Patrones de diseño aplicados a sistemas embebidos de control

por

Ignacio Petruskevicius

Departamento de Ciencias de la Computación

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD NACIONAL DE ROSARIO

June 2025

Tesinista: Ignacio Petruskevicius

May 18, 2025

Directora: Laura Pomponio
Licenciada en Ciencias de la Computación

Patrones de diseño aplicados a sistemas embebidos de control

by

Ignacio Petruskevicius

Submitted to the Departamento de Ciencias de la Computación
on May 18, 2025 in partial fulfillment of the requirements for the degree of

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

RESUMEN

En el ámbito de la robótica, los sistemas embebidos desempeñan un papel fundamental al gestionar el acceso al hardware, garantizar la concurrencia, y responder en tiempo real a los requerimientos del entorno. Sin embargo, el diseño de software en este dominio a menudo sigue prácticas tradicionales, como la descomposición funcional y el uso de estructuras condicionales anidadas, lo cual dificulta la capacidad del sistema para adaptarse a cambios en el hardware o a nuevos requisitos funcionales.

A partir del trabajo realizado sobre el diseño del robot desmalezador del CIFASIS y su posterior implementación y verificación [Pom+24], en el cual se mostró la viabilidad de realizar el diseño para este tipo de sistemas, se propone buscar problemas de diseño comunes en el ámbito y darles una solución utilizando los conceptos y buenas practicas de la ingeniería de software. Los problemas fueron extraídos de distintos libros [Dou11][Whi11] que a su vez intentar dar soluciones que como luego veremos no siempre se alinean con los estándares de la IS.

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Conceptos previos

1.1 Sistemas embebidos

Veamos las siguientes definiciones de sistemas embebidos extraídas de diferentes autores:

"*Un sistema computarizado dedicado a realizar un conjunto específico de funciones del mundo real, en lugar de proporcionar un entorno de computación generalizado.*" [Dou11]

"*Un sistema embebido es un sistema computarizado diseñado específicamente para su aplicación.*" Debido a que su misión es más limitada que la de una computadora de propósito general, un sistema embebido tiene menos soporte para aspectos no relacionados con la ejecución de la aplicación. [Whi11]

"*Un sistema embebido es un sistema informático aplicado, a diferencia de otros tipos de sistemas informáticos como las computadoras personales (PCs) o las supercomputadoras.*" [Noe05]

El último autor comenta que los sistemas embebidos cumplen las siguientes afirmaciones:

- Los sistemas embebidos son más limitados en funcionalidad de hardware y/o software que una computadora personal.
- Un sistema embebido está diseñado para realizar una función dedicada.
- Un sistema embebido es un sistema informático con requisitos de mayor calidad y fiabilidad que otros tipos de sistemas informáticos.
- Algunos dispositivos que se denominan sistemas embebidos, como los PDA[1] o las tabletas web, no son realmente sistemas embebidos.

De las definiciones podemos concluir que un sistema embebido es una pieza clave que permite que el hardware especializado cumpla con su propósito específico. A diferencia de los sistemas de propósito general, el software en un sistema embebido está diseñado para interactuar estrechamente con los componentes de hardware, respondiendo en tiempo real

[1] Asistentes Personales Digitales

a eventos del entorno, ya sea para controlar actuadores, monitorear sensores o gestionar comunicaciones. Este software está optimizado para requisitos específicos como velocidad, consumo energético, y confiabilidad, lo que lo hace esencial en aplicaciones críticas como dispositivos médicos, sistemas automotrices y controles industriales.

En resumen, el software de un sistema embebido actúa como el cerebro que dirige y coordina los recursos del hardware para realizar funciones concretas. En la tabla 1.1 encontramos ejemplos de dispositivos en los que se utilizan sistemas embebidos extraída de [Noe05].

Cuadro 1.1: Ejemplos sistemas embebidos.

Mercado	**Dispositivo Embebido**
Automotriz	Sistema de encendido Control del motor Sistema de frenos (Sistema Antibloqueo de Frenos - ABS)
Electrónica de consumo	Decodificadores (DVDs, VCRs, Cajas de cable, etc.) Asistentes Personales Digitales (PDAs) Electrodomésticos (Refrigeradores, Tostadoras, Microondas) Automóviles Juguetes/Juegos Teléfonos/Celulares/Bípers Cámaras Sistemas de Posicionamiento Global (GPS)
Control Industrial	Sistemas de control y robótica (Manufactura)
Médico	Bombas de infusión Máquinas de diálisis Prótesis Monitores cardíacos
Redes	Routers Hubs Puertas de enlace
Automatización de Oficina	Máquinas de fax Fotocopiadoras Impresoras Monitores Escáneres

Otros autores [LS17] describen sistemas *ciber-físicos* (CSP[2]) como la integración de la computación con procesos físicos. Usualmente llevada a cabo utilizando sistemas embebidos con ciclos de retroalimentación. En los cuales la parte computacional afecta al ámbito físico y viceversa. Para realizarlo se utiliza una serie de sensores y actuadores que permiten la comunicación entre ambos mundos. Además, si tomamos en cuenta los ejemplos que se presentan tanto en [Noe05] como en [LS17], podemos decir que la mayoría de los sistemas embebidos realizan tareas de **control** sobre el mundo físico.

[2]por sus siglas en inglés (cyber-physical system)

1.2 Ingeniería de Software

La arquitectura y el diseño del software se consideran herramientas esenciales para lograr atributos importantes de calidad del software, como la modificabilidad, reusabilidad, mantenibilidad, etc.[SG96; GJM03; BCK03; TMD10] Tradicionalmente, el software para robots tiende a desarrollarse de manera monolítica, con pocas funciones de gran tamaño y numerosos condicionales anidados, o mediante una descomposición funcional básica que resulta ineficiente para mantener y reutilizar componentes [22; 20]. Frente a esto, se propone un diseño sistemático basado en principios de la ingeniería de software, aplicados para anticipar y manejar los cambios [Gam+95; BHS07].

El diseño para el cambio, como principio fundamental, se centra en prever modificaciones probables en el software [Par72; SG96; GJM03; BCK03; TMD10], ya sea por cambios en los objetivos de comportamiento, en el hardware o en los algoritmos de control. Diseñar anticipando estos cambios permite reducir los esfuerzos necesarios para implementar ajustes y facilita la reutilización de componentes en familias de software[3] que comparten funcionalidades similares pero adaptadas a diferentes plataformas de hardware[Par78; CN02].

Para abordar estos retos, el diseño modular es clave [Par72]. Este enfoque organiza el software como un conjunto de módulos simples con responsabilidades claramente definidas y relaciones explícitas entre ellos. Cada módulo se diseña para ocultar información específica y gestionar un cambio probable de forma aislada, lo que minimiza el impacto de las modificaciones en el resto del sistema. Por ejemplo, si un sensor cambia su forma de reportar datos, el ajuste puede limitarse al módulo responsable de interactuar con ese sensor.

Además, el principio de diseño abierto-cerrado [Mey97] se implementa para garantizar que el sistema pueda extenderse mediante nuevos módulos en lugar de modificar los existentes, reduciendo el riesgo de introducir errores al alterar componentes ya probados. Es decir, que un sistema debe estar abierto a extensiones pero cerrado a modificaciones [Mey97]. Esto se complementa con la aplicación de patrones de diseño y estilos arquitectónicos, que aportan soluciones probadas para manejar cambios recurrentes en dominios específicos [Gam+95; BHS07]. Por ejemplo, en sistemas robóticos, un estilo arquitectónico basado en bucles de control puede destacar las características esenciales del sistema y facilitar decisiones de diseño óptimas [SG96].

Herencia de Interfaz La herencia de interfaz es una técnica mediante la cual una clase hija hereda la interfaz (es decir, el conjunto de métodos públicos) de una clase padre. Este enfoque permite que las clases hijas implementen o amplíen la funcionalidad definida en la clase base. Según el archivo, este método se utiliza típicamente cuando las subclases comparten comportamientos comunes, y el objetivo principal es reutilizar código y garantizar una coherencia en la interfaz entre las clases relacionadas. Sin embargo, una limitación clave de la herencia es que introduce un acoplamiento fuerte entre la clase base y las subclases, lo que puede dificultar la adaptabilidad a cambios inesperados en el sistema.

Composición La composición consiste en estructurar sistemas combinando módulos a través de sus interfaces en lugar de depender de relaciones de herencia. Esto se logra mediante la inclusión de referencias a otros dentro de una clase, lo que permite que las instancias deleguen tareas a estos módulos asociados. En el diseño presentado en el archivo,

[3] en este caso familias de robots

la composición es destacada como una solución que favorece la flexibilidad, ya que evita el acoplamiento jerárquico y permite reemplazar componentes sin afectar el resto del sistema. Además, este enfoque está alineado con el principio de diseño "preferir la composición sobre la herencia" [Gam+95], que enfatiza la modularidad y la apertura al cambio.

1.2.1 Metodología de Parnas

La metodología de **Parnas**[Par72], conocida como Diseño Basado en Ocultación de la Información (DBOI), es una estrategia de diseño modular que tiene como objetivo preparar los sistemas de software para gestionar el cambio de manera eficiente y con el menor costo posible. Esta metodología parte del principio de que los requerimientos de un sistema no son inmutables, sino que evolucionarán durante su vida útil. Por ello, el diseño debe anticipar y facilitar la incorporación de cambios sin comprometer la integridad del sistema.

> Principio de Ocultación de la Información: Los ítem con alta probabilidad de cambio son el fundamento para descomponer un sistema en módulos. Cada módulo de la descomposición debe ocultar un **único** ítem con alta probabilidad de cambio, y debe ofrecer a los demás módulos una interfaz insensible a los cambios anticipados. [Par72]

El núcleo de esta metodología es la identificación de los ítems con alta probabilidad de cambio dentro del sistema. Estos ítems representan aspectos de diseño o implementación susceptibles de modificaciones futuras, como algoritmos, estructuras de datos, interfaces con hardware o incluso requerimientos del usuario (ver lista extendida en 1.2.2). Una vez identificados, Parnas sugiere aislar cada ítem en módulos independientes, asegurando que cada módulo oculte las decisiones de diseño específicas que podrían cambiar. Esto se logra diseñando interfaces que no expongan detalles de implementación, permitiendo que los módulos interactúen sin conocer sus detalles internos.

La razón por la que queremos aplicar esta metodología es clara: minimizar los costos asociados al desarrollo y mantenimiento del software. Al aislar las áreas susceptibles de cambio, cualquier modificación futura afectará únicamente al módulo correspondiente, sin propagarse al resto del sistema. Además, esta aproximación mejora la capacidad de escalar el sistema, facilita la colaboración en equipos de desarrollo grandes y permite que diferentes programadores trabajen en módulos específicos de manera independiente.

La metodología de Parnas nos ayuda a diseñar para el cambio porque impone una disciplina clara en la forma en que los módulos se estructuran e interactúan. Al encapsular las decisiones de diseño que podrían cambiar, evitamos la degradación de la integridad conceptual del sistema y reducimos significativamente el riesgo de introducir errores al realizar modificaciones. En definitiva, el DBOI fomenta un diseño robusto y adaptable, preparado para enfrentar la evolución inevitable de los sistemas de software.

Los pasos de la metodología son:

1. Identificar los ítem con probabilidad de cambio presentes en los requerimientos.
2. Analizar la diversas formas en que cada ítem puede cambiar.
3. Se asigna una probabilidad de cambio a cada variación analizada.

4. Aislar en módulos separados los ítem cuya probabilidad de cambio sea alta; implícitamente este punto indica que en cada módulo se debe aislar un único ítem con probabilidad de cambio.

5. Diseñar las interfaces de los módulos de manera que resulten insensibles a los cambios anticipados.

1.2.2 Items de cambio comunes

Cuando se diseña pensando en el cambio, una tarea que se agrega es identificar las características o requerimientos del sistema que pueden variar en el tiempo. Naturalmente existen elementos que son mas probables a cambiar que otros, por lo que resulta importante anticiparse a esos cambios en particular. Algunos autores mencionaron algunos items de cambio comunes entre múltiples sistemas [Par78].

- Contracción o extension de requisitos.
- Configuraciones de hardware.
- Formato de los datos de entrada y salida.
- Estructuras de datos.
- Algoritmos.
- Algunos usuarios pueden requerir solo un subconjunto de los servicios o características que otros usuarios necesitan.
- Dispositivos periféricos.
- Entorno socio-cultural (moneda, impuestos, fechas, idioma, etc.).
- Cambios propios del dominio de aplicación.
- Cambios propios del negocio de la compañía desarrolladora.
- Interconexión con otros sistemas.

Es útil consultar estos items a la hora de diseñar siguiendo los criterios de modularización de [Par72].

1.2.3 Patrones de Diseño

En [Gam+95], el autor trae a colación la definición de patrón de diseño que dio Christopher Alexander:

"cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución al problema, de modo que pueda aplicarse un millón de veces esta solución sin hacer lo mismo dos veces"

Christopher era arquitecto, pero su definición puede ser aplicada al ámbito del software, en lugar de paredes, vigas y columnas, trabajamos con módulos e interfaces.

Un patrón tiene tres elementos principales:

- El **problema** al que se intenta dar solución. Posee una explicación del mismo, con el fin de que el usuario pueda saber si aplica o no a su situación en cuestión.
- La **solución** al problema, dada como los elementos de del diseño, sus relación responsabilidades y colaboraciones. Es una plantilla que puede ser aplicada en diferentes condiciones.
- Las **consecuencias**, que son los resultados de aplicar esta solución. Es decir, que beneficios y costos obtenemos de la aplicación. Así como las formas que el patrón provee para anticiparse a los posibles cambios futuros.

Determinar que es o no un patrón resulta objetivo, pero utilizaremos el mismo criterio que el autor de [Gam+95]:

"'descripciones de módulos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto"

Notar no solo se quiere saber como resolver un problema, sino que se busca que la solución se alinee con los principios de la ingeniería de software y provea un buen diseño que permita lograr las propiedades que se buscan, modificabilidad, reusabilidad, mantenibilidad, etc. Esto será importante para diferenciar algunos "patrones de diseño" que no tiene en cuenta el objetivo mencionado.

Para anticiparse al cambio, los patrones aseguran que un sistema pueda cambiar de manera concreta, es decir, que se deja que algún aspecto de la estructura varíe de manera independiente y esperada.

1.2.4 Arquitectura de software

Según [SG96] la arquitectura de software se define como la estructura fundamental de un sistema de software, que está compuesta por sus componentes y las relaciones entre estos. Este campo aborda la organización y los patrones utilizados para estructurar los sistemas de software de manera que sean eficientes, sostenibles y capaces de manejar cambios a lo largo del tiempo.

