Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Patrones de diseño aplicados a sistemas embebidos de control

Alumno:
Petruskevicius Ignacio Tomás

Directora:
Dra. Pomponio Laura

Septiembre 2025

Patrones de diseño aplicados a sistemas embebidos de control

Ignacio Tomás 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][Whi11] 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 órdenes propuestas desde una PC o un control remoto.

Para diseñar este sistema se optó por emplear un estilo arquitectónico de control de procesos [SG96, 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 exhaustiva 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 [Whi11]. [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 propuestos 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 definición 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 usuales 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 manejar interrupciones del MCU y la verificación de precondiciones en métodos.

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 de los capítulos siguientes.

Por otro lado, en el Capítulo 5 (Soluciones útiles) se describen dos soluciones de diseño útiles, que serán aplicadas en las distintas resoluciones a los problemas comunes presentados

en el Capítulo 6.

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 último, 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 7 (Conclusión) presenta las conclusiones y trabajos futuros de esta tesina. Además, Apéndice A resume brevemente los patrones de [Gam+95] que son utilizados a lo largo de este trabajo.

Capítulo 2

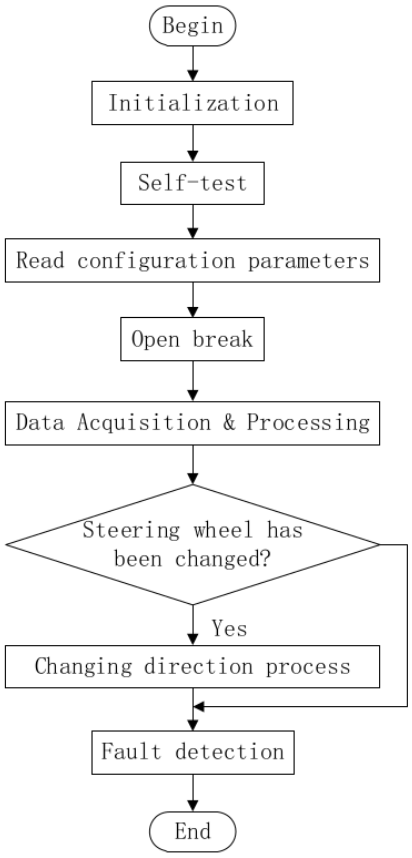
Estado del arte

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; SG96; Bus+94; GJM03; 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; Tam+12; TBJ19], 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:
            self._py_timestamp = PyTimestamp(coordinates, is_top, is_bottom)
        elif coordinates is not None and not is_bottom and not is_top:
            self._py_timestamp = PyTimestamp(coordinates, is_top, is_bottom)
        else:
            raise ValueError(
                "Timestamp should either have coordinates"
                "or be either Top or Bottom"
             )
```

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, no se evidencia la aplicación de patrones de diseño.

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 ciertas interacciones.

- [Her+13], 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 significativo.

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 específico. 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 veremos más adelante en la tesina en el libro de [Dou11], lo que allí se denominan patrones de diseño en realidad

corresponden a 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 implementación de soluciones, 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 los desarrolladores pueden empezar a trabajar. Son ampliamente utilizados en la industria y aplicados a grandes sistemas. No así para aplicaciones 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 propone incorporar formación relacionada con la IS en cursos de robótica y sistemas embebidos, otorgando igual énfasis tanto a la IS como a la Robótica. El objetivo del curso es enseñar a los estudiantes cómo aplicar los principios de la IS al campo de la Robótica. En particular, se dan dos lecturas principales que contienen los conceptos de patrones de diseño y arquitectura de software. En la primera 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

[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 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 [MRT03], 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 presentarán 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. [Whi11]

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

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

- Los sistemas embebidos son más limitados en funcionalidad de hardware y/o software que una computadora personal.
- Un sistema embebido está diseñado para realizar una función dedicada.
- Un sistema embebido es un sistema informático con requisitos de mayor calidad y fiabilidad que otros tipos de sistemas informáticos.
- Algunos dispositivos que se denominan sistemas embebidos, como los 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 destacar 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 voátiles.
- **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 integracón 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.

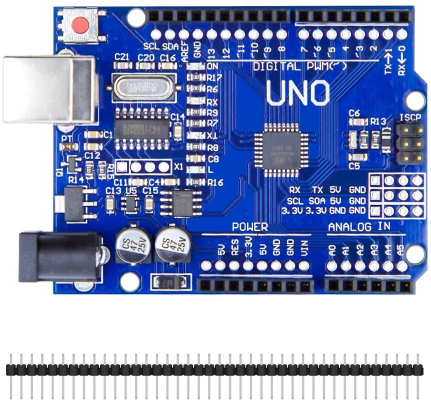


Figura 3.1: Arduino UNO

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 Interrupt Service Routine (ISR). 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`.

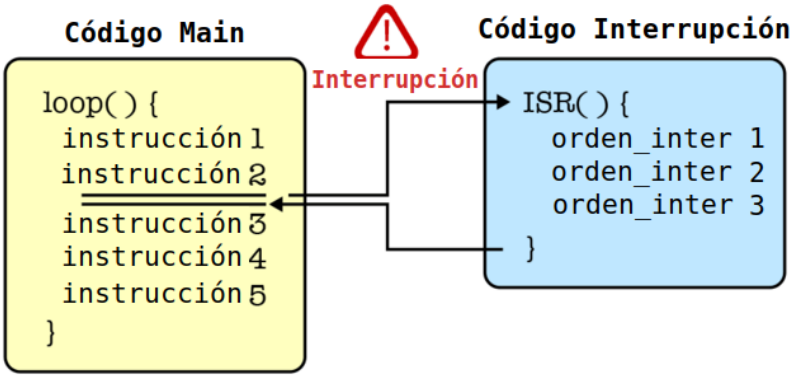


Figura 3.3: Diagrama del manejo de una interrupción extraído de [Neo14].

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 órdenes 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 componentes 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 alcanzables, se tiende a obtener los resultados deseados.

Los sistemas embebidos constituyen el núcleo esencial 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.[SG96; GJM03; 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; SG96; GJM03; BCK03; TMD10]. Estas pueden ser provocadas por 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 cuenta aquello que probablemente cambiará y se piensa cómo debe ser el diseño para que los cambios impliquen modificar porciones mínimas y aisladas del software [GJM03].

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 también 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 reemplazado y, por tanto, cambia su forma de reportar datos, el ajuste del software debería limitarse al módulo que oculta los detalles de implementación del uso de 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, 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 preveer cambios recurrentes en dominios específicos [Gam+95; BHS07]. Por ejemplo, en sistemas robóticos, un estilo arquitectónico basado en bucles de control puede destacar las características esenciales del sistema y facilitar decisiones de diseño óptimas [SG96].

Herencia de Interfaz

En el diseño modular, un módulo consta de una parte visible llamada *interfaz* y una oculta la llamada *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.

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 clientes (aquellos otros módulos que usan el que se está definiendo) dependan solo de las abstracciones y no directamente de las implementaciones

[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.

concretas. Esta separación permite modificar, reemplazar o extender funcionalidades sin afectar el resto del sistema, y facilita la aplicación del concepto de composición, como veremos a continuación.

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 un 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 manera que cuando el módulo principal 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 funcionalidades 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 ítems 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]

Extracto del apunte de clase [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 la Sección 4.3). Una vez

identificados, Parnas sugiere ocultar y aislar cada ítem 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 interno.

