas interrupciones físicas provenientes de un pin del CR y calculará el período de tiempo entre estas.

`CRpinCollector`, siendo el constructor, recibirá un recolector básico `TimeCollector` en el que delegará la operación de obtener el instante de tiempo actual del sistema.

`currentTime` guarda internamente los dos últimos instantes de tiempo en los que ocurrieron interrupciones provenientes del pin del CR. Esto lo hace obteniendo y guardando el último instante registrado, a través de `TimeCollector::getCurrentTime()`, y luego registrando el instante de tiempo actual, invocando `TimeCollector::currentTime()`.

`getCurrentTime` devuelve el último **instante** de tiempo registrado, invocando `TimeCollector::getCurrentTime()`.

`periodTime` devuelve el último **período** registrado. Esto lo hace calculando la diferencia entre los dos instantes de tiempo registrados por `currentTime`.

`initPeriod` establece como penúltimo instante de tiempo, al último registrado (`_preT= TimeCollector::getCurrentTime()`). El recolector definido por este módulo es utilizado por un buffer del CR `CRBuffer`. Cuando el sistema obtiene el valor registrado en dicho buffer, y este es vaciado, este método debe ser invocado para establecer cuál es el inicio del período de tiempo que se registrará ante una señal del CR.

#### Secreto

Oculta cómo registra y calcula el período de tiempo determinado por las dos últimas interrupciones físicas provocadas por las señales recibidas en un pin del CR.

#### 7.1.1.2.4 CRBuffer

MI DP1 DP2 F

##### Función

Este es un módulo físico-abstracto el cual define la interfaz de un buffer de las señales recibidas en un pin receptor del CR. Ver requerimientos RF-22, y RF-27. Los métodos declarados son los siguientes.

getVal retorna el valor registrado en el buffer, resultante de cálculos llevados a cabo sobre los tiempos de las interrupciones recibidas un el pin del CR.

empty determina si el buffer está vacío.

calcOrientation establece cuál es el sentido de avance (adelante/atrás) dado en la orden recibida.

Este módulo solo implementa el método plantilla getVal, cuya funcionalidad es descripta como sigue.

```
coll:CRpinCollector
data:CRData
fmap:MapFunction

getVal(){
  _period=coll.peridTime()
  coll.initPeriod()
  _calc=_period - data.getCalibration()
  data.setArg(_calc)
  calcOrientation()
  fmap.execute(data)
  return data.getResult()
}
```

##### Secreto

Oculto los distintos tipos de buffer utilizados para recibir los distintos tipos de órdenes provenientes del CR; y el método plantilla por el cual se obtiene el valor guardado en el buffer.

#### 7.1.1.2.5 VelBuffer

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa el buffer que registrará señales del pin de velocidad del CR. Los métodos que implementa son los siguientes.

VelBuffer, siendo el constructor, recibe como argumento un recolector CRpinCollector el cual mantiene internamente. Además, inicializa ciertos valores constantes que utiliza (ver req. RF-18, y RF-20.) y crea su estructura de datos CRData (que mantendrá internamente), la función de mapeo MapFunction y la función de cálculo de orientación OrientationCalc.

```
VelBuffer(CRpinCollector pinColl){
  fmap=new MapFunction()
  forient=new OrientationCalc()
  VCALIBRACION=49
  const=Constants::instance()
  MAXRPM_CR=const.getMaxRPMCR()
  ANCHOPULSO_CR=const.getPWM()
  coll=pinColl
  data=new CRData(VCALIBRACION, new MapData(0,ANCHOPULSO_CR,0,MAXRPM_CR, new SimpleData()))
}
```

(Ver req. RF-22.4.)

empty determina si el buffer está vacío retornando un valor booleano; para esto evalúa si el último período de tiempo registrado en el recolector es igual a 0, y en tal caso devuelve TRUE. Esto lo hace obteniendo dicho período mediante CRpinCollector::periodTime.

calcOrientation calcula el sentido de orientación (adelante/atrás) ejecutando la función de cálculo de orientación OrientationCalc::calculate y pasándole a esta sus datos CRData como argumento.

getOrientation obtiene y retorna el sentido de avance (adelante/atrás) resultante, del cálculo llevado a cabo por medio del método calcOrientation. Para obtener dicho valor invoca CRData::getOrientation.

##### Secreto

Oculto cómo son interpretadas, en términos de órdenes de velocidad, las señales físicas provenientes del CR y recibidas en el pin de velocidad.

#### 7.1.1.2.6 DirBuffer

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa el buffer que registrará señales del pin de dirección del CR. Los métodos que implementa son los siguientes.

DirBuffer, siendo el constructor, recibe como argumento un recolector CRpinCollector el cual mantiene internamente. Además, inicializa ciertos valores constantes que utiliza (ver req. RF-26.) y crea su estructura de datos CRData (que mantendrá internamente) y la función de mapeo MapFunction.

```
DirBuffer(CRpinCollector pinColl){
    fmap=new MapFunction()
    DCALIBRACION=49
    coll=pinColl
    const=Constants::instance()
    ANCHOPULSO_CR=const.getPWM()
    data=new CRData(DCALIBRACION, new MapData(-ANCHOPULSO_CR, ANCHOPULSO_CR, -100,100, new SimpleData()))
}
```

(Ver req. RF-27.2.)

empty determina si el buffer está vacío retornando un valor booleano; para esto evalúa si el último período de tiempo registrado en el recolector es igual a 0, y en tal caso devuelve TRUE. Esto lo hace obteniendo dicho período mediante CRpinCollector::periodTime.

calcOrientation este método no hace nada. Su implementación es vacía.

##### Secreto

Oculta cómo son interpretadas, en términos de órdenes de dirección, las señales físicas provenientes del CR y recibidas en el pin de dirección.

#### 7.1.1.3. PCtoMCU

##### Función

Este es un módulo lógico que agrupa los módulos que implementan una comunicación serie, mediante un dispositivo UART, entre la PC y el MCU; donde el primero es el transmisor y el segundo es el receptor.

#### 7.1.1.3.1 ConnectPCtoMCU

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa la comunicación serie desde la PC hacia el MCU.

Este módulo está constituido por un estado de tipo ConnectionState y delegará en éste la lectura del mensaje y el procesamiento de cada carácter que llegue desde la PC. Dependiendo del estado de la conexión, la lectura de la llegada de un carácter o del mensaje será diferente. El comportamiento en los distintos estados de conexión está especificado mediante un Statecharts, en la Sección 2.3.

Internamente este módulo va guardando los caracteres que va recibiendo hasta completar el mensaje. Una vez completado el mensaje, lo mantiene hasta que el mismo es leído, o bien, hasta que se inicia la recepción de un nuevo mensaje.

charRCVhandler es un manejador de la interrupción provocada por la llegada de un carácter. Cuando un carácter llega, el módulo delega en su estado el procesamiento del carácter mediante la llamada a ConnectionState::processChar(this, c) pasándose a sí mismo como argumento y pasando también como argumento el char recibido.

read permite leer el mensaje recibido en la conexión. Para esto el módulo delega en su estado la lectura, pasándose a sí mismo como argumento, mediante la llamada a ConnectionState::read(this). Una vez obtenido el String leído, lo retorna como resultado.

changeState recibe un estado de conexión ConnectionState y lo registra como el estado actual. Este método permite que otro módulo cambie el estado interno de la conexión. En particular, serán los diferentes estados herederos de ConnectionState los responsables por llevar a cabo esta tarea.

addChar recibe un carácter y lo guarda internamente en un buffer que mantendrá la cadena de caracteres que se irá formando con cada recepción.

getBuffer devuelve la cadena de caracteres guardadas en el módulo.

emptyBuffer vacía el buffer interno de caracteres, descartándolos.

##### Secreto

Oculta el modo en el cual va recibiendo los caracteres provenientes de la PC y cómo los va procesando.

#### 7.1.1.3.2 ConnectionState

MI DP F

##### Función

Este es un módulo físico-abstracto que provee la interfaz de los distintos estado en los que puede estar una conexión **ConnectPCtoMCU** desde la PC y hacia el MCU.

Provee los siguientes métodos.

processChar permite que la conexión procese un carácter recibido; para esto recibe la conexión y el carácter a procesar.

read permite leer el mensaje recibido en la conexión; para esto recibe la conexión como argumento.

Ver especificación en Sección 2.3.

##### Secreto

Oculta cómo se llevará a cabo el procesamiento de la recepción de un carácter y la lectura de un mensaje, dependiendo del estado de la conexión.

#### 7.1.1.3.3 NoMessage

MI DP F

##### Función

Este es un módulo físico-concreto, implementa los métodos correspondientes para procesar un carácter o leer un mensaje, cuando en la conexión no hay mensajes.

NoMessage, siendo el constructor de un estado en el que no hay mensajes, recibe el siguiente estado **Transmitting** al que deberá pasar la conexión ante ciertas circunstancias.

processChar procesa el carácter recibido. Recibe el carácter a procesar y la conexión **ConnectPCtoMCU** de la cual este módulo es estado. Si el carácter no es el de fin de mensaje ('\n'), guarda en la conexión dicho carácter mediante ConnectPCtoMCU::addChar y cambia el estado de la conexión a **Transmitting**, mediante ConnectPCtoMCU::changeState. Si por el contrario, el carácter recibido es el de fin de mensaje, el método no hace nada.

read(i ConnectPCtoMCU) devuelve un String vacío.

Ver especificación en Sección 2.3.

##### Secreto

Oculta cómo procesa la llegada de un carácter y la lectura de la conexión cuando no hay mensajes. También oculta cómo cambia el estado de la conexión.

#### 7.1.1.3.4 Transmitting

MI DP F

##### Función

Este es un módulo físico-concreto, implementa los métodos correspondientes para procesar un carácter o leer un mensaje cuando la conexión está transmitiendo caracteres.

processChar procesa el carácter recibido. Recibe el carácter a procesar y la conexión **ConnectPCtoMCU** de la cual este módulo es estado. Para procesar el carácter, primero lo agrega a la secuencia de caracteres recibidos (buffer interno). Luego evalúa si el carácter agregado es el de fin de mensaje ('\n'), en tal caso cambia el estado de la conexión a **ReadyMessage** mediante ConnectPCtoMCU::changeState; en caso contrario no hace nada más.

read recibe la conexión como argumento y devuelve un String vacío.

setNextState recibe como argumento el siguiente estado que tendrá la conexión ante ciertas circunstancias. Este método será invocado solo una vez, luego de construir el objeto. En dicho momento, se determinará como siguiente estado, el estado **ReadyMessage**.

Ver especificación en Sección 2.3.

##### Secreto

Oculta cómo procesa la llegada de un carácter y la lectura de la conexión cuando se están recibiendo caracteres. También oculta cómo cambia el estado de la conexión.

#### 7.1.1.3.5 ReadyMessage

MI DP F

##### Función

Este es un módulo físico-concreto, implementa los métodos correspondientes para procesar un carácter o leer un mensaje cuando en la conexión hay un mensaje listo para ser leído.

ReadyMessage, siendo el constructor de un estado de la conexión, recibe los dos siguientes estados posible a este. Los estados



son **NoMessage** y **Transmitting**.

processChar procesa el carácter recibido. Recibe el carácter a procesar y la conexión **ConnectPCtoMCU** de la cual este módulo es estado. Primero evalúa si el carácter es el mismo que el carácter de fin de mensaje ('\n'); en tal caso, no hace nada y termina. En caso contrario, vacía el buffer que guarda el mensaje ya recibido, mediante ConnectPCtoMCU::emptyBuffer, y agrega el nuevo carácter mediante ConnectPCtoMCU::addChar. Luego, cambia el estado de la conexión a **Transmitting** mediante ConnectPCtoMCU::changeState.

read recibe la conexión **ConnectPCtoMCU** como argumento. Primero lee el mensaje del buffer de la conexión y luego lo vacía; esto lo hace llamando a ConnectPCtoMCU::getBuffer y ConnectPCtoMCU::emptyBuffer. Luego cambia el estado de la conexión a **NoMessage** llamando a ConnectPCtoMCU::changeState. Finalmente, retorna el mensaje leído. Ver especificación en Sección 2.3.

#### Secreto

Oculta cómo procesa la llegada de un carácter y la lectura de la conexión cuando un mensaje está listo para su lectura. También oculta cómo cambia el estado de la conexión.

## 7.1.2. READERSWRITERS

### Función

Este es un módulo lógico que agrupa los módulos responsables de leer o escribir desde y hacia diferentes orígenes o destinos de información.

#### 7.1.2.1. Reader



#### Función

Este es un módulo físico-abstracto el cual provee la interfaz para poder leer información desde cierto origen. En este caso, la información será una orden proveniente del CR o de la PC.

readSetpointDD, readSetpointDI, readSetpointTD y readSetpointTI permiten leer los *setpoints* provenientes indistintamente de la PC o del CR, para la rueda delantera-derecha, la delantera-izquierda, la trasera-derecha y la trasera-izquierda, respectivamente.

readSetpointOrient permite leer a partir de una orden, cuál es la orientación del movimiento (adelante/atrás) indicado.

readAlgorithmDD, readAlgorithmDI, readAlgorithmTD y readAlgorithmTI permiten establecer a partir de la orden leída, los algoritmos correspondientes para la rueda delantera-derecha, la delantera-izquierda, la trasera-derecha y la trasera-izquierda, respectivamente.

readSetpointDir permite leer la orden correspondiente al sistema de dirección.

readMode recibe como argumento el modo de operación **Mode** (PC/CR). Dependiendo de la orden recibida, el modo recibido será modificado a modo PC o CR.

newMessage devuelve un valor booleano que establece si un nuevo mensaje ha sido recibido.

readStop devuelve un valor booleano que establece si una orden de frenado ha sido recibida.

isRPM devuelve un valor booleano indicando si las velocidades de referencia establecidas en la orden, de todas las ruedas, han sido dadas en RPM.

changeOrientation devuelve un valor booleano determinando si la orden recibida requiere un cambio en la orientación (adelante/atrás) del movimiento que se está llevando a cabo.

setModes recibe el conjunto de modos de operación **ModePool** (PC/CR) para guardarlos internamente.

readRoot recibe un objeto **Serializable** y hace una lectura inicial de este.

#### Secreto

Oculta el modo en el cual cierta información puede ser accedida desde distintos orígenes de datos.

#### 7.1.2.2. SerialReader



#### Función

Este es un módulo físico-concreto que permite obtener la información de una conexión **ConnectPCtoMCU**; esto es, de los mensajes provenientes de la PC. Este módulo sabe cómo interpretar el String que constituye el mensaje recibido. Dicha interpretación resultará en los objetos correspondientes que describen la orden recibida desde la PC.

SerialReader, siendo el constructor, recibe la conexión **ConnectPCtoMCU** desde donde leer la orden, y los tres algoritmos posibles sobre el funcionamiento de una rueda. Dependiendo del tipo de valor de referencia dado en una orden para una rueda (tensión, velocidad o corriente), se deberá utilizar uno de los siguientes tres algoritmos: **TensAlgorithm**, **VelAlgorithm** y **CntAlgorithm**. Este módulo es responsable por interpretar el mensaje y determinar qué algoritmo se debe utilizar para cada rueda.

readSetpointDD, readSetpointDI, readSetpointTD y readSetpointTI devuelven un valor en una unidad de medida **Measure** determinada. Estos métodos interpretan la orden recibida y devuelven el valor de referencia, en la unidad de medida correspondiente, para la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente.

readAlgorithmDD, readAlgorithmDI, readAlgorithmTD y readAlgorithmTI devuelven el algoritmo **Algorithm** que le corresponde a la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente. La decisión de qué algoritmo corresponde, dependerá de la información recibida en la orden sobre los valores de referencia. Para cualquier rueda; si el valor de referencia recibido es la tensión, el método retornará el algoritmo **TensAlgorithm**; si el valor de referencia es la velocidad, el método retornará el algoritmo **VelAlgorithm** y si el valor de referencia es la corriente, el método retornará el algoritmo **CntAlgorithm**.

readSetpointDir lee de la orden el valor de referencia del dispositivo de dirección y devuelve el valor correspondiente **SignedPerc**.

readMode recibe el modo **Mode** en el que está funcionando el sistema. Lee en la orden recibida si hay un cambio de modo y en tal caso cual sería (**PC/CR**); y asigna al modo recibido el modo correspondiente. En caso de que el modo no pueda ser leído, el modo recibido no es modificado.

```
mdPC: PC
mdCR: CR

readMode(Mode md){
    if modoPC then
        md=mdPC
    else if modoCR
        md=mdCR
}
```

newMessage devuelve un valor booleano indicando si hay o no un nuevo mensaje. Esto lo hace leyendo el mensaje desde la conexión mediante ConnectPCtoMCU::read. Si el valor de retorno de este último es una cadena vacía o no cumple con el formato descrito en **RF-29**, newMessage retornará **FALSE**, en caso contrario devolverá **TRUE**.

readStop devuelve un valor booleano, dependiendo si en el mensaje recibido hay una orden de frenado.

isRPM devuelve un valor booleano, dependiendo si en la orden recibida los valores de referencia de las cuatro ruedas, están dados en RPM. En el caso en que el valor de las cuatro ruedas esté en RPM, devolverá **TRUE**; en caso contrario **FALSE**.

changeOrientation devuelve un valor booleano dependiendo de si la orden recibida implica un cambio en el sentido (adelante/atrás) de avance de las ruedas. Para esto obtiene de los datos del controlador principal la orientación actual del sistema, esto lo hace mediante MainCtrlData::getOrientation. Del resultado **MovementSense** obtenido, accede al valor numérico mediante MovementSense::Value y devuelve la comparación de igualdad entre este y el valor numérico obtenido de la orden. Cabe mencionar, que este módulo accederá a los datos del controlador principal cuando su método readRoot sea invocado.

setModes este método recibe el conjunto **ModePool** de modos de operación posibles del sistema, y los guarda internamente. Este método será solo utilizado una vez que este módulo haya sido creado y luego no volverá a ser necesario. El objetivo es que este módulo cuente con los modos de operación posible para poder utilizarlos al momento de decidir en qué modo debe operarse; decisión que toma mediante el método readMode.

readRoot recibe un elemento **Serializable** para iniciar la lectura. En particular recibirá al controlador principal **MainController** y obtendrá de este sus datos, mediante MainController::getData.

*Secreto*

Oculto que lee la información de una conexión **ConnectPCtoMCU** y cómo lo hace.

### 7.1.2.3. BufferReader

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que permite obtener la información de una conexión con el CR; esto es, obtener las órdenes recibidas en los pines **Pin** de velocidad y de dirección.

BufferReader, siendo el constructor, recibirá y mantendrá los buffers **VelBuffer** (de velocidad) y **DirBuffer** (de dirección); desde los cuales leerá las órdenes recibidas. Además, el constructor recibirá el algoritmo **VelAlgorithm** que será utilizado para los cálculos de control de las ruedas; y obtendrá el valor máximo de RPM asociado al CR, por medio de Constants::getMAXRPMCR, el cual será utilizado por el módulo para ciertos cálculos.

readSetpointDD, readSetpointDI, readSetpointTD y readSetpointTI devuelven los valores **RPM** de referencia de la rueda

delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda; respectivamente. Para esto, en todos los casos, el método obtiene el valor desde el buffer de velocidad por medio de `VelBuffer::getVal`. Luego lo establece como una velocidad RPM mediante `RPM::setVal` y retorna la mencionada velocidad.

`readSetpointOrient` devuelve la orientación `MovementSense` del movimiento (atrás/adelante) indicada en la orden recibida. Para esto, lee del buffer de velocidad la orientación registrada mediante `VelBuffer::getOrientation`, dicho valor es establecido como una orientación en el movimiento a través de `MovementSense::setVal`, y entonces, dicha orientación es retornada.

`readAlgorithmDD`, `readAlgorithmDI`, `readAlgorithmTD` y `readAlgorithmTI` devuelven el algoritmo `Algorithm` que le corresponde a la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda; respectivamente. En este caso, como los valores de referencia provenientes del CR son únicamente en RPM; en todos los casos, los métodos retornarán un algoritmo `VelAlgorithm`.

`readSetpointDir` devuelve el valor de referencia `SignedPerc` del dispositivo de dirección, obtenido de la orden proveniente del CR. Para esto, obtiene del buffer de dirección el valor, mediante `DirBuffer::getVal`. Luego establece dicho valor como una posición del dispositivo de dirección, mediante `SignedPerc::setVal`, y la retorna.

`readMode` recibe un `Mode` pero su implementación es vacía. El método no hace nada. Esto se debe a que la orden de cambio de modo de operación, solo puede provenir de la PC y no del CR.

`newMessage` devuelve un valor booleano indicando si se ha recibido un mensaje. Para esto, devuelve la conjunción negada de los resultados obtenidos a partir de `VelBuffer::empty` y `DirBuffer::empty`

`readRoot` recibe un elemento `Serializable` para iniciar la lectura. En particular recibirá al controlador principal `MainController` y obtendrá de este sus datos, mediante `MainController::getData`.

*Secreto*

Oculto que lee la información de los buffers de conexión con el CR y cómo lo hace.

#### 7.1.2.4. Writer



*Función*

Este es un módulo físico-abstracto el cual provee la interfaz para poder escribir información en cierto destino. En este caso, la información será los valores medidos, el resultado de los cálculos llevados a cabo, el modo en el que el sistema está funcionando y si ha habido una orden de frenado.

`writeVelDD`, `writeVelDI`, `writeVelTD` y `writeVelTI` recibirán como argumento las velocidades medidas `Measure`, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y permitirán escribir dichas velocidades.

`writeCntDD`, `writeCntDI`, `writeCntTD` y `writeCntTI` recibirán como argumento las corrientes medidas `Measure`, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y permitirán escribir dichas corrientes.

`writePosDD`, `writePosDI`, `writePosTD` y `writePosTI` recibirán como argumento las posiciones medidas `Measure`, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y permitirán escribir dichas posiciones.

`writeTensDD`, `writeTensDI`, `writeTensTD` y `writeTensTI` recibirán como argumento las tensiones medidas `Measure`, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y permitirán escribir dichas tensiones.

`writeDPosition` recibirá la posición medida del dispositivo de dirección `Measure` y permitirá escribirla.

`writeOrientation` recibirá la orientación `Measure` de desplazamiento (adelante/atrás) de las ruedas y permitirá escribirla.

`writeStopOrder` recibirá un valor de verdad `Bool` indicando si se ha recibido una orden de frenado, o no, y permitirá escribir dicha información.

`writeModeId` recibirá un identificador `Int` indicando el modo de operación del sistema y permitirá escribir dicha información.

`writeAll` permitirá escribir toda la información junta.

*Secreto*

Oculto el modo en el que la información es escrita en ciertos destinos de datos.

#### 7.1.2.5. SerialWriter



*Función*

Este es un módulo físico-concreto que permite escribir información, en el formato adecuado, en una conexión serial con la PC.

Implementa todos los métodos que hereda de su padre.

Los métodos de este módulo que reciben como argumento una medida **Measure**, un valor de verdad **Bool** o un identificador **Int**, transformarán los mencionados argumentos en los **String** correspondientes y los guardarán internamente. Dichos **String** deberán mantener el adecuado formato según el requerimiento **RF-77**. El método writeAll será entonces, el responsable de organizar todos esos **String** de modo tal de construir un mensaje con el formato descrito en el antedicho requerimiento; y luego, escribirlo en la conexión mediante ConnectMCUtoPC::write.

SerialWriter, siendo el constructor, recibe como argumento el conector **ConnectMCUtoPC** que permitirá la conexión con la PC.

writeVelDD, writeVelDI, writeVelTD y writeVelTI recibirán como argumento las velocidades medidas **Measure**, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y transformarán los valores de dichas velocidades a las cadenas de caracteres numéricos correspondientes. Los resultados de las mencionadas transformaciones será guardados internamente.

writeCntDD, writeCntDI, writeCntTD y writeCntTI recibirán como argumento las corrientes medidas **Measure**, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y transformarán los valores de dichas corrientes a las cadena de caracteres numéricos correspondientes. . Los resultados de las mencionadas transformaciones será guardados internamente.

writePosDD, writePosDI, writePosTD y writePosTI recibirán como argumento las posiciones medidas **Measure**, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y transformarán los valores de dichas posiciones a las cadenas de caracteres numéricos correspondientes. Los resultados de las mencionadas transformaciones será guardados internamente.

writeTensDD, writeTensDI, writeTensTD y writeTensTI recibirán como argumento las tensiones medidas **Measure**, de la rueda delantera-derecha, delantera-izquierda, trasera-derecha y trasera-izquierda, respectivamente; y transformarán los valores de dichas tensiones a las cadenas de caracteres numéricos correspondientes. Los resultados de las mencionadas transformaciones será guardados internamente.

writeDPosition recibirá la posición medida del dispositivo de dirección **Measure** y transformará el valor de dicha posición a la cadena de caracteres numéricos correspondientes. El resultado de la mencionada transformación será guardado internamente.

writeOrientation recibirá la orientación **Measure** de desplazamiento de las ruedas (adelante/atrás) y transformará el valor de dicha orientación a la cadena de caracteres numéricos correspondientes. El resultado de la mencionada transformación será guardado internamente.

writeStopOrder recibirá un valor de verdad **Bool** indicando si se ha recibido una orden de frenado, o no; y transformará dicha información en la cadena de caracteres "1" en caso afirmativo y en la cadena "0" en caso negativo. El resultado de dicha transformación será guardado internamente.

writeModeId recibirá un identificador **Int** indicando el modo de operación del sistema; y transformarán dicho identificador a la cadena de caracteres numéricos correspondientes. El resultado de la mencionada transformación será guardado internamente.

writeAll construirá una cadena de caracteres a partir de las cadenas de caracteres guardadas internamente por los otros métodos. La cadena resultante respetará el formato establecido en los requerimientos. Luego, invocará ConnectMCUtoPC::write pasándole como argumento la mencionada cadena de caracteres.

### *Secreto*

Oculto que escribe en una conexión **ConnectMCUtoPC** y el modo en que lo hace.

## 7.2. PRINCIPALCONTROLLER

### *Función*

Este es un módulo lógico que agrupa la porción del sistema responsable de sincronizar y llevar a cabo el funcionamiento de los subsistemas de control de ruedas y dirección.

### 7.2.1. Serializable



### *Función*

Este es un módulo físico-abstracto que provee la interfaz para recibir un módulo lector como medio para poder leer de un origen de datos, y un módulo escribiente como medio para escribir en un destino de datos.

Los métodos que provee son readFrom y writeTo, los cuales no implementa. Serán los módulos herederos de este, los que

implementarán los mencionados métodos y harán uso de los objetos de lectura o escritura recibidos.

#### Secreto

Oculta cómo un módulo lee o escribe desde o en, un repositorio de información.

### 7.2.2. MainController

MI DP1 DP2 DP3 DP4 F

#### Función

Este es un módulo físico-concreto que implementa el control principal del sistema, recibiendo las órdenes provenientes del CR o la PC, procesándolas y re-direccionándolas a los subsistemas que lo componen.

El módulo está compuesto por un modo de operación **Mode**, un estado de operación **OperationState**, un grupo de sistemas de control **ControlSystemPool** y cierta información agrupada en un elemento **MainCtrlData**.

MainController, siendo el constructor, recibirá el conjunto de sistemas de control **ControlSystemPool**, el que mantendrá, y creará un elemento **MainCtrlData**.

readFrom recibe un lector **Reader**. En particular, recibirá un lector **SerialReader** que es el que determinará si hay una orden de cambio de modo de operación. readFrom será responsable de indicarle al lector recibido, que interprete y determine el modo de operación que está indicado en una orden. Para esto, este método solo invocará SerialReader::readMode pasándole como argumento la variable en la que se guarda el modo de operación actual del controlador principal. Será entonces el lector que decidirá si modificar el modo o no, dependiendo de la orden leída.

writeTo recibe un escribiente **Writer**, en particular un elemento **SerialWriter**. Este método es responsable de enviar a la PC la información correspondiente. Para esto, delega en el estado de operación **OperationState** la tarea; solo invoca OperationState::write, pasándose a sí mismo como argumento y pasando como argumento el escribiente recibido.

changeState recibe un estado de operación **OperationState** y lo guarda como el estado actual.

updateOrientation actualiza en la información interna del módulo, la orientación del desplazamiento (adelante/atrás) que se está llevando a cabo. Para esto, obtiene la orientación que fue requerida, mediante MainCtrlData::getNewOrientation, y la guarda como la orientación actual, mediante MainCtrlData::saveOrientation.

changeMode recibe un modo de operación **Mode** y lo guarda como el modo actual.

getMode devuelve el modo **Mode** en el que está operando el sistema.

getData devuelve los datos **MainCtrlData** del controlador.

getCtrlSysPool devuelve el conjunto de sistemas de control **ControlSystemPool**.

control lleva adelante el control de los subsistemas de control. Para esto, delega en el estado de operación **OperationState** la tarea. Este método, solo invoca OperationState::control pasándose a sí mismo como argumento.

read lee las órdenes de control que deben ser llevadas a cabo. Para esto, delega la tarea en el estado de operación **OperationState**, invocando OperationState::read pasándose a sí mismo como argumento. Notar que la diferencia entre readFrom y este método, reside fundamentalmente en que el primero solo leerá la información relativa al cambio de modo de operación (que solo proviene de la PC), mientras que el segundo leerá la información correspondiente a los valores de referencia deseados.

