Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Patrones de diseño aplicados a sistemas embebidos de control

Alumno:
Petruskevicius Ignacio

Directora:
Dra. Pomponio Laura

Mayo 2025

Patrones de diseño aplicados a sistemas embebidos de control

Ignacio Petruskevicius

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 un criterio de 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 prácticas de la Ingeniería de Software (IS). Los problemas fueron extraídos de distintos libros [Dou11][Whi11a] los cuales a su vez intentan dar soluciones que, como luego se verá, no siempre se alinean con los estándares de la IS.

Índice general

Índice de figuras

Índice de cuadros

Índice de códigos

Capítulo 1

Introducción

El principal objetivo de la investigación realizada en [Pom+24] fue diseñar el software de control para un robot desmalezador que corre en un Unidad de Microcontrolador (MCU) situado en el robot en cuestión. El diseño consiste en construir y documentar módulos e interfaces siguiendo las metodologías de la Ingeniería de Software (IS), incluyendo el uso de patrones de diseño y estilos arquitectónicos de software para lograr cumplir los requerimientos propuestos. Entre ellos, llevar a cabo ordenes propuestas desde una PC o un control remoto.

Para diseñar este sistema se optó por emplear un estilo arquitectónico de control de procesos [SG96a, pág. 27] y múltiples patrones de diseño clásicos descritos en [Gam+95], como los patrones *state*, *command* o *strategy*.

Esta tesina se propuso dar respuesta a las siguientes preguntas: ¿existen patrones de diseño específicos para robótica, sistemas de control y/o sistemas embebidos? ¿Están debidamente documentados? ¿Pueden ser aplicados al diseño del robot desmalezador basado en los patrones clásicos de Gamma? Para llevar adelante los objetivos propuestos, se realizó una búsqueda exhausitva sobre el uso de patrones de diseño en el dominio de los sistemas embebidos. Si bien los resultados fueron escasos, entre ellos se destacaron dos libros: [Dou11] y [Whi11b]. [Dou11] propone, cito textualmente, "Design Patterns for Embedded Systems in C" (Patrones de diseño para sistemas embebidos en C). Estos se encuentran documentados utilizando la misma estructura que se utiliza en [Gam+95]. Por lo tanto, a primera vista pareció cumplir con los requisitos de la búsqueda realizada.

Esta tesina tenía como propósito inicial partir de aquellos propuesto como patrones de diseño descritos en los libros mencionados para analizarlos, adecuarlos al formato presentado por Gamma [Gam+95] y considerar su aplicación al diseño del robot desmalezador previamente realizado en [Pom+24].

En el primer paso, al analizar el contenido de los libros, se observó que los supuestos patrones descritos no siguen las prácticas ni los principios establecidos por la IS. Entre los errores detectados se identificó que algunas soluciones propuestas como patrones son, en realidad, aplicaciones de principios básicos de la IS, como la ocultación de información, mientras que otras contradecían principios fundamentales al emplear, por ejemplo, interfaces gruesas o un fuerte acoplamiento entre módulos. El criterio de definicion de patrones parece discrepar al utilizado en la IS.

Por lo tanto, su aplicación en el diseño del robot desmalezador fue desestimada y se reformularon los objetivos de esta tesina.

Objetivos

Al no poder utilizar los libros mencionados como fuente de patrones de diseño, la tesina tuvo que ser reorientada. En particular, se decidió emplear los libros como fuente de problemas de diseño comunes a los que se enfrentan quienes trabajan en sistemas embebidos, robóticos y de control, con el fin de realizar un análisis constructivo en el que se identifiquen las deficiencias de las soluciones presentadas, se propongan alternativas superadoras y se expongan sus ventajas.

El nuevo objetivo es documentar estos problemas y sus soluciones de diseño basadas en la IS. Además, se busca contrastar las soluciones definidas en [Dou11] con las propuestas en la tesina, identificando ventajas, desventajas, puntos de interés e inconvenientes comunes de implementación, acompañados de ejemplos prácticos.

El trabajo, pretende ser una herramienta para desarrolladores de sistemas embebidos de control. Su objetivo es facilitar la realización de buenos diseños aplicando patrones, incluso si los desarrolladores no poseen todos los conocimientos propios de un ingeniero de software. Entre los problemas comunes seleccionados se encuentran: el acceso a diferentes componentes de hardware, es decir, establecer una interfaz que permita la comunicación con sensores o actuadores; problemas relacionados con la obtención de información a partir de diversas fuentes; el manejo de estados mediante máquinas de estados; y el control de múltiples dispositivos de manera coordinada para realizar tareas complejas que requieran cooperación. Asimismo, se incluyen otros aspectos relacionados con la estructura del código, como la organización de la ejecución explotando la capacidad de majear interrupciones del MCU, la verificación de precondiciones en métodos, y la integridad de la información almacenada en memoria.

Las soluciones propuestas consisten en la aplicación de la metodología de Parnas [Par72] y el uso de patrones de diseño. En algunos casos, también se presentan ejemplos de implementaciones que siguen el diseño propuesto, demostrando las ventajas de dichas soluciones.

Estructura

A fin de brindar un conjunto de herramientas útiles y autocontenidas, la tesina se estructuró de la siguiente manera. En primer lugar, el Capítulo 2 (Estado del arte) proporciona contexto al trabajo, describiendo cómo suelen abordarse los problemas de diseño en el ámbito de los sistemas embebidos robóticos y enumerando distintos trabajos científicos relacionados con la temática a fin de justificar la importancia de la tesina.

A continuación, el Capítulo 3 (Sistemas embebidos) aborda los conceptos necesarios para comprender el tipo de sistemas en el cual se busca aplicar la IS en esta tesina. En el Capítulo 4 (Ingeniería de Software) se muestran nociones generales de IS, como la definición diseño para el cambio, la metodología de Parnas, los patrones de diseño y la documentación, entre otros. Además, se presenta un estilo arquitectónico de software clave para el tipo de software tratado en este trabajo. En este capítulo se definen las nociones que hacen a un diseño, un buen diseño en términos de la IS. Las mismas serán aplicadas a lo largo del capítulo siguiente.

El Capítulo 6 (Problemas comunes) presenta el aporte central de esta tesina, donde se desarrollan problemas de diseño tomados de [Dou11] y se proponen soluciones basadas en

la IS. En particular cada sección de este capítulo describe un problema o un conjunto de inconvenientes de la misma naturaleza. Primero, se ofrece una explicación general del problema. Luego, se presenta la solución propuesta en la literatura o la aplicada tradicionalmente, junto con los inconvenientes que esta podría generar ante posibles cambios futuros. Por ultimo, se propone una nueva solución desde la perspectiva de la IS y se enumeran sus ventajas y diferencias con respecto a las alternativas previas.

En el Capítulo **??** (**??**) prenseta las conclusiones y trabajos futuros de esta tesina. Además, Apéndice A resume brevemente algunos patrones de [Gam+95] que son utilizados a lo largo de este trabajo.

Capítulo 2

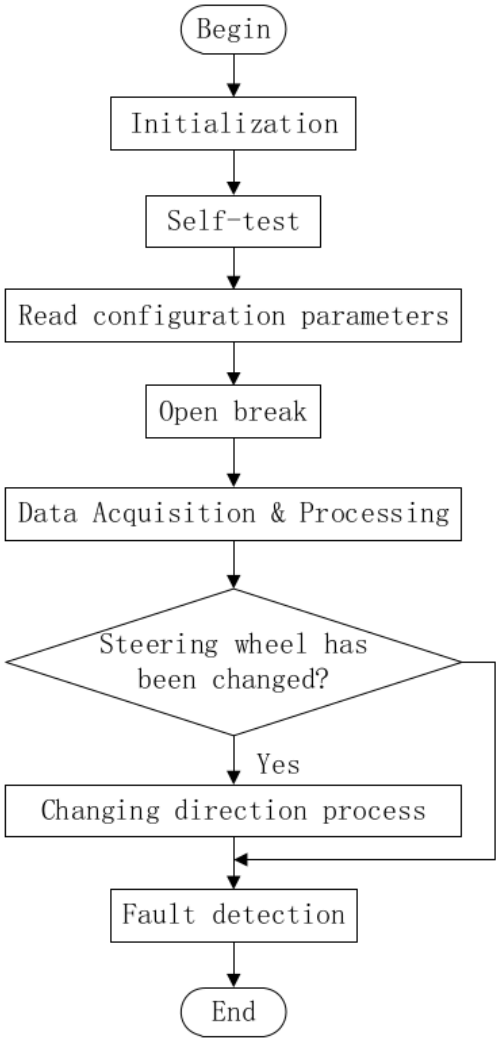
Estado del arte

Para lograr atributos de calidad en el software, tales como modificabilidad, reusabilidad o mantenibilidad, es fundamental realizar un diseño del mismo basado en estilos arquitectónicos y patrones de diseño [Gam+95; SG96b; Bus+94; GJM03a; TMD10; BCK03]. Es decir, aplicar las nociones centrales de la Ingeniería de Software (IS).

Los sistemas de software para robots por lo general poseen alguna de las siguientes características: son distribuidos, embebidos, en tiempo real o manejan muchos datos [Noe05; Brä03]. Su complejidad no termina ahí, deben encargarse del acceso al hardware, los algoritmos de navegación y decisión, entre otras responsabilidades, por lo tanto, muchas veces el diseño es considerado menos prioritario por quienes desarrollan este tipo de sistemas. Esto provoca que el esfuerzo se concentre en solucionar inconvenientes de implementación y no en diseñar cumpliendo los principios de la IS [BP09].

En los trabajos que incluyen información relacionada con el diseño del software [CY12; Zha+16; API22; ERD20; ZZ09; Mas12; Mil19], se observa que esta suele ser escasa y, en general, representada mediante diagramas de flujo. Por ejemplo, en la Figura 2.1 puede verse cómo se presenta el diseño del sistema principal de control de dirección asistida de un vehículo. Este se muestra mediante un diagrama de flujo con el propósito de ilustrar el comportamiento del software, lo cual evidencia que la implementación se ha realizado siguiendo un criterio de división funcional del código. Por otro lado, en diversos trabajos se identifican prácticas de programación poco adecuadas, tales como el uso excesivo de estructuras condicionales anidadas (*ifs*) y la escasa utilización de funciones, lo que revela una deficiente modularidad. Un ejemplo de ello se presenta en el Código 2.1, donde se observan hasta tres niveles de anidamiento. Este tipo de enfoque y el uso de prácticas no recomendables resultan en un código menos modificable, reutilizable y mantenible [Par72].

Figura 2.1: Diagrama de flujo que describe el comportamiento del programa principal de control en [CY12]



.

Código 2.1: Extracto de código de [ERD20].

```
if _py_timestamp is not None:
        # Initialize from PyTimestamp, if available.
        self._py_timestamp = _py_timestamp
    elif timestamp is not None:
        # If Timestamp is available, copy its contents.
        self._py_timestamp = timestamp._py_timestamp
    else:
        if is_top and not is_bottom and coordinates is None:
            self._py_timestamp = PyTimestamp(coordinates, is_top,
                is_bottom)
        elif is_bottom and not is_top and coordinates is None:
```

```
11              self._py_timestamp = PyTimestamp(coordinates, is_top,
                    is_bottom)
12          elif coordinates is not None and not is_bottom and not
               is_top:
13              self._py_timestamp = PyTimestamp(coordinates, is_top,
                    is_bottom)
14          else:
15              raise ValueError(
16                  "Timestamp should either have coordinates"
17                  "or be either Top or Bottom"
18               )
```

En contraposición a estos trabajos, existen otros [Dur+15; BD09] que tienen en cuenta algunos principios fundamentales de la IS a la hora de diseñar y destinan esfuerzo en crear software de cierta calidad. Sin embargo, la aplicación de patrones de diseño no se evidencia.

Por otro lado, existen múltiples trabajos que abordan supuestos patrones de diseño aplicados a ámbitos particulares, inspirándose en la manera de documentar que utilizan los autores en [Gam+95]. Sin embargo, como veremos, no cumplen con los criterios establecidos para ser considerados patrones de diseño.

Entre ellos encontramos los siguientes:

- [PCB18], se definen patrones de diseño para enjambres robóticos en conjunto a un formato de documentación particular. Los patrones se centran en diferentes comportamientos comunes que deben realizar los robots en este tipo de sistemas, como intercambiar información o llevar a cabo cierta interacciones.

- [Car13], presenta tres patrones orientados al control autónomo de robots. Parecen estar más cerca de arquitecturas ya que definen el funcionamiento general de ciertos sistemas sin definir módulos y sus interfaces.

- [BS06] donde se describen patrones que ayudan a desarrollar familias de sistemas robóticos estables, definiendo familia de sistemas como "estable" a una familia de sistemas modelada, diseñada e implementada de manera que las aplicaciones específicas de dicha familia puedan desarrollarse reutilizando, adaptando y especializando conocimientos, arquitecturas y componentes existentes.

- [Arm10] se encuentran patrones de diseño tanto para hardware como software aplicados a sistemas donde la seguridad es crítica debido a su naturaleza. Las aplicaciones de este tipo pueden provocar consecuencias considerables en caso de fallos. La mayoría de los supuestos patrones de diseño descriptos se centran en la redundancia y control de resultados. Además de documentarlos, el autor agrega un análisis de impacto en el coste computacional, con el fin de justificar que su aplicación no genera un impacto considerable.

Podría pensarse entonces que existen múltiples trabajos que abordan patrones de diseño para sistemas embebidos de control o robóticos, sin embargo, no es exactamente así. En los trabajos mencionados, se utiliza el concepto de patrón de diseño, pero no al nivel de

diseño que se trabaja en la IS. Los patrones presentados son *soluciones probadas para ciertos problemas recurrentes*, pero el cambio **no** aparece involucrado entre esos problemas. Es decir, los autores detallan cómo aplicar una solución a diversos inconvenientes comunes en sus áreas de estudio, pero no incluyen al diseño para el cambio a la hora de su confección. Por ejemplo, un patrón de diseño presentado en [Gar+14] es *Patrón de Diseño para el control de un robot diferencial*. Este provee una solución que permite implementar de manera simple un sistema de control especifico. Pero no se pone al diseño para el cambio como una variable a tener en cuenta, la solución no lo considera. Por lo que el resultado de su aplicación no es software preparado para el cambio.

En algunos de los trabajos previamente enumerados y, como mas adelante en la tesina veremos, en el libro de [Dou11], lo que son llamados patrones de diseño en realidad son patrones idiomáticos. Estos también conocidos como *idioms*, son soluciones recurrentes a problemas específicos de implementación que aprovechan las características particulares de un lenguaje de programación. Estos patrones encapsulan prácticas efectivas y convenciones que han demostrado ser útiles para resolver ciertos desafíos técnicos de forma eficiente y elegante dentro del contexto de un lenguaje concreto. A diferencia de otros enfoques más abstractos, los *idioms* operan a nivel del código, abordando aspectos como el manejo de memoria, el control de errores, la gestión de recursos o el uso avanzado de estructuras del lenguaje. Su valor radica en facilitar la escritura de código más claro, robusto y mantenible, al mismo tiempo que aprovechan al máximo las capacidades del compilador o del entorno de ejecución. Debido a su naturaleza dependiente del lenguaje, un idiom que es válido y útil en un lenguaje puede no tener sentido o incluso ser contraproducente en otro [Bus+96].

Además, en estos trabajos no se estudian módulos ni interfaces, sino componentes del sistema, algo más parecido a una arquitectura de software. De hecho, se llegan a documentar patrones de diseño para hardware[1], como ocurre con muchos de los patrones definidos en [Arm10]. En el trabajo [BS06] se considera el hecho de reutilizar software, pero los patrones descriptos corresponden a un nivel de abstracción superior al que se busca en esta tesina. Tratan la interacción de diferentes componentes, similar a una arquitectura de software.

De todas formas, es notable como la comunidad de Robótica ha comenzado a discutir sobre la necesidad de aplicar técnicas y principios de IS para construir software robótico mantenible, reusable y modificable [BP09; KSB16]. Una prueba del interés son desarrollos como [FG03; BFS13; KKL05; Bye05; Dom14], en los cuales se integran de diferentes maneras, prácticas de la IS en el desarrollo de sistemas embebidos. Por un lado desarrollando frameworks y arquitecturas orientadas a este tipo de software, como también definiendo metodologías de trabajo. Estos frameworks no son los únicos orientados a este tipo de sistemas, existen por ejemplo [Bru+07; Qui+09], los cuales son una combinación de sistema operativo y framework. Representan fuertes herramientas para el desarrollo, principalmente, solucionan algunos inconvenientes recurrentes y le quitan responsabilidades al desarrollador resolviendo cuestiones relacionadas con el acceso al hardware, concurrencia, etc.. Es decir, proveen una capa de abstracción a partir de la cual lo desarrolladores pueden empezar a trabajar. Son ampliamente utilizados en la industria aplicados a grandes sistemas. No así para aplicaciones

[1] Un patrón de diseño para hardware es una solución reutilizable y probada a un problema recurrente en el diseño de sistemas físicos o digitales. Suele describir estructuras o componentes de hardware que pueden ser replicados o adaptados, facilitando la estandarización, la reutilización y la eficiencia en el desarrollo.

de menor tamaño, ya que proveen una estructura mayor a la necesaria, complejizando innecesariamente el sistema.

Otra prueba de interés son trabajos como [Shi+15], en los cuales se considera agregar formación relacionada a la IS en cursos de robótica y sistemas embebidos. Con igual énfasis en IS y Robótica. El curso busca enseñar a los estudiantes cómo se aplica la IS a la Robótica. En particular, se dan dos lecturas principales que contienen los conceptos de patrones de diseño y arquitectura de software. En la primer lectura se trabaja sobre los patrones *Observer*, *State*, *Strategy* y *Visitor*. El patrón *Observer* se utiliza extensivamente en ROS[Qui+09] para la comunicación. El patrón *State* es útil para implementar el algoritmo de evitación de obstáculos, que consta de varios estados. El patrón *Strategy* permite intercambiar fácilmente diferentes estrategias de planificación de rutas. El patrón *Visitor* es útil para implementar diferentes métodos de predicción y actualización en la localización. En la segunda lectura, se discuten diferencias entre múltiples arquitecturas utilizadas en el campo, tales como CARMEN [Mic03], MOOS [Pau06], Microsoft Robotics Studio [Jar07] y ROS[Qui+09].

Una de las principales motivaciones es la introducción de la robótica al mercado de consumo masivo fomentando la demanda de reducir los costos del software preservando sus cualidades [BP09]. A medida que los sistemas embebidos incluyen más funciones para nuevos servicios, el software crece gradualmente en tamaño, y los costos y el tiempo de desarrollo también [Bye05]. En particular, el diseño orientado al cambio, es una de las herramientas claves de la IS para conseguirlo, ya que promueve la reusabilidad del software, haciéndolo aplicable a distintos proyectos y entornos. Esta es una muestra de que la implementación de patrones de diseño es una solución potencialmente útil y deseada.

En línea con este enfoque, en esta tesina se presentaran algunos problemas recurrentes en el dominio de la robótica y se propondrán algunas soluciones de diseño basadas en patrones de diseño con el objetivo de construir software de calidad.

Capítulo 3

Sistemas embebidos

En este capítulo se abordan conceptos relacionados al tipo de sistemas con los que se trabaja en el ámbito de la robótica. Entre ellos se encuentran algunas definiciones como la de *Sistemas embebidos*, microcontroladores, interrupciones y ciertos comportamientos comunes de este tipo de sistemas.

3.1 Definición

Distintos autores proponen diferentes definiciones de sistemas embebidos:

"*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. [Whi11a]

"*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 Asistentes Digitales Personales (PDAs).

De las definiciones se puede concluir que un sistema embebido es una pieza clave que permite que 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 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 3.1 extraída de [Noe05] encontramos ejemplos de dispositivos en los que se utilizan sistemas embebidos.

Cuadro 3.1: 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[1]) como la integración de la computación con procesos físicos. Esta integración usualmente se lleva a cabo utilizando sistemas embebidos con ciclos de retroalimentación; en los cuales la parte computacional afecta al ámbito físico y viceversa. Los componentes que permiten la comunicación entre

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

ambos mundos son sensores y actuadores. 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.

Con respecto al hardware en donde corren los sistemas embebidos podemos descatar que suele consistir en una placa o chip compacto que incluye:

- **Unidad de Microcontrolador (MCU):** Un microcontrolador que integra un procesador, memoria y periféricos en un solo chip. Es el componente principal que ejecuta el software.
- **Memoria Flash:** Utilizada para almacenar el programa y los datos no volátiles.
- **Memoria RAM:** Proporciona almacenamiento temporal para datos en tiempo de ejecución.
- **Interfaces de entrada/salida:** Puertos GPIO, ADC, PWM y otros para interactuar con sensores, actuadores y otros dispositivos externos.
- **Fuente de Energía:** Puede provenir de baterías, adaptadores de corriente o incluso energía recolectada del entorno.

El tamaño compacto y la integración de componentes distinguen a los sistemas embebidos de otros sistemas más grandes como las computadoras de escritorio o los servidores. Se diferencian, además, en su poder de cómputo limitado tanto por el hardware (baja disponibilidad de memoria RAM, clock de la CPU bajo, etc) como por la disponibilidad de energía eléctrica.

Existen varios microcontroladores multipropósito de uso comercial, tales como los producidos por la empresa Arduino [Ard] o los similares de la familia Raspberry Pi [Ras]. La presentación de los mismos suele ser en forma de una placa preparada para conectar los inputs y outputs, como las que se observan en Figuras 3.1 y 3.2.

Por defecto, el microcontrolador ejecuta el software almacenado en su unidad flash, la cual debe ser grabada cada vez que se actualice el código. Dado esto y las limitaciones de hardware ya mencionadas, se deben tener ciertas consideraciones a la hora de escribir el código. En particular, se debe prestar atención a la performance del sistema, a la cantidad de librerías a utilizar, al uso de memoria RAM, etc.

3.2 Interrupciones

Como se mencionó, el dispositivo en el que el software corre suele ser un microcontrolador (MCU). Estos dispositivos poseen ciertas características particulares que los hacen ideales para trabajar en entornos cercanos al mundo real; entre ellas las interrupciones. Las mismas son un mecanismo de 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.

Figura 3.1: Arduino UNO

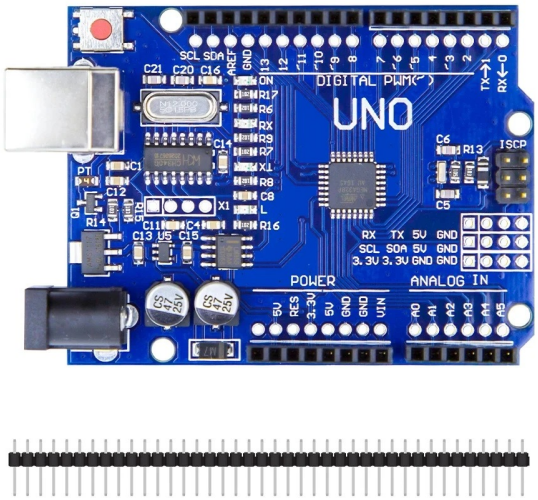


Figura 3.2: Raspberry Pico 2

Cuando ocurre una interrupción (por ejemplo, un cambio en un sensor o una solicitud de un actuador), el MCU detiene su ejecución actual y salta a una rutina de servicio 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. Una vez finalizada la ejecución de la ISR, el MCU retorna automáticamente al punto en el que se había interrumpido, reanudando el programa principal sin perder el flujo de ejecución. Siguiendo la Figura 3.3, el MCU se encuentra ejecutando el código principal cuando se produce una interrupción. En ese momento, detiene temporalmente dicha ejecución para atender la rutina de interrupción. Al completarse esta, el MCU continúa la ejecución del código principal desde la primera instrucción que no había sido ejecutada, en este caso, la *instrucción 3*.

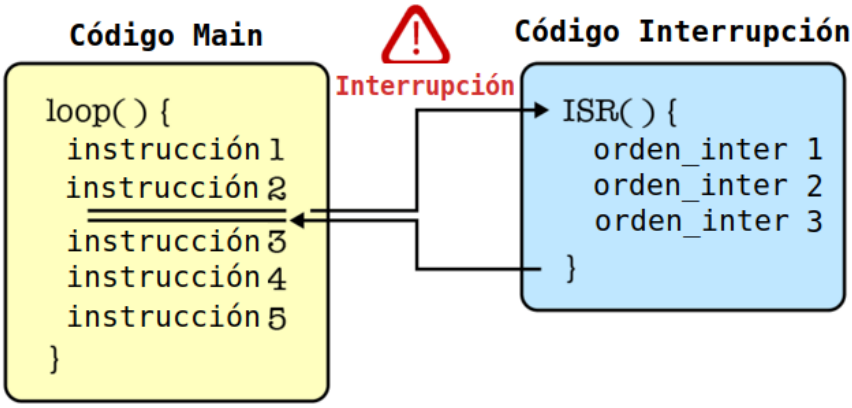


Figura 3.3: Diagrama interrupción extraído de [Neo14].