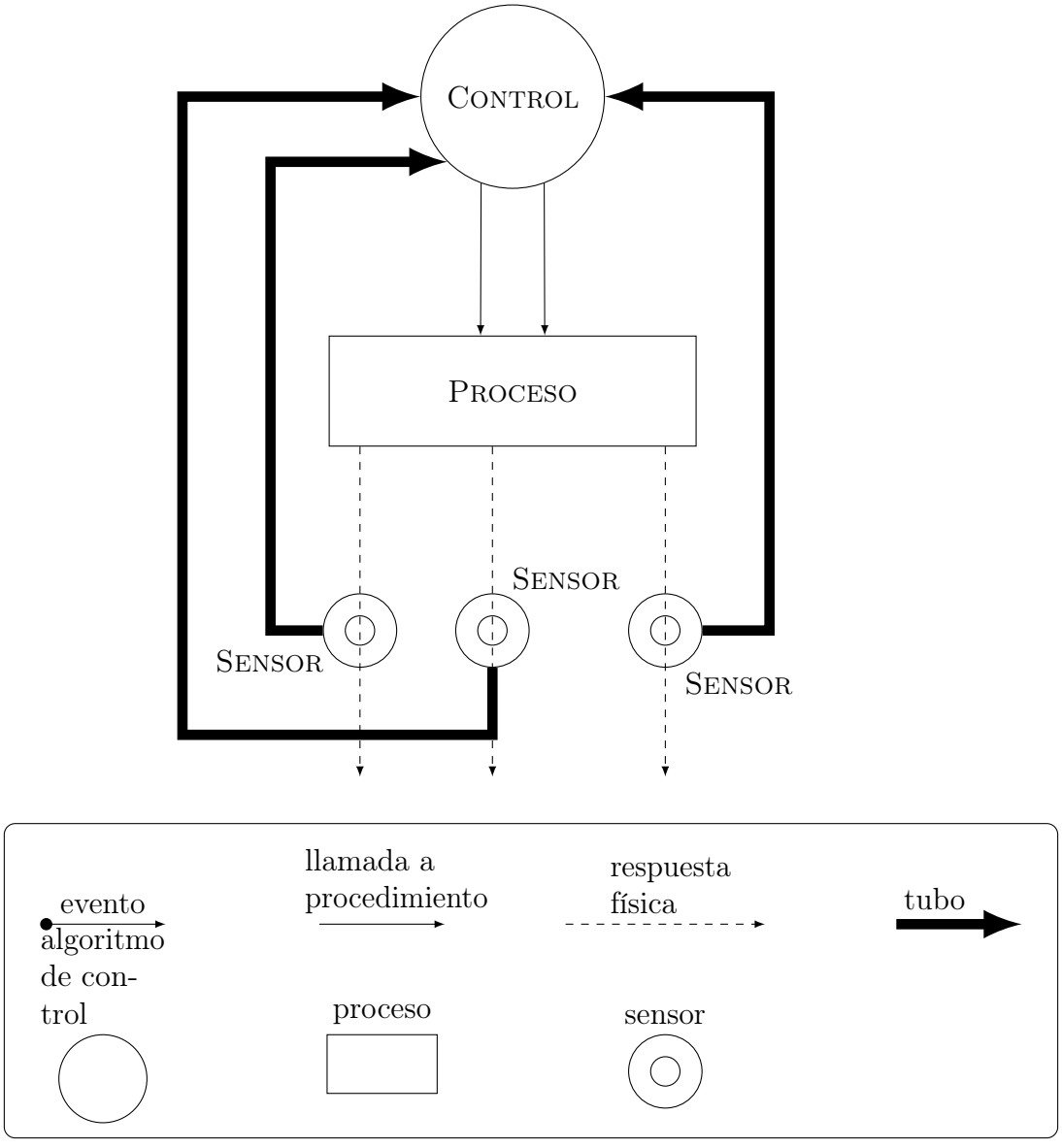
Se destaca que la arquitectura de software no solo se trata de la estructura del código o la implementación técnica, sino de las decisiones de alto nivel que afectan la organización y el comportamiento del sistema. Estas decisiones incluyen cómo dividir un sistema en partes modulares, qué patrones arquitectónicos aplicar para facilitar la extensión y el mantenimiento, y cómo gestionar las interacciones entre diferentes componentes del sistema. Notar que cuando se habla de componentes estos pueden estar formados por múltiples módulos, son de una capa de abstracción superior a los conceptos que se manejan cuando hablamos de patrones de diseño.

1.3 Arquitectura Control de Procesos

Si nos centramos en los sistemas embebidos de control, encontramos que existen trabajos sobre arquitecturas de software orientadas al control de procesos, por ejemplo el estilo arquitectónico de *Control del Procesos* presentado en [SG96]. El mismo está definido para ser usado en sistemas de control donde se quiere mantener ciertas propiedades de la salida del proceso cerca de valores de referencia. Como por ejemplo la velocidad de giro de una rueda, la posición del extrusor de una impresora 3D, la temperatura del agua en una caldera, etc.

Para llevar a cabo el enfoque se fundamenta en tres componentes básicos: **Control**, **Proceso** y **Sensores**, los cuales trabajan de manera independiente.

Figura 1.1: Diagrama de la arquitectura control de procesos



El componente **Control** es responsable de implementar el algoritmo de control, procesar datos provenientes de los sensores y realizar ajustes al proceso para mantener las variables dentro de los valores deseados. Además, se encarga de activar y desactivar el sistema, y de configurar los rangos de operación o *set-points*. Por otro lado, el **Proceso** encapsula los dispositivos que generan las salidas controladas, proporcionando interfaces para modificar sus variables según las instrucciones del **Control**. Finalmente, los **Sensores** miden las variables clave del proceso y transmiten estos datos al componente **Control**, ocultando la complejidad de los dispositivos de medición.

Esta arquitectura emplea conectores como eventos, llamadas a procedimiento y tubos (pipes) para gestionar la comunicación y las acciones entre los componentes. A nivel computacional, el sistema funciona en un ciclo continuo de retroalimentación donde los sensores miden las variables, el **Control** evalúa estas mediciones y, de ser necesario, ajusta el **Proceso** para garantizar el cumplimiento de los objetivos definidos. Este enfoque modular facilita la incorporación de nuevos sensores, actualizaciones en el algoritmo de control y cambios en el hardware, promoviendo flexibilidad y escalabilidad.

Cuadro 1.2: Conceptos clave en la arquitectura de control de procesos.

Término	**Definición**
Variable del proceso	Propiedades del proceso que pueden medirse y monitorearse, como temperatura, presión, flujo o velocidad. Estas variables reflejan el estado del sistema.
Variable controlada	Una variable del proceso cuyo valor el sistema intenta mantener en un rango deseado, como la temperatura de un horno o el nivel de agua en un tanque.
Variable manipulada	Variable del proceso que el controlador puede ajustar directamente para influir en la variable controlada, como la válvula de flujo en un sistema de bombeo.
Variable de entrada	Variable que mide una entrada al proceso, como la potencia suministrada a un motor o la cantidad de material en una cinta transportadora.
Set Point	El valor deseado para una variable controlada. El controlador busca ajustar el sistema para alcanzar y mantener este valor.

Capítulo 2

Técnicas útiles

2.1 Desacople de módulos

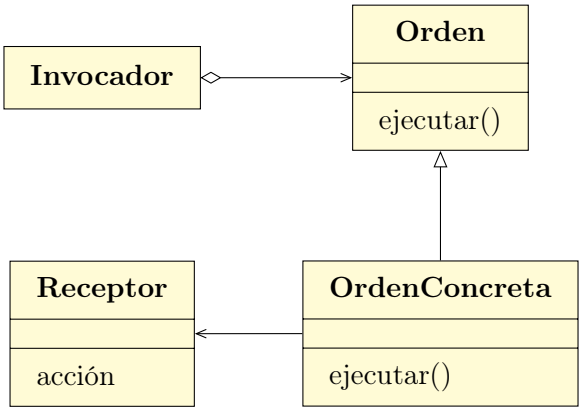
A lo largo de los ejemplos del documento, veremos repetidas veces el uso de la noción de *orden* o *comando*. Cada vez que sea nombrado haremos referencia a la aplicación de un patrón de diseño de Gamma [Gam+95], el patrón *Command*.

La función principal que cumple este patrón es la de desacoplar el objeto que invoca una orden, la orden en si y aquel que sabe como llevarla a cabo. Es decir, reemplaza las *callbacks* entre módulos. Como objetivo fundamental, buscamos no tener que modificar la implementación del módulo invocador en caso de un cambio en el que se encarga de realizar las tareas. Ademas, se le quita la responsabilidad de saber exactamente que acciones realizar para llevar a cabo un procedimiento particular. Por ejemplo, un módulo necesita que otro se inicie, pero este ultimo para hacerlo requiere que se invoque una serie de sus funciones de manera ordenada. Si lo hacemos de la forma clásica, el primer módulo debe ajustar su implementación al segundo. En cambio, con una orden movemos la responsabilidad a un nuevo módulo.

- **Invocador**: le pide a la Orden que ejecute la acción.
- **Orden**: declara la interfaz para ejecutar una acción.
- **OrdenConcreta**: implementa la función ejecución la cual se encarga de llamar la o las funciones del **Receptor** con el objetivo de llevar a cabo la acción.
- **Receptor**: cualquier clase, es sobre la cual se realiza la acción.

Al encapsular cada solicitud de una operación dentro de un objeto, el patrón permite que los módulos que incovan acciones no necesiten conocer los detalles de implementación de los módulos que las ejecutan. Esto reduce significativamente las dependencias y hace que el sistema sea más fácil de mantener, ya que cada módulo se concentra en su propia responsabilidad, sin acoplarse a los detalles de otros módulos. El patrón *Command* es una herramienta efectiva para lograr el desacoplamiento entre módulos en sistemas complejos, especialmente útil cuando se busca modularidad y flexibilidad en la organización de comandos y acciones. Al encapsular cada solicitud de una operación dentro de un objeto de comando, el

Figura 2.1: Estructura patrón *Command*



patrón *Command* permite que los módulos que emiten instrucciones no necesiten conocer los detalles de implementación de los módulos que las ejecutan. Esto reduce significativamente las dependencias y hace que el sistema sea más fácil de mantener, ya que cada módulo se concentra en su propia responsabilidad, sin acoplarse a los detalles de otros módulos. Además, la extensión de funcionalidades se simplifica considerablemente. Dado que cada acción está representada por un módulo independiente, se pueden agregar nuevos comandos al sistema sin modificar los módulos existentes. Esta estructura es particularmente útil cuando se necesita modificar o añadir funcionalidades de manera frecuente. Al mismo tiempo, los comandos encapsulados pueden almacenarse, reutilizarse y combinarse en secuencias, lo que facilita la implementación de operaciones complejas que se repiten o que requieren ser acumuladas para un procesamiento posterior. Por otro lado, como el comando es un módulo puede ser extendido para implementar múltiples funcionalidades, como deshacer operaciones o registrar cambios.

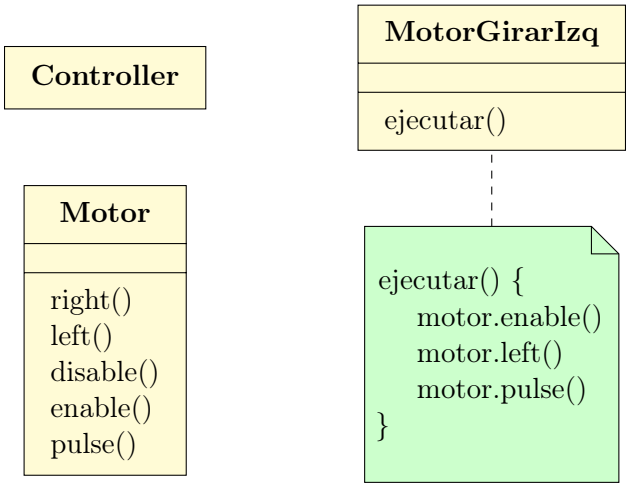
El módulo **Controller** aplicará la acción de girar el motor a la izquierda un paso sin saber como llevarla a cabo específicamente. Y en caso de que el motor cambie, solo debemos modificar la orden, e incluso podemos dejar la existente y crear una nueva con la nueva implementación.

En caso de no aplicar el patrón tendríamos todo el código que se encarga de girar el motor hacia la izquierda en **Controller**, como en el código 2.1.

Listing 2.1: Ejemplo de implementación sin usar el patrón *Command*.

```
control() {
    .
    .
    .
    motor.enable()
    motor.left()
    motor.pulse()
    .
    .
```

Figura 2.2: Ejemplo de aplicación básica del patrón *Command*




```
10      .
11  }
```

En caso de que el modo de operación del motor cambie deberemos modificar el código del método `control`.

Otro uso interesante del patrón, es cuando se define una cierta estructura conceptual en el sistema. Esta puede responder a la naturaleza de la aplicación del mismo. Por ejemplo, en un sistema de control, se puede definir que los módulos que realizan el control del mismo invoquen a los sensores para obtener información. Generando así, una cierta jerarquía en la cual los sensores no deben invocar métodos de módulos superiores. En caso de ser necesaria la comunicación de manera inversa se puede hacer uso del patrón *command* para reemplazar el uso de *callbacks*.

2.2 Interrupciones

Las interrupciones en un microcontrolador son eventos que pausan la ejecución del programa principal para atender una tarea urgente. Funcionan como un mecanismo de respuesta automática que permite que el microcontrolador responda inmediatamente a eventos externos o internos sin depender de que el programa principal revise continuamente el estado de los dispositivos o variables asociadas a la generación de la interrupción.

Cuando ocurre una interrupción (por ejemplo, un cambio en un sensor o una solicitud de un actuador), el microcontrolador detiene su ejecución actual y salta a una rutina de interrupción (ISR, Interrupt Service Routine). Esta rutina es un fragmento de código predefinido que realiza las tareas necesarias, como leer un sensor o activar un actuador. Después de ejecutar la ISR, el microcontrolador regresa automáticamente al punto donde fue interrumpido, reanudando el programa principal sin perder el flujo de ejecución.

Este mecanismo es esencial en sistemas embebidos, especialmente en aquellos que controlan

sensores y actuadores, porque permite un control eficiente de múltiples dispositivos. Por ejemplo, un microcontrolador podría usar interrupciones para:

- Leer la temperatura de un sensor cada vez que detecta un cambio.
- Activar un motor o alarma inmediatamente al detectar un evento específico.

Gracias a las interrupciones, el microcontrolador puede realizar tareas en tiempo real y responder rápidamente a eventos críticos, asegurando un control preciso de los sensores y actuadores sin necesidad de monitorear activamente cada dispositivo constantemente.

Capítulo 3

Problemas comunes

3.1 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 3.1.