#### Secreto

Oculta cómo y de dónde lee las órdenes para los subsistemas que lo constituyen, y cómo las lleva a cabo. Además oculta hacia dónde y cómo envía la información que recolecta de los correspondientes controles.

### 7.2.3. ControlSystemPool

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa un conjunto de sistemas de control que serán manejados por el controlador principal **MainController**. Su interfaz provee los métodos para acceder a sus elementos, el método para leer la información que sus elementos necesitan y el método que permite enviar a la PC la información relativa a los elementos que lo componen.

ControlSystemPool, siendo su constructor, recibe y mantiene los cuatro sistemas de control **WheelSystem**, correspondientes a las cuatro ruedas, y el sistema de control de dirección **DirSystem**. Además, recibe los estados de operación **DActive** y **DStopping** del sistema de control de dirección, y los guarda. Por otra parte, este constructor creará y mantendrá los algoritmos **Stop**, **Advance**, **Reverse** y **ReverseRPM**; y establecerá como algoritmo de control inicial para las ruedas el algoritmo **Advance**.

```

ControlSystemPool(WheelSystem wsDD,...){
    stop=new Stop()
    advance=new Advance()
    reverse=new Reverse(advance)
    reverseRPM=new ReverseRPM(stop,reverse)

    wCtrlDD=wsDD.getController()
    wCtrlDD.setCtrlAlgorithm(advance)
    ...
}

```

readFrom recibe un lector **Reader**. Este método es el responsable de solicitar al lector la información necesaria para llevar a cabo el control de los subsistemas. Primero, evalúa si hay una orden de frenado mediante Reader::readStop. En tal caso, establece **Stop** como algoritmo de control para las ruedas, mediante WheelController::setCtrlAlgorithm, y **DStopping** como estado de operación del sistema de dirección, mediante DirController::changeOpState. En caso contrario, establece **DActive** como estado de operación del sistema de dirección y evalúa si no hay orden de cambio de dirección, con Reader::changeOrientation. En tal caso, establece para todas las ruedas, el algoritmo **Advance** mediante WheelController::setCtrlAlgorithm. En caso contrario, se evalúa si los valores de referencia de todas las ruedas son en RPM. En ese caso, se establece **ReverseRPM** como algoritmo de control en todas las ruedas mediante WheelController::setCtrlAlgorithm. En otro caso, se establece **Reverse** como algoritmo de control de todas las ruedas, por medio de WheelController::setCtrlAlgorithm. Una vez decidido sobre el estado de operación del sistema de dirección y sobre los algoritmos de control de las ruedas; el método obtiene y establece los valores de referencia para cada rueda por medio de Reader::readSetpointDD, Reader::readSetpointDI, Reader::readSetpointTD, Reader::readSetpointTI y WheelController::setSetpoint. A continuación, lleva a cabo lo análogo pero respecto al algoritmo a aplicar a cada rueda, esto lo hace mediante Reader::readAlgorithmDD, Reader::readAlgorithmDI, Reader::readAlgorithmTD, Reader::readAlgorithmTI y WheelController::setAlgorithm. Luego, establece la orientación para cada rueda por medio de Reader::readSetpointOrient y WheelController::setOrientation. Finalmente, determina el valor de referencia para el sistema de dirección utilizando Reader::readSetpointDir y DirController::setSetpoint.

```

readFrom(Reader rdr){
    ...
    if rdr.readStop()
        wCtrlDD.setCtrlAlgorithm(stop)
        wCtrlDI.setCtrlAlgorithm(stop)
        ...
        dCtrl.changeOpState(dStoppingSt)
    else
        dctrl.changeOpState(dActiveSt)
        if (not rdr.changeOrientation())
            wCtrlDD.setCtrlAlgorithm(advance)
            ...
        else
            if (rdr.isRPM())
                wCtrlDD.setCtrlAlgorithm(reverseRPM)
                ...
            else
                wCtrlDD.setCtrlAlgorithm(reverse)
                ...

        wCtrlDD.setSetpoint(rdr.readSetpointDD())
        wCtrlDI.setSetpoint(rdr.readSetpointDI())
        ...
        wCtrlDD.setAlgorithm(rdr.readAlgorithmDD())
        ...
        wCtrlDD.setOrientation(rdr.readSetpointOrient())
        ...
        dCtrl.setSetpoint(rdr.readSetpointDir())
}

```

writeTo recibe un escribiente **Writer**. Este método es responsable de indicar la escritura de la información correspondiente que debe ser enviada a la PC. Este método sabe cuál es la información que se encuentra en los sistemas de control, que es de interés. Para llevar a cabo su objetivo, este método obtiene de cada sistema de control de ruedas **WheelSystem**, el controlador correspondiente, y a partir de este accede al conjunto de datos mediante WheelController::getData. De este modo, con los datos de cada rueda, realizará lo siguiente. Obtendrá la posición medida de una rueda invocando WheelCtrlData::getPosition y la escribirá mediante Writer::writePosDD, Writer::writePosDI, Writer::writePosTD o Writer::writePosTI, según corresponda. Obtendrá la velocidad medida de una rueda invocando WheelCtrlData::getVel y la escribirá mediante Writer::writeVelDD, Writer::writeVelDI, Writer::writeVelTD o Writer::writeVelTI, según corresponda. Obtendrá la corriente medida de una rueda invocando WheelCtrlData::getCnt y la escribirá mediante Writer::writeCntDD, Writer::writeCntDI, Writer::writeCntTD o Writer::writeCntTI, según corresponda. Obtendrá la tensión aplicada a una rueda invocando WheelCtrlData::getTens y la escribirá mediante Writer::writeTensDD, Writer::writeTensDI, Writer::writeTensTD o Writer::writeTensTI, según corres-

ponda. Luego, obtendrá los datos del controlador del sistema de dirección, invocando `DirController::getData`. Accederá entonces a la posición medida del dispositivo de dirección, por medio de `DirCtrlData::getPosition`, y la escribirá a través de `Writer::writeDPosition`.

`wheelSystemIt` devuelve un iterador sobre los sistemas de control de ruedas `WheelSystem`. Este iterador es como los iteradores que están definidos en estructuras de datos iterables, que habitualmente están predefinidas en ciertos lenguajes de programación; por ejemplo, en C++ o Java.

`getDirSystem` devuelve el sistema de control de dirección `DirSystem`.

#### *Secreto*

Este módulo oculta qué estructuras de datos utiliza para mantener los sistemas de control. En particular, requiere que los sistemas de control de ruedas puedan ser iterados y este iterador pueda ser obtenido a través de la interfaz. Además oculta qué información lee por medio del lector y qué información escribe a través del escribiente.

### 7.2.4. ControllerTimeout

MI DP F

#### *Función*

Este es un módulo físico-concreto que implementa un comando `Command` que será ejecutado a partir de cada pulso del reloj principal `FirstTimer`.

`ControllerTimeout`, siendo el constructor, recibe el controlador principal `MainController`, un lector `SerialReader` y un escribiente `SerialWriter`.

`execute` ejecuta el comando, llevando a cabo las siguientes acciones. Primero, le indica al controlador que debe leer del lector indicado, si hay una orden de cambio de modo de operación; esto lo hace mediante una llamada a `MainController::readFrom`, pasándole el lector `SerialReader` como argumento. Luego, le indica al controlador que debe leer las órdenes correspondientes, a los valores de referencia de las ruedas y del dispositivo de dirección; para esto invoca a `MainController::read`. Después, llama al método `MainController::control` para que se lleve a cabo el control de los subsistemas de control. Finalmente, le indica al controlador principal que debe enviar la correspondiente información mediante la llamada a `MainController::writeTo`, pasándole como argumento el escribiente `SerialWriter`.

#### *Secreto*

Oculta las acciones que llevará a cabo, ante un *tick* del reloj principal, y sobre qué elementos dichas acciones tienen efecto.

### 7.2.5. OPERATIONMODES

#### *Función*

Este es un módulo lógico que agrupa los modos de operación en los que puede operar el sistema. Dichos modos de operación están definidos de acuerdo al origen de las órdenes: si estas provienen del CR, o bien, de la PC.

#### 7.2.5.1. Mode

MI DP1 DP2 F

#### *Función*

Este es un módulo físico-abstracto que provee la interfaz para leer las órdenes a procesar; las cuales serán leídas desde distintos orígenes dependiendo del modo de funcionamiento en el que esté el sistema (modo CR o PC).

`newMessage` devuelve un valor booleano indicando si hay un mensaje (u orden) que pueda ser leído.

`read` recibe como argumento el controlador principal `MainController` y lee la orden a procesar.

`changeMode` recibe el controlador principal `MainController` para establecer en este, un cambio en el modo de recepción de órdenes.

#### *Secreto*

Oculta los distintos modos de operación en los que el sistema puede recibir órdenes.

#### 7.2.5.2. BasicMode

MI DP F

#### *Función*

Este es un módulo físico-concreto que implementa un modo de operación básico en el que las órdenes son leídas desde un lector `Reader`.

`BasicMode`, siendo el constructor, recibe un lector `Reader` que mantendrá internamente.



newMessage devuelve un valor booleano indicando si hay un mensaje (u orden) que pueda ser leído. Para esto, delega la tarea en el lector retornando el valor resultante de invocar a Reader::newMessage

read recibe como argumento el controlador principal **MainController** y lee la orden a procesar. Para esto, primero invoca Reader::readRoot pasándole como argumento el controlador. Después, obtiene la orientación (adelante/atrás) pedida y la orden relativa al frenado. Esto lo hace a través de Reader::readSetpointOrient y Reader::readStop, respectivamente. A continuación, obtiene los datos del controlador principal y registra la orientación pedida y la orden relativa al frenado, mediante MainController::getData, MainCtrlData::saveNewOrientation y MainCtrlData::saveStopOrder. Luego, obtiene el grupo de sistemas de control **ControlSystemPool** y les indica que deben leer cada uno su información. Esto lo hace, por medio de MainController::getCtrlSysPool y ControlSystemPool::readFrom; pasándole a este último el lector **Reader**.

```
read(MainController mCtrl){
    rdr.readRoot(mCtrl)
    orient=rdr.readSetpointOrient()
    stopBool=rdr.readStop()
    mData=mCtrl.getData()
    mData.saveNewOrientation(orient)
    mData.saveStopOrder(stopBool)
    csp=mCtrl.getCtrlSysPool()
    csp.readFrom(rdr)
}
```

*Secreto*

Oculta cómo lleva a cabo la lectura de órdenes a partir de un lector.

### 7.2.5.3. LectureMode

**MI** **DP** **F**

*Función*

Este es un módulo físico-abstracto que provee la interfaz de un modo de operación que extenderá la funcionalidad de un modo básico de operación **BasicMode**. Todos los módulos herederos de este, tendrán un elemento **Mode** que mantendrán internamente, dicho elemento puede ser declarado en este módulo.

newMessage devuelve un valor booleano indicando si hay un mensaje (u orden) que pueda ser leído.

read recibe como argumento el controlador principal **MainController** y lee la orden a procesar.

changeMode recibe el controlador principal **MainController** para establecer en este, un cambio en el modo de recepción de órdenes.

*Secreto*

Oculta que internamente mantiene un elemento **Mode**.

### 7.2.5.4. CR

**MI** **DP1** **DP2** **F**

*Función*

Este es un módulo físico-concreto que implementa el modo de operación en el que las órdenes son leídas desde el CR. Internamente, guardará el siguiente modo de operación, que deberá tener el sistema, si en el presente modo no es posible leer las órdenes por pérdida de conexión con el CR. En particular, el siguiente modo será **PC**. Recordar que ante una pérdida de conexión, el sistema deberá intentar leer alternadamente, desde la PC y desde el CR.

CR, siendo el constructor, recibe un modo de operación básico **BasicMode**. En particular, el modo de operación básico debe haber sido previamente construido con el lector **BufferReader**.

newMessage devuelve un valor booleano indicando si hay un mensaje (u orden) que pueda ser leído. Para esto, delega la tarea en el modo básico que contiene, invocando a BasicMode::newMessage y retornando el valor recibido.

read recibe como argumento el controlador principal **MainController** y lee la orden a procesar. Para esto, primero registra el identificador propio de este modo (CR), mediante MainController::getData y MainCtrlData::saveModeId, siendo 0 dicho identificador. Luego, para leer la orden, delega en el modo básico que contiene la mencionada tarea, invocando a BasicMode::read y pasándole como argumento el controlador.

```
read(MainController mCtrl){
    mData=mCtrl.getData()
    mData.saveModeId(0)
    md.read(mCtrl)
}
```



changeMode recibe el controlador principal **MainController** y le cambia, a este, el modo de operación al siguiente correspondiente. Para esto, invoca MainController::changeMode pasándole como argumento el siguiente modo que el módulo haya registrado a través de setNextMode. En particular, este será **PC**.

setNextMode recibe un **Mode** y lo guarda como el siguiente modo, al modo que implementa este módulo. Este método será utilizado por única vez, cuando sea construido este modo de operación; a fin de establecer cuál será el siguiente modo correspondiente a este.

#### *Secreto*

Oculto cómo lleva a cabo la lectura de órdenes cuando el sistema está en modo CR.

### 7.2.5.5. PC

**MI** **DP1** **DP2** **F**

#### *Función*

Este es un módulo físico-concreto que implementa el modo de operación en el que las órdenes son leídas desde la PC. Internamente, guardará el siguiente modo de operación, que deberá tener el sistema, si en el presente modo no es posible leer las órdenes por pérdida de conexión con la PC. En particular, el siguiente modo será **CR**. Recordar que ante una pérdida de conexión, el sistema deberá intentar leer alternadamente, desde la PC y desde el CR.

PC, siendo el constructor, recibe un modo de operación básico **BasicMode** y el modo de operación **CR**. En particular, el modo de operación básico debe haber sido previamente construido con el lector **SerialReader**.

newMessage devuelve un valor booleano indicando si hay un mensaje (u orden) que pueda ser leído. Para esto, delega la tarea en el modo básico que contiene, invocando a BasicMode::newMessage y retornando el valor recibido.

read recibe como argumento el controlador principal **MainController** y lee la orden a procesar. Para esto, primero registra el identificador propio de este modo (PC), mediante MainController::getData y MainCtrlData::saveModeId, siendo 1 dicho identificador. Luego, para leer la orden, delega en el modo básico que contiene la mencionada tarea, invocando a BasicMode::read y pasándole como argumento el controlador.

```
read(MainController mCtrl){
    mData=mCtrl.getData()
    mData.saveModeId(1)
    md.read(mCtrl)
}
```

changeMode recibe el controlador principal **MainController** y le cambia, a este, el modo de operación al siguiente correspondiente. Para esto, invoca MainController::changeMode pasándole como argumento el siguiente modo que el módulo haya registrado a través de su constructor. En particular, este será **CR**.

#### *Secreto*

Oculto cómo lleva a cabo la lectura de órdenes cuando el sistema está en modo PC.

### 7.2.5.6. ModePool

**MI** **DP** **F**

#### *Función*

Este es un módulo físico-concreto que agrupa los modos de operación del sistema y que provee los métodos para acceder a estos. Este módulo es utilizado por **MCBuilder** para la construcción del controlador principal **MainController**.

ModePool, siendo el constructor, recibe un modo **CR** y un modo **PC**.

getPCMode devuelve un modo **Mode**; en particular, el modo **PC**.

getCRMode devuelve un modo **Mode**; en particular, el modo **CR**.

#### *Secreto*

Oculto cómo mantiene internamente los modos de operación del sistema.

## 7.2.6. STATES

#### *Función*

Este es un módulo lógico que agrupa los módulos que implementan los posibles estados de operación del controlador principal.

### 7.2.6.1. OperationState

MI DP1 DP2 F

#### Función

Este es un módulo físico-abstracto el cual provee la interfaz de un estado de operación del controlador principal **MainController**. Dicha interfaz permitirá leer órdenes, controlar los sub-sistemas de control y enviar hacia el exterior del sistema la información correspondiente. Estas tareas tendrán una implementación diferente en cada estado concreto heredado de este módulo. El presente módulo solo implementa el *método plantilla* read.

read recibe el controlador principal **MainController** como argumento. A partir de este, obtiene el modo de operación en el que está operando el sistema, invocando MainController::getMode. Luego evalúa si hay una orden nueva a procesar, mediante Mode::newMessage. En tal caso, invoca al método actionWithMsg, pasándole como argumento el controlador principal y el modo de operación. En caso contrario, invoca el método actionNoMsg pasándole como argumento el controlador principal.

```
read(MainController mCtrl){
    md=mCtrl.getMode()
    if md.newMessage()
        actionWithMsg(mCtrl,md)
    else
        actionNoMsg(mCtrl)
}
```

actionWithMsg recibe el controlador principal **MainController** y el modo **Mode** en el que está operando el sistema; y establece las acciones a llevar a cabo cuando hay una orden que puede ser procesada.

actionNoMsg recibe el controlador principal **MainController** y establece las acciones a llevar a cabo cuando no hay una orden que pueda ser procesada.

control recibe el controlador principal **MainController** y lleva a cabo el control de los subsistemas.

write recibe el controlador principal **MainController** y el escribiente **Writer**; y envía la información medida y calculada hacia el exterior; esto es, hacia la PC.

Ver especificación en Sección 2.3.

#### Secreto

Oculta cómo cambia cierto funcionamiento dependiendo del estado en el que esté el controlador principal.

### 7.2.6.2. WaitingN

MI DP1 DP2 DP3 F

#### Función

Este es un módulo físico-concreto que implementa cierto comportamiento dependiente del estado; en particular, cuando el estado es de espera por una orden a recibir. El sistema esperará a lo sumo  $NMAX$  ciclos sin recibir un mensaje de la PC o del CR. Este módulo implementa el  $n$ -ésimo estado de espera.

WaitingN, siendo el constructor, recibe dos estados de operación que serán los siguientes estados posibles, al estado que este módulo representa. Uno de los estados siguientes será **Working** y el otro un estado **OperationState**, este último pudiendo ser otro estado **WaitingN** o bien el estado **WaitingMAX**. En el momento en el que se construyen los estados de operación **WaitingN**, los estados  $i$ -ésimos, siendo  $i \in [1, NMAX-2]$ , recibirán un estado **WaitingN** como siguiente. Sin embargo, cuando el estado de operación **WaitingN** sea el  $i$ -ésimo, con  $i = NMAX-1$ , el siguiente estado que recibirá el constructor, será **WaitingMAX**. La construcción de estos estados es llevada a cabo por el módulo **MCBuilder**.

actionWithMsg recibe como argumento el controlador principal **MainController** y el modo de operación **Mode** en el que está el sistema. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está en espera y se ha recibido una orden. Este método hace lo siguiente. Primero indica al modo de operación que lleve a cabo la lectura de la orden recibida, esto lo hace invocando Mode::read, pasándole como argumento el controlador principal. Luego, le cambia el estado de operación, al controlador principal, invocando MainController::changeState, pasándole como argumento el estado **Working**.

actionNoMsg recibe como argumento el controlador principal **MainController**. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está en espera y no se ha recibido una orden. Este método simplemente cambia el estado de operación del controlador principal invocando MainController::changeState, pasándole como argumento el siguiente estado de espera que haya recibido en el constructor (**WaitingN** o **WaitingMAX**).

control recibe un controlador principal **MainController** y lleva a cabo el control de los sub-sistemas. Su implementación coincide con aquella del método Working::control.

write recibe el controlador principal **MainController** y un escribiente **Writer**. Este método es responsable por iniciar la escritura de la información que debe ser enviada al exterior del sistema. Su implementación coincide con aquella del método Working::write.

Ver especificación en Sección 2.3.

*Secreto*

Oculta cuál es el/los siguiente/s estado/s y cómo se llevan a cabo el control y la escritura de datos en un estado de espera.

### 7.2.6.3. WaitingMAX

MI DP1 DP2 DP3 F

*Función*

Este es un módulo físico-concreto que implementa cierto comportamiento dependiente del estado; en particular, cuando es el último estado de espera por una orden a recibir.

WaitingMAX, siendo el constructor, recibe los posibles estados de operación siguientes, al estado que este módulo representa. Los posibles estados siguientes son **Working** y **Reconnecting**. Además este constructor recibe la orden **ControllersStop** y la orden **MainCtrlOrder**.

actionWithMsg recibe como argumento el controlador principal **MainController** y el modo de operación **Mode** en el que está el sistema. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está en el último estado de espera y se ha recibido una orden. Este método hace lo siguiente. Primero indica al modo de operación que lleve a cabo la lectura de la orden recibida, esto lo hace invocando Mode::read, pasándole como argumento el controlador principal. Luego, le cambia el estado de operación, al controlador principal, invocando MainController::changeState, pasándole como argumento el estado **Working**.

actionNoMsg recibe como argumento el controlador principal **MainController**. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está en el último estado de espera y no se ha recibido una orden. Este método hace lo siguiente. Primero obtiene el grupo de sistemas de control **ControlSystemPool**, invocando MainController::getCtrlSysPool. Después, le indica a todos estos sistemas que deben detenerse; para esto, llama a ControllersStop::execute, pasándole como argumento el grupo de sistemas. Finalmente, cambia el estado de operación del controlador principal, mediante MainController::changeState, pasándole como argumento el estado **Reconnecting**.

control recibe un controlador principal **MainController** y lleva a cabo el control de los sub-sistemas. Su implementación coincide con aquella del método Working::control.

write recibe el controlador principal **MainController** y un escribiente **Writer**. Este método es responsable por iniciar la escritura de la información que debe ser enviada al exterior del sistema. Su implementación coincide con aquella del método Working::write

Ver especificación en Sección 2.3.

*Secreto*

Oculta cuál es el/los siguiente/s estado/s y cómo se llevan a cabo el control y la escritura de datos en el último estado de espera.

### 7.2.6.4. Reconnecting

MI DP1 DP2 DP3 F

*Función*

Este es un módulo físico-concreto que implementa cierto comportamiento dependiente del estado; en particular, cuando el sistema está intentando una reconexión con la PC y el CR.

Reconnecting, siendo el constructor, recibe el estado de operación siguiente **Working**, al estado que este módulo representa; y la orden **MainCtrlOrder**.

actionWithMsg recibe como argumento el controlador principal **MainController** y el modo de operación **Mode** en el que está el sistema. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema recibe una orden, estando en el estado de reconexión, con el CR y la PC. Este método hace lo siguiente. Primero indica al modo de operación que lleve a cabo la lectura de la orden recibida, esto lo hace invocando Mode::read, pasándole como argumento el controlador principal. Luego, le cambia el estado de operación, al controlador principal, invocando MainController::changeState, pasándole como argumento el estado **Working**.

actionNoMsg recibe como argumento el controlador principal **MainController**. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está en el estado de reconexión y no se ha recibido una orden. Este método hace lo siguiente. Primero obtiene el modo de operación del sistema, invocando MainController::getMode. Luego, delega en dicho modo, el cambio de modo de operación del controlador principal. Esto lo hace con una llamada a Mode::changeMode, pasándole como argumento el controlador principal.

control recibe un controlador principal **MainController** y lleva a cabo el control de los sub-sistemas. Su implementación coincide con aquella del método Working::control.

write recibe el controlador principal **MainController** y un escribiente **Writer**. Este método es responsable por iniciar la escritura de la información que debe ser enviada al exterior del sistema. Su implementación coincide con aquella del método Working::write

Ver especificación en Sección 2.3.

#### Secreto

Oculto cuál es el/los siguiente/s estado/s y cómo se llevan a cabo el control y la escritura de datos cuando el sistema está en el estado de reconexión.

### 7.2.6.5. Working

MI DP1 DP2 DP3 F

#### Función

Este es un módulo físico-concreto que implementa cierto comportamiento dependiente del estado; en particular, cuando el sistema está activo y llevando a cabo los controles correspondientes.

Working, siendo el constructor, recibe la orden MainCtrlOrder.

actionWithMsg recibe como argumento el controlador principal MainController y el modo de operación Mode en el que está el sistema. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema recibe una orden, estando el estado de funcionamiento correcto. Este método simplemente delega en el modo de operación, la lectura de la orden recibida. Esto lo hace invocando Mode::read, pasándole como argumento el controlador principal.

actionNoMsg recibe como argumento el controlador principal MainController. Este método será invocado en el método plantilla OperationState::read definido en el padre de este módulo. Implementa las acciones a llevar a cabo cuando el sistema está funcionando correctamente pero no se ha recibido una orden. Este método solo cambia el estado de operación del controlador principal, mediante MainController::changeState, pasándole como argumento el siguiente estado de operación establecido por el método setNextState. Este será, el primer estado WaitingN.

control recibe un controlador principal MainController y lleva a cabo el control de los sub-sistemas, invocando MainCtrlOrder::execute, pasándole como argumento el mencionado controlador.

write recibe el controlador principal MainController y un escribiente Writer. Este método es responsable por iniciar la escritura de la información que debe ser enviada al exterior del sistema. Esto lo hace del siguiente modo. Mediante MainController::getData y MainController::getCtrlSysPool obtendrá, respectivamente, los datos MainCtrlData del controlador principal y el grupo de sistemas de control ControlSystemPool. De los datos del controlador toma: la orientación en la que se está desplazando el robot (adelante/atrás), por medio de MainCtrlData::getOrientation; información sobre si ha habido una orden de frenado, invocando MainCtrlData::getStopOrder; y el modo de operación del sistema, por medio de MainCtrlData::getModeId. Así, la mencionada orientación, la información sobre la orden de frenado y el modo de operación, serán pasados como argumento a Writer::writeOrientation, Writer::writeStopOrder y Writer::writeModeId, respectivamente, para ser escritos.

Luego, el presente método indicará a los sistemas de control que deben escribir su información. Para esto, invocará ControlSystemPool::writeTo, pasándole como argumento el escribiente Writer. Finalmente, este método indicará que toda la información recavada debe ser enviada llamando a Writer::writeAll.

```
write(MainController mCtrl, Writer wtr){
    mData=mCtrl.getData()
    csp=mCtrl.getCtrlSysPool()
    orient=mData.getOrientation()
    boolStop=mData.getStopOrder()
    modeId=mData.getModeId()
    wrt.writeOrientation(orient)
    wrt.writeStopOrder(boolStop)
    wrt.writeModeId(modeId)
    csp.writeTo(wtr)
    wrt.writeAll() }
```

setNextState recibe el siguiente estado OperationState, al estado que este módulo representa. Este método será utilizado por única vez, al momento de su construcción, y recibirá el primer estado WaitingN.

Ver especificación en Sección 2.3.

#### Secreto

Oculto cuál es el/los siguiente/s estado/s y cómo se llevan a cabo el control y la escritura de datos cuando el sistema está funcionando correctamente.

### 7.2.6.6. REQUESTS

#### Función

Este es un módulo lógico que agrupa aquellos módulos que permitirán implementar las órdenes que el controlador principal impartirá a los diferentes agentes (sensores, controladores) que constituyen los sub-sistemas de control, a fin de llevar a cabo el

control total del sistema.

#### 7.2.6.6.1 MainCtrlOrder

MI DP F

##### Función

Este es un módulo físico-concreto que implementa la orden que llevará acabo el controlador principal cuando deba realizar el control de los subsistemas. Esta orden implementa una secuencia de órdenes, sobre los sistemas de control.

