

The Atomic Theory as Applied To Gases, with Some Experiments on the Viscosity of Air

by
Silas W. Holman

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE IN PHYSICS

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1876

© 1876 Silas W. Holman. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Silas W. Holman
Department of Physics
May 18, 1876

Certified by: Edward C. Pickering
Professor of Physics, Thesis Supervisor

Accepted by: Tertius Castor
Professor of Log Dams
Graduate Officer, Department of Research

THESIS COMMITTEE

THESIS SUPERVISOR

Marcus Gavius Apicius

*Professor of Cooking Arts
Department of Food Science*

THESIS READERS

Marie-Antoine Carême

*Professor of Haute Cuisine
Department of Food Science*

Julia Child

*Professor of French Cuisine
Department of Food Science*

Miles Gloriosus

*Professor of Personal Pronouns
Department of Rhetoric*

The Atomic Theory as Applied To Gases, with Some Experiments on the Viscosity of Air

by

Silas W. Holman

Submitted to the Department of Physics
on May 18, 1876 in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE IN PHYSICS

RESUMEN

The developments of the “kinetic theory” of gases made within the last ten years have enabled it to account satisfactorily for many of the laws of gases. The mathematical deductions of Clausius, Maxwell and others, based upon the hypothesis of a gas composed of molecules acting upon each other at impact like perfectly elastic spheres, have furnished expressions for the laws of its elasticity, viscosity, conductivity for heat, diffusive power and other properties. For some of these laws we have experimental data of value in testing the validity of these deductions and assumptions. Next to the elasticity, perhaps the phenomena of the viscosity of gases are best adapted to investigation.¹

Thesis supervisor: Edward C. Pickering

Title: Professor of Physics

¹Text from Holman (1876): doi:[10.2307/25138434](https://doi.org/10.2307/25138434).

Acknowledgments

Write your acknowledgments here.

Biographical Sketch

Silas Whitcomb Holman was born in Harvard, Massachusetts on January 20, 1856. He received his S.B. degree in Physics from MIT in 1876, and then joined the MIT Department of Physics as an Assistant. He became Instructor in Physics in 1880, Assistant Professor in 1882, Associate Professor in 1885, and Full Professor in 1893. Throughout this period, he struggled with increasingly severe rheumatoid arthritis. At length, he was defeated, becoming Professor Emeritus in 1897 and dying on April 1, 1900.

Holman's light burned brilliantly before his tragic and untimely death. He published extensively in thermal physics, and authored textbooks on precision measurement, fundamental mechanics, and other subjects. He established the original Heat Measurements Laboratory. Holman was a much admired teacher among both his students and his colleagues. The reports of his department and of the Institute itself refer to him frequently in the 1880's and 1890's, in tones that gradually shift from the greatest respect to the deepest sympathy.

Holman was a student of Professor Edward C. Pickering, then head of the Physics department. Holman himself became second in command of Physics, under Professor Charles R. Cross, some years later. Among Holman's students, several went on to distinguish themselves, including: the astronomer George E. Hale ('90) who organized the Yerkes and Mt. Wilson observatories and who designed the 200 inch telescope on Mt. Palomar; Charles G. Abbot ('94), also an astrophysicist and later Secretary of the Smithsonian Institution; and George K. Burgess ('96), later Director of the Bureau of Standards.

Índice general

<i>Índice de figuras</i>	13
<i>Índice de cuadros</i>	15
1. Problemas comunes	17
1.1. Acceso al hardware	17
1.2. Interfaces que no se ajustan perfectamente	22
1.3. Control en conjunto de dispositivos	24
1.3.1. Subsistemas de control	25
1.3.2. Ejemplo	28
1.4. Obtención de información	35
1.5. Control anti-rebote	38
1.6. Manejo de interrupciones	41
1.7. Máquinas de estado	41
1.8. Integridad de la información	46
1.9. Verificación de precondiciones.	48
1.10. Organización de la ejecución	50
A. Code listing	53

Índice de figuras

1.1. Conexionado módulo DRV8838	17
1.2. Interfaz MotorDC	20
1.3. Módulo MotorDC abstractor y estructura de herencia.	21
1.4. Display 7 segmentos 4 digitos	23
1.5. Configuración original	23
1.6. Configuración nueva	24
1.7. Módulos de un subsistema	26
1.8. Control	27
1.9. Diagrama componentes sistema brazo	29
1.10. Actuadores paso a paso	30
1.11. Sensores	31
1.12. Timer	32
1.13. Módulos necesarios para complementar el Controller	33
1.14. Controller	34
1.15. Iterator y ordenes	35
1.16. MainController	37
1.17. Componente sensor activo.	38
1.18. Ejemplo	39
1.19. State chart microondas	42
1.20. Ejemplo <i>Decorator</i> integridad de la información	47
1.21. Ejemplo de handler usando <i>State</i>	51

Índice de cuadros

Capítulo 1

Problemas comunes

1.1 Acceso al hardware

Una de las características distintivas de los sistemas embebidos es que trabajan directamente con dispositivos de hardware. Cada uno de estos tiene sus propios protocolos de comunicación y estándares de funcionamiento (por ejemplo, direcciones de memoria, codificación de bits, etc), por lo tanto el software se debe ajustar a sus requerimientos. Como se puede entender esta tarea no es simple y puede demandar mucho esfuerzo cada vez que se quiera modificar o agregar un componente de hardware. A su vez, puede que múltiples módulos de nuestro sistema embebido quieran acceder al dispositivo, por lo que cada uno debe encargarse de la comunicación creando código repetido y complicando aún más las modificaciones.

Para entender los inconvenientes que puede conllevar no diseñar pensando en el cambio veamos un ejemplo simple. Suponga que nuestro sistema embebido debe controlar un motor de corriente continua (DC) y que el software para hacerlo corre en un microcontrolador Arduino. Los requerimientos definen que es necesario poder asignar el sentido y la velocidad de rotación del motor según se necesite. Para controlar el motor se utiliza un módulo *DRV8838*, el cual se coloca entre el microcontrolador y el motor. Es necesario ya que el microcontrolador y su plataforma no pueden manejar las potencias necesarias para hacer funcionar el dispositivo. Podemos ver el conexionado del mismo en la figura 1.

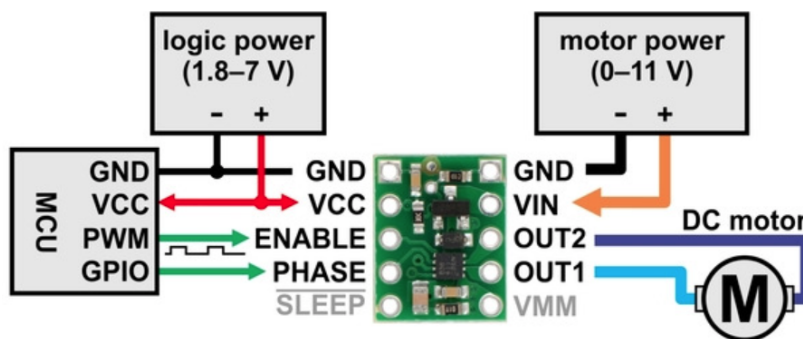


Figura 1.1: Conexionado módulo DRV8838

Notemos que el DRV8838 tiene 3 pines de control, *ENABLE*, *PHASE* y *SLEEP* pero solo

utilizaremos los 2 primeros para simplificar el ejemplo. De todas formas, esta es la función de cada pin:

PHASE	direccion de rotacion	digital
ENABLE	velocidad de rotación	analógico
SLEEP	liberar fuerza	digital

Por lo tanto, con los dos primeros pines podremos cumplir los requerimientos mencionados. Repasando, tendremos dos cables conectados a nuestro microcontrolador Arduino, uno que se dirige a *ENABLE* y otro a *PHASE*. Veamos ahora que código necesitamos escribir para poder hacer uso del motor:

Listing 1.1: Configuración inicial

```
1 # Notar que los numeros asignados a los pines son arbitrarios dentro
  del conjunto de pines disponibles en nuestro Aruino.
2
3 # Constantes globales
4 const int DIR\_pin = 7;
5 const int VEL\_pin = 9;
6
7 void setup() {
8     .
9     .
10    .
11    pinMode(DIR\_pin, OUTPUT);
12    pinMode(VEL\_pin, OUTPUT);
13    .
14    .
15    .
16 }
```

Listing 1.2: Máxima velocidad giro horario

```
1 digitalWrite(DIR\_pin, HIGH);
2 analogWrite(VEL\_pin, 255); # Maximo valor aceptado, PWM siempre
  encendido
```

Listing 1.3: Detenerce

```
1 analogWrite(VEL\_pin, 0);
```

No es necesario entender por completo que hace cada llamada, pero si es importante comprender que es necesario ejecutar ese código para controlar el motor. Es decir, que si cualquier cliente (se supone que ya definimos que es cliente) del motor en nuestro sistema deberá saber que para hacer que el motor vaya hacia adelante a la mayor velocidad posible es necesario ejecutar las dos funciones indicadas sobre los pines correspondientes al motor. Un pequeño ejemplo podría ser el siguiente, en el cual se consulta una bandera asociada al estado de un valor y en base a este se acciona el motor:

```

1      .
2      .
3      .
4  if (valor > 100) {
5      digitalWrite(DIR\_pin, HIGH);
6      analogWrite(VEL\_pin, 255);
7  } else {
8      analogWrite(VEL\_pin, 0);
9  }
10     .
11     .
12     .

```

¿Qué problemas tiene esta estrategia de cara al cambio?

- Hace que el código sea poco evidente, es decir, no es fácil saber de que se trata una cierta porción de código con solo leerlo. Y por lo tanto provoca que sea difícil de modificar, requiere un trabajo extra de entendimiento antes de poder aplicar cualquier cambio.
- Imagine el caso en el que por cierto motivo se debe invertir el sentido de giro del motor, de manera que lo que era ir hacia adelante ahora es hacia atrás. Para llevar a cabo el cambio, debemos modificar todas las llamadas a `digitalWrite(DIR_pin, cambiando HIGH por DOWN y viceversa`. Es fácil cometer un error y dejar al sistema en un estado inconsistente.
- Ahora, qué pasa si tenemos que agregar un segundo motor del mismo tipo con el mismo controlador? Ya sea por duplicación de la potencia o un motor que cumpla otra función en el sistema. En el código tenemos que declarar nuevamente 2 constantes (suponga `DIR_pin2` y `VEL_pin2`, una para cada nuevo pin de control, también debemos setear esos pines como `OUTPUT` y ahora en diferentes partes del código tendremos llamadas a `digitalWrite` y `analogWrite` sobre diferentes pines lo cual es cada vez más confuso.
- Por cierto motivo se descompuso el controlador del motor, y no se consigue un reemplazo idéntico, sino que se adquiere un nuevo controlador de otra marca, por ejemplo, un *Pololu Simple Motor Controller G2*. En este caso, este controlador no utilizar la misma interfaz de control que el `DVR8838`, sino para controlarlo hay que acceder a el mediante comunicación serial (utiliza un solo pin específico). Incluso utilizando las herramientas provistas por el entorno de Arduino, el código no es similar:

Listing 1.4: Configuración

```

1      void set_up() {
2          .
3          .
4          .
5          Serial.begin(9000);
6          .
7          .
8          .
9      }

```

Figura 1.2: Interfaz MotorDC

Listing 1.5: Máxima velocidad giro horario

```
1 Serial.write(0xAA);
2 Serial.write(0x0C);
3 Serial.write(0x85);
4 Serial.write(0x7F);
```

Listing 1.6: Detenerse

```
1 Serial.write(0xAA);
2 Serial.write(0x0C);
3 Serial.write(0xE0);
```

Por lo tanto debemos modificar todos los usos de la antigua implementación por la nueva, lo cual además requerir un esfuerzo considerable, da pie a errores y obliga a re-probar código que ya se sabia que funcionaba correctamente.

Todos estos inconvenientes son derivados de que el *hardware* comprende un ítem de cambio frecuente; por lo que si seguimos la metodología de Parnas, debemos asignarle un módulo. Este es una representación virtual de hardware que debe proveer una interfaz lo suficientemente insensible a la implementación. Es decir, debemos pensar en lo que, en este caso, el motor siempre va a hacer independientemente de los posibles cambios que sufra el hardware subyacente. Un motor DC siempre recibirá ordenes para definir su sentido y velocidad de rotación. Veamos como podemos definir este módulo y que ventajas nos trae hacerlo.