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.

3.3 Sistemas Embebidos de Control Robótico

Un sistema embebido de control robótico es una unidad de procesamiento diseñada para gestionar el funcionamiento de un robot, integrando sensores, actuadores y algoritmos de control. Por lo general estos sistemas están diseñados para operar en tiempo real y ejecutar tareas específicas llevando a cabo comportamientos básicos del robot relacionados al movimiento o adquisición de información. Por ejemplo, en [Pom+24] el sistema se encarga realizar las ordenes provenientes de una PC, las cuales pueden establecer el desplazamiento en cuanto a velocidad y giro. Para concretar las acciones se necesita llevar un control en tiempo real de los componenetes físicos y para hacerlo se hace uso de todo el hardware disponible (sensores, actuadores, sistemas de comunicación, etc.).

Existen dos enfoques básicos para llevar a cabo el control, lazo cerrado y lazo abierto.

- Control en lazo abierto: en este enfoque, el sistema embebido envía comandos al actuador sin recibir retroalimentación del entorno. Es una estrategia más simple y rápida, pero menos precisa, ya que no puede corregir desviaciones en la ejecución de la tarea. Un ejemplo es un motor que gira a una velocidad fija sin verificar si realmente alcanza la velocidad deseada.

- Control en lazo cerrado: aquí, el sistema embebido recibe información en tiempo real de sensores y ajusta su comportamiento en función de la retroalimentación. Esto permite corregir errores y mejorar la precisión del control. Un ejemplo clásico es el control de velocidad de un motor, donde sensores miden la velocidad real y ajustan la potencia suministrada para mantener el valor deseado.

Para implementar un control por lazo cerrado se pueden aplicar la noción de ciclos de control. Se ejecutan ciertas operaciones de manera repetida a fin de lograr que una cierta característica llegue al valor deseado. Las operaciones que se llevan a cabo en el ciclo son las siguientes:

- Establecer valores de referencia (posición deseada, velocidad deseada, etc.).
- Medición: obtener datos de sensores (posición, velocidad, temperatura, etc.).
- Cálculo: se compara los valores medidos con los de referencia a fin de determinar si se alcanzaron y en caso de no haberlo logrado definir las acciones necesarias. Para hacerlo se utilizan algoritmos de control como, por ejemplo, algoritmos PID.
- Actuación: envío de comandos a motores, servomecanismos u otros actuadores para ejecutar las acciones calculadas.

De esta manera, luego de una serie de iteraciones, si los valores de referencia son aclcanzables, se tiende a obtener los resultados deseados.

Los sistemas embebidos constituyen el núcleo escencial de numerosos dispositivos que interactúan con el mundo físico. A lo largo del capítulo se abordaron sus fundamentos, desde la definición y el hardware involucrado hasta conceptos clave como las interrupciones y los ciclos de control. Este conocimiento resulta fundamental para enfrentar los desafíos del diseño y la programación en entornos donde la respuesta en tiempo real y la interacción con el entorno son requisitos centrales.

Capítulo 4

Ingeniería de Software

En este capítulo se abordan conceptos centrales a la IS y que resultan esenciales para el entendimiento del trabajo realizado en la tesina. Entre ellos, se tratan las definiciones principales, metodologías de trabajo y técnicas utilizadas en el ámbito.

4.1 Definiciones

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.[SG96a; GJM03a; BCK03; TMD10] Tradicionalmente, el software para robots tiende a desarrollarse de manera monolítica, con pocas funciones de gran tamaño y numerosas sentencias condicionales anidadas, o mediante una descomposición funcional básica que resulta ineficiente para mantener y reutilizar componentes [API22; ERD20]. Frente a esto, se propone un diseño sistemático basado en principios de la IS, 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; SG96a; GJM03a; BCK03; TMD10]. Ya sean probocadas cambios en los objetivos de comportamiento, en el hardware o en los algoritmos de control, entre otros. Para esto, cuando se diseña se tiene en ceunta aquello que probablemente cambiará y se piensa cómo debe ser el diseño para que los cambios impliquen modificar porciones minimas y aisaldas del software [GJM03a].

Diseñar anticipando los cambios permite reducir los esfuerzos necesarios para implementar ajustes y facilita la reutilización de componentes entre distintos sistemas [Par78; CN02].

Para abordar el reto que implica anticiparse al cambio, el diseño modular es clave [Par72]. Un **módulo** es un elemento de diseño que, en un lenguaje de programación orientado a objetos, suele implementarse como una **clase**. A su vez, una **instancia** de un módulo, en este tipo de lenguajes, corresponde a un **objeto**. Sin embargo, cuando se habla de diseño, el término adecuado es módulo e instancia, ya que la implementación podría realizarse en un lenguaje que no sea orientado a objetos. Este enfoque de diseño organiza el software como un conjunto de módulos simples con responsabilidades claramente definidas y relaciones tambien bien definidas entre ellos. Cada módulo se diseña para ocultar aquello que puede cambiar, lo que minimiza el impacto de las modificaciones en el resto del sistema. Por ejemplo, si un

sensor de velocidad es reemplazada y por tanto cambia su forma de reportar datos, el ajuste del software debería limitarse al módulo que encapsula ese sensor físico.

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. 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 [SG96a].

Herencia de Interfaz

En el diseño modular, un módulo consta de una parte visible llamada *interfaz* y una oculta la *implementación*. La interfaz define el conjunto de métodos accesibles externamente, mientras que la implementación gestiona cómo el módulo lleva a cabo los comportamientos de cada método definido en la interfaz. Se dice que la implementación es invisible para el resto de módulos de sistema, esta misma oculta el posible cambio que encapsula en módulo.

Existen esencialmente dos tipos de herencia:

- **Herencia de clases**: Un módulo hijo hereda tanto la interfaz como la implementación de un módulo padre, es decir, hereda todo su comportamiento. Si bien esto permite la reutilización de código, introduce un acoplamiento[1] fuerte y, por lo tanto, reduce la flexibilidad ante cambios. Lo cual es contraproducente al objetivo de diseñar anticipando el cambio.

- **Herencia de interfaces**: En este caso, un módulo hijo solo hereda la interfaz, permitiendo que cada módulo implemente su propio comportamiento. Este enfoque es más flexible y facilita la adaptabilidad del sistema. Cuando hablemos de herencia en el resto de la tesina nos referiremos a este enfoque, herencia de interfaces.

A parir de esta relación entre módulos podemos definir dos tipos de módulos, los abstractos y los concretos. Un módulo abstracto define un conjunto de operaciones que pueden ser utilizadas por otros módulos. Este tipo establece un contrato de uso, permitiendo a otros módulos interactuar con él sin conocer sus detalles internos. Su función principal es favorecer la flexibilidad y extensibilidad del sistema, ya que distintos módulos pueden cumplir con ese mismo contrato ofreciendo implementaciones diferentes (en la Sección 6.1 se estudia caso de aplicación). Un módulo concreto, en cambio, es aquel que proporciona una implementación específica del comportamiento definido por un módulo abstracto. En los diseños que aplican patrones, se busca que los módulos consumidores dependan solo de las abstracciones y no directamente de las implementaciones concretas. Esta separación permite modificar, reemplazar o extender funcionalidades sin afectar el resto del sistema, y facilita la aplicación de técnicas como la composición (que veremos a continuación).

[1] El grado de dependencia o vinculación entre dos módulos. Un acoplamiento bajo implica que los módulos están independientes, mientras que un acoplamiento alto implica que conocer o modificar uno requiere conocer o modificar el otro.

Composición y delegación

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 módulos dentro de un módulo, lo que permite que las instancias deleguen tareas a estos módulos asociados. En lugar de que un módulo herede directamente un comportamiento específico, como un tipo particular de algoritmo de ordenamiento, la composición propone que el módulo mantenga una referencia a otro que se encargue de esa tarea. Por ejemplo, en vez de que una módulo de procesamiento de datos herede una forma fija de ordenar elementos, con lo cual debe contener un método `ordenar(lista)`, puede delegar esa responsabilidad a un componente intercambiable que implemente distintas estrategias. De marnera que cuando el módulo principar necesite ordenar una lista simplemente llame a un método del módulo compuesto, delegando esa tarea.

La composición es 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.

Dado que la herencia de clases tiende a ser rígida, los patrones de diseño la evitan, combinando en su lugar herencia de interfaces con composición de módulos y delegación de responsabilidades. Mediante la composición, un módulo puede reutilizar funcionalidad sin necesidad de heredar implementación, mientras que la delegación permite distribuir tareas entre distintos módulos de manera flexible.

4.2 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; Cri22]

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 Sección 4.3). Una vez identificados, Parnas sugiere ocultar y aislar cada item de cambio en un módulo independiente, asegurando que cada módulo oculte las decisiones de diseño específicas que podrían cambiar. Esto se logra diseñando interfaces que expongan tan poco como sea imposible sobre sus detalles de implementación, permitiendo que los módulos interactúen sin conocer su trabajo 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[2] 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 [Par78; Cri22]:

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

4.3 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; Cri22].

- Contracción o extension de requisitos (agregar, quitar o editar funcionalidades).
- Configuraciones de hardware (cambio de MCU, plataforma, etc.).
- Formato de los datos de entrada y salida (protocolos de comunicación, etc).
- Estructuras de datos (listas, tablas hash, queues, stacks, etc.).
- Algoritmos (de control, de decición, ordenamiento, etc.).

[2]Encapsular funcionalidades refiere al principio de ocultar los detalles internos de un módulo y exponer solo lo necesario a los demás módulos del sistema. La idea clave es que cada módulo tenga una interfaz bien definida y que sus detalles internos (implementación, estructuras de datos, algoritmos) sean ocultos.

- Algunos usuarios pueden requerir solo un subconjunto de los servicios o características que otros usuarios necesitan (limitar las funcionalidades, por ejemplo).
- Dispositivos periféricos (actuadores, sensores, etc.).
- 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 (Interfaz de Programación de Aplicaciones (API), por ejemplo).

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

4.4 Patrones de Diseño

A la hora de estructurar código podemos distinguir dos niveles, uno se tratará en esa sub-sección y el otro en Sección 4.5. Los patrones de diseño se aplican en el nivel de diseño. En este los elementos son módulos y la comunicación se lleva a cabo mediante llamadas a procedimiento. Además de las llamadas, se aplican dos conceptos previamente mencionados en la Sección 4, la composición y la herencia de interfaces. Estas son las herramientas con las se cuenta en el nivel de diseño para estructurar el código.

En [Gam+95], los autores traen 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 fue 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 diseño (módulos), sus relacipnes 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, qué beneficios y costos obtenemos de la aplicación. Así como las formas que el patrón provee para anticiparse a los posibles cambios futuros.

Determinar qué es o no un patrón resulta objetivo y el criterio de selección varía entre autores. Para este trabajo se utilizará el mismo criterio que los autores eligieron en [Gam+95]:

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

Notar que no solo se quiere saber cómo resolver un problema, sino que se busca que la solución se alinee con los principios de la IS y provea un buen diseño que permita lograr las propiedades que se buscan; modificabilidad, reusabilidad, mantenibilidad, etc.

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

4.5 Arquitectura de Software

En contraste con los patrones de diseño, la arquitectura de software no utiliza módulos e interfaces como actores principales, sino que los elementos son componentes, los cuales pueden estar formados por uno o múltiples módulos y tener cierta complejidad. La forma de comunicarse en este nivel se realiza mediante conectores los cuales pueden no ser necesariamente llamadas a procesos, sino otras estructuras como protocolos, pipes, etc. [BCK03; Bus+96].

Según [SG96a] 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 de diseño aplicar para facilitar la extensión y el mantenimiento, y cómo gestionar las interacciones entre diferentes componentes del sistema.

4.6 Documentación

Quienes diseñan un sistema no serán necesariamente quienes lo implementen e incluso es deseable que no lo sean, por lo que sus decisiones deben estar disponibles para que diversos interesados en el sistema puedan revisarlas y comprenderlas. Por ello, documentar o describir el diseño de manera adecuada es tan importante como el diseño mismo. En este sentido, Parnas, Clements y Weiss introducen en el concepto de "*design through documentation*" (diseño a través de la documentación), lo que nos lleva a enunciar el siguiente principio de diseño: *Un diseño sin documentación carece de utilidad práctica.* Ese enfoque de diseño de software que consiste en avanzar en el desarrollo estructurando y escribiendo documentación técnica precisa desde las primeras etapas del proyecto. En lugar de centrarse inicialmente en el código o en representaciones informales, se propone utilizar documentos (que más adelante en esta Sección veremos) como la guía de módulos, las especificaciones de interfaces y los requisitos

formales como herramientas fundamentales para organizar, analizar y comunicar las decisiones de diseño. Permiiendo identificar inconsistencias, delegar responsabilidades claramente entre módulos y facilitar el mantenimiento y la incorporación de nuevos desarrolladores al proyecto. Según los autores, diseñar mediante documentación no solo clarifica el proceso, sino que mejora la calidad y reutilización del software resultante.

Para lograr una buena descripción del diseño se utilizan múltiples documentos que aportan información sobre distintos aspectos a tener en cuenta [Cle+10], entre ellos se encuentran:

- **Documentos de módulos**. Los elementos de estos documentos son módulos o unidades de implementación. Los módulos representan una forma basada en el código de considerar al sistema. Cada módulo tiene asignada y es responsable de llevar adelante una función. Estos son: Especificación de Interfaces, Estructura de Módulos, Guía de Módulos, Estructura de Herencia y Estructura de Uso.
- **Documentos de aspectos dinámicos**. En estos documentos los elementos son componentes presentes en tiempo de ejecución y los conectores que permiten que los componentes interactúen entre sí. Los documentos que conforman este tipo son: Estructura de Procesos, Estructura de Objetos, Diagrama.
- **Documentos con referencias externas**. Estos documentos muestran la relación entre las partes en que fue descompuesto el sistema y elementos externos (tales como archivos, procesadores, personas, etc.). Enre ellos: Estructura Física o de Despliegue.

Esta documentación debe estar bien estructurada con un formato bien definido. Los autores establecen cómo se define cada tipo de documento. En particular para documentar módulos de manera estructurada se utiliza un lenguaje llamado **2MIL** basado en TDN presentado en [GJM03b]. Por ejemplo, un módulo llamado ModuloA que exporta dos métodos se documenta de la forma mostrada en la Figura 4.1.

Figura 4.1: Documentación de un módulo utilizando 2MIL.

Module	ModuloA
exports	metodo1 metodo2
comments	En esta sección se muestra comentarios relevantes al usuario.

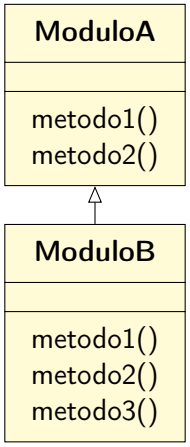
Y un heredero de ModuloA llamado ModuloB se documenta como en la Figura 4.2.

Figura 4.2: Documentación de un módulo heredero utilizando 2MIL.

Module	ModuloB **inherits from** ModuloA
exports	metodo3
comments	Este módulo hereda el comportamiento de ModuloA y lo extiende con funcionalidades adicionales (metodo3).

Esto puede ser a su vez documentado de manera gráfica siguiendo el diagrama mostrado en la Figura 4.3.

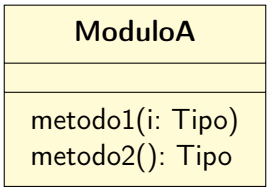
Figura 4.3: Documentación gráfica de un módulo y su heredero.



Además, es posible indicar el tipo de los parámetros de entrada de un método así como el tipo del valor de retorno del mismo. Para eso utilizamos la gramática presentada en la Figura 4.4. Recordar que asumimos que un módulo es, genera o define un tipo. Es decir que para tipar nuestros métodos usamos módulos.

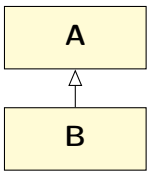
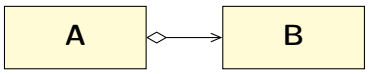
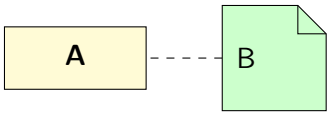
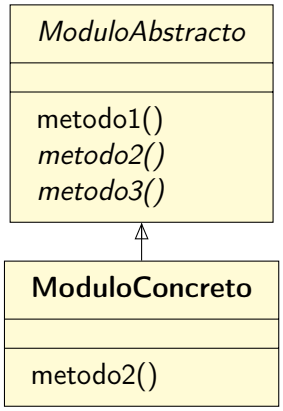
Este formato gráfico de documentación puede ser extendido haciendo uso de los símbolos definidos en la Figura 4.5.

Figura 4.4: Documentación de un módulo indicando el tipo de sus parámetros.



Por otro lado, la aplicación de patrones de diseño también debe ser documentada. Para hacerlo se utiliza de igual forma el lenguade **2MIL** de la manera indicada en la Figura 4.6.

Figura 4.5: Significado de los símbolos

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

En este formato se establece el patrón que se está aplicando y se da una breve descripción del caso de aplicación. Además, se enumeran los cambios previstos y comó el patrón funciona en el caso particular. Por último, se define qué módulo de nuestro sistema cumple el rol de cada participante del patrón.

Figura 4.6: Documentación aplicación de patrón.

PatternApp	**Breve descripción**
based on	Patrón (Pattern)
why	**Cambios previstos**: Descripción de los cambios previstos relacionados al patrón. **Funcionalidad**: Explicación de la aplicación del patrón en el caso particular.
where	ModuloParticipante1 **is** Participante1 ModuloParticipante2 **is** Participante2

A fines didácticos en esta tesina utilizaremos solo la representacion grafica para dpcumentar módulos y sus relaciones. Para documentar patrones si se mostrará utilizando **2MIL**. Información adicional sobre cómo documentar utilizando este lenguaje puede encontrarse en [Cri22; Cri15; Pom24].

Capítulo 5

Soluciones útiles

5.1 Un estilo arquitectonico para sistemas embebidos

Si nos centramos en los sistemas embebidos de control como los que se definieron en la Sección 3.3, encontramos que existen trabajos sobre estilos arquitectonicos de software orientados al control de procesos; por ejemplo el estilo arquitectónico de *Control del Procesos* presentado en [SG96a]. El mismo está definido para ser aplicado en sistemas de control de procesos físicos donde se quiere mantener ciertas propiedades de la salida del proceso, cerca de valores de referencia. Algunos ejemplos son el control de la velocidad de giro de una rueda, la posición del extrusor de una impresora 3D, la temperatura del agua en una caldera, etc.

Antes de explicar en que se basa el estilo veamos las definiciones presentes en el Cuadro 5.1. Los tres conceptos hacen referencia al mundo físico, no son definiciones de software. Podemos decir entonces, que nuestro sistema de control se relacionará con estas variables físicas para lograr el objetivo.

Cuadro 5.1: Conceptos clave de un proceso físico.

Término	**Definición**
Variable controlada	Propiedad o valor físico que puede ser medido mediante un sensor, velocidad de giro de una rueda, temperatura del agua, etc..
Variable manipulada	Variable del proceso que se puede ajustar directamente para influir en las variables controladas, como la tensión suministrada al un motor.
Set Point	El valor deseado o de referencia para una variable controlada. Se busca ajustar el sistema para alcanzar y mantener este valor.

El control se basa en un proceso a ciclo cerrado (*clossed-loop*) con retroalimentación hacia atrás (feedback). En este se realizan las siguientes operaciones:

- El sistema recibe valores de referencia llamados *set-points*.

- Se leen los valores de las variables controladas a través de sensores.
- Con los valores recolectados, se realizan los cálculos a fin de modificar mediante actuadores las variables manipuladas intentando que las variables controladas lleguen a los valores de referencia.
- Una vez decidida la acción, se aplica el ajuste sobre las variables manipuladas y la iteración se repite.

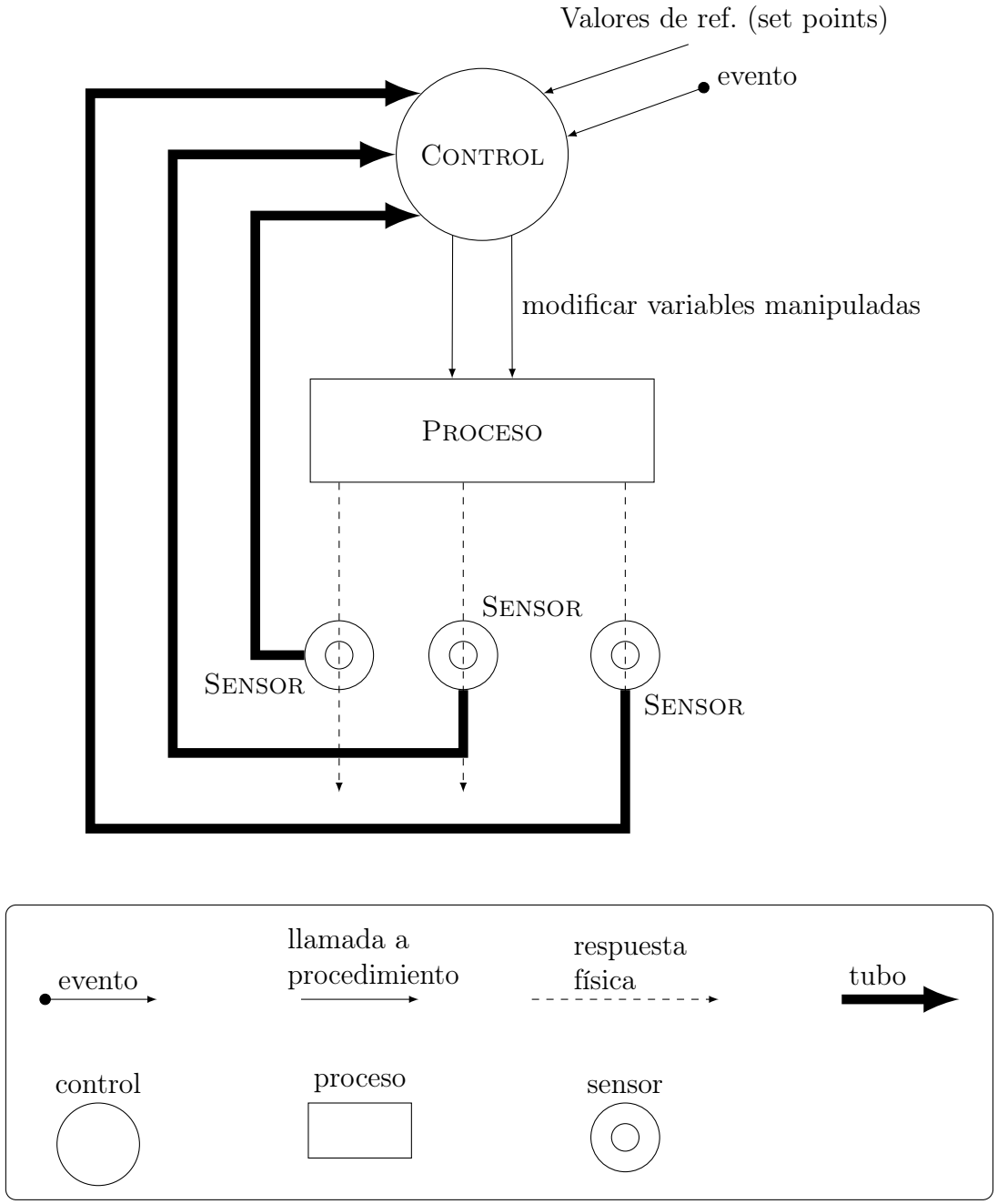
De esta manera se logra alcanzar, si es posible, a los valores deseados luego de una serie de iteraciones.

Supóngase que se está trabajando con un robot móvil y, en particular, con una de sus ruedas motorizadas. El objetivo del control es lograr que la velocidad de la rueda alcance los valores de referencia configurados por el usuario. La variable controlada es la velocidad de giro de la rueda, medida mediante con un velocímetro, y la variable manipulada es la tensión aplicada al motor. Modificando dicha tensión se logra alterar la velocidad de giro del motor y, por ende, la de la rueda.