MainCtrlOrder, siendo el constructor, recibe como argumento las siguientes órdenes [SensorsWrite](#), [ControllersRead](#), [ControllersControl](#), [SaveWheelPositions](#) y [UpdateOrder](#).

execute recibe como argumento el controlador principal [MainController](#) y lleva a cabo una secuencia de órdenes sobre los sistemas de control. Para esto, primero obtiene el grupo de sistemas de control [ControlSystemPool](#), por medio de MainController::getCtrlSysPool. Luego, aplicará las órdenes descriptas a continuación, pasándoles como argumento el grupo de sistemas mencionado. La secuencia de órdenes es como sigue. Invoca SensorsWrite::execute, para que los sensores escriban los valores medidos en los correspondientes [Pipes](#). Llama a ControllersRead::execute para que los controladores de los sistemas lean los correspondientes valores escritos por los sensores. Invoca a UpdateOrder::execute para que se actualice, si corresponde, el estado de sistema de dirección (de acuerdo a si la velocidad de las ruedas delanteras es nula o no). Le indica a los controladores de los subsistemas que lleven adelante el control, mediante ControllersControl::execute. Por último, invoca SaveWheelPositions::execute para que sea registrada la posición de cada rueda. Luego de estas órdenes sobre el grupo de sistemas, lleva a cabo la actualización de la orientación (adelante/atrás) del movimiento que está llevando a cabo el robot; esto lo hace, llamando a MainController::updateOrientation.

##### Secreto

Oculto la secuencia de órdenes que llevará a cabo el controlador principal, sobre los sistemas de control.

#### 7.2.6.6.2 UpdateOrder

MI DP F

##### Función

Este es un módulo físico-concreto que implementa una orden, de actualización del estado de operación del sistema de dirección, en el caso en que la velocidad medida de una o ambas ruedas delanteras sea nula. En tal caso, esta orden indicará al sistema dirección tal situación, y este delegará en su estado interno las acciones a tomar.

execute recibe como argumento el grupo de sistemas de control [ControlSystemPool](#). A partir de estos, el método obtendrá los controladores [WheelController](#) de las dos ruedas delanteras, mediante WheelSystem::getController; y el controlador del sistema de dirección [DirController](#), por medio de DirSystem::getController. De los controladores de las ruedas tomará los datos [WheelCtrlData](#), invocando WheelController::getData; y a su vez, a partir de estos datos, podrá evaluar si la velocidad de cada rueda es nula o no, utilizando WheelCtrlData::velIsNull. En el caso en que la velocidad medida de ambas ruedas sea nula, el método indica dicha situación al sistema de dirección invocando DirController::twoWNull. En el caso en que la velocidad medida sea nula en solo una rueda, el método indicará al sistema de dirección dicha situación invocando DirController::oneWNull. En otro caso, el método no hará nada.

```
execute(ControlSystemPool csp){
    it=csp.wheelSystemIt()
    it.first()
    wSysDD=it.element()
    it.next()
    wSysDI=it.element()
    wCtrlDD=wSysDD.getController()
    wCtrlDI=wSysDI.getController()
    dDD=wCtrlDD.getData()
    dDI=wCtrlDI.getData()
    if dDD.velIsNull() and dDI.velIsNull()
        dSys=csp.getDirSystem()
        dCtrl=dSys.getController()
        dCtrl.twoWNull()
    else if (dDD.velIsNull() and (not dDI.velIsNull()) ) or ((not dDD.velIsNull()) and dDI.velIsNull())
        dSys=csp.getDirSystem()
        dCtrl=dSys.getController()
        dCtrl.oneWNull()
}
```

##### Secreto

Oculto las acciones que debe llevar a cabo el sistema, respecto al dispositivo de dirección, en el caso en que la velocidad medida en las ruedas delanteras sea nula.

### 7.2.6.6.3 Order

MI DP F

#### Función

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo una orden impartida por el controlador principal. Implementa solo el método plantilla `execute`.

`execute` recibe un grupo de sistemas `ControlSystemPool` y lleva a cabo acciones sobre cada sistema de control. Para esto, obtiene el iterador de los sistemas de ruedas mediante `ControlSystemPool::wheelSystemIt` y aplica sobre cada sistema una acción mediante la llamada a `actionOnWheelSys`. De un modo similar, aplica al sistema de dirección una acción mediante la llamada a `actionOnDirSys`.

```
execute(ControlSystemPool csp){
    it=csp.wheelSystemIt()
    it.first()
    while not it.end()
        wheelSys=it.element()
        actionOnWheelSys(wheelSys)
        it.next()
    dirSys=csp.getDirSystem()
    actionOnDirSys(dirSys)
}
```

`actionOnWheelSys` llevará a cabo una acción sobre un sistema de control de ruedas.

`actionOnDirSys` llevará a cabo una acción sobre un sistema de control de dirección.

#### Secreto

Ocultas las diferentes acciones que pueden ser llevadas a cabo sobre los sub-sistemas de control.

### 7.2.6.6.4 SaveWheelPositions

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa la orden para que los sistemas de control de ruedas registren las posiciones de las mismas.

`SaveWheelPositions`, siendo el constructor, recibirá la orden `SaveWPosition`, que le indica a un sistema de control de rueda que debe registrar la posición de la misma.

`actionOnWheelSys` recibe un sistema de control de ruedas `WheelSystem` e invoca `SaveWPosition::execute` pasándole como argumento el sistema de control recibido.

`actionOnDirSys` recibe un sistema de dirección `DirSystem` pero no hace nada, su implementación es vacía.

#### Secreto

Ocultas cómo indicar que la posición de las ruedas deben ser registradas.

### 7.2.6.6.5 SensorsWrite

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa la orden impartida a los sensores de los distintos sub-sistemas de control, para que emitan su señal; esto es, escribir los correspondientes valores en de sus conectores `Pipe`.

`SensorsWrite`, siendo el constructor, recibirá la orden `SensorWritesVel`, para que un sensor de velocidad escriba su valor en el pipe, y la orden `SensorWritesCnt`, para que un sensor de corriente escriba su valor en el pipe correspondiente.

`actionOnWheelSys` recibe un sistema de control de ruedas `WheelSystem` e invoca `SensorWritesVel::execute` y `SensorWritesCnt::execute`, pasándoles en ambos casos el sistema de control recibido.

`actionOnDirSys` recibe un sistema de control de dirección `DirSystem`, obtiene el sensor de dirección llamando a `DirSystem::getDirSensor` e indica al sensor que emita su señal por medio de una llamada `DirSensor::signal`.

#### Secreto

Ocultas cómo indicar a los sensores que emitan sus señales.

### 7.2.6.6.6 ControllersRead

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa la orden impartida a los controladores de los sistemas de control para que estos lean sus conectores `Pipe`.

ControllersRead, siendo el constructor, recibe la orden **ControllerReadsVel**, para que un controlador lea la velocidad del pipe asociado, y la orden **ControllerReadsCnt**, para que un controlador lea la corriente del pipe correspondiente.

actionOnWheelSys recibe un sistema de control de ruedas **WheelSystem** e invoca ControllerReadsVel::execute y ControllerReadsCnt::execute, pasándoles en ambos casos el sistema de control recibido.

actionOnDirSys recibe un sistema de dirección **DirSystem**, obtiene el controlador de dirección **DirController** llamando a DirSystem::getController y le ordena al controlador que lleve a cabo la lectura de la conexión, invocando DirController::readConnection.

*Secreto*

Oculto cómo indicar a los controladores de los sistemas de control que deben leer sus conectores.

#### 7.2.6.6.7 ControllersControl

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa la orden impartida a los controladores de los sistemas de control para que estos lleven a cabo el control requerido.

actionOnWheelSys recibe un sistema de control de ruedas **WheelSystem**, obtiene el correspondiente controlador **WheelController** llamando a WheelSystem::getController y le indica al controlador que lleve a cabo el control de la rueda, mediante una llamada a WheelController::control.

actionOnDirSys recibe un sistema de control de dirección **DirSystem**, obtiene su controlador mediante DirSystem::getController y le indica al controlador que lleve a cabo el control de la dirección, invocando DirController::control.

*Secreto*

Oculto cómo indicar a los controladores de los sistemas de control que lleven a cabo el control requerido.

#### 7.2.6.6.8 ControllersStop

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa la orden impartida a los controladores de los sistemas de control para que estos detengan el funcionamiento.

ControllersStop, siendo el constructor, recibe el algoritmo de control de rueda **ResetVel**, que permite indicar como nula la velocidad de una rueda, y el estado de operación **DStopping** del controlador de dirección, para que sea establecido en el mismo, a fin de iniciar la detención del dispositivo.

actionOnWheelSys recibe un sistema de control de ruedas y obtiene el correspondiente controlador llamando a WheelSystem::getController. Luego, establece el algoritmo de control que deberá ejecutarse, invocando WheelController::setCtrlAlgoritm y pasándole **ResetVel** como argumento. Después, indica que se lleve a cabo el control llamando a WheelController::control.

actionOnDirSys recibe un sistema de control de dirección **DirSystem** y obtiene su controlador mediante DirSystem::getController. Luego, cambia el estado de operación del controlador invocando DirController::changeOpState y pasándole como argumento **DStopping**. Después indica que se debe realizar el control, llamando a DirController::control.

*Secreto*

Oculto cómo indicar a los controladores de los sistemas de control que detengan el funcionamiento de los dispositivos (ruedas y dirección).

### 7.3. CONTROLSYSTEMS

*Función*

Este es un módulo lógico que contiene la porción del sistema que implementa los distintos sistemas de control.

#### 7.3.1. Pipe

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa un conector tipo tubo. Los clientes pueden leer y escribir en este.

write recibe una medida **Measure** y la escribe en el tubo; esto es, la guarda internamente.

read devuelve la medida **Measure**, guardada internamente.



*Secreto*

Oculto cómo se implementa un conector tubo.

## 7.3.2. DIRCONTROLSYSTEM

---

*Función*

Este es un módulo lógico que agrupa los módulos que constituyen un sistema de control de dirección.

### 7.3.2.1. DIRCONTROLLERSTATES

---

*Función*

Este es un módulo lógico que agrupa los módulos que constituyen el comportamiento, dependiente de los estados de operación, del sistema de dirección.

#### 7.3.2.1.1 DCtrlCommand

MI DP F

*Función*

Este es un módulo físico-abstracto que provee la interfaz de un comando que tendrá efecto sobre algún elemento del sistema de dirección.

execute ejecuta el comando.

*Secreto*

Oculto los comandos que pueden ejecutarse sobre el sistema de control de dirección.

#### 7.3.2.1.2 Turn

MI DP F

*Función*

Este es un módulo físico-concreto que implementa un comando que tendrá efecto sobre el dispositivo de dirección. En particular este comando enviará una señal de pulso al dispositivo, para que este de un paso de giro.

Turn, siendo el constructor, recibe el dispositivo de dirección **SteeringDevice** que hará girar.

execute invoca SteeringDevice::pulse.

*Secreto*

Oculto cómo realizar un paso de giro en el dispositivo de dirección.

#### 7.3.2.1.3 SetDirection

MI DP F

*Función*

Este es un módulo físico-concreto que implementa un comando que tendrá efecto sobre el dispositivo de dirección. En particular, este comando establecerá en el dispositivo, la dirección de giro a realizar (derecha/izquierda).

SetDirection, siendo el constructor, recibe el dispositivo de dirección **SteeringDevice** y los datos **DriCtrlData** del sistema de dirección.

execute obtiene de los datos, la dirección hacia dónde debe girar el dispositivo (izquierda/derecha), invocando DirCtrlData::getDirection. Si el valor obtenido es 0, enviará la señal de giro a la izquierda, llamando a SteeringDevice::left. En caso contrario, enviará una señal de giro hacia la derecha mediante SteeringDevice::right.

*Secreto*

Oculto cómo establecer hacia dónde debe girar el dispositivo de dirección.

#### 7.3.2.1.4 Enable

MI DP F

*Función*

Este es un módulo físico-concreto que implementa un comando que habilita el dispositivo de dirección.

Enable, siendo el constructor, recibe el dispositivo de dirección **SteeringDevice** a habilitar.

execute invoca SteeringDevice::enable

*Secreto*

Oculto cómo habilitar el dispositivo de dirección.



#### 7.3.2.1.5 Disable

MI DP F

##### Función

Este es un módulo físico-concreto que implementa un comando que deshabilita el dispositivo de dirección.

Disable, siendo el constructor, recibe el dispositivo de dirección **SteeringDevice** a deshabilitar.

execute invoca SteeringDevice::disable

##### Secreto

Oculta cómo deshabilitar el dispositivo de dirección.

#### 7.3.2.1.6 DeviceState

MI DP F

##### Función

Este es un módulo físico-abstracto que representa un estado, de encendido o no, del dispositivo dirección. Este módulo provee la interfaz para encender y apagar el dispositivo, en un cierto estado.

on enciende el dispositivo.

off apaga el dispositivo.

##### Secreto

Oculta los estados de encendido y apagado del dispositivo de dirección.

#### 7.3.2.1.7 OnState

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que provee una interfaz para responder a un pedido de encender o apagar el dispositivo de dirección, estando el dispositivo encendido.

OnState, siendo el constructor, recibe el comando **Disable** para deshabilitar el dispositivo de dirección y el siguiente estado **OffState**, para el caso en que el dispositivo sea apagado.

on recibe el controlador del sistema de dirección pero no hace nada; su implementación es vacía.

off recibe el controlador del sistema de dirección **DirController** para poder cambiarle estado si corresponde. Primero, el método deshabilita el sistema de dirección, invocando Disable::execute; luego, cambia el estado en el controlador mediante DirController::changeDvState pasándole como argumento **OffState**.

##### Secreto

Oculta las acciones a tomar, ante un pedido de encender o de apagar el dispositivo, cuando este está encendido.

#### 7.3.2.1.8 OffState

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que provee una interfaz para responder a un pedido de encender o apagar el dispositivo de dirección, estando el dispositivo apagado.

OffState, siendo el constructor, recibe el comando **Enable** para habilitar el dispositivo de dirección.

on recibe el controlador del sistema de dirección **DirController**, invoca el comando Enable::execute y cambia el estado del dispositivo, en el controlador, llamando a DirController::changeDvState y pasándole como argumento el estado **OnState**.

off recibe el controlador del sistema de dirección pero no hace nada; su implementación es vacía.

setNextState recibe un estado **DeviceState** del dispositivo y lo mantiene como el siguiente estado, del estado que este módulo representa. Este método será utilizado por única vez, cuando el módulo sea creado, para establecer cual será el siguiente estado. En particular, será el estado de encendido **OnState**.

##### Secreto

Oculta las acciones a tomar, ante un pedido de encender o de apagar el dispositivo, cuando este está apagado.

#### 7.3.2.1.9 DOperationState

MI DP F

##### Función

Este es un módulo físico-abstracto que provee la interfaz de un estado de operación, del sistema de dirección, en el que se debe

procesar una orden de giro. Ver Statecharts 2.10.

control recibe el controlador del sistema de dirección **DirController** y lleva a delante el control del sistema de dirección, si corresponde.

turn recibe el controlador del sistema de dirección **DirController** y da un paso de giro en el sistema de dirección, si corresponde.

oneWNull recibe el controlador del sistema de dirección **DirController** y, asumiendo que sólo en una de las ruedas delanteras se a medido velocidad nula, llevará a cabo las acciones correspondientes.

twoWNull recibe el controlador del sistema de dirección **DirController** y, asumiendo que en las dos ruedas delanteras se a medido velocidad nula, llevará a cabo las acciones correspondientes.

#### *Secreto*

Oculta los estado de operación del sistema de dirección y los distintos comportamientos que se llevarán adelante, dependiendo del estado en el que esté el sistema.

### 7.3.2.1.10 DInactive

**MI** **DP** **F**

#### *Función*

Este es un módulo físico-concreto que implementa el comportamiento que debe tener el sistema de dirección, si está inactivo, ante la velocidad nula medida en una o ambas ruedas delanteras; o bien, ante un pedido de control o de giro.

control recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

turn recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

oneWNull recibe el controlador del sistema de dirección **DirController** y, asumiendo que la velocidad medida de las ruedas delanteras es nula en solo una, enciende el sistema de dirección invocando DirController::on.

```
oneWNull(DirController dCtrl){    dCtrl.on() }
```

twoWNull recibe el controlador del sistema de dirección **DirController** y, asumiendo que la velocidad medida de ambas ruedas delanteras es nula, apaga el sistema de dirección invocando DirController::off.

```
twoWNull(DirController dCtrl){    dCtrl.off() }
```

#### *Secreto*

Oculta las acciones a tomar cuando hay un pedido de controlar o de girar el dispositivo de dirección, pero el sistema está inactivo.

### 7.3.2.1.11 DTurning

**MI** **DP1** **DP2** **F**

#### *Función*

Este es un módulo físico-concreto que implementa el comportamiento que debe tener el sistema de dirección, ante un pedido de control o de giro, si el sistema está girando el dispositivo.

DTruning, siendo el constructor, recibe el comando **Turn** para llevar adelante un paso de giro en el dispositivo y lo guarda internamente. Además, este constructor establece el error tolerado  $ERROR_{inf}$ , entre la posición deseada y la posición medida, para determinar que estas posiciones coinciden. De acuerdo al requerimiento **RF-75.2.**,  $ERROR_{inf}$  es igual a 0.

control recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

turn recibe el controlador del sistema de dirección **DirController** y hace lo siguiente. Primero registra la posición temporal del dispositivo de dirección invocando DirController::readTempPos. Luego, obtiene dicha posición, accediendo a los datos por medio de DirController::getData y obteniendo el valor mediante DirCtrlData::getTempPos. Después, toma el valor de referencia llamando a DirCtrlData::getSetpoint. Entonces, calcula el valor absoluto de la diferencia entre la posición temporal y el valor de referencia. Si dicho valor es mayor a 0 ( $ERROR_{inf}$ ), indicará hacer un paso de giro, invocando Turn::execute. En caso contrario, cambiará el estado de operación del controlador de dirección, llamando a DirController::changeOpState y pasándole como argumento el estado **DActive**.

```
turn(DirController dCtrl){
    dCtrl.readTempPos()
    data=dCtrl.getData()
    pos=data.getTempPos()
    sp=data.getSetpoint()
    if abs(dif(pos,sp)) > ERRORinf
        cmdTurn.execute()
    else
        dCtrl.changeOpState(activeSt)
}
```

oneWNull recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

twoWNull recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

setNextState recibe un estado **DOperationState** y lo mantiene como el siguiente estado, al estado que representa este módulo. Este método será utilizado por única vez, al momento de construir el módulo, para establecer cuál será el siguiente estado. En particular, el método recibirá como argumento el estado **DActive**.

*Secreto*

Oculta las acciones a tomar cuando hay un pedido de controlar o de girar el dispositivo de dirección y el sistema está girando.

### 7.3.2.1.12 DActive

MI DP1 DP2 DP3 DP4 F

*Función*

Este es un módulo físico-concreto que implementa el comportamiento que debe tener el sistema de dirección, si está activo a la espera de una orden de giro, ante la velocidad nula medida en una o ambas ruedas delanteras; o bien, ante un pedido de control o de giro.

**DActive**, siendo el constructor, recibe el siguiente estado de operación **DTurning**, en el que se realizará el giro, y el comando **SetDirection**, para establecer la dirección de giro (izquierda/derecha). Además, mediante **Constants::getM**, este método obtendrá y guardará el valor de giro mínimo  $GIRO_{min}$  requerido, para dar curso al proceso de girar el dispositivo. Ver requerimiento **RF-73**.

```
DActive(DTurning trn, DStopping stpp, DInactive inact, SetDirection cmd){
    const=Constants::instance()
    GIROmin= const.getM()
    turnSt=trn
    stoppingSt=stpp
    inactSt=inact
    cmdSetDir=cmd
}
```

control recibe el controlador del sistema de dirección **DirController** como argumento. Primero, el método enciende el dispositivo invocando **DirController::on**. Luego, obtiene los datos **DirCtlData** y el algoritmo **DirAlgorithm**, mediante **DirController::getData** y **DirController::getAlgorithm**. Después, ejecuta el algoritmo llamando a **DirAlgorithm::execute** pasándole como argumento los datos mencionados. Entonces, toma la diferencia entre la posición del dispositivo y la posición deseada, mediante **DirCtlData::getErrorPos**. En el caso en que dicha diferencia sea menor que el giro mínimo  $GIRO_{min}$  establecido, el método no hará nada. En caso contrario, el método dará inicio al giro del dispositivo del siguiente modo. Establecerá el sentido de giro invocando **SetDirection::execute** y luego cambiará el estado de operación en el controlador, mediante **DirController::changeOpState** pasándole como argumento el estado **DTurning**.

```
control(DirController dCtrl){
    dCtrl.on()
    data=dCtrl.getData()
    alg=dCtrl.getAlgorithm()
    alg.calculate(data)
    error=data.getErrorPos()
    if error >= GIROmin
        cmdSetDir.execute()
        dCtrl.changeOpState(turnSt)
}
```

turn recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

oneWNull recibe el controlador **DirController**, enciende el sistema de dirección mediante **DirController::on** y cambia el estado del controlador mediante **DirController::changeOpState**, pasándole como argumento el estado **DInactive**.

```
oneWNull(DirController dCtrl){
    dCtrl.on()
    dCtrl.changeOpState(inactSt)
}
```

twoWNull recibe el controlador **DirController** y cambia el estado del controlador mediante **DirController::changeOpState**, pasándole como argumento el estado **DStopping**.

```
twoWNull(DirController dCtrl){
    dCtrl.changeOpState(stoppingSt)
}
```

*Secreto*

Ocultas las acciones a tomar cuando hay un pedido de controlar o de girar el dispositivo de dirección y el sistema está activo a la espera de una orden.

#### 7.3.2.1.13 DStopping

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa el comportamiento que debe tener el sistema de dirección, ante un pedido de control o de giro, si el sistema está deteniéndose.

DStopping, siendo el constructor, recibe el siguiente estado de operación **DInactive**, del estado de operación que este módulo representa.

control recibe el controlador del sistema de dirección **DirController** como argumento. Primero, el método apaga el dispositivo invocando DirController::off. Luego, cambia el estado de operación en el controlador, mediante DirController::changeOpState pasándole como argumento el estado **DInactive**.

turn recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

oneWNull recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

twoWNull recibe el controlador del sistema de dirección **DirController** pero no hace nada; su implementación es vacía.

##### Secreto

Ocultas las acciones a tomar cuando hay un pedido de controlar o de girar el dispositivo de dirección y el sistema está deteniéndose.

#### 7.3.2.2. DirController

MI DP1 DP2 DP3 F

##### Función

Este es un módulo físico-concreto que implementa un controlador del dispositivo de dirección. Está constituido por un conjunto de datos **DirCtrlData**, un algoritmo de control **DirAlgorithm**, un estado de operación **DOperationState** y un conector **Pipe**.

**DirController**, siendo el constructor, recibirá un conjunto de datos **DirCtrlData** y lo mantendrá internamente. Además este método, construirá y mantendrá el algoritmo **DirAlgorithm** que lleva a cabo los cálculos de la dirección.

setConnection(i Pipe) establece cuál será el conector que utilizará para conectarse con el sensor **DirSensor**. Este método toma el pipe recibido y lo guarda internamente.

readConnection lee del **Pipe** y registra la posición del dispositivo. Para esto, obtiene del pipe la posición mediante Pipe::read. Luego, guarda dicha posición invocando DirCtrlData::savePosition y pasándosela como argumento. Además, registra la posición mencionada como una posición temporal, llamando a DirCtrlData::saveTempPos y pasándosela como argumento.

readTempPos lee del **Pipe** la posición mediante Pipe::read y la guarda invocando saveTempPos, pasándole como argumento a este último la mencionada posición.

setSetpoint recibe un valor de referencia **Measure** y lo guarda en el conjunto de datos. En particular, recibirá un valor **SignedPerc**. Para esto, llama a DirCtrlData::saveSetpoint y le pasa como argumento el valor recibido.

getData devuelve datos **Data** del controlador. En particular, datos **DirCtrlData**.

getAlgorithm devuelve el algoritmo **Algorithm** que lleva a delante los cálculos del dispositivo de dirección. En particular, devolverá un algoritmo **DirAlgorithm**.

changeDvState recibe un estado del dispositivo de dirección **DeviceState** y lo registra internamente como el estado actual del dispositivo.

changeOpState recibe un estado de operación **DOperationState** y lo registra internamente como el estado de operación actual del sistema de dirección.

turn indica dar un paso de giro en el dispositivo de dirección. Este método delega tal acción en el estado de operación actual, invocando DOperationState::turn y pasando como argumento el presente módulo.

control indica que se debe realizar un control. Para esto, delega tal acción en el estado de operación actual invocando DOperationState::control y pasando como argumento el presente módulo.

onWNull indica que se ha detectado en las ruedas delanteras, que solo una mide velocidad nula. Para esto, delega tal acción en el estado de operación actual invocando DOperationState::oneWNull y pasando como argumento el presente módulo.

twoWNull indica que se ha detectado en las dos ruedas delanteras, velocidad nula. Para esto, delega tal acción en el estado de operación actual invocando DOperationState::twoWNull y pasando como argumento el presente módulo.

on indica que se debe encender el dispositivo de dirección. El método delega dicha acción en el estado del dispositivo, llamando a DeviceState::on y pasándole la referencia del presente módulo como argumento.

off indica que se debe apagar el dispositivo de dirección. Este método delega la mencionada acción en el estado del dispositivo de dirección, por medio de la llamada a DeviceState::off pasándole como argumento el presente módulo.

*Secreto*

Oculta cómo se lleva a cabo el control del dispositivo de dirección.

### 7.3.2.3. DirSystem

MI DP F

*Función*

Este es un módulo físico-concreto que agrupa los elementos de un sistema de control de dirección. Está compuesto por un controlador de dirección **DirController** y un sensor **DirSensor**; y provee la interfaz para acceder a estos.

DirSystem, siendo el constructor, recibe un controlador **DirController** y un sensor de dirección **DirSensor** los cuales mantendrá internamente.

getController devuelve el controlador **DirController** del sistema de control de dirección.

getDirSensor retorna el sensor de dirección **PassiveSensor** del sistema. En particular, retornará un sensor **DirSensor**.

*Secreto*

Oculta cómo mantiene los elementos que constituyen un sistema de control de dirección.

### 7.3.2.4. DirCtrlTimeOut

MI DP1 DP2 F

*Función*

Este es un módulo físico-concreto que implementa un comando que llevará a cabo acciones sobre el controlador **DirController** del sistema de dirección y sobre el sensor de dirección **DirSensor**. Este comando será invocado ante cada interrupción física ocasionada por el temporizador secundario **SecondTimer**.

DirCtrlTimeOut, siendo el constructor, recibirá el controlador **DirController** del sistema de dirección y el sensor del dispositivo de dirección **DirSensor**.

execute indica al sensor que debe enviar el valor medido, invocando DirSensor::signal; y luego, ordena al controlador que realice un paso de giro llamando a DirController::turn.

*Secreto*

Oculta sobre qué módulos del sistema de dirección llevará a cabo acciones, cuando sea invocado por el temporizador secundario **SecondTimer**.

## 7.3.3. WHEELCONTROLSYSTEMS

*Función*

Este es un módulo lógico que agrupa los módulos que constituyen un sistema de control de rueda.

### 7.3.3.1. VELOCITYSENSOR

*Función*

Este es un módulo lógico que agrupa los módulos que implementan un sensor de velocidad.

#### 7.3.3.1.1 VelSensor

MI DP F

*Función*

Este es un módulo físico-concreto que está compuesto por un acumulador de las señales físicas provenientes de un sensor Hall. Dicho acumulador es un objeto tipo **SensorCollector**. Además, este módulo mantendrá internamente la posición **Position** medida de la reuda y el **Pipe** por medio del cual enviará la información.

VelSensor, siendo el constructor, recibe un recolector **SensorCollector** el cual guardará internamente. Además, mediante

`Constants::getDELTAT`, obtendrá el período de tiempo  $\Delta T$  (`deltaT`) de un ciclo de ejecución y lo guardará internamente para futuros cálculos.

```
VelSensor(SensorCollector sc){
    vColl=sc
    const=Constants::instance()
    deltaT=const.getDELTAT()
}
```

`setConnection` recibe un `Pipe` y lo mantiene internamente. Dicho pipe será el que permitirá la conexión entre este sensor y el controlador de la rueda.