MotorDC
MotorDC(i: int, i: int)
set_dir(i: Dir) set_vel(i: int)

Donde los parámetros del constructor **MotorDC** son los pines de control, el parámetro de **set_dir** es de tipo **Dir** el cual es un enum que tiene dos entradas, horario y anti-horario y por ultimo el input de **set_vel** es un int entre 0 y 255 que representa la velocidad porcentual. Veamos una posible implementación de esta interfaz utilizando el controlador DVR8838:

Listing 1.7: Constructor

```
1 void MotorDC(int dir_pin, int vel_pin) {
2     this->dir_pin = dir_pin;
3     this->vel_pin = vel_pin;
4     pinMode(this->dir_pin, OUTPUT);
5     pinMode(this->vel_pin, OUTPUT);
6 }
```

Figura 1.3: Módulo MotorDC abstractor y estructura de herencia.

```
1
2 void set_dir(Dir dir) {
3     if (dir == Dir.HORARIO) {
4         digitalWrite(DIR_pin, HIGH);
5     } else {
6         digitalWrite(DIR_pin, DOWN);
7     }
8 }
9 void set_vel(int vel) {
10     if vel > 0 && vel <= 255 {
11         analogWrite(VEL_pin, vel);
12     }
13 }
```

Ya son evidentes las primeras ventajas, es mucho más claro el siguiente código:

```
1 motor->set_dir(Dir.HORARIO)
2 motor->set_vel(255)
```

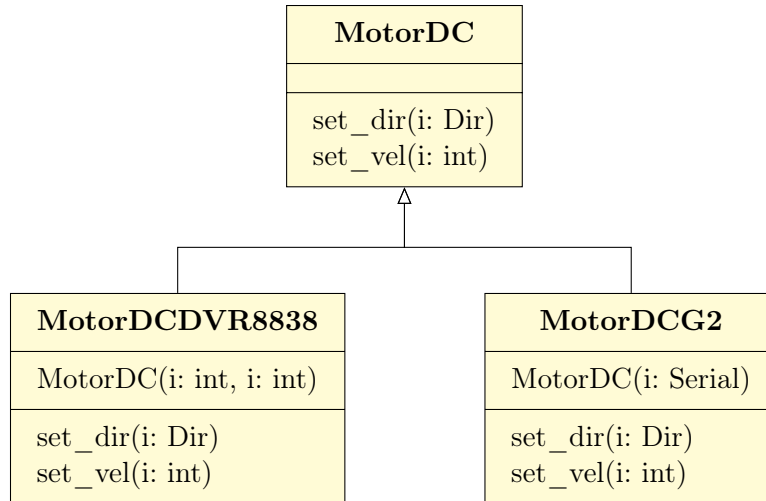
Además si debemos modificar el sentido de giro es tan fácil como cambiar la función `set_dir` y listo, los clientes no notaran el cambio. A su vez, si queremos controlar otro motor podemos hacer lo siguiente:

```
1 MotorDC motor_delantero = MotorDC(18, 19)
2
3 motor_delantero->set_dir(Dir.ANTIHORARIO)
4 motor_delantero->set_vel(10)
```

Y para un cambio de componente, como el explicado anteriormente, los clientes del módulo lo ignoraran, ya que sera prácticamente la misma (a menos un cambio en el constructor, donde ya no se necesitaran los pines). Por ejemplo, ahora `set_dir` será:

```
1 void set_vel(vel: int) {
2     if vel > 0 && vel <= 255 {
3         this->serial.write(0xAA);
4         this->serial.write(0x0C);
5         hex_vel = int_to_hex(vel)
6         this->serial.write(hex_vel);
7     }
8 }
```

Aún podemos mejorar el diseño haciendo uso del concepto de herencia (anteriormente explicado). Esto nos permitirá reutilizar módulos ya implementados y abstraer aun más a los clientes de la implementación. Para hacerlo definiremos un módulo MotorDC abstracto del cual heredarán la interfaz cada modelo o combinación de motor y controlador. En este caso tendremos:



Al cliente no le importa con cual de los dos tipos de controladores esta tratando, solo llama a las funciones provistas. También es posible agregar mas herederos para cada modelo de controlador/motor, y reutilizar las módulos implementados en caso de utilizar hardware idéntico.

De esta manera seguimos las prácticas recomendadas en las IS y logramos un diseño orientado al cambio.

1.2 Interfaces que no se ajustan perfectamente

Muchas veces el proveedor del hardware incluye con este librerías para su control, otras veces las conseguimos en internet o las extraemos previos proyectos. El problema recae en que estas interfaces pueden no ajustarse a las expectativas del sistema, generando la necesidad de ajustar la implementación de múltiples módulos. El hecho de que no se ajusten al sistema no quiere decir que este ultimo este mal diseñado o que las interfaces lo estén. Simplemente puede ocurrir que se diseñaron teniendo en cuenta diferentes puntos de vista, probablemente influenciados por los requerimientos particulares.

Para solventar este inconveniente, podemos aplicar el patrón *Adapter* de Gamma. En donde:

- **Target:** es la interfaz que utilizará el cliente.
- **Client:** cualquier módulo del sistema que requiera utilizar el hardware.
- **Adaptee:** interfaz que necesita ser adaptada para cumplir con lo que requiere el sistema.
- **Adapter:** módulo que adapta la interfaz.

Veamos un ejemplo, supóngase que un cierto sistema embebido está utilizando un display de 7 segmentos de 4 dígitos para mostrar la temperatura de funcionamiento y posición en grados de cierto actuador. Como el de la siguiente imagen:

Este display recibe la información utilizando comunicación en serie, con un protocolo propio del fabricante. Para facilitar su uso, este provee una librería que implementa la

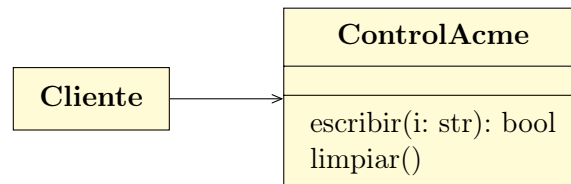


Figura 1.4: Display 7 segmentos 4 digitos

comunicación y provee funciones simples de usar, tales como *escribir(i: str): bool* y *limpiar()*. Por lo tanto, se implemento el sistema utilizando las funciones provistas para comunicarse, múltiples módulos llaman esas funciones.

En cierto momento el módulo display dejó de funcionar y se reemplazó por otro de un fabricante distinto, que funciona con otro protocolo de comunicación. De la misma manera, la empresa provee una librería para utilizar el display. Pero la interfaz no es la misma que la anterior e incluso algunos comportamientos son diferentes. Por ejemplo, en la primer librería la función de escritura devolvía False si se quería escribir y ya estaba mostrándose algo en el display, en la nueva si el display esta mostrando algo es pisado al momento de imprimir. Pero, provee una nueva función para verificar que se está mostrando en el momento en que es invocada *get_current(): str*. Entonces para adaptarnos al nuevo display tenemos que modificar todas las llamadas a las viejas funciones al rededor del sistema agregando la lógica nueva.

Figura 1.5: Configuración original



Para permitir el cambio propuesto sin tener que realizar todas las modificaciones mencionadas, podemos aplicar el patrón *Adapter*, creando un módulo intermedio con la misma interfaz que **DisplayLib**.

Por lo tanto el cliente sigue utilizando la misma interfaz para utilizar el display, reduciendo la cantidad y complejidad de los cambios necesarios para utilizar el nuevo display.

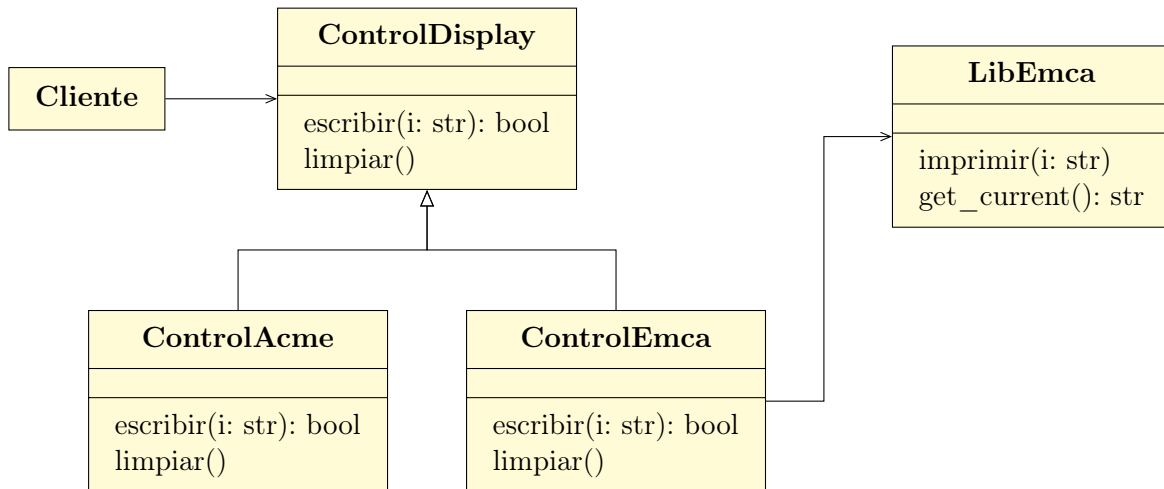
Una posible implementación para las funciones *escribir(i: str): bool* y *limpiar()* del módulo **ControlEmca** es la siguiente:

```

1 bool escribir(char* cadena) {
2     if (libEmca.get_current() != "") {
3         return False;

```

Figura 1.6: Configuración nueva



```

4     }
5     libEmca.imprimir(cadena);
6     return True;
7 }
8
9 void limpiar() {
10     libEmca.imprimir("");
11 }

```

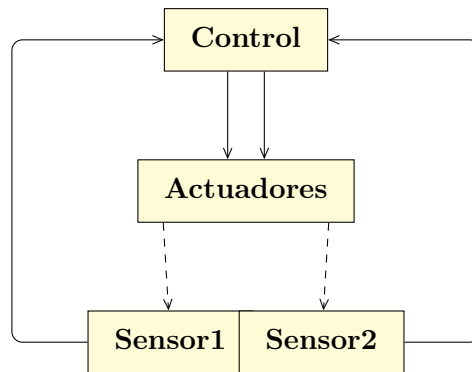
1.3 Control en conjunto de dispositivos

Muchas aplicaciones embebidas robóticas controlan actuadores que deben trabajar en conjunto para lograr el efecto deseado. Por ejemplo, para lograr el movimiento coordinado de un brazo robótico con múltiples articulaciones, todos los motores deben trabajar a la par. De manera similar, el uso de propulsores en una nave espacial en tres dimensiones requiere que muchos de estos dispositivos actúen en el momento preciso y con la cantidad correcta de fuerza para lograr la estabilización de la actitud. En ambos casos existe comunicación entre todos los componentes, ya sea para encadenar la ejecución de ciertos movimientos o para avisar de restricciones. Esto no es tarea simple y requiere de muchas líneas de código, por lo que un diseño orientado al cambio resulta clave.

Como se discutió previamente, en casos como este se puede aplicar la arquitectura de *Control de procesos*. Este hecho no resuelve todos nuestros problemas de diseño, solo nos brinda una guía y un mecanismo de funcionamiento para el sistema. Veamos que estructuras podemos usar para llevar a cabo esta arquitectura de una manera simple y probada (por ejemplo, en el robot desmalezador).

1.3.1 Subsistemas de control

Proponemos al creación de un *subsistema de control* por cada propiedad atómica¹ que sea necesaria controlar por el sistema. Como se puede intuir, un subsistema de control se encarga del control de una propiedad en particular y para hacerlo lleva a cabo todas las tareas de la arquitectura a su nivel. Por lo tanto, debe proveer una interfaz que permita setear un valor al que se quiera llevar la propiedad (*setPoint*) y que indique el comienzo de la tarea de control. Un subsistema tiene la siguiente estructura y componentes (similar a la arquitectura):



Veamos ahora de manera mas concreta qué módulos forman parte de cada componente:

¹Generalmente modificada por un actuador o por la cantidad mínima posible, por ejemplo, dada la propiedad no atómica de la posición en el plano, teniendo dos actuadores uno que modifica la posición en el eje x y otra en el, podemos dividirla en dos propiedades atómicas, posición en el eje x y posición en el eje y

La manera en la que un *cliente* utiliza el subsistema para lograr que la propiedad que controla vaya al valor que se desea es así:

```
1 control.setPoint(valorDeseado)
2 sensor.signal()
3 control.connectionRead()
4 control.control()
```

Notar que aparecen un par de módulos que no mencionamos anteriormente, por un lado tenemos *Algoritmo* el cual se encarga de el calculo necesario para determinar de que manera llamar aplicar un cambio con el/los actuadores. Por ejemplo, determina si se lleo o no al setPoint. Por otro lado, tenemos *Data* que desacopla el almacenamiento de información relacionada al subsistema. En el caso mas básico solo tenemos el setPoint, pero puede agregarse todo lo necesario, incluso llevar registro de los valores actuales y pasados.

Ahora, para brindar un comportamiento más complejo, es posible que el control necesite realizar múltiples ciclos para ajustar la variable al valor deseador, como pasa cuando trabajamos con motores paso a paso (luego veremos un ejemplo). ¿Cómo podemos hacer esto usando nuestra estructura? Necesitamos dos modificaciones/adiciones, por un lado, una interrupción de control que llame a cierta función del módulo *Control* cada determinado tiempo (tiempo del ciclo), generalmente resuelto con una interrupción *temporizada* y por el otro, necesitamos crear la noción de estado a nuestro módulo de control. Tendremos dos estados básico, en espera, cuando no se esta realizando un ciclo de control y trabajando, cuando se haya establecido un setPoint y se este trabajando para llegar. Para hacerlo utilizamos una nueva función y el patrón *state* ².