Siguiendo los pasos descriptos, el sistema leerá primero el valor de velocidad configurado por el usuario y luego el valor medido por el velocímetro. Con estos dos datos, decidirá qué tensión aplicar. Por ejemplo, si el usuario solicita 30 revoluciones por minuto (RPM) y el velocímetro mide 20 RPM, el sistema incrementará la tensión. En una iteración futura, si se alcanza la velocidad deseada, se ajustará nuevamente la tensión para no sobrepasar el valor configurado.

Para llevar este comportamiento a cabo, el estilo arquitectonico propone los componentes de software que se muestran en la Figura 5.1.

Figura 5.1: Diagrama de la arquitectura control de procesos



Como se puede observar el estilo se basa en tres componenes básicos **Control**, **Proceso** y **Sensores**. Cada uno de ellos trabaja de manera independiente y se comunican mediante dos mecanismos principales, llamadas a prodedimientos y tubos (*pipes*). Veamos qué información oculta cada uno:

- **Proceso**: encapsula los mecanismos y dispositivos que permiten modificar las variables manipuladas del proceso físico. Por ejemplo, el hardware y los drivers que permiten

ajustar la tensión que se entrega al motor de una rueda. Nótese que este es un componente de software; no debe confundirse con el proceso físico que ocurre en el mundo real.

- **Control**: encapsula el algoritmo que decide cómo modificar las variables manipuladas para que las variables controladas alcancen los valores de referencia deseados. Siguiendo el ejemplo planteado anteriormente, oculta el algoritmo que, a partir de la velocidad deseada y la medida por el velocímetro, determina cuál es la tensión que debe aplicarse al motor para alcanzar el valor requerido.

- **Sensores**: encapsulan los dispositivos y mecanismos necesarios para obtener los valores de las variables controladas. Por ejemplo, el funcionamiento del velocímetro.

Además, el estilo cuenta con otro tipo de componente importante: los **tubos**. Estos se utilizan para transmitir los valores de las variables controladas desde los sensores al algoritmo de control de manera desacoplada. El objetivo es que el **Control** pueda operar sin conocer directamente los sensores, logrando así una mayor independencia. Comúnmente, cada **tubo** es un módulo que provee un método para escribir y otro para leer. En este esquema, los **Sensores** escriben los valores obtenidos del entorno físico, y el **Control** los recupera utilizando el método de lectura del **tubo**.

La ventaja de este estilo arquiectónico es que brinda un enfoque modular con independencia entre componentes. Por ejemplo, un sensor emite información colocándola en un tubo sin conocer al destinatario. Esto permite que los sensores puedan ser modificado o reemplazado sin afectar al controlador, así como agregar nuevos sensores. De manera similar, existe una separación clara entre el algoritmo de control, que realiza los cálculos para alcanzar los valores de referencia, y el proceso, que se encarga de aplicar dichos cálculos. Este desacople permite que cambios en el algoritmo de control no impacten directamente en el proceso y viceversa. En conclusión se facilita la incorporación de nuevos sensores, actualizaciones en el algoritmo de control y cambios en el hardware, promoviendo flexibilidad y escalabilidad.

5.2 Un patrón muy conveniente para sistemas embebidos

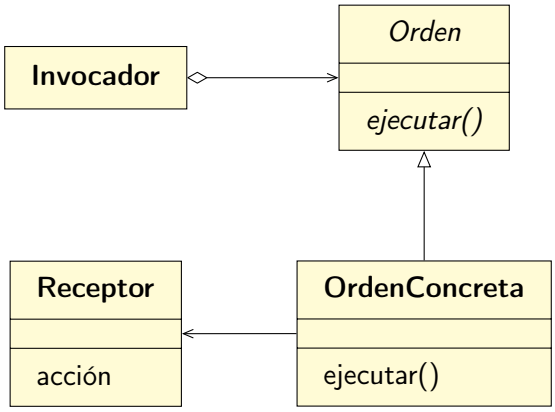
Parnas en [Par72] establece que desacoplar se refiere a la idea de reducir la dependencia entre módulos en un sistema de software. Un sistema está acoplado cuando los cambios en un módulo requieren modificaciones en otros módulos. Desacoplar significa diseñar los módulos de manera que puedan funcionar y cambiar independientemente.

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], llamdo *Command*. Este patrón es el que más veces se aplicó en el trabajo del robot desmalezador [**laura**], por lo que es importante conocerlo.

La función principal que cumple este patrón es la de desacoplar el módulo que invoca una orden, la orden en si y aquel que sabe como llevarla a cabo. En particular, este patrón puede ser utilizado para reemplazar las *callbacks* entre módulos. Es común utilizar *callbacks* para mantener los niveles de abstracción, en donde módulos de bajo nivel de abstracción desconocen la existencia de módulos más abstractos.

En la Figura 5.2 podemos observar la estructura de módulos del mismo.

Figura 5.2: Estructura patrón *Command*



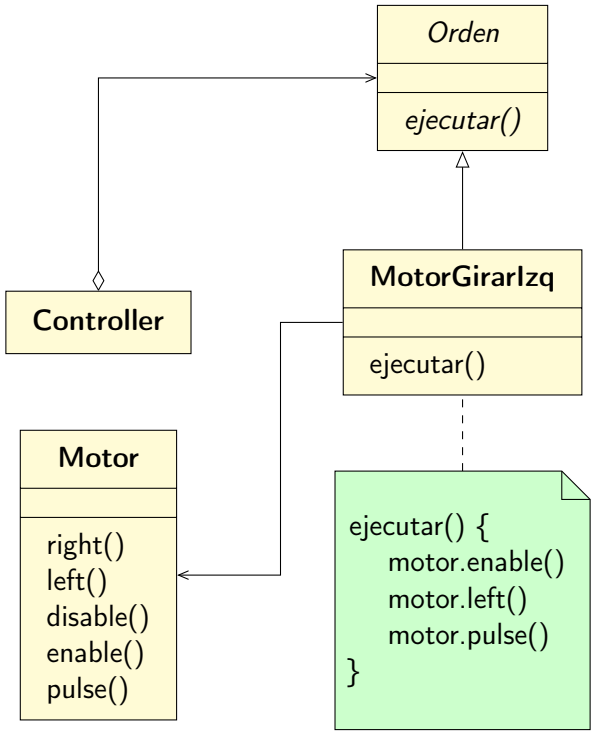
- Invocador: le pide a la *Orden* que ejecute la acción.
- *Orden*: declara la interfaz para ejecutar una acción.
- OrdenConcreta: implementa el método ejecución el cual se encarga de llamar a el o los métodos del Receptor con el objetivo de llevar a cabo la acción.
- Receptor: cualquier módulo sobre la cual se realiza la acción.

Como objetivo fundamental, buscamos no tener que modificar la implementación del módulo invocador en caso de un cambio en el módulo que se encarga de realizar las tareas. Ademas, se le quita la responsabilidad de saber exactamente qué acciones realizar para llevar a cabo un procedimiento particular. Por ejemplo, un módulo necesita que otro se inicie, pero este último para hacerlo requiere que se invoque una serie de sus métodos de manera ordenada. Si se lo hace de la forma clásica, el primer módulo debe ajustar su implementación al segundo. En cambio, con una orden se mueve la responsabilidad a un nuevo módulo.

Al encapsular cada solicitud de una operación dentro de un módulo, el patrón permite que los módulos que invocan 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 del resto de módulos. 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 la orden es un módulo puede ser extendido para implementar múltiples funcionalidades, como deshacer operaciones o registrar cambios.

5.2.0.1 Ejemplo

Figura 5.3: Ejemplo de aplicación básica del patrón *Command*



PatternApp	**Orden para controlar motor paso a paso**
based on	Orden (Command)
why	**Cambios previstos**: Se pueden agregar o modificar ordenes. **Funcionalidad**: Desacoplar al que invoca una orden de que la lleva a cabo, proporcionando flexibilidad ante cambios en las ordenes o el receptor.
where	Orden **is** Orden MotorGirarIzq **is** OrdenConcreta Motor **is** Receptor Controller **is** Invocador

Un módulo Controller, necesita manejar un motor paso a paso reprensentado en el diseño por el módulo Motor. Para hacerlo se deben ejecutar una serie de métodos de la interfaz del Motor. En particular, primero se debe habilitar el giro del motor llamando al método `enable`, luego configurar el sentido de giro con el método `left` o `right` (izquierda o derecha) y

por ultimo avanzar un paso de giro invocando `pulse`. Tradicionalmente esto seria realizado desde el módulo Controller agregando el código en cierto método de este (ver ejemplo en Código 5.1), provocando así, un fuerte acoplamiento entre Controller y Motor. Esto es, ante cualquier cambio de la interfaz de Motor, se debe actualizar la implementación del módulo Controller, es decir, hay que modificar dos módulos y como dijimos el sistema debe estar cerrado a cambios y abierto a extensiones. Reduciendo al mínimo la modificación de código existente lo que podría traer como resultado la introducción de errores.

Código 5.1: Ejemplo de implementación sin usar el patrón *Command*.

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

Para aplicar el patrón *Command* se debe crear el módulo heredero de *Orden* que representa el comando en cuestión, en este caso, MotorGirarIzq. Este encapsula cómo deben invocarse los métodos del módulo Motor para que este realice un paso de giro hacia la izquierda. Por lo tanto, Controller invocará el método ejecutar de MotorGirarIzq para realizar dicha acción.

Los problemas del diseño tradicional se solucionan al aplicar este patrón. Se logra desacoplar al Motor del Controller: los posibles cambios en el módulo Motor afectan solo al módulo MotorGirarIzq. El Controller desconoce cómo se lleva a cabo la acción de girar el motor un paso hacia la izquierda.

En caso de que la interfaz de Motor cambie por cualquier motivo, se puede crear un nuevo módulo heredado de *Orden*, el cual implemente cómo ejecutar la acción de girar a la izquierda sobre el nuevo motor.

Otro uso interesante del patrón surge cuando se define una cierta estructura conceptual en el sistema. Esta puede responder a la naturaleza de la aplicación. Por ejemplo, en un sistema de control, se puede definir que los módulos que toman decisiones sobre el control sean quienes invoquen a los sensores para obtener información, generando así una jerarquía en la que los sensores no deben invocar métodos de módulos superiores. En caso de ser necesaria la comunicación en sentido inverso, se puede utilizar el patrón *Command* para reemplazar el uso de *callbacks*.

Este uso del patrón *Command* será aplicado en varias de las soluciones a problemas comunes en sistemas embebidos que se proponen más adelante en esta tesina.

Capítulo 6

Problemas comunes

En este capítulo se enumeran problemas comunes en el desarrollo de software embebido de control, extraídos principalmente del libro "*Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*" [Dou11]. Cabe recordar que, si bien el título del libro indica que propone patrones de diseño, las soluciones que presenta no lo son realmente y, en algunos casos, resultan inadecuadas. Además, se analiza la solución de diseño propuesta en dicho libro y, a su vez, se aporta una alternativa desde la perspectiva de la Ingeniería del Software (IS) enfocada en el diseño para el cambio. En muchos casos, se identifica la posibilidad de aplicar ciertos patrones de diseño descritos en [Gam+95], mientras que en otros se utilizan conceptos y metodologías clave del diseño para el cambio, propuestos por David L. Parnas [Par78; Par72; Par77].

6.1 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 un 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 se trabajará sobre un ejemplo simple. Suponga que se tiene un sistema embebido que debe controlar un motor de corriente continua (DC) y que el software para hacerlo corre en un MCU de la empresa **Arduino**. Los requerimientos definen que es necesario poder asignar el sentido de giro (horario o antihorario) y la velocidad de rotación del motor según se necesite. Para controlar el motor se utiliza una placa DRV8838, la cual se coloca entre el MCU y el motor. Es necesario ya que el microcontrolador y su plataforma no pueden manejar las potencias requeridas para hacer funcionar el dispositivo. Podemos ver el conexionado de la mismo en la Figura 6.1.

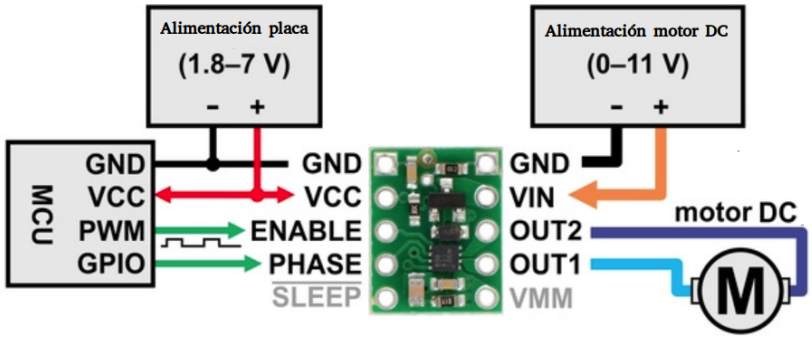


Figura 6.1: Conexionado de la placa de control DRV8838.

Como se observa la placa DRV8838 tiene múltiples pines. De estos podemos identificar dos grupos: los que aparecen a la izquierda, que llamaremos pines de entrada, y los de la derecha, que denominaremos pines de salida. El MCU se conecta a los pines de entrada y, a través de estos, controlará el funcionamiento de la placa. En cambio, en los pines de salida se conecta el motor y transmiten la potencia necesaria.

Dentro de cada uno de estos grupos se encuentran pines destinados a la comunicación entre componentes, y otros que se utilizan para alimentar tanto la lógica de la placa DRV8838 como al motor. Estos últimos son *GND*, *VCC* y *VIN*. Los pines de control de entrada son *ENABLE*, *PHASE* y *SLEEP*, aunque en este ejemplo se utilizarán solo los dos primeros para simplificar la explicación. De todas formas, en la Tabla 6.1 se puede consultar la función de cada pin y si la señal correspondiente es analógica o digital. Por último, *OUT1*, *OUT2* y *VMM* son los pines de salida hacia el motor, y son los que finalmente transmiten la tensión y potencia necesarias para lograr el comportamiento deseado del motor.

MCU se conecta a los pines de *ENABLE*, *PHASE* y *SLEEP* el software deberá gestionarlos. Pero, como se mencionó, para acotar el ejemplo se restringieron los requerimientos y para complirlos solo necesitaremos trabajar con los pines *ENABLE* y *PHASE*. Tendremos entonces dos cables conectados desde microcontrolador a la placa DRV8838, uno que se dirige a *ENABLE* y otro a *PHASE*. En el pseudo-código 6.1 se encuentran las lineas necesarias para

PHASE	dirección de rotación	digital
ENABLE	velocidad de rotación	analógico
SLEEP	liberar fuerza	digital

Cuadro 6.1: Funciones de cada pin del módulo DRV8838

poder configurar el motor para luego poder utilizarlo en el resto del sistema. En estas se establece que el pin número siete del MCU está conectado a el pin *PHASE* y que el nueve a *ENABLE*. Y a su vez se inicializa el pin dentro del software como pin de salida (*OUTPUT*). Esto es necesario para que otras funciones que se llamen en el futuro se comporten como esperamos.

Código 6.1: Configuración inicial del control del motor DC.

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

# Constantes globales
DIR_pin = 7
VEL_pin = 9

def setup()
      .
      .
      .
  pinMode(DIR_pin, OUTPUT)
  pinMode(VEL_pin, OUTPUT)
      .
      .
      .
```

Una vez configurados los pines podemos hacer uso de las funciones `pinMode`, `digitalWrite` y `analogWrite` 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` (entrada o salida). Y tanto `digitalWrite` como `analogWrite`, configuran en un PIN el valor especificado. `digitalWrite` admite dos valores definidos por el entorno `HIGH` y `DOWN`. Por lo tanto, si se quiere establecer la máxima velocidad de giro en el motor se haría algo como en el Código 6.2. Y en caso de querer detenerlo usamos `analogWrite` de la forma que se muestra en el Código 6.3.

Código 6.2: Establecer máxima velocidad giro en sentido horario.

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

Código 6.3: Detener giro del motor DC.

```
analogWrite(VEL_pin, 0)
```

No es necesario entender por completo qué hace cada llamada, pero sí es importante comprender que ejecutar el Código 6.2 y 6.3 es fundamental para controlar el motor. Es decir, cualquier cliente del motor en el sistema debe saber que, para hacer que el motor gire a la máxima velocidad, es necesario ejecutar las dos lineas mostradas en el Código 6.2 sobre los pines correspondientes al motor. Por ejemplo, si se desea realizar una acción con el motor en función del valor de alguna variable del sistema (`valor`), **tradicionalmente** se implementa algo similar al código mostrado en el Código 6.4. En el cual si se cumple la condición de que `valor` es mayor a 100 se ordenará al motor que avance a máxima velocidad, en caso contrario se indicará velocidad nula.

Código 6.4: Ejemplo uso del motor DC.

```
def controlar_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?

- Consideremos un caso en el que un segundo motor es agregado. Este es controlado por otra placa DRV8838 que también se conecta con dos pines al MCU. Para poder utilizarlo en el código debemos definir al, al igual que con el primer motor, los dos pines que utilizará, supongamos en este caso `DIR_pin2` y `VEL_pin2`. Si queremos modificar la función `controlar_motor` para incluir a este nuevo motor podríamos hacer algo como en el Código 6.5.

Código 6.5: Extención de la función controlar_motor para controlar dos motores.

```
def controlar_motor(motor)

  if (valor > 100)
    if (motor = IDMotor1)
        digitalWrite(DIR_pin, HIGH)
        analogWrite(VEL_pin, 255)
      else if (motor = IDMotor2)
        digitalWrite(DIR_pin2, HIGH)
        analogWrite(VEL_pin2, 255)
        .
        .
        .
  else
    if (motor = IDMotor1)
        analogWrite(VEL_pin, 0)
      else if (motor = IDMotor2)
```

```
17          analogWrite(VEL_pin2, 0)
18          .
19          .
20          .
```

La modificación consiste en verificar que motor es el que se quiere manipular y en base a eso enviar la señal a traves de los pines correspondientes. Pero, ¿qué pasaría si se quiere añadir un tercer motor? Deberíamos agregar aún mas sentencias `if else`, extendiendo aun más el código. ¿Y si debemos realizar operaciones diferentes según cada motor? Por ejemplo, en el caso que `valor` sea mayor a 100 con el motor 1 se debe establecer velocidad máxima pero con el motor 2 nula. El cambio en el código es pequeño (ver Código 6.6), solo cambia un valor en la linea 9. Pero introduce complejidad en la lectura y comprensión facilitando la introducción de errores en futuras modificaciones. Es insostenible en el tiempo este enfoque de diseño.

Código 6.6: Modificación de la función controlar_motor para cambiar comportamiento al utilizar el motor 2.

```
1  def controlar_motor(motor)
2
3    if (valor > 100)
4      if (motor = IDMotor1)
5          digitalWrite(DIR_pin, HIGH)
6          analogWrite(VEL_pin, 255)
7      else if (motor = IDMotor2)
8          digitalWrite(DIR_pin2, HIGH)
9          analogWrite(VEL_pin2, 0)
10         .
11         .
12         .
13   else
14     if (motor = IDMotor1)
15         analogWrite(VEL_pin, 0)
16     else if (motor = IDMotor2)
17         analogWrite(VEL_pin2, 0)
18         .
19         .
20         .
```

- 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 giro horario ahora es anti horario. Para llevar a cabo el cambio, debemos modificar **todas** las llamadas a `digitalWrite(DIR_pin, HIGH)`, tanto en el código que de la función `controlar_motor` como en el resto del sistema, cambiando `HIGH` por `DOWN` y viceversa. Por ejemplo, en el Código 6.6 debemos modificar las lineas 5 y 8. Es fácil cometer un error y dejar al sistema en un estado inconsistente. Ni hablar en el caso que se planteó en el punto anterior, puede ser necesario discriminar

entre motores. La precaución a la hora de modificar debe ser mayor aún y a su vez la probabilidad de introducir errores aumenta.

- Por cierto motivo se descompuso la placa controladora del motor 1, y no se consigue un reemplazo idéntico, sino que se adquiere una nueva placa de otra marca, por ejemplo, una *Pololu Simple Motor Controller G2*. En este caso, este placa no utiliza la misma interfaz de control que el , sino para controlarla se accede a ella mediante comunicación serial (utiliza un solo pin especifico). Incluso utilizando las herramientas provistas por el entorno de **Arduino Uno**, el nuevo código de configuración (ver Codigo 6.7) y uso (ver Codigos 6.8 y 6.9) difiere significativamente del anterior (Codigo 6.1 para configuración y Codigos 6.2 y 6.3 para uso).

Código 6.7: Configuración de la placa de control del motor DC utiliza comunicación serie.

```
def set_up()
    .
    .
    .
    Serial.begin(9000)
    .
    .
    .
```

Código 6.8: Establecer máxima velocidad giro horario para el caso de comunicación en serie.

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

Código 6.9: Establecer detención para el caso de comunicación en serie.

```
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-verificar código que ya se sabía que funcionaba correctamente. Se debe modificar las lineas 5, 6 y 15 de la función `controlar_motor` de la manera mostrada en el Código 6.10.

Código 6.10: Modificación de la función controlar_motor para utilizar placa de control serial para controlar el motor 1.

```
def controlar_motor(motor)

  if (valor > 100)
    if (motor = IDMotor1)
        Serial.write(0xAA)
```

```
        Serial.write(0x0C)
        Serial.write(0x85)
        Serial.write(0x7F)
    else if (motor = IDMotor2)
        digitalWrite(DIR_pin2, HIGH)
        analogWrite(VEL_pin2, 0)
        .
        .
        .
  else
    if (motor = IDMotor1)
        Serial.write(0xAA)
        Serial.write(0x0C)
        Serial.write(0xE0)
    else if (motor = IDMotor2)
        analogWrite(VEL_pin2, 0)
        .
        .
        .
```

Pero... ¿y si en el futuro se consigue la placa DRV8838? Muchas veces se suele añadir una bandera que indique el tipo de hardware. En este caso, se podría definir una constante, por ejemplo, `TIPO_MOTOR1`, que identifique el tipo de placa controladora. Al incorporar esta bandera en nuestra función `controlar_motor`, se obtiene el Código 6.11.

Código 6.11: Modificación de la función controlar_motor para utilizar bandera indicadora de tipo de placa controladora.

```
def controlar_motor(motor)

  if (valor > 100)
    if (motor = IDMotor1)
      if (TIPO_MOTOR1 = DVR8838):
          digitalWrite(DIR_pin, HIGH)
          analogWrite(VEL_pin, 255)
      else if (TIPO_MOTOR1 = Pololu):
          Serial.write(0xAA)
          Serial.write(0x0C)
          Serial.write(0x85)
          Serial.write(0x7F)
    else if (motor = IDMotor2)
        digitalWrite(DIR_pin2, HIGH)
        analogWrite(VEL_pin2, 0)
        .
        .
        .
```

```
19    else
20      if (motor = IDMotor1)
21        if (TIPO_MOTOR1 = DVR8838):
22          analogWrite(VEL_pin1, 255)
23        else if (TIPO_MOTOR1 = Pololu):
24          Serial.write(0xAA)
25          Serial.write(0x0C)
26          Serial.write(0xE0)
27        else if (motor = IDMotor2)
28          analogWrite(VEL_pin2, 0)
29          .
30          .
31          .
```

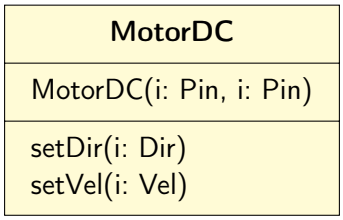
Esto no es una buena solución, ya que lo unico que logra es generar más dificulad a la hora de introducir un nuevo cambio. Ahora, si se debe cambiar la lógica de la función, debemos tener en cuenta más lineas a modificar. Introduciendo así más posibilidades de cometer errores.

- Claramente, el código obtenido es poco claro; es decir, no resulta fácil comprender de qué se trata una determinada porción de código con solo leerla. Basta con intentar entender la función del Código 6.11 para comprobarlo. A su vez, el código resulta difícil de modificar, ya que requiere un esfuerzo adicional de comprensión antes de poder aplicar cualquier cambio. En el ejemplo, la lógica principal de la función es sencilla —solo una sentencia condicional—, pero en general las funciones suelen ser más complejas e incluyen múltiples sentencias, estructuras anidadas y bucles.

En el libro, la solución de este problema es nombrada como un patrón de diseño, el patrón *"Hardware Proxy"*. Pero desde el punto de vista de la IS, no es un patrón de diseño, sino uno de los principios fundamentales de la IS, que es el DBOI.

Estos inconvenientes son derivados de que el *hardware* comprende un ítem de cambio frecuente; por lo que si seguimos la metodología de Parnas (ver sección 4.2), se debe aislar ese posible cambio en un módulo. En el ejemplo que se está trabajado, se debe crear un módulo que encapsula el hardware en cuestión, el motor DC. Este módulo oculta cómo debe ser usado el hardware y provee una interfaz lo suficientemente insensible a la implementación. Es decir, al momento de confeccionar la interfaz del módulo se debe pensar en lo que el motor siempre va a hacer independientemente de los posibles cambios que sufra el hardware subyacente. Para esto, se debe elegir la cantidad de mínima de métodos del modo más abstracto posible, sin agregar métodos que puedan ser reemplazados utilizando otros que ya fueron definidos [Par78; Par77]. Un motor DC siempre recibirá ordenes para definir su sentido y velocidad de rotación. La Figura 6.2 presenta la interfaz del módulo MotorDC que encapsula el hardware relacionado al motor.

Figura 6.2: Interfaz MotorDC



Si se cuenta con dos o mas motores del mismo tipo, se crearán dos a mas instancias del módulo. Para esto el contrsuctor recibirá como parámetros los pines de control, el método de `setDir` toma el valor de dirección a establecer y el input de `setVel` el valor de velocidad deseado. En el Código 6.12 se puede ver un ejemplo de una posible implementación de esta interfaz utilizando el controlador .

Código 6.12: Posible implementación de la interfaz del módulo MotorDC.