`signal` implementa la emisión de una señal del sensor; esto es, la escritura de cierta información en el `Pipe` correspondiente. Este método será invocado en cada interrupción del temporizador principal `FirstTimer`, a fin de que el sensor envíe su señal al controlador. Primero, el método indica al recolector de señales que se inicia un período, invocando `SensorCollector::initPeriod`. Luego, obtiene el último período de tiempo  $\Delta t_s$  registrado por el recolector y la cantidad de interrupciones  $n\_interrup$  registradas en este. Esto lo hace llamando a `SensorCollector::periodTime` y `SensorCollector::preSum`. Con los valores mencionados y el valor de un período de tiempo de un ciclo de ejecución  $\Delta T$  (`deltaT`), el método calcula la velocidad medida como descrito en el requerimiento **RF-52**. Finalmente, la velocidad medida **RPM** será enviada por el pipe, llamando a `Pipe::write` y pasándole la mencionada velocidad como argumento.

```
vColl:SensorCollector

signal(){
    vColl.initPeriod()
    t = vColl.periodTime()
    nInterrup = vColl.preSum()
    velRPM = func(t,nInterrup,deltaT)
    pipe.write(velRPM)
}
```

`getPosition` devuelve la posición de la rueda `Measure`, en particular un elemento `Position`. Para esto, el método toma el valor mediante una invocación a `SensorCollector::total`; luego, establece dicho valor como una posición llamando a `Position::setValue` y pasándole el valor mencionado. Finalmente, retorna el elemento `Position`.

#### Secreto

Oculta los cálculos llevados a cabo para obtener la posición y la velocidad medida de una rueda.

### 7.3.3.1.2 SensorCollector

MI DP1 DP2 F

#### Función

Este es un físico-módulo concreto responsable de registrar las interrupciones provenientes de un sensor Hall (`ActiveSensor`) y establecer el período de tiempo transcurrido entre la primera interrupción y la última. En particular el módulo guardará la cantidad de interrupciones ocurridas en el período de control anterior y el tiempo transcurrido entre la primera y la última de estas. A su vez, irá contando las interrupciones ocurridas en el ciclo de control actual, y el tiempo transcurrido entre la primera interrupción y la última llevada a cabo.

`SensorCollector`, siendo el constructor, recibe un recolector de tiempo `TimeCollector` en el que delegará algunas tareas.

`currentTime` registra internamente el momento actual de cuando el método es invocado. En particular, este método es invocado ante la ocurrencia de una interrupción de un sensor Hall. Para registrar el mencionado momento, delega la tarea al recolector de tiempo, por medio de una invocación a `TimeCollector::currentTime`. Luego, si el número de interrupciones hasta el momento es 0 (`nbr==0`), esto indica que se ha iniciado un ciclo de control y por tanto se debe registrar el instante de tiempo de la primera interrupción del ciclo. En tal caso, esta función obtiene dicho momento invocando `TimeCollector::getCurrentTime` y lo guarda en una variable interna, digamos `initialT`. En caso contrario, la función termina.

```
currentTime(){
    coll.currentTime()
    if nbr == 0
        initialT = coll.getCurrentTime()
}
```

`getCurrentTime` devuelve un valor real, siendo este el último instante de tiempo registrado por el recolector. Para esto, delega la acción en el recolector de tiempo, retornando el valor resultante de invocar `TimeCollector::getCurrentTime`.

`addOne` incrementa un contador interno `nbr`, que cuenta las ocurrencias de interrupción de un sensor Hall.

`preSum` devuelve la cantidad total de interrupciones `preN`, ocurridas durante el ciclo de control anterior. Dicha cantidad es mantenida internamente en una variable, el método solo retorna la cantidad mencionada.

total devuelve la cantidad total de interrupciones **totalN**, ocurridas desde el primer ciclo de control llevado a cabo hasta el ciclo de control anterior al actual. Dicha cantidad es mantenida internamente en una variable y el método solo la devuelve.

periodTime devuelve el tiempo **periodT** transcurrido entre la primer interrupción y la última, del ciclo de control anterior. Dicho valor es mantenido en una variable y este método solamente retorna el valor.

initPeriod es llamado cuando se inicia un nuevo ciclo de control; esto es, cuando tiene lugar una interrupción del temporizador principal **FirstTimer**. Este método es el que lleva a cabo todos los cálculos con cada inicio de ciclo. Primero, el método guarda la cantidad de interrupciones ocurridas en el ciclo anterior (**nbr**), en una variable interna (**preN=nbr**) e incrementa el contador total de interrupciones, digamos **totalN**, con la mencionada cantidad (**totalN=totalN+nbr**). Luego calcula el período de tiempo entre la ocurrencia de la primera interrupción y la ocurrencia de la última. Es decir, calcula la diferencia entre el valor obtenido mediante TimeCollector::getCurrentTime e **initialT**, y lo guarda internamente (**periodT**). Finalmente, inicializa con 0 el número de interrupciones del ciclo (**nbr=0**).

```
coll:TimeCollector

initPeriod(){
    preN = nbr
    totalN = totalN + nbr
    periodT = coll.getCurrentTime() - initialT
    nbr = 0
}
```

*Secreto*

Oculta el modo en el que se van registrando la cantidad interrupciones y los períodos transcurridos entre estas.

### 7.3.3.1.3 CountSignal

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa un comando que tiene efecto sobre un recolector **SensorCollector**. El método es responsable de invocar los métodos del acumulador de interrupciones **SensorCollector**, cada vez que ocurre una interrupción de un sensor Hall (**ActiveSensor**). De este modo, se registrará cada ocurrencia de una interrupción y en qué instante esta tuvo lugar.

CountSignal, siendo el constructor, recibirá como argumento un acumulador **SensorCollector**.

execute registra en el acumulador el instante actual, mediante SensorCollector::currentTime, y luego incrementa el contador de interrupciones invocando SensorCollector::addOne.

*Secreto*

Oculta las acciones a llevar a cabo para registrar la ocurrencia de una interrupción de sensor Hall. Oculta sobre qué módulo o módulos tiene efecto su ejecución.

### 7.3.3.2. CURRENTSENSOR

*Función*

Este es un módulo lógico que agrupa los módulos que implementan un sensor de corriente.

#### 7.3.3.2.1 CntSensor

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que está compuesto por un acumulador, de los valores de la corriente medida de una rueda. Dicho acumulador es un objeto tipo **ValueCollector**.

CntSensor, siendo el constructor, recibe un recolector **ValueCollector** el cual mantendrá internamente.

setConnection recibe un **Pipe** y lo mantiene internamente. Dicho pipe será el que permitirá la conexión entre este sensor y el controlador de la rueda.

signal implementa la emisión de una señal del sensor; esto es, la escritura de cierta información en el **Pipe** correspondiente. Este método será invocado en cada interrupción del temporizador principal **FirstTimer**, a fin de que el sensor envíe su señal al controlador. Primero, el método obtiene del recolector el valor medido de la corriente, invocando ValueCollector::getCurrentVal para luego iniciar un período en el acumulador, llamando a ValueCollector::initPeriod. Luego, el valor obtenido es establecido como un porcentaje signado, por medio de SignedPerc::setValue, para luego este último ser enviado por el pipe, mediante Pipe::write.

```

cColl:ValueCollection
signedPerc:SignedPerc

signal(){
    vCnt=cColl.getCurrentVal()
    cColl.initPeriod()
    signedPerc.setValue(vCnt)
    pipe.write(signedPerc)
}

```

*Secreto*

Oculto los cálculos llevados a cabo para obtener la corriente medida de una rueda.

### 7.3.3.2.2 ValueCollector

**MI** **DP1** **DP2** **F**

*Función*

Este es un módulo físico-concreto responsable de registrar las lecturas de los valores, de la corriente medida de una rueda. En particular, el módulo acumulará estos valores sumándolos. A su vez, irá contando la cantidad de lecturas realizadas, a fin de calcular el promedio y devolverlo.

ValueCollection, siendo el constructor, recibe un identificador del registro desde dónde serán leídos los valores. El constructor, con dicho identificador, logrará identificar internamente, desde qué registro leerá los valores de la corriente. Además, este método creará y mantendrá internamente un repositorio de datos **SimpleData** y una función **InverseFunction**.

initPeriod inicia un ciclo de lectura de valores. Este método pone en 0, el contador (**nbr**) de las lecturas realizadas y la variable (**cnt**) en donde se suman los valores leídos.

getCurrentVal devuelve un valor real, que es la corriente medida. Dicho valor es el promedio de las lecturas realizadas hasta ese momento (**cnt/nbr**).

currentVal registra una lectura de la corriente medida. Primero, el método lee del registro correspondiente el valor; esto es, un entero de 16bits. Luego, transforma dicho valor en un valor real en el intervalo [-100,100]. Para esto, establece el entero de 16bits como el argumento de una función, mediante **SimpleData::setArg**. Luego invoca la función **InverseFunc** pasándole el mencionado argumento. Una vez transformado el valor, el método lo obtiene por medio de **SimpleData::getResult** y lo suma a la variable (**cnt**) que acumula dichas lecturas.

addOne incrementa en uno un contador interno **nbr**, que cuenta las lecturas llevadas a cabo.

*Secreto*

Oculto desde dónde se leen los valores de la corriente registrada y cómo se computa la corriente medida que el sensor de corriente deberá emitir.

### 7.3.3.2.3 ReadCnt

**MI** **DP1** **DP2** **F**

*Función*

Este es un módulo físico-concreto que implementa un comando que tiene efecto sobre un recolector **ValueCollection**. Este comando es responsable de invocar los métodos del recolector, cada vez que ocurre una interrupción provocada por el temporizador secundario **SecondTimer**. Dicha interrupción, establece el momento en el que se debe leer y acumular los valores de la corriente registrada de una rueda, los cuales serán promediados más tarde para establecer cuál es la corriente medida que el sensor **CntSensor** enviará al controlador.

ReadCnt, siendo el constructor, recibirá como argumento un acumulador **ValueCollection**.

execute registra en el acumulador, el valor de la corriente de una rueda leído en ese el instante, mediante **ValueCollection::currentVal**, y luego incrementa en uno el contador de lecturas realizadas invocando **ValueCollection::addOne**.

*Secreto*

Oculto cómo se llevan adelante las lecturas de la corriente de una rueda, en cada instante marcado por el temporizador secundario.

### 7.3.3.3. CONTROLALGORITHMS

*Función*

Este es un módulo lógico que agrupa los módulos que implementan los distintos comandos y algoritmos que permiten controlar



los subsistemas de control de ruedas.

#### 7.3.3.3.1 WSysOrder

MI DP F

##### Función

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo órdenes sobre un sistema de control de ruedas.

execute recibe un sistema de control de ruedas **WheelSystem** y lleva a cabo la orden sobre éste.

##### Secreto

Oculta las distintas órdenes que pueden ser ejecutadas sobre un sistema de control de ruedas.

#### 7.3.3.3.2 SaveWPosition

MI DP F

##### Función

Este es un módulo físico-concreto, que implementa la orden que establece que se debe registrar la posición en la que está una rueda.

execute recibe como argumento un sistema de control de ruedas **WheelSystem** y a partir de este hace lo siguiente. Primero, obtiene el sensor de velocidad **VelSensor** para luego tomar de este la posición de la rueda. Esto lo hace mediante una llamada a WheelSystem::getVelSensor y a VelSensor::getPosition. Después, obtiene el controlador **WheelController** y de este toma los datos **WheelCtrlData**. Para esto, invoca WheelSystem::getController y WheelController::getData. Finalmente, registra la posición tomada del sensor, mediante WheelCtrlData::savePosition pasándole como argumento la posición mencionada.

##### Secreto

Oculta cómo se registra la posición en la que está una rueda.

#### 7.3.3.3.3 SensorWritesVel

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto, que implementa la orden que le indica al sensor de velocidad, de un sistema de control de ruedas, que debe emitir el valor de la velocidad medida.

execute recibe como argumento un sistema de control de ruedas **WheelSystem** y a partir de este hace lo siguiente. Primero, obtiene el sensor de velocidad **VelSensor** y luego le indica que emita su señal. Esto lo hace mediante una llamada a WheelSystem::getVelSensor y a VelSensor::signal.

##### Secreto

Oculta cómo se le indica a un sensor de velocidad que debe emitir su valor.

#### 7.3.3.3.4 ControllerReadsVel

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto, que implementa la orden que le indica a un controlador, de un sistema de control de ruedas, que debe leer la velocidad medida.

execute recibe como argumento un sistema de control de ruedas **WheelSystem** y a partir de este hace lo siguiente. Primero, obtiene el controlador **WheelController**; y luego, le indica a este que lea la conexión correspondiente al sensor de velocidad. Esto lo hace mediante una llamada a WheelSystem::getController y a WheelController::readConnectionV.

##### Secreto

Oculta cómo se le indica a un controlador de rueda que debe leer la velocidad medida en la correspondiente conexión.

#### 7.3.3.3.5 SensorWritesCnt

MI DP F

##### Función

Este es un módulo físico-concreto, que implementa la orden que le indica al sensor de corriente, de un sistema de control de ruedas, que debe emitir el valor de la corriente medida.

execute recibe como argumento un sistema de control de ruedas **WheelSystem** y a partir de este hace lo siguiente. Primero, obtiene el sensor de corriente **CntSensor** y luego le indica que emita su señal. Esto lo hace mediante una llamada a WheelSystem::getCntSensor y a CntSensor::signal.

##### Secreto

Oculta cómo se le indica a un sensor de corriente que debe emitir su valor.

#### 7.3.3.3.6 ControllerReadsCnt

MI DP F

##### Función

Este es un módulo físico-concreto, que implementa la orden que le indica a un controlador, de un sistema de control de ruedas, que debe leer la corriente medida.

execute recibe como argumento un sistema de control de ruedas **WheelSystem** y a partir de este hace lo siguiente. Primero, obtiene el controlador **WheelController**; y luego, le indica a este que lea la conexión correspondiente al sensor de corriente. Esto lo hace mediante una llamada a WheelSystem::getController y a WheelController::readConnectionC.

##### Secreto

Oculto cómo se le indica a un controlador de rueda que debe leer la corriente medida en la correspondiente conexión.

#### 7.3.3.3.7 WCtrlCommand

MI DP F

##### Función

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo la ejecución de un comando que tendrá efecto sobre ciertos elementos de un sistema de control de ruedas. Dichos comandos serán utilizados en los algoritmos de control **WCtrlAlgorithm** dependiendo si se quiere detener la rueda o si hay o no cambio de sentido (adelante/atrás) en el movimiento.

execute ejecuta un comando.

##### Secreto

Oculto distintos comandos que tendrán efecto sobre ciertos elementos de un sistema de control de ruedas.

#### 7.3.3.3.8 VelNull

MI DP F

##### Función

Este es un módulo físico-concreto que implementa el comando, por medio del cual se les asigna tensión 0 a una rueda.

VelNull, siendo el constructor, recibe la rueda **Wheel** sobre la cual el comando tendrá efecto y la guarda internamente.

execute le envía un porcentaje signado **SignedPerc** con valor 0 a la rueda, mediante Wheel::setVoltage.

##### Secreto

Oculto cómo le provee a la rueda la orden de velocidad nula.

#### 7.3.3.3.9 Brake

MI DP F

##### Función

Este es un módulo físico-concreto que implementa el comando, por medio del cual se frena la rueda.

Brake, siendo el constructor, recibe la rueda **Wheel** sobre la cual el comando tendrá efecto y la guarda internamente.

execute le envía la orden de frenado a la rueda, invocando Wheel::brake.

##### Secreto

Oculto cómo indica el frenado de la rueda.

#### 7.3.3.3.10 SetTension

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa el comando, por medio del cual se le provee la tensión adecuada a la rueda.

SetTension, siendo el constructor, recibe la rueda **Wheel** sobre la cual el comando tendrá efecto, el conjunto de datos **WheelCtrlData** del sistema de control de ruedas y el controlador **WheelController**; y los guarda internamente.

execute ejecuta el comando realizando los siguientes pasos. Primero, obtiene el algoritmo que hará los cálculos correspondientes sobre la tensión a aplicar. Esto lo hace, invocando WheelController::getAlgorithm. Luego, realiza los cálculos por medio de una llamada a Algorithm::calculate, pasándole como argumento los datos que mantiene internamente; para después obtener de estos, la tensión calculada llamando a WheelCtrlData::getTens. Finalmente, invoca Wheel::setVoltage pasándole como argumento la tensión mencionada, para enviarla a la rueda.

##### Secreto

Oculto cómo determinar la tensión a aplicar a una rueda y cómo aplicarla.

### 7.3.3.3.11 ChangeOrientation

MI DP F

#### Función

Este es un módulo físico-concreto que implementa el comando, por medio del cual se llevará a cabo la orden de cambiar el sentido (adelante/atrás) del desplazamiento de una rueda.

ChangeOrientation, siendo el constructor, recibe la rueda **Wheel** sobre la cual el comando tendrá efecto, el conjunto de datos **WheelCtrlData** del sistema de control de ruedas, el mencionado sistema **WheelSystem**, la orden de escritura **SensorWritesVel** para el sensor de velocidad y la orden de lectura de velocidad **ControllerReadsVel** para el controlador; y los guarda internamente.

execute ejecuta el comando realizando los siguientes pasos. Primero, envía tensión 0 a las ruedas. Esto lo hace invocando Wheel::setVoltage, pasándole como argumento un porcentaje signado **SignedPerc** con valor 0. Luego el método leerá continuamente la velocidad medida, hasta que esta sea 0. Para esto, evalúa en un bucle el valor de la velocidad, mediante una llamada a WheelCtrlData::velIsNull. Si dicha velocidad no es nula, el método indica al sensor que envíe su valor medido nuevamente; e indica al controlador que lea dicho valor. Esto lo hace por medio de SensorWritesVel::execute y ControllerReadsVel::execute, pasando en ambos casos como argumento, el sistema **WheelSystem**. Una vez que el valor medido llega a 0, el método obtiene la orientación requerida previamente por una orden, utilizando WheelCtrlData::getOrientation, y la establece en la rueda mediante una llamada Wheel::setOrientation, pasándole como argumento dicha orientación.

Ver requerimiento RF-67.(4)

```
wheel: Wheel
data: WheelCtrlData
sensorWritesVel: SensorWritesVel
ctrlrlerReadsVel: ControllerReadsVel
wSys: WheelSystem

execute(){
    volt0=new SignedPerc()
    wheel.setVoltage(volt0)
    while (not data.velIsNull())
        sensorWritesVel.execute(wSys)
        ctrlrlerReadsVel.execute(wSys)
    orient=data.getOrientation()
    wheel.setOrientation(orient)
}
```

#### Secreto

Oculto cómo se lleva a cabo un cambio de sentido (adelante/atrás) en el movimiento de una rueda.

### 7.3.3.3.12 CtrlCmdPool

MI DP F

#### Función

Este es un módulo físico-concreto que implementa un contenedor de comandos que pueden ser utilizados dentro de los algoritmos de control de rueda **WCtrlAlgorithm**. Cada uno de estos comandos, tienen efecto sobre los elementos que constituyen un sistema de control de rueda en particular. El módulo provee una interfaz para acceder a cada uno de los comandos que lo constituyen.

CtrlCmdPool, siendo el constructor, recibe los cuatro comandos **WCtrlCommand** que lo componen: **VelNull** establece la velocidad nula para la rueda; **Brake** frena la rueda del sistema; **SetTension** envía la tensión adecuada a la rueda y **ChangeOrientation** cambia la orientación del sentido de movimiento de la rueda.

#### Secreto

Oculto los comandos que lo componen.

### 7.3.3.3.13 WCtrlAlgorithm

MI DP1 DP2 F

#### Función

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo un algoritmo que permitirá el movimiento o no de las ruedas.

execute recibe un grupo de comandos **CtrlCmdPool** que tendrán efecto sobre los elementos de un sistema de control de rueda particular, y llevará a cabo determinado algoritmo.

#### Secreto

Oculto distintos algoritmos que pueden realizarse dependiendo de las órdenes recibidas por el sistema.

#### 7.3.3.3.14 Stop

MI DP1 DP2 DP3 F

##### Función

Este es un módulo físico-concreto que implementa el algoritmo que permite frenar la rueda del sistema de control.

execute recibe como argumento el grupo de comandos **CtrlCmdPool**. El método obtiene el comando **Brake**, mediante CtrlCmdPool::getBrake, y luego lo ejecuta invocando Brake::execute.

##### Secreto

Oculto los pasos a seguir para frenar una rueda.

#### 7.3.3.3.15 ResetVel

MI DP1 DP2 DP3 F

##### Función

Este es un módulo físico-concreto que implementa el algoritmo que permite restablecer la velocidad de la rueda a su valor inicial.

execute recibe como argumento el grupo de comandos **CtrlCmdPool**. El método obtiene el comando **VelNull**, mediante CtrlCmdPool::getVelNull, y luego lo ejecuta invocando VelNull::execute.

##### Secreto

Oculto los pasos a seguir para restablecer la velocidad de la rueda a su valor inicial.

#### 7.3.3.3.16 Advance

MI DP1 DP2 DP3 F

##### Función

Este es un módulo físico-concreto que implementa el algoritmo que permite que la rueda se desplace en el sentido que previamente haya sido establecido.

execute recibe como argumento el grupo de comandos **CtrlCmdPool**. El método obtiene el comando **SetTension**, mediante CtrlCmdPool::getSetTension, y luego lo ejecuta invocando SetTension::execute.

##### Secreto

Oculto los pasos a seguir para que la rueda se desplace en el sentido que haya sido previamente establecido.

#### 7.3.3.3.17 ReverseOrientation

MI DP F

##### Función

Este es un módulo físico-abstracto que provee una interfaz común, a distintos algoritmos que extenderán las responsabilidades establecidas en los algoritmos **WCtrlAlgorithm**. Los algoritmos herederos de este permitirán mover la rueda en sentido inverso al que estuviere funcionando.

execute recibe un grupo de comandos **CtrlCmdPool** que tendrán efecto sobre los elementos de un sistema de control de rueda particular, y llevará a cabo determinado algoritmo.

##### Secreto

Oculto distintos algoritmos que pueden realizarse para cambiar el sentido del desplazamiento de una rueda.

#### 7.3.3.3.18 Reverse

MI DP1 DP2 DP3 F

##### Función

Este es un módulo físico-concreto que implementa el algoritmo que permite que la rueda cambie el sentido de orientación (adelante/atrás) y luego se desplace. Este algoritmo se utiliza cuando se requiere un cambio de sentido y la orden para mover alguna de las ruedas, ha sido dada en tensión o en corriente.

Reverse, siendo el constructor, recibe como argumento el algoritmo **Advance** el cual mantendrá internamente.

execute recibe como argumento el grupo de comandos **CtrlCmdPool**. El método cambia el sentido de orientación de la rueda y luego la hace avanzar. Para esto, primero obtiene el comando **ChangeOrientation**, mediante CtrlCmdPool::getChangeOrient, y entonces lo ejecuta invocando ChangeOrientation::execute. Después, hace avanzar la rueda delegando tal tarea en el algoritmo **Advance**. Esto lo hace, invocando Advance::execute y pasándole como argumento el conjunto de comandos **CtrlCmdPool**.

##### Secreto

Oculto los pasos a seguir para que la rueda cambie el sentido de orientación (adelante/atrás) y se desplace. En particular, es para el caso en el que la orden para alguna de las ruedas a sido dada en tensión o corriente.

### 7.3.3.3.19 ReverseRPM

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa el algoritmo que permite que la rueda cambie el sentido de orientación (adelante/atrás) y luego se desplace. Este algoritmo se utiliza cuando se requiere un cambio de sentido y la orden para mover las ruedas, ha sido dada en **RPM** en todos los casos.

ReverseRPM, siendo el constructor, recibe como argumento los algoritmos **Stop** y **Reverse**, los cuales mantendrá internamente.

execute recibe como argumento el grupo de comandos **CtrlCmdPool**. El método frena la rueda, cambia el sentido de orientación y luego la hace desplazar. Para esto, primero delega el frenado de la rueda en el algoritmo **Stop**, invocando Stop::execute. Después, cambia el sentido de la rueda y la hace avanzar, delegando dicha tarea en el algoritmo **Reverse**. Esto lo hace, invocando Reverse::execute.

#### Secreto

Oculta los pasos a seguir para que la rueda cambie el sentido de orientación (adelante/atrás) y se desplace. En particular, es para el caso en el que la orden para cada una de las ruedas a sido dada **RPM**.

### 7.3.3.4. WheelController

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa un controlador de un sistema de control de rueda. Está constituido por un grupo de datos **Data**, un grupo de comandos **CtrlCmdPool**, algún algoritmo de control **WCtrlAlgorithm** sobre el desplazamiento de la rueda, algún algoritmo **Algorithm** para realizar los cálculos sobre la tensión a aplicar a la rueda, y dos conectores **Pipe** que permiten la conexión entre los sensores y el controlador que representa este módulo.

WheelController, siendo el constructor, recibe el grupo de datos **WheelCtrlData** que se usarán para controlar la rueda, y lo guarda internamente.

setConnectionV recibe como argumento un conector **Pipe** y lo establece como aquel que será el utilizado para conectarse al sensor de velocidad **VelSensor**. El conector recibido es mantenido internamente.

setConnectionC recibe como argumento un conector **Pipe** y lo establece como aquel que será utilizado para conectarse al sensor de corriente **CntSensor**. El conector recibido es mantenido internamente.

readConnectionV lee la velocidad medida, de la conexión asociada al sensor de velocidad, y la guarda como dato. Para esto, obtiene la velocidad medida invocando a Pipe::read; y luego, la guarda llamando a WheelCtrlData::saveVel, pasándole como argumento la mencionada velocidad.

readConnectionC lee la corriente medida, de la conexión asociada al sensor de corriente, y la guarda como dato. Para esto obtiene la corriente medida llamando a Pipe::read; y luego, la guarda invocando WheelCtrlData::saveCnt, pasándole como argumento dicha corriente.

setSetpoint recibe como argumento un valor de referencia **Measure**, para la velocidad de las ruedas, y lo guarda en el conjunto de datos. Esto lo hace invocando WheelCtrlData::saveSetpoint, pasándole como argumento el valor mencionado.

setOrientation recibe como argumento la orientación de desplazamiento deseada (adelante/atrás) **Measure** y la guarda en los datos. Para esto, invoca a WheelCtrlData::saveOrientation, pasándole como argumento la orientación mencionada.

setAlgorithm recibe como argumento un algoritmo **Algorithm**, que realizará el cálculo para determinar la tensión a aplicar a una rueda, y lo guarda internamente.

getAlgorithm devuelve el algoritmo **Algorithm**, que ha sido previamente establecido mediante el método setAlgorithm.

setCtrlAlgorithm recibe como argumento un algoritmo **WCtrlAlgorithm** y lo guarda internamente.

setCtrlCmdPool recibe como argumento el conjunto de comandos **CtrlCmdPool**, que actúa sobre los elementos del sistema de ruedas, y lo guarda internamente. Este método será utilizado por única vez, al momento de la construcción de este módulo, para determinar los comandos que tendrán efecto sobre los elementos del sistema de rueda, que el presente módulo controla.

getData retorna el conjunto de datos **Data** del módulo; en particular, un elemento **WheelCtrlData**.

control lleva a cabo el control de la rueda del siguiente modo. Utilizando el algoritmo de control **WCtrlAlgorithm**, que previamente haya sido asignado, lo ejecuta invocando WCtrlAlgorithm::execute, pasándole como argumento el grupo de comandos **CtrlCmdPool**.

#### Secreto

Oculta cómo llevar a cabo ciertas tareas que constituyen el control de una rueda.

### 7.3.3.5. WheelSystem

MI DP1 DP2 F

#### Función

Este es un módulo físico-concreto que implementa un sistema de control de rueda. Está compuesto por un controlador **WheelController** y dos sensores, uno de velocidad **VelSensor** y otro de corriente **CntSensor**.

**WheelSystem**, siendo el constructor, recibe como argumento un sensor de velocidad **VelSensor** y uno de corriente **CntSensor**; y los guarda internamente.