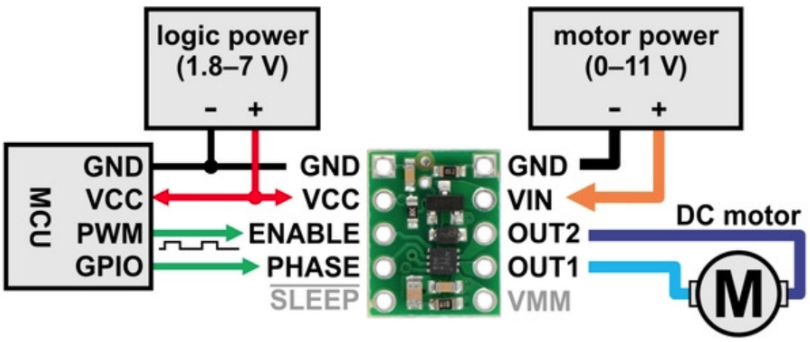


Figura 3.1: Conexionado módulo DRV8838

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, en esta tabla 3.1 podemos ver la función de cada pin.

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

Cuadro 3.1: Funciones de cada pin del módulo DRV8838

Por lo tanto, con los dos primeros pines podremos cumplir los requerimientos mencionados. Tendremos dos cables conectados a nuestro microcontrolador **Arduino Uno**, uno que se dirige a *ENABLE* y otro a *PHASE*. En el código 3.1 se encuentran las lineas necesarias para poder configurar el motor para luego poder utilizarlo como en los códigos 3.2 y 3.3. Las funciones `pinMode`, `digitalWrite` y `analogWrite` son provistas por el entorno de desarrollo de Arduino. Sus nombres son bastastantes descriptivos de su comportamiento, `pinMode` configura el modo de operacion de un PIN en particular, puede ser `OUTPUT` o `INPUT`. Y tanto `digitalWrite` como `analogWrite`, escriben en un PIN el valor especificado.

Listing 3.1: Configuración inicial

```
# Notar que los numeros asignados a los pines son arbritarios
   dentro del conjunto de pines disponibles en nuestro Aruino.

# Constantes globales
const int DIR_pin = 7;
const int VEL_pin = 9;

void setup() {
      .
      .
      .
  pinMode(DIR_pin, OUTPUT);
  pinMode(VEL_pin, OUTPUT);
      .
      .
      .
}
```

Listing 3.2: Máxima velocidad giro horario

```
digitalWrite(DIR_pin, HIGH);
analogWrite(VEL_pin, 255); # Maximo valor aceptado, PWM siempre
    encendido
```

Listing 3.3: Detenerce

```
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 3.4, en el cual se consulta una bandera asociada al estado de un valor y en base a este se acciona el motor.

Listing 3.4: Ejemplo uso del motor

```
    .
    .
    .
if (valor > 100) {
    digitalWrite(DIR_pin, HIGH);
    analogWrite(VEL_pin, 255);
} else {
    analogWrite(VEL_pin, 0);
}
    .
    .
    .
```

¿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 especifico). Incluso utilizando las herramientas provistas por el entorno de Arduino, el código no es similar 3.5.

Listing 3.5: Configuración

```
void set_up() {
    .
    .
    .
    Serial.begin(9000);
    .
    .
    .
}
```

Listing 3.6: Máxima velocidad giro horario

```
Serial.write(0xAA);
Serial.write(0x0C);
Serial.write(0x85);
Serial.write(0x7F);
```

Listing 3.7: Detenerse

```
Serial.write(0xAA);
Serial.write(0x0C);
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.

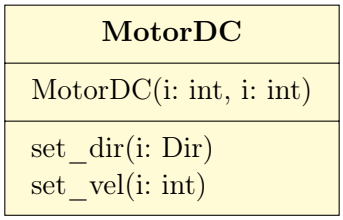
Todos estos inconvenientes son derivados de que el *hardware* comprende un ítem de cambio frecuente; por lo que si seguimos la metodología de Parnas (ver 1.2.1), debemos asignarle un módulo 3.2. Este es una representación virtual de hardware que debe proveer una interfaz lo suficientemente insensible 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. Como afirma Parnas en [Par78; Par77], debemos construir interfaces con la cantidad de mínima de métodos del modo más abstracto posible, sin agregar métodos que puedan ser reemplazados utilizando otros ya definidos, *abstracción* y *encapsulando* con el objetivo de ocultar la implementación. 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.

Donde los parámetros del constructor `MotorDC` son los pines de control, el parámetro de `set_dir` es de tipo `Dir` el cual es un int que puede tomar dos valores uno para el sentido de giro horario y otro anti-horario y por último el input de `set_vel` es un int entre 0 y 255 que representa la velocidad porcentual. Veamos en el ejemplo 3.8 una posible implementación de esta interfaz utilizando el controlador DVR8838.

Listing 3.8: Posible implementación de la interfaz del módulo MotorDC

```
void MotorDC(int dir_pin, int vel_pin) {
```

Figura 3.2: Interfaz MotorDC




```
    this->dir_pin = dir_pin;
    this->vel_pin = vel_pin;
    pinMode(this->dir_pin, OUTPUT);
    pinMode(this->vel_pin, OUTPUT);
}

void set_dir(Dir dir) {
    if (dir == Dir.HORARIO) {
        digitalWrite(DIR_pin, HIGH);
    } else {
        digitalWrite(DIR_pin, DOWN);
    }
}

void set_vel(int vel) {
    if vel > 0 && vel <= 255 {
        analogWrite(VEL_pin, vel);
    }
}
```

Ya son evidentes las primeras ventajas, es mucho más claro en el código 3.9

Listing 3.9: Ejmplo de uso de la interfaz del módulo MotorDC

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

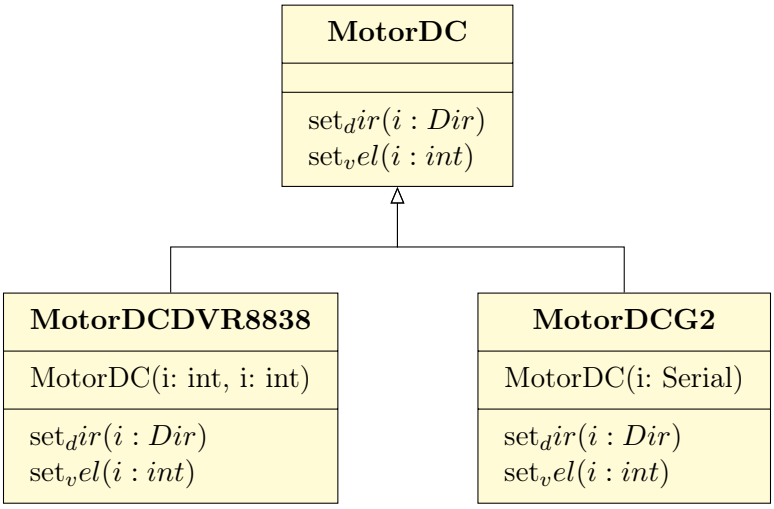
Además si debemos modificar el sentido de giro es tan fácil como cambiar el método `set_dir`, los clientes no notarán el cambio. A su vez, si queremos controlar otro motor podemos hacerlo como en 3.10.

Listing 3.10: Ejemplo control nuevo motor

```
    MotorDC motor_delantero = MotorDC(18, 19)

    motor_delantero->set_dir(Dir.ANTIHORARIO)
    motor_delantero->set_vel(10)
```

Figura 3.3: Módulo MotorDC abstracto y estructura de herencia.



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 definida como en el listing 3.11.

Listing 3.11: Nueva definición set_dir

```
void set_vel(vel: int) {
    if vel > 0 && vel <= 255 {
        this->serial.write(0xAA);
        this->serial.write(0x0C);
        hex_vel = int_to_hex(vel)
        this->serial.write(hex_vel);
    }
}
```

Aún podemos mejorar el diseño haciendo uso del concepto de herencia de interfaz (explicado en la sección 1.2). 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 la estructura de la figura 3.3.

El cliente no tiene noción 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 la IS [SG96; GJM03; BCK03; TMD10] y logramos un diseño orientado al cambio [Gam+95].

3.2 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 último 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 imagen 3.4.

Figura 3.4: Display 7 segmentos 4 digitos

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 comunicación y provee funciones simples de usar, tales como `escribir(i: str): bool` y `limpiar()`. La primera intenta escribir la cadena de caracteres indicada, pero solo lo hace si el display no está mostrando nada. En ese caso devuelve `True` indicando que la acción fue completada con éxito, en caso contrario, es decir el display está mostrando texto al momento de llamar la función, retorna `False`. `limpiar()`, siempre limpia el display. Por lo tanto, se implementó el sistema utilizando las funciones provistas para comunicarse, múltiples módulos llaman esas funciones.

En cierto momento el módulo display dejó de funcionar y se lo 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 el método de escritura devolvía `False` si se quería escribir y ya estaba mostrándose algo en el display, en la nueva si el display está mostrando algo el texto es pisado al momento de imprimir. Pero, provee un nueva método para verificar qué es lo 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 alrededor del sistema agregando la lógica nueva, como en el código 3.12. El cambio puedo parecer no muy profundo, pero hay que tener en cuenta que se deben verificar **todos** los usos de la librería, actualizarlos a mano y re-verificarlos. Además, este es un ejemplo simple, en el mundo real los cambios pueden ser mucho mas profundo y de diferente naturaleza.

Listing 3.12: Ejemplo de modificaciones necesarias para adaptar la nueva librería.

```
libAcme = LibAcme()
libEmca = LibEmca()


// Con ACME

if libAcme.escribir("Hola mundo!") {
  print("El display estaba vacio, se pudo escribir el nuevo
     texto.")
} else {
  print("El display esta ocupado mostrando algo, no se pudo
     escribir")
}

// Con EMCA

if libEmca.get_current() == "" {
  libEmca.imprimir("Hola mundo!")
  print("El display estaba vacio, se pudo escribir el nuevo
     texto.")
} else {
  print("El display esta ocupado mostrando algo, no se pudo
     escribir")
}
```

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**. El cual utilice los métodos provistos por la librería que encapsula el display con el objetivo de proveer la misma interfaz que la librería original. Podemos ver la nueva estructura en 3.7.

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()`

Figura 3.5: Configuración original

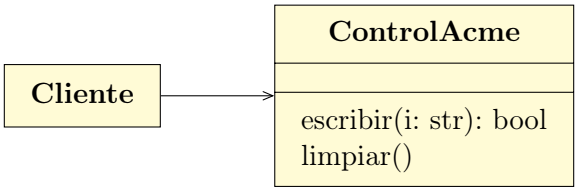
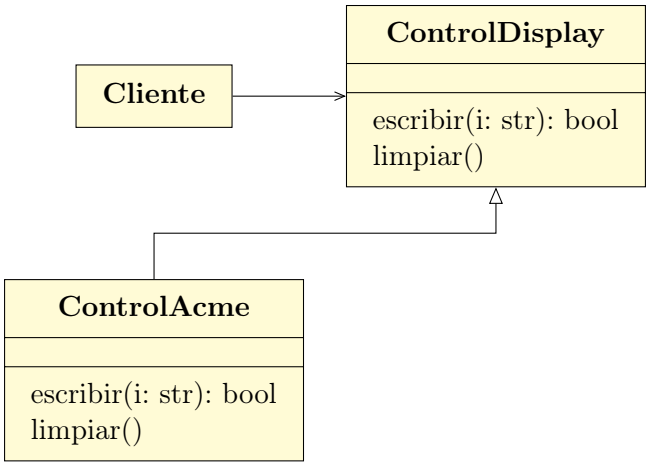


Figura 3.6: Configuración utilizando la solución de la sección anterior 3.1.



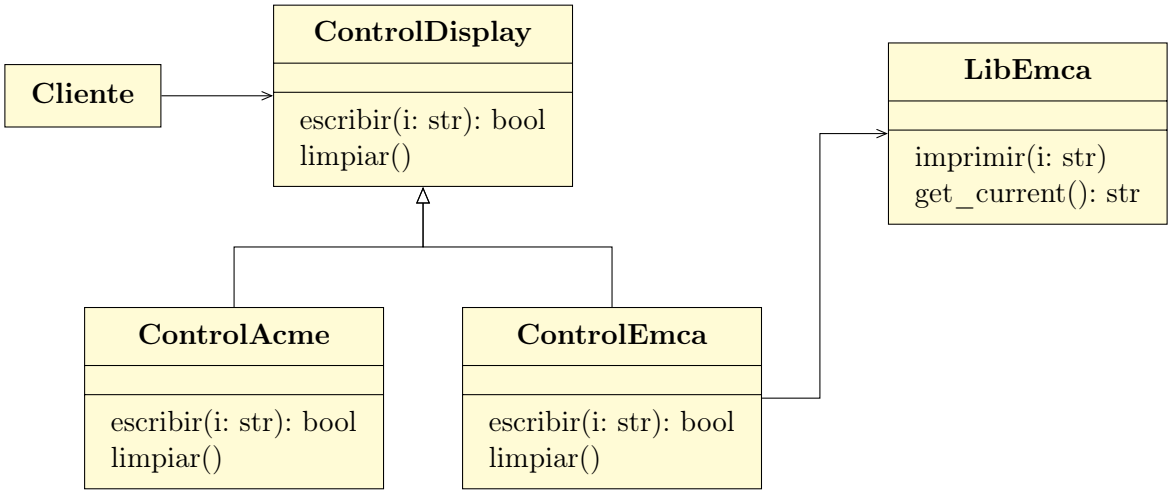
del módulo **ControlEmca** es la del código 3.13.

Listing 3.13: Ejemplo implementación módulo Adapter.

```
bool escribir(char* cadena) {
    if (libEmca.get_current() != "") {
        return False;
    }
    libEmca.imprimir(cadena);
    return True;
}

void limpiar() {
    libEmca.imprimir("");
}
```

Figura 3.7: Configuración nueva



3.3 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 lineas 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).