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 ítems con probabilidad de cambio presentes en los requerimientos.
2. Analizar las 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 ítems 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 más probables a cambiar que otros, por lo que resulta importante anticiparse a esos cambios en particular. Algunos autores mencionaron algunos ítems de cambio comunes entre múltiples sistemas [Par78; Cri22].

- Contracción o extensión de requisitos (agregar, quitar o editar funcionalidades).
- Configuraciones de hardware (cambio de MCU, plataforma, 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.

- 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 decisión, ordenamiento, etc.).
- 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 sociocultural (moneda, impuestos, fechas, idioma, etc.).
- Cambios propios del dominio de aplicación.
- Cambios propios del negocio de la compañía desarrolladora.
- Interconexión con otros sistemas (cambios en Interfaz de Programación de Aplicaciones (API), por ejemplo).

Es útil consultar estos ítems 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 subsección 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, sus interfaces y las relaciones entre ellos.

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

Se destaca que la arquitectura de software no solo se trata de la estructura del código o la implementación técnica, sino de las decisiones de alto nivel que afectan la organización y el comportamiento del sistema. Estas decisiones incluyen cómo dividir un sistema en partes modulares, qué patrones 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 esto sería deseable. 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 consiste en avanzar en el desarrollo estructurando y redactando 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, como la guía de módulos, las especificaciones de interfaces y los requisitos formales, que se describirán más adelante en esta sección, como herramientas fundamentales para organizar, analizar y comunicar las decisiones de diseño. De esta manera, es posible identificar inconsistencias, delegar responsabilidades de manera clara entre los módulos y facilitar tanto el mantenimiento como la incorporación de nuevos desarrolladores al proyecto. Según los autores, diseñar a través de la documentación no solo clarifica el proceso, sino que también mejora la calidad y la 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. Los módulos son una representación cuyo correlato y consecuencia el es código. Cada módulo tiene asignada y es responsable de llevar adelante una función. Esto debe ser descripto por medio de los siguientes documentos: 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**[Cri22; Pom25] basado en TDN presentado en [GJM03]. 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.

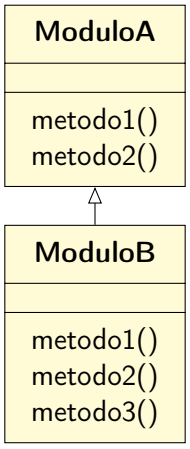
Y un heredero de ModuloA llamado ModuloB se documenta como en la Figura 4.2.

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.

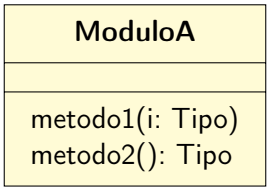
Figura 4.3: Documentación gráfica de un módulo y su heredero.



Además, es posible indicar el tipo de los parámetros de entrada de un método así como el tipo del valor de retorno del mismo, como presenta el diagrama de Figura 4.4. Asumimos que un módulo es, genera o define un tipo. Es decir que para dotar de tipos nuestros métodos usamos módulos.

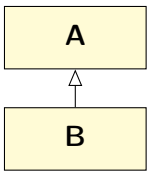
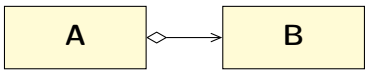
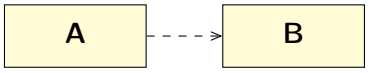
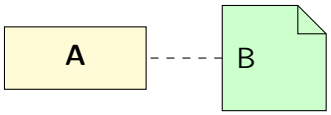
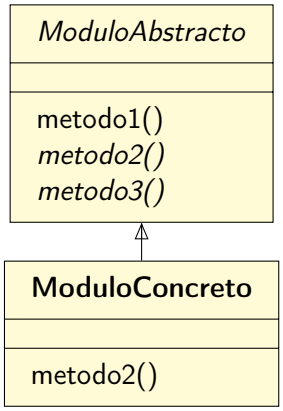
Este formato gráfico de documentación puede ser extendido haciendo uso de los símbolos definidos en la Figura 4.5.

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 lenguaje **2MIL** de la manera indicada en la Figura 4.6. En este formato se establece el patrón que se está aplicando y se da una breve descripción

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().

del caso de aplicación. Además, se enumeran los cambios previstos y cómo 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 de la aplicación del patrón.

PatternApp	**Breve descripción**
based on	Patrón (Pattern)
why	**Cambios previstos**: Descripción de los cambios previstos relacionados con el 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 representación gráfica para documentar módulos y sus relaciones. Para documentar patrones sí se utilizará **2MIL**. Información adicional sobre cómo documentar utilizando este lenguaje puede encontrarse en [Cri22; Cri15; Pom24; Pom25].

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 arquitectónicos de software orientados al control de procesos; por ejemplo el estilo arquitectónico de *Control de Procesos* presentado en [SG96]. 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 qué 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.

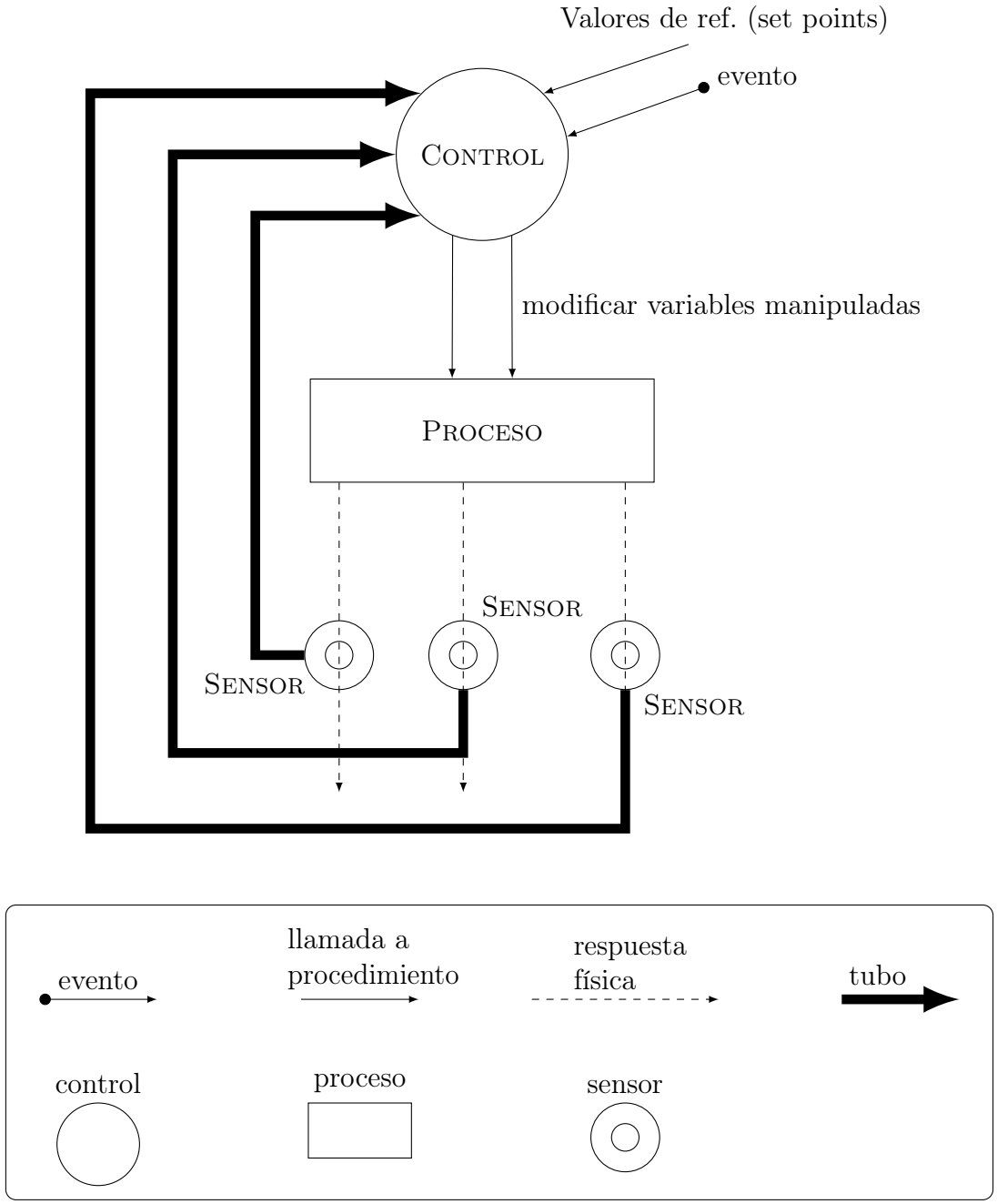
De esta manera se logra alcanzar, si es posible, a los valores deseados luego de una serie de iteraciones.

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 establecido 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 arquitectónico 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 componentes básicos **Control**, **Proceso** y **Sensores**. Cada uno de ellos trabaja de manera independiente y se comunican mediante dos mecanismos principales, llamadas a procedimientos 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** (*pipes*). 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 arquitectónico 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 modificados o reemplazados 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

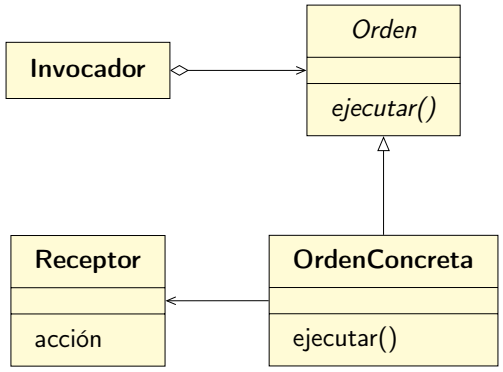
Parnas en [Par72] establece que desacoplar se refiere a la idea de reducir la dependencia entre módulos en un sistema de software. Un sistema está acoplado cuando los cambios en un módulo requieren modificaciones en otros módulos. Desacoplar significa diseñar los módulos de manera que puedan funcionar 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* A.2. Este patrón es el que más veces se aplicó en el trabajo del robot desmalezador [Pom+24], 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 sí 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 del patrón *Command*



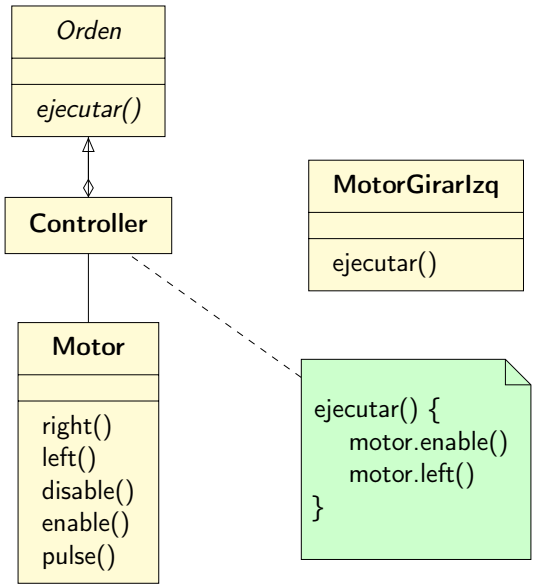
- Invocador: le pide a la *Orden* que ejecute la acción.
- *Orden*: declara la interfaz para ejecutar una acción.
- OrdenConcreta: implementa el método ejecución el cual se encarga de llamar a el o los métodos del Receptor con el objetivo de llevar a cabo la acción.
- Receptor: cualquier módulo sobre 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. Además, 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 un motor paso a paso**
based on	Orden (Command)
why	**Cambios previstos**: Se pueden agregar o modificar órdenes. **Funcionalidad**: Desacoplar al que invoca una orden de que la lleva a cabo, proporcionando flexibilidad ante cambios en las órdenes o el receptor.
where	Orden **is** Orden MotorIniciarGiroIzq **is** OrdenConcreta Motor **is** Receptor Controller **is** Invocador

Un módulo Controller, necesita manejar un motor paso a paso representado 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` y luego configurar el sentido de giro con el método `left` o `right` (izquierda o derecha). Tradicionalmente, esto sería realizado desde el módulo Controller agregando el código en el método `control()` 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*.