```
def MotorDC(dir_pin, vel_pin);
    pinMode(this->dir_pin, OUTPUT);
    pinMode(this->vel_pin, OUTPUT);


def set_dir(dir)
    if (dir == HORARIO)
        digitalWrite(dir_pin, HIGH);
    else
        digitalWrite(dir_pin, DOWN);


def setVel(vel)
    analogWrite(vel_pin, vel);
```

En el código 6.13 se evidencian las primeras ventajas, el uso del motor en el código es mucho más claro en comparación al diseño tradicional, donde teniamos que escribir las lineas de los Códigos 6.1 y 6.2 para lograr lo mismo.

Código 6.13: Ejemplo de uso de la interfaz del módulo MotorDC

```
motor = MotorDC(1, 2)

motor.setDir(Dir.HORARIO)
motor.setVel(255)
```

Además, si se debe invertir el sentido de giro como en el ejemplo propuesto al comienzo de la sección, es tan fácil como cambiar la implementación del método `setDir`, los clientes no notarán el cambio. A su vez, si se quiere controlar otro motor es posible hacerlo fácilmente como en el Código 6.14.

Código 6.14: Ejemplo control nuevo motor DC.

```
motor_delantero = MotorDC(18, 19)

motor_delantero->setDir(ANTIHORARIO)
motor_delantero->setVel(10)
```

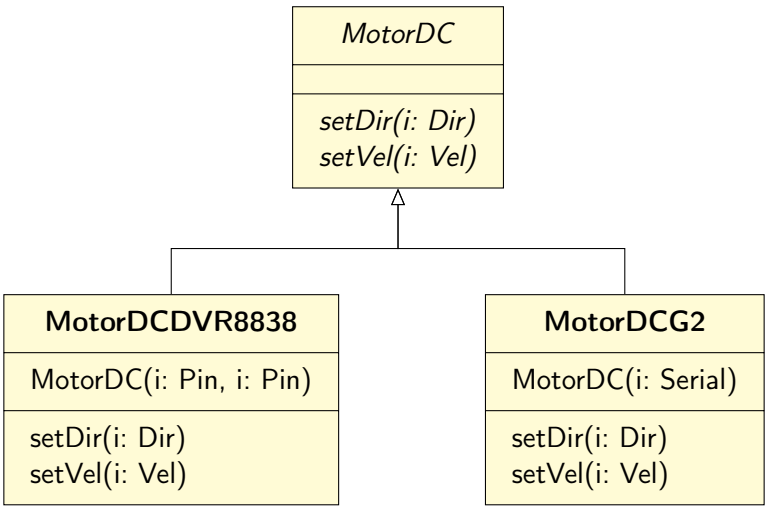
En caso de un cambio de componente de hardware, como el explicado anteriormente, los clientes del módulo no lo notarán, dado que la interfaz se mantendrá intacta. Por ejemplo, para el motor que utiliza comunicación serie, `setDir` será redefinida como en el código 6.15 teniendo que solo volver a verificar el módulo MotorDC.

Código 6.15: Implementación método setVel para el motor que utiliza comunicación serie.

```
def setVel(vel)
        serial.write(0xAA;
        serial.write(0x0C)
        hex_vel = int_to_hex(vel)
        serial.write(hex_vel)
```

Sin embargo, haciendo uso del concepto de herencia de interfaz y la noción de abierto-cerrado (explicados en la sección 4). Esto permite reutilizar módulos ya implementados y abstraer aún más la implementación. Para hacerlo se define un módulo abstraco *MotorDC* del cual hereda la interfaz cada modelo o combinación de motor y placa controladora. ELa Figura 6.3 presenta la estructura de interfaces de esta solución.

Figura 6.3: Módulo MotorDC abstracto y estructura de herencia.



El cliente solo sabe que manipula un elemento *MotorDC*, no tiene noción con cual de los dos tipos de de placa controladora esta tratando. También es posible agregar más herederos, uno por cada modelo de placa controladora/motor, y reutilizar las módulos implementados en caso de utilizar hardware idéntico.

Siguiendo con el ejemplo de la función `controlar_motor` veamos en la Figura 6.16 cómo es una posible implementación de esta misma utilizando la encapsulación del hardware propuesta. Con esta solución el codigo de la funcion no deberá ser modificado por más que cambien las marcas/tipos de placas controladoras.

Código 6.16: Implementación de la función controlar_motor utilizando encapsulación del hardware.

```
def controlar_motor(motor: MotorDC)
  if (valor > 100)
    motor.setDir(HORARIO)
    motor.setDir(VelMaxima)
  else
    motor.setDir(VelNula)
```

De esta manera se siguen las prácticas recomendadas en la IS [SG96a; GJM03a; BCK03; TMD10] y se obtiene un diseño orientado al cambio [Gam+95].

6.2 Interfaces que no se ajustan perfectamente

Muchas veces el proveedor del hardware incluye con este librerías para su control, otras veces se consiguen en internet o en 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.

Suponga el siguiente ejemplo, en un cierto sistema embebido se 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 figura 6.4.

Figura 6.4: 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 brinda 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 el método, 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 hacer uso del nuevo display tradicionalmente se modifican todas las llamadas a las viejas funciones alrededor de todo sistema agregando la lógica nueva, como en el código 6.17. El cambio puedo parecer no muy profundo, pero se debe tener en cuenta que con los nuevos cambios es necesario recopilar **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 más complicados en su lógica y de diferente naturaleza. La configuración inicial del diseño es la que se muestra en 6.5. Es decir, la solución tradicional es no tener en cuenta el cambio en el diseño, lo que provoca las desventajas comentadas.

Código 6.17: 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")
20  }
```

Figura 6.5: Configuración original del controlador del display.

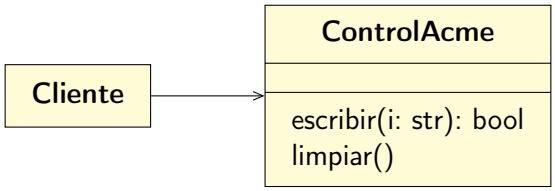
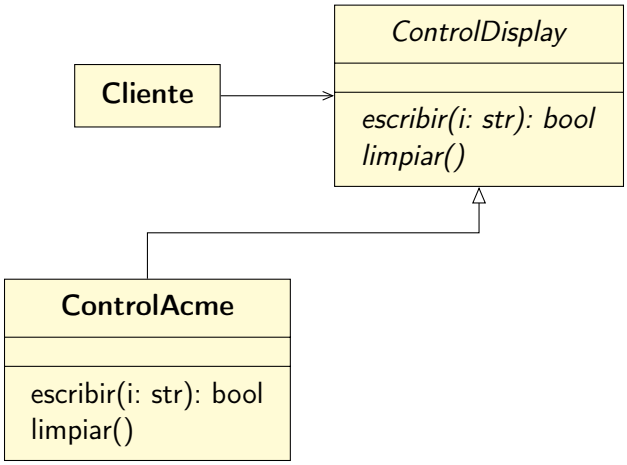


Figura 6.6: Configuración utilizando los conceptos de la sección anterior 6.1.



Para construir un diseño orientado al cambio, primero se aplican los conceptos vistos en la sección Acceso al hardware, logrando el diseño 6.6. Luego, para permitir el cambio propuesto sin tener que realizar todas las modificaciones mencionadas, se propone aplicar el patrón *Adapter*, creando un módulo intermedio con la misma interfaz que **DisplayLib**. El cual utiliza 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 6.7 y su debida documentación de aplicación del patrón en 6.8.

Figura 6.7: Nueva configuración utilizando el patrón *Adapter*.

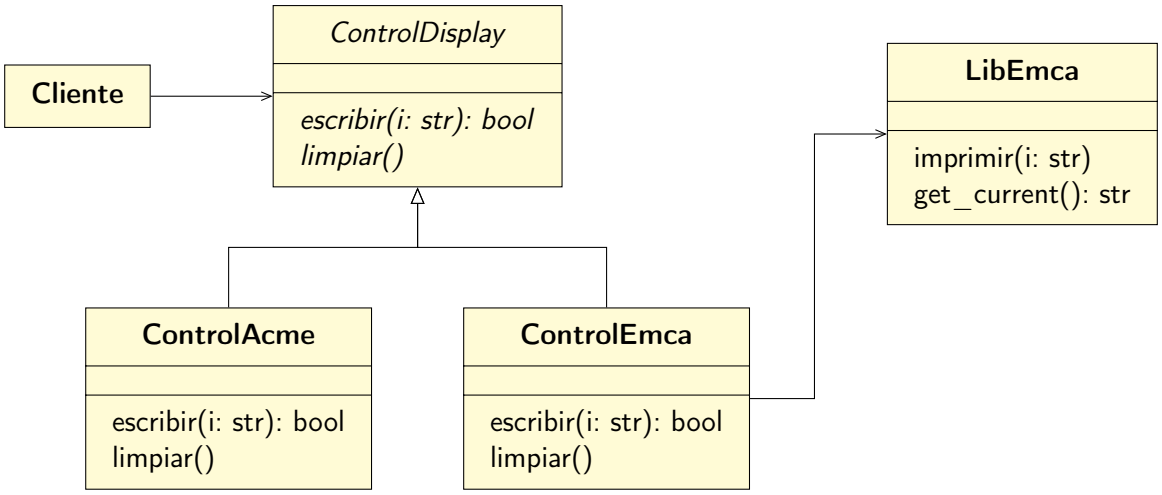


Figura 6.8: Documentación de la aplicación del patrón Adapter al ejemplo del display.

PatternApp	**Adaptar nuevo controlador de display**
based on	Adaptador (Adapter)
why	**Cambios previstos**: Se pueden agregar diferentes displays pero manteniendo una interfaz común. **Funcionalidad**: En caso de agregar un nuevo display que provee una intefaz diferente a la utilizada en el sistema, se crea un módulo que la adapta para que corresponda a la usada.
where	ControlDisplay **is** Target LibEmca **is** Adaptee ControlEmca **is** Adapter

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

solución se obtienen las siguientes ventajas:

- Facilitar la sustitución de hardware: permite cambiar el display sin necesidad de modificar el código del cliente, reduciendo el impacto del cambio de hardware en el sistema.
- Mantiener la coherencia en la interfaz: el cliente sigue interactuando con la misma interfaz abstracta (*ControlDisplay*), evitando la necesidad de modificar múltiples módulos en el sistema.
- Minimizar el riesgo de errores: al encapsular las diferencias de implementación en el adaptador (**ControlEmca**), se reduce la posibilidad de introducir errores al modificar manualmente todas las llamadas en el código.
- Mejorar la mantenibilidad: cualquier nuevo display con una interfaz distinta solo requiere la creación de un nuevo adaptador.
- Se promueve la reutilización de código: la abstracción permite reutilizar la lógica del cliente sin importar qué display se use, evitando la duplicación de código y mejorando la modularidad.

En este ejemplo, una posible implementación para las funciones `escribir(i: str): bool` y `limpiar()` del módulo **ControlEmca** es la del código 6.18.

Código 6.18: Ejemplo implementación módulo ControlEmca.

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

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

6.3 Control en conjunto de dispositivos

Muchas aplicaciones embebidas robóticas controlan **actuadores** que deben trabajar en conjunto para lograr el efecto deseado. Por ejemplo, para conseguir mover de manera coordinada 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 estabilidad necesaria. 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.

Solución tradicional

Antes de pasar a explicar la solución propuesta y analizar cómo aplicarla al ejemplo presentado por Douglass en su libro [Dou11], se describirá cómo se aborda tradicionalmente esta problemática. Para ello, se tomará como ejemplo el software desarrollado para el robot desmalezador antes de la propuesta del nuevo diseño [Pom+24]. En particular, existieron dos desarrollos creados en conjunto por ingenieros electrónicos como parte del trabajo final de carrera [GIM19; BCD18]. Los requerimientos son similares a los que se consideraron para el desarrollo del nuevo diseño.

Estructura y funcionamiento general

El sistema desarrollado controla el siguiente hardware:

El robot cuenta con cuatro ruedas y un dispositivo de dirección que les permite girar. Cada rueda tiene sensores Hall¸, que permiten medir su posición y velocidad, y un sistema asociado de medición de corriente. El dispositivo de dirección permite determinar su posición angular en cada momento. Tanto las ruedas como el dispositivo de dirección pueden ser operados de forma remota mediante un control remoto (RC), capaz de enviar señales de dirección y velocidad a un módulo receptor de radiofrecuencia (RF) situado en el robot. Además, una computadora (PC) situada en el robot envía órdenes al dispositivo de dirección y a las ruedas para la navegación autónoma del robot. Las órdenes provenientes tanto del RC como de la PC son procesadas por un microcontrolador ubicado en el robot.

El código que conforma el sistema de control se encuentra dividido en unos pocos archivos, concentrando todo el flujo de control en `main.c`, el resto contienen métodos que son invocados desde este último y proveen utilidades. No se utiliza programación orientada a objetos, en cambio, como estructura de organización del código se utilizan las funciones clásicas de C. Estas parece que encapsulan operaciones específicas como:

- Configuración de hardware.
- Lectura de entradas (sensores, botones, etc.).
- Control de salidas (motores, luces, etc.).

La información común entre muchas funciones se almacena en variables globales definidas en el mismo archivo. Entre las variables, encontramos algunas que se encargan de almacenar información referida al estado de operacion del sistema. Es decir, que los estados se manejan con sentencias `if` o `switch case` (se comenta sobre esta solución en la sección Máquinas de estado).

La función principal del sistema es `main`, la misma se encarga de inicializar y calibrar los sensores y actuadores, de realizar el ciclo de control y terminar la ejecución. Para las primeras dos tareas llama a dos funciones que realizan el trabajo. El ciclo dSolución orientada

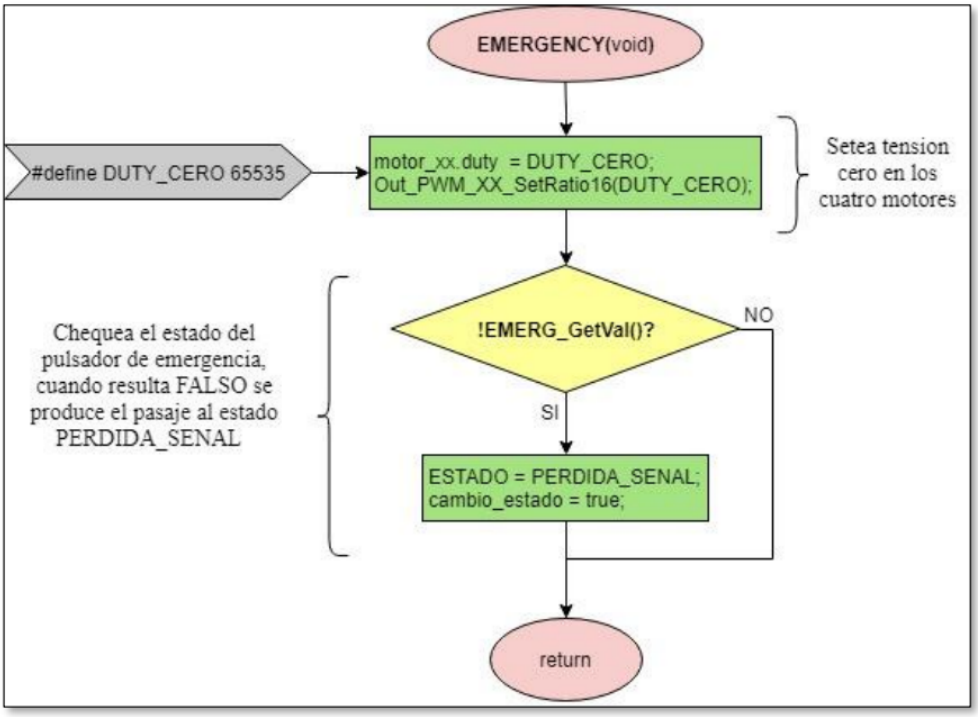
al cambioe control se ve representado por un bucle infinito en el cual se realizan las siguientes tareas principales:

- Lectura de información de los sensores, dirección, velocidad y corriente.
- En base al estado de ejecución actual, se realizan ciertas tareas.
- Se aplican cambios a los actuadores, el ciclo de trabajo (PWM) de los motores y la dirección.

Observaciones

El criterio de división del código parece ser funcional, tanto por el aspecto del código, como por la documentación adjunta en los informes [GIM19, pág. 78-85], [BCD18, pág. 110-149]. Esta última, se centra es describir la funcionalidad de cada método y para hacerlo se muestra su comportamiento utilizando diagramas de flujo como el de la figura 6.9.

Figura 6.9: Diagrama de flujo que explica el comportamiento de la función emergency [GIM19, pág. 82].



Por otro lado, a lo largo del código se utilizan estructuras condicionales (if, switch, etc.) para determinar el flujo de ejecución. A su vez, las funciones que acceden directamente al hardware están directamente integradas en la lógica del control, lo que indica una baja separación entre la capa de abstracción del hardware y la lógica de "alto" nivel. Esto es acompañado con un diseño procedimental, con una serie de pasos secuenciales y un control centralizado en el flujo principal. Ademas, el hardware se encuentra integrado a la lógica, se aplican estructuras condicionales directamente a este.

Otros inconvenientes referidos al cambio que están presentes en el código:

- El código parece estar compuesto por funciones largas y bloques monolíticos sin modularidad clara. Esto dificulta la localización y modificación de funcionalidades específicas, ya que los cambios pueden propagarse a otras partes del sistema.
- Hay valores "hardcodeados" (constantes definidas fijas en el código). Si estos valores cambian, es necesario modificar el código fuente, aumentando el riesgo de introducir errores.
- Las dependencias entre funciones están estrechamente acopladas. Lo que provoca que los cambios puedan requerir modificaciones significativas en diferentes secciones de código.
- El manejo de errores parece ser inconsistente o inexistente en varias secciones. Esto puede llevar a comportamientos impredecibles y dificultar el diagnóstico de problemas.

Conclusión

El diseño del código parece estar orientado a cumplir con un objetivo específico mediante un flujo procedimental y un control directo de los periféricos del hardware. Este enfoque es funcional, pero carece de modularidad y abstracción, lo que lo hace menos flexible y más difícil de mantener. La estructura actual no parece diseñada para escalar con nuevas funcionalidades. Lo que tiene sentido, si se tiene en cuenta el contexto del desarrollo.

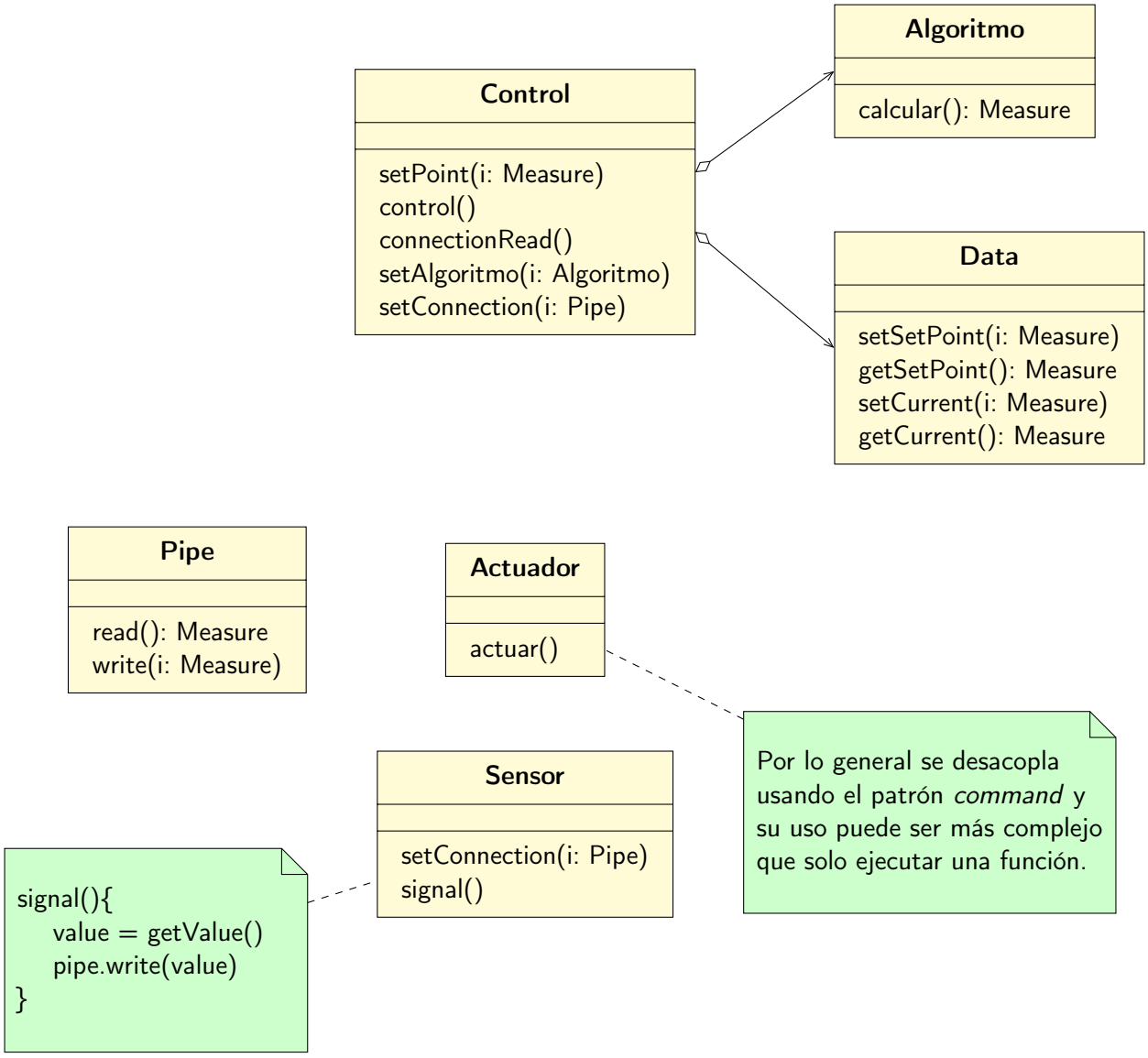
6.3.1 Subsistemas de control

Para dar la solucion orientada al cambio, primero se introducirá el concepto de **Subsistema de control**. Un subsistema de control es una estructura de módulos relacionados mediante herencia, composición o invocación, que responde al control de una propiedad física particular del sistema embebido de control en el que se va a aplicar. Para hacerlo lleva a cabo todas las tareas de la arquitectura de Un estilo arquitectonico para sistemas embebidos en el nivel conceptual de módulos.

A fin de aplicar la arquitectura de software de control de procesos, se propone la creación de un *subsistema de control* por cada variable a manipular[1] que sea necesaria controlar por el sistema. Para permitir la interacción con un cliente externo debe proveer una interfaz que permita setear un valor al que se quiera llevar la propiedad (*setPoint* (ver tabla 5.1)) y que indique el comienzo de la tarea de control. Un subsistema tiene una estructura similar a la de la arquitectura 5.1, pero su alcance estará restringido a la variable a manipular y las variables medidas relacionadas. En la figura 6.10 se puede observar de manera concreta los módulos que forman parte de cada componente de un subsistema.