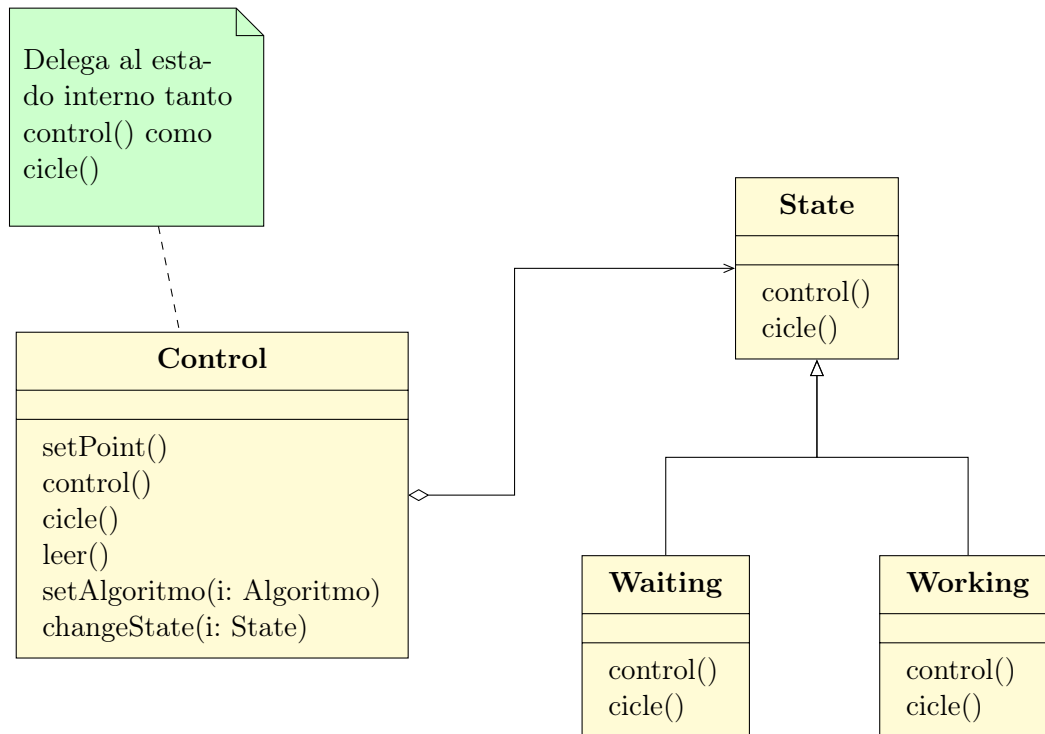
El cliente ejecuta control() y la interrupción que marca el ciclo de trabajo ejecuta cycle(). De esta manera, cuando el estado es *waiting*, cycle no hace nada y cuando esta en *working* control no hace nada y la otra función se encarga de realizar el ciclo de control. Por supuesto, estas dos funciones son las encargadas de llamar a changeState cuando sea necesario. Control cambia a *working* y cuando se alcanza el setPoint cycle devuelve el estado a *waiting*.

En el diseño del robot desmalezador existe un timer que desencadena una interrupción cada 1.5ms la cual ejecuta el ciclo de control del subsistema de dirección del robot. Este consta de un motor paso a paso, por lo que cada ciclo verifica la diferencia de posición actual con la deseada y si aun no se lleo al mínimo deseado se envía un pulso al motor para que avance.

Antes de ver un ejemplo repasemos que conseguimos al usar los subsistemas de control. Principalmente encapsulamos el control de cada propiedad de manera independiente, permitiendo que cada una pueda modificarse de manera aislada. Además, se logra que agregar nuevas propiedades con su conjunto de actuadores y sensores conste únicamente en crear nuevos módulos. Se provee una capa de abstracción para construir sobre los sistemas un controlado general (ej. MainController) el cual organizará los esfuerzos de cada uno con el fin de llevar a cabo comportamientos más complejos. Luego veremos como incluso podemos aplicar para esto el patrón *mediator*.

²La aplicación de este patrón está explicada en la sección 1.7 Máquinas de estado

Figura 1.8: Control

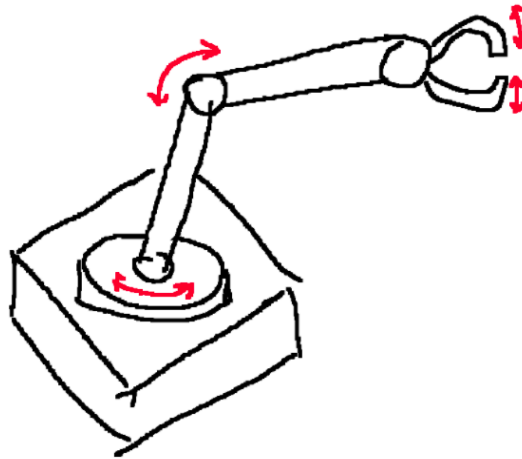


1.3.2 Ejemplo

Veamos como podemos resolver el ejemplo propuesto en el libro de Douglass en la sección de Mediator Pattern.

Requisitos:

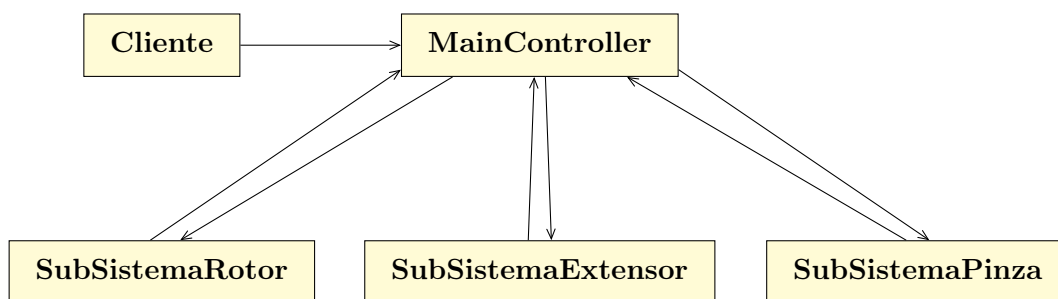
Se necesita desarrollar el software de control de un brazo robótico que consta de 3 actuadores, 2 servomotores (uno para rotar y otro para extender o retraer el brazo) y una pinza que se puede cerrar o abrir. Para ello, se provee una función compleja la cual toma coordenadas en el espacio y genera una secuencia de ordenes para que el brazo tome un objeto en la posición determinada por las coordenadas. La secuencia es una serie de pasos, y cada uno consta de una orden para cada actuador del brazo robótico. Estos se deben ejecutar de manera secuencial, es decir que el paso número 2 empezará su ejecución solo si el primero culmino completamente y con éxito. En caso de que las coordenadas sean inalcanzables devuelve 0 pasos. A su vez cada actuador puede informar un error al intentar ejecutar cada movimiento, y si esto pasa se requiere frenar la ejecución total del sistema.



Diseño:

La idea es utilizar el concepto de subsistemas que se introdujo previamente, para ello primero necesitamos identificar las propiedades del mundo físico que debemos controlar. Claramente, lo que se quiere modificar es la posición y el estado de la pinza del brazo, para hacerlo tenemos distintos actuadores que intervienen en diferentes propiedades atómicas. Además, en los requisitos nos especifican que poseemos una función que nos genera una orden para cada actuador. De esta manera definimos un subsistema de control por cada actuador que serán los encargados de llevar a cabo las ordenes generadas. Un ejemplo de orden es rotar 30° , es claro el *setPoint* que se está indicando. Para coordinar los subsistemas proponemos un controlador principal llamado *MainController*, el cual provee la función `graspAt(i: Coordenadas)` al cliente, realiza la generación de los pasos y controla su ejecución. La estructura sería algo así:

Figura 1.9: Diagrama componentes sistema brazo



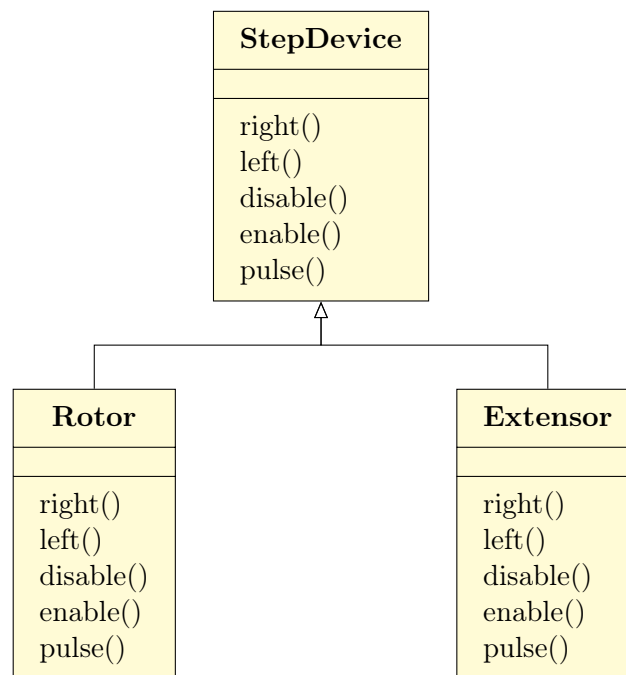
(Añadir al gráfico que significa cada flecha, `control()`, `notifyEnd()` y `onError()`)

Podemos pensar que el gráfico tiene cierta similaridad conceptual con el patrón *mediator*, donde tenemos un módulo *Mediator* que coordina el trabajo de los *College* en este caso los subsistemas. Gamma indica que el patrón suele ser apropiado para casos en lo que se tiene un conjunto de objetos que se comunican de manera compleja pero fija. En este caso la comunicación no es tan compleja, pero puede considerarse lo suficiente como para implementar el patrón. En particular, el *MainController* indica los *setPoints* de cada subsistema

y desencadena el proceso de control en cada uno. Y viceversa, los subsistemas indican cuando llegan al setPoint o cuando se encuentran un error. A su vez, los subsistemas no se comunican de manera directa entre si, todo pasa por el MainController.

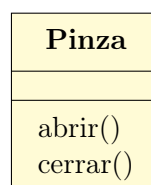
Veamos en detalle los módulos conforman cada componente comenzando por los módulos básicos que debemos crear para representar el hardware dado:

Figura 1.10: Actuadores paso a paso



Estos son motores paso a paso ³, lo cuales para controlarlos debemos primero setear su dirección llamando a las funciones **right** o **left** para luego avanzar un paso llamando a la función pulse. Claramente para que lleguen a la posición deseada puede ser necesario invocar reiteradas veces la función pulse. Esto será importante a la hora de diseñar el subsistema que controlará cada dispositivo.

En cambio, en el caso de la pinza se utiliza un dispositivo que tiene dos estados, abierto o cerrado, por lo que solo tenemos dos funciones para cambiar entre los estados.

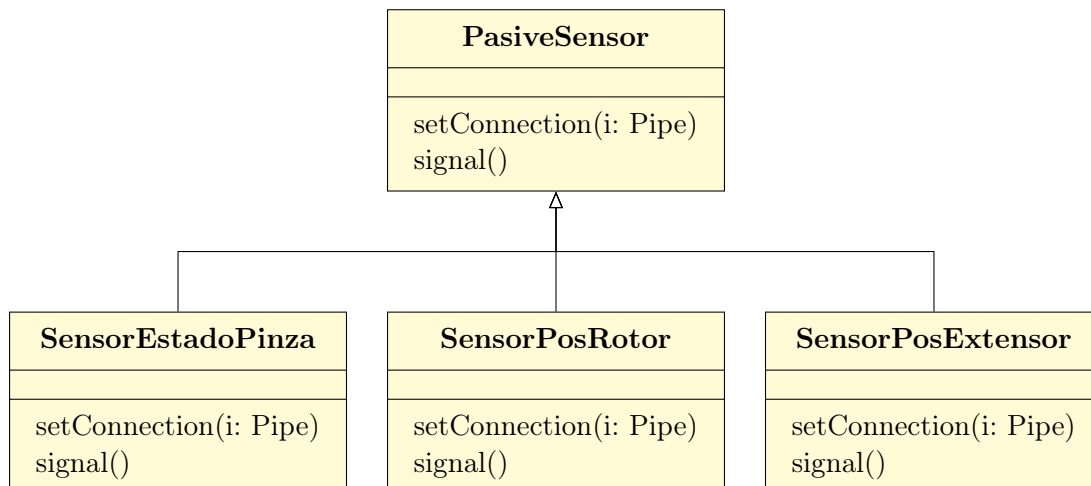


Vemos ahora los sensores asociados a cada actuador, estos heredaran del módulo sensor pasivo el cual provee dos funciones, **setConnection(i: Pipe)** la cual configura el Pipe por

³Se utiliza el mismo diseño propuesto para el robot desmalezador.

el cual se enviará la información obtenida del sensor cuando la otra funciones, `signal()` sea invocada.

Figura 1.11: Sensores



Para completar los módulos que conforman a un subsistema de control nos falta mostrar como se define el Controller. Vamos a diferenciar dos tipos de Controllers, uno para los sensores que requieren un control durante un determinado lapso de tiempo para poder llegar a su `setPoint` y los que no. En este ejemplo, tenemos dos motores paso a paso que representan el primer tipo y la pinza que corresponde al segundo. Vemos primero el caso más “complejo”, como el proceso de control se extiende en el tiempo necesitamos añadir al sistema un mecanismo por el cual cada cierto intervalo ejecute un ciclo de control, es decir, lea la posición actual, decida y actúe. Para eso, en el diseño del robot desmalezador se propuso la creación de una interrupción temporizada, la cual se ejecuta cada 1.5ms y desencadena el control de los subestimas necesarios.