```
Controller::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, . 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 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 . 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.

Como se mencionó, 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 empleo 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. Esto 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 misma en la Figura 6.1.

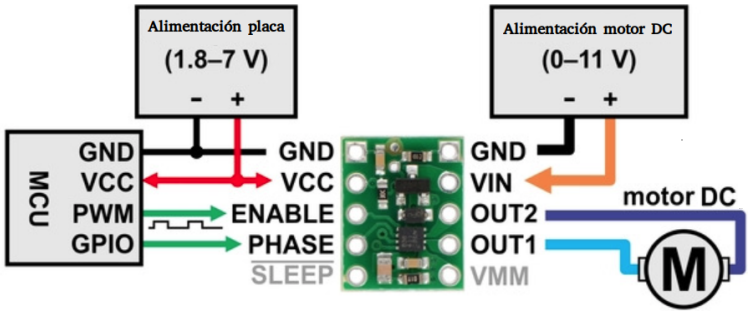


Figura 6.1: Conexionado de la placa de control DRV8838.

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.

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

Cuadro 6.1: Funciones de cada pin del módulo DRV8838

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 cumplirlos solo necesitaremos trabajar con los pines *ENABLE* y *PHASE*. Tendremos entonces dos cables conectados desde el microcontrolador a la placa DRV8838, uno que se dirige a *ENABLE* y otro a *PHASE*. En el pseudocódigo 6.1 se encuentran las líneas necesarias para 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 al 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 bastantes descriptivos de su comportamiento, `pinMode` configura el modo de operación 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 los Códigos 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 líneas 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)
            analogWrite(VEL_pin2, 0)
        .
        .
        .
```

La modificación consiste en verificar qué motor es el que se quiere manipular y basándose en eso enviar la señal a través de los pines correspondientes. Pero, ¿qué pasaría si se quiere añadir un tercer motor? Deberíamos agregar aún más sentencias `if else`, extendiendo aún más el código. ¿Y si debemos realizar operaciones diferentes según cada motor? Por ejemplo, en el caso de 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 línea 8. 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 su comportamiento al utilizar el motor 2.

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

- 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 antihorario. 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 líneas 4 y 7. 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, esta placa no utiliza la misma interfaz de control, sino para controlarla se accede a ella mediante comunicación serial (utiliza un solo pin específico). Incluso utilizando las herramientas provistas por el entorno de **Arduino Uno**, el nuevo código de configuración (ver Código 6.7) y uso (ver Códigos 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 reverificar código que ya se sabía que funcionaba correctamente. Se debe modificar las líneas 4, 5 y 14 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)
            .
            .
            .
    else
        if (motor = IDMotor1)
            if (TIPO_MOTOR1 = DVR8838):
                analogWrite(VEL_pin1, 255)
            else if (TIPO_MOTOR1 = Pololu):
                Serial.write(0xAA)
                Serial.write(0x0C)
                Serial.write(0xE0)
        else if (motor = IDMotor2)
            analogWrite(VEL_pin2, 0)
            .
            .
            .
```

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

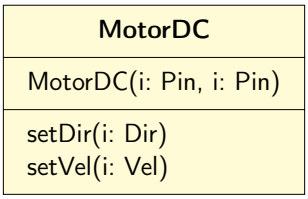
- Claramente, el código obtenido es poco claro; es decir, no resulta fácil comprender de qué se trata una determinada porción de código con solo leerla. Basta con intentar entender la función del Código 6.11 para comprobarlo. A su vez, el código resulta difícil de modificar, ya que requiere un esfuerzo 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 órdenes para definir su sentido y velocidad de rotación. La Figura 6.2 presenta la interfaz del módulo MotorDC que encapsula el hardware relacionado con el motor.