[1]Ver tabla conceptos arquitectura Control de Procesos 5.1

Figura 6.10: Módulos de un subsistema de control.



Los módulos principales son los siguientes:

- **Control**: encargado de proveer una interfaz a clientes desde la cual se realizan todas las tareas referidas al control.
- **Actuador**: encapsula el dispositivo de hardware que manipula la variable física.
- **Sensor**: encapsula el dispositivo de hardware utilizado para recibir información del mundo físico, pueden existir múltiples sensores.
- **Pipe**: módulo utilizado para desacoplar la comunicación entre **Control** y los sensores.

La manera en la que un *cliente* utiliza el subsistema para lograr que la propiedad que controla se ubique en el valor que se desea es de la indicada en el código 6.19. Primero se configura el *set-point* deseado, luego se una señal a los sensores para que escriban en el pipe los valores leídos, luego se lee la información y se indica el ciclo de control invocando `control`.

Código 6.19: Ejemplo de uso del subsistema.

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

Notar que en 6.10 aparecen algunos módulos que no fueron mencionados anteriormente, por un lado **Algoritmo** el cual se encarga de los cálculos necesario para determinar de que manera aplicar un cambio con el/los actuadores. Determina si se llego o no al *set-point* y a su vez define qué cambio es necesario aplicar para llegar al mismo. En el ámbito de la robótica se aplican diferentes técnicas de estabilización[2] de variables físicas, las cuales permiten alcanzar un cierto *set-point* y mantenerse en el, por ejemplo, los controladores PID[Min22] (controlador proporcional, integral y derivativo). Los cuales representan un mecanismo de control que, a través de un lazo de retroalimentación, permiten manejar una variable física. Este tipo de técnicas puede ser usada siguiendo la arquitectura propuesta y, en particular, los cálculos asociados se definirían en este módulo **Algoritmo**.

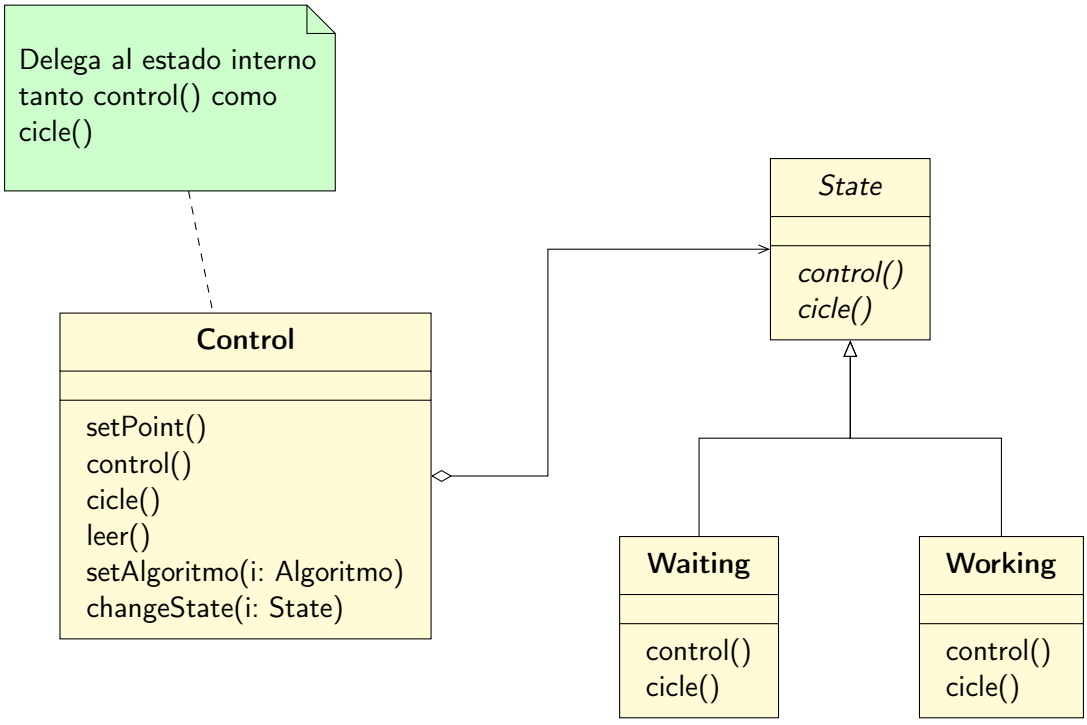
Por otro lado, **Data** desacopla el almacenamiento de información relacionada al subsistema. En el caso más básico solo se almacena el *set-point*, pero puede agregarse todo lo necesario, incluso llevar registro de los valores actuales y pasados de cada variable de interés.

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 se manejan motores paso a paso (luego se expandirá esto en un ejemplo). ¿Cómo se puede adaptar la estructura propuesta para dar soporte a esta característica? Se necesitan 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, se debe añadir la noción de estado al módulo de control. De modo que existan 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 activamente trabajando para llegar a él. Para hacerlo se hace uso de un nuevo método y el patrón *state*[3] y se obtiene el esquema 6.11 (en 6.12 se encuentra la documentación de la aplicación del patrón).

[2]La estabilización es crucial para evitar oscilaciones, reducir el tiempo de respuesta y minimizar sobrepasos, garantizando un control preciso y eficiente. Un sistema bien ajustado responde de manera estable ante perturbaciones externas y optimiza el consumo energético, mejorando la fiabilidad y el desempeño en aplicaciones como robótica y automatización.

[3]El uso de este patrón está explicada en la sección Máquinas de estado.

Figura 6.11: 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 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 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 que involucre múltiples subsistemas. Este controlador general organizará los esfuerzos de cada uno con el fin de llevar a cabo comportamientos más complejos.

Figura 6.12: Documentacion de la aplicación del patrón State en el módulo Control.

PatternApp	**Estados de operación del controlador**
based on	Estado (State)
why	**Cambios previstos**: El controlador lleva a cabo el control dependiendo del estado en el que se encuentre. Podrían cambiar el comportamiento requerido de algunos de los estados definidos o bien podría ser necesario agregar nuevos estados con sus correspondientes comportamientos. **Funcionalidad**: Dependiendo del estado, los métodos control y cicle deben comportarse de manera diferente. A su vez, pueden cambiar de manera dinámica. En caso de que no se esté realizando una acción sobre alguno de los actuadores que
where	Control **is** Contexto State **is** Estado Waiting **is** EstadoConcreto Working **is** EstadoConcreto

6.3.2 Ejemplo

Se utilizará como ejemplo el propuesto en el libro [Dou11] en la sección de 3.4 *Mediator Pattern*.

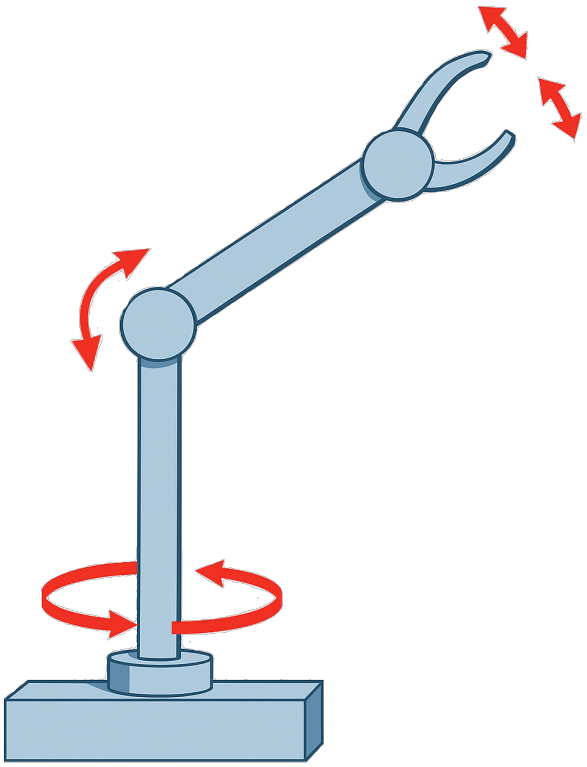
Requisitos

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.

Solución propuesta en el libro de Douglass

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 comporta-

Figura 6.13: Esquema del brazo robótico.



miento. 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 4.3. 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 aplicación del patrón *Mediator*, los autores en [Gam+95] establecen 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 submódulos.

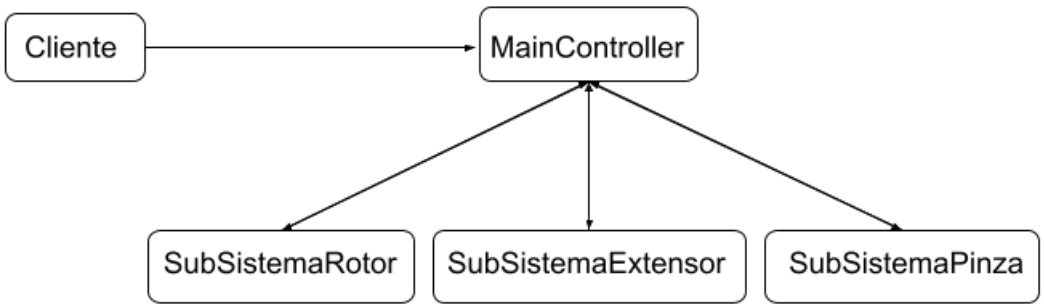
La estructura lograda es similar al patrón, pero originalmente los módulos encargados de ejecutar tareas relacionadas con un actuador específico no necesitaban comunicarse entre sí. Según el propósito del patrón, el objetivo principal es reducir el acoplamiento, evitando que los módulos se refieran directamente entre ellos. Y este objetivo parece estar logrado en el diseño propuesto.

Solución orientada al cambio

Se aplica el concepto de subsistemas que se introdujo previamente, para ello primero se debe identificar las propiedades del mundo físico a controlar. Claramente, lo que se requiere es modificar la posición y el estado de la pinza del brazo, para hacerlo se cuenta con distintos actuadores que intervienen diferentes propiedades físicas. Además, en los requisitos se especifica que se cuenta con una función que genera una orden para cada actuador. De esta manera se define 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 se propone 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 es como la que se describe en la figura 6.14.

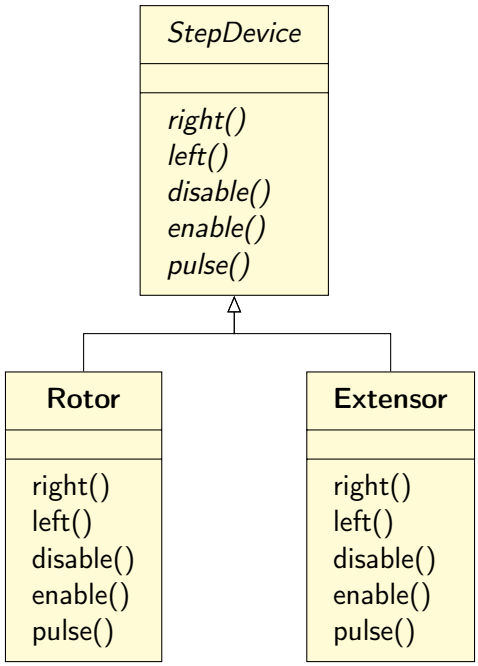
Se puede pensar que el gráfico tiene cierta similaridad conceptual con el patrón *mediator*, donde un módulo **Mediator** coordina el trabajo de los **College** en este caso los subsistemas. Gamma en [Gam+95] indica que el patrón suele ser apropiado para casos en lo que se tiene un conjunto de objetos que se comunican de manera compleja pero fija. En este

Figura 6.14: Diagrama de los componentes del sistema brazo robótico.



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**.

Figura 6.15: Actuadores paso a paso.



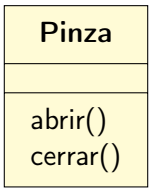
Ahora, se detallarán los módulos que conforman cada componente comenzando por los módulos básicos que debemos crear para representar el hardware dado. Los módulos del gráfico 6.15 representan motores paso a paso[4], los cuales para ser controlados se deben primero

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

configurar su dirección invocando a los métodos `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.

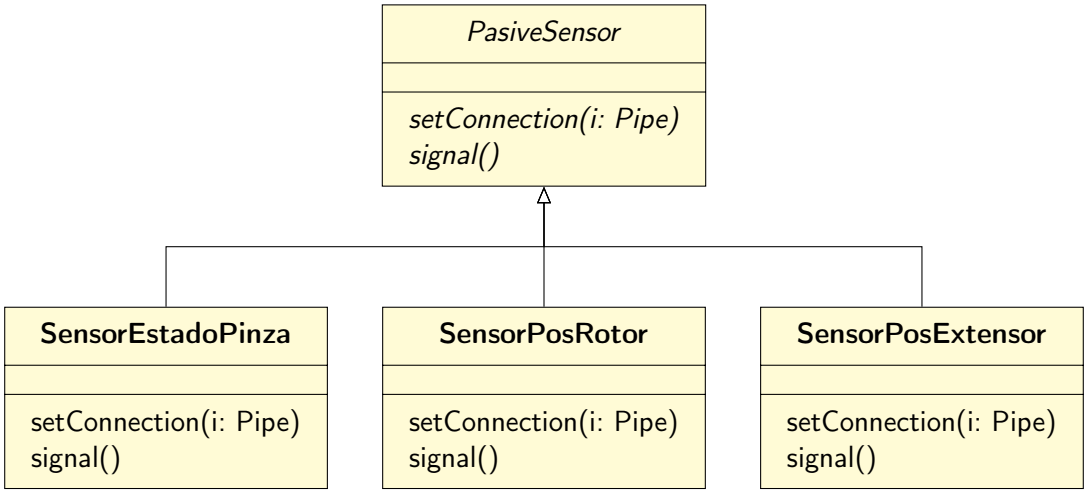
En cambio, en el caso de la pinza (ver 6.16) se utiliza un dispositivo que tiene dos estados, abierto o cerrado, por lo que solo se tienen dos métodos para cambiar entre estados.

Figura 6.16: Interfaz módulo Pinza.



En 6.17 se encuentran definidos los sensores asociados a cada actuador, estos heredan 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.

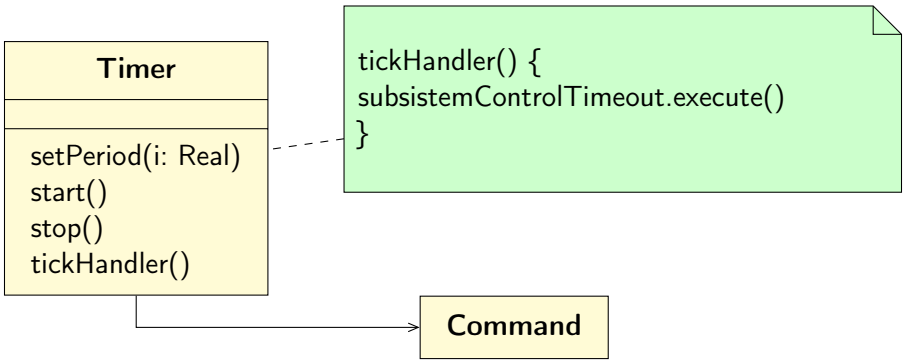
Figura 6.17: Sensores del brazo robótico.



Para completar los módulos que conforman a un subsistema de control falta mostrar como se define el **Controller**. Se diferencian dos tipos de **Controllers**, uno para los actuadores 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 posición actual, decida

y actúe. Para eso, en el diseño del robot desmalezador [Pom+24] se propuso la creación de una interrupción temporizada, la cual se ejecuta cada $1.5ms$ y desencadena el control de los subestimas necesarios. El módulo encargado de esta tarea es **Timer** y su interfaz se puede ver en la figura 6.18.

Figura 6.18: Módulo Timer



El método `tickHandler()` ejecutará comandos siguiendo patrón *Command* por cada subsistema que lo necesite, por lo tanto se iniciará en cada uno el ciclo de control. En particular, la orden es utilizada para desacoplar como es invocado el inicio del **Timer**. En la figura 6.19 se puede observar la documentación de la aplicación del patrón *Command* para este caso.

Una vez solucionada la invocación, se debe introducir un nuevo módulo que será usado en el **Controller**. Ahora, se tienen dos modos de operación en este último, cuando se está trabajando, es decir, se tiene configurado un *set-point* y aun no se alcanzó, y cuando no. Dependiendo de cual de los dos esté activo el comportamiento será diferente, por ejemplo, como no se apaga el **Timer** cuando no se está haciendo un movimiento, es necesario ignorar las llamadas de control. Para esto se hace uso del patrón *State* de Gamma[Gam+95], el cual está explicado en la sección 6.6. Esta aplicacion del patron *State* al caso de manejo de una interrupcion está explicado en la sección Organización de la ejecución. En particular, el **Controller** delegará dos métodos de su interfaz al módulo del patron *state*, `control()` y `move()` (luego lo se verá en detalle). La estructura de módulos resultante es la presente en 6.20.

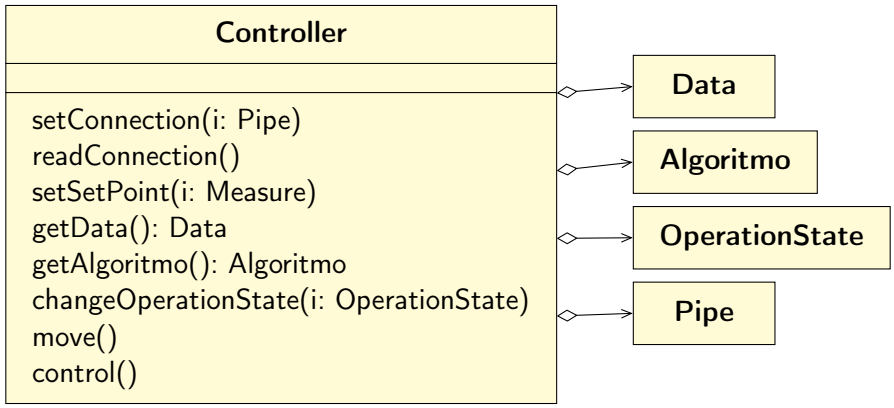
Siguiendo con la guía de creación de subsistemas, también se debe definir el módulo **Data** que almacena la información utilizada por el mismo y el módulo **Algoritmo** que encapsula los cálculos necesarios para determinar que cambio aplicar. Se pueden observar en la figura 6.21.

Con todos los módulos necesario ya definidos, la estructura del **Controller** se puede ver en la figura 6.22.

Figura 6.19: Documentacion de la aplicación del patrón Command para el desacople de ejecuciones que invoca el Timer.

PatternApp	**Comando para manejar interrupciones generadas por el Timer. Sustitución de callback**
based on	Orden (Command)
why	**Cambios previstos**: Las acciones a llevar a cabo ante una interrupcion provocada por el Timer podrían cambiar; o incluso podría cambiar el receptor de dichas acciones, que actualmente es Controller. **Funcionalidad**: Se mantienen los niveles de abstracción. El módulo Timer, desconoce la existencia de módulos de niveles superiores como el Controller.
where	Command **is** OrdenConcreta Timer **is** Invocador Controller **is** Receptor

Figura 6.22: Estructura módulo Controller



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.

Figura 6.20: Módulos que forman parte del patrón State que son necesarios para complementar al módulo Controller

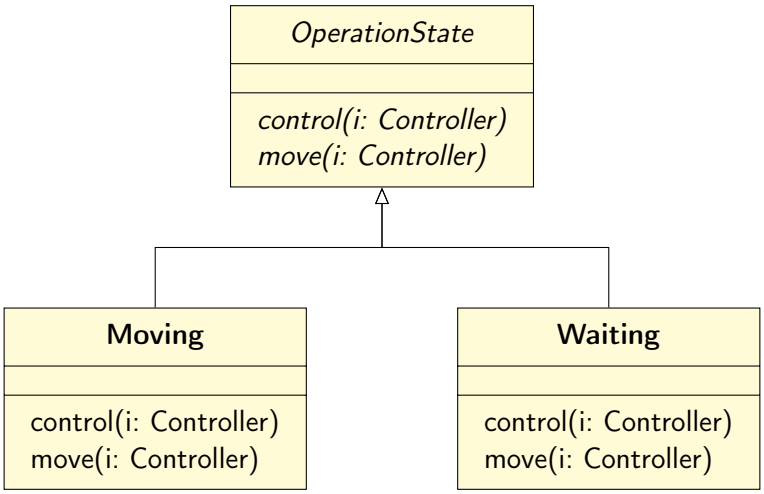
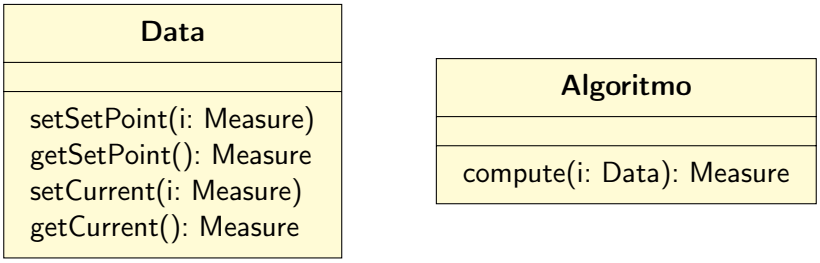


Figura 6.21: Módulos complementarios a Controller.



- `getData`: devuelve el objeto Data asociado al **Controller**.
- `getAlgoritmo`: lo mismo que con Data.
- `changeOperationState`: cambia el estado del modo de operación, en la configuración básica las transiciones posibles son de *Waiting* a *Moving* y viceversa.
- `move`: es 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.

Suponga que un cliente quiere utilizar el subsistema, ¿qué métodos tiene que ejecutar si ya está inicializado? 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 6.20 (recordar que el *set-point* del subsistema del rotor es una medida en grados).

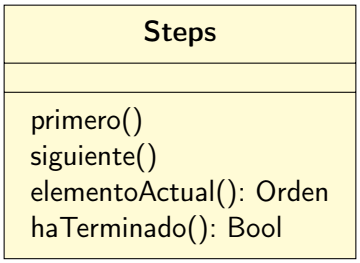
Código 6.20: Ejemplo de implementación del método control del módulo Controller.

```
1   void control(Controller controller) {
2       setPoint = data.getSetPoint();
3       current = data.getCurrent();
4       dif = controller.algoritmo.calculate()
5       if abs(dif) > LIMIT_ACCEPT && dif > 0 {
6           rotor.left();
7           rotor.pulse;
8           controller.changeOperationState(moving);
9           return;
10      }
11      if abs(dif) > LIMIT_ACCEPT && dif < 0 {
12          rotor.right();
13          rotor.pulse;
14          controller.changeOperationState(moving);
15          return;
16      }
17  }
```

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.