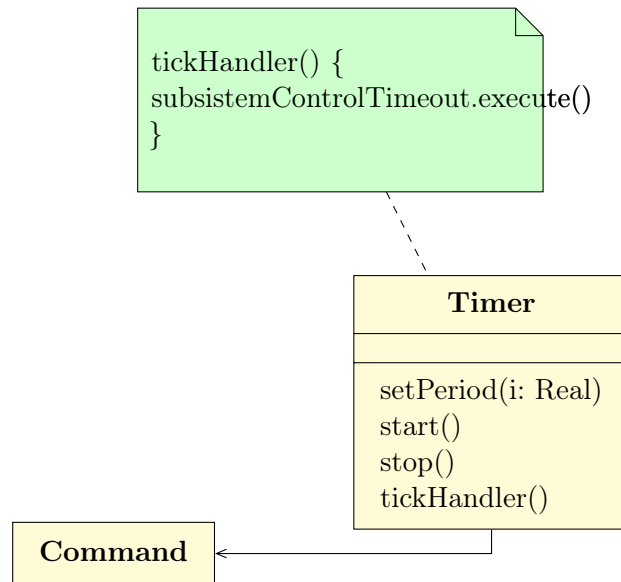
La función `tickHandler()` ejecutara *Commands* por cada subsistema que lo necesite, por lo tanto se iniciará en cada uno el ciclo de control. En particular, la orden la usamos para desacoplar como triggerear el comienzo del Timer.

Una vez solucionada la invocación, tenemos que introducir un nuevo módulo que será usado en el Controller. Ahora, tenemos dos modos de operación de este ultimo, cuando no se está trabajando y cuando si. Dependiendo de cual de los dos esté activo el comportamiento será diferente, ya que por ejemplo, no apagamos el Timer cuando no se está haciendo un movimiento, sino que cuando este triggeree la ejecución del ciclo el Controller ignorará la orden. Para esto hacemos uso del patrón *State* de Gamma, el cual está explicado en la tesina. En particular, el Controller delegará dos funciones de su interfaz al state, `control()` y `move()` (luego lo veremos en más detalle).

Siguiendo con la guía de creación de subsistemas, también debemos definir el módulo Data que almacena la información utilizada por el mismo y el módulo Algoritmo que encapsula los cálculos necesarios para determinar que cambio aplicar.

Ahora si tenemos todos los módulos necesarios para mostrar nuestro Controller.

Figura 1.12: Timer

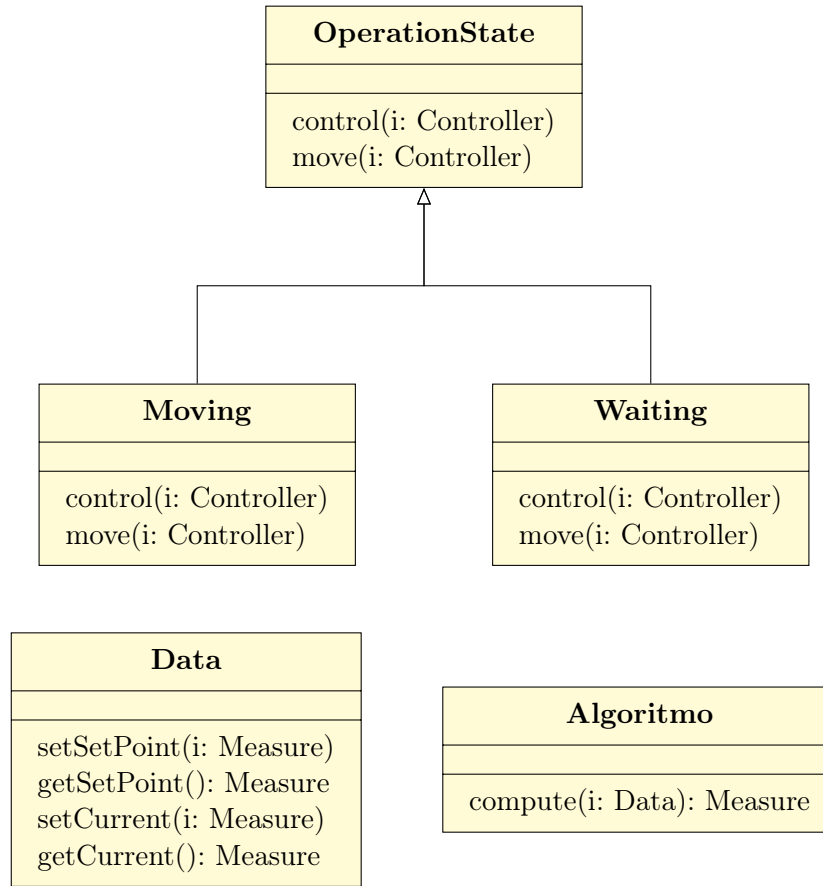


Veamos la funcionalidad de cada método que provee la interfaz de Controller:

- `setConnection`: configura el tubo por el cual llegará la información proveniente de el/los sensores.
- `readConnection`: lee del pipe y almacenar la información en el módulo data.
- `setSetPoint`: se utiliza antes de comenzar el control para configurar el valor al que se quiere llegar.
- `getData`: devuelve el objeto Data asociado al Controller.
- `getAlgoritmo`: lo mismo que con Data.
- `changeOperationState`: cambia el estado del modo de operación, en la configuración básica las transiciones posibles son de *Waiting* a *Moving* y viceversa.
- `move`: es la función ejecutada en cada timeout del timer de control, realiza un paso del ciclo de control si el estado es *Moving*.
- `control`: se utiliza para comenzar el ciclo de control, es el puntapié inicial que desencadena el comportamiento de todo el subsistema.

Imaginemos que un cliente quiere utilizar el subsistema, qué métodos tiene que ejecutar si ya está configurado? Primero debe indicarle a los sensores que escriban el valor de lectura en el pipe, para esto hay que ejecutar la función **signal** en cada uno. Luego el Controller debe leer esta información del pipe y almacenarla, para esto ejecutamos **readConnection**. Ahora, seteamos el `setPoint` deseado y por ultimo ejecutamos **control**, la cual tomará los

Figura 1.13: Módulos necesarios para complementar el Controller

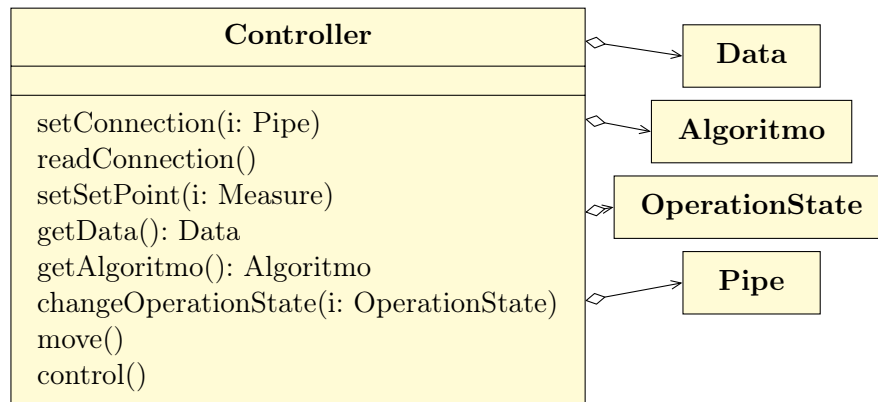


valores actuales, el `setPoint` y decidirá utilizando el módulo **Algoritmo** que cambio realizar en los actuadores. En el caso del Rotor, por ejemplo, podrá tanto cambiar la dirección de giro, como avanzar un paso ejecutando `pulse`. Un ejemplo de función `control` para el caso del subsistema del Rotor y estado de operación `Waiting`, puede ser la siguiente (recordar que el `setPoint` del subsistema del rotor es una medida en grados):

```

1  void control(Controller controller) {
2      setPoint = data.getSetPoint();
3      current = data.getCurrent();
4      dif = controller.algoritmo.calculate()
5      if abs(dif) > LIMIT_ACCEPT && dif > 0 {
6          rotor.left();
7          rotor.pulse;
8          controller.changeOperationState(moving);
9          return;
10     }
11     if abs(dif) > LIMIT_ACCEPT && dif < 0 {
12         rotor.right();
  
```


Figura 1.14: Controller



```

13         rotor.pulse;
14         controller.changeOperationState(moving);
15         return;
16     }
17 }
  
```

La función `move` tendrá un comportamiento similar pero en el uso de cambiar de estado lo hará a `Waiting`. Para el caso de la Pinza, el `Controller` es más simple ya que no necesitamos los estados, pero el uso y comportamiento es similar. Por lo que para este ejemplo, vamos a necesitar crear dos subsistemas con estados y uno sin.

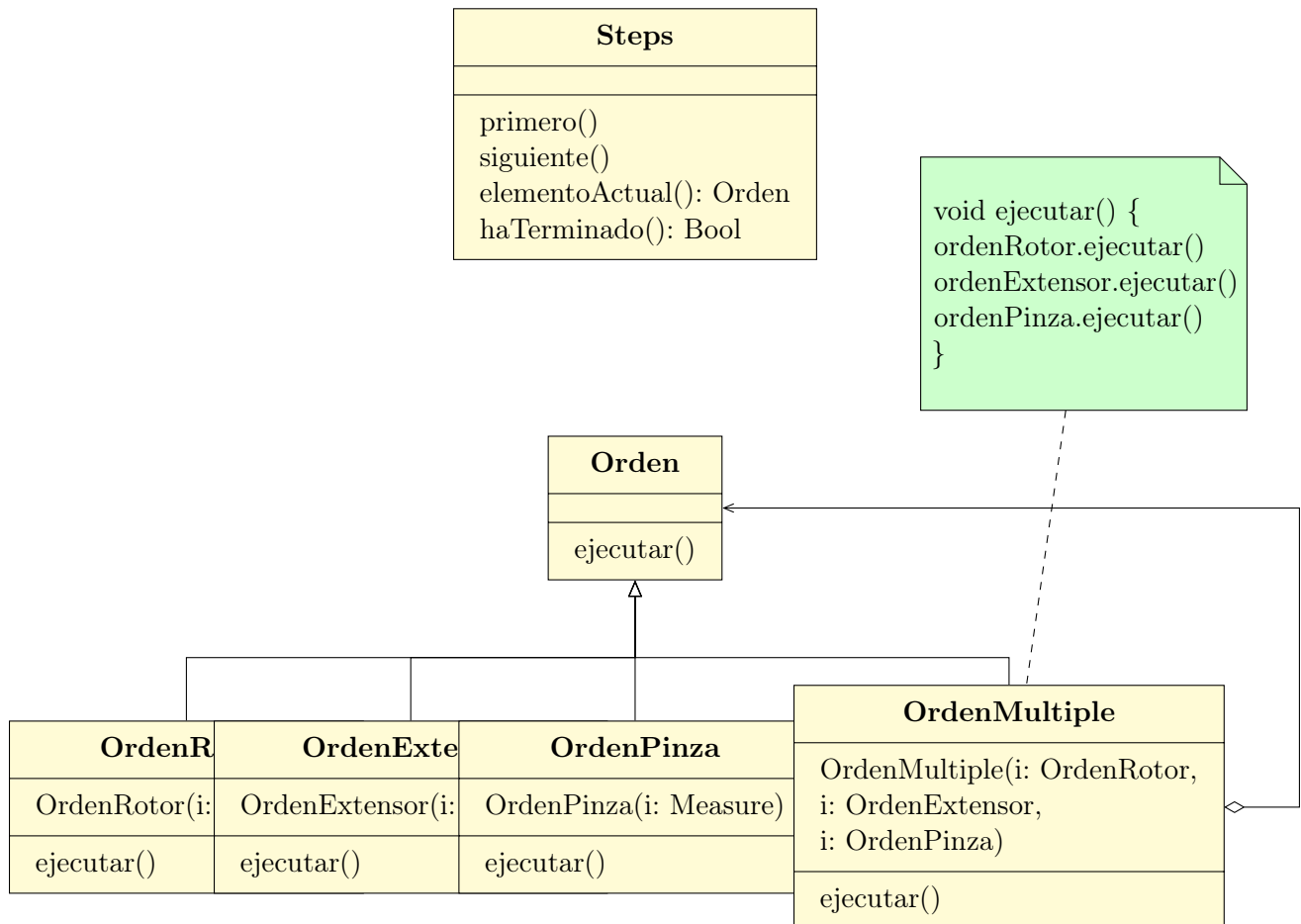
Ya tenemos los subsistemas, por lo que siguiendo con nuestro diagrama, nos falta la parte del `MainController`. El cual será el encargado de brindar una interfaz al cliente y a su vez coordinar cada subsistema. Recordando los requerimientos, sabemos que tenemos una función que genere una secuencia de pasos donde cada uno implica una acción sobre los tres actuadores. ¿Cómo podemos diseñar este comportamiento? Se me ocurre usar un *iterator* para recorrer los pasos y usar el patrón *command* para cada uno. En particular, aplicaremos una de las modificaciones del patrón mencionada por Gamma, en donde una orden al ser ejecutada desencadena una ejecución del resto de las ordenes. Veámoslo en el siguiente gráfico:

Tenemos un iterador de ordenes, en particular múltiples, que almacena todos los pasos de ejecución que la función provista generará. Por lo que el `MainController` podrá recorrerlos ejecutando cada orden de manera sencilla. Logramos, almacenar el procedimiento a realizar y desacoplar como se pone en marcha el ciclo de control en cada subsistema.