Figura 6.2: Interfaz MotorDC



Si se cuenta con dos o más motores del mismo tipo, se crearán dos a más instancias del módulo. Para esto el constructor 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 DRV8838.

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 setDir(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 teníamos que escribir las líneas 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.

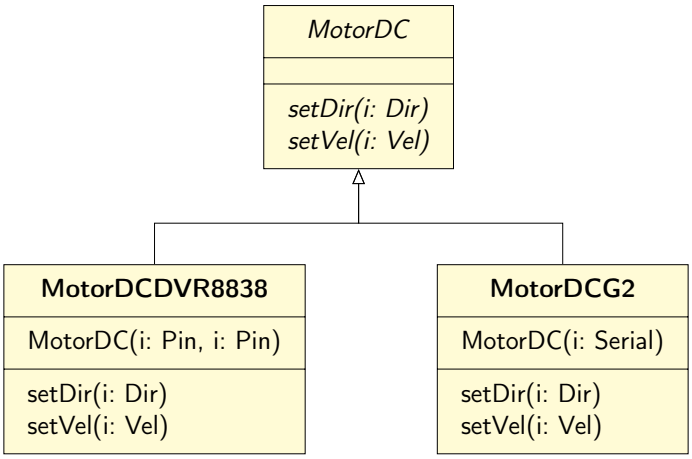
Código 6.15: Implementación método setVel para el motor que utiliza comunicación serie.

```
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. La 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 cuál de los dos tipos de placa controladora está tratando. También es posible agregar más herederos, uno por cada modelo de placa controladora/motor, y reutilizar los módulos implementados en caso de utilizar hardware idéntico.

Siguiendo con el ejemplo de la función `controlar_motor` (Código 6.11) veamos en el Código 6.16 cómo sería una nueva implementación de esta misma utilizando la encapsulación del hardware propuesta. Con esta solución el código de la función 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.

```
1 def controlar_motor(motor: MotorDC)
2     if (valor > 100)
3         motor.setDir(HORARIO)
4         motor.setDir(VelMaxima)
5     else
6         motor.setDir(VelNula)
```

De esta manera se siguen las prácticas recomendadas en la IS [SG96; GJM03; 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 librerías para su control, otras veces se consiguen en internet o se extraen de previos proyectos. Esto permite ahorrar tiempo de implementación, pero puede causar algunos inconvenientes si no se tiene en cuenta al cambio.

Tradicionalmente, se toman las librerías necesarias para controlar el hardware y se las utiliza directamente a lo largo del sistema. Lo que provoca que ante un cambio de hardware se debe modificar todos los usos de la librería con el nuevo código. Veamos un ejemplo en donde se evidenciará el inconveniente y cómo podemos anticiparnos al mismo.

Suponga el siguiente ejemplo: en un cierto sistema embebido se utiliza un display de 7 segmentos que forman un 8. Encendiendo o apagando independientemente cada segmento, se pueden mostrar distintos caracteres. Este display es de 4 dígitos y se emplea para mostrar la temperatura de funcionamiento y la posición, en grados, de cierto actuador, como se muestra en 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, el fabricante brinda una librería llamada LibAcme que implementa la comunicación y provee funciones simples de usar, tales como `escribir(i: string): 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 de la manera expuesta en el Código 6.17.

Código 6.17: Ejemplo de uso de la libreria LibAcme.

```
libAcme = LibAcme()

if libAcme.escribir("24")
    print("El display estaba vacio, se pudo escribir el nuevo texto")
else
    print("El display esta ocupado mostrando algo, no se pudo escribir")
```

En cierto momento, el módulo display dejó de funcionar y fue reemplazado por otro de un fabricante distinto, el cual utiliza un protocolo de comunicación diferente. Al igual que en el caso anterior, la empresa provee una librería LibEmca para utilizar el nuevo display.

Sin embargo, la interfaz no es la misma que la anterior, e incluso algunos comportamientos también difieren.

Por ejemplo, en la primera librería, el método de escritura devolvía `False` si se intentaba escribir mientras el display ya estaba mostrando información. En cambio, en la nueva implementación, si el display está mostrando algo, el nuevo texto lo sobrescribe de inmediato. No obstante, esta nueva librería proporciona un método adicional `get_current(): string` que permite consultar el contenido que se está mostrando en el momento en que es invocado.

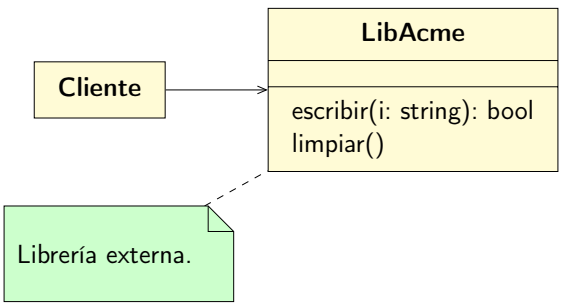
Tradicionalmente, para utilizar el nuevo display, se modifican todas las llamadas a las funciones de la librería anterior en todo el sistema, incorporando la nueva lógica. En nuestro ejemplo, el Código 6.17 debe ser reemplazado por el Código 6.18. Si bien el cambio puede parecer superficial, es importante considerar que con cada modificación será necesario actualizar manualmente y reverificar **todos** los usos de la librería a lo largo del sistema.

Código 6.18: Ejemplo de modificaciones necesarias para adaptar la nueva librería.

```
libEmca = LibEmca()