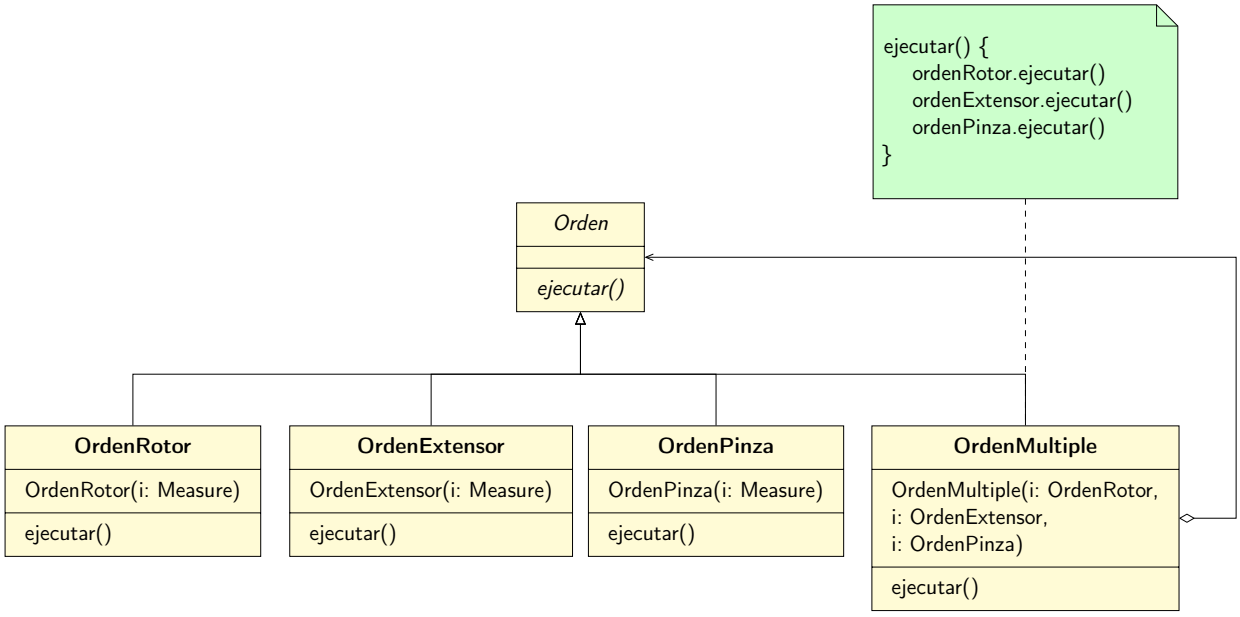
Ya se definieron los subsistemas, por lo que siguiendo con el diagrama 6.14, falta entonces abarcar el **MainController**. Este último se encarga de brindar una interfaz al cliente y a su vez coordinar cada subsistema. Recordando los requerimientos, sabemos que tenemos una función que generar una secuencia de pasos donde cada uno implica una acción sobre los tres actuadores. ¿Cómo se puede diseñar este comportamiento? Una opción es usar un *iterator* para recorrer los pasos y aplicar el patrón *command* para cada uno. En particular, se aplica una de las modificaciones del patrón mencionada por Gamma[Gam+95], en donde una orden al ser ejecutada desencadena la ejecución del resto de las ordenes. Esto se puede ver en los gráficos 6.23, 6.24 y 6.25. A su vez en 6.21 se encuentra un ejemplo de una posible implementación del módulo OrdenRotor.

Figura 6.23: Interfaz módulo Steps y documentación de aplicación del patrón Iterator.



PatternApp	**Ordenes a ejecutar para realizar el movimiento requerido.**
based on	Iterador (Iterator)
why	**Cambios previstos**: Se pueden agregar, quitar o modificar ordenes pero siempre con la posibilidad de iterar. Además, puede cambiar la estructura de datos subyacente. **Funcionalidad**: Se logra recorrer todas las ordenes de manera secuencial.
where	Steps **is** IteradorConcreto MainControler **is** AgregadorConcreto

Figura 6.24: Interfaces de las ordenes de ejecución para cada subsistema.



Código 6.21: Ejemplo de implementación del módulo OrdenRotor

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

Figura 6.25: Documentación de la aplicación del patrón Command para el desacople de ordenes a ejecutar en cada actuador del brazo en un paso.

PatternApp	**Comando para manejar las acciones que deben ser ejecutadas en un paso en cada actuador.**
based on	Orden (Command)
why	**Cambios previstos**: Las acciones a llevar a cabo para cada actuador pueden cambiar, se pueden agregar o quitar actuadores. **Funcionalidad**: Se logra ejecutar una acción sobre todos los actuadores configurados con una sola acción del invocador.
where	Orden **is** Orden OrdenRotor **is** OrdenConcreta OrdenExtensor **is** OrdenConcreta OrdenPinza **is** OrdenConcreta OrdenMultiple **is** OrdenConcreta MainController **is** Invocador RotorController **is** Receptor ExtensorController **is** Receptor PinzaController **is** Receptor

Se tiene un iterador de ordenes 6.23, 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.

Figura 6.26: Estructura del módulo MainController.

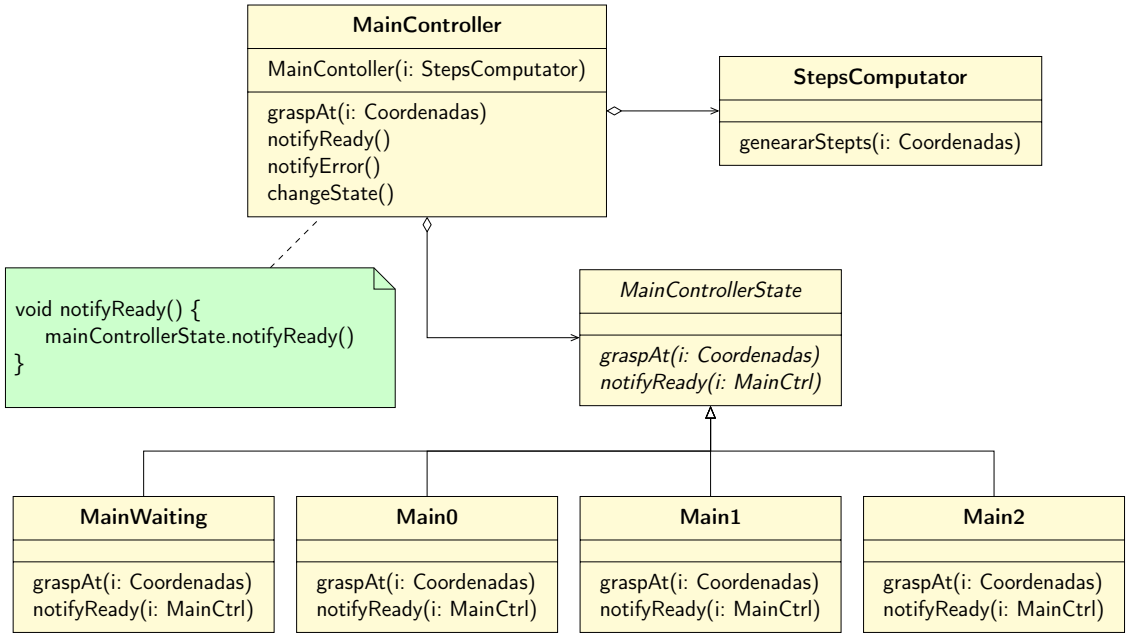
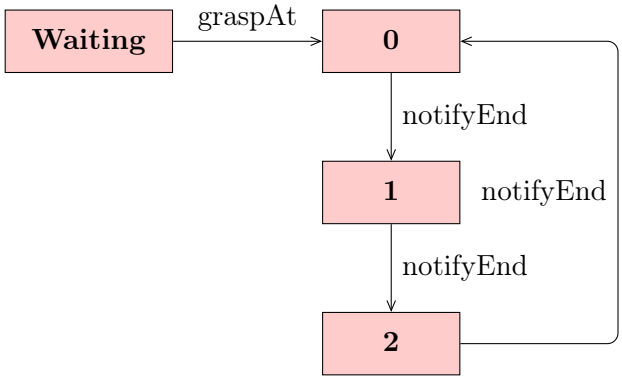


Figura 6.27: Documentación de la aplicación del patrón *State* para el manejo de ejecución de pasos completos.

PatternApp	**Estados de operación del controlador principal**
based on	Estado (State)
why	**Cambios previstos**: El controlador principal llevará a cabo el control de los subsistemas de control, dependiendo del estado en el que se encuentre. Podrían cambiar el comportamiento requerido de algunos de los estados definidos o bien podría ser necesario agregar nuevos estados con sus correspondientes comportamientos. **Funcionalidad**: Teniendo en cuenta que para poder ejecutar un nuevo paso se deben haber finalizado con éxito todas las operaciones sobre actuadores, se introducen estados por cada orden terminada, a fin de que solo al hacer una transición completa se puede ejecutar un nuevo paso.
where	MainController **is** Contexto MainControllerState **is** Estado MainWaiting **is** EstadoConcreto Main1 **is** EstadoConcreto Main2 **is** EstadoConcreto Main3 **is** EstadoConcreto

Hasta la introducción de **MainController** todo lo que se hizo fue aplicar el concepto de subsistemas. Pero para poder cumplir los requerimientos particulares del ejemplo, es necesario introducir ciertos cambios. Como se debe ejecutar un paso a la vez, se tiene que esperar a que todas las ordenes enviadas en el paso anterior a los subsistemas se encuentren finalizadas. Para ello se hace uso del patrón *state* (ver la documentación de la aplicación del mismo en 6.27), al cual se le delegarán los métodos `graspAt` y `notifyReady`. La idea es cambiar el comportamiento de estos dependiendo de en qué estadio se encuentra el sistema. Es decir, por cada orden enviada y terminada se cambia de estado, llevando así una cuenta de la cantidad de acciones finalizadas. Solo cuando se llegue a las cuatro ordenes finalizadas se podrá ejecutar un nuevo paso. Esto se puede ver en el siguiente gráfico de estados 6.28.

Figura 6.28: Transiciones de estados del MainController



Por lo que, el estado *Waiting* no implementa `notifyOrder` y a su vez 0, 1 y 2 no implementan `graspAt`. En cambio, *Waiting* implementan `graspAt` la cual computa los pasos y ejecuta el primero utilizando el iterator. Luego, por cada `notifyEnd` los subsistemas le avisan al MainController que terminaron la orden y por cada una 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.

Conclusión

La solución propuesta mejora varios aspectos a la que utiliza como criterio de división la funcionalidad. Por un lado, se anticipa a cambios probables tales como los de la tabla 6.2.

Cuadro 6.2: Anticipos al cambio del diseño propuesto para el brazo robótico.

Item de cambio	**Naturaleza del cambio**	**Manejo del cambio**
Hardware	- Agregar, quitar o modificar actuadores y sensores. - Cambios en la comunicación.	Los subsistemas están desacoplados del control principal, permitiendo agregar, quitar o modificar componentes de manera independiente respetando las interfaces definidas. La modularización permite gestionar sensores, actuadores y comunicación por separado. La lógica de control puede modificarse fácilmente cambiando la implementación del módulo Algoritmo.
Lógica de control	- Cambios en decisiones de trabajo, como ejecución secuencial o criterios de avance.	El comportamiento está encapsulado en módulos. La máquina de estados que controla la ejecución puede modificarse fácilmente agregando o editando estados en el iterador Steps.

Además, se fomenta la reutilización del código, permitiendo aplicar las soluciones existentes a distintos problemas. Cualquier modificación futura requerirá menos esfuerzo por parte de los desarrolladores, ya que el sistema será más fácil de comprender. La documentación también contribuye a esta facilidad de mantenimiento. Los cambios están encapsulados, por lo que el control de calidad se reduce a verificar módulos individuales y no secciones de código enteras.

6.4 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 está 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 una operación 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.

Tradicionalmente esta tarea es centralizada en un único módulo, representado por un método que se encarga de manejar la interrupción y almacenar el estado resultante. De esa manera se obtiene una estructura poco resiliente al cambio, en la cual se dificulta reutilizar código y las funcionalidades están altamente acopladas. Esto es, por ejemplo, si se modifican los cálculos para obtener el resultado deseado, se debe modificar a su vez toda la implementación. Así como en caso de que cambie la manera de transmitir la información relacionada a la interrupción.

Es por esto que resulta útil una forma general de atacar este problema desde el punto de vista del diseño para el cambio, en el trabajo que se realizó sobre el robot desmalezador [Pom+24], se utilizó una estructura de módulos que lleva a cabo las actividades necesarias para integrar al sistema los sensores que generan interrupciones. La estructura del mismo se puede observar en la figura 6.29.

Se distinguen 4 módulos principales, **Pin** que es la representación en el diseño del hardware asociado al dispositivo que genera e introduce la interrupción, esta última 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). Los datos por lo general son un *timestamp* y la cantidad de veces que sucedió dicho evento. Por último, **Buffer** actúa como buffer de la información recibida, posee un método que permite obtener el valor recibido. Cuando es invocado toma los valores necesarios que se almacenan en el **Collector** y realiza los cómputos necesarios para calcular el dato pedido.

Para analizar esta estructura en mayor detalle, se presenta un ejemplo de aplicación. Se considera un sensor *Hall* montado en una rueda, utilizado para medir la velocidad de giro. Cada vez que el sensor detecta un campo magnético (ver gráfico 6.30), emite una interrupción que es gestionada por el módulo **Count**, el cual actúa como un comando y, por lo tanto, conoce qué funciones ejecutar dentro del módulo **Collector**.

Figura 6.29: Estructura componentes para la lectura de un sensor activo.

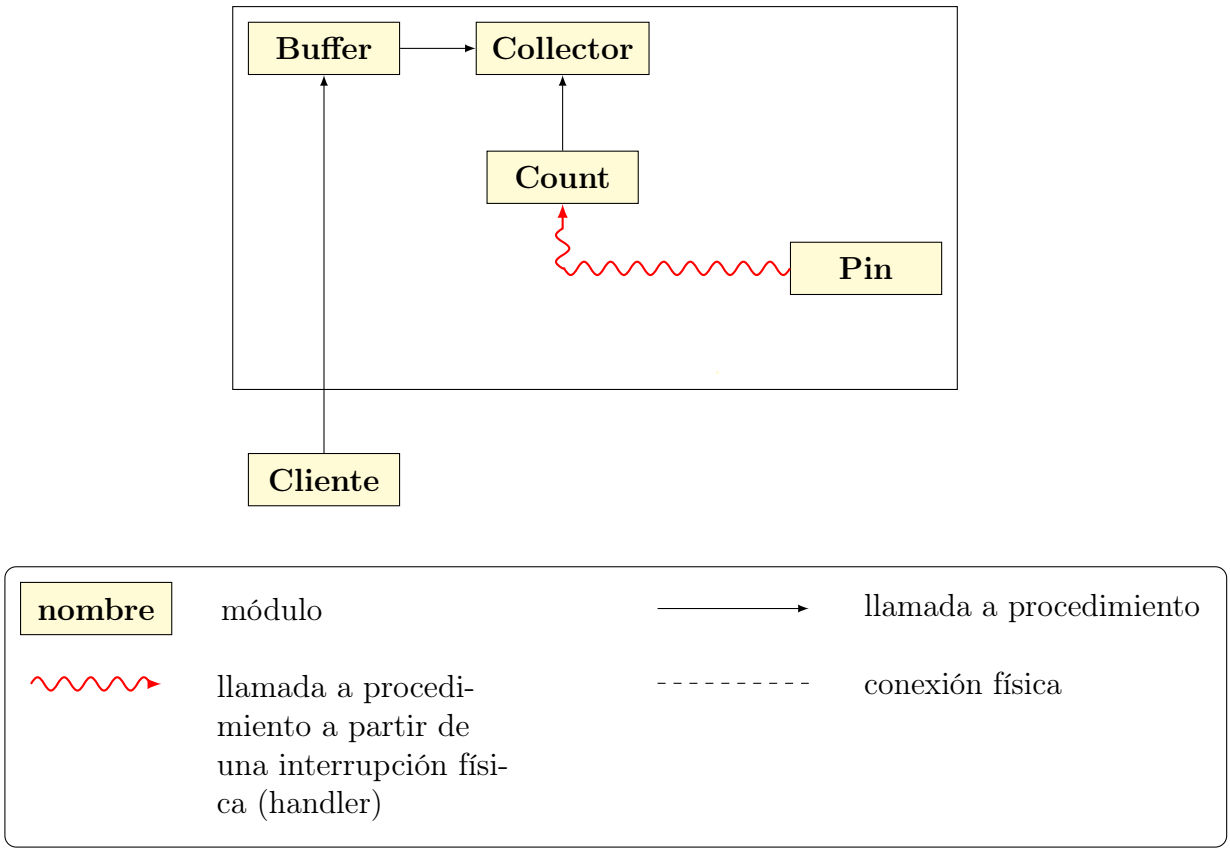
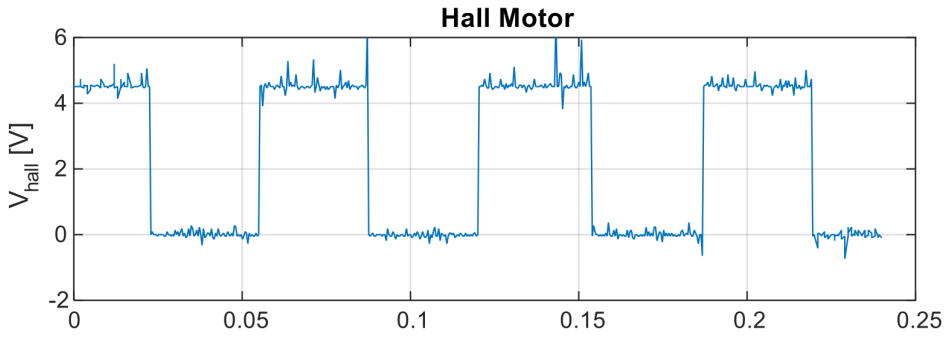


Figura 6.30: Ejemplo señales emitidas por un sensor hall al detectar una variación del campo magnético. Cada pico de voltaje provoca que el microcontrolador genere una interrupción. Imagen extraída de [BCD18].



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.

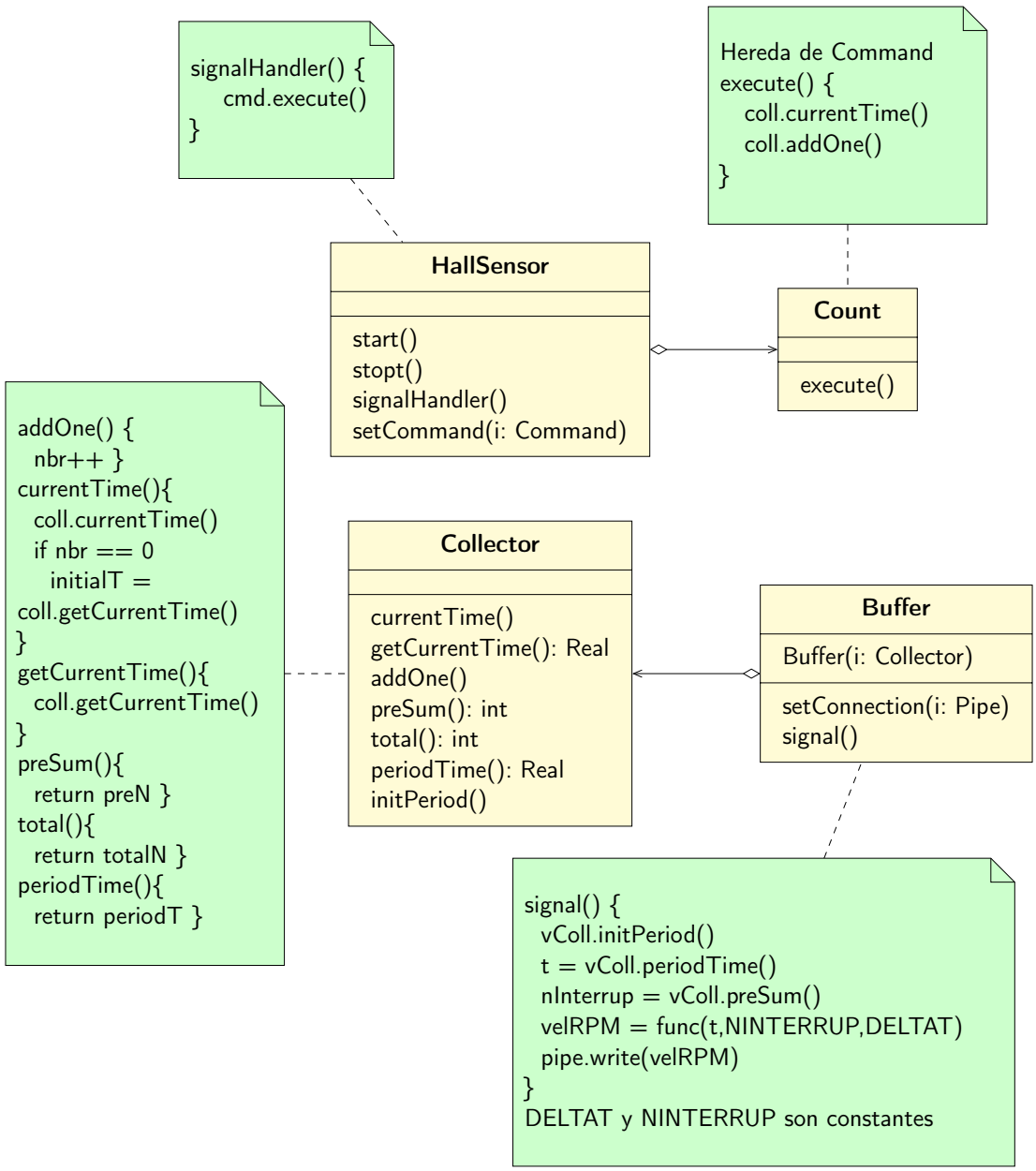
Y el módulo **Collector**:

- `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 requiera el valor de velocidad al **Buffer** este llamará a los métodos necesarios del **Collector** para obtener información, computará el valor y lo retornará. De esta forma un cliente 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.

La estructura de módulos resultante de aplicar la técnica al ejemplo seria la de la figura 6.31.

Figura 6.31: Ejemplo módulos para obtener la información referida a la velocidad.



Periodicidad

Algunos sistemas se encargan de mostrar, almacenar o verificar información de sensores cada cierto tiempo. No resulta crítico perder valores intermedios, es decir, no 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 se verifique la información

de los sensores, llamando a ciertos métodos que obtengan esta directamente del hardware, y luego se ejecute una función *sleep* que bloquea la ejecución determinada cantidad de tiempo. Esto tiene varias desventajas, primero el tiempo de ejecución la rutina alarga el periodo, si se quiere agregar otras funcionalidades a realizar mientras se espera el periodo se debe dividir el *sleep* y calcular tiempo de ejecución.

Una solución que 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 Uno**. Se registrará 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 se tienen 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 se logra librear al procesador en los tiempos de espera y se desacopla esta funcionalidad.

6.5 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.

Código 6.22: Código ejemplo

```
1  void ButtonDriver_eventReceive(ButtonDriver* const me) {
2      Timer_delay(me->itsTimer, DEBOUNCE_TIME);
3      if (Button_getState(me->itsButton) != me->oldState) {
4          /* must be a valid button event */
5
6          me->oldState = me->itsButton->deviceState;
7
8          if (!me->oldState) {
9
10             /* must be a button release, so update toggle value
                  */
11             if (me->toggleOn) {
12                 me->toggleOn = 0; /* toggle it off */
13                 Button_backlight(me->itsButton, 0);
```

```
14                  MicrowaveEmitter_stopEmitting(me->
                       itsMicrowaveEmitter);
15              }
16              else {
17                  me->toggleOn = 1; /* toggle it on */
18                  Button_backlight(me->itsButton, 1);
19                  MicrowaveEmitter_startEmitting(me->
                       itsMicrowaveEmitter);
20              }
21          }
22          /* if  i t s  not a button release, then it must
23          be a button push, which we ignore.
24          */
25      }
26  }
```

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

Parece ser un problema 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.

6.6 Máquinas de estado

Muchos sistemas ajustan su comportamiento durante la ejecución en función de diversas causas, como interacciones con el entorno o requisitos internos. Por ejemplo, un sistema de control de microondas no iniciará el calentamiento si la puerta está abierta. En este caso, la condición de la puerta representa un estado interno que restringe ciertos comportamientos, permitiendo otros en estados diferentes. De manera similar, distintas partes de un sistema pueden responder a estados específicos. Un ejemplo es la dirección de giro de un motor en un robot, donde el estado del motor puede ser "girar hacia adelante" o "girar hacia atrás", determinando su comportamiento. Además, un sistema en ejecución puede encontrarse en estado *trabajando* y, al finalizar una operación, cambiar a estado *esperando*, lo que también altera su comportamiento. Otro caso es un sistema de climatización, donde el termostato puede estar en estado *calentando* o *enfriando*, según la configuración establecida. Por lo tanto, la gestión de estados, sus transiciones y los cambios de comportamiento asociados a estos es un aspecto fundamental en el diseño de sistemas.

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 el código 6.23.

Código 6.23: Ejemplo de manejo de estados tradicional, en el caso del microondas.