Hasta la introducción de `MainController` todo lo que estábamos haciendo era solo seguir con el concepto de subsistemas. Pero para poder cumplir los requerimientos particulares del ejemplo, necesitamos introducir ciertos cambios. Como necesitamos ejecutar un paso a la vez, tenemos que esperar que todas las ordenes que enviamos en el paso anterior a los subsistemas. Para ello hacemos uso del patrón *state*, al cual delegaremos las funciones `graspAt` y `notifyReady`. La idea es cambiar el comportamiento de estas dependiendo de en qué estadio estamos, seguiremos el siguiente gráfico:

Por lo que, el estado `Waiting` no implementa `notifyOrder` y a su vez 0, 1 y 2 no

Figura 1.15: Iterator y ordenes



implementan **graspAt**. En cambio, **Waiting** implementan **graspAt** la cual computa los pasos y ejecuta el primero utilizando el iterator. Luego, por cada **notifyEnd** los subsistemas le avisan al **MainController** que terminaron la orden y por cada una transicionamos a un estado nuevo. Luego de que se ejecuta **graspAt** se transiciona al estado 0 en el cual al recibir un **notifyEnd** se transiciona a 1 y lo mismo hasta llegar a 2. En cambio, cuando estamos en el estado 2 y se ejecuta **notifyEnd** lo que se hará es ejecutar un nuevo paso si en el iterator de pasos quedan pasos, en caso contrario se transicionará a **Waiting** ya que se terminó la ejecución. Está claro que para poder llevar a cabo este comportamiento es necesario que los subsistemas respondan a las necesidades. Por lo que cada uno tendrá que llamar a la función **notifyEnd** del **MainController** cuando efectivamente terminen la ejecución de la orden, es decir, el ciclo de control. Para hacerlo se puede agregar la llamada en la cuando se decide transicionar de **Moving** a **Waiting** o en el caso de los subsistemas sin estados, luego de aplicar los cambios en los actuadores.

Listing 1.8: Ejemplo OrdenRotor

```

1 void ejecutar {
2     rotorController.setSetPoint(setPoint)
3     rotorSensor.signal()
4     rotorController.readConnection()
5     rotorController.control()
6 }

```

1.4 Obtención de información

Generalmente una tarea importante que tienen los sistemas embebidos es recavar información proveniente de sensores. Existen diferentes formas en las que los sensores transmiten información al sistema, algunos, por ejemplo un sensor de temperatura, setea en el pin en el que esta conectado un cierto valor de tensión, por lo que el sistema solo debe consultar el valor del pin. Otros, en cambio, se comunican mediante interrupciones, por ejemplo un sensor de efecto **Hall** genera una interrupción por cada detección de campo magnético. Por lo tanto, si lo estamos usando para calcular las **RPM** de cierto componente giratorio, debemos llevar una cuenta de las interrupciones que generó en cierto periodo de tiempo y realizar cierta matemática. Evidentemente, es necesario que alguna porción de nuestro sistema se encargue de hacerlo y maneje las interrupciones generadas por el sensor. Algo similar pasa con otro tipos de sensores como joysticks, botones, etc. Es por esto que resulta útil una forma general de atacar este problema, en el diseño del robot desmalezador, Laura utilizo una estructura de módulos que lleva a cabo las actividades necesarias para integrar al sistema los sensores que generan interrupciones.

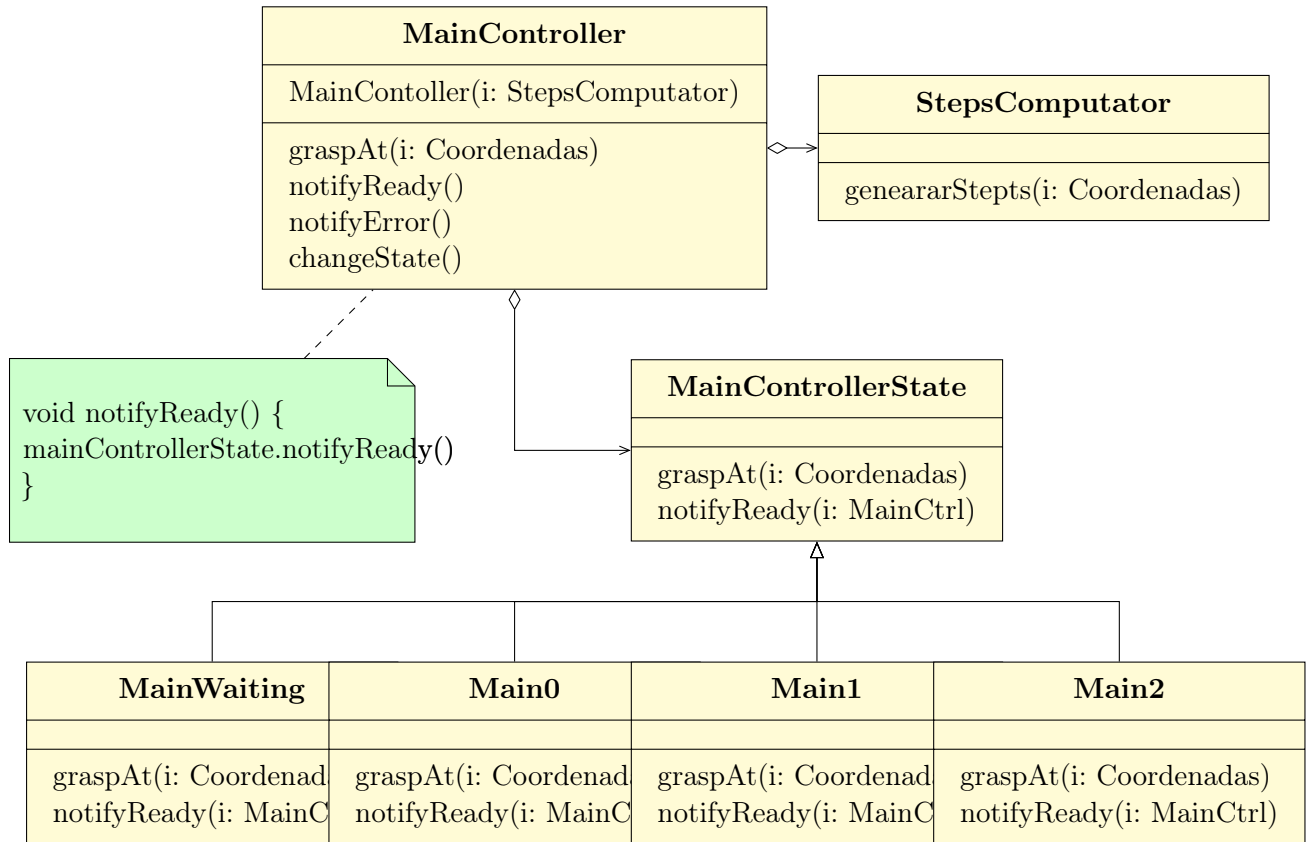
Podemos distinguir 4 módulos principales, **Sensor** que es la representación en el diseño del hardware asociado al dispositivo de hardware que sensa, este genera la interrupción que es manejada por **Count** que es un módulo perteneciente al patrón *Command* el cual cumple la función de comando para reemplazar la utilización de *callbacks*. En particular, opera sobre **Collector** almacenando en este la información relevante del evento (interrupción). Esta por lo general es un *timestamp* y la cantidad de veces que sucedió. Por ultimo, **Buffer** actúa como buffer de la información recibida a través, posee una función que permite obtener el valor recibido. Cuando es invocada toma los valores necesarios que se almacenan en el **Collector** y realiza los cálculos necesarios para calcular el dato pedido.

Veamos esta estructura más en detalle con un ejemplo de aplicación. Supongamos que tenemos un sensor *Hall* montado en una rueda y con este queremos medir la velocidad a la que se está moviendo. Por lo tanto, cada vez que el sensor detecte un campo magnético emitirá una interrupción que será manejada por el módulo **Count** que es un comando, por lo que sabe que funciones ejecutar del módulo **Collector**. **Count** tiene la siguiente interfaz:

- `execute()` registra en el acumulador el instante actual, mediante `Collector::currentTime`, y luego incrementa el contador de interrupciones invocando `Collector::addOne`.

Y la de el módulo **Collector**:

Figura 1.16: MainController



- `currentTime()`: registra internamente el momento actual de cuando el método es invocado.
- `getCurrentTime()`: devuelve el último instante de tiempo registrado por el módulo.
- `addOne()`: incrementa el contador interno que cuenta las ocurrencias de interrupción del sensor *Hall*.
- `total()`: retorna el valor del contador interno.

Por lo tanto cuando se pida el valor de velocidad al **Buffer** este llamará a las funciones necesarias del **Collector** para obtener información, computará el valor y lo retornará.

De esta forma un cliente de nuestro componente puede obtener la información recibida a través del pin sin necesidad de manejar las interrupciones y todas las operaciones asociadas al tratamiento de los datos.

Por ejemplo, si tenemos un sensor de efecto Hall que emite una interrupción en cada detección del campo magnético y lo usamos para medir la velocidad de una rueda, podemos utilizar el siguiente diseño (obviamente, aplicando lo que acabamos de ver):

Periodicidad

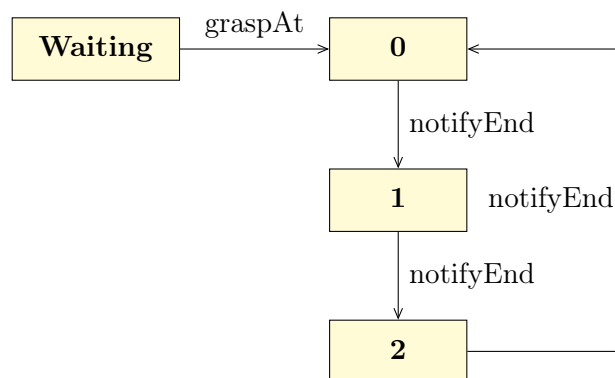
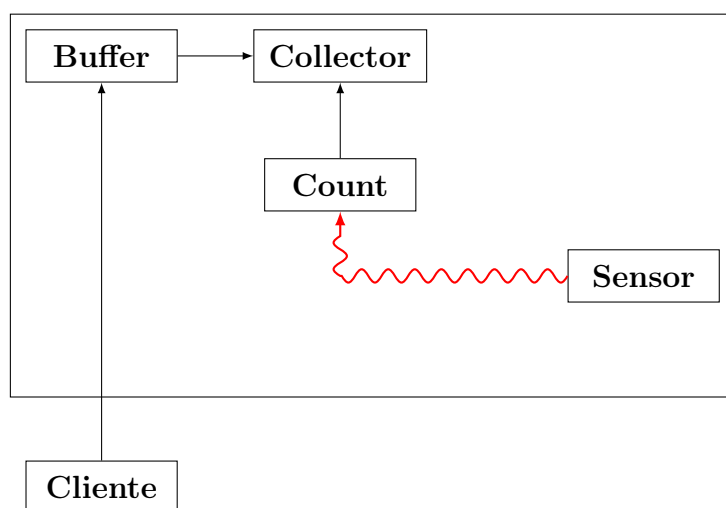



Figura 1.17: Componente sensor activo.