// Con EMCA

if libEmca.get_current() == ""
    libEmca.imprimir("24")
    print("El display estaba vacio, se pudo escribir el nuevo texto")
else
    print("El display esta ocupado mostrando algo, no se pudo escribir")
```

Además, este es un ejemplo simple; en el mundo real, los cambios pueden ser mucho más complejos en su lógica y de distinta naturaleza. Este diseño puede representarse como se muestra en la Figura 6.5: un módulo que constituye la librería del fabricante y otro que actúa como cliente de esta, es decir, el resto del sistema.

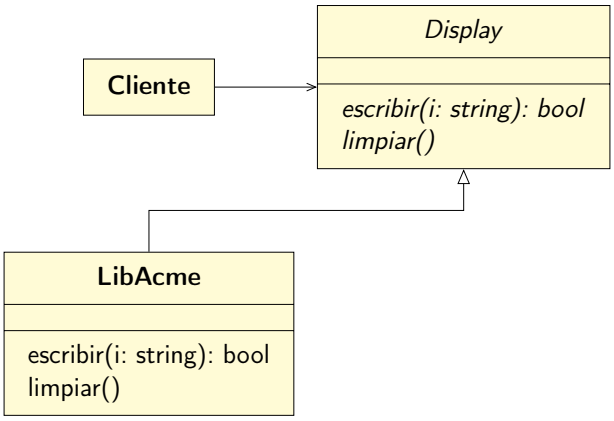
Figura 6.5: Diseño tradicional del uso de la librería del fabricante.



Este diseño presenta un fuerte acoplamiento entre el cliente y la LibAcme, por lo que cualquier cambio en el hardware repercutirá en el resto del sistema, trayendo consigo todas las desventajas mencionadas tanto en este capítulo como en el anterior. Se podría decir que el principal problema radica en que no se siguió la metodología de diseño propuesta por Parnas [Par72]. El hardware representa un ítem con alta probabilidad de cambio, por lo tanto, debe ser encapsulado.

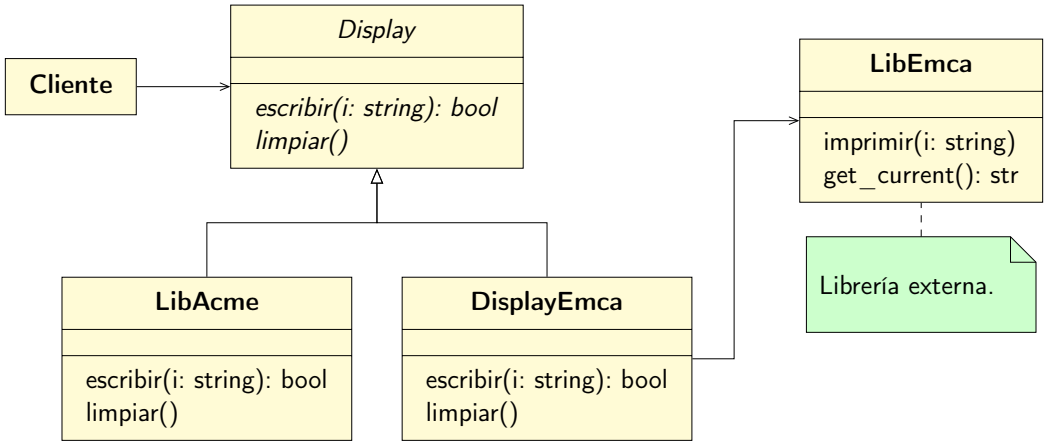
En el libro de Douglass [Dou11] se observa que, en la práctica, este tipo de escenarios son comunes: ya se tiene un sistema funcionando y se produce un cambio de hardware. Por ello, se intentará proponer una posible solución desde el punto de vista de la IS. En lugar de reescribir todo el sistema, veremos cómo, aplicando un patrón de diseño de Gamma [Gam+95], es posible adaptar el sistema y preparar el diseño del mismo para futuros cambios de índole similar. Para encapsular el hardware sin modificar todo el sistema se creará un módulo abstracto llamado *Display*. En su interfaz proveerá los mismos métodos que LibAcme, de esta manera evitaremos cambiar la implementación del resto del sistema. Al compartir interfaz podemos decir que LibAcme hereda la interfaz de *Display*, esto lo podemos ver en la Figura 6.6.

Figura 6.6: Introducción del nuevo módulo abstracto *Display*.



Como mencionamos, LibEmca se utiliza para controlar el nuevo display, pero su interfaz y funcionalidad no son las mismas que las de LibAcme. Por lo tanto, aplicaremos el patrón *Adapter* (Adaptador) propuesto por Gamma [Gam+95]. Este patrón consiste, en este caso, en crear un módulo DisplayEmca, que se encarga de adaptar la interfaz de LibEmca a la de *Display*. Es decir, utilizará los métodos de LibEmca para implementar la funcionalidad esperada por los métodos de *Display*. De esta manera, se obtiene la estructura de módulos mostrada en la Figura 6.7.

Figura 6.7: Diseño aplicando el patrón *Adapter*.



No olvidar que cada aplicación de un patrón de diseño debe ser seguida por su correspondiente documentación **2MIL**. En este caso se la presenta en la Figura 6.8.

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 interfaz diferente a la utilizada en el sistema, se crea un módulo que la adapta para que corresponda a la usada.
where	Display **is** Target LibEmca **is** Adaptee DisplayEmca **is** Adapter

En el Código 6.19 se presenta un ejemplo de implementación del módulo DisplayEmca. En este se utilizan los métodos provistos por la librería del fabricante, y se emula la interfaz de *Display*. De este modo, se permite reemplazar el hardware sin necesidad de modificar ni reverificar ninguna otra parte del sistema.

Código 6.19: Ejemplo de implementación del módulo

```
escribir(string palabra)
    if (libEmca.get_current() != "")
        return false
    else
        libEmca.imprimir(cadena)
        return true