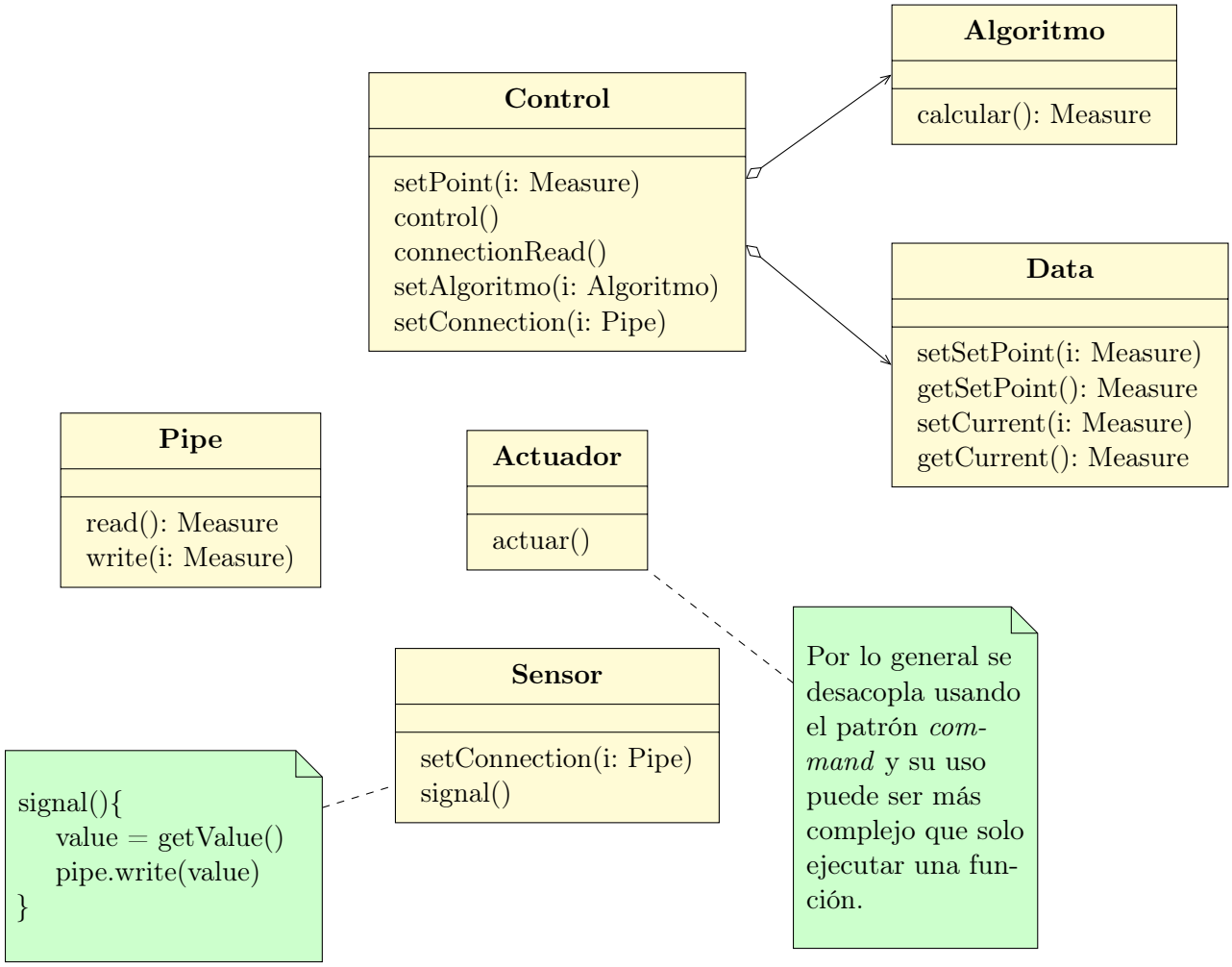
3.3.1 Subsistemas de control

Proponemos al creación de un *subsistema de control* por cada variable a manipular[1] 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* (ver **??**)) y que indique el comienzo de la tarea de control. Un subsistema tiene una estructura similar a la de la arquitectura 1.1, pero su alcance estará restringido a la variable a manipular y las variables medidas relacionadas.

[1]Ver tabla conceptos arquitectura Control de Procesos **??**

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

Figura 3.8: Módulos de un subsistema



La manera en la que un *cliente* utiliza el subsistema para lograr que la propiedad que controla vaya al valor que se desea es de la forma indicada en el código 3.14.

Listing 3.14: Ejemplo de uso del subsistema.

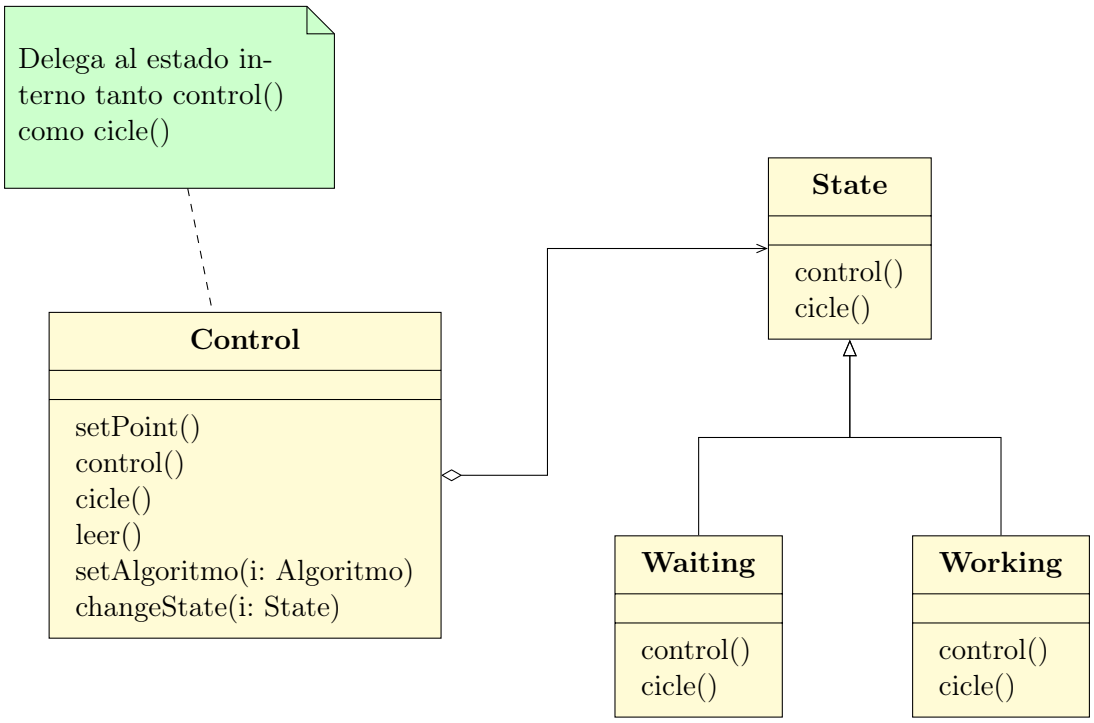
```
control.setPoint(valorDeseado)
sensor.signal()
control.connectionRead()
control.control()
```

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 llego o no al *set-point*. Por otro lado, tenemos **Data** que desacopla el almacenamiento de información

relacionada al subsistema. En el caso mas básico solo tenemos el *set-point*, 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 cierto método 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 *set-point* y se este trabajando para llegar. Para hacerlo utilizamos un nuevo método y el patrón *state* [2].

Figura 3.9: Estructura módulo **Control** extendida con estado.



El cliente ejecuta `control()` y la interrupción que marca el ciclo de trabajo ejecuta `cicle()`. De esta manera, cuando el estado es *waiting*, `cicle()` no hace nada y cuando está 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 *set-point*, `cicle()` cambia el estado a *waiting*.

En el diseño del robot desmalezador[Pom+24] existe un *timer* que desencadena una interrupción cada $1.5ms$ la cual ejecuta el ciclo de control del subsistema de dirección del

[2]La aplicación de este patrón está explicada en la sección 3.6 Máquinas de estado

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 llego al mínimo de diferencia 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*.

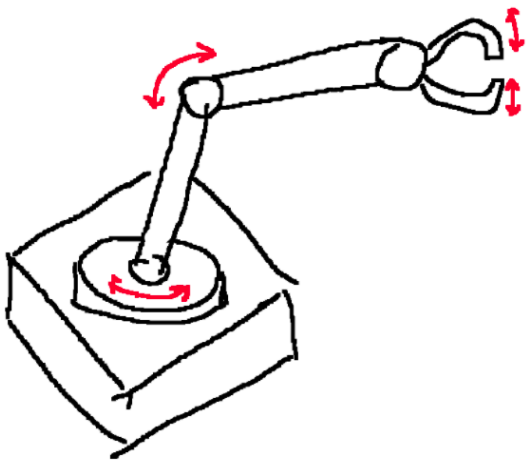
3.3.2 Ejemplo

Veamos como podemos resolver el ejemplo propuesto en el libro de Douglass[Dou11] en la sección de 3.4 Mediator Pattern.

Requisitos

Se necesita desarrollar el software de control de un brazo robótico que consta de tres actuadores, dos 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.

Figura 3.10: Brazo robótico.



Solución libro

El ejemplo que plantea el libro es similar al descrito, pero el brazo robótico tiene más articulaciones y actuadores. Como solución, se propone la creación de un módulo llamado **RobotArmManager**, cuya función es gestionar los actuadores y coordinar su comportamiento. Además, para cada tipo de articulación o actuador, se crea un módulo específico encargado de su control. Este módulo proporciona métodos para consultar el estado actual (posición, longitud, etc.) y otros para configurar un valor similar a un *set-point*. Dichos métodos desempeñan el rol de ejecutores de la acción, es decir, toman un valor *set-point*, ejecutan la acción y retornan `True` si fue satisfactoria, o `False` en caso contrario.

El comportamiento del sistema comienza con la generación de una lista de pasos a realizar. Luego, se itera sobre esta lista ejecutando las acciones definidas en cada paso. Como el sistema debe interrumpir la ejecución si encuentra un error, el **RobotArmManager** verifica el valor de retorno tras cada acción. La ejecución de un movimiento completo finaliza cuando se completan todos los pasos generados previamente por el método `graspAt(i: Coordenadas)`.

La solución propuesta parece estar un nivel por encima de lo que se esperaría para este caso. Aunque no hay suficiente información sobre el hardware del brazo robótico, generalmente mover una articulación no es tan simple como invocar un método. Este proceso suele requerir el control continuo de un motor paso a paso, que opera mediante pulsos que avanzan un paso. Por lo tanto, los módulos encargados de los movimientos probablemente tengan más responsabilidades de las que se plantean en el libro. Además, sería necesario implementar un sistema de control más complejo, utilizando algún tipo de *timer* o espera, para garantizar que el motor paso a paso tenga el tiempo necesario para actuar.

De todas formas, suponiendo que los módulos mencionados se adaptan al hardware subyacente, la manera en la que el **RobotArmManager** interactúa con ellos resulta rígida, tanto por la invocación directa como por la dependencia del valor de retorno. Esto se evidencia en el código resultante, que incluye múltiples sentencias *if* consecutivas.

Por otro lado, un ítem de cambio común son las estructuras de datos, como se mencionó en 1.2.2. Por ello, establecer el uso de una lista directamente en el diseño no responde a una buena práctica.

Es posible que el problema haya sido simplificado con fines didácticos. Sin embargo, la forma planteada parece alejarse de una implementación realista, dejando requisitos menos específicos que podrían dar lugar a diferentes interpretaciones.

Por le lado de la aplicacion del patron *Mediator*, Gamma establece que es aplicable en los siguientes casos:

- Un conjunto de módulos se comunica de maneras bien definidas pero complejas. Las interdependencias resultantes son desestructuradas y difíciles de comprender.
- Reutilizar un módulo resulta complicado porque este se refiere y se comunica con muchos otros módulos.
- Un comportamiento distribuido entre varios módulos debería ser personalizable sin requerir una gran cantidad de subclases.

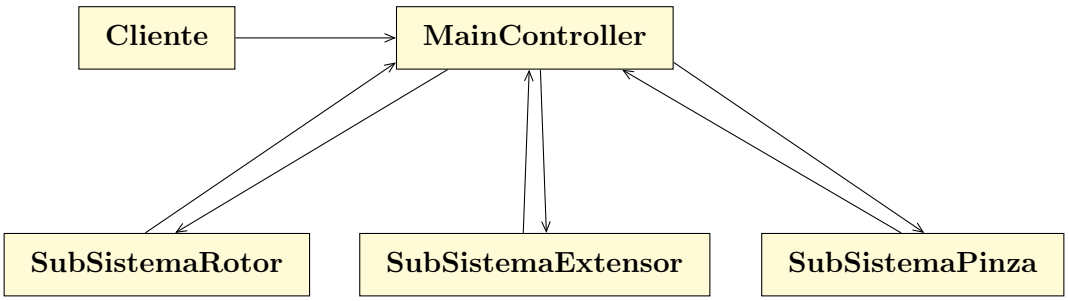
Entiendo que la estructura lograda sea similar al patrón, pero originalmente los módulos encargados de ejecutar tareas relacionadas con un actuador específico no necesitaban comunicarse entre sí. Según el propósito del patrón, Gamma establece que el objetivo principal es

reducir el acoplamiento, evitando que los módulos se refieran directamente entre ellos. Y este objetivo parece estar logrado en el diseño propuesto.

Solución orientada al cambio

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 *set-point* que se está indicando. Para coordinar los subsistemas proponemos un controlador principal llamado **MainController**, el cual provee el método `graspAt(i: Coordenadas)` al cliente, realiza la generación de los pasos y controla su ejecución. La estructura seria algo así:

Figura 3.11: Diagrama componentes sistema brazo



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

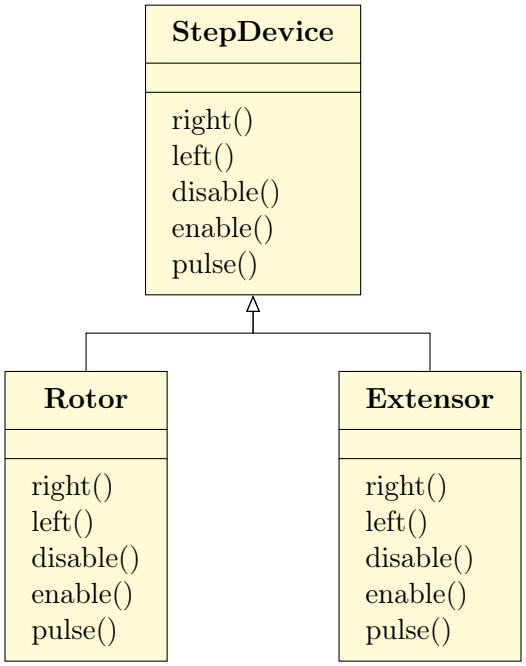
Podemos pensar que el gráfico tiene cierta similaridad conceptual con el patrón *mediator*, donde tenemos un módulo **Mediator** que coordina el trabajo de los **College** en este caso los subsistemas. Gamma indica que el patrón suele ser apropiado para casos en lo que se tiene un conjunto de objetos 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 *set-point* de cada subsistema y desencadena el proceso de control en cada uno. Y viceversa, los subsistemas indican cuando llegan al *set-point* 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 (ver grafico **??**):

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 al método `pulse`. Claramente para que lleguen a la posición deseada puede ser necesario invocar reiteradas veces al método `pulse`. Esto será importante a la hora de diseñar el subsistema que controlará cada dispositivo.