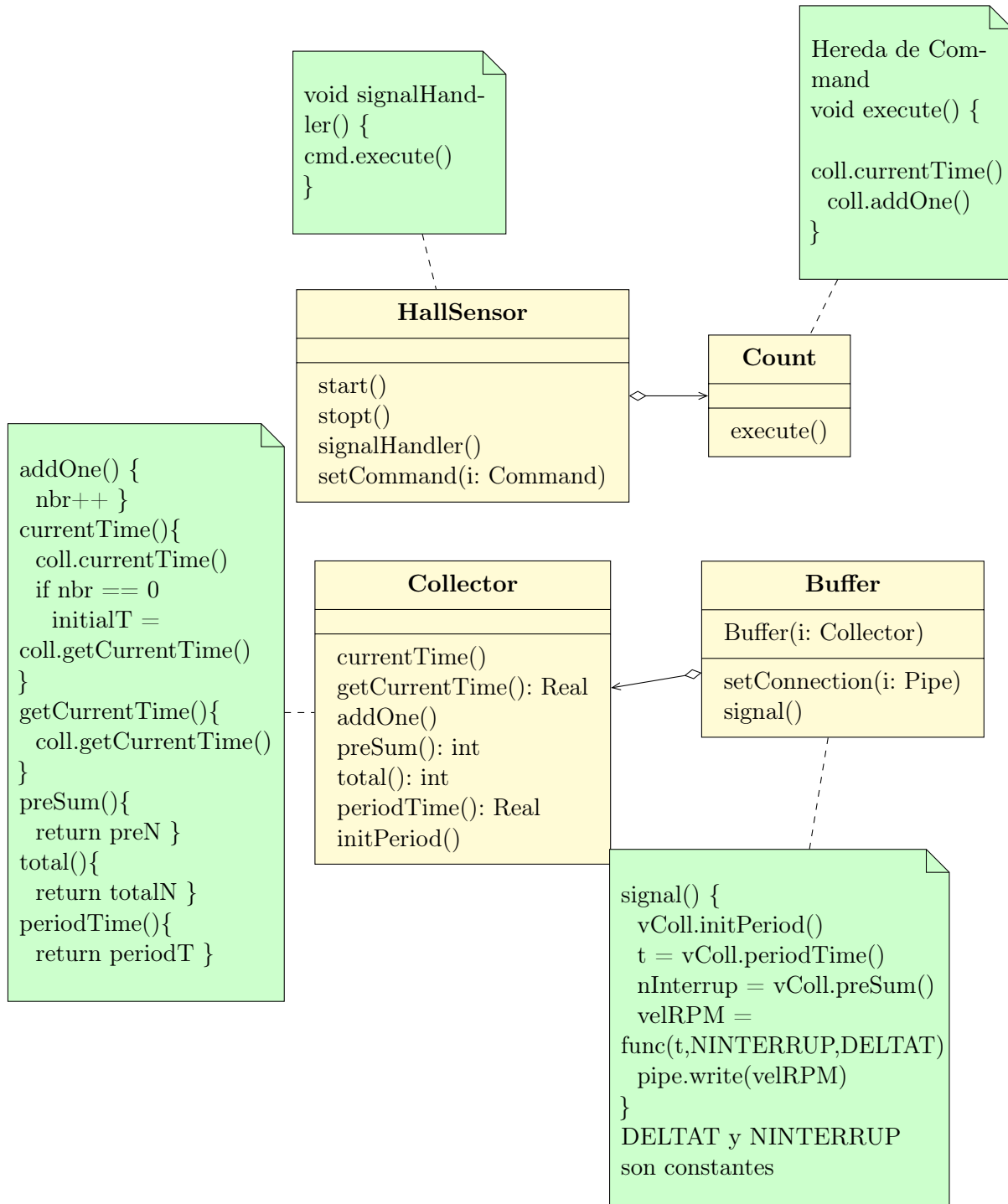


nombre	módulo	→	llamada a procedimiento
	llamada a procedimiento a partir de una interrupción física (handler)	- - - - -	conexión física

Algunos sistemas se encargan de mostrar, almacenar o verificar información de sensores cada cierto tiempo. No resulta crítico perder valores intermedios, es decir que no nos interesa una respuesta inmediata. Un ejemplo puede ser una estación meteorológica o un dispositivo médico de control como un tensiómetro que registra los valores de presión del paciente cada cierto periodo de tiempo.

Una implementación intuitiva es escribir un *loop* y en el que verificamos la información de los sensores y luego ejecutamos una función *sleep* que bloquea la ejecución determinada cantidad de tiempo. Esto tiene varias desventajas, primero el tiempo de ejecución de nuestra

Figura 1.18: Ejemplo



rutina alarga el periodo, si queremos agregar otras funcionalidades a realizar mientras esperamos el periodo tendríamos que dividir el *sleep* y calcular tiempo de ejecución.

Una solución que nos parece más adecuada desde el punto de vista del diseño orientado al cambio es la siguiente. Se registra un temporizador que cada x cantidad de tiempo gatilla una interrupción, esto es provisto por muchos entornos de desarrollo para sistemas embebidos como *Arduino*. Registraremos un *handler* para esta que será un módulo orden del patrón *Command* el cual encapsula que funciones son necesarias ejecutar para llevar a cabo la funcionalidad deseada. Por ejemplo, pedir la información de los sensores, ya sea a sus respectivos **Buffers** (si tenemos sensores como en el ejemplo anterior) o a los sensores en si, si son de otro tipo y luego registrarla en cierta módulo (que oculte una estructura de datos). De esta forma liberamos el procesador en los tiempos de espera y desacoplamos esta funcionalidad.

1.5 Control anti-rebote

Muchos dispositivos de entrada para sistemas embebidos utilizan contacto metal con metal para indicar eventos de interés, como botones, interruptores y relés. A medida que el metal entra en contacto, se produce una deformación física que resulta en un contacto intermitente de las superficies. Esto genera señales que de no ser filtradas pueden causar una lectura errónea.

En la bibliografía encontramos una propuesta de solución que desde mi punto de vista está más cerca de la implementación que del diseño. Esta consiste en utilizar un temporizador para dar un tiempo de gracia antes de confirmar una transición en el estado. Esto es, al percibir un cambio en la entrada se inicia el *timer* y se confirma el cambio de estado solo si el valor de la entrada sigue siendo distinto que antes de percibir la modificación.

Listing 1.9: Código ejemplo

```
1 void ButtonDriver_eventReceive(ButtonDriver* const me) {
2     Timer_delay(me->itsTimer, DEBOUNCE_TIME);
3     if (Button_getState(me->itsButton) != me->oldState) {
4         /* must be a valid button event */
5
6         me->oldState = me->itsButton->deviceState;
7
8         if (!me->oldState) {
9
10            /* must be a button release, so update toggle value */
11            if (me->toggleOn) {
12                me->toggleOn = 0; /* toggle it off */
13                Button_backlight(me->itsButton, 0);
14                MicrowaveEmitter_stopEmitting(me->itsMicrowaveEmitter
15                );
16            }
17            else {
18                me->toggleOn = 1; /* toggle it on */
19                Button_backlight(me->itsButton, 1);
20            }
21        }
22    }
23 }
```

```

19         MicrowaveEmitter_startEmitting(me->
                itsMicrowaveEmitter);
20     }
21 }
22 /* if i t s not a button release, then it must
23    be a button push, which we ignore.
24 */
25 }
26 }

```

Desde el punto de vista de la IS, creo que se están combinando múltiples responsabilidades en un mismo módulo, lo cual no ayuda a lograr el objetivo de diseñar para el cambio. El estado y la decisión de transicionar son responsabilidad de un módulo, lo que puede generar inconvenientes si se quiere cambiar el criterio de aceptación de la señal, por ejemplo. Esto se evidencia en el código ejemplo del libro de Douglass donde tenemos múltiples sentencias *if* anidadas. Una solución que se ajusta más a nuestros principios involucraría el patrón *State* de Gamma y otro módulo que cumpla la función de verificar el cambio.

Parece ser un problema vigente, se habla mucho en internet sobre eso, por ejemplo *A Guide to Debouncing* de Jack G. Ganssle.

1.6 Manejo de interrupciones

Algunos microcontroladores como **Arduino** proveen en su entorno de desarrollo herramientas para manejar interrupciones. En particular, permiten asociar una función (*handler*) a un cierto evento por ejemplo un cambio en un pin digital.

1.7 Máquinas de estado

Muchos sistemas ajustan su comportamiento durante la ejecución en base a diferentes causas como interacciones externas, su propios requerimientos, etc. Por ejemplo, el sistema de control de un microondas no iniciará el calentamiento si la puerta está abierta, es decir que existe un estado interno en el que el microondas está al tener la puerta abierta y no permite cierto comportamiento que en otro estado si sería posible. De manera similar, podemos pensar que algunas partes de los sistemas responden a diferentes estados. Un ejemplo sería la dirección de giro de un motor en un robot, que podría representarse como un estado: si el motor está en un estado de "girar hacia adelante." "girar hacia atrás", su comportamiento cambia en consecuencia. Además, si el sistema está llevando a cabo una operación, su estado podría ser *trabajando*, y al finalizar, podría retornar al estado *esperando*, lo que también modificaría su comportamiento. Otro ejemplo es un sistema de climatización: el estado del termostato podría ser *calentando* o *enfriando*, dependiendo de lo configurado por el usuario. Por lo tanto, es común enfrentarse a la gestión de estados, transiciones y todos los cambios en el comportamiento que resultan de estos.

Una solución de implementación intuitiva para aplicar estados es almacenarlos en una variable y verificar esta última para cambiar el comportamiento de las funciones. Por ejemplo:


```

1 calentar() {
2     if (estado == PuertaAbierta) {
3         return
4     } else {
5         magnetron.encender()
6     }
7 }

```

Este enfoque de es una solución directa. Sin embargo, presenta algunas desventajas cuando se desea modificar o extender el código. Por ejemplo, si se necesita agregar nuevos estados o cambiar el comportamiento del sistema, este método puede volverse rápidamente difícil de gestionar. Una de las principales desventajas es la complejidad creciente. A medida que se agregan más estados, cada función requerirá más verificaciones de estado, lo que llevará a una proliferación de estructuras *if-else* o *switch-case*. Esto no solo hace que el código sea más difícil de leer, sino que también aumenta la probabilidad de errores, ya que cada nuevo estado debe ser cuidadosamente incorporado en todas las partes relevantes del sistema. Además, este enfoque dificulta la sostenibilidad, cuando el comportamiento de un estado debe cambiar, es posible que se necesiten modificaciones en varias funciones. Esto puede resultar en código duplicado, o incluso puede ser difícil saber que modificar, lo que complica el proceso de realizar cambios sin introducir errores. Otra desventaja es la poca modularidad, no es claro el comportamiento asociado a cada estado. Si cada uno requiere comportamientos más complejos, el código se vuelve monolítico y difícil de extender sin alterar funciones existentes. Por ejemplo, si se agregara un estado “EnPausa”, se tendría que modificar la lógica de muchas funciones para verificar este nuevo estado y ajustar el comportamiento de manera adecuada. Todas estas desventajas evidentemente conllevan a que reutilizar el código sea difícil.

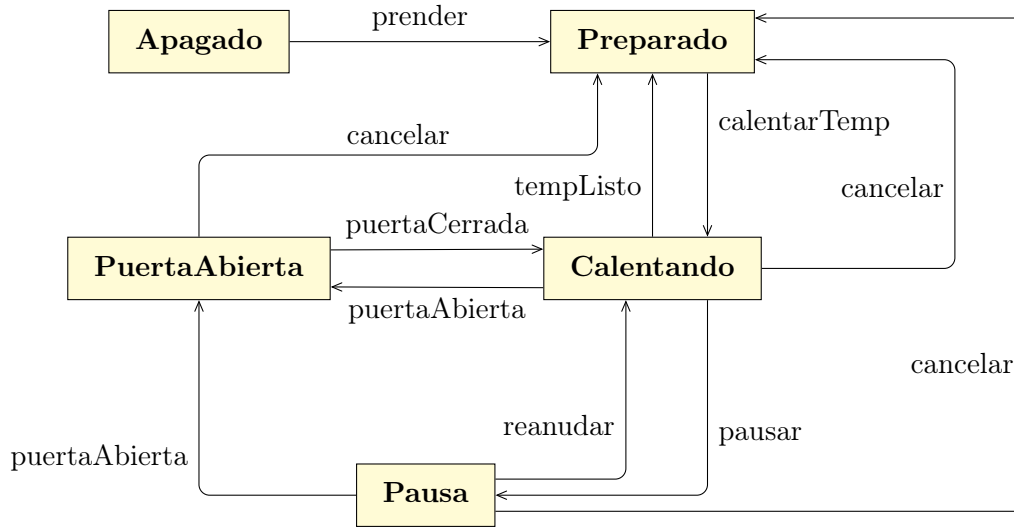
Antes de ver un enfoque desde el punto de vista del diseño orientado al cambio, un poco mas en profundidad el ejemplo planteado en el libro y como se aplica una solución similar a la nombrada.

Transiciones basadas en eventos. Supongamos un sistema en donde las transiciones entre los estados son dadas por eventos particulares, por ejemplo si trabajamos con el sistema de un microondas, abrir la puerta sería un evento que desencadena una transición, apretar el botón de Cancelar mientras está calentando también. Estamos hablando de un sistema con característica de máquina de estados, en el cual el comportamiento se basa en una sucesión de transiciones. Considere el siguiente “*state chart*” extraído del libro que describe el funcionamiento del microondas:

Todo el funcionamiento principal del sistema es representado utilizando estados y transiciones, en cada transición se realizan cierto comportamiento relacionado con la salida del estado actual y la llegada al siguiente. Por ejemplo, si se está en el estado *PuertaAbierta* y se recibe el evento *puertaCerrada* se transicionará al estado *Calentando*. En el proceso se encenderá el magnetrón para que este emita las ondas que finalmente calienten la comida.