limpiar()
    libEmca.imprimir("")
```

Las ventajas de aplicar este patrón en esta situación particular en la cual se parte de un diseño **no** orientado al cambio son:

- 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.
- Mantener la coherencia en la interfaz: el cliente sigue interactuando con la misma interfaz abstracta (*Display*), evitando la necesidad de modificar múltiples módulos en el sistema.
- Minimizar el riesgo de errores: al encapsular las diferencias de implementación en el adaptador (DisplayEmca), se reduce la posibilidad de introducir errores al modificar manualmente todas las llamadas en el código.
- Mejorar la mantenibilidad: cualquier nuevo display con una librería de control 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.

6.3 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, establece en el pin en el que está conectado un 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 un 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 otros tipos de dispositivos como joysticks, botones, etc.

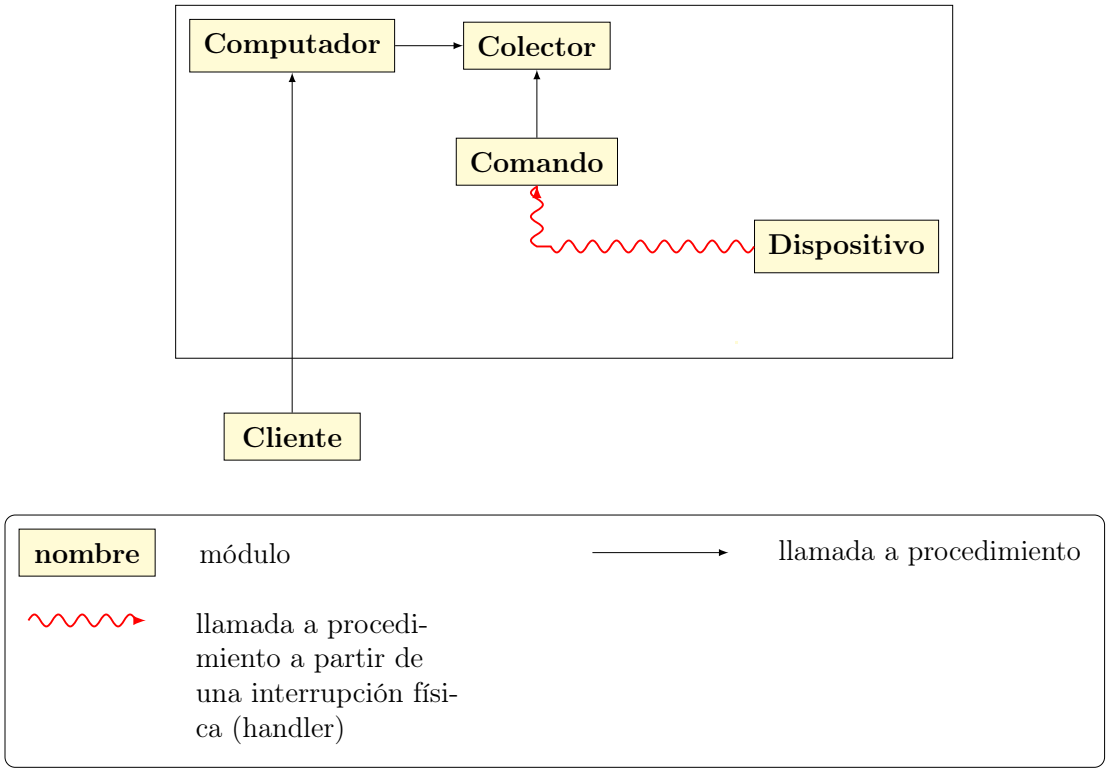
Tradicionalmente, esta tarea se centraliza en un único módulo que provee dos métodos: uno encargado de manejar la interrupción, procesar y almacenar la información, y otro que se

utiliza para acceder a ella. Al aplicar esta solución, se obtiene una estructura poco resiliente al cambio. Por ejemplo, si se modifican los cálculos para obtener el resultado deseado, es necesario modificar también la implementación del módulo, lo que da lugar a la posible introducción de errores. Asimismo, la manera de transmitir la información relacionada con la interrupción puede variar debido a modificaciones en el hardware (tipo o modelo del sensor, componentes mecánicos, etc.). En IS, decimos que el módulo oculta más de un ítem de cambio y, como vimos previamente, esto no se ajusta a las prácticas recomendadas.

Además, en muchos casos ni siquiera se construye este módulo, sino que cada cliente del dispositivo se encarga de capturar y procesar los datos provenientes, lo que provoca que, ante cualquier cambio, deba modificarse el código en diferentes módulos a lo largo del sistema.

Es debido a las problemáticas mencionadas que resulta útil contar con una forma general de abordar este problema desde el punto de vista del diseño orientado al cambio. Para ello, tomaremos como referencia el trabajo realizado sobre el robot desmalezador en [Pom+24], en el cual se empleó la estructura modular presentada en la Figura 6.9, destinada a llevar a cabo las actividades necesarias para el uso de sensores que generan interrupciones en el sistema. Replicaremos esta estructura, siguiéndola casi como si fuera un patrón de diseño.

Figura 6.9: Estructura componentes para la lectura de un sensor activo.

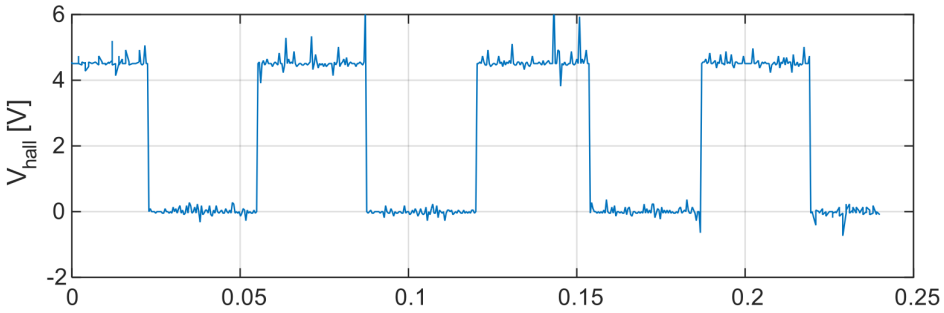


En esta estructura se distinguen cuatro módulos principales: Dispositivo, que encapsula el hardware asociado al dispositivo encargado de generar e introducir la interrupción. Este módulo cuenta con un manejador de interrupción configurado para invocar al comando encapsulado en el módulo Comando cuando la interrupción es lanzada. Este último sigue el patrón de diseño *Command* y ejecuta los métodos del módulo Colector para registrar la

información requerida sobre la interrupción (por ejemplo, el momento exacto en que ocurrió el evento o la cantidad de veces que sucedió). De este modo, podemos deducir que Colector almacena la información "cruda" proveniente de la interrupción o del sensor. Finalmente, se encuentra el módulo Computador, encargado de procesar la información cuando esta es solicitada. Para ello, lee los valores almacenados en Colector y aplica cálculos o algoritmos que computan la información final aportada por el dispositivo. Un ejemplo de ello podría ser el cálculo de la velocidad de rotación de una rueda o la obtención de los inputs de un joystick.

A continuación se presenta un ejemplo de aplicación con el objetivo de analizar en mayor profundidad la estructura modular. Se considera un sensor de efecto Hall montado de manera tal que, al girar una rueda, imanes permanentes pasen cerca de este. De esta forma, el sensor puede utilizarse para medir la velocidad de rotación. En la Figura 6.10 se observa la variación de voltaje producida por el paso de los imanes frente al sensor. Cada vez que el dispositivo detecta un campo magnético, emite una interrupción que es gestionada por el módulo Comando, el cual actúa como un comando y, por lo tanto, conoce qué funciones ejecutar dentro del módulo Colector.

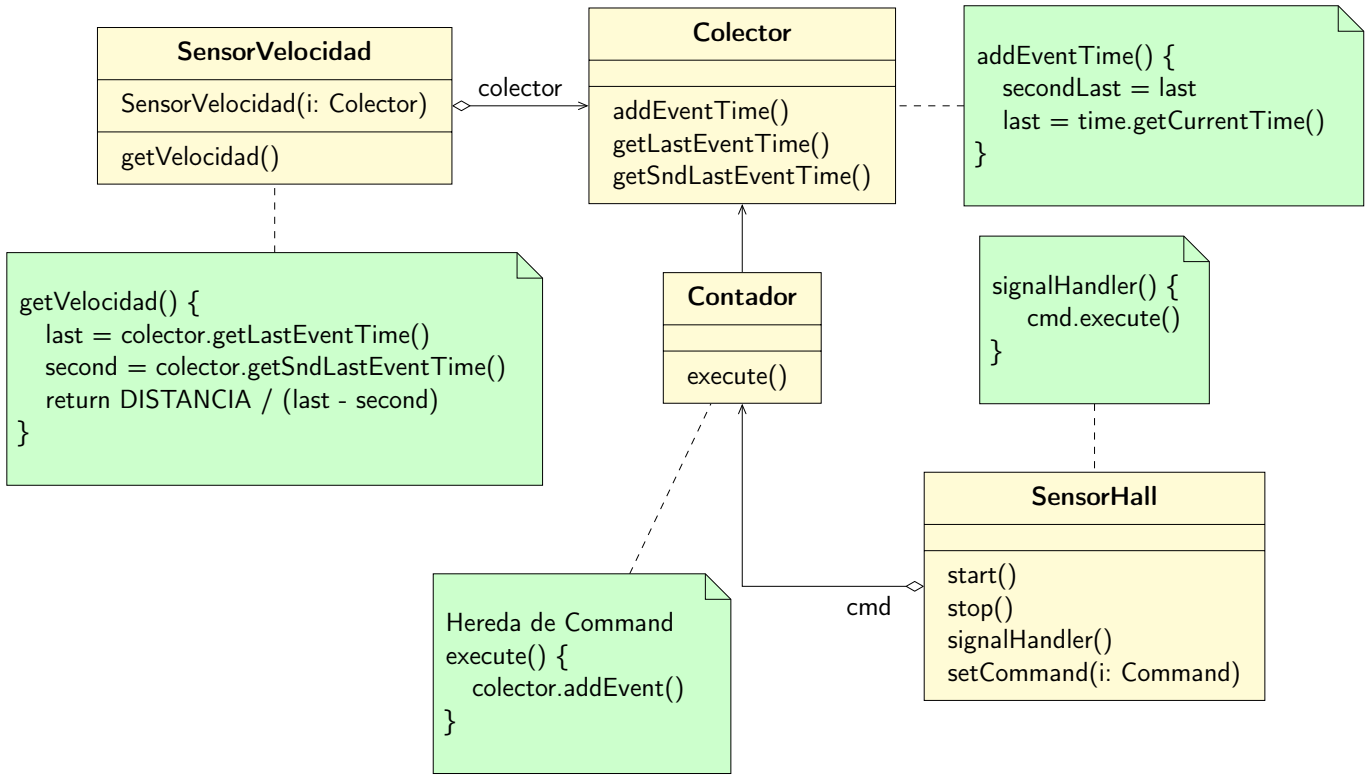
Figura 6.10: Ejemplo de la variación de voltaje producida por un sensor de efecto Hall al detectar un cambio del campo magnético. Cada pico de voltaje provoca que el microcontrolador genere una interrupción. Imagen extraída de [BCD18].



En la Figura 6.11 se presenta la estructura de módulos de esta solución, junto con pseudocódigo que ilustra una posible implementación.

Con los imanes colocados a una distancia conocida, para calcular la velocidad basta con registrar los dos últimos instantes de tiempo en que fueron detectados. De forma simplificada, la velocidad se obtendrá al dividir la distancia entre los imanes por el tiempo transcurrido entre ambas detecciones. En consecuencia, el módulo Colector debe proveer un método para almacenar el momento de cada detección. Posteriormente, el módulo SensorVelocidad, que cumple la función de Computador, tomará la información almacenada en Colector y realizará el cálculo matemático para determinar la velocidad de giro de la rueda. Los cálculos, que pueden ser complejos, se ejecutarán únicamente bajo demanda, es decir, cuando el cliente lo requiera. Esto permite optimizar el rendimiento del sistema al disminuir la carga computacional necesaria. Cabe aclarar que este ejemplo está simplificado y omite situaciones que añadirían complejidad innecesaria para mostrar la idea de la solución propuesta.

Figura 6.11: Ejemplo módulos para obtener la información referida a la velocidad.



Al aplicar esta solución orientada al cambio, preparamos esta sección del sistema para cambios probables, tanto de hardware como de algoritmos. Además, las responsabilidades se separan en módulos independientes, logrando que una modificación en uno de ellos no afecte al resto del sistema. De este modo, si la rueda fuera modificada y cambiara la distancia entre los imanes, bastaría con actualizar la implementación del módulo SensorVelocidad. A su vez, si el sensor Hall fuese reemplazado, solo se cambiaría el módulo que lo encapsula, manteniendo intacto el resto de la aplicación.

En esencia, al lograr este nivel de bajo acoplamiento, se minimiza el costo y el riesgo asociados al mantenimiento, ya que los cambios quedan confinados y son esperados. Esto no solo previene la propagación de errores, sino que genera que el sistema sea resiliente y flexible.

Recolectar información de manera periódica.

Algunos sistemas se encargan de mostrar, almacenar o verificar la información de los sensores a intervalos regulares. En estos casos no resulta crítico perder valores intermedios, es decir, no se requiere una respuesta inmediata. Un ejemplo de ello puede ser una estación meteorológica o un dispositivo médico de monitorización, como un tensiómetro que registra los valores de presión del paciente cada cierto período.

Una implementación intuitiva consiste en escribir un *loop* en el que se verifique la información de los sensores, llamando a métodos que la obtengan directamente del hardware,

para luego ejecutar una función *sleep* que bloquea el programa durante un tiempo determinado. Este enfoque presenta varias desventajas: en primer lugar, el tiempo de ejecución de la propia rutina alarga el período; en segundo lugar, si se desean agregar otras funcionalidades durante la espera, es necesario dividir la llamada a *sleep* y recalcular los tiempos de ejecución.

Una solución más adecuada desde el punto de vista del diseño orientado al cambio es la siguiente: se configura un temporizador para que, cada x cantidad de tiempo, genere una interrupción. Esta es una funcionalidad provista por muchos entornos de desarrollo para sistemas embebidos, como **Arduino Uno**. Para dicha interrupción, se registrará un manejador que consistirá en un módulo implementado según el patrón de diseño *Command* (ver Sección 5.2 para más información sobre este tipo de aplicación del patrón). Este encapsula las funciones que deben ejecutarse para llevar a cabo la funcionalidad deseada como, por ejemplo, solicitar la información a sus respectivos módulos Computador o a los sensores en sí, para luego registrarla en un cierto módulo que oculte la estructura de datos. De esta forma, se logra liberar al procesador durante los tiempos de espera y desacoplar esta funcionalidad.

6.4 Máquinas de estado

Muchos sistemas ajustan su comportamiento durante su ejecución en función de diversas causas, como interacciones con el entorno o requisitos internos. Un ejemplo sencillo es un sistema de control de microondas, que no iniciará el calentamiento si la puerta está abierta. En este caso, la condición de la puerta representa un estado interno que restringe un comportamiento (calentar) mientras permite otro. Del mismo modo, el comportamiento de un motor en un robot está determinado por su estado de dirección, que puede ser "giro horario" o "giro antihorario". Otro ejemplo es un sistema en el que, al finalizar una operación, se cambia del estado trabajando al estado esperando, lo que también altera el comportamiento del sistema. De manera similar, un sistema de climatización puede estar en estado calentando o enfriando, dependiendo de la configuración del termostato. Por lo tanto, la gestión de estados, sus transiciones y los cambios de comportamiento asociados a estos son aspectos fundamentales en el diseño de sistemas complejos y representan un ítem de cambio bastante probable.

Una solución de implementación tradicional para gestionar estados consiste en almacenarlos en una variable. Posteriormente, se verifica el valor de esta variable para modificar el comportamiento de las funciones, como se ilustra en el código 6.20.

Código 6.20: Ejemplo de manejo de estados tradicional, en el caso del microondas.