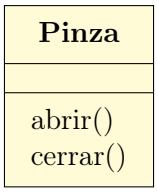
[3] Se utiliza el mismo diseño propuesto para el robot desmalezador[Pom+24].

Figura 3.12: Actuadores paso a paso.



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.

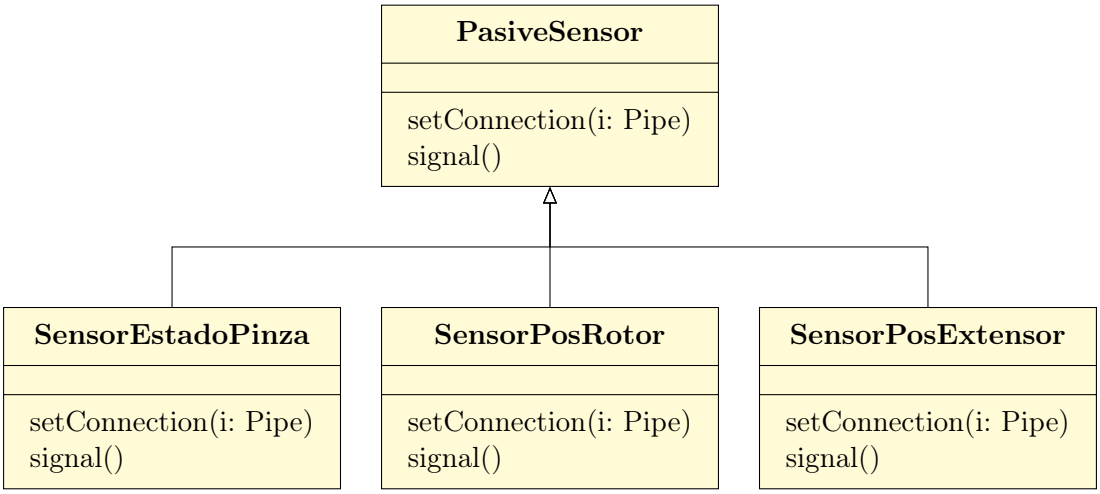
Figura 3.13: Interfaz módulo Pinza.



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

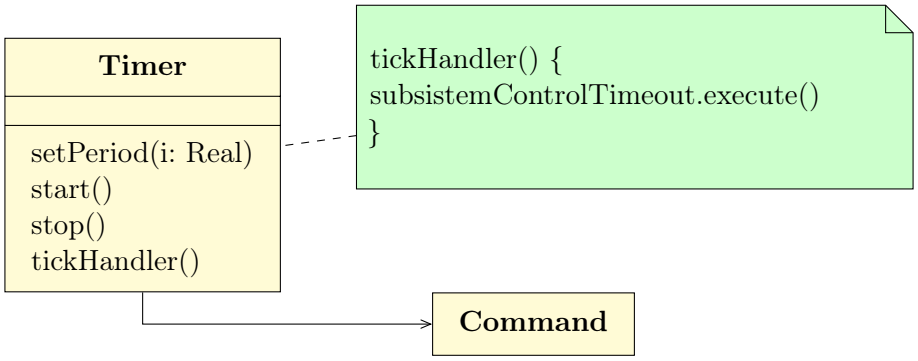
Para completar los módulos que conforman a un subsistema de control nos falta mostrar como se define el **Controller**. Vamos a diferenciar dos tipos de **Controllers**, uno para los sensores que requieren un control durante un determinado lapso de tiempo para poder llegar a su *set-point* y los que 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

Figura 3.14: Sensores



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.

Figura 3.15: Timer

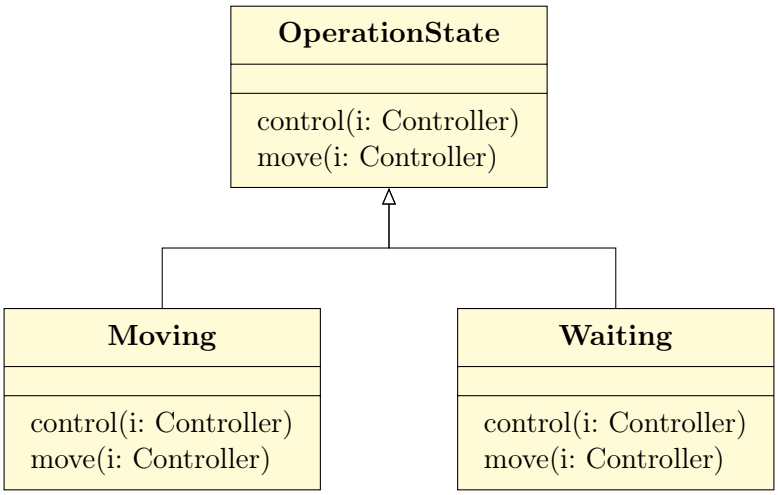


El método `tickHandler()` ejecutará *Commands* por cada subsistema que lo necesite, por lo tanto se iniciará en cada uno el ciclo de control. En particular, la orden la usamos para desacoplar como es invocado el inicio del **Timer**.

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 desencadene la ejecución del ciclo el **Controller** ignorará la orden. Para esto hacemos uso del patrón *State* de Gamma[Gam+95], el cual está explicado en la sección 3.6. En particular, el **Controller** delegará dos funciones de su interfaz al *state*,

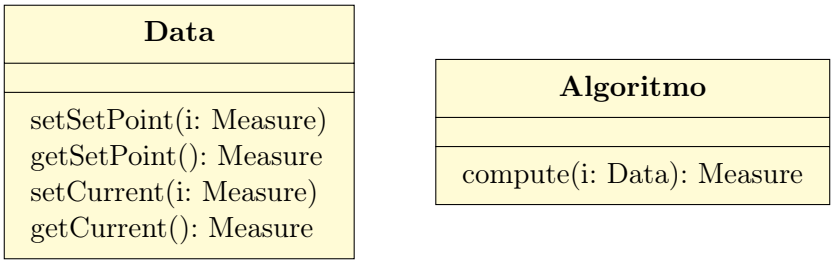
`control()` y `move()` (luego lo veremos en más detalle).

Figura 3.16: Módulos necesarios para complementar el Controller



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.

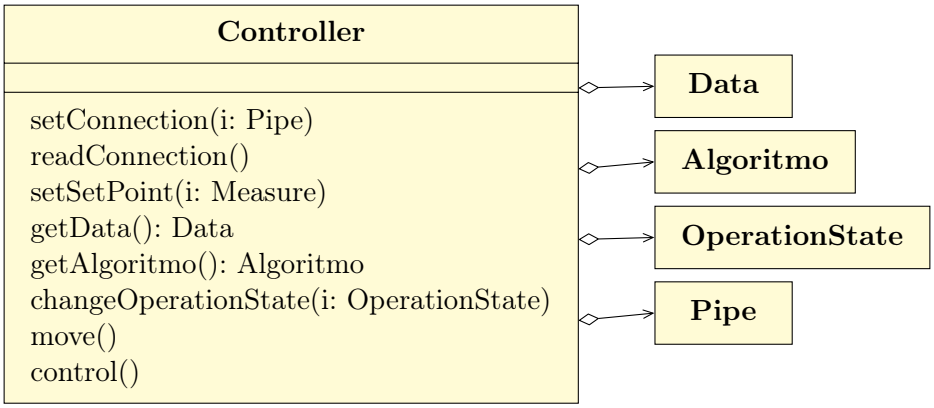
Figura 3.17: Módulos complementarios a Controller.



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

Figura 3.18: 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 el método ejecutado 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 el método `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 valores actuales, el *set-point* 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 del metodo `control()` para el caso del subsistema del **Rotor** y estado de operación *Waiting*, puede ser la del código 3.15 (recordar que el *set-point* del subsistema del rotor es una medida en grados).

Listing 3.15: Ejemplo implementación control

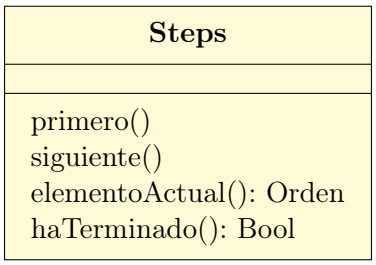
```
1    void control(Controller controller) {
2        setPoint = data.getSetPoint();
3        current = data.getCurrent();
4        dif = controller.algoritmo.calculate()
5        if abs(dif) > LIMIT_ACCEPT && dif > 0 {
6            rotor.left();
7            rotor.pulse;
8            controller.changeOperationState(moving);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

```
            return;
        }
        if abs(dif) > LIMIT_ACCEPT && dif < 0 {
            rotor.right();
            rotor.pulse;
            controller.changeOperationState(moving);
            return;
        }
    }
```

El método `move` tendrá un comportamiento similar pero en el aso 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 sera el encargado de brindar una interfaz al cliente y a su vez coordinar cada subsistema. Recordando los requerimientos, sabemos que tenemos una función que generar 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[Gam+95], en donde una orden al ser ejecutada desencadena una ejecución del resto de las ordenes. Veamoslo en los gráficos 3.19 y 3.20.

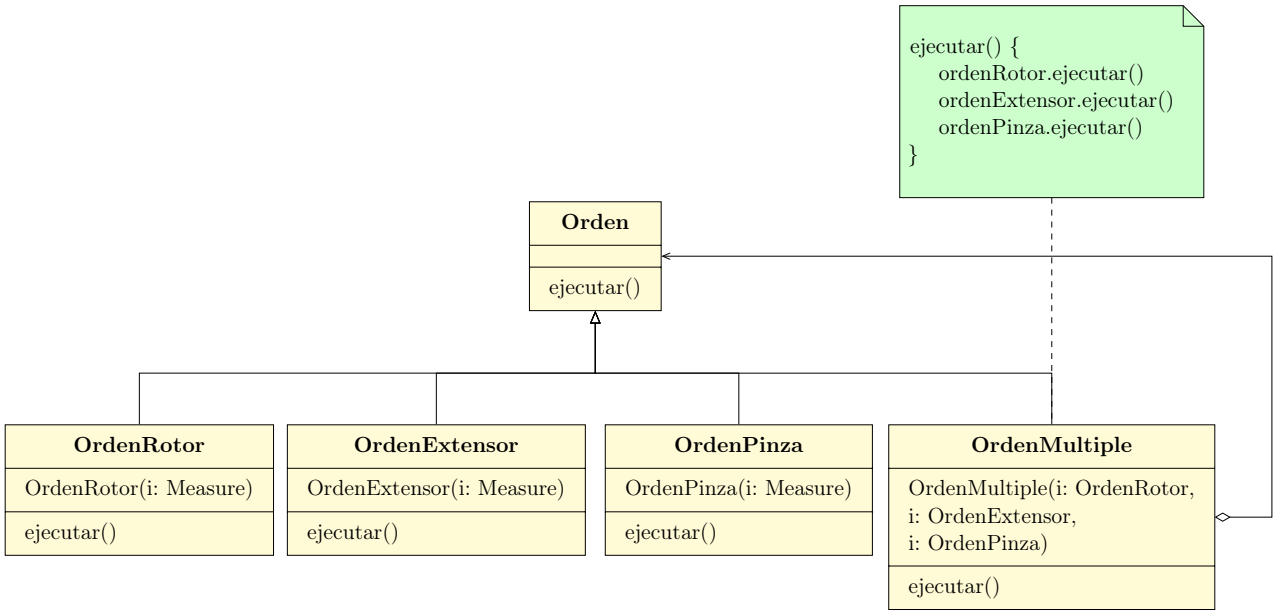
Figura 3.19: Iterator



Tenemos un iterado 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 ??.

Figura 3.20: Orden

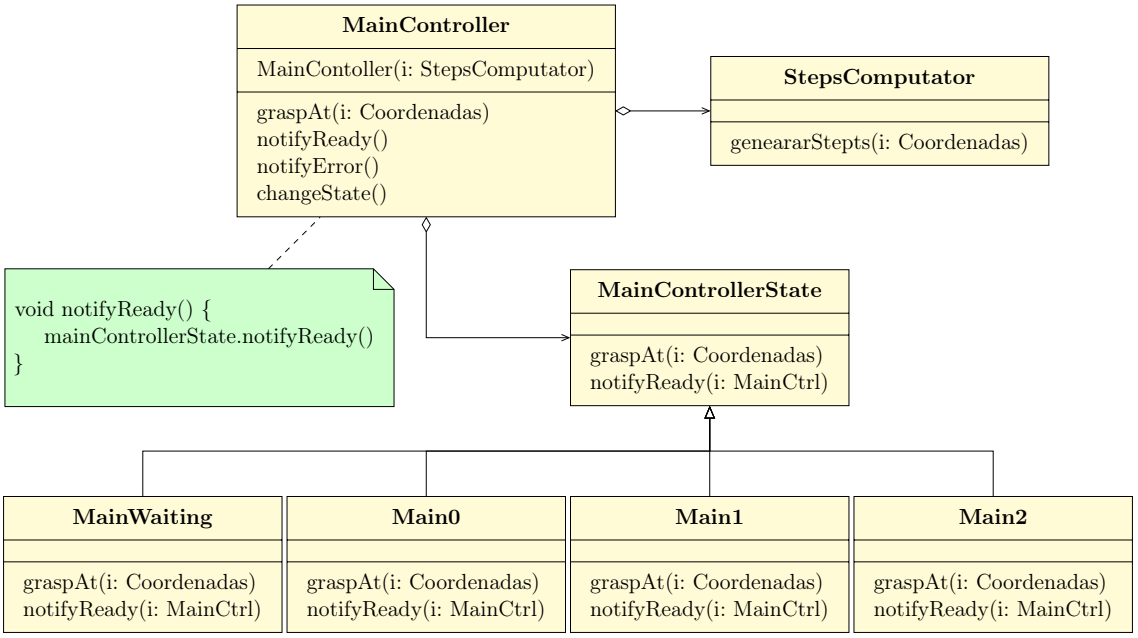


Listing 3.16: Ejemplo OrdenRotor