**setController** recibe el controlador **WheelController** del sistema de control de rueda, y lo mantiene internamente.

**getController** retorna el controlador **WheelController** del sistema.

**getVelSensor** retorna el sensor de velocidad **PassiveSensor**; en particular, un sensor **VelSensor**.

**getCntSensor** retorna el sensor de corriente **PassiveSensor**; en particular, un sensor **CntSensor**.

#### Secreto

Oculta cómo mantiene los elementos que constituyen el sistema de control de rueda.

## 7.4. PHYSICALDEVICES

#### Función

Este es un módulo lógico que agrupa los módulos que permiten la implementación de los dispositivos físicos del sistema.

### 7.4.1. Collector

MI DP F

#### Función

Este es un módulo físico-abstracto que provee una interfaz para registrar instantes de tiempo y para poder recuperarlos.

**currentTime** será responsable de registrar el instante de tiempo en el que el método sea invocado.

**getCurrentTime** devuelve el instante de tiempo registrado previamente por **currentTime**.

#### Secreto

Oculta distintos tipos de recolectores de instantes de tiempo.

### 7.4.2. TimeCollector

MI DP F

#### Función

Este es un módulo físico-concreto que implementa los métodos que permiten registrar instantes de tiempo y recuperarlos.

**currentTime** guardará en una variable interna el instante de tiempo actual.

**getCurrentTime** será responsable de devolver el instante de tiempo registrado previamente por **currentTime**.

#### Secreto

Oculta cómo obtiene y guarda el instante de tiempo actual.

### 7.4.3. DecoCollector

MI DP F

#### Función

Este es un módulo físico-abstracto que provee una interfaz para registrar instantes de tiempo y para poder recuperarlos. Además, declara internamente un **Collector** en el cual sus herederos delegarán tareas.

**currentTime** será responsable de registrar el instante de tiempo en el que el método sea invocado.

**getCurrentTime** será responsable de devolver el instante de tiempo registrado previamente por **currentTime**.

#### Secreto

Oculta los distintos tipos de recolectores de instantes de tiempo con funcionalidad extendida.

#### 7.4.4. Command

MI DP1 DP2 DP3 DP4 DP5 F

##### Función

Este es un módulo físico-abstracto que provee una interfaz para ejecutar comandos. Sus herederos implementarán diferentes comandos, los cuales serán invocados ante ciertas acciones de los dispositivos físicos y de otros módulos.

execute será responsable de ejecutar el correspondiente comando cuando sea invocado.

##### Secreto

Oculta los distintos comandos definidos, que permitirán sustituir la utilización *callbacks*.

#### 7.4.5. PassiveSensor

MI DP F

##### Función

Este es un módulo físico-abstracto que provee la interfaz para implementar un sensor pasivo. Los sensores pasivos serán aquellos que no envían señales al sistema de forma autónoma. Por el contrario, el sistema deberá solicitar de algún modo el valor medido por el sensor. Dichos sensores estarán conectados al controlador mediante conectores *Pipe*; y será a través de estos que enviará su información a dicho controlador.

setConnection, recibirá un conector *Pipe*, que mantendrá internamente.

signal será responsable de emitir el valor medido por el sensor.

##### Secreto

Oculta los posibles sensores, en los que los valores registrados no son emitidos por estos en forma autónoma.

#### 7.4.6. ACTIVE

##### Función

Este es un módulo lógico que agrupa los módulos que implementan dispositivos físico capaces de enviar interrupciones en forma independiente.

##### 7.4.6.1. ActiveSensor

MI DP1 DP2 F

##### Función

Este es un módulo físico-concreto que implementa un sensor Hall asociado a una rueda. Internamente está compuesto por un comando *Command* que será invocado cada vez que tenga lugar una señal física del sensor. En particular, el comando será *CountSignal*.

ActiveSensor recibirá un identificador que guardará internamente y que permitirá determinar el sensor Hall que este módulo representa.

start da inicio al funcionamiento del sensor físico mediante la interfaz provista por el dispositivo.

stop desactiva el sensor físico mediante la interfaz provista por el dispositivo.

setCommand, recibe como argumento un comando *Command* y lo mantiene internamente. Dicho comando será el que se ejecutará ante cada interrupción física emitida por el sensor. En particular, este comando será *CountSignal*.

signalHandler es un manejador de interrupciones y será invocado cuando tenga lugar una interrupción del sensor. Para esto, el método será registrado en el dispositivo físico mediante la interfaz que el mismo provea. El método simplemente invoca el comando correspondiente, mediante Command::execute.

##### Secreto

Oculta el funcionamiento de un sensor Hall.

##### 7.4.6.2. Timer

MI DP F

##### Función

Este es un módulo físico-abstracto que provee la interfaz de un temporizador. Esta interfaz define los métodos para iniciar y detener un reloj físico; establecer cada cuánto tiempo dicho reloj emitirá una señal (*tick*) y establecer cómo será manejada una interrupción, provocada por la mencionada señal.

setPeriod recibirá un valor real que determinará cada cuánto tiempo el reloj emitirá una señal.

start es responsable por dar inicio al reloj.

stop es responsable por desactivar el reloj.

tickHandler es responsable por manejar la ocurrencia de una interrupción física proveniente del reloj.

*Secreto*

Oculta distintos tipos de temporizadores que emiten señales físicas.

#### 7.4.6.3. FirstTimer

MI DP F

*Función*

Este es un módulo físico-concreto que implementa el temporizador principal del sistema. Este temporizador será el que maneje las interrupciones provenientes de un reloj físico principal, el cual emitirá interrupciones cada  $\Delta T$  ms a fin de llevar a cabo los ciclos de control de las ruedas y del dispositivo de dirección (ver requerimiento RF-47. - RF-48.). Cuando el reloj físico emite una señal, se produce una interrupción en el sistema; el manejador asociado a esta, definido en este módulo, invocará un comando **Command** que será el responsable de realizar las acciones correspondientes.

FirstTimer, siendo el constructor, recibe un comando **ControllerTimeOut** que será el que sea invocado cuando tenga lugar una interrupción originada por un *tick* del reloj.

setPeriod recibe un valor real que determina cada cuánto tiempo el reloj emitirá una señal. Internamente este método utiliza la interfaz correspondiente provista por el dispositivo físico para configurar este valor.

start inicia el reloj físico mediante la interfaz provista por el dispositivo.

stop desactiva el reloj físico mediante la interfaz provista por el dispositivo.

tickHandler es un manejador de señal que será invocado cada vez que tenga lugar una interrupción provocada por una señal del reloj. Para esto, el método será registrado en el dispositivo físico mediante la interfaz que este provea. Cuando este método es invocado, simplemente ejecuta el comando que fue registrado en el módulo, mediante una llamada a ControllerTimeOut::execute.

*Secreto*

Oculta cómo se configura, se activa y se desactiva un reloj físico; y las acciones a tomar ante la ocurrencia de una interrupción provocada por una señal del reloj.

#### 7.4.6.4. SecondTimer

MI DP1 DP2 DP3 F

*Función*

Este es un módulo físico-concreto que implementa el temporizador secundario del sistema. Este temporizador será el que maneje las interrupciones provenientes de un reloj físico secundario, el cual emitirá interrupciones cada  $\Delta t_q$  ms a fin de que en cada interrupción, se lean los valores de corriente registrados en cada rueda y se lleve a cabo un paso de giro en el dispositivo de dirección, si correspondiese (ver requerimiento RF-57. - RF-58.). Cuando el reloj físico emite una señal, se produce una interrupción en el sistema; el manejador asociado a esta, definido en este módulo, invocará comandos **Command** que serán los responsables de realizar las acciones correspondientes.

SecondTimer, siendo el constructor, recibe cuatro comandos **ReadCnt**, uno por cada rueda; y el comando **DirCtrlTimeOut** asociado al sistema de dirección.

setPeriod recibe un valor real que determina cada cuánto tiempo el reloj emitirá una señal. Internamente este método utiliza la interfaz correspondiente provista por el dispositivo físico para configurar este valor.

start inicia al temporizador mediante la interfaz provista por el dispositivo.

stop desactiva el temporizador mediante la interfaz provista por el dispositivo.

tickHandler es un manejador de señal que será invocado cada vez que tenga lugar una interrupción provocada por una señal del reloj. Para esto, el método será registrado en el dispositivo físico mediante la interfaz que este provea. Este método ejecuta, por cada rueda, el comando ReadCnt::execute, que lleva a cabo la lectura de la corriente de la misma. Además, invoca a DirCtrlTimeOut::execute para indicar, en el sistema dirección, que podría ser llevado a cabo un paso de giro.

```
cmdReadC1, cmdReadC1, cmdReadC1, cmdReadC1: ReadCnt
cmdDirTimeOut: DirCtrlTimeOut

tickHandler(){
    cmdReadC1.execute()
    cmdReadC2.execute()
    cmdReadC3.execute()
    cmdReadC4.execute()
    cmdDirTimeOut.execute()
}
```



### Secreto

Oculta cómo se configura, se activa y se desactiva un reloj físico; y las acciones a tomar ante la ocurrencia de una interrupción provocada por una señal del reloj.

## 7.4.7. PASSIVE

### Función

Este es un módulo lógico que agrupa los módulos que implementan dispositivos físicos pasivos; esto es, dispositivos que no emiten interrupciones.

### 7.4.7.1. Wheel

MI DP1 DP2 F

### Función

Este es un módulo físico-concreto que implementa una rueda. Provee los métodos para frenar una rueda, establecer una tensión determinada a la misma; o bien, indicar la orientación en la que debe desplazarse (adelante/atrás). Este módulo internamente mantiene un conjunto de datos **MapData** y una función **MapFunction** que utilizará para realizar los cálculos correspondientes a la tensión a aplicar a la rueda.

**Wheel**, siendo el constructor, recibe un identificador que permite determinar la rueda física que este módulo representará en el sistema. Con dicho identificador, el sistema accederá a la interfaz provista por el dispositivo físico. Este método es responsable de construir el conjunto de datos **MapData** y la función **MapFunction** que permitirán transformar un valor de tensión a aplicar a una rueda, en un valor de registro de 16bits. Ver requerimiento **RF-65**.

```
Wheel(Int id){  
    ident=id  
    sData=new SimpleData()  
    mData=new MapData(100,0,4200,0,sData)  
    mFunc=mapFunction()  
}
```

**setVoltage** recibe como argumento una tensión **Measure**, en particular un valor **Percentage**, lo transforma a un valor de registro y lo escribe en el registro correspondiente. Para esto, primero obtiene el valor mediante **Measure::value** y luego lo guarda en los datos, por medio de **MapData::setArg** pasándole como argumento el valor obtenido previamente. Luego aplica la función que lleva a cabo la transformación del valor, invocando **MapFunction::calculate**, pasándole como argumento los datos **MapData** del módulo. Finalmente, el método toma de los mencionados datos el valor de registro, por medio de **MapData::getResult**, y lo escribe en el registro correspondiente.

**setOrientation** recibe como argumento un sentido de orientación del movimiento **Measure**, en particular un valor **MovementSense**. De acuerdo a este valor determinará el movimiento hacia adelante o hacia atrás de la rueda. Para esto obtiene, del argumento recibido, el valor invocando **MovementSense::value**. Si dicho valor es 1, utilizará la interfaz provista por el dispositivo físico, para indicar el movimiento hacia adelante de la rueda. Por el contrario, si el valor es 0, indicará que dicho movimiento será hacia atrás.

**brake** envía la señal de frenado a la rueda, mediante la interfaz provista por el dispositivo.

### Secreto

Oculta cómo enviar órdenes a una rueda.

### 7.4.7.2. SteeringDevice

MI DP F

### Función

Este es un módulo físico-concreto que implementa el dispositivo de dirección *paso-a-paso* de las ruedas. Los métodos del módulo, llevarán a cabo su funcionalidad mediante la interfaz provista por el dispositivo físico. Para realizar el giro, es necesario enviar al dispositivo una señal binaria, en forma secuencial alternando sus valores. Esto es, 0101010101, representando 0 un valor bajo de tensión y 1 un valor alto de tensión. Ver requerimiento **RF-75**.

**SteeringDevice**, siendo el constructor, deberá iniciar una variable interna **signal** de señal, en 0 o 1, para poder generar la secuencia alternada de señales.

**enable** activa el dispositivo.

**disable** desactiva el dispositivo.

**left** establece que el giro a llevar a cabo será hacia la izquierda.

**right** establece que el giro a llevar a cabo será hacia la derecha.

pulse envía una señal al dispositivo para que de un paso de giro. El método debe determinar si la señal a enviar debe ser alta o baja, en función de la señal enviada previamente. Para esto, calcula  $\text{signal}=(\text{signal}+1) \bmod 2$ , lo que dará como resultado un 0 o un 1, dependiendo del valor anterior enviado. El método entonces, procede a enviar la tensión correspondiente.

*Secreto*

Oculto el funcionamiento del dispositivo físico de dirección que permite el giro de las ruedas.

### 7.4.7.3. DirSensor

MI DP F

*Función*

Este es un módulo físico-concreto que implementa un sensor de la posición del dispositivo de dirección. Dado que dicha posición es registrada como señales en **Pines** que conectan el dispositivo con el MCU, el módulo es responsable de acceder tales pines y enviar por una conexión **Pipe** la posición medida, para que otro módulo pueda acceder a la información.

DirSensor siendo el constructor, guarda internamente el valor de las constantes  $m$  y  $h$  definidas en el requerimiento **RF-69.1.**, que el módulo utilizará para realizar el cálculo del ángulo medido. Por un lado, obtiene y guarda el valor de  $m$  mediante Constants::getM; por el otro, define y guarda internamente el valor de  $h$  de acuerdo al requerimiento mencionado.

```
DirSensor(){
    const=Constants::instance()
    m = const.getM()
    h = 196
}
```

setConnection(i Pipe) establece cuál será la conexión **Pipe** por la cual el módulo enviará el valor medido.

signal lee la posición del dispositivo de dirección y la envía por la conexión; para esto, lee los 8 pines asociados a tal dispositivo. La secuencia de estos 8 valores determina un código Gray. El método entonces, transforma dicho código en un valor en el intervalo  $[-100, 100]$ , de acuerdo a lo especificado en el requerimiento **RF-69**. El valor obtenido, se establece como un porcentaje signado mediante SignedPerc::setValue, pasándolo como argumento. Este porcentaje, será la posición medida del dispositivo, que será escrita en la conexión por medio de Pipe::write, pasándole como argumento dicho porcentaje.

*Secreto*

Oculto cómo acceder a la posición medida del dispositivo de dirección.

## 7.5. CALCULATIONS

*Función*

Este es un módulo lógico que agrupa los módulos que permiten llevar a cabo los cálculos requeridos por el sistema. Esto es, unidades de medidas, datos y algoritmos.

### 7.5.1. MEASUREMENT UNITS

*Función*

Este es un módulo lógico que agrupa los módulos que implementan unidades de medida.

#### 7.5.1.1. Measure

MI DP F

*Función*

Este es un módulo concreto-abstracto que provee la interfaz para determinar y obtener un valor en una unidad de medida determinada.

setValue recibe un valor **Real** y es responsable por guardarlo internamente.

value devuelve el valor **Real** que mantiene internamente.

checkValue recibe un valor **Real** y determina si cumple con las condiciones correspondientes a la unidad de medida. En tal caso devolverá el valor de verdad **True**; en caso contrario, **false**.

*Secreto*

Oculto las representaciones de valores en distintas unidades de medida.

#### 7.5.1.2. Position

MI DP F

##### Función

Este es un módulo físico-concreto que implementa una posición, cuyos valores serán un entero no negativo. En particular, este módulo implementará la posición medida de una rueda.

setValue recibirá un valor **Real**, el cual controlará que sea un **entero no negativo**. En tal caso, guardará el valor. En caso contrario, tendrá lugar una excepción.

value devolverá el valor, previamente guardado mediante setValue.

checkValue recibirá un valor **Real** y determinará si se trata de un **entero no negativo**. En tal caso, devolverá verdadero (**True**); en caso contrario, devolverá falso (**False**).

##### Secreto

Oculto la representación interna de la posición de una rueda.

#### 7.5.1.3. Percentage

MI DP F

##### Función

Este es un módulo físico-concreto que implementa un porcentaje. En particular, será utilizado para representar la tensión a aplicar a una rueda.

setValue recibirá un valor **Real**, el cual controlará que pertenezca al **intervalo [0,100]**. En tal caso, guardará el valor. En caso contrario, tendrá lugar una excepción.

value devolverá el valor, previamente guardado mediante setValue.

checkValue recibirá un valor **Real** y determinará si se trata de un valor en el intervalo [0,100]. En tal caso, devolverá verdadero (**True**); en caso contrario, devolverá falso (**False**).

##### Secreto

Oculto la representación interna de valores porcentuales.

#### 7.5.1.4. RPM

MI DP F

##### Función

Este es un módulo físico-concreto que implementa un valor de velocidad en RPM (revoluciones por minuto). Será utilizado para representar la velocidad de una rueda.

setValue recibirá un valor **Real**, el cual controlará que pertenezca al **intervalo [0, MAXRPM]**; siendo *MAXRPM* el valor establecido en el requerimiento **RF-7**. En tal caso, guardará el valor. En caso contrario, tendrá lugar una excepción.

value devolverá el valor, previamente guardado mediante setValue.

checkValue recibirá un valor **Real** y determinará si se trata de un valor en el intervalo [0, *MAXRPM*]; siendo *MAXRPM* el valor establecido en el requerimiento **RF-7**. En tal caso, devolverá verdadero (**True**); en caso contrario, devolverá falso (**False**).

##### Secreto

Oculto la representación interna de valores de velocidad representados en RPM.

#### 7.5.1.5. SignedPerc

MI DP F

##### Función

Este es un módulo físico-concreto que implementa un porcentaje signado; esto es, un valor en el intervalo [-100,100]. En particular, será utilizado para representar la posición medida del dispositivo de dirección, la posición de referencia de este, la corriente medida o la corriente de referencia.

setValue recibirá un valor **Real**, el cual controlará que pertenezca al **intervalo [-100,100]**. En tal caso, guardará el valor. En caso contrario, tendrá lugar una excepción.

value devolverá el valor, previamente guardado mediante setValue.

checkValue recibirá un valor **Real** y determinará si se trata de un valor en el intervalo [-100, 100]. En tal caso, devolverá verdadero (**True**); en caso contrario, devolverá falso (**False**).

### 7.5.1.6. MovementSense

MI DP F

#### Función

Este es un módulo físico-concreto que implementa un sentido de movimiento binario; por ejemplo, hacia adelante o hacia atrás; o bien, hacia la derecha o hacia la izquierda. En el primer caso, será utilizado para representar el sentido de orientación (adelante/atrás) del movimiento de una rueda. En el segundo caso, se usará para representar el sentido de giro (derecha/izquierda) del dispositivo de dirección. Los valores posibles que mantendrá este módulo serán únicamente 0 o 1.

Se conviene que en el caso de tratarse del movimiento de las ruedas; el valor 0 significará el movimiento hacia atrás, mientras que el valor 1 significará el movimiento hacia adelante.

Por otra parte, se conviene que en el caso de tratarse del movimiento del dispositivo de dirección; el valor 0 significará la dirección de giro hacia la izquierda, mientras que el valor 1 significará la dirección de giro hacia la derecha.

setValue recibirá un valor **Real**, el cual controlará que **sea 0 o 1**. En tal caso, guardará el valor. En caso contrario, tendrá lugar una excepción.

value devolverá el valor, previamente guardado mediante setValue.

checkValue recibirá un valor **Real** y determinará si se trata del valor 0 o del valor 1. En el caso en que se trate de alguno de los valores, devolverá verdadero (**True**); en caso contrario, devolverá falso (**False**).

#### Secreto

Oculto la representación interna de valores que representan un sentido de movimiento (binario).

## 7.5.2. INFORMATION

#### Función

Este es un módulo lógico que agrupa los módulos que guardarán información necesaria para llevar a cabo los cálculos requeridos por el sistema.

### 7.5.2.1. Constants

MI DP F

#### Función

Este es un módulo físico-concreto que implementa un conjunto de valores constantes que serán requeridos por otros módulos. Todos los valores que se agrupan aquí son requeridos por más de un módulo. La implementación de este módulo será un *Singleton*; esto es, no podrán construirse más que un solo objeto de este. Es responsabilidad de este módulo construir una instancia única de sí mismo y por tanto **no provee en su interfaz un constructor**. Por el contrario, provee el método instance que permite obtener la instancia de este. Ver el patrón de diseño *Singleton* [ERRJ03].

instance evalúa si se ha creado una instancia de este módulo y en dicho caso la retorna; en caso contrario, la crea, la guarda y la devuelve.

```
instance(){
    if _instancia==NULL
        _instancia=new Constants()

    return _instancia
}
```

Constants, siendo el constructor, asigna y guarda los valores constantes de: el ancho del pulso del CR (*ANCHOPULSO\_CR*), el valor máximo de RPM asociado al CR (*MAXRPM\_CR*), el período de tiempo del ciclo de control principal ( $\Delta T$ ) y el valor constante  $m$  ( $M$ ) correspondiente al valor del giro mínimo y al cálculo del ángulo medido. Ver requerimientos RF-16., RF-18., RF-48., RF-69.1. y RF-73.1.

```
Constants(){
    ANCHOPULSO_CR=50
    MAXRPM_CR=50
    DELTA_T=100
    M=0,4
}
```

getPWM devuelve el valor de ancho de pulso del CR (*ANCHOPULSO\_CR*).

getMAXRPMCR devuelve valor máximo de RPM asociado al CR (*MAXRPM\_CR*).

getDELTAT devuelve período  $\Delta T$  de tiempo del ciclo de control principal ( $DELTA.T$ ).

getM devuelve el valor de  $M$  correspondiente a giro mínimo y al cálculo del ángulo medido.

*Secreto*

Oculto la representación de ciertos valores constantes del sistema.

### 7.5.2.2. MainCtrlData

MI DP F

*Función*

Este es un módulo físico-concreto que define una interfaz para poder guardar y recuperar la orientación (adelante/atrás) en la que se desplaza el robot. Esta información será utilizada por el controlador principal **MainController**.

saveOrientation, recibe como argumento la orientación **Measure** en la que se desplaza el robot y la guarda internamente. En particular, dicho valor será un elemento **MovementSense**.

getOrientation devuelve la orientación **Measure** en la que se desplaza el robot. En particular, dicho valor será un elemento **MovementSense**.

saveNewOrientation, recibe como argumento la nueva orientación de desplazamiento **Measure**, solicitada por una orden, y la guarda internamente. En particular, dicho valor será un elemento **MovementSense**.

getNewOrientation devuelve la nueva orientación de desplazamiento **Measure** requerida en una orden. En particular, dicho valor será un elemento **MovementSense**.

saveStopOrder recibe como argumento, y guarda internamente, un valor de verdad **Bool** indicando si ha habido una orden de frenado.

getStopOrder, devuelve un valor de verdad **Bool** indicando si ha habido una orden de frenado. Dicho valor será aquel que fuere registrado previamente por medio de saveStopOrder.

saveModeId, recibe como argumento, y guarda internamente, un número entero **Int** identificando el modo de operación del sistema (CR/PC).

getModeId devuelve un número entero **Int** que será el identificador del modo de operación en el cual esté funcionando el sistema. Dicho identificador será aquel que fuere guardado previamente por medio de saveModeId.

*Secreto*

Oculto la representación interna de cierta información utilizada por el controlador principal.

### 7.5.2.3. Data

MI DP F

*Función*

Este es un módulo físico-abstracto que define una interfaz para poder guardar y recuperar valores de referencia (*setpoints*), para llevar a cabo el control de las ruedas y del dispositivo de dirección.

saveSetpoint, recibe como argumento un valor **Measure** y lo guarda como *setpoint*.

getSetpoint devuelve el valor correspondiente al *setpoint* guardado.

*Secreto*

Oculto la representación interna de cierta información.

### 7.5.2.4. WheelCtrlData

MI DP F

*Función*

Este es un módulo físico-concreto que implementa un repositorio de información, asociada a un sistema de control de rueda.

saveSetpoint recibe como argumento un valor **Measure**, como *setpoint*, y lo guarda internamente. Este valor será un valor de referencia para el control de una rueda; en particular, será un valor de velocidad **RPM**, un valor de corriente **SignedPerc** o un valor de tensión **Percentage**.

getSetpoint devuelve el valor de referencia **Measure** guardado previamente por saveSetpoint.

saveOrientation recibe como argumento la orientación **Measure**, en la que debe desplazarse la rueda (adelante/atrás), y la guarda internamente. En particular, será un valor **MovementSense**.

getOrientation retorna la orientación **Measure**, en la que debe desplazarse la rueda (adelante/atrás). En particular, será el valor

MovementSense guardado previamente por saveOrientation.

saveVel recibe como argumento la velocidad medida **Measure** de la rueda y la guarda internamente. En particular, será un valor **RPM**.

getVel retorna la velocidad medida de la rueda, guardada previamente por saveVel.

savePosition recibe como argumento la posición **Measure** de la rueda y la guarda internamente. En particular, será un valor **Position**.

getPosition devuelve la posición **Measure** de la rueda, guardada previamente por savePosition.

velIsNull devuelve un valor booleano indicando si la velocidad de la rueda es nula. Para esto, el método accede a la velocidad medida, registrada previamente por saveVel.

saveCnt recibe como argumento la corriente **Measure** medida de una rueda. En particular, este valor será un **SignedPerc**.

getCnt retorna la corriente medida de una rueda, previamente guardada por saveCnt.

saveTens recibe como argumento la tensión **Measure** a aplicar a una rueda y la guarda internamente. En particular, dicha tensión será un valor **Percentage**.

getTens devuelve la tensión **Measure** a aplicar a una rueda, previamente guardada por el método saveTens.

addErrorVel recibe como argumento un valor real. Dicho valor será el error de velocidad calculado (la diferencia entre la velocidad de referencia y la velocidad medida). Este método sumará el valor recibido a una variable que guarda internamente; registrando así, el error de velocidad acumulado.

getErrorVel devuelve el error de velocidad acumulado, que ha sido registrado mediante el método addErrorVel.

resetErrorVel inicializa en 0 el valor del error de velocidad acumulado; esto es, asigna el valor 0 a la variable utilizada por addErrorVel.

addErrorCnt recibe como argumento un valor real. Dicho valor es el error de corriente calculado (diferencia entre el valor de referencia de corriente y la corriente medida). El método incrementará con el valor recibido, una variable que guarda internamente; registrando así, el error de corriente acumulado.

getErrorCnt devuelve el error de corriente acumulado, que ha sido registrado mediante addErrorCnt.

resetErrorCnt inicializa en 0 el valor del error de corriente acumulado; esto es, asigna el valor 0 a la variable utilizada por addErrorCnt.

### *Secreto*

Oculta la representación interna de información asociada a un sistema de control de rueda.

## 7.5.2.5. DirCtrlData



### *Función*

Este es un módulo físico-concreto que implementa un repositorio de información, asociada a un sistema de control de dirección.

saveSetpoint recibe como argumento un valor **Measure**, como *setpoint*, y lo guarda internamente. Dicho valor será un valor de referencia para el sistema de control de dirección; en particular, será un valor **SignedPerc**.

getSetpoint devuelve el valor de referencia **Measure** guardado previamente por saveSetpoint.

saveDirection recibe como argumento la dirección de giro **Measure**, que se debe llevar a cabo (izquierda/derecha), y la guarda internamente. En particular, será un elemento **MovementSense**.

getDirection devuelve la dirección de giro **Measure**, guardada previamente por saveDirection.

savePosition recibe como argumento la posición medida **Measure** del dispositivo de dirección y la guarda internamente. En particular, dicha posición será un valor **SignedPerc**. El valor guardado por este método se mantendrá durante todo el ciclo de control, ya que será necesario como dato para ser enviado a la PC. Esto es lo que lo diferencia del método saveTempPos.

getPosition devuelve la posición medida, previamente registrada por savePosition.

saveTempPos recibe como argumento la **posición temporal medida Measure** del dispositivo de dirección y la guarda internamente. En particular, dicha posición será un valor **SignedPerc**. Esta posición será la que el controlador del dispositivo de dirección irá registrando, en cada paso de giro, hasta alcanzar la posición deseada. Así, este método irá actualizando la posición medida inicialmente, durante el ciclo de control; esto es lo que lo diferencia del método savePosition.

getTempPosition devuelve la **posición temporal medida Measure** del dispositivo de dirección, registrada previamente por

saveTempPos.

saveErrorPos recibe como argumento un valor real. Dicho valor es el error de posición calculado (diferencia entre el valor de referencia de la posición y la posición medida). El método guarda en una variable interna el valor recibido. Notar que la posición medida, es aquella registrada por savePosition y no la posición **temporal** medida, registrada por saveTempPos.

getErrorPos devuelve el error de posición, registrado mediante el método saveErrorPos.