```
calentar():
    if (estado == PuertaAbierta):
        return
    else if (estado == Preparado):
        magnetron.encender()
    else if (estado == Pausa):
        magnetron.encender()
```

Este enfoque, si bien es directo, presenta varias desventajas al modificar o extender el código. Por ejemplo, al agregar nuevos estados o cambiar el comportamiento del sistema, la gestión se vuelve rápidamente compleja. Esta creciente complejidad es una de las principales

desventajas de su implementación. A medida que se añaden más estados, cada método requiere más verificaciones, lo que conduce a una proliferación de estructuras *if-else*. Esto no solo dificulta la lectura del código, sino que también aumenta la probabilidad de introducir errores, ya que cada nuevo estado debe ser cuidadosamente incorporado y verificado en todas las partes relevantes del sistema. Además, este enfoque compromete la sostenibilidad del software. En el Código 6.21 se muestra cómo se agrega un nuevo estado al método `calentar()`.

Código 6.21: Ejemplo de introducción de un nuevo estado a la solución tradicional.

```
calentar():
    if (estado == PuertaAbierta):
        return
    else if (estado == Preparado):
        magnetron.encender()
    else if (estado == Pausa):
        magnetron.encender()
    else if (estado == Calentando):
        magnetron.apagar()
```

Cuando el comportamiento de un estado debe cambiar, es posible que se necesiten modificaciones en múltiples métodos. Esto puede llevar a la duplicación de código o dificultar la identificación de las partes que requieren cambios, complicando el proceso y aumentando el riesgo de errores.

Otra desventaja clave es la baja modularidad, ya que el comportamiento asociado a cada estado no está claramente delimitado. Si cada estado requiere comportamientos complejos, el código se vuelve monolítico y difícil de extender sin alterar los métodos existentes. Por ejemplo, al agregar un estado *Grill*, sería necesario modificar la lógica de múltiples métodos para verificar este nuevo estado y ajustar el comportamiento de manera apropiada. Todas estas deficiencias dificultan la reutilización del código y hacen que la verificación del correcto funcionamiento del sistema sea cada vez más compleja.

Este tipo de solución es ampliamente utilizado. Un ejemplo relevante se encuentra en el software original del robot desmalezador, como se describe en el informe [GIM18]. En particular, se definen diferentes estados de operación del robot y se utiliza una gran estructura de control *switch-case* dentro del bucle principal. Esta estructura decide la acción a ejecutar en función del estado actual, como se detalla en el Código 6.22.

Código 6.22: Main loop del previo firmware del robot desmalezador [GIM18]

```
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 abordar una perspectiva de diseño para gestionar el cambio, se analizará en detalle el ejemplo presentado en el libro de Douglass [Dou11] y la aplicación de una otra solución al manejo de estados.