```
void ejecutar {
    rotorController.setSetPoint(setPoint)
    rotorSensor.signal()
    rotorController.readConnection()
    rotorController.control()
}
```

Por lo que, el estado Waiting no implementa `notifyOrder` y a su vez 0, 1 y 2 no implementan `graspAt`. En cambio, Waitating 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 transaccionamos 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 iterador 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 al método `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.

Figura 3.21: MainController



3.4 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, **Pin** que es la representación en el diseño del hardware asociado al dispositivo que genera e introduce la interrupción, esta 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 un método que permite obtener el valor recibido.

Figura 3.22: Transiciones de estados del MainController

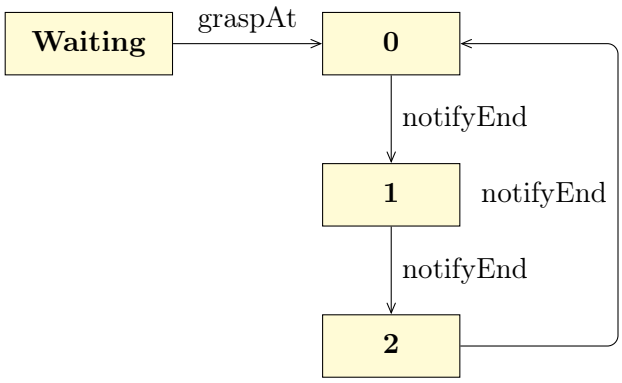
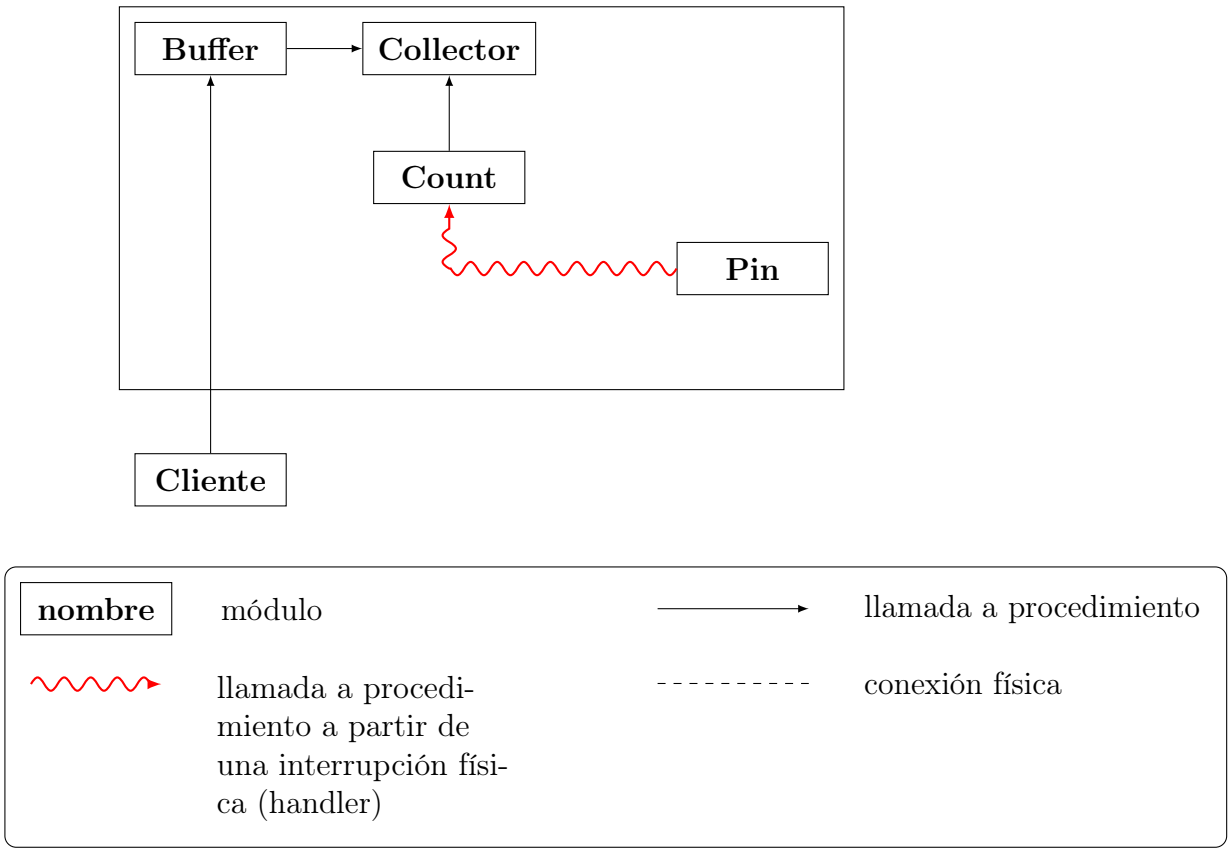


Figura 3.23: Componente sensor activo.



Cuando es invocada toma los valores necesarios que se almacenan en el **Collector** y realiza los cómputos 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**:

- `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):

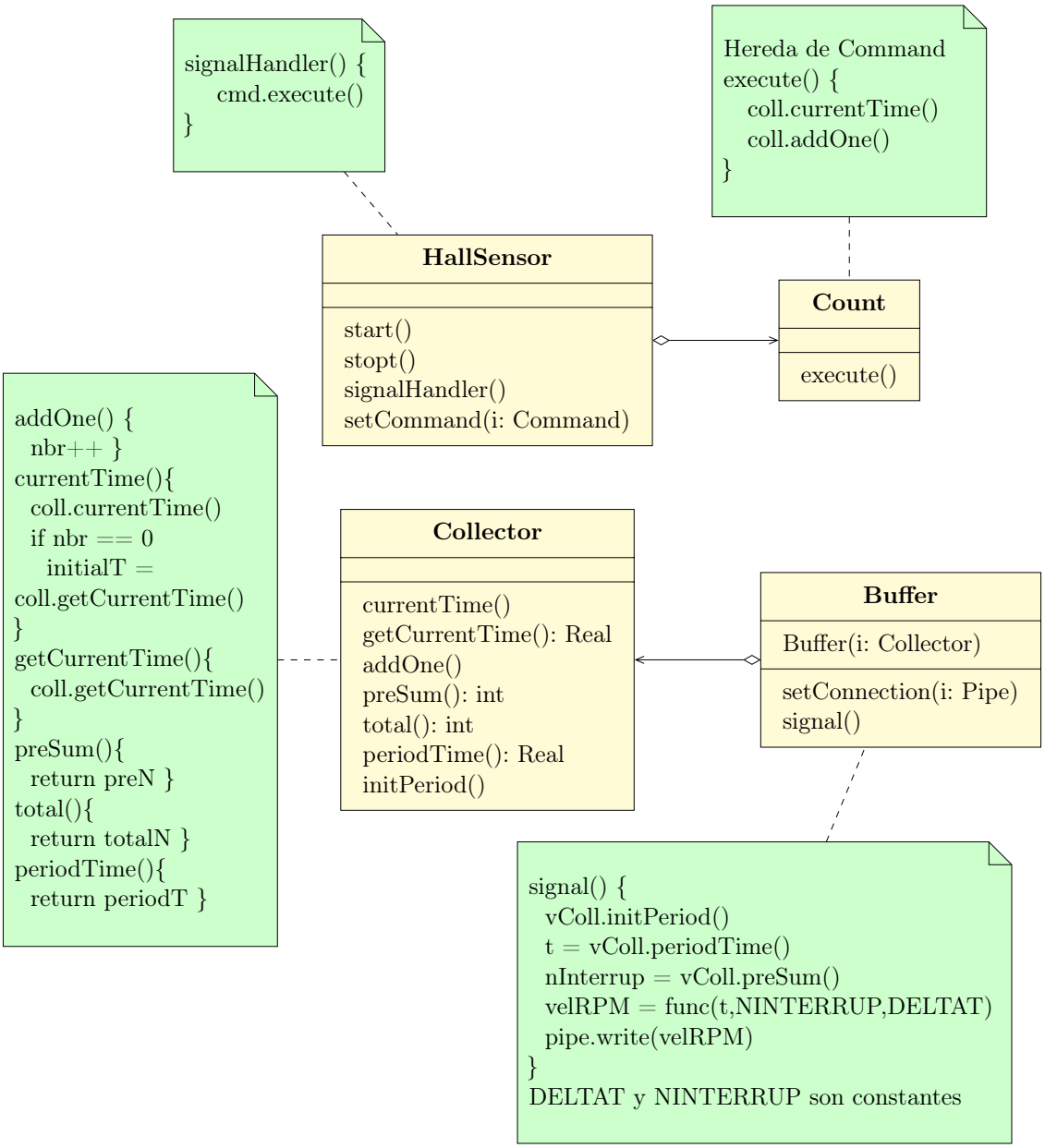
Periodicidad

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 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.

Figura 3.24: Ejemplo



3.5 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 3.17: Código ejemplo

```
void ButtonDriver_eventReceive(ButtonDriver* const me) {
    Timer_delay(me->itsTimer, DEBOUNCE_TIME);
    if (Button_getState(me->itsButton) != me->oldState) {
        /* must be a valid button event */

        me->oldState = me->itsButton->deviceState;

        if (!me->oldState) {

            /* must be a button release, so update toggle value
               */
            if (me->toggleOn) {
                me->toggleOn = 0; /* toggle it off */
                Button_backlight(me->itsButton, 0);
                MicrowaveEmitter_stopEmitting(me->
                   itsMicrowaveEmitter);
            }
            else {
                me->toggleOn = 1; /* toggle it on */
                Button_backlight(me->itsButton, 1);
                MicrowaveEmitter_startEmitting(me->
                   itsMicrowaveEmitter);
            }
        }
        /* if  i t s  not a button release, then it must
        be a button push, which we ignore.
        */
    }
}
```

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 común, por ejemplo en [Gan04] se realiza un estudio de las diferentes formas en las que se manifiesta este inconveniente. Además, presenta algoritmos utilizados para determinar la ocurrencia y poder ignorarla.

3.6 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 seria 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.º "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:

```
calentar() {
    if (estado == PuertaAbierta) {
        return
    } else {
        magnetron.encender()
    }
}
```

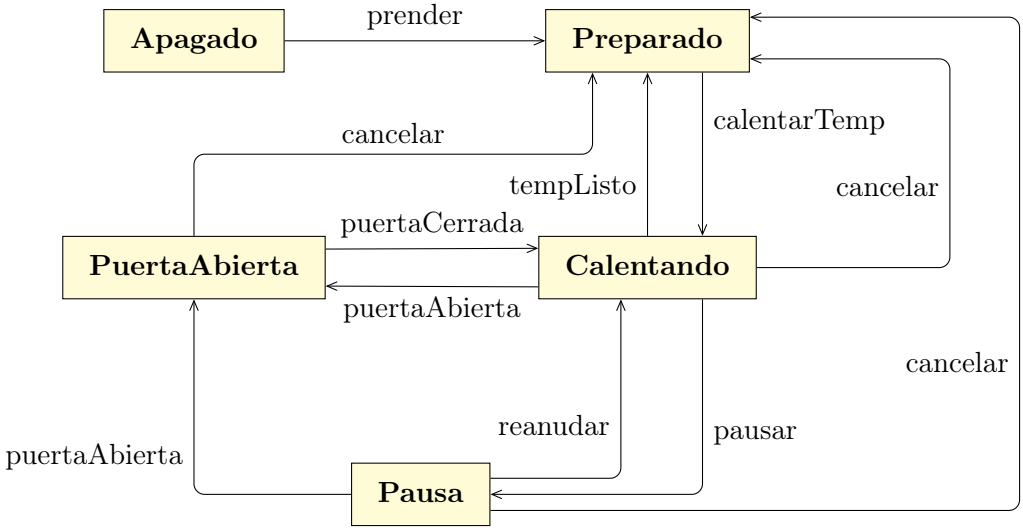
Este enfoque de es una solución directa. Sin embargo, presenta algunas desventajas cuando se desea modificar o extender el código. Por ejemplo, si se necesita agregar nuevos estados o cambiar el comportamiento del sistema, este método puede volverse rápidamente difícil de gestionar. Una de las principales 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 una sucesión de transiciones. Considere el siguiente "*state chart*" extraído del libro que describe el funcionamiento del microondas:

Figura 3.25: State chart 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 así ejecuta el método de "salida" del estado actual, ejecuta el método 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 quitarlo 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                 else
14                     takeTransition =(me->table[me->stateID][e.
                         eType].guardPtr(me));
15                 if (takeTransition) {
16                     if (me->table[me->stateID][e.eType].
                         exitActionPtr != NULL)
17                         if (me->table[me->stateID][e.eType].
                             exitActionPtr->nParams == 0)
18                             me->table[me->stateID][e.eType].
                                 exitActionPtr->aPtr.a0(me);
19                         else
20                             me->table[me->stateID][e.eType].
                                 exitActionPtr->aPtr.a1(me, e.ed.
                                 c);
21                     if (me->table[me->stateID][e.eType].
```

```
                        transActionPtr != NULL)
22                       if (me->table[me->stateID][e.eType].
                            transActionPtr->nParams == 0)
23                           me->table[me->stateID][e.eType].
                                transActionPtr->aPtr.a0(me);
24                       else
25                           me->table[me->stateID][e.eType].
                                transActionPtr->aPtr.a1(me, e.ed
                                .c);
26                   if (me->table[me->stateID][e.eType].
                        entryActionPtr != NULL)
27                       if (me->table[me->stateID][e.eType].
                            entryActionPtr->nParams == 0)
28                           me->table[me->stateID][e.eType].
                                entryActionPtr->aPtr.a0(me);
29                       else
30                           me->table[me->stateID][e.eType].
                                entryActionPtr->aPtr.a1(me, e.ed
                                .c);
31                   me->stateID = me->table[me->stateID][e.
                        eType].newState;
32               }
33           }
34       }
35   }
36
37 Mutex_release(me->itsMutex);
38 }
```

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 dinamicamente 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 agrear una referencia de los posibles siguientes estados, para que en caso de recibir el evento adecuado pueda invocar el método 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:

```
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
}
```

Y para llamar a una transición de estado:

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

Este uso del patrón *State* también es propuesto en [Dou11, *Chapter 10 : Finite State Machine Patterns Part III: New Patterns as Design Components*].