*Secreto*

Oculta la representación interna de información asociada a un sistema de control de dirección.

#### 7.5.2.6. CalculationData

MI DP F

*Función*

Este es un módulo físico-abstracto que define una interfaz que permite guardar y acceder, a un valor de entrada (argumento) y a un valor de salida (resultado) de una función.

setArg recibe como argumento un valor real y lo guarda internamente. Dicho valor será utilizado como argumento de una función.

getArg devuelve un valor real, aquel registrado previamente por el método setArg.

setResult recibe como argumento un valor real y lo guarda internamente. Dicho valor será el resultado de aplicar una función a cierto valor.

getResult devuelve el valor registrado por setResult.

getInMin devuelve el valor real correspondiente a  $in_{min}$  introducido en el requerimiento RF-13.

getInMax devuelve el valor real correspondiente a  $in_{max}$  introducido en el requerimiento RF-13.

getOutMin devuelve el valor real correspondiente a  $out_{min}$  introducido en el requerimiento RF-13.

getOutMax devuelve el valor real correspondiente a  $out_{max}$  introducido en el requerimiento RF-13.

getCalibration devuelve el valor real, correspondiente a un valor de la calibración.

setOrientation recibe y guarda internamente un valor real que representará una cierta orientación (adelante/atrás) a registrar.

getOrientation devuelve un valor real que representa cierto sentido de orientación registrado.

*Secreto*

Oculta distintos grupos de datos, requeridos para aplicar una función.

#### 7.5.2.7. SimpleData

MI DP F

*Función*

Este es un módulo físico-concreto que implementa el valor de entrada y el valor de salida de una función. Esto es; permite guardar y acceder, a un valor que será utilizado como argumento de una función **Function**, y a un valor que será el resultado de aplicar dicha función al mencionado argumento.

setArg recibe como argumento un valor real y lo guarda internamente. Dicho valor será utilizado como argumento de una función.

getArg devuelve un valor real, aquel registrado previamente por el método setArg.

setResult recibe como argumento un valor real y lo guarda internamente. Dicho valor será el resultado de aplicar una función a cierto valor.

getResult devuelve el valor registrado por setResult.

*Secreto*

Oculta la representación interna del argumento de una función y del resultado de esta.

#### 7.5.2.8. MoreData

MI DP F

*Función*

Este es un módulo físico-abstracto que hereda la interfaz de **CalculationData** y provee una interfaz que permite acceder a grupos de datos que son extensiones de datos más simples; por ejemplo extensiones de **SimpleData**. Este módulo define una variable

interna de tipo **CalculationData** de modo tal que sus herederos mantengan internamente un elemento de dicho tipo.

setArg recibe como argumento un valor real y lo guarda internamente. Dicho valor será utilizado como argumento de una función.

getArg devuelve un valor real, aquel registrado previamente por el método setArg.

setResult recibe como argumento un valor real y lo guarda internamente. Dicho valor será el resultado de aplicar una función a cierto valor.

getResult devuelve el valor resultante, registrado mediante setResult.

getInMin devuelve el valor real correspondiente a  $in_{min}$  introducido en el requerimiento **RF-13**.

getInMax devuelve el valor real correspondiente a  $in_{max}$  introducido en el requerimiento **RF-13**.

getOutMin devuelve el valor real correspondiente a  $out_{min}$  introducido en el requerimiento **RF-13**.

getOutMax devuelve el valor real correspondiente a  $out_{max}$  introducido en el requerimiento **RF-13**.

getCalibration devuelve el valor real, correspondiente a un valor de la calibración.

setOrientation recibe y guarda internamente un valor real que representará una cierta orientación (adelante/atrás) a registrar.

getOrientation devuelve un valor real que representa cierto sentido de orientación registrado.

*Secreto*

Oculta diferentes grupos de datos que son extensiones de datos más simples.

#### 7.5.2.9. MapData



*Función*

Este es un módulo físico-concreto que implementa los valores de entrada y el valor de salida de una función. Este módulo mantendrá internamente un elemento **CalculationData**; en particular, un **SimpleData**. Los datos que este módulo implementa serán utilizados por la función **MapFunction**. Ver requerimiento **RF-13**.

**MapData**, siendo el constructor, recibe como argumento cuatro valores reales y un grupo de datos **SimpleData**. Todos estos elementos son guardados internamente. Los cuatro valores reales son los argumentos  $in_{min}$ ,  $in_{max}$ ,  $out_{min}$  y  $out_{max}$  presentes en la definición de la función *map* descrita en el requerimiento **RF-13**. Esta función es utilizada por distintos módulos con distintos valores pasados como argumentos. Ver requerimientos **RF-22.4** y **RF-27.2**. Los datos **SimpleData** tendrán el valor del argumento  $x$  de la función y el resultado obtenido de aplicarla a dicho valor, junto con los valores mencionados más arriba.

getInMin devuelve el valor real correspondiente a  $in_{min}$  asignado en el constructor.

getInMax devuelve el valor real correspondiente a  $in_{max}$  asignado en el constructor.

getOutMin devuelve el valor real correspondiente a  $out_{min}$  asignado en el constructor.

getOutMax devuelve el valor real correspondiente a  $out_{max}$  asignado en el constructor.

setArg recibe como argumento un valor real, que será el argumento  $x$  de la función *map* implementada por **MapFunction**, y delega en **SimpleData** la tarea de guardarlo internamente. Para esto, el método invoca **SimpleData::setArg** pasándole como argumento el valor recibido.

getArg devuelve un valor real, aquel registrado previamente por el método setArg. Para esto, delega la tarea en **SimpleData** invocando **SimpleData::getArg** y retornando el valor obtenido.

setResult recibe como argumento un valor real y lo guarda internamente. El método delega en **SimpleData** dicha acción; esto es, invoca **SimpleData::setResult**, pasándole como argumento el valor recibido.

getResult devuelve el valor registrado por setResult. Para esto invoca **SimpleData::getResult** y devuelve el valor obtenido.

*Secreto*

Oculta la representación interna de los valores de entrada y salida de la función *map* definida en el requerimiento **RF-13**.

#### 7.5.2.10. CRData



*Función*

Este es un módulo físico-concreto que implementa los valores de entrada y el valor de salida de una función. Este módulo



mantendrá internamente un elemento `CalculationData`; en particular, un `MapData`. Los datos que este módulo implementa serán utilizados por las funciones `MapFunction` y `OrientationCalc`; ambas utilizadas por los buffers `CRBuffer` asociados a los pines de dirección y del velocidad del CR. Ver requerimientos `RF-22`. y `RF-27`.

`CRData`, siendo el constructor, recibe como argumento un valor real y un grupo de datos `MapData`. Todos estos elementos son guardados internamente. El valor real es el valor de calibración `DCALIBRATION` o `VCALIBRATION` del CR, descrito en los requerimientos `RF-19`. - `RF-22`. y `RF-25`. - `RF-27`. Los datos `MapData` tendrán los valores máximos y mínimos requeridos por la función `map` descritos en los requerimientos `RF-22`. y `RF-27`.

`getInMin` devuelve el valor real correspondiente a  $in_{min}$  guardado en el elemento `MapData`. Para esto, invoca `MapData::getInMin` y devuelve el valor retornado por este.

`getInMax` devuelve el valor real correspondiente a  $in_{max}$  guardado en el elemento `MapData`. Para esto, invoca `MapData::getInMax` y devuelve el valor retornado por este.

`getOutMin` devuelve el valor real correspondiente a  $out_{min}$  guardado en el elemento `MapData`. Para esto, invoca `MapData::getOutMin` y devuelve el valor retornado por este.

`getOutMax` devuelve el valor real correspondiente a  $out_{max}$  guardado en el elemento `MapData`. Para esto, invoca `MapData::getOutMax` y devuelve el valor retornado por este.

`setArg` recibe como argumento un valor real, que será el argumento  $x$  de la función `map` implementada por `MapFunction`, y delega en `MapData` la tarea de guardarlo internamente. Para esto, el método invoca `MapData::setArg` pasándole como argumento el valor recibido.

`getArg` devuelve un valor real, aquel registrado previamente por el método `setArg`. Para esto, delega la tarea en `MapData` invocando `MapData::getArg` y retornando el valor obtenido.

`setResult` recibe como argumento un valor real y lo guarda internamente. El método delega en `MapData` dicha acción; esto es, invoca `MapData::setResult`, pasándole como argumento el valor recibido.

`getResult` devuelve el valor registrado por `setResult`. Para esto invoca `MapData::getResult` y devuelve el valor obtenido.

`getCalibration` devuelve el valor real, correspondiente a la calibración del CR, asignado en el constructor.

`setOrientation` recibe un valor real; en particular, será 0 o 1, y lo guarda internamente. Este valor, indicará la orientación (adelante/atrás) requerida por una orden del CR. El valor 1 indicará que el movimiento ordenado es hacia adelante y el valor 0 indicará que es hacia atrás. Ver requerimiento `RF-22`.

`getOrientation` devuelve un valor real que es el sentido de orientación registrado mediante `setOrientation`.

#### *Secreto*

Oculta la representación interna los valores de entrada y salida de las funciones `MapFunction` y `OrientationCalc`.

### 7.5.3. OPERATIONS

---

#### *Función*

Este es un módulo lógico que agrupa los módulos que implementan los diferentes cálculos, funciones y algoritmos utilizados por el sistema.

#### 7.5.3.1. Algorithm

MI DP1 DP2 F

#### *Función*

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo los cálculos establecidos por cierto algoritmo. Cuenta con un único método `calculate` que recibe un repositorio de datos `Data`, para obtener los valores que serán utilizados en el cálculo y, para guardar los resultados del mismo.

Sus módulos herederos implementarán los diferentes algoritmos. Cada módulo heredero sabrá qué requiere del repositorio de datos `Data` para realizar su cálculo.

#### *Secreto*

Oculta la existencia de diferentes algoritmos.

### 7.5.3.2. TensAlgorithm

MI DP F

#### Función

Este es un módulo físico-concreto que implementa el algoritmo de control que permite calcular la tensión que debe aplicarse a una rueda, a partir de la tensión deseada recibida en una orden.

`calculate` recibe un repositorio de datos `Data` como argumento, en particular un elemento `WheelCtrlData`, y calcula la tensión que debe aplicarse a una rueda. Los cálculos son descriptos en el requerimiento RF-64. El método simplemente toma la tensión recibida como `setpoint` y la registra como la tensión a aplicar a la rueda. Esto lo hace invocando `WheelCtrlData::getSetpoint`, para obtener la tensión, y llamando a `WheelCtrlData::saveTens`, pasándole la tensión mencionada como argumento.

Este método, también es responsable de inicializar los errores de velocidad y corriente acumulados. Esto lo hace mediante `Data::resetErrorVel` y `Data::resetErrorCnt`. Ver requerimientos RF-54. y RF-61.

```
calculate(Data d){
    d.resetErrorVel()
    d.resetErrorCnt()
    sp=d.getSetpoint()
    d.saveTens(sp)
}
```

#### Secreto

Oculta cómo calcula la tensión a aplicar a una rueda, a partir de la tensión deseada recibida en una orden.

### 7.5.3.3. VelAlgorithm

MI DP F

#### Función

Este es un módulo físico-concreto que implementa el algoritmo de control que permite calcular la tensión que debe aplicarse a una rueda, a partir de la velocidad deseada y la velocidad medida.

`VelAlgorithm`, siendo el constructor, establece el valor de dos constantes utilizadas en el cálculo del método `calculate`. Las constantes son `TENSION_VEL1` y `TENSION_VEL2`, a las cuales se les asignará los valores correspondientes de acuerdo a lo descrito en el requerimiento RF-55.

`calculate` recibe un repositorio de datos `Data` como argumento, en particular un elemento `WheelCtrlData`, y calcula la tensión que debe aplicarse a una rueda. Para esto, obtiene primero la velocidad deseada mediante una llamada a `WheelCtrlData::getSetpoint`, y toma la velocidad medida invocando `WheelCtrlData::getVel`. A partir de estos valores obtiene el **error** calculando la diferencia entre ellos. Después, calcula el **error acumulado de velocidad** invocando `WheelCtrlData::addErrorVel`, pasándole como argumento el error calculado. Luego, obtiene el error acumulado mediante `WheelCtrlData::getErrorVel`. Con estos valores de error y los valores constantes definidos en el módulo, lleva a cabo el cálculo de la tensión a aplicar, de acuerdo a lo descrito en el requerimiento RF-56. El resultado de este cálculo es guardado en los datos mediante `WheelCtrlData::saveTens`, pasándoselo como argumento.

Este método, también es responsable de inicializar el error de corriente acumulado (ver requerimiento RF-61.). Esto lo hace por medio de `Data::resetErrorVel`.

```
calculate(Data d){
    d.resetErrorCnt()
    sp=d.getSetpoint()
    vel=d.getVel()
    error= diff(sp,vel)
    d.addErrorVel(error)
    errorAc=d.getErrorVel()
    volt=calcVoltios(error,errorAc)
    d.saveTens(volt)
}
```

#### Secreto

Oculta los cálculos requeridos para establecer la tensión de una rueda, necesaria para alcanzar cierta velocidad deseada, establecida a partir de una orden recibida. Además oculta cómo obtiene la información necesaria para llevar a cabo el cálculo y cómo registra los resultados de este.

### 7.5.3.4. CntAlgorithm

MI DP F

#### Función

Este es un módulo físico-concreto que implementa el algoritmo de control que permite calcular la tensión que debe aplicarse a una rueda, a partir de la corriente deseada y la corriente medida.

`CntAlgorithm`, siendo el constructor, establece el valor de dos constantes utilizadas en el cálculo del método `calculate`. Las constantes son `TENSION_CNT1` y `TENSION_CNT2`, a las cuales se les asignará los valores correspondientes de acuerdo a lo descrito en el requerimiento RF-62.

`calculate` recibe un repositorio de datos **Data** como argumento, en particular un elemento **WheelCtrlData**, y calcula la tensión que debe aplicarse a una rueda. Para esto, obtiene primero la corriente deseada mediante una llamada a `WheelCtrlData::getSetpoint`, y toma la corriente medida invocando `WheelCtrlData::getCnt`. A partir de estos valores obtiene el **error** calculando la diferencia entre ellos. Después, calcula el **error de corriente acumulado** invocando `WheelCtrlData::addErrorCnt`, pasándole como argumento el error calculado. Luego, obtiene el error acumulado invocando `WheelCtrlData::getErrorCnt`. Con estos valores de error y los valores constantes definidos en el módulo, lleva a cabo el cálculo de la tensión a aplicar, de acuerdo a lo descrito en el requerimiento **RF-63**. El resultado de este cálculo es guardado en los datos mediante `WheelCtrlData::saveTens`, pasándoselo como argumento.

Este método, también es responsable de inicializar el error de velocidad acumulado (ver requerimiento **RF-54**). Esto lo hace mediante `Data::resetErrorVel`.

```
calculate(Data d){
    d.resetErrorVel()
    sp=d.getSetpoint()
    cte=d.getCnt()
    error= diff(sp,cte)
    d.addErrorCnt(error)
    errorAc=d.getErrorCnt()
    volt=calcVoltios(error,errorAc)
    d.saveTens(volt)
}
```

#### Secreto

Oculto los cálculos requeridos para establecer la tensión de una rueda, necesaria para alcanzar una corriente deseada. Además oculta cómo obtiene la información necesaria para llevar a cabo el cálculo y cómo registra los resultados de este.

### 7.5.3.5. DirAlgorithm

**MI** **DP** **F**

#### Función

Este es un módulo físico-concreto que implementa el algoritmo que calcula el error entre la posición medida y la deseada; y en función de esto, el sentido del giro.

`calculate` recibe un repositorio de datos **Data**, en particular **DirCtrlData** y, a partir de este, obtiene la posición deseada mediante `DirCtrlData::getSetpoint`, y la posición medida por medio de `DirCtrlData::getPosition`. El método calcula el valor absoluto de la diferencia entre estos valores y lo guarda invocando `DirCtrlData::saveErrorPos`. Además, calcula el sentido de giro del dispositivo; esto lo hace del siguiente modo. Si la posición deseada es mayor que la medida, el sentido será hacia la derecha y por tanto se indicará el valor 1, pasándolo como argumento a `MovementSense::setValue` para ser registrado. Luego, dicho sentido de giro **MovementSense** será pasado como argumento a `DirCtrlData::saveDirection` para ser guardado. Si por el contrario la posición deseada es menor que la posición medida, el sentido de giro será hacia la izquierda y se indicará el valor 0, pasándolo como argumento a `MovementSense::setValue` que registrará la dirección deseada. Luego, mediante `DirCtrlData::saveDirection`, el método guardará el mencionado sentido de giro **MovementSense**. Ver requerimientos **RF-73.2**, **RF-73.4** y **RF-73.5**.

#### Secreto

Oculto cómo calcula el sentido del giro del dispositivo de dirección y cómo guarda el resultado.

### 7.5.3.6. Function

**MI** **DP** **F**

#### Función

Este es un módulo físico-abstracto que provee una interfaz para llevar a cabo los cálculos de una función. Cuenta con un único método `calculate` que recibe datos **CalculationData**, para obtener los valores que serán utilizados en el cálculo y, para guardar los resultados del mismo. Sus módulos herederos implementarán diferentes funciones. Cada módulo heredero sabrá qué requiere de los datos **CalculationData** para realizar su cálculo.

#### Secreto

Oculto la existencia de diferentes funciones.

### 7.5.3.7. OrientationCalc

**MI** **DP** **F**

#### Función

Este es un módulo físico-concreto que implementa una función que calcula el sentido de orientación (adelante/atrás) solicitado en una orden proveniente del CR.

`calculate` recibe datos **CalculationData**, en particular **CRData**. De estos datos obtiene el valor asignado al argumento invocando `CRData::getArg`. Si este valor es mayor o igual a 0, el sentido de orientación será hacia adelante y por tanto se indicará el valor 1, pasándolo como argumento a `CRData::setOrientation`. Si por el contrario el valor es menor que 0, el sentido de orientación será hacia atrás y por tanto se indicará el valor 0, pasándolo como argumento a `CRData::setOrientation`.

Esta función es utilizada por el buffer **VelBuffer** asociado al pin de velocidad del CR. Ver requerimiento **RF-22**.

*Secreto*

Oculta cómo realiza el cálculo de la orientación establecida en una orden de velocidad proveniente del CR.

#### 7.5.3.8. MapFunction

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa una función de mapeo de valores. Esta función es utilizada en algunos cálculos por los buffers **CRBuffer** del CR; y en cada rueda **Wheel** para transformar el valor de la tensión a aplicar, en un valor de registro de 16bits que será el que se le asigne realmente a la rueda.

calculate recibe datos **CalculationData**, en particular **CRData** o **MapData**. De estos datos obtiene el valor asignado al argumento mediante CalculationData::getArg y los valores máximos y mínimos invocando CalculationData::getInMin, CalculationData::getInOut, CalculationData::getInMax y CalculationData::getOutMax. A partir de estos valores, este método realiza el cálculo descrito en el requerimiento **RF-13**. Luego, el resultado obtenido es guardado mediante CalculationData::setResult.

*Secreto*

Oculta cómo realiza el cálculo de la función de mapeo descrita en el requerimiento **RF-13**.

#### 7.5.3.9. InverseFunction

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que implementa la función que transforma un valor de registro de 16bits en un valor real. Esta función es utilizada en el recolector de mediciones de corriente de una rueda **ValueColector**.

calculate recibe datos **CalculationData**, en particular un elemento **SimpleData** manipulado por el recolector de mediciones de corriente. De estos datos obtiene el valor asignado al argumento mediante SimpleData::getArg, calcula la traducción de este a un valor real de acuerdo a lo descrito en el requerimiento **RF-59.3**. Luego, el resultado obtenido es guardado mediante SimpleData::setResult.

*Secreto*

Oculta cómo realiza el cálculo de la función que traduce un entero de 16bits en un valor real.

## 7.6. MCU CONSTRUCTION

*Función*

Este es un módulo lógico que agrupa los módulos responsables por crear los objetos del sistema. En particular, el controlador principal **MainController**.

#### 7.6.1. MCDirector

**MI** **DP** **F**

*Función*

Este es un módulo físico-concreto que es responsable de dirigir la construcción del controlador principal; esto es, la construcción de un objeto **MainController**.

El módulo está compuesto por un elemento **MainControllerBuilder** en el que delega la construcción de las partes del objeto a construir. En su interfaz, este módulo provee un único método build que implementa.

MCDirector, siendo el constructor del módulo, recibe un elemento **MainControllerBuilder** y lo guarda internamente.

build dirige la construcción del controlador principal solicitando la construcción de órdenes para los sensores, de órdenes para los controladores, del grupo de sistemas de control, de órdenes del controlador principal y de estados de operación de este. Esto lo hace mediante las llamadas en forma secuencial de los siguientes métodos: MainControllerBuilder::buildSensorOrders, MainControllerBuilder::buildCtrlOrders, MainControllerBuilder::buildCtrlSysPool, MainControllerBuilder::buildMainOrders y MainControllerBuilder::buildOpStates. Luego de estas llamadas, termina.

*Secreto*

Oculta los pasos requeridos para construir el controlador principal del sistema.

#### 7.6.2. MainControllerBuilder

**MI** **DP** **F**

*Función*

Este es un módulo físico-abstracto que provee una interfaz para permitir la construcción de las partes que componen un con-

trolador principal **MainController**.

buildSensorOrders creará órdenes que se aplican sobre los sensores de los sistemas.

buildCtrlSysPool creará el conjunto de sistemas de control que constituyen el controlador principal.

buildCtrllersOrders creará órdenes que se aplican sobre los controladores de los sistemas.

buildMainOrders creará órdenes del controlador principal.

buildOpStates creará los estados de operación del controlador principal.

*Secreto*

Ocultas distintas maneras en que podrían construirse las partes de un sistema de control principal.

### 7.6.3. MCBUILDER



*Función*

Este es un módulo físico-concreto que implementa la construcción de las distintas partes que constituyen un sistema de control **MainController**. Implementa los métodos heredados de **MainControllerBuilder**.

MCBuilder, siendo el constructor de este módulo, recibe los posibles modos de operación (PC/CR) encapsulados en **ModePool**, los cuales utilizará en el proceso de construcción del controlador principal.

buildSensorOrders creará órdenes que se aplican sobre los sensores de corriente y velocidad de los sistemas.

```
buildSensorOrders(){
    wrVelOrd= new SensorWritesVel()
    wrCntOrd= new SensorWritesCnt()
    senWrOrd=new SensorsWrite(wrVelOrd,wrCntOrd)
}
```

buildCtrlSysPool creará el conjunto de sistemas de control que constituyen el controlador principal. Primero, utiliza las órdenes construidas por buildSensorOrders y aquellas construidas por buildCtrllersOrders para crear el constructor **CSPBuilder** y el director **CSPDirector**. A partir de estos, solicita la construcción del grupo de sistemas de control mediante **CSPDirector::build**. Luego, obtiene el temporizador secundario **SecondTimer** invocando **CSPBuilder::getLocalTimer** y lo guarda internamente; obtiene el grupo de sistemas de control creado, mediante **CSPBuilder::getCSP** y lo guarda; y obtiene, mediante **CSPBuilder::getDStoppingState**, el estado de operación **DStopping** del sistema de dirección y lo guarda.

```
wrVelOrd: SensorWritesVel
rVelOrd: ControllerReadsVel

buildCtrlSysPool(){
    cspBuilder= new CSPBuilder(wrVelOrd, rVelOrd)
    cspDirector=new CSPDirector(cspBuilder)
    cspDirector.build()
    timer15=cspBuilder.getLocalTimer()
    csp=cspBuilder.getCSP()
    dStopSt=cspBuilder.getDStoppingState()
}
```

buildCtrllersOrders creará órdenes que se aplican sobre los controladores de los sistemas, para que estos lean los valores de los sensores y lleven a cabo la tarea de control.

```
buildCtrllersOrders(){
    rVelOrd=new ControllerReadsVel()
    rCntOrd=new ControllerReadsCnt()
    ctrlsROrd=new ControllersRead(rVelOrd,rCntOrd)
    ctrlsCtrl=new ControllersControl()
}
```

buildMainOrders creará órdenes del controlador principal. Primero, utilizando el estado de operación **DStopping** obtenido en el método buildCtrlSysPool, crea una orden **UpdateOrder** de actualización del estado de operación del sistema de dirección. Luego, crea la orden **SaveWheelPositions** que permite registrar la posición de las ruedas; para esto crea previamente la orden **SaveWPosition** sobre una rueda. A partir de estas órdenes, de algunas creadas en el método buildSensorOrders y de otras creadas en el método buildCtrllersOrder, crea una orden **MainCtrlOrder** del controlador principal. Finalmente, crea una orden de parada de todos los sistemas de control. Para esto, crea el algoritmo **ResetVel**. Utilizando este y el estado de operación **DStopping**, crea la orden **ControllersStop** y la guarda.

```

dStopSt: DStopping
senWrOrd: SensorsWrite
ctrlsROrd: ControllersRead
ctrlsCtrl: ControllersControl

buildMainOrders(){
    updOrd=new UpdateOrder(dStopSt)
    saveWP=new SaveWPosition()
    saveWPoss=new SaveWheelPositions(saveWP)
    mCtrlOrd=new MainCtrlOrder(senWrOrd, ctrlsROrd, ctrlsCtrl,saveWPoss,updOrd)
    ----orden de parada----
    algResetVel=new ResetVel()
    ctrlsStopOrd=new ControllersStop(algResetVel, dStopSt)
}

```

buildOpStates creará los estados de operación del controlador principal. Estos serán: el estado de funcionamiento correcto **Working**, los  $n$  estados de espera **WaitingN**, el último estado de espera **WaitingMAX** y el estado de reconexión **Reconnecting**. Los estados serán los responsables de establecer cuál es el nuevo estado que los sucede de acuerdo a ciertas circunstancias (ver Figura 2.8). Por tanto, deberán recibir en su constructor los correspondientes estados sucesores; lo cual determina el orden de construcción de los mismos. Además, los constructores de los estados recibirán las correspondientes órdenes que se llevarán a cabo en ellos.

El método tendrá un valor constante  $MAX$  que será el máximo tiempo de espera ante una pérdida de señal, antes de pasar al estado de reconexión. Este tiempo será la cantidad máxima de ciclos de control que el sistema esperará; esto es  $MAX = 50$ . El origen de este valor se debe a que el requerimiento **RF-48**. establece que los ciclos duran ciertos  $n$  milisegundos, y el requerimiento **RF-34**. indica que el tiempo de espera es una cantidad  $m$  de segundos. Por tanto, el valor constante  $MAX$  será  $m * 1000/n$ . Este valor constante será utilizado en un bucle para construir los  $MAX - 1$  estados **WaitingN**.

```

mCtrlOrd: MainCtrlOrder
ctrlsStopOrd: ControllersStop

buildOpStates(){
    MAX=50
    workSt=new Working(mCtrlOrd)
    recSt=new Reconnecting(workSt, mCtrlOrd)
    waitSt=new WaitingMAX(workSt,recSt,ctrlsStopOrd,mCtrlOrd)

    nMax=MAX-1
    while nMax > 0
        temp=new WaitingN(workSt,waitSt,mCtrlOrd)
        waitSt=temp
        nMax--

    workSt.setNextState(waitSt)
    opState=workSt
}

```

getLocalTimer devuelve un temporizador **Timer**; en particular, el temporizador secundario **SecondTimer** obtenido en el método buildCtrlSysPool.

getMC finaliza la construcción del controlador principal **MainController** y lo devuelve. Para esto utiliza **ModePool** recibido en el constructor de este módulo, el conjunto de sistemas de control **ControlSystemPool** creado por buildCtrlSysPool y el estado de operación **Working** construido por builOpStates.

```

mdPool: ModePool
csp: ControlSystemPool
opState: Working

getMC(){
    md=mdPool.getCRMode()
    mCtrl=new MainController(csp)
    mCtrl.changeMode(md)
    mCtrl.changeState(opState)
    return mCtrl
}

```

getActiveSensors solicita al grupo de sistemas de control, mediante ControlSystemPool::getActiveSensors, un iterador de los sensores hall **ActiveSensor** de los sistemas de ruedas, y lo retorna.