A simple vista parece un comportamiento complejo y eso que solo es un ejemplo simplificado, por lo que en la vida real encontraremos casos mucho más extensos. Este es uno de los motivos por los cuales es útil contar con un buen diseño. Debemos tener en cuenta que sea fácil modificar las transiciones y también agregar y quitar estados. Doublass construye un módulo encargado de mantener el estado, el cual que sabe qué funciones ejecutar cuando se da una transición. Esto es, recibe el evento, verifica si corresponde a un cambio de estado y de ser

Figura 1.19: State chart microondas



así ejecuta la función de “salida” del estado actual, ejecuta la función de “entrada” del nuevo estado y actualiza el valor del estado actual. Además, propone una implementación utilizando una tabla bidimensional lo cual lo hace un sistema eficiente computacional hablando. En el caso de querer modificar estados, agregarlos o quitarlos es necesario cambiar la implementación de este módulo. Gamma explica esto en la sección implementación del patrón *State*, menciona que Cargill propuso esta implementación y efectivamente es una manera de convertir código condicional en algo que se parece a una tabla.

```

1 void TokenizerStateTable_eventDispatch(TokenizerStateTable* const me,
    Event e) {
2     int takeTransition = 0;
3     Mutex_lock(me->itsMutex);
4     /* first ensure the entry is within the table boundaries */
5     if (me->stateID >= NULL_STATE && me->stateID <=
        GN_PROCESSINGFRACTIONALPART_STATE) {
6         if (e.eType >= EVDIGIT && e.eType <= EVENDOFSTRING) {
7             /* is there a valid transition for the current state and
                event? */
8             if (me->table[me->stateID][e.eType].newState !=
                NULL_STATE) {
9                 /* is there a guard? */
10                if (me->table[me->stateID][e.eType].guardPtr == NULL)
11                    /* is the guard TRUE? */
12                    takeTransition = TRUE; /* if no guard, then it "
                        evaluates" to TRUE */
13            }
14            else
                takeTransition = (me->table[me->stateID][e.eType].
                    guardPtr(me));

```

```

15         if (takeTransition) {
16             if (me->table[me->stateID][e.eType].exitActionPtr
17                 != NULL)
18                 if (me->table[me->stateID][e.eType].
19                     exitActionPtr->nParams == 0)
20                     me->table[me->stateID][e.eType].
21                         exitActionPtr->aPtr.a0(me);
22             else
23                 me->table[me->stateID][e.eType].
24                     exitActionPtr->aPtr.a1(me, e.ed.c);
25         if (me->table[me->stateID][e.eType].
26             transActionPtr != NULL)
27             if (me->table[me->stateID][e.eType].
28                 transActionPtr->nParams == 0)
29                 me->table[me->stateID][e.eType].
30                     transActionPtr->aPtr.a0(me);
31             else
32                 me->table[me->stateID][e.eType].
33                     transActionPtr->aPtr.a1(me, e.ed.c);
34         if (me->table[me->stateID][e.eType].
35             entryActionPtr != NULL)
36             if (me->table[me->stateID][e.eType].
37                 entryActionPtr->nParams == 0)
38                 me->table[me->stateID][e.eType].
39                     entryActionPtr->aPtr.a0(me);
40             else
41                 me->table[me->stateID][e.eType].
42                     entryActionPtr->aPtr.a1(me, e.ed.c);
43         me->stateID = me->table[me->stateID][e.eType].
44             newState;
45     }
46 }
47 }
48 }
49
50 Mutex_release(me->itsMutex);
51 }

```

En este ejemplo de código, que es una porción de la solución propuesta en el libro, podemos ver que aunque el enfoque cumple los requerimientos, el código no parece utilizar buenas prácticas y hace empleo de múltiples sentencias *if* anidadas, lo cual complejiza el entendimiento y modificación del mismo. Por otro lado, las estructuras de datos son un item de cambio común, nuestras interfaces tienen que intentar ser independientes de la estructura de datos que se utiliza para implementarlas.

Como el otro enfoque, proponemos utilizar el patrón *State* de Gamma con ciertos ajustes para permitir a los estados transicionar a otros por si mismos. Básicamente, el patrón establece la creación de un módulo por cada estado del sistema con el fin de cambiar la implementación

para que se ajuste al comportamiento cuando el sistema se encuentra en dicho estado. Esto nos permite transicionar dinámicamente entre estados, ya que cambiar de estado es usar otro módulo que hereda la interfaz.

Para permitir que cada estado pueda transicionar de manera independiente a otros estados lo que hacemos es agregar una referencia de los posibles siguientes estados, para que en caso de recibir el evento adecuado pueda invocar la función de cambiar estado pasando la referencia del nuevo. Por lo tanto, al constructor de cada estado hay que agregarle como argumento cada posible estado siguiente. Esto lo podemos ver en la siguiente porción de código perteneciente al diseño del robot desmalezador:

```
1 buildOpStates(){
2     MAX=50
3     workSt=new Working(mCtrlOrd)
4     recSt=new Reconnecting(workSt, mCtrlOrd)
5     waitSt=new WaitingMAX(workSt,recSt,ctrlsStopOrd,mCtrlOrd)
6     nMax=MAX-1
7     while nMax > 0
8         temp=new WaitingN(workSt,waitSt,mCtrlOrd)
9         waitSt=temp
10        nMax--
11    workSt.setNextState(waitSt)
12    opState=workSt
13 }
```

Y para llamar a una transición de estado:

```
1 actionWithMsg(MainController mCtrl, Mode md){
2     md.read(mCtrl)
3     mCtrl.changeState(workSt) // workSt class attribute
4 }
```

Este uso del patrón *State* también es propuesto en *Chapter 10 : Finite State Machine Patterns Part III: New Patterns as Design Components*.

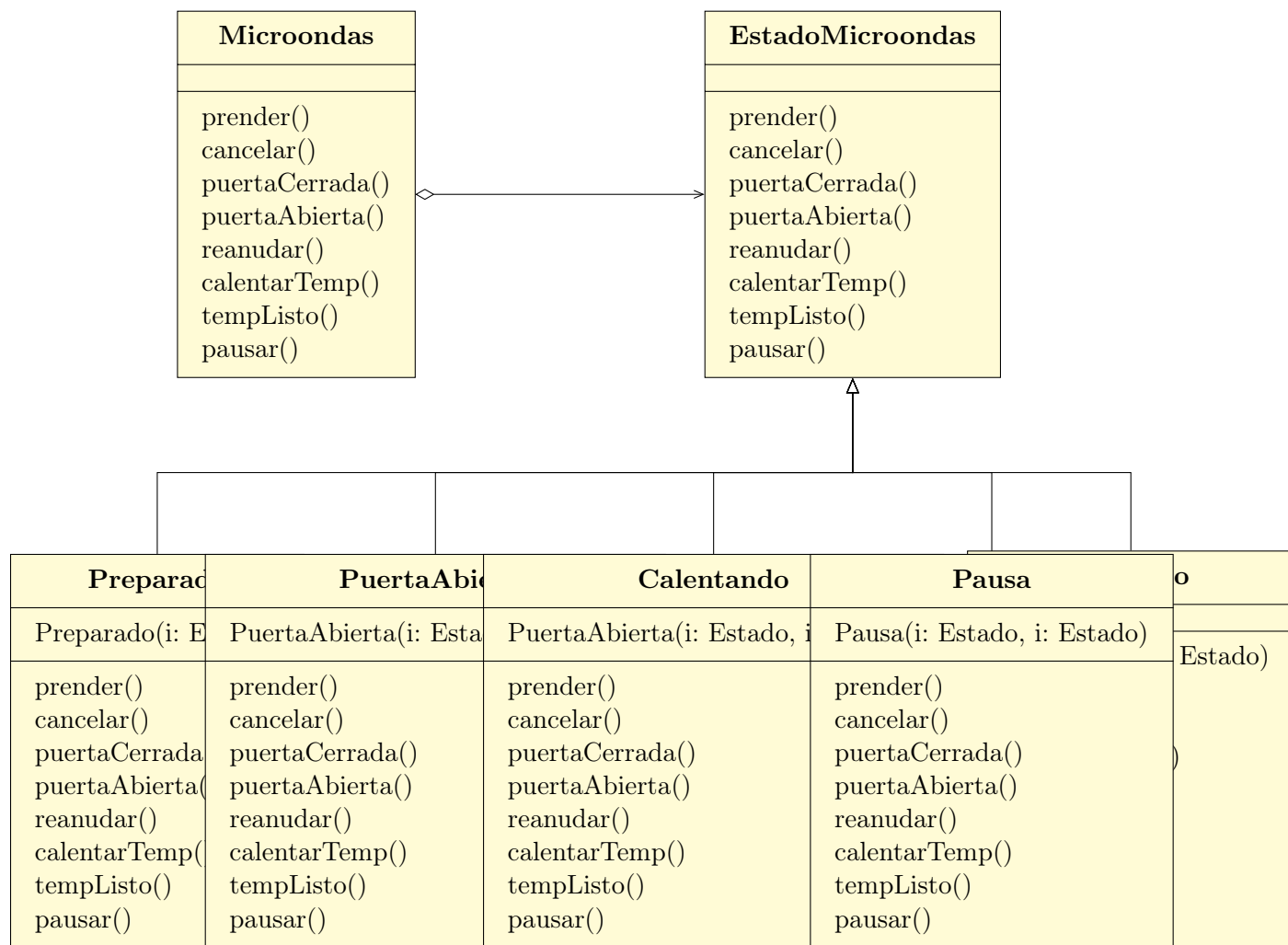
Volviendo al ejemplo del microondas, aplicando este enfoque de diseño, tendremos un módulo que provee una función que maneja cada evento y otro que delega al estado cada manejo de evento.

Por lo tanto el módulo PuertaAbierta implementará la función `puertaAbierta`, completando en ella el comportamiento definido para la transición PuertaAbierta → Calentando.

```
1 puertaAbierta() {
2     magnetron.calentar()
3     microondas.changeEstate(calentando)
4 }
```

En cambio, en el caso del evento *prender* no hará nada, pues no existe transición posible desde PuertaAbierta que involucre al evento *prender*.

De esta forma, logramos una solución más elegante y escalable para manejar estados, superando muchas de las limitaciones del enfoque basado en verificaciones de estado dentro de las funciones. Una de las principales ventajas de este patrón es que mejora significativamente la legibilidad y modularidad del código. En lugar de manejar el comportamiento de todos los



estados en una misma función, cada estado tiene su propia clase, lo que permite que el código esté mejor organizado y sea más fácil de entender. Los módulos encapsulan el comportamiento específico de cada estado, lo que elimina la necesidad de múltiples verificaciones condicionales y simplifica el flujo de ejecución. Otro beneficio importante es que facilita la adición de nuevos estados. Cuando se necesita agregar un nuevo estado, basta con definir una nueva clase que implemente el comportamiento correspondiente a ese estado, sin necesidad de modificar las funciones existentes que dependen de otros estados. Esto hace que el sistema sea mucho más flexible y escalable a medida que crece en complejidad. Además, el State elimina la duplicación de código, ya que cada clase de estado gestiona su propio comportamiento. En el enfoque tradicional, muchas funciones deben realizar repetidas verificaciones del estado y aplicar el comportamiento adecuado, lo que conduce a código repetido y potencialmente inconsistente. Una ventaja clave que mencionamos es que permite cambiar el comportamiento del sistema dinámicamente. A medida que el estado cambia, el comportamiento del objeto principal también cambia automáticamente.

Todo esto hace que el sistema sea más robusto y adaptable, ya que los cambios de estado y las transiciones se manejan de manera interna sin depender de verificaciones externas.

Cuando es necesario modificar el comportamiento de un estado, simplemente se actualiza la implementación del módulo correspondiente, lo que facilita la localización y modificación del código relevante. Y como resultado es más fácil de mantener y reduce el riesgo de introducir errores al cambiar o extender el sistema.

Existen otros casos de usos del patrón, como en los ejemplos que se nombraron al principio, puede ser usado para cambiar el comportamiento de cierta parte del sistema o a modo de configuración del mismo. El concepto principal es el mismo, crear un módulo por cada estado posible y cambiar la implementación para que se ajuste a lo requerido.

1.8 Integridad de la información

Un pulso electromagnético o fallas en el hardware pueden causar daños información dejándola comprometida, por ejemplo un *bit flip*. Si los datos afectados son críticos el problema puede derivar en un error grave del sistema. Para hacer frente a esto existen diferentes técnicas, desde calcular *checksums* con la intención de verificar integridad hasta almacenar la información múltiples veces. En el libro Douglass propone el uso de dos estrategias, una para datos que ocupan mucha memoria y otra para pequeños valores.

Dejando de lado como exactamente funcionan, la forma en la que son agregados a la estructura que almacena los datos es similar. Esta es, agregando funcionalidad extra a este ultimo, de manera se añade la capacidad de realizar ciertos cálculos de verificación.

Data
value: dato invertedValue: dato
get_data(): dato set_data(i: dato) invert()

Desde el punto de vista del diseño de software podemos aportar una mirada critica hacia las propuestas el libro, dado que proponen modificar la interfaz e implementación de los módulos encargados de almacenar la información con el objetivo de añadir esta “capa” de verificación. Es decir que los módulos no solo ocultan como se almacena la información sino que se encargan de verificar su integridad. Se me ocurre que una posible forma de solucionar esto desde el diseño orientado al cambio es utilizar el patrón *Decorator* que nos permite añadir de manera dinámica funcionalidades. En este caso, la capacidad de verificar la información. Otra ventaja, además de poder hacerlo de manera dinámica, es que mantenemos separadas las responsabilidades de cada módulo. A diferencia de lo propuesto en el libro, no juntos el almacenaje de la información con su verificación, de manera que cada porción puede ser implementada de manera independiente.

```
1 CRCData:set_data(i: dato){  
2     calculated_crc = calcular_crc(i)  
3     data.set_data(dato)
```

Figura 1.20: Ejemplo *Decorator* integridad de la información

	CRCData
Data	calculated_crc: crc error_handler: error_handler
get_data(): dato set_data(i: dato)	get_data(): dato set_data(i: dato) calcular_crc(i: dato) set_error_handler(i: error_handler)

```

4 }
5
6 CRCData: get_data() {
7     dato = data.get_data()
8     if calculate_crc(dato) == calculated_crc
9         return dato
10    else
11        error_handler()
12 }

```

De esta forma logramos añadir una capa de protección a la integridad de la información, a su vez permitimos combinar diferentes técnicas usando este patrón.