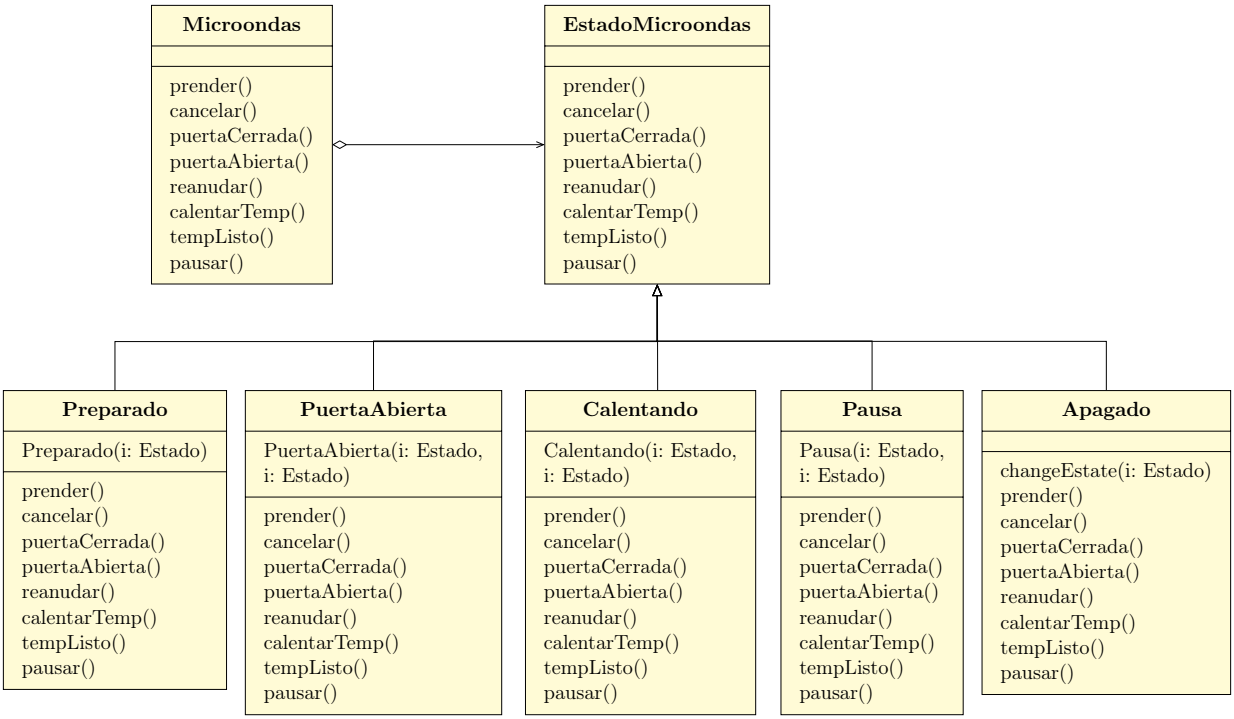
Volviendo al ejemplo del microondas, aplicando este enfoque de diseño, tendremos un módulo que provee un método que maneja cada evento y otro que delega al estado cada manejo de evento.

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

```
puertaAbierta() {
    magnetron.calentar()
    microondas.changeEstate(calentando)
}
```

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.

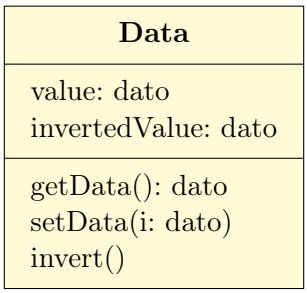
Todo esto hace que el sistema sea más robusto y adaptable, ya que los cambios de estado y las transiciones se manejan de manera interna sin depender de verificaciones externas. Cuando es necesario modificar el comportamiento de un estado, simplemente se actualiza la implementación del módulo correspondiente, 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.

3.7 Integridad de la información

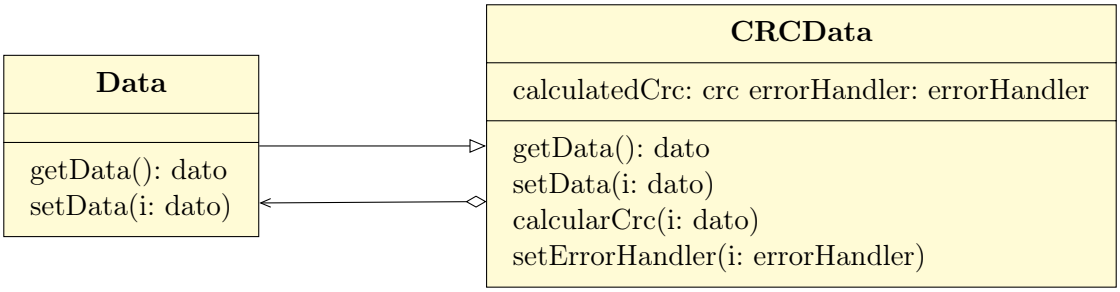
Un pulso electromagnético o fallas en el hardware pueden causar daños información dejándola comprometida, por ejemplo un *bit flip*. Si los datos afectados son críticos el problema puede derivar en un error grave del sistema. Para hacer frente a esto existen diferentes técnicas, desde calcular *checksums* 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 cómputos de verificación.



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, ademas de poder hacerlo de manera dinámica, es que mantenemos separadas las responsabilidades de cada módulo. A diferencia de los 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.

Figura 3.26: Ejemplo *Decorator* integridad de la información



```
CRCData:setData(i: dato){
    calculatedCrc = calcularCrc(i)
    data.setData(dato)
}

CRCData:getData() {
    dato = data.getData()
    if calculateCrc(dato) == calculatedCrc
        return dato
    else
        errorHandler()
}
```

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.

3.8 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 tiene 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.

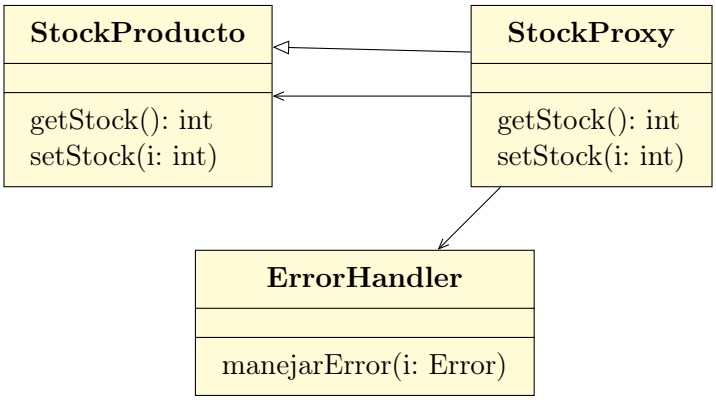
Este enfoque es usado también para otro tipo de funciones como aquellas que requieren que sus argumentos cumplan cierta precondición para poder computar o realizar su trabajo. Y como Douglass comenta, esta es la realidad de la mayoría de las funciones.

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 elegante[4]: 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.

3.9 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 cómputos 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 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.

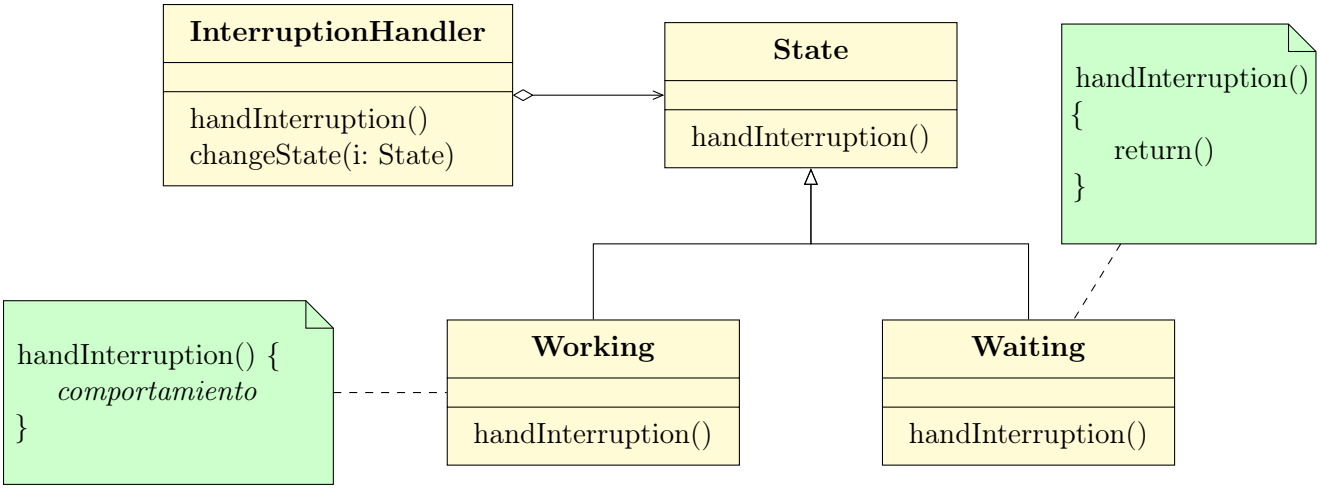
[4]Incorporating Graceful Degradation into Embedded System Design

- 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 3.27: 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

Patrones de diseño de Gamma

A.1 Adapter

Intención

Convierte la interfaz de una clase en otra interfaz que los clientes esperan, permitiendo que clases con interfaces incompatibles trabajen juntas. Es una solución para integrar clases existentes sin modificar su código original, asegurando que cumplan con los requisitos de una aplicación específica.

Aplicabilidad

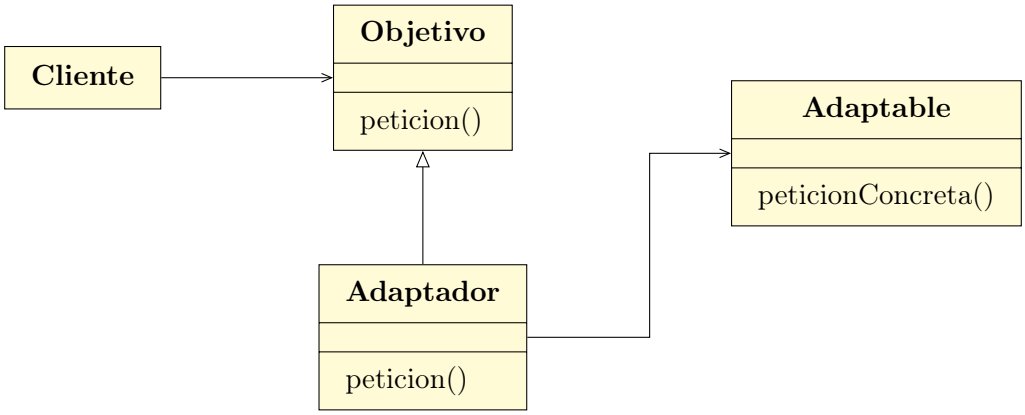
- Se desea usar una clase existente cuya interfaz no coincide con la requerida.
- Se necesita crear una clase reutilizable que coopere con clases no relacionadas o no previstas inicialmente.
- Se necesita adaptar varias subclases existentes sin modificar su interfaz de manera individual.

Participantes

- **Objetivo**
  Define la interfaz específica del dominio que el cliente utiliza.
- **Clientes**
  Colabora con objetos que cumplen con la interfaz del **Objetivo**.
- **Adaptable**
  Define una interfaz existente que necesita ser adaptada.
- **Adaptador**
  Adapta la interfaz del **Adaptable** para que cumpla con la interfaz del **Objetivo**.

Estructura

Figura A.1: Estructura patrón **Adapter**



A.2 Command

Intención

El patrón encapsula una solicitud como un módulo, permitiendo parametrizar clientes con diferentes solicitudes, encolar o registrar solicitudes, y admitir operaciones reversibles. Este enfoque facilita la creación de sistemas flexibles y extensibles que manejan comandos de manera uniforme.

Aplicabilidad

- Parametrizar objetos con una acción a realizar. Esta parametrización puede expresarse en un lenguaje procedimental mediante una función de callback, es decir, una función registrada para ser llamada posteriormente. Los comandos representan una solución orientada a objetos que reemplaza los callbacks.
- Especificar, encolar y ejecutar solicitudes en diferentes momentos. Un módulo Command puede tener una vida útil independiente de la solicitud original. Si el receptor de una solicitud puede representarse de forma independiente del espacio de direcciones, puedes transferir un objeto Command a otro proceso y ejecutar la solicitud allí.
- Soportar la funcionalidad de deshacer (ündo"). El método Execute del Command puede almacenar el estado necesario para revertir sus efectos. La interfaz del Command debe incluir una operación Unexecute para revertir los efectos de una ejecución previa. Los comandos ejecutados se almacenan en una lista de historial, lo que permite deshacer y rehacer a múltiples niveles navegando hacia adelante y hacia atrás en la lista mientras se llaman a Unexecute y Execute.
- Registrar cambios para que puedan reaplicarse en caso de una falla del sistema. Al ampliar la interfaz del Command con operaciones de carga y almacenamiento, puedes mantener un registro persistente de los cambios. Recuperar un sistema tras una falla implica recargar los comandos registrados desde el disco y reejecutarlos mediante la operación Execute.

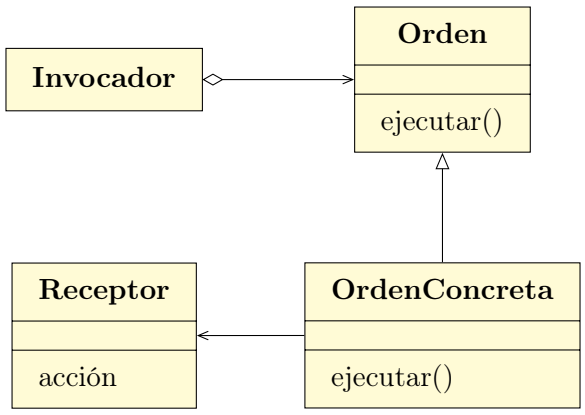
- Estructurar un sistema en torno a operaciones de alto nivel basadas en operaciones primitivas. Esta estructura es común en sistemas de información que admiten transacciones. Una transacción encapsula un conjunto de cambios a los datos. El patrón Command proporciona una forma de modelar transacciones, ya que los comandos tienen una interfaz común, lo que permite invocar todas las transacciones de la misma manera. Además, el patrón facilita la extensión del sistema con nuevas transacciones.

Participantes

- **Orden**
  Declara una interfaz para ejecutar una operación.
- **OrdenConcreta**
  Define una asociación entre un módulo receptor (Receiver) y una acción. Implementa el método Execute invocando las operaciones correspondientes en el receptor.
- **Cliente**
  Utiliza OrdenConcreta y configura su receptor.
- **Invocador**
  Solicita al comando que lleve a cabo la solicitud.
- **Receptor**
  Conoce cómo realizar las operaciones asociadas con la ejecución de una solicitud. Cualquier clase puede actuar como un receptor.

Estructura

Figura A.2: Estructura patrón **Command**



A.3 State

Intención

Permitir que un módulo altere su comportamiento cuando su estado interno cambia. El módulo parecerá cambiar de clase.