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

// otra opcion

calentar() {
    switch (estado) {
      case PuertaAbierta {
          return
        }
      case otherwise {
          magnetron.encender()
      }
  }
}
```

Este enfoque 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 varios métodos. 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.

Este tipo de soluciones es ampliamente utilizado, incluso en el software del robot desmalezar previo al trabajo realizado en [Pom+24]. En particular, se definen diferentes estados de operación del robot, y se utiliza un gran *switch-case* que es llamado en loop. En este se decide que hacer en base al estado actual. En el código 6.24, se puede ver en detalle la estrategia utilizada.

Código 6.24: Main loop del previo firmware del robot desmalezador

```
switch (ESTADO) {
  case DUTY_REMOTO:
    duty_remoto();
    break;
  case RPM_REMOTO:
    rpm_remoto();
    break;
  case DUTY_PC:
    duty_pc();
    break;
  case RPM_PC:
    rpm_pc();
    break;
  case CALIBRACION:
    calibracion();
    break;
  case PERDIDA_SENAL:
    perdida_senal();
    break;
  case EMERGENCIA:
    EMERGENCY();
    break;
  default:
    ESTADO = PERDIDA_SENAL;
    break;
}
```

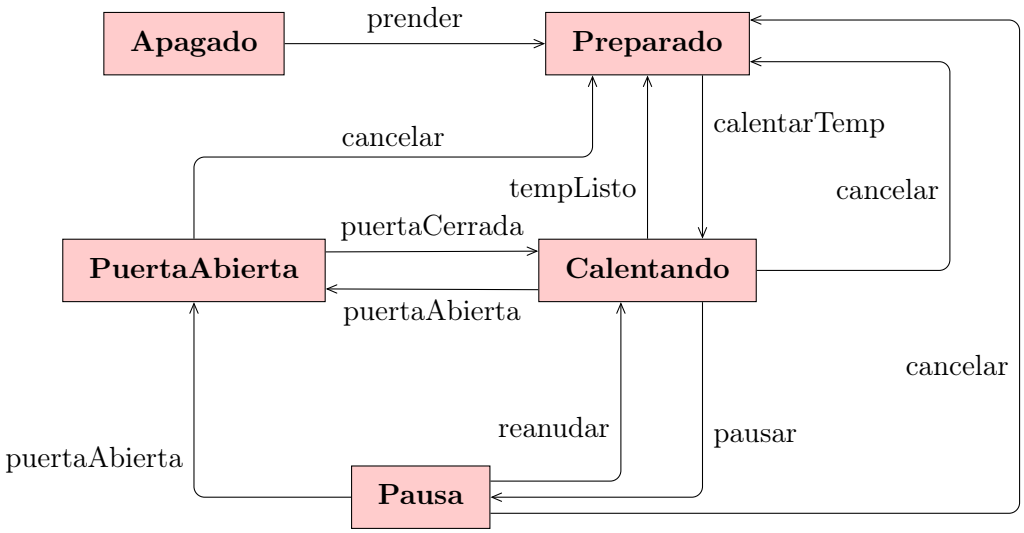
Antes de ver un enfoque desde el punto de vista del diseño orientado al cambio, se tratará en mayor profundidad el ejemplo planteado en el libro [Dou11] y cómo se aplica una solución similar a la nombrada.

Transiciones basadas en eventos

Se puede describir un sistema en el que las transiciones entre los estados son provocadas por eventos específicos. Por ejemplo, en el caso de un sistema de microondas, la apertura de la puerta constituye un evento que desencadena una transición, al igual que la pulsación del botón de Cancelar durante el calentamiento. Este tipo de sistema presenta características de una máquina de estados, donde el comportamiento se define mediante una secuencia de transiciones. A continuación, se presenta un "*state chart*" extraído del libro que describe el funcionamiento del microondas:

El funcionamiento principal del sistema se representa mediante estados y transiciones, donde cada transición implica un comportamiento específico relacionado con la salida del estado actual y la entrada al siguiente. Por ejemplo, si el sistema se encuentra en el estado *PuertaAbierta* y se recibe el evento *puertaCerrada*, se realizará una transición al estado *Calentando*. Durante este proceso, el magnetrón será activado para emitir las ondas necesarias que calentarán la comida.

Figura 6.32: State chart microondas



A simple vista parece un comportamiento complejo y eso que solo es un ejemplo simplificado, por lo que en la vida real los casos pueden ser mucho más extensos. Este es uno de los motivos por los cuales es útil contar con un buen diseño. El desarrollador debe tener en cuenta que sea fácil modificar las transiciones y también agregar y quitar estados. En [Dou11], el autor 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 [Gam+95, State], menciona que Tom Cargill[5] propuso esta implementación y efectivamente es una manera de convertir código condicional en algo que se parece a una tabla.

Código 6.25: Código ejemplo Douglass State Table

```
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? */
```

[5]Escritor de *C++ Programming Style.Addison-Wesley*, 1992

```
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
37Mutex_release(me->itsMutex);
```