1.9 Verificación de precondiciones.

El autor del libro comenta que uno de los problemas más grandes que ve en el desarrollo de sistemas embebidos es que prácticamente todas las funciones tienen precondiciones para ejecutarse correctamente pero que casi nunca se verifica si todas estas se cumplen. Además, la manera común de informar precondiciones inadecuadas en una función en **C** es utilizando el valor de retorno (-1 en caso de errores, 0 en el contrario). Por lo tanto el encargado de manejar el error es la función o método que llamó en primer lugar a la función, generando así un acoplamiento extra entre módulos. Esto provoca una complicación derivando en muchas veces no manejarlo de la manera más adecuada. Por ejemplo, supongamos que tenemos un módulo que exporta una función que permite guardar un valor, `setValor(i: Valor)`. Existen múltiples posibles implementaciones, pero veamos algunas:

- Una posible implementación, es que la función no realice ningún chequeo y simplemente guarde en una variable el valor pasado como argumento. Esto puede generar un error en el momento si el tipo no coincide por ejemplo o en el futuro si el valor está por fuera de ciertos parámetros requeridos por el sistema. Por ejemplo, si se está almacenando un entero negativo pero se requiere que siempre sea positivo, podemos provocar un error en el futuro, si es que los clientes no chequean que sea positivo y a su vez, estamos obligando a que todos los que usen este valor deban verificar que sea positivo. Escribiendo código repetido y que en caso de que el requisito cambie deba ser actualizado manualmente.

- Otra forma es que la función verifique que el valor es permitido y retorne 0 o -1 en caso de que no. Esto es mejor que nada, pero como se comento en el principio estamos acoplando al usuario de la función con el módulo que la exporta, ya que este tiene que verificar el valor de retorno para determinar si existió un error o no.

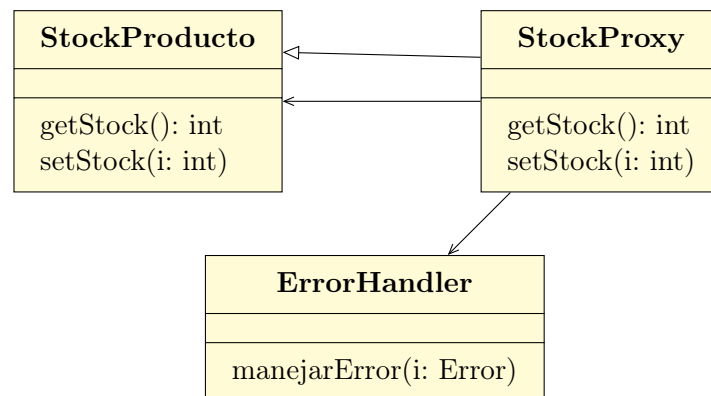
Este enfoque es usado también para otro tipo de funciones como aquellas que requieren que sus argumentos cumplan cierta precondition para poder computar o realizar su trabajo. Y como Douglass comenta, esta es la realidad de la mayoría de las funciones.

La solución que se propone en el libro es una combinación de patrón y practica de programación, tiene los siguientes conceptos claves:

- Construir tipos de auto-verificación siempre que sea posible
- Verificar los valores de los parámetros entrantes para un rango adecuado
- Verificar la consistencia y razonabilidad entre uno o un conjunto de parámetros

De esa manera se estaría separando la responsabilidad de verificar precondiciones de las funciones y de los clientes. Para conseguirlo se puede utilizar el patrón *Proxy* al rededor de un tipo básico (un *array* de *ints*, por ejemplo), y de esa manera proveer múltiples funciones que permitan verificar los rangos, consistencia y demás de *features* de los datos. Y en caso de detectar un problema se llamará a un módulo que cumple el rol de *handler* de errores, responsable de decidir como continuar. Este uso del patrón es mencionado por Gamma en el capitulo del mismo como una posible aplicación. Es decir, realizar operaciones extras cuando se accede a un objeto. Además, comenta que se pueden agregar más funcionalidades que las que menciona Douglass, como contar la cantidad de referencias a cierto objeto con el fin de liberar la memoria si nadie lo esta referenciando, verificar si un objeto está bloqueado por otro (mutex) o controlar el acceso a la información mediante permisos.

Veamos el siguiente ejemplo, se tiene un módulo *DatosProductos* el cual almacena información relacionada al *stock* de los productos, como es información importante se quiere añadir validaciones al momento de guardar como de extraer. Al guardar se quiere ver que la cantidad ingresada no sea negativa y al consultar se quiere estar seguro de que el usuario tiene los permisos requeridos.



ErrorHandler decide como manejar el error, esto puede ser desde loggearlo, terminar la ejecución del sistema, ignorarlo, etc. Dos tipos de estrategias comunes al tratar con errores son:

- Reset de componentes: reiniciar o restablecer componentes defectuosos ayuda a borrar errores y restaurarlos a un estado funcional.
- Degradación elegante⁴: permite que el sistema siga funcionando a una capacidad reducida en lugar de fallar por completo. Esto implica deshabilitar las funciones que funcionan mal o cambiar a copias de seguridad, por lo que el sistema continúa funcionando con una funcionalidad limitada en lugar de apagarse por completo.

Con este diseño añadimos una capa de verificación sin modificar el módulo original, ya que `getStock` y `setStock` de *StockProxy* van a realizar la verificación indicada y luego si todo es correcto invocarán las funciones de *DatosProductos*. Con esto conseguimos las siguientes ventajas:

- encapsular la lógica de validación y control de acceso en un solo lugar, sin necesidad de modificar el módulo original que gestiona los datos. Facilitando la reutilización y la separación de responsabilidades.
- centraliza el control sobre cómo y cuándo se accede o modifica el dato sensible. Esto facilita la implementación de reglas de negocio más complejas, como la validación de entradas, sin tener que modificar cada parte del código que interactúa con DatosProductos.
- extender funcionalidades, el proxy puede evolucionar independientemente del objeto real, añadiendo nuevas funciones o cambiando las reglas de validación sin afectar la implementación original del módulo que contiene los datos. Por ejemplo, se pueden implementar restricciones de acceso, límites en las operaciones permitidas o incluso operaciones en diferido, sin necesidad de modificar el objeto real.

1.10 Organización de la ejecución

En un sistema robótico embebido se tienen que ejecutar múltiples tareas para lograr el objetivo del mismo. Ya sea, si se eligió una arquitectura de diseño de tipo “Control de procesos” o no, por lo general se debe verificar información recibida a través de sensores y otras fuentes (comunicación serial, web, etc), realizar cálculos y efectuar acciones con los actuadores del robot.

Como primer solución uno piensa en crear funciones para cada parte del proceso y llamarlas en *loop* en el *main*, y si es necesario añadir un tiempo de espera entre cada ejecución mediante *sleeps*. El principal problema que tiene este enfoque es que los sistemas robóticos son en tiempo real, es decir podemos perder *inputs* de sensores si no los manejamos de manera correcta. Además, los tiempos de espera son bloqueantes por lo que perdemos acceso a computo. Una estrategia que logra mejorar estos puntos es la utilizada por Laura en el

⁴Incorporating Graceful Degradation into Embedded System Design

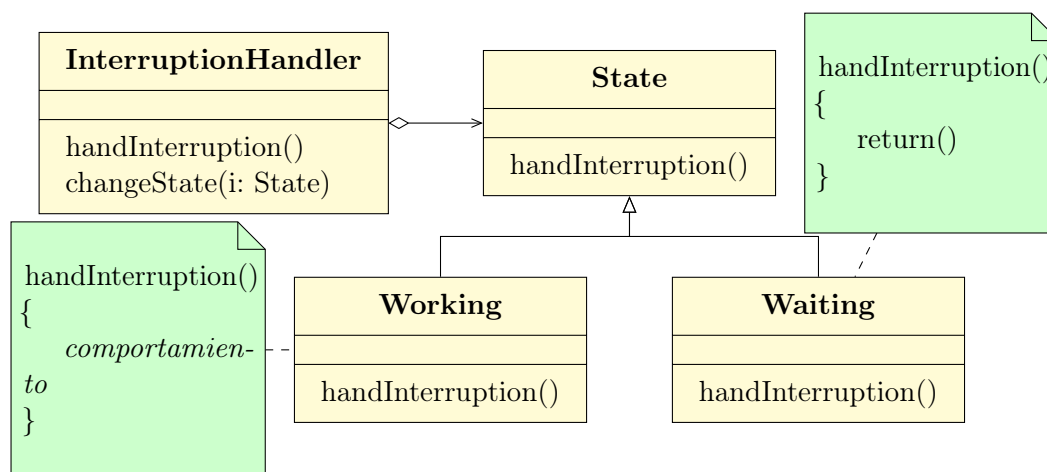
diseño del robot desmalezador. Esta utiliza las interrupciones del microcontrolador, y permite atender a todos los *inputs*. En el robot desmalezador esta estrategia se utiliza en conjunto con una arquitectura de control de procesos, por lo tanto tenemos 3 principales tareas:

- Recibir información de los sensores y de la PC (por ejemplo, puerto serial).
- Procesar la información recibida y decidir que valores aplicar en los actuadores.
- Aplicar los nuevos valores a los actuadores. Notar que no es tan simple como setear un numero y dejar que actúe, en muchos casos se necesita un seguimiento durante el tiempo.

De las tareas que tenemos, 2 las debemos llamar nosotros y otra (recibir información) en muchos casos se dará en forma de interrupciones que debemos manejar. Ya se comentó un poco de ese proceso en la sección **Obtención de información**, solo es importante recordar que la estructura propuesta nos provee de una interfaz en la cual podemos llamar para obtener la información recibida de manera simple. Una vez resulta esa cuestión ahora tenemos que ejecutar las otras dos tareas, dado que se quieren hacer los ajustes cada determinado tiempo, se propone crear una nueva interrupción que sea disparada de manera periódica y que su *handler* se encargue de realizar todo el proceso de control y cálculo. En el manejo de esta interrupción, se accederá a la interfaces propuestas para obtener información de los sensores. Y por ultimo, para los actuadores que necesitan un seguimiento temporal para su control, agregamos una nueva interrupción que puede ser individual para el actuador y el tiempo de disparo puede estar determinado de manera particular.

Tener interrupciones periódicas puede general una bola de nieve de ejecución si el manejo de una tarda más que el tiempo entre disparos. Para prevenirlo se aplica el patrón *State*, haciendo que cuando se comience a manejar la interrupción cambie el estado y toda nueva llamada al *handler* resulte en un retorno rápido (solo *return*). Una vez terminada la ejecución de este ciclo se cambia el estado otra vez permitiendo la ejecución de nuevos llamados.

Figura 1.21: Ejemplo de handler usando *State*.



Notar que esta estrategia explota la capacidad de manejar interrupciones del microcontrolador, por lo tanto es necesario que este las soporte para llevarla a cabo.

Apéndice A

Code listing

This example uses the listings package.

```
1 function print_rate(kappa,xMin,xMax,npoints,option)
2     local c = 1-kappa*kappa
3     local croot = (1-kappa*kappa)^(1/2)
4     local logx = math.log(xMin)
5     local psi = 0
6
7     local xstep = (math.log(xMax)-math.log(xMin))/(npoints-1)
8
9     arg0 = math.sqrt(xMin/c)
10    psi0 = (1/c)*math.exp((kappa*arg0)^2)*(erfc(kappa*arg0)-erfc(
        arg0))
11
12    if option~= [[]] then
13        tex.sprint("\addplot+[\"..option..\" coordinates{")
14        -- addplot+ for color cycle to work
15    else
16        tex.sprint("\addplot+ coordinates{")
17    end
18    tex.sprint("("..xMin..","..psi0..")")
19
20    for i=1, (npoints-1) do
21        x = math.exp(logx + xstep)
22        arg = math.sqrt(x/c)
23        karg = kappa*arg
24        if karg<5 then
25            -- this break compensates for exp(karg^2), which multiplies the
                error in the erf approximation...
26            logpsi = -math.log(croot) + karg^2 + math.log(erfc(karg)-
                    erfc(arg))
27            psi = math.exp(logpsi)
28        else
```

```

29         psi = (1/(karg) - 1/(2*(karg^3)) + 3/(4*(arg^5)) )/(1
               .77245385*croot)
30         -- this is the large x asymptote of the reaction rate
31     end
32     logx = math.log(x)
33     tex.sprint("(" .. x .. ", " .. psi .. ")")
34 end
35 tex.sprint("}")
36 end
37 \end{luacode*}

```