*Secreto*

Oculta cómo se construye cada parte del controlador principal y cómo se combinan las partes resultantes para crear el mencio-

nado controlador.

## 7.6.4. CSPCONSTRUCTION

### Función

Este es un módulo lógico que agrupa los módulos responsables de construir un conjunto de sistemas de control **ControlSystemPool**.

### 7.6.4.1. CSPDirector

**MI** **DP** **F**

### Función

Este es un módulo físico-concreto que es responsable de dirigir la construcción del conjunto de sistemas de control. Este módulo delega en un módulo **CtrlSysPoolBuilder** la construcción de las partes del objeto a construir.

CSPDirector, siendo el constructor de este módulo, recibe y mantiene internamente un constructor **CtrlSysPoolBuilder** del conjunto de sistemas de control.

build dirige la construcción del conjunto de sistemas de control solicitando la construcción los sistemas de control de ruedas y la construcción del sistema de control de dirección. Esto lo hace mediante las llamadas en forma secuencial a los siguientes métodos: CtrlSysPoolBuilder::buildWheelSystem y CtrlSysPoolBuilder::buildDirSystem. Luego de estas llamadas, termina.

### Secreto

Oculta los pasos requeridos para construir el conjunto de sistemas de control.

### 7.6.4.2. CtrlSysPoolBuilder

**MI** **DP** **F**

### Función

Este es un módulo físico-abstracto que provee una interfaz para permitir la construcción de las partes que componen un conjunto de sistemas de control **ControlSystemPool**.

buildWheelSystem creará los correspondientes sistemas de control de ruedas que constituyen el sistema.

buildDirSystem creará el sistema de control de dirección.

### Secreto

Oculta las distintas maneras en que podrían construirse las partes que constituyen el conjunto de sistemas de control.

### 7.6.4.3. CSPBuilder

**MI** **DP** **F**

### Función

Este es un módulo físico-concreto que implementa la construcción de las distintas partes que conforman un conjunto de sistemas de control **ControlSystemPool**. Implementa los métodos heredados de **CtrlSysPoolBuilder**.

CSPBuilder, siendo el constructor de este módulo, recibe como argumentos una orden **SensorWritesVel** y una orden **ControllerReadsVel**. Estas órdenes serán utilizadas para indicar a un sensor de velocidad que emita el valor medido y a un controlador que lea dicho valor, en un sistema de control de rueda.

buildWheelSystem construye los sistemas de control utilizando las órdenes recibidas en el constructor. Primero, crea un objeto constructor **WheelSysBuilder**. Luego, pasando este último como argumento, crea un objeto director **WSDirector** y le indica que lleve a cabo la construcción del sistema de control de ruedas; esto lo hace, mediante WSDirector::build indicándole un número identificador de la rueda. Después, obtiene y guarda internamente: el sistema de control de la rueda, mediante WheelSysBuilder::getSystem, el comando **ReadCnt** asociado a la lectura de la corriente de la rueda, mediante WheelSysBuilder::getCntCmd y el sensor hall **ActiveSensor** de la rueda. Estos cuatro últimos pasos los llevará a cabo para cada una de las ruedas.

```
sensWrVel: SensorWritesVel
ctrllerRdVel: ControllerReadsVel

buildWheelSystem(){
    wsBuilder= new WSBuilder(sensWrVel, ctrllerRdVel)
    wsDirector=new WSDirector(wsBuilder)
    //-----Rueda DD -----
    wsDirector.build(id1)
    wSysDD=wsBuilder.getSystem()
    cntCmdDD=wsBuilder.getCntCmd()
    hallSensorDD=wsBuilder.getActiveSensor()
```



```

//-----Rueda DI -----
    wDirector.build(id2)
    wSysDI=wsBuilder.getSystem()
    ...
//-----Rueda TD -----
    wSysTD...
    ...
//-----Rueda TI -----
    wSysTI...
    ...
}

```

buildDirSystem construye el sistema de control de dirección del siguiente modo. Primero, crea un objeto constructor **DirSysBuilder**. Luego, pasando este último como argumento, crea un objeto director **DSDirector** y le indica que lleve a cabo la construcción del sistema de control de dirección; esto lo hace, mediante DSDirector::build. Después, a partir de **DirSysBuilder** obtiene y guarda internamente los siguientes elementos: el comando **DirCtrlTimeOut** asociado al temporizador secundario, mediante DirSysBuilder::getDirCmd; el estado activo del sistema de dirección **DActive**, mediante DirSysBuilder::getDActiveState; el estado de parada **DStopping**, mediante DirSysBuilder::getDStoppingState y el sistema de control de dirección **DirSystem**, por medio de DirSysBuilder::getSystem.

```

buildDirSystem(){
    dsBuilder= new DSBuilder()
    dsDirector=new DSDirector(dsBuilder)
    dsDirector.build()
    dCmdTimeOut=dsBuilder.getDirCmd()
    dActSt=dsBuilder.getDActiveState()
    dStopSt=dsBuilder.getDStoppingState()
    dSys=dsBuilder.getSystem()
}

```

getLocalTimer construye y devuelve el temporizado secundario **SecondTimer**. Para esto, invoca al constructor del mismo pasándole como argumento: los cuatro comandos **ReadCnt** asociados a la corriente de cada rueda, obtenidos en buildWheelSystem; y el comando **DirCtrlTimeOut** obtenido en buildDirSystem.

getCSP construye y devuelve el conjunto de sistemas de control **ControlSystemPool**. Para esto, invoca al constructor del mismo pasándole como argumento: los cuatro sistemas de control de ruedas **WheelSystem**, obtenidos en buildWheelSystem; el sistema de dirección **DirSystem**, obtenido en buildDirSystem; los estados de operación **DActive** y **DStopping**, ambos obtenidos en buildDirSystem.

getActiveSensors agrupa los cuatro sensores hall **ActiveSensor** de las ruedas en una estructura y devuelve el iterador de dicha estructura que permite obtener cada sensor. Dichos sensores son obtenidos en el método buildWheelSystem.

getDStoppingState devuelve el estado de operación **DStopping** del sistema de dirección, obtenido en el método buildDirSystem.

*Secreto*

Oculto cómo se construye cada parte del conjunto de sistemas de control.

#### 7.6.4.4. WSCONSTRUCTION

*Función*

Módulo lógico que agrupa los módulos responsables por la construcción de un sistema de control de rueda.

##### 7.6.4.4.1 WSDirector



*Función*

Módulo físico-concreto, responsable de dirigir la construcción de un sistema de control de rueda **WheelSystem**. Este módulo delega en el módulo **WheelSysBuilder** la construcción de las partes del sistema de control de rueda.

WSDirector, siendo el constructor de este módulo, recibe como argumento un módulo constructor **WheelSysBuilder** y lo guarda internamente.

build recibe el identificar de una cierta rueda y dirige la construcción del sistema de control para la misma. El método solicita la construcción ordenada de los siguientes elementos: de los datos internos del sistema de control de ruedas **WheelCtrlData**, invocando WheelSysBuilder::buildData; del controlador **WheelController**, mediante una llamada a WheelSysBuilder::buildController; de los sensores, invocando WheelSysBuilder::buildSensors y pasándole como argumento el identificador recibido; del sistema de control **WheelSystem**, llamando a WheelSysBuilder::buildWSys; del grupo de comandos **CtrlCmdPool**, mediante la invocación de WheelSysBuilder::buildCmdPool, pasándole como argumento a este último el identificador de la rueda; y de las



conexiones entre los sensores y el controlador, invocando WheelSysBuilder::buildConnections.

```
build(Int id){
    wsBuilder.buildData()
    wsBuilder.buildController()
    wsBuilder.buildSensors(id)
    wsBuilder.buildWSystem()
    wsBuilder.buildCmdPool(id)
    wsBuilder.buildConnections()
}
```

*Secreto*

Oculta los pasos requeridos para construir un sistema de control de rueda.

#### 7.6.4.4.2 WheelSysBuilder

MI DP F

*Función*

Este es un módulo físico-abstracto que provee una interfaz para permitir la construcción de las partes que componen un sistema de control de rueda **WheelSystem**.

buildData será responsable de construir los datos internos de un sistema de control de rueda.

buildController será responsable de construir el controlador del sistema de control de una rueda.

buildSensors recibirá un entero que permitirá identificar la rueda a controlar y construirá los sensores del sistema de control de la rueda.

buildWSystem será responsable de construir un sistema de control de una rueda.

buildCmdPool recibirá como argumento un entero que será el identificador de una rueda y construirá los comandos que permitirán el control de la misma.

buildConnections será responsable de crear las conexiones entre los sensores y el controlador de un sistema de control de rueda.

*Secreto*

Oculta las distintas formas en que podrían construirse las partes de un sistema de control de rueda.

#### 7.6.4.4.3 WSBuilder

MI DP1 DP2 DP3 DP4 F

*Función*

Este es un módulo físico-concreto que implementa la construcción de las distintas partes que conforman un sistema de control de rueda **WheelSystem**. Implementa los métodos heredados de **WheelSysBuilder**.

WSBuilder, siendo el constructor, recibe como argumento las órdenes **SensorWritesVel** y **ControllerReadsVel** y las guarda internamente.

buildData crea y guarda internamente un elemento **WheelCtrlData**.

buildController crea un controlador **WheelController**, pasándole al constructor del mismo los datos **WheelCtrlData** creados por el método buildData.

buildSensors recibe un entero que será utilizado como identificador de una rueda y crea los sensores asociados a esta. El método crea el sensor de corriente de la rueda, y el comando asociado; y el sensor de velocidad, junto con el correspondiente comando. Para crear el sensor de corriente, primero crea un recolector de valores **ValueCollector** pasándole como argumento el identificador de la rueda. Luego crea un sensor de corriente **CntSensor** pasándole como argumento el mencionado recolector. Finalmente, construye el comando para leer la corriente de la rueda **ReadCnt**, pasándole a su constructor el recolector construido previamente.

Para crear el sensor de velocidad, el método primero construye un recolector de señales **SensorCollector**. Luego crea el sensor **VelSensor** pasándole como argumento a su constructor, el recolector mencionado. Después crea el comando **CountSignal** pasándole también a su constructor el recolector creado. Finalmente, construye un sensor hall **ActiveSensor** pasándole al constructor el identificador de la rueda; y registra en dicho sensor el comando previamente construido, pasándolo como argumento a ActiveSensor::setCommand.

```

buildSensors(Int id){
//---sensor de corriente---
    cColl= new ValueCollector(id)
    cSensor=new CntSensor(cColl)
    cCmd=new ReadCnt(cColl)
//---sensor de velocidad---
    tColl= new TimeCollector()
    vColl= new SensorCollector(tColl)
    vSensor= new VelSensor(vColl)
    vCmd= new CountSignal(vColl)
    hallSensor= new ActiveSensor(id)
    hallSensor.setCommand(vCmd)
}

```

buildWSystem crea un sistema de control de rueda **WheelSystem** y lo guarda internamente. Para esto, le pasa como argumento al constructor del mismo, el sensor de velocidad **VelSensor** y el sensor de corriente **CntSensor** creados por el método buildSensors.

buildCmdPool recibe un valor entero que permitirá identificar la rueda a controlar y crea el grupo de comandos que actuarán sobre la misma y permitirán su control.

Primero, el método construye la rueda **Wheel** pasándole a su constructor el identificador de la misma. Luego, crea el comando de frenado **Brake** y el comando de velocidad nula **VelNull**, pasándoles a sus constructores la rueda creada. Después, crea el comando **SetTension** que permite establecer la tensión requerida a la rueda. El constructor de este comando recibirá la rueda creada, los datos **WheelCtrlData** creados por buildData y el controlador **WheelController** construido por buildController. El último comando a construir por el método es **ChangeOrientation** que cambia el sentido (adelante/atrás) del movimiento de las ruedas. El constructor de este comando recibirá la rueda **Wheel** y los datos **WheelCtrlData** construidos, el sistema **WheelSystem** creado por buildWSystem y las órdenes **SensorWritesVel** y **ControllerReadsVel** recibidos en el constructor de este módulo. Una vez construidos los cuatro comandos enumerados, el método crea y guarda un contenedor de comandos **CtrlCmdPool**, pasándole a su constructor los comandos mencionados.

```

data: WheelCtrlData
wCtrl: WheelController
wSys: WheelSystem
senWrVel: SensorWritesVel
ctrllderRdVel: ControllerReadsVel

buildCmdPool(Int id){
    wheel= new Wheel(id)
    cmdBrake=new Brake(wheel)
    cmdVelNull= new VelNull(wheel)
    cmdSetTens=new SetTension(wheel, data, wCtrl)
    cmdChangeO=new ChangeOrientation(wheel, data, wSys, senWrVel, ctrllderRdVel)
    cmdPool=new CtrlCmdPool(cmdVelNull, cmdBrake, cmdSetTens, cmdChangeO)
}

```

buildConnections determina las conexiones entre los sensores creados por buildSensors y el controlador creado por buildController. Primero, crea un conector **Pipe** y lo establece como conexión entre el sensor de velocidad y el controlador. Para esto utiliza los métodos VelSensor::setConnection y WheelController::setConnectionV, pasándoles a ambos como argumento el conector creado. Luego, crea otro conector **Pipe** y lo establece como conexión entre el sensor de corriente y el controlador. Esto lo hace mediante las llamadas a CntSensor::setConnection y WheelController::setConnectionC pasándoles como argumento el segundo conector creado.

```

vSensor: VelSensor
wCtrllder: WheelController

buildConnections(){
    pipeV= new Pipe()
    vSensor.setConnection(pipeV)
    wCtrllder.setConnectionV(pipeV)
    pipeC= new Pipe()
    cSensor.setConnection(pipeC)
    wCtrllder.setConnectionC(pipeC)
}

```

getSystem establece en el sistema **WheelSystem**, creado por buildWSystem, el controlador que le corresponde; y retorna el mencionado sistema. Para esto, invoca WheelSystem::setController, pasándole como argumento el controlador **WheelController** creado por buildController. Luego el método devuelve el sistema mencionado.

getCntCmd devuelve el comando **ReadCnt** de lectura de corriente de la rueda, creado por buildSensors.

getActiveSensor devuelve el sensor hall **ActiveSensor** de la rueda, creado por buildSensors.

*Secreto*

Oculta cómo se construyen y se ensamblan las partes de un sistema de control de rueda.

#### 7.6.4.5. DSCONSTRUCTION

*Función*

Módulo lógico que agrupa los módulos responsables por la construcción del sistema de control de dirección.

##### 7.6.4.5.1 DSDirector

MI DP F

*Función*

Módulo físico-concreto, responsable de dirigir la construcción de un sistema de control de dirección **DirSystem**. Este módulo delega en el módulo **DirSysBuilder** la construcción de las partes del sistema de control de dirección.

**DSDirector**, siendo el constructor de este módulo, recibe como argumento un módulo constructor **DirSysBuilder** y lo guarda internamente.

**build** dirige la construcción del sistema de control de dirección. El método solicita la construcción ordenada de los siguientes elementos: del controlador del sistema invocando **DirSysBuilder::buildController**; del sensor del dispositivo, mediante una llamada a **DirSysBuilder::buildSensor**; y de la conexión entre el controlador y el sensor, por medio de una invocación a **DirSysBuilder::buildConnection**.

*Secreto*

Oculta los pasos requeridos para construir el sistema de control de dirección.

##### 7.6.4.5.2 DirSysBuilder

MI DP F

*Función*

Este es un módulo físico-abstracto que provee una interfaz para permitir la construcción de las partes que componen el sistema de control de dirección **DirSystem**.

**buildController** construirá el controlador asociado al dispositivo de dirección.

**buildSensor** construirá el sensor correspondiente al dispositivo de dirección.

**buildConnection** construirá la conexión entre el sensor y el controlador del dispositivo de dirección.

*Secreto*

Oculta las distintas formas en que podrían construirse las partes de un sistema de control de dirección.

##### 7.6.4.5.3 DSBuilder

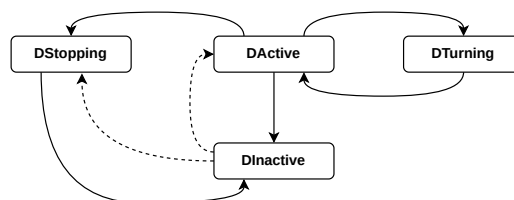
MI DP1 DP2 DP3 F

*Función*

Este es un módulo físico-concreto que implementa la construcción de las distintas partes que conforman un sistema de control de dirección **DirSystem**. Implementa los métodos heredados de **DirSysBuilder**.

**buildController** construye el controlador del sistema de dirección **DirController**. Para esto, previamente crea: el dispositivo de dirección **SteeringDevice**; los datos internos **DirCtrlData**; los comandos **Turn**, **SetDirection**, **Enable** y **Disable** que utiliza el controlador, los estados de operación **DInactive**, **DStopping**, **DTurning** y **DInactive** de este; y los estados de funcionamiento del dispositivo **OnState** y **OffState**.

Los estados de funcionamiento y los estados de operación, son los responsables por establecer el siguiente estado de una transición. En algunos casos, la transición de estado es establecida desde el exterior por otros módulos. Las posibles transiciones de estados de operación se presentan en la siguiente figura. Las líneas sólidas indican que en una transición, el estado origen conoce cuál es el estado destino. Las transiciones con líneas de punto indican que el estado origen desconoce el estado destino y que dicha transición es determinada externamente por otros módulos que no son los estados.



Por tanto, hay una dependencia entre ellos que requiere una construcción en un orden determinado de modo tal que las transiciones de línea continua puedan ser definidas. Lo mismo ocurre con los estados de funcionamiento del dispositivo de dirección.

```
buildController(){
    std=new SteeringDevice()
    data=new DirCtrlData()
    //--crear comandos--
    cmdTurn=new Turn(std)
    cmdSetDir=new SetDirection(std,data)
    cmdEnable=new Enable(std)
    cmdDisable=new Disable(std)
    //--crear estados de op---\\
    inactSt=new DInactive()\\
    stopSt=new DStopping(inactSt)\\
    turnSt=new DTurning(cmdTurn)\\
    actSt=new DActive(turnSt,stopSt,inactSt,cmdSetDir)\\
    turnSt.setNextState(actSt)\\
    //--crear estados on/off---
    offSt=new OffState(cmdEnable)
    onSt=new OnState(cmdDisable,offSt)
    offSt.setNextState(onSt)

    //--crear controlador---
    dCtrl=new DirController(data)
    dCtrl.changeDvState(offSt)
    dCtrl.changeOpState(inactSt)
}
```

buildSensor crea un sensor de dirección **DirSensor** y lo guarda internamente.

buildConnection determina la conexión entre el sensor y el controlador. Primero, crea un conector **Pipe**. Luego utiliza los métodos **DirSensor::setConnection** y **DirController::setConnection**, pasándoles como argumento el conector creado, para establecer la conexión entre el sensor y el controlador. Habiendo sido estos últimos, creados previamente por los métodos buildSensor y buildController.

getSystem construye un sistema de control de dirección **DirSystem**, pasándole a su constructor el controlador construido por buildController y el sensor creado por buildSensor. Luego, devuelve el elemento **DirSystem** creado.

getDirCmd crea y retorna el comando **DirCtrlTimeOut**, pasándole a su constructor el controlador **DirController** construido por buildController y el sensor **DirSensor** creado por buildSensor.

getDActiveState retorna el estado **DActive** creado por buildController.

getDStoppingState retorna el estado **DStopping** creado por buildController.

*Secreto*

Oculta cómo se construyen y se combinan las partes de un sistema de control de dirección.

## 7.7. Main

**MI** **DP1** **DP2** **F**

*Función*

Este módulo representa el programa principal que proveerá el funcionamiento del MCU del robot desmalezador. La función main describirá la funcionalidad necesaria que se debe llevar a cabo para que el sistema inicie su funcionamiento.

main ordena la construcción de los elementos del sistema y da inicio a la tarea de control que debe realizar el MCU. El método ordenará la construcción de: los pines **Pin** y los buffers **CRBuffer** de velocidad y de dirección, del CR; el escritor **SerialWriter** que permitirá enviar información a la PC; los lectores **SerialReader** y **BufferReader** por medio de los cuales se leerán las órdenes a llevar a cabo; los modos de operación (CR/PC) del sistema; el controlador principal **MainController** y; los temporizadores **FirstTimer** y **SecondTimer** del sistema. Luego, el método iniciará los sensores hall y los temporizadores; e ingresará en un estado de espera por las interrupciones originadas por los distintos actores que constituyen el sistema de control.

```

main(){
-----pin de CR de velocidad-----
    vPinColl=new CRpinCollector()
    vCmdCountT=new CountTime(vPinColl)
    vpin=new Pin(id1)
    vpin.setCommand(vCmdCountT)
    vBuffer=new CRBuffer(vPinColl)
-----pin de CR de dirección-----
    dPinColl=new CRpinCollector()
    dCmdCountT=new CountTime(dPinColl)
    dpin=new Pin(id2)
    dpin.setCommand(dCmdCountT)
    dBuffer=new CRBuffer(dPinColl)
-----Escribiente-----
    mcuToPc=new ConnectMCUtoPC
    serialWr=new SerialWriter(mcuToPC)
-----Lectores-----
    tensAlg=new TensAlgorithm()
    velAlg=new VelAlgorithm()
    cntAlg=new CntAlgorithm()
    bufferRdr=new BufferedReader(vBuffer,dBuffer,velAlg)
    transmittingSt=new Transmitting()
    noMessageSt=new NoMessage(transmittingSt)
    readyMessageSt=new ReadyMessage(noMessageSt,transmittingSt)
    transmittingSt.setNextState(readyMessageSt)
    pcToMCU=new ConnectPCtoMCU()
    pcToMCU.changeState(noMessageSt)
    serialRdr=new SerialReader(pcToMCU,tensAlg,velAlg,cntAlg)
-----modos de operación-----
    bMdCR=new BasicMode(bufferRdr)
    cr=new CR(bMdCR)
    bMdPC=new BasicMode(serialRdr)
    pc=new PC(bMdPC, cr)
    cr.setNextMode(pc)
    mdPool=new ModePool(pc,cr)
    serialRdr.setModes(mdPool)
-----construir el MainController-----
    mcBuilder= new MCBUILDER(mdPool)
    mcDirector= new MCDirector(mcBuilder)
    mcDirector.build()
    mCtrl= mcBuilder.getMC()
-----construir el temporizadores-----
    cmdTimeOut100=new ControllerTimeOut(mCtrl,serialRdr,serialWr)
    const=Constants::instance()
    deltaT=const.getDELTAT()
    t100= new FirstTimer(cmdTimeOut100)
    t100.setPeriod(deltaT)
    t15=mcBuilder.getLocalTimer()
    t15.setPeriod(1,5)
-----iniciar el sistema-----
    itHallSensors=mcBuilder.getActiveSensors()
    itHallSensors.first()
    while not itHallSensors.end()
        hallSensor=itHallSensors.getElement()
        hallSensor.start()
        itHallSensors.next()
    t15.start()
    t100.start()
    while(1){...espera interrupciones...}
}

```

## Capítulo 8

# Patrones de Diseño

Pattern	Conexión entre la PC y el MCU	F
based on	Estado (State)	
because	<p><b>Cambios previstos:</b> El modo en el que se comunican el MCU y la PC depende del estado de conexión entre éstos, podría cambiar dicho modo de comunicación, modificando el comportamiento ante cierto estado o bien requiriendo nuevo comportamiento asociado a nuevos estados en la conexión.</p> <p><b>Funcionalidad:</b> La comunicación entre el MCU y la PC requiere cierta sincronización, por tanto el comportamiento de dicha comunicación depende del estado de la conexión entre éstos. Es decir, el comportamiento cambia dinámicamente dependiendo del estado interno de la conexión.</p>	
where	<p><code>ConnectionState</code> is Estado <code>NoMessage</code> is EstadoConcreto <code>ReadyMessage</code> is EstadoConcreto <code>Transmitting</code> is EstadoConcreto <code>ConnectPCtoMCU</code> is Contexto</p>	

Pattern	Algoritmos de control para las ruedas	F
based on	Estrategia (Strategy)	
because	<p><b>Cambios previstos:</b> Los algoritmos que determinan la tensión a aplicar a una rueda podrían cambiar o incluso podría requerirse que se incorporen otros nuevos.</p> <p><b>Funcionalidad:</b> Las órdenes para el movimiento de las ruedas pueden estar dadas por medio de un valor de tensión, de corriente o de velocidad, los cuales a través de los correspondientes algoritmos serán utilizados para calcular la tensión adecuada, para proveer a las ruedas. De este modo es necesario que la tensión sea calculada a partir algoritmos distintos. Abstraer estos algoritmos permite que el cliente no deba hacer diferencias al momento de llevar a cabo el cálculo.</p>	
where	<p><code>Algorithm</code> is Estrategia <code>TensAlgorithm</code> is EstrategiaConcreta <code>VelAlgorithm</code> is EstrategiaConcreta <code>CntAlgorithm</code> is EstrategiaConcreta <code>SetTension</code> is Contexto</p>	

<b>Pattern</b>	<b>Algoritmo de control para la dirección</b>
<b>based on</b>	Estrategia (Strategy)
<b>because</b>	<b>Cambios previstos:</b> El algoritmo de control que determina la dirección de giro del robot podrían cambiar.
<b>where</b>	<b>Algorithm</b> is Estrategia <b>DirAlgorithm</b> is EstrategiaConcreta <b>DActive</b> is Contexto

<b>Pattern</b>	<b>Valores constantes</b>
<b>based on</b>	Único (Singleton)
<b>because</b>	<b>Cambios previstos:</b> Los valores constantes que son utilizados por más de un módulo residirán en <b>Constants</b> . Podría ser necesario modificar dichos valores o incorporar nuevos, que deban ser utilizados en forma global.  <b>Funcionalidad:</b> Ciertos valores constantes son utilizados por más de un módulo a fin de llevar a cabo cálculos. Se utiliza un único módulo que los contenga y los oculte para tener bajo control que los valores establecidos no serán replicados innecesariamente ni serán modificados.
<b>where</b>	<b>Constants</b> is Singleton

<b>Pattern</b>	<b>Argumentos de funciones</b>
<b>based on</b>	Decorador (Decorator)
<b>because</b>	<b>Cambios previstos:</b> El sistema requiere distintos cálculos sobre valores reales que son implementados mediante módulos que llevan a cabo el cálculo de alguna función. <i>Todas las funciones requieren un argumento básico, pero difieren en otros argumentos que también son requeridos.</i> Este patrón agrega al argumento básico de una función, argumentos adicionales, abstrayendo estas diferencias. Podría ser necesario, agregar más argumentos debido a que el cálculo de alguna función lo requiera, o bien porque sea necesario definir nuevas funciones que necesiten nuevos argumentos.  <b>Funcionalidad:</b> Hay funciones que requieren solo un valor real como argumento, otras que requieren este valor más un conjunto de otros argumentos, y otras que necesitan todos los argumentos mencionados más algunos adicionales. Más allá de estas diferencias, los clientes tratan a todas estas funciones y sus argumentos del mismo modo; permitiendo que ante los cambios posibles, los mismos sean transparentes para los clientes.
<b>where</b>	<b>CalculationData</b> is Componente <b>SimpleData</b> is ComponenteConcreto <b>MoreData</b> is Decorador <b>MapData</b> is DecoradorConcreto <b>CRData</b> is DecoradorConcreto

**Pattern****Funciones****based on**

Estrategia (Strategy)

**because**

**Cambios previstos:** El sistema requiere distintos cálculos sobre valores reales, provenientes del exterior del sistema. Estos cálculos son implementados en distintos módulo como funciones que reciben un argumento y dan un valor como resultado. Los cálculos implementados podrían cambiar; o bien podrían ser necesario nuevos cálculos.

**Funcionalidad:** El sistema debe implementar distintas funciones que realicen un cálculo que transforme cierto valor real provenientes del exterior en otro valor útil para el sistema. Todas las funciones tienen la misma interfaz y reciben el mismo tipo de argumento que es encapsulado en un módulo **CalculationData**. Sin embargo, cada función sabe qué elementos necesita de estos argumentos para llevar a cabo el cálculo y los toma del mencionado módulo. Por tanto, los clientes simplemente invocan la función y le pasan los argumentos a esta; sin tener que conocer qué algoritmo elegir ni qué datos enviarle.