```
38 }
```

En el ejemplo de código 6.25, que es una porción de la solución propuesta en el libro, es posible 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, las interfaces deben intentar ser independientes de la estructura de datos que se utiliza para implementarlas.

Como otro enfoque, se propone utilizar el patrón *State* de Gamma con ciertos ajustes para permitir a los estados transicionar a otros por si mismos. Básicamente, el patrón establece la creación de un módulo por cada estado del sistema con el fin de cambiar la implementación para que se ajuste al comportamiento cuando el sistema se encuentra en dicho estado. Esto nos permite transicionar dinámicamente entre estados, ya que el cambio de estados esta representado por cambiar de módulo.

Para permitir que cada estado pueda transicionar de manera independiente a otros estados lo que se hace 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 porción de código 6.26 perteneciente al diseño del robot desmalezador. Y para llamar a una transición de estado se ejecuta el código 6.27.

Código 6.26: Código ejemplo transición robot desmalezador

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

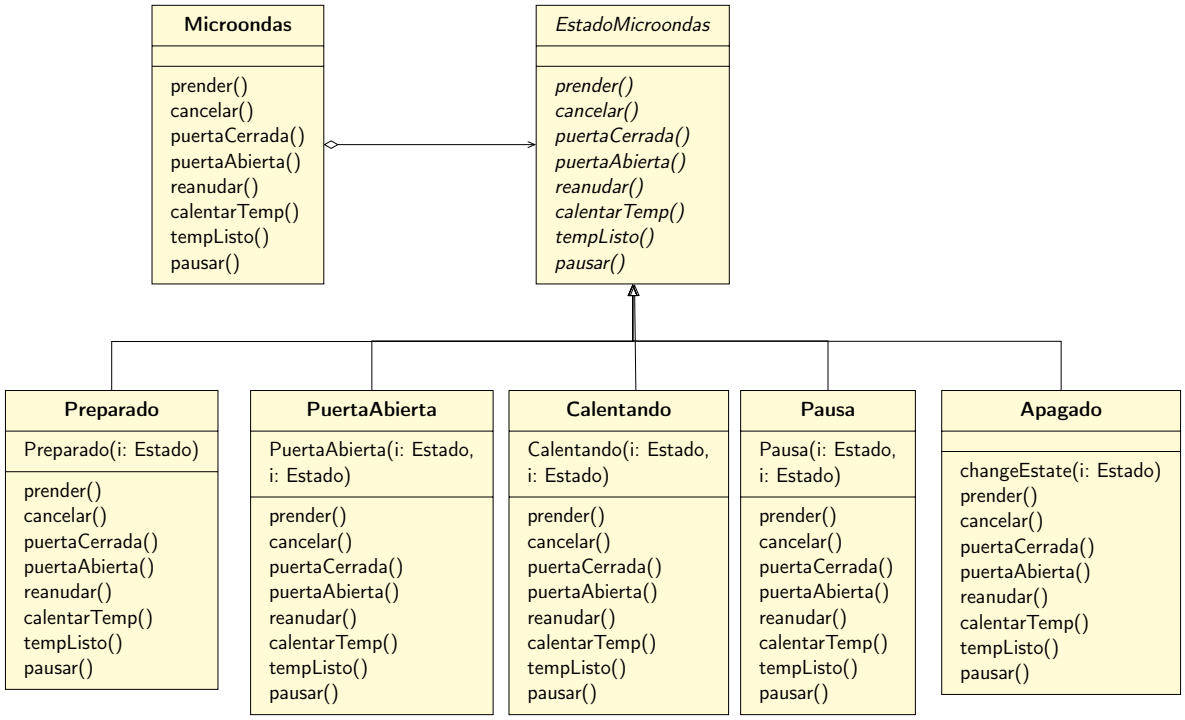
Código 6.27: Ejemplo transición de estado.

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

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

Volviendo al ejemplo del microondas, aplicando este enfoque de diseño, se obtiene un módulo que provee un método que maneja cada evento y otro que delega al estado cada manejo de evento. Como se puede observar en la figura 6.33.

Figura 6.33: Módulos participantes del patrón state en el ejemplo del horno microondas.



PatternApp	**Estados del microondas**
based on	Estado (State)
why	**Cambios previstos**: En base al estado el microondas responde a diferentes eventos de múltiples maneras. Estos estados pueden modificarse, agregar nuevos o quitar otros. **Funcionalidad**: Cada método del módulo Microondas es un handler para un evento particular. La respuesta del sistema ante los eventos cambia en base al estado actual del mismo. Ademas, los eventos pueden desencadenar cambios de estado durante la ejecución de su handler.
where	Microondas **is** Contexto EstadoMicroondas **is** Estado Preparado **is** EstadoConcreto PuertaAbierta **is** EstadoConcreto Calentando **is** EstadoConcreto Pausa **is** EstadoConcreto Apagado **is** EstadoConcreto

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

Código 6.28: Implementación puertaAbierta

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

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

De esta forma, se logra 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 propio módulo, 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 un nuevo módulo 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 módulo 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.

6.7 Integridad de la información

Douglass en [Dou11], afirma que cuando se intenta construir software embebido es importante poner esfuerzo en garantizar de alguna manera las siguientes propiedades:

- Reliability (confiabilidad) en los sistemas embebidos se refiere a la capacidad del sistema de operar de manera continua y correcta durante un período de tiempo determinado, sin fallos que interrumpan su funcionamiento. Es fundamental porque muchos sistemas embebidos, como los que controlan robots, vehículos o dispositivos médicos, deben garantizar la ejecución exitosa de sus tareas bajo condiciones normales o adversas. Una alta confiabilidad se traduce en una mayor disponibilidad del servicio, menor tasa de fallos y una experiencia más segura para los usuarios.

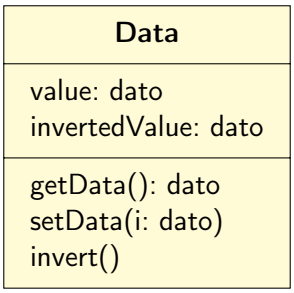
- Safety (seguridad) es igualmente crucial en sistemas embebidos, ya que garantiza que el sistema no genere riesgos inaceptables para personas, equipos o el entorno. No basta con que el sistema funcione correctamente; debe hacerlo de manera que prevenga accidentes, daños físicos o pérdidas. Esto implica identificar peligros potenciales, estimar su riesgo (producto de su severidad y probabilidad) y diseñar mecanismos de mitigación adecuados, como pasar a estados seguros ante fallos o suministrar información confiable a operadores. En aplicaciones críticas, como aeronáutica, medicina o automatización industrial, la seguridad es un requisito esencial porque cualquier error puede tener consecuencias graves e irreversibles.

Además, agrega que uno de los origenes más comunes de inconvenientes relacionados fallas de seguridad o confiabilidad es la corrupcion de la información. Un pulso electromagnético o fallas en el hardware pueden causar daños en la información dejándola comprometida, por ejemplo un Bit flip o perdida parcial de cierta zona de la memoria. 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 creando **redundancia**. En el libro, Douglass

propone el uso de dos estrategias, una para datos que ocupan mucha memoria y otra para pequeños valores.

Estas estrategias suelen ser agregadas al código modificando la implementación de los métodos del módulo encargado de almacenar la informacion, o agregando verificaciones en las partes del código que la utilizan. Es decir, si por ejemplo se utiliza el módulo **Data** (ver figura 6.34) se realiza `data = data.getData()` para obtener el valor y luego se ejecutan las validaciones necesarias para verificar la integridad del mismo. Esto último provoca código repetido en cada llamado a `getData()` y a su vez ante cualquier cambio en el dato, la estrategia de validación utilizada puede dejar de ser aplicable, obligando a modificar múltiples porciones de código.

Figura 6.34: Interfaz del módulo Data,



Desde el punto de vista del diseño de software se puede aportar una mirada crítica hacia las propuestas el libro [Dou11, pág. 357], dado que plantean 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 oculten como se almacena la información sino que se encarguen de verificar su integridad.

En cambio, una posible forma de solucionar esto desde el diseño orientado al cambio es utilizar el patrón *Decorator* que nos permite añadir y remover de manera dinámica funcionalidades. En este caso, la capacidad de verificar la información. Otra ventaja, además de poder hacerlo de manera dinámica, es que mantenemos separadas las responsabilidades de cada módulo. A diferencia de los propuesto en el libro, no se combina el almacenaje de la información con su verificación, de manera que cada porción puede ser implementada de manera independiente. La solución se muestra en el gráfico 6.35 con su correspondiente documentación en 6.36.

Figura 6.35: Ejemplo de aplicación del patrón *Decorator* para garantizar la integridad de la información.

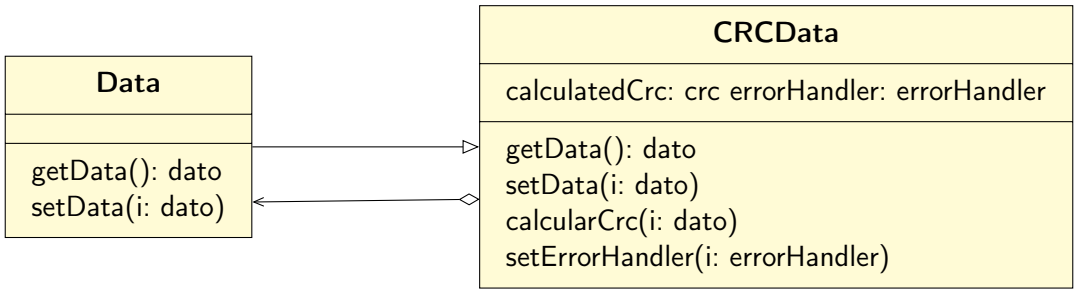


Figura 6.36: Documentación de la aplicación del patrón *Decorator* al caso de verificación de la integridad de la información.

PatternApp	**Modulo decorador que extiende dinámicamente las funcionalidades de otro módulo sin cambiar la implementación.**
based on	Decorador (Decorator)
why	**Cambios previstos**: Pueden agregarse y quitarse métodos de verificación. **Funcionalidad**: Antes de devolver la información se realiza una verificación de la integridad de la información almacenada.
where	Data **is** ComponenteConcreto CRCData **is** DecoradorConcreto

Código 6.29: Ejemplo de implementación métodos setData y getData del módulo CRCData.

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

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

En la figura 6.29 se observa un ejemplo de implementación de los métodos `setData` y `getData`. En el cual, se utiliza el método `calcularCrc` para obtener el código de detección de errores de tamaño fijo basado en un polinomio cíclico. Entonces, comparando el código calculado cuando se guardo la información con el calculado en el momento de obtenerla se puede saber si la información está o no corrupta.

De esta forma, aplicando el patron *decorator* se logra añadir una capa de protección a la integridad de la información, a su vez permite combinar diferentes técnicas usando este patrón.

6.8 Verificación de precondiciones.

El autor del libro [Dou11] comenta que uno de los problemas más grandes que observa en el desarrollo de sistemas embebidos es que muchas funciones tiene precondiciones para ejecutarse correctamente pero que rara vez se verifica que todas se cumplan. Además, el procedimiento común de informar precondiciones inadecuadas en una función en **C**[6] es a traves del valor de retorno (-1 en caso de errores, 0 en el contrario). Por lo tanto, el encargado de manejar el error es la función que invocó a la segunda, generando así un acoplamiento extra en el código. Esto provoca una complicación que muchas veces deriva en un mal manejo de los errores. Por ejemplo, supongamos que se tiene un módulo que exporta una función que permite guardar un valor, `setValor(i: Valor)`. Existen múltiples posibles implementaciones, estas son algunas:

- Una posible implementación, consiste en que la función no realice ninguna verificación y simplemente guarde en una variable el valor pasado como argumento. Esto puede generar un error en el momento si el tipo no coincide 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 algún función requiere como precondición que siempre sea positivo, se puede provocar un error en el futuro. Por otro lado, esta implementación obliga a que todos las funciones 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, se basa en verificar si el valor es permitido, haciendo que la función retorne un valor inidicando el resultado de la validación, 0 en caso afirmativo o -1 en caso contrario. Este enfoque es mejor que no realizar manejo alguno, pero como se comentó en el principio, se está 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:

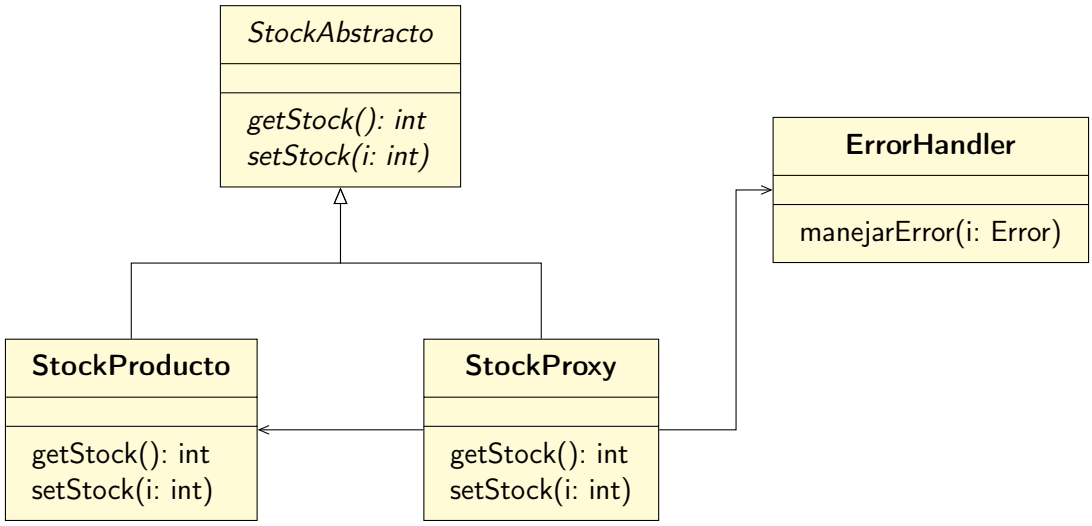
[6] Lenguaje de programación comúnmente utilizado en sistemas embebidos.

- 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.

Siguiendo esos conceptos, las precondiciones necesarias por las funciones son garantizadas desde los módulos que almacenan los datos utilizados por estas últimas. 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.

A continuación, se presenta un ejemplo en el que se dispone de un módulo denominado *StockProducto*, encargado de almacenar información relacionada con el *stock* de los productos. Dado que se trata de información relevante, es necesario incorporar validaciones tanto en el momento del almacenamiento como en la consulta de datos. Para el almacenamiento, se debe verificar que la cantidad ingresada no sea negativa, mientras que, al consultar, es fundamental asegurar que el usuario cuente con los permisos requeridos.

Figura 6.37: Uso del patrón Proxy en el ejemplo del módulo stock y documentación del mismo.



PatternApp	**Acceso seguro stock de productos**
based on	Proxy (Apoderado)
why	**Cambios previstos**: Las diferentes precondiciones a cumplir pueden variar con el tiempo. **Funcionalidad**: Controlar el acceso al stock, en particular con el fin de preservar la integridad.
where	StockAbstracto **is** Sujeto StockProducto **is** SujetoReal StockProxy **is** Proxy

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 [Gla+] 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 los métodos de *StockProducto*. 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 **StockProducto**.
- Extender funcionalidades, el proxy puede evolucionar independientemente del modulo SujetoReal (ver A.6), 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.

6.9 Organización de la ejecución

En un sistema robótico embebido se suelen ejecutar múltiples tareas para lograr un cierto objetivo. 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 presentes en el sistema en tiempo real.

Una implementación simple para realizar el comportamiento mencionado, consite en crear funciones para cada parte del proceso y llamarlas en *loop* desde la función *main*, y si es necesario añadir un tiempo de espera entre cada ejecución mediante *sleeps*. La principal desventaja que conlleva este enfoque es que los sistemas robóticos son en tiempo real, es decir, por ejemplo se pueden perder *inputs* de sensores si no se los maneja de manera correcta. Además, los tiempos de espera son bloqueantes por lo que se desperdicia acceso a computo.

Una estrategia que logra mejorar estos puntos consta de hacer uso de las interrupciones del microcontrolador permitiendo atender a todos los *inputs* y eventos del mundo real. En el robot desmalezador [Pom+24] se utiliza esta estrategia en conjunto con una Un estilo arquitectonico para sistemas embebidos, por lo tanto existen 3 tareas principales:

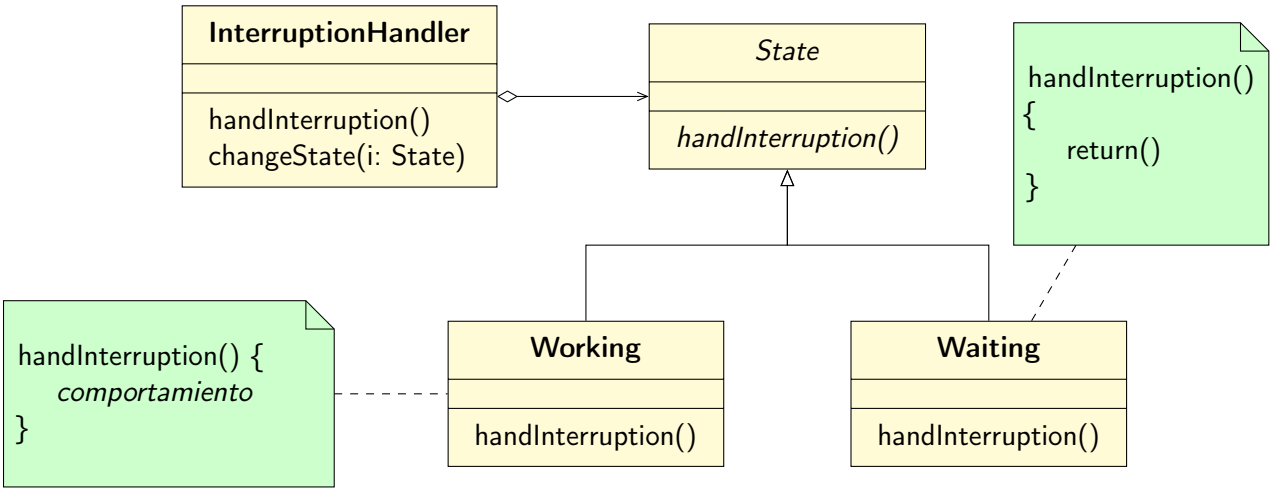
- Recibir información de los sensores y de la PC (por ejemplo, a traves del puerto serial).
- Procesar la información recibida y decidir que valores aplicar en los actuadores.
- Aplicar los nuevos valores a los actuadores. Notar que no es tan simple como configurar un número y dejar que actúe, en muchos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

De las tareas que se tienen, dos deben ser llamadas por el sistema y la otra (recibir información) en muchos casos se dará en forma de interrupciones que el sistema debe atender.

Ya se comentó de ese proceso en la sección Obtención de información, solo es importante recordar que la estructura propuesta provee una interfaz la cual se puede invocar para obtener la información recibida de manera simple. Una vez resuelta esa cuestión ahora se debe 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 esa interrupción, se accederá a la interfaces propuestas para obtener información de los sensores. Y por último, para los actuadores que necesitan un seguimiento temporal para su control, se agrega 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 propone aplicar 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. Un ejemplo de la estructura de módulos subyacente a esta solución es la que se puede ver en la figura 6.38, seguida de la documentación del uso del patrón *State*.

Figura 6.38: Ejemplo de handler usando *State* con su correspondiente documentación.



PatternApp	**Control de manejo de interrupciones**
based on	State (Estado)
why	**Cambios previstos**: Agregar, quitar o editar estados de manejo de la interrupción. **Funcionalidad**: Configura el estado actual de manejo de una interrupción a fin de que una interrupción corte la ejecución de otra.
where	InterruptionHandler **is** Contexto State **is** Estado Working **is** EstadoConcreto Waiting **is** EstadoConcreto

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.

Al aplicar el suo de interrupciones junto con patrones de diseño como State, el sistema embebido de control robótico adquiere una estructura más robusta, eficiente y mantenible. Se logran superar las limitaciones del enfoque secuencial simple, evitando la pérdida de información crítica proveniente de sensores y eliminando los tiempos de espera bloqueantes. Además, la modularización del comportamiento en tareas bien definidas, combinada con la sincronización mediante interrupciones periódicas y específicas, permite un control más preciso y en tiempo real de los actuadores. Este enfoque no solo mejora la capacidad de respuesta del sistema, sino que también facilita la extensión y modificación del software frente a nuevos requerimientos, en línea con los principios del diseño orientado al cambio.

Apéndice A

Patrones de diseño de Gamma

En este apéndice se resumen los patrones de diseño de Gamma [Gam+95] utilizados en las soluciones a los problemas comunes. En el libro se encuentra una descripción completa de los mismos. Aquí solo se menciona la intención, aplicabilidad, participantes y estructura de cada patrón. El propósito es utilizar este apéndice a manera de complemento al entendimiento de cada aplicación de patrón.

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.

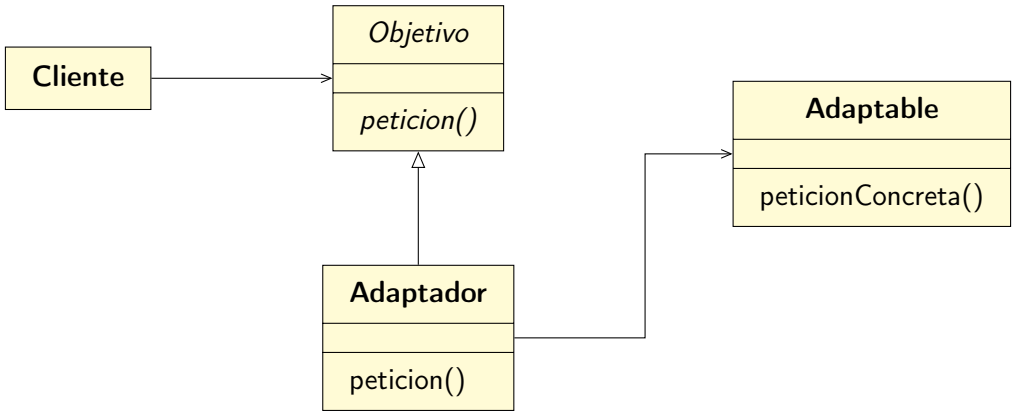
Participantes

- **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 ("undo"). 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.

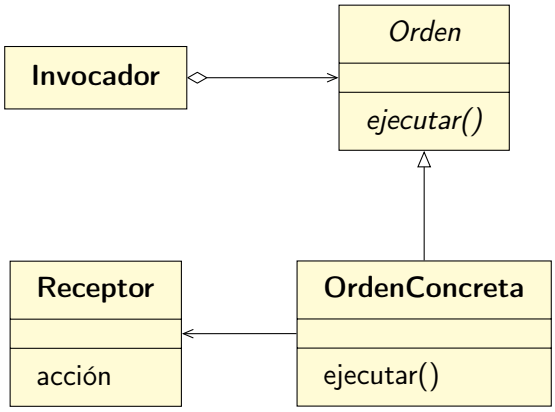
Estructura

A.3 State

Intención

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

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



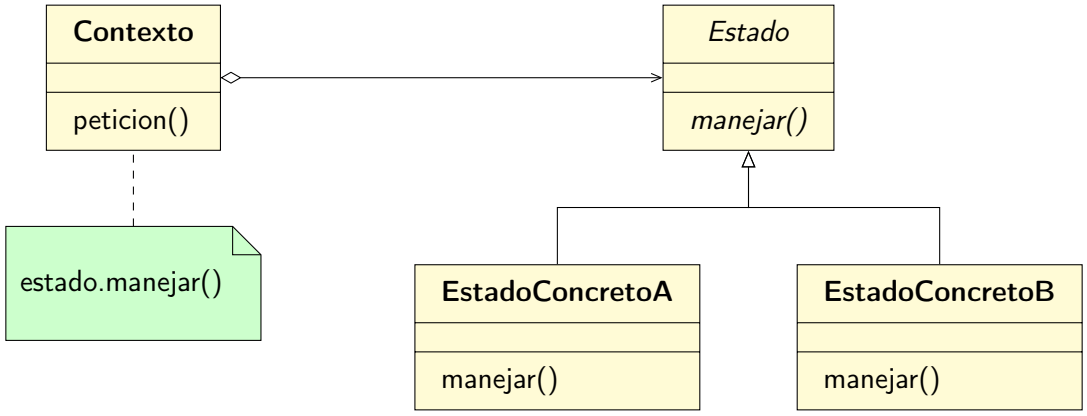
Aplicabilidad

- 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.

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



Estructura

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

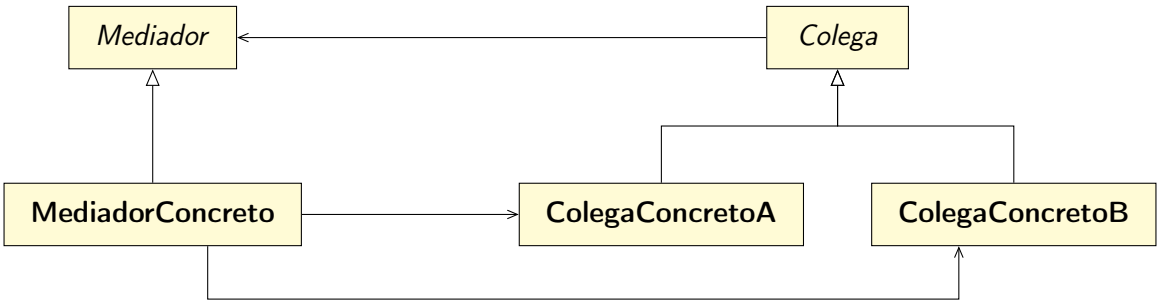
- Un conjunto de módulos se comunica de maneras bien definidas pero complejas. Las interdependencias resultantes son desestructuradas y difíciles de comprender.
- Reutilizar un módulo resulta complicado porque este se refiere y se comunica con muchos otros módulos.
- Un comportamiento distribuido entre 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

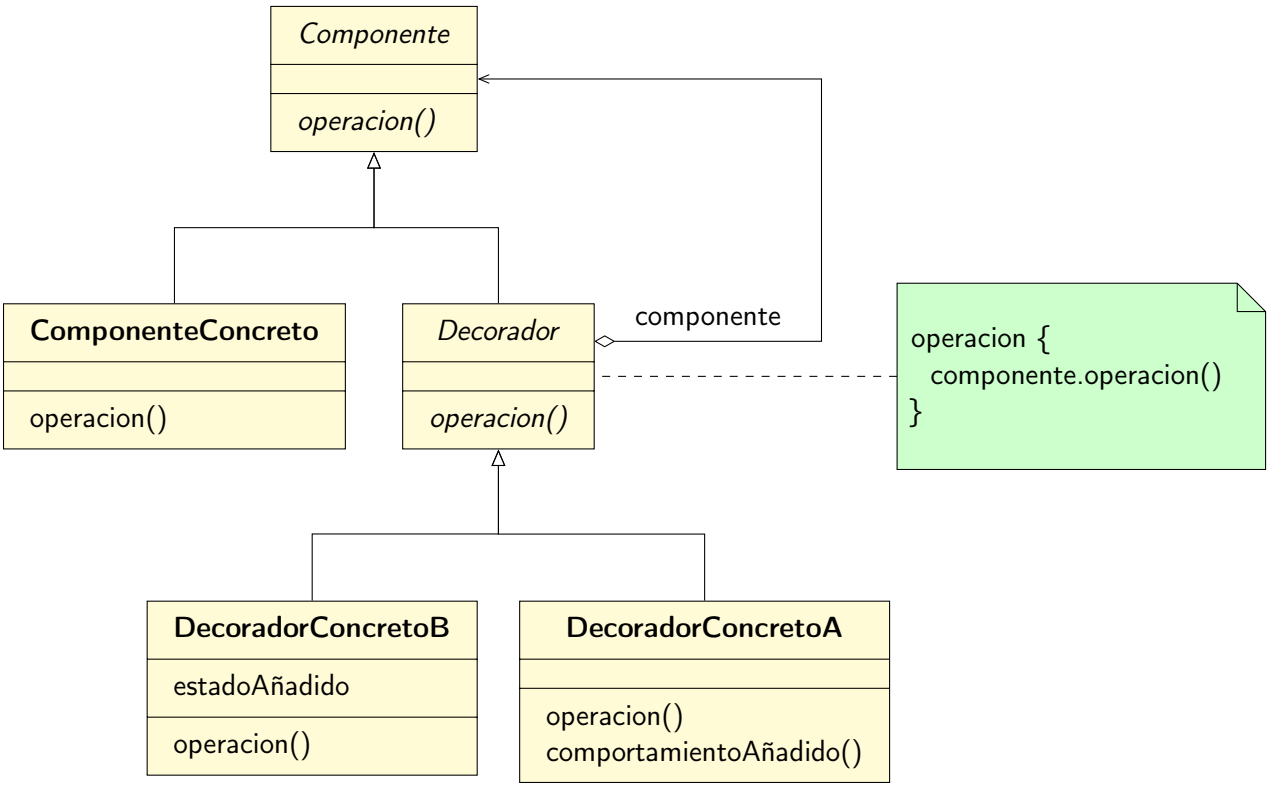
- Agregar responsabilidades a objetos individuales de manera dinámica y transparente, es decir, sin afectar a otros objetos.
- Para responsabilidades que pueden ser eliminadas.
- Cuando la extensión mediante herencia es impracticable. A veces, es posible tener una gran cantidad de extensiones independientes, 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

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



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:

   - 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.

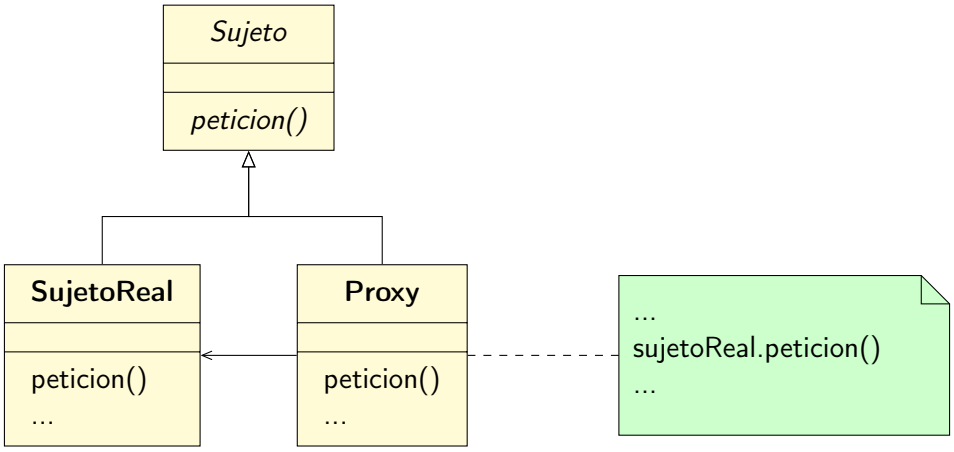
Estructura

A.7 Iterator

Intención

Proporcionar un modo de acceder secuencialmente a elementos de un módulo sin exponer su representación interna.

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



Aplicabilidad

Úsese el patrón Iterador:

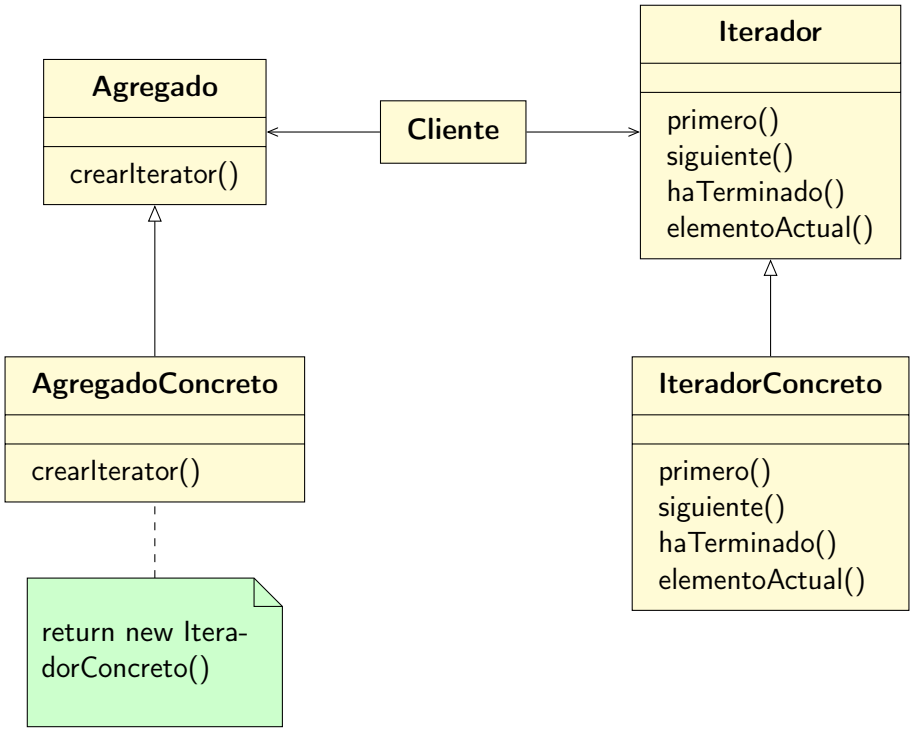
- Para acceder al contenido de un módulo agregado sin exponer su representación interna.
- Para permitir varios recorridos sobre objetos agregados.
- Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la iteración polimórfica).

Participantes

- **Iterador** Define una interfaz para recorrer los elementos y acceder a ellos.
- **IteradorConcreto** Implementa la interfaz de **Iterator**.
- **Agregado** Define una interfaz para crear una instancia de **Iterator**.
- **AgregadoConcreto** Implementa la interaz de Agregado.

Estructura

Figura A.7: Estructura patrón **Iterador**



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. 44, 76

Arduino Arduino es una compañía de desarrollo de software y hardware libres, así como una comunidad internacional que diseña y manufactura placas de desarrollo de hardware para construir dispositivos digitales y dispositivos interactivos que puedan detectar y controlar objetos del mundo real. 42

actuadores Dispositivos capaces de transformar energía hidráulica, neumática o eléctrica en la activación de un proceso con la finalidad de generar un efecto sobre un proceso automatizado. 52

ADC El ADC (acrónimo de convertidor analógico digital) es un dispositivo electrónico que convierte una señal analógica en una señal digital. 23

Bit flip Un "bit flip" (o inversión de bits) en informática se refiere a un error donde el valor de un bit (un dígito binario, 0 o 1) cambia a su opuesto. En otras palabras, un 0 se convierte en 1, o viceversa. Este cambio puede ocurrir debido a diversos factores, como errores de hardware (memoria defectuosa, sobrecalentamiento) o errores de software.. 85

callback Es una función A que se usa como argumento de otra función B. De esta forma, al llamar a B, esta ejecutará A.. 96

clock El clock de la CPU, también conocido como velocidad de reloj o velocidad del procesador, es la cantidad de ciclos que la unidad central de procesamiento puede ejecutar por segundo. Se mide en hercios (Hz) y es un factor fundamental en el rendimiento de una computadora. 23

CPU Unidad central de procesamiento (conocida por las siglas CPU, del inglés Central Processing Unit) o procesador es un componente del hardware dentro de un ordenador, teléfonos inteligentes, y otros dispositivos programables (como los embebidos). 36

DBOI Diseño basado en ocultación de la información. 29, 45

DC Corriente continua. 42, 45

DRV8838 Controlador para un único motor de corriente continua (DC) con escobillas. 42

enjambres robóticos Enfoque de diseño suele implicar una relación "micro-macro", donde los comportamientos individuales de los robots (micro) contribuyen a un comportamiento colectivo emergente del enjambre (macro). Los robots intercambian información para lograr tareas colectivas, como la búsqueda y recolección de recursos, imitando comportamientos inspirados en la naturaleza (como abejas o hormigas). 17

framework Entorno de trabajo, estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto. 18

GPIO GPIO (General Purpose Input/Output, Entrada/Salida de Propósito General) es un pin genérico en un chip, cuyo comportamiento (incluyendo si es un pin de entrada o salida) se puede controlar (programar) por el usuario en tiempo de ejecución. 23

Hall Sensor que se sirve del efecto Hall para la medición de campos magnéticos o corrientes o para la determinación de la posición en la que está. En particular se puede utilizar para determinar la velocidad de giro de una rueda.. 52

IS Ingeniería de Software. 11–13, 15, 19

MCU Unidad de Microcontrolador. 11, 12, 23

microcontrolador Circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria . 42

PDAs PDA (del inglés Personal Digital Assistant, Asistente Digital Personal), computadora de bolsillo, organizador personal o agenda electrónica de bolsillo es una microcomputadora de mano originalmente diseñada como agenda personal electrónica (para tener uso de calendario, lista de contactos, bloc de notas, recordatorios, dibujar, etc.) con un sistema de reconocimiento de escritura. 21, 22

PID Los controladores PID (Proporcional-Integral-Derivativo) son sistemas de control ampliamente utilizados en automatización y robótica para regular variables como velocidad, temperatura o posición. Funcionan calculando una señal de control basada en tres términos: Proporcional (P): Genera una respuesta proporcional al error actual (diferencia entre el valor deseado y el valor medido). Integral (I): Suma los errores pasados para corregir desviaciones acumuladas y eliminar el error en estado estacionario. Derivativo (D): Predice la tendencia del error, proporcionando una respuesta anticipada para evitar sobrepasos y mejorar la estabilidad. 26, 56

PWM PWM significa modulación por ancho de pulso, una técnica que cambia el ciclo de trabajo de un pulso digital periódico. PWM se utiliza comúnmente para convertir un valor digital en voltaje analógico mediante el envío de pulsos con un ciclo de trabajo que da como resultado el voltaje analógico deseado. 23, 53

ROS Sistema Operativo Robótico (en inglés Robot Operating System, ROS) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. 19

RPM revoluciones por minuto. 36

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.

[Whi11a] E. White. *Making Embedded Systems: Design Patterns for Great Software*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[SG96a] 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.

[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.

[Whi11b] E. White. "Making Embedded Systems - Design Patterns for Great Software". En: (2011).

[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.

[SG96b] M. Shaw y D. Garlan. "Software Architecture Perspectives on an Emerging Discipline". En: (1996).

[Bus+94] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal. "Pattern-Oriented Software Architecture Volume 1: A System of Patterns". En: (1994).

[GJM03a] C. Ghezzi, M. Jazayeri y D. Mandrioli. *Fundamentals of software engineering (2nd. Edition)*. Prentice Hall, 2003.

[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.

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

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

[Brä03] T. Bräunl. *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. 2003.

[BP09] D. Brugali y E. Prassler. "Software engineering for robotics [From the Guest Editors]". En: *Robotics & Automation Magazine, IEEE* 16 (abr. de 2009), págs. 9-15. DOI: 10.1109/MRA.2009.932127.

[CY12] F. Chang-sheng y G. Yan-ling. "Design of the Auto Electric Power Steering System Controller". En: (2012).

[Zha+16] Y. Zhang, X. Peng, L. Hou, lun Zhao, G. Sun y R. Feng. "A Design of Hand-held Remote Controller for An Implantable Stimulator Based on 51 MCU and WiFi". En: (2016).

[API22] D. API. "Dorna 2 Python API". En: (2022). URL: https://github.com/dorna-robotics/dorna2-python/blob/master/dorna2/dorna.py#L93.

[ERD20] ERDOS. "ERDOS". En: (2020). URL: https://github.com/erdos-project/erdos/blob/master/python/erdos/timestamp.py#L34.

[ZZ09] J. Zhang y M. Zhang. "Research and Design of Embedded Tank Car Monitoring System Based on ARM9". En: *2009 Second International Symposium on Computational Intelligence and Design*. Vol. 2. 2009, págs. 292-295. DOI: 10.1109/ISCID.2009.219.

[Mas12] T. M. T. Masayoshi Tamura Tatsuya Kamiyama. "A Model Transformation Environment for Embedded Control Software Design with Simulink Models and UML Models". En: (2012).

[Mil19] S. Milan Tkáčik Adam Březina. "Design of a Prototype for a Modular Mobile Robotic Platform". En: (2019).

[Dur+15] H. Durmuş, E. O. Güneş, M. Kırcı y B. B. Üstündağ. "The Design of General Purpose Autonomous Agricultural Mobile-Robot: AGROBOT". En: (2015).

[BD09] C. R. Baker y J. M. Dolan. "Street Smarts for Boss". En: (2009).

[PCB18] L. Pitonakova, R. Crowder y S. Bullock. "Information Exchange Design Patterns for Robot Swarm Foraging and Their Application in Robot Control Algorithms". En: (2018).

[Car13] I. R. Carlos Hernández Julita Bermejo-Alonso. "Three Patterns for Autonomous Robot Control Architecting". En: (2013).

[BS06] D. Brugali y P. Salvaneschi. "Stable Aspects in Robot Software Development". En: (2006).

[Arm10] A. Armoush. "Design Patterns for Safety-Critical Embedded Systems". En: (2010).

[Gar+14] R. Garro, L. Ordinez, R. Santos y J. Orozco. "Estrategias de Diseño Basadas en Patrones de un Subsistema de Movimiento para un Robot Pulverizador." En: (2014).

[Bus+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. 1996.

[KSB16] D. Kortenkamp, R. Simmons y D. Brugali. “Robotic Systems Architecture and Programming”. En: (2016).

[FG03] J. Fernández-Madrigal y J. González. *Integrating Heterogeneous Robotic Software: The BABEL Development System*. Inf. téc. System Engineering y Automation Department, University of Málaga, 2003.

[BFS13] M. Bonfè, C. Fantuzzi y C. Secchi. “Design patterns for model-based automation software design and implementation”. En: (2013).

[KKL05] B. Kang, Y.-J. Kwon y R. Y. Lee. “A Design and Test Technique for Embedded Software”. En: (2005).

[Bye05] R. Byeongdo Kang Young-Jik Kwon. “A Design and Test Technique for Embedded Software”. En: (2005).

[Dom14] H. Dominick Vanthienen Markus Klotzbücher. “The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming”. En: (2014).

[Bru+07] D. Brugali, A. Agah, B. MacDonald, I. A. Nesnas y W. D. Smart. “Software Engineering for Experimental Robotics”. En: (2007).

[Qui+09] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler y A. Y. Ng. “ROS: an open-source Robot Operating System”. En: (2009).

[Shi+15] S. Y. Shin, Y. Brun, L. J. Osterweil, H. Balasubramanian y P. L. Henneman. “Resource Specification for Prototyping Human-Intensive Systems”. En: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2015.

[Mic03] S. Michael Montemerlo Nicholas Roy. “Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CAR- MEN) toolkit”. En: (2003).

[Pau06] Paul Newman. “Moos - mission orientated operating suite”. En: (2006).

[Jar07] Jared Jackson. “Microsoft robotics studio: A technical introduction”. En: (2007).

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

[Ard] Arduino. “Arduino Microcontrollers Family”. En: (). URL: https://www.arduino.cc/en/hardware#classic-family.

[Ras] Raspberry. “Raspberry Pi Microcontrollers Family”. En: (). URL: https://www.raspberrypi.com/products/.

[Neo14] NeoPixels. “Using NeoPixels and Servos Together”. En: (2014). URL: https://learn.adafruit.com/neopixels-and-servos.

[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.

[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.

[Cri22] M. Cristiá. *Diseño de Software*. Apunte de cátedra de Ingeniería de Software 2. 2022.

[Cle+10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord y J. Stafford. *Documenting Software Architectures Views and Beyond, 2nd Edition*. English. Boston: Addison-Wesley, 2010. ISBN: 0321552687 9780321552686.

[GJM03b] C. Ghezzi, M. Jazayeri y D. Mandrioli. *Fundamentals of software engineering (2nd ed.)* 2003.

[Cri15] M. Cristiá. *Estándar para documentar el uso de patrones de diseño en un diseño de software*. Apunte de cátedra de Ingeniería de Software 2. 2015.

[Pom24] L. Pomponio. *Estilo LATEX para Documentar Requerimientos y Diseño de Software*. 2024.

[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.

[GIM19] G. Giacomino, J. F. Imsand y S. Martinez. “Desarrollo e implementación de funciones de navegación autónoma para un prototipo de robot pulverizador”. En: (2019).

[BCD18] E. Bongiovanni, T. Costamagna y J. C. Dellarossa. “Desarrollo e Implementación de un Sistema de Tracción y Dirección en Prototipo de Robot Desmalezador”. En: (2018).

[Min22] N. Minorsky. *Directional Stability of Automatically Steered Bodies*. Journal of the American Society of Naval Engineers, 1922.

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

[Gla+] M. Glaß, C. Haubelt, M. Lukasiewycz y J. Teich. “Incorporating Graceful Degradation into Embedded System Design”. En: *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2009)*.