Aplicabilidad

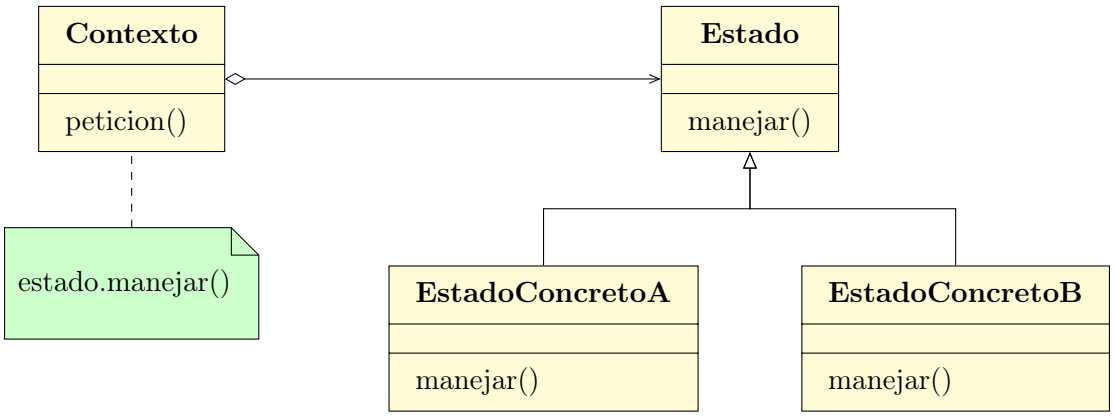
- El comportamiento de un objeto depende de su estado, y debe cambiar su comportamiento en tiempo de ejecución según ese estado.
- Las operaciones suelen tener declaraciones condicionales grandes y complejas que dependen del estado del módulo. Este estado generalmente está representado por una o más constantes enumeradas. Frecuentemente, varias operaciones comparten la misma estructura condicional. El patrón separa cada rama de la estructura condicional en una clase independiente. Esto permite tratar el estado del módulo como un módulo por derecho propio, que puede variar independientemente de otros módulos.

Participantes

- **Contexto**
  Define la interfaz de interés para los clientes. Mantiene una instancia de una subclase de EstadoConcreto que define el estado actual.
- **Estado**
  Define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto.
- **EstadoConcreto**
  Cada subclase implementa un comportamiento asociado con un estado del contexto.

Estructura

Figura A.3: Estructura patrón **State**



A.4 Mediator

Intención Define un modulo que encapsula cómo interactúa un conjunto de módulos. Fomenta un acoplamiento débil al evitar que los objetos se refieran explícitamente entre sí, y permite variar sus interacciones de manera independiente.

Aplicabilidad

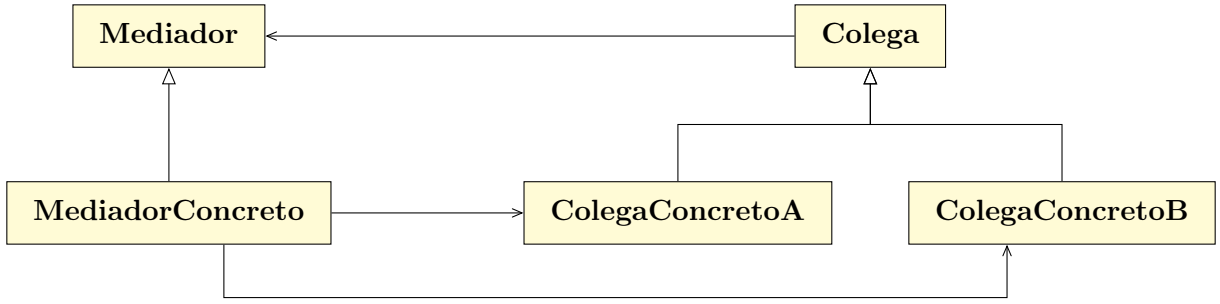
- Un conjunto de módulos se comunica de maneras bien definidas pero complejas. Las interdependencias resultantes son desestructuradas y difíciles de comprender.
- Reutilizar un módulo resulta complicado porque este se refiere y se comunica con muchos otros módulos.
- Un comportamiento distribuido entre varios módulos debería ser personalizable sin requerir una gran cantidad de subclases.

Participantes

- **Mediador** Define una interfaz para comunicarse con los módulos Colega.
- **MediadorConcreto** Implementa un comportamiento cooperativo coordinando los módulos Colega. Conoce y mantiene a sus colegas.
- **Colega** Conoce a su objeto Mediator y se comunica con se siempre que, de otra forma, se habría comunicado con otro Colega.

Estructura

Figura A.4: Estructura patrón **Mediator**



A.5 Decorator

Intención Agregar responsabilidades adicionales a un objeto de manera dinámica. Los decoradores ofrecen una alternativa flexible a la herencia para extender la funcionalidad.

Aplicabilidad

- Agregar responsabilidades a objetos individuales de manera dinámica y transparente, es decir, sin afectar a otros objetos.
- Para responsabilidades que pueden ser eliminadas.
- Cuando la extensión mediante herencia es impracticable. A veces, es posible tener una gran cantidad de extensiones independientes, lo que produciría una explosión de subclases para admitir cada combinación. O bien, la definición de una clase puede estar oculta o no estar disponible para la subclase.

Participantes

- **Componente** Define la interfaz para módulos a los que se les pueden agregar responsabilidades de manera dinámica.
- **ComponenteConcreto** Define un módulo al que se le pueden adjuntar responsabilidades adicionales.
- **Decorador** Mantiene una referencia a un módulo **Componente** y define una interfaz que se ajusta a la interfaz del **Componente**.
- **DecoradorConcreto** Agrega responsabilidades al componente.

Estructura

A.6 Proxy

Intención Proporcionar un sustituto o marcador de posición para otro objeto con el fin de controlar el acceso a este.

Aplicabilidad

El patrón Proxy es aplicable siempre que se necesite una referencia más versátil o sofisticada a un objeto que un simple puntero. Estas son varias situaciones comunes en las que se puede aplicar el patrón:

1. Proxy Remoto:

   Proporciona un representante local para una instancia en un espacio de direcciones diferente.

2. Proxy Virtual:

   Crea instancias costosas bajo demanda.

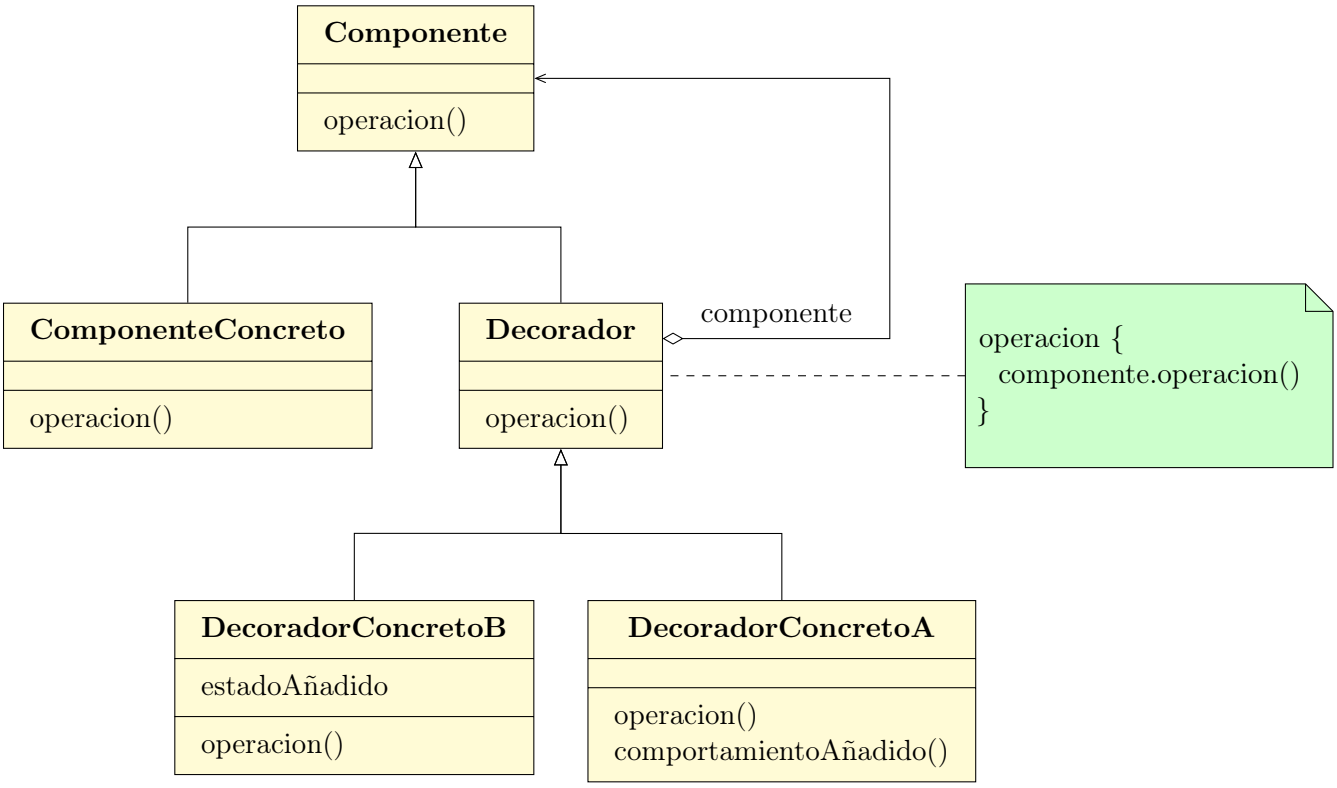
3. Proxy de Protección:

   Controla el acceso a la instancia original. Es útil cuando las instancias necesitan diferentes derechos de acceso.

4. Referencia Inteligente:

   Es un reemplazo para un puntero básico que realiza acciones adicionales cuando se accede a un objeto. Usos típicos incluyen:

Figura A.5: Estructura patrón **Decorator**



- Contar el número de referencias a la instancia real para que pueda ser liberado automáticamente cuando no queden más referencias (también llamado punteros inteligentes).
- Cargar una instancia persistente en memoria cuando se referencia por primera vez.
- Verificar que la instancia real esté bloqueado antes de acceder a ella, para asegurar que ningún otra instancia pueda modificarlo.
- Este patrón ofrece flexibilidad, seguridad y eficiencia en la gestión de interacciones entre instancias.

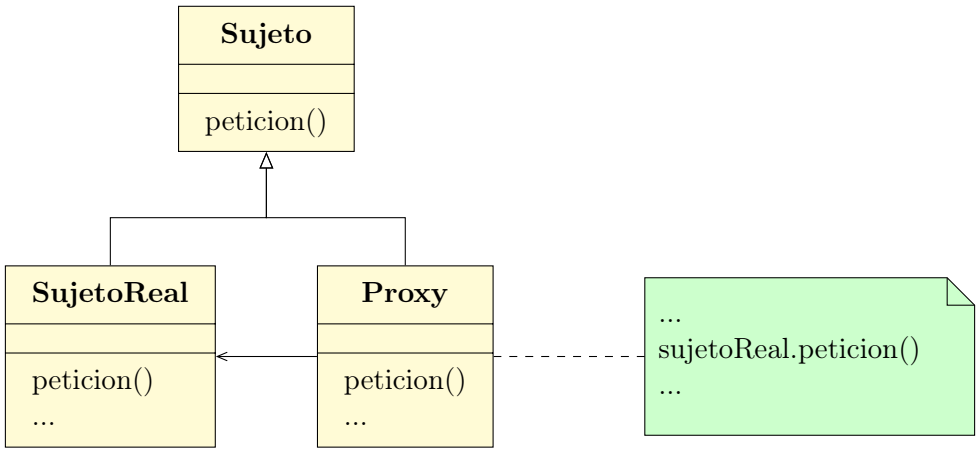
Participantes

- **Proxy** Mantiene una referencia que permite al proxy acceder al sujeto real. El Proxy puede referirse a un **Sujuteo** si las interfaces de **SujetoReal** y **Sujeto** son las mismas. Proporciona una interfaz idéntica a la de **Sujeto**, de manera que un proxy puede ser sustituido por el sujeto real. Controla el acceso al sujeto real y puede ser responsable de crearlo y eliminarlo.
- **Sujeto** Define la interfaz común para **SujetoReal** y **Proxy**, de modo que un **Proxy** se pueda usar en cualquier lugar donde se espere un **SujetoReal**.

- **RealSubject** Define el objeto real que el proxy representa.

Estructura

Figura A.6: Estructura patrón **Proxy**



Glosario

Arduino Uno Microcontrolador basado en el microchip ATmega328 y desarrollado por Arduino. La placa está equipada con conjuntos de pines de E/S digitales y analógicas que pueden conectarse a varias placas de expansión y otros circuitos. La placa tiene 13 pines digitales, 6 pines analógicos y programables con el Arduino IDE (Entorno de desarrollo integrado) a través de un cable USB tipo B. 26

DBOI Diseño basado en ocultación de la información. 16

DC Corriente continua. 25

DRV8838 Controlador para un único motor de corriente continua (DC) con escobillas. 25

microcontrolador Circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria. . 26

Referencias

[Pom+24] L. Pomponio, M. Cristiá, E. R. Sorazábal y M. García. "Reusability and modifiability in robotics software (extended version)". En: (2024). URL: https://arxiv.org/abs/2409.07228.

[Dou11] B. P. Douglass. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Oxford, UK: Newnes, 2011.

[Whi11] E. White. *Making Embedded Systems: Design Patterns for Great Software*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[Noe05] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Boston, MA: Elsevier, 2005.

[LS17] E. A. Lee y S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. Cambridge, MA: MIT Press, 2017.

[SG96] M. Shaw y D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-182957-2.

[GJM03] C. Ghezzi, M. Jazayeri y D. Mandrioli. *Fundamentals of software engineering (2nd. Edition)*. Prentice Hall, 2003.

[BCK03] L. Bass, P. Clements y R. Kazman. *Software architecture in practice*. English. Boston: Addison-Wesley, 2003. ISBN: 0321154959 9780321154958.

[TMD10] R. N. Taylor, N. Medvidovic y E. M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010. ISBN: 978-0-470-16774-8. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000180.html.

[22] "Dorna 2 Python API". En: (2022). URL: https://github.com/dorna-robotics/dorna2-python/blob/master/dorna2/dorna.py#L93.

[20] "ERDOS". En: (2020). URL: https://github.com/erdos-project/erdos/blob/master/python/erdos/timestamp.py#L34.

[Gam+95] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[BHS07] F. Buschmann, K. Henney y D. C. Schmidt. *Pattern-oriented software architecture, 4th Edition*. Wiley series in software design patterns. Wiley, 2007. ISBN: 9780470059029. URL: https://www.worldcat.org/oclc/314792015.

[Par72] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. En: *Commun. ACM* 15.12 (dic. de 1972), págs. 1053-1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: https://doi.org/10.1145/361598.361623.

[Par78] D. L. Parnas. “Designing software for ease of extension and contraction”. En: *ICSE ’78: Proceedings of the 3rd international conference on Software engineering*. Atlanta, Georgia, United States: IEEE Press, 1978, págs. 264-277. ISBN: none.

[CN02] P. Clements y L. M. Northrop. *Software product lines - practices and patterns*. SEI series in software engineering. Addison-Wesley, 2002. ISBN: 978-0-201-70332-0.

[Mey97] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. ISBN: 0-13-629155-4. URL: http://www.eiffel.com/doc/oosc/page.html.

[Par77] D. L. Parnas. “Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems”. En: *NRL Report No. 8047* (1977). Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

[Gan04] J. G. Ganssle. *A Guide to Debouncing*. The Ganssle Group, 2004.