**where**

**Function** is Estrategia  
**OrientationCalc** is EstrategiaConcreta  
**MapFunction** is EstrategiaConcreta  
**InverseFunction** is EstrategiaConcreta  
**VelBuffer** is Contexto  
**DirBuffer** is Contexto  
**Wheel** is Contexto  
**ValueCollector** is Contexto

**Pattern****Recolectores de instantes de tiempo****based on**

Decorador (Decorator)

**because**

**Cambios previstos:** El sistema requiere que ciertos elementos registren distintos instantes de tiempo para llevar a cabo ciertos cálculos. Podría cambiar algún cómputo que se hace con ellos o bien podría ser necesario agregar nuevos cálculos basados en los instantes de tiempo registrados.

**Funcionalidad:** Tanto los pines del CR como los sensores Hall requieren registrar los instantes de tiempo en los que tiene lugar una interrupción provocada por estos. Pero tales registros son utilizados de modos diferentes en cada caso. De esta manera, este patrón permite implementar la funcionalidad de registrar un instante de tiempo donde ocurre una interrupción, y luego extender dicha funcionalidad en otros módulos; ocultando así las diferencias en una interfaz común.

**where**

**Collector** is Componente  
**TimeCollector** is ComponeneteConcreto  
**DecoCollector** is Decorador  
**CRpinCollector** is DecoradorConcreto  
**SensorCollector** is DecoradorConcreto



<b>Pattern</b>	<b>Buffers del CR</b>
<b>based on</b>	Método Plantilla (Template Method)
<b>because</b>	<p><b>Cambios previstos:</b> Los buffers, asociados a los pines del CR, registran y realizan cálculos sobre las señales recibidas en los pines; de modo tal, que estas sean interpretadas. El modo en el cual las señales son interpretadas, y por tanto los cálculos asociados, podría cambiar.</p> <p><b>Funcionalidad:</b> Tanto el buffer asociado al pin de velocidad del CR, como aquel asociado a la dirección, deben realizar el mismo algoritmo principal para calcular el valor registrado; salvo, por el paso en el que se calcula la orientación (adelante/atrás) de las ruedas, en el cual las funcionalidades difieren. Dicho paso es implementado como una primitiva, en las clases concretas, que será invocada en el método plantilla declarado en la clase abstracta.</p>
<b>where</b>	<p><b>CRBuffer</b> is ClaseAbstracta  <b>VelBuffer</b> is ClaseConcreta  <b>DirBuffer</b> is ClaseConcreta  <b>getVal()</b> is MetodoPlantilla  <b>calcOrientation()</b> is OperaciónPrimitiva</p>

<b>Pattern</b>	<b>Órdenes sobre un sistema de control de rueda</b>
<b>based on</b>	Estrategia (Strategy)
<b>because</b>	<p><b>Cambios previstos:</b> El sistema deberá llevar a cabo distintas órdenes sobre un sistema de control de rueda. Por ejemplo, registrar la posición de la rueda, leer el valor medido por un sensor, etc. El modo en el cual cada orden se realiza podría cambiar, como así también, podría surgir la necesidad de implementar nuevas órdenes.</p> <p><b>Funcionalidad:</b> El sistema debe implementar distintas órdenes, que el controlador principal indicará que se realicen, en cada uno de los sistemas de control de rueda. De este modo cada orden será implementada como un módulo, que recibirá un sistema de control de rueda y llevará a cabo sobre este las acciones correspondientes.</p>
<b>where</b>	<p><b>WSysOrder</b> is Estrategia  <b>SaveWPosition</b> is EstrategiaConcreta  <b>SensorWritesVel</b> is EstrategiaConcreta  <b>ControllerReadsVel</b> is EstrategiaConcreta  <b>SensorWritesCnt</b> is EstrategiaConcreta  <b>ControllerReadsCnt</b> is EstrategiaConcreta  <b>SaveWheelPositions</b> is Contexto  <b>SensorsWrite</b> is Contexto  <b>ControllersRead</b> is Contexto</p>

<b>Pattern</b>	<b>Comando para controlar una rueda</b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo sobre una rueda a fin de controlar su funcionamiento podrían cambiar, incluso podría cambiar el receptor de tal acción. Por ejemplo, para frenar una rueda el receptor en lugar de ser la rueda, podría cambiar a un dispositivo intermedio que luego provoque el frenado de la misma.</p> <p><b>Funcionalidad:</b> Las órdenes como frenar una rueda, establecer una tensión determinada, etc. pueden tener efecto sobre la rueda y sobre otros elementos del sistema. Este patrón permite ocultar sobre qué elementos tiene efecto el comando y por tanto quienes los invocan no requieren de esta información.</p>
<b>where</b>	<p>WCtrlCommand is Orden</p> <p>VelNull is OrdenConcreta</p> <p>Brake is OrdenConcreta</p> <p>SetTension is OrdenConcreta</p> <p>ChangeOrientation is OrdenConcreta</p> <p>Wheel is Receptor</p> <p>WheelCtrlData is Receptor</p> <p>WheelController is Receptor</p> <p>WheelSystem is Receptor</p> <p>SensorWritesVel is Receptor</p> <p>ControllerReadsVel is Receptor</p> <p>ResetVel is Invocador</p> <p>Stop is Invocador</p> <p>Advance is Invocador</p> <p>Reverse is Invocador</p> <p>WSBuilder is Cliente</p>

<b>Pattern</b>	<b>Algoritmos para controlar una rueda</b>
<b>based on</b>	Estrategia (Strategy)
<b>because</b>	<p><b>Cambios previstos:</b> El sistema requiere diferentes algoritmos para llevar a cabo órdenes del sistema, que permitan el control de una rueda. Estos algoritmos podrían cambiar, o bien podría ser necesario utilizar otros adicionales.</p> <p><b>Funcionalidad:</b> El sistema requiere diferentes algoritmos para llevar a cabo el control de una rueda. Ante distintas órdenes, deberán ejecutarse distintos algoritmos. Y en algunos casos, ante la misma orden, pero distintas circunstancias, deberán llevarse a cabo diferentes algoritmos. Por ejemplo, una orden de dar marcha atrás, será llevada a cabo de distintas formas dependiendo de las condiciones bajo las cuales esté dada la orden.</p>
<b>where</b>	<p>WCtrlAlgorithm is Estrategia</p> <p>Stop is EstrategiaConcreta</p> <p>ResetVel is EstrategiaConcreta</p> <p>Advance is EstrategiaConcreta</p> <p>Reverse is EstrategiaConcreta</p> <p>ReverseRPM is EstrategiaConcreta</p> <p>WheelController is Contexto</p>

<b>Pattern</b>	<b>Extensión de los algoritmos para controlar una rueda</b>
<b>based on</b>	Decorador (Decorator)
<b>because</b>	<p><b>Cambios previstos:</b> El sistema requiere diferentes algoritmos para llevar a cabo órdenes del sistema, que permitan el control de una rueda. Estos algoritmos podrían cambiar, podrían ser necesarios otros adicionales, o podría requerirse nuevos que sean la extensión de algunos existentes.</p> <p><b>Funcionalidad:</b> Algunos algoritmos para controlar una rueda son el resultado de combinar otros algoritmos más simples. Por ejemplo, uno de los algoritmos para dar marcha atrás, debe utilizar el algoritmo que detiene la rueda.</p>
<b>where</b>	<p><b>WCtrlAlgorithm</b> is Componente</p> <p><b>Stop</b> is ComponenteConcreto</p> <p><b>ResetVel</b> is ComponenteConcreto</p> <p><b>Advance</b> is ComponenteConcreto</p> <p><b>Reverse</b> is ComponenteConcreto</p> <p><b>ReverseOrientation</b> is Decorador</p> <p><b>Reverse</b> is DecoradorConcreto</p> <p><b>ReverseRPM</b> is DecoradorConcreto</p>

<b>Pattern</b>	<b>Comandos para controlar el dispositivo de dirección</b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo sobre el dispositivo de dirección, a fin de controlar su funcionamiento, podrían cambiar; incluso podría cambiar el receptor de tal acción o bien ser necesarias nuevas acciones.</p> <p><b>Funcionalidad:</b> Órdenes como establecer el sentido de giro, deshabilitar el dispositivo, etc., pueden tener efecto sobre el dispositivo de dirección y eventualmente sobre otros elementos. Este patrón permite ocultar sobre qué elementos tiene efecto la orden y por tanto quienes los invocan no requieren de esta información.</p>
<b>where</b>	<p><b>DCtrlCommand</b> is Orden</p> <p><b>Turn</b> is OrdenConcreta</p> <p><b>SetDirection</b> is OrdenConcreta</p> <p><b>Enable</b> is OrdenConcreta</p> <p><b>Disable</b> is OrdenConcreta</p> <p><b>SteeringDevice</b> is Receptor</p> <p><b>DirCtrlData</b> is Receptor</p> <p><b>DTurning</b> is Invocador</p> <p><b>DActive</b> is Invocador</p> <p><b>OnState</b> is Invocador</p> <p><b>OffState</b> is Invocador</p> <p><b>DSBuilder</b> is Cliente</p>

<b>Pattern</b>	<b>Estados de encendido o apagado del dispositivo de dirección</b>
<b>based on</b>	Estado (State)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones necesarias para habilitar o deshabilitar el dispositivo de dirección podrían cambiar.</p> <p><b>Funcionalidad:</b> Ciertas acciones en el sistema, requieren encender o apagar el dispositivo de dirección, lo cual deberá cambiar en forma dinámica durante el proceso de control. Por ejemplo, una orden de encendido se llevará a cabo, dependiendo si el dispositivo ya está o no encendido. Utilizando este patrón, quien indica la orden no debe preocuparse por verificar previamente en qué estado está el mismo; simplemente da la orden y es el estado el que debe determina si efectivamente es necesario habilitar o deshabilitar el mencionado dispositivo.</p>
<b>where</b>	<p><b>DirController</b> is Contexto</p> <p><b>DeviceState</b> is Estado</p> <p><b>OnState</b> is EstadoConcreto</p> <p><b>OffState</b> is EstadoConcreto</p>

<b>Pattern</b>	<b>Estados de operación del controlador del dispositivo de dirección</b>
<b>based on</b>	Estado (State)
<b>because</b>	<p><b>Cambios previstos:</b> El controlador del dispositivo de dirección deberá llevar a cabo diferentes acciones dependientes del estado en el que esté el sistema. Estas acciones podrían cambiar; esto es, ser modificadas, extendidas o eliminadas. Podrían además, ser necesarias nuevas acciones a realizar en nuevos estados.</p> <p><b>Funcionalidad:</b> El controlador del sistema de dirección realizará básicamente dos tareas: controlar, lo cual implica decidir si debe realizarse o no el giro del dispositivo; y enviar los pulsos al dispositivo para que gire, si correspondiese. Estas tareas implicarán distintas acciones dependiendo de en qué estado esté el controlador. Por ejemplo, realizar el control si el dispositivo está inactivo, implica no hacer nada; sin embargo, controlar en el estado en el que se está deteniendo el dispositivo, implica deshabilitar el mismo.</p>
<b>where</b>	<p><b>DOperationState</b> is Estado</p> <p><b>DInactive</b> is EstadoConcreto</p> <p><b>DTurning</b> is EstadoConcreto</p> <p><b>DActive</b> is EstadoConcreto</p> <p><b>DStopping</b> is EstadoConcreto</p> <p><b>DirController</b> is Contexto</p>

<b>Pattern</b>	<b>Modos de operación del sistema principal: CR y PC</b>
<b>based on</b>	Estado (State)
<b>because</b>	<p><b>Cambios previstos:</b> La manera en la que el sistema obtiene una orden desde el CR o desde la PC podría cambiar. También podrían incorporarse nuevos modos por medio de los cuales obtener las órdenes provenientes desde el exterior del sistema.</p> <p><b>Funcionalidad:</b> Para obtener una orden desde el exterior, el sistema debe poder evaluar si hay una nueva orden a procesar y, en tal caso, leerla. La forma en la que se deben llevar a cabo estas dos tareas difiere entre los distintos orígenes de las órdenes (CR/PC). De este modo, dependiendo de en qué modo de lectura de órdenes esté el sistema, deberá realizar diferentes acciones para leer una orden. Por otra parte, cuando hay pérdida de señal el sistema deberá leer en forma alternada desde la PC y el CR. Dicha alternancia, será llevada a cabo por los mismos estados concretos (CR/PC); es decir, cada estado concreto sabe qué debe hacer ante un pedido de lectura y de no poder llevarlo a cabo debe indicar a su estado sucesor como el nuevo estado; esto es, con el objetivo de que el sistema intente la nueva lectura desde el otro modo.</p>
<b>where</b>	<p><b>Mode is</b> Estado</p> <p><b>PC is</b> EstadoConcreto</p> <p><b>CR is</b> EstadoConcreto</p> <p><b>Working is</b> Contexto</p> <p><b>WaitingN is</b> Contexto</p> <p><b>WaitingMAX is</b> Contexto</p> <p><b>Reconnecting is</b> Contexto</p>

<b>Pattern</b>	<b>Modos de operación CR y PC como extensiones de una tarea básica</b>
<b>based on</b>	Decorador (Decorator)
<b>because</b>	<p><b>Cambios previstos:</b> Los posibles modos de lectura de órdenes recibidas desde el exterior del sistema, podrían cambiar. También podrían ser necesarios nuevos modos de lectura.</p> <p><b>Funcionalidad:</b> Los pasos a seguir para evaluar si hay una nueva orden a procesar o para leerla, son los mismos tanto si se trata de la PC como si se trata del CR. Sin embargo, lo que cambia es el lector <b>Reader</b> que provee la información. De esta manera, el modo básico que realiza estas tareas será decorado para constituir los modos CR y PC. Dicha decoración extenderá la funcionalidad del modo básico, permitiendo que los modos PC y CR cambien el modo de lectura de órdenes del controlador principal.</p>
<b>where</b>	<p><b>Mode is</b> Componente</p> <p><b>BasicMode is</b> ComponenteConcreto</p> <p><b>LectureMode is</b> Decorador</p> <p><b>PC is</b> DecoradorConcreto</p> <p><b>CR is</b> DecoradorConcreto</p>

<b>Pattern</b>	<b>Órdenes del controlador principal sobre los sistemas de control</b>
<b>based on</b>	Método Plantilla (Template Method)
<b>because</b>	<p><b>Cambios previstos:</b> Las órdenes que debe llevar a cabo el controlador principal sobre los distintos sistemas de control, podría cambiar. También podría ser necesario incorporar nuevas órdenes.</p> <p><b>Funcionalidad:</b> El controlador principal debe llevar a cabo el control de los sistemas de control de rueda y del sistema de dirección. Para esto, se replican ciertas órdenes para todos los sistema mencionados; por ejemplo, que los sensores emitan sus valores. Es decir, la iteración sobre los sistemas de rueda para dar una orden y luego la indicación de una orden para el sistema de dirección; es igual cualquiera sea la orden dada. Por tanto, se define un método plantilla que realiza esto, el cual invoca métodos generales a todas las clases concretas. Cada clase concreta implementará el método invocado, de acuerdo a la orden que represente.</p>
<b>where</b>	<p><b>Order</b> is ClaseAbstracta  <b>SaveWheelPositions</b> is ClaseConcreta  <b>SensorsWrite</b> is ClaseConcreta  <b>ControllersRead</b> is ClaseConcreta  <b>ControllersControl</b> is ClaseConcreta  <b>ControllersStop</b> is ClaseConcreta  <b>execute(i ControlSystemPool)</b> is MetodoPlantilla()  <b>actionOnWheelSys(i WheelSystem)</b> is OperacionPrimitiva()  <b>actionOnDirSys(i DirSystem)</b> is OperacionPrimitiva()</p>

<b>Pattern</b>	<b>Estados de operación del controlador principal</b>
<b>based on</b>	Estado (State)
<b>because</b>	<p><b>Cambios previstos:</b> El controlador principal llevará a cabo el control de los subsistemas de control, dependiendo del estado en el que se encuentre. Podrían cambiar el comportamiento requerido de algunos de los estados definidos o bien podría ser necesario agregar nuevos estados con sus correspondientes comportamientos.</p> <p><b>Funcionalidad:</b> El comportamiento del controlador principal está sujeto a la llegada de órdenes desde la PC o el CR; por tanto, debe haber cierta sincronización entre el controlador y los elementos que proveen las órdenes. Esto lleva a que el comportamiento del controlador dependa del estado en el que se encuentra; y cambie de estado ante ciertas circunstancias. Por ejemplo, si el sistema no recibe órdenes por cierto tiempo; el controlador principal pasará a estar en un estado de espera, el que abandonará si recibe una orden; o bien, pasará a un estado de reconexión si ha esperado más de lo establecido.</p>
<b>where</b>	<p><b>MainController</b> is Contexto  <b>OperationState</b> is Estado  <b>WaitingN</b> is EstadoConcreto  <b>WaitingMAX</b> is EstadoConcreto  <b>Working</b> is EstadoConcreto  <b>Reconnecting</b> is EstadoConcreto</p>

<b>Pattern</b>	<b>Lectura de órdenes en los estados de operación del controlador principal</b>
<b>based on</b>	Método Plantilla (Template Method)
<b>because</b>	<p><b>Cambios previstos:</b> Podría cambiar las acciones a realizar ante la presencia de un mensaje; o bien, ante la ausencia; dependiendo del estado de operación en el que se encuentre el sistema.</p> <p><b>Funcionalidad:</b> Las acciones que debe realizar el controlador principal ante la presencia o la ausencia de nuevos mensajes, cambia dependiendo del estado en el que este se encuentre. Sin embargo, en cualquiera de los estados en los que esté, deberá evaluar si hay o no un nuevo mensaje; y en función de esto, hacer una u otra cosa. Esta funcionalidad es implementada entonces en un método plantilla que utilizarán todos los estados. Cada estado entonces, implementará qué acciones tomará en caso de estar ante la presencia de un mensaje; y cuales, en caso contrario.</p>
<b>where</b>	<p><b>OperationState</b> is ClaseAbstracta</p> <p><b>WaitingN</b> is ClaseConcreta</p> <p><b>WaitingMAX</b> is ClaseConcreta</p> <p><b>Reconnecting</b> is ClaseConcreta</p> <p><b>Working</b> is ClaseConcreta</p> <p>read(i MainController) is MetodoPlantilla()</p> <p>actionWithMsg(i MainController, i Mode) is OperacionPrimitiva()</p> <p>actionNoMsg(i MainController) is OperacionPrimitiva()</p>

<b>Pattern</b>	<b>Envío y recepción de información entre el MCU y, la PC y el CR</b>
<b>based on</b>	Serializador (Serializer)
<b>because</b>	<p><b>Cambios previstos:</b> El formato de la información que se intercambia entre el MCU y, la PC y el CR, podría cambiar; como así también el modo de la comunicación que se lleva a cabo entre éstos. También podría cambiar el origen o destino de datos; es decir, incorporar un nuevo origen de información con diferentes modos de conexión y formato de datos.</p> <p><b>Funcionalidad:</b> El MCU debe recibir información de dos orígenes diferentes (CP y CR) y debe también poder enviar información a uno de éstos (PC). De este modo es necesario, ocultar los diferentes modos en los que la información es accedida o enviada, y los diferentes formatos que pueda tener la información transmitida.</p>
<b>where</b>	<p><b>Serializable</b> is Serializable</p> <p><b>MainController</b> is ElementoConcreto</p> <p><b>ControlSystemPool</b> is ElementoConcreto</p> <p><b>Reader</b> is Lector</p> <p><b>SerialReader</b> is LectorConcreto</p> <p><b>BufferReader</b> is LectorConcreto</p> <p><b>ConnectPCtoMCU</b> is ParteTrasera</p> <p><b>CRBuffer</b> is ParteTrasera</p> <p><b>Writer</b> is Escribiente</p> <p><b>SerialWriter</b> is EscribienteConcreto</p> <p><b>ConnectMCUtoPC</b> is ParteTrasera</p>

<b>Pattern</b>	<b>Comando para manejar interrupciones físicas provenientes de sensores Hall. Sustitución de <i>callback</i></b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo ante una señal física de un sensor Hall podrían cambiar o incluso podría cambiar el receptor de dichas acciones.</p> <p><b>Funcionalidad:</b> Se mantienen los niveles de abstracción. Los módulos de más bajo nivel como <b>ActiveSensor</b>, desconocen la existencia de módulos de niveles superiores de abstracción como <b>SensorCollector</b> o <b>VelSensor</b>.</p>
<b>where</b>	<p><b>Command</b> is Orden</p> <p><b>CountSignal</b> is OrdenConcreta</p> <p><b>SensorCollector</b> is Receptor</p> <p><b>ActiveSensor</b> is Invocador</p> <p><b>WSBuilder</b> is Cliente</p>

<b>Pattern</b>	<b>Comando para manejar interrupciones físicas del temporizador secundario en el sistema de dirección. Sustitución de <i>callback</i></b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo ante una señal física, del temporizador secundario que tendrán efecto sobre elementos del sistema de dirección, podrían cambiar; o incluso podrían cambiar los receptores de dichas acciones.</p> <p><b>Funcionalidad:</b> Se mantienen los niveles de abstracción. El módulo de más bajo nivel, como el temporizador secundario, desconoce la existencia de módulos de niveles superiores como el controlador del sistema de dirección o el sensor de dirección.</p>
<b>where</b>	<p><b>Command</b> is Orden</p> <p><b>DirCtrlTimeOut</b> is OrdenConcreta</p> <p><b>DirController</b> is Receptor</p> <p><b>DirSensor</b> is Receptor</p> <p><b>SecondTimer</b> is Invocador</p> <p><b>DSBuilder</b> is Cliente</p>



<b>Pattern</b>	<b>Comando para manejar interrupciones físicas del temporizador secundario en el sistema control de rueda. Sustitución de <i>callback</i></b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo ante una señal física, del temporizador secundario que tendrán efecto sobre elementos del sistema de control de rueda, podrían cambiar; o incluso podría cambiar el receptor de dichas acciones.</p> <p><b>Funcionalidad:</b> Se mantienen los niveles de abstracción. El módulo de más bajo nivel, como el temporizador secundario, desconoce la existencia de módulos de niveles superiores como el sensor de corriente.</p>
<b>where</b>	<p><b>Command</b> is Orden</p> <p><b>ReadCnt</b> is OrdenConcreta</p> <p><b>ValueCollector</b> is Receptor</p> <p><b>SecondTimer</b> is Invocador</p> <p><b>WSBuilder</b> is Cliente</p>

<b>Pattern</b>	<b>Comando para manejar interrupciones físicas provenientes del un pin del CR. Sustitución de <i>callback</i></b>
<b>based on</b>	Orden (Command)
<b>because</b>	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo ante una señal física, proveniente del pin de velocidad o del pin de dirección del CR, podrían cambiar; o incluso podría cambiar el receptor de dichas acciones.</p> <p><b>Funcionalidad:</b> Se mantienen los niveles de abstracción. El módulo de más bajo nivel, como el pin asociado al CR, desconoce la existencia de módulos de niveles superiores como los buffers del CR o los lectores de estos.</p>
<b>where</b>	<p><b>Command</b> is Orden</p> <p><b>CountTime</b> is OrdenConcreta</p> <p><b>CRpinCollector</b> is Receptor</p> <p><b>Pin</b> is Invocador</p> <p><b>Main</b> is Cliente</p>

F

Pattern	<b>Comando para manejar interrupciones físicas del temporizador principal. Sustitución de <i>callback</i></b>
based on	Orden (Command)
because	<p><b>Cambios previstos:</b> Las acciones a llevar a cabo ante una señal física del temporizador principal que marca los ciclos de control, podría cambiar; o incluso podrían cambiar los receptores de dichas acciones.</p> <p><b>Funcionalidad:</b> Se mantienen los niveles de abstracción. Los módulos de más bajo nivel, como el temporizador principal, desconoce la existencia de módulos de niveles superiores de abstracción como el controlador principal.</p>
where	<p><b>Command</b> is Orden</p> <p><b>ControllerTimeOut</b> is OrdenConcreta</p> <p><b>MainController</b> is Receptor</p> <p><b>FirstTimer</b> is Invocador</p> <p><b>Main</b> is Cliente</p>

F

Pattern	<b>Construcción de un sistema de control de ruedas</b>
based on	Constructor (Builder)
because	<p><b>Cambios previstos:</b> El sistema de control de una rueda está constituido por un conjunto de partes que se vinculan de cierto modo, dicho conjunto y el modo en que se vinculan los elementos podría cambiar; por ejemplo, además de los sensores de corriente y de velocidad podría incorporarse otro sensor o bien ser eliminado alguno de los mencionados.</p>
where	<p><b>WSDirector</b> is Director</p> <p><b>WheelSysBuilder</b> is Constructor</p> <p><b>WSBuilder</b> is ConstructorConcreto</p> <p><b>WheelSystem</b> is Producto</p> <p><b>ReadCnt</b> is Producto</p> <p><b>ActiveSensor</b> is Producto</p>

F

Pattern	<b>Construcción del sistema de control de dirección</b>
based on	Constructor (Builder)
because	<p><b>Cambios previstos:</b> El sistema de control de dirección está constituido por un conjunto de partes que se vinculan de cierto modo, dicho conjunto y el modo en que se vinculan los elementos podría cambiar; por ejemplo, el mecanismo (y por tanto el conjunto de módulos involucrados) por el cual esta parte del sistema hace girar las ruedas.</p>
where	<p><b>DSDirector</b> is Director</p> <p><b>DirSysBuilder</b> is Constructor</p> <p><b>DSBuilder</b> is ConstructorConcreto</p> <p><b>DirSystem</b> is Producto</p> <p><b>DirCtrlTimeOut</b> is Producto</p> <p><b>DActive</b> is Producto</p> <p><b>DStopping</b> is Producto</p>

<b>Pattern</b>	<b>Construcción del conjunto de sistemas de control</b>
<b>based on</b>	Constructor (Builder)
<b>because</b>	<b>Cambios previstos:</b> El grupo de sistemas que permiten controlar la actividad del traslado del robot podrían cambiar. Por ejemplo, podría haber más de un sistema de dirección, uno asociado a cada rueda.
<b>where</b>	<b>CSPDirector</b> is Director <b>CtrlSysPoolBuilder</b> is Constructor <b>CSPBuilder</b> is ConstructorConcreto <b>ControlSystemPool</b> is Producto <b>SecondTimer</b> is Producto
<b>comments</b>	El constructor del grupo de sistemas de control ( <b>ControlSysPool</b> ) está compuesto por los directores de construcción de los sistemas de control de rueda y dirección ( <b>WSDirector</b> y <b>DSDirector</b> ); y delega en dichos directores la construcción de los mencionados sistemas.

<b>Pattern</b>	<b>Construcción del controlador principal</b>
<b>based on</b>	Constructor (Builder)
<b>because</b>	<b>Cambios previstos:</b> El controlador principal está constituido por un conjunto de partes que se vinculan de cierto modo, dicho conjunto y el modo en que se vinculan los elementos podría cambiar. Por ejemplo, podría ser necesario incorporar un sistema que controle otro tipo de dispositivo, además de las ruedas y la dirección.
<b>where</b>	<b>MCDirector</b> is Director <b>MainControllerBuilder</b> is Constructor <b>MCBuilder</b> is ConstructorConcreto <b>MainController</b> is Producto
<b>comments</b>	El constructor del controlador principal ( <b>MCBuilder</b> ) está compuesto por el director de construcción del grupo de sistemas de control ( <b>CSPDirector</b> ); y delega en éste la construcción del grupo.

# Bibliografía

- [Cri06] Maximiliano Cristiá. Catálogo incompleto de estilos arquitectónicos. 2006.
- [ERRJ03] Gamma Eric, Helm Richard, Johnson Ralph, and Vlissides John. Patrones de Diseño. Pearson Education. ADDISON WESLEY, 2003.
- [Pom22a] Laura Pomponio. Requerimientos Funcionales de Software del Microcontrolador del Robot Desmalezador. Versión 1.2.1. 2022.
- [Pom22b] Laura Pomponio. Semántica del Diseño. Un puente entre los requerimientos y el diseño. Versión 0.8. 2022.
- [SG96] Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.