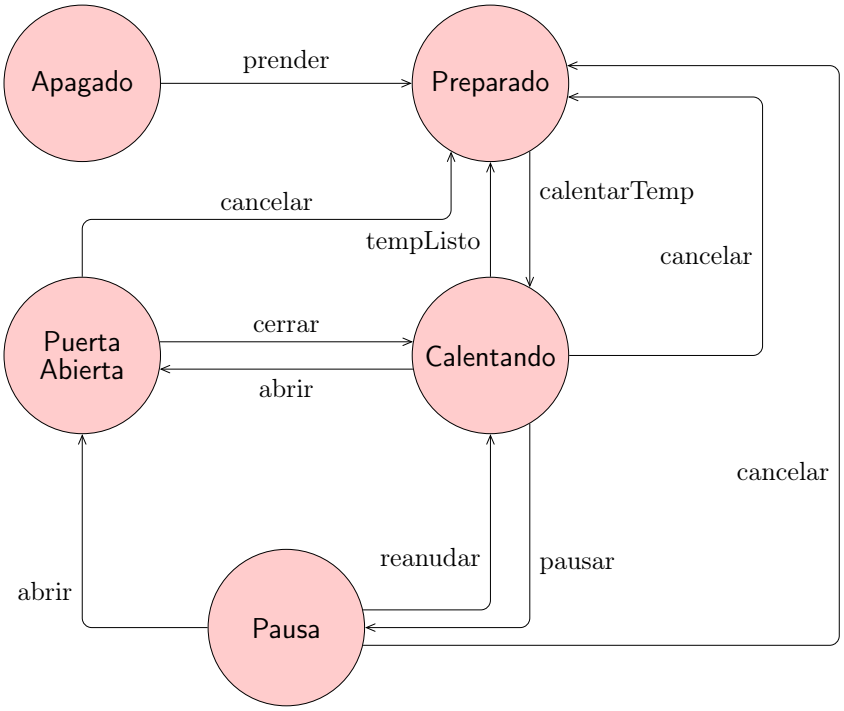
El ejemplo propuesto consiste en el sistema de control de un microondas. Para representar el comportamiento del mismo, el autor presenta una máquina de estados como la de la Figura 6.12. Aunque contiene algunas incoherencias y no es del todo exacta, es suficiente para el propósito del ejemplo que se busca exponer.

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 *cerrar*, se realizará una transición al estado *Calentando*. Durante este proceso, el magnetrón será activado para emitir las ondas necesarias que calentarán la comida.

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 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,

Figura 6.12: Máquina de estados extraída del libro [Dou11].



propone una implementación utilizando una tabla bidimensional lo cual lo hace un sistema eficiente computacionalmente hablando. En el caso de querer modificar estados, agregarlos o quitarlo es necesario cambiar la implementación de este módulo.

En el Código 6.23 se presenta la implementación de uno de los métodos fundamentales de la solución propuesta en el libro. Aunque este enfoque cumple con la gestión de estados, el código no es legible y además no parece seguir buenas prácticas de diseño, ya que utiliza al menos cuatro niveles de sentencias *if* anidadas. Esto no solo complejiza su entendimiento, sino que también dificulta futuras modificaciones. Por otro lado, dado que las estructuras de datos son un elemento común de cambio en el desarrollo de software, las interfaces deben diseñarse para ser lo más independientes posible de la estructura de datos que se utiliza para implementarlas. Al no considerar esta independencia, cualquier cambio en una de ellas puede repercutir en todo el sistema, lo que aumenta la probabilidad de introducir errores y la necesidad de una reverificación exhaustiva.

Código 6.23: Código ejemplo extraído de libro de Douglass State Table Pág. 305.

```
void TokenizerStateTable_eventDispatch(TokenizerStateTable* const me, Event e)
    {
int takeTransition = 0;
Mutex_lock(me->itsMutex);
/* first ensure the entry is within the table boundaries */
  if (me->stateID >= NULL_STATE && me->stateID <=
      GN_PROCESSINGFRACTIONALPART_STATE) {
    if (e.eType >= EVDIGIT && e.eType <= EVENDOFSTRING) {
      /* is there a valid transition for the current state and event? */
      if (me->table[me->stateID][e.eType].newState != NULL_STATE) {
        /* is there a guard? */
        if (me->table[me->stateID][e.eType].guardPtr == NULL)
          /* is the guard TRUE? */
          takeTransition = TRUE; /* if no guard, then it "evaluates" to TRUE
              */
        else
          takeTransition =(me->table[me->stateID][e.eType].guardPtr(me));
        if (takeTransition) {
          if (me->table[me->stateID][e.eType].exitActionPtr != NULL)
            if (me->table[me->stateID][e.eType].exitActionPtr->nParams == 0)
              me->table[me->stateID][e.eType].exitActionPtr->aPtr.a0(me);
            else
              me->table[me->stateID][e.eType].exitActionPtr->aPtr.a1(me, e.ed.
                  c);
          if (me->table[me->stateID][e.eType].transActionPtr != NULL)
            if (me->table[me->stateID][e.eType].transActionPtr->nParams == 0)
              me->table[me->stateID][e.eType].transActionPtr->aPtr.a0(me);
            else
              me->table[me->stateID][e.eType].transActionPtr->aPtr.a1(me, e.ed
                  .c);
          if (me->table[me->stateID][e.eType].entryActionPtr != NULL)
            if (me->table[me->stateID][e.eType].entryActionPtr->nParams == 0)
              me->table[me->stateID][e.eType].entryActionPtr->aPtr.a0(me);
            else
              me->table[me->stateID][e.eType].entryActionPtr->aPtr.a1(me, e.ed
                  .c);
          me->stateID = me->table[me->stateID][e.eType].newState;
        }
      }
    }
  }

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

Como un enfoque preparado para el cambio se propone el uso del patrón *State* de Gamma A.3. Con ciertos ajustes, este patrón permite que los estados realicen transiciones de forma autónoma. Básicamente, el patrón establece la creación de un módulo por cada estado del sistema, con el objetivo de ajustar la implementación al comportamiento que corresponde a dicho estado. Esto permite una transición dinámica entre estados, ya que el cambio de estado está representado por el cambio de módulo.

El patrón admite distintas opciones para gestionar la transición entre estados. En este caso, para permitir que cada estado transicione de manera independiente, se añade una referencia a los posibles estados siguientes. De esta forma, si se recibe el evento adecuado, un estado puede invocar el método para cambiar de estado, pasando la referencia del nuevo. Por lo tanto, el constructor de cada estado debe incluir como argumento una referencia a cada uno de sus posibles estados siguientes.

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

Para aplicar el patrón al ejemplo extraído del libro[Dou11], se debe seguir el siguiente procedimiento. En primer lugar, se crea un módulo que contenga un método para manejar cada evento y que, a su vez, delegue dicha gestión al estado actual. Posteriormente, se define un módulo que defina el estado del cual heredarán todos los estados concretos del sistema. En este caso, se creará un módulo por cada estado que se muestra en la máquina de estados de la Figura 6.12. En la Figura 6.13 podemos observar la estructura obtenida junto a seudocódigo que describen una posible implementación a modo de guía.

Siguiendo esta estructura modular, el módulo Microondas provee un método para cada evento existente. De esta forma, cuando un método es invocado, delega la funcionalidad al estado actual, lo que permite modificar el comportamiento del microondas según su estado. Un ejemplo de cómo implementar este proceso se presenta en el Código 6.24. El Microondas delega en el estado la acción de reanudar pasándose a sí mismo como argumento (`estado.reanudar(this)`) permitiendo la transición entre estados. Una posible implementación de este comportamiento se observa en el Código 6.25, donde el módulo Calentando apaga el magnetrón para proteger al usuario y luego transiciona al estado PuertaAbierta. Para realizar este cambio, el módulo Calentando necesita una referencia a PuertaAbierta, la cual fue almacenada previamente al ser pasada como argumento en su constructor.

Código 6.24: Ejemplo de implementación método reaundar del módulo Microondas

```
reaundar(){
    this.estado.reanudar(this)
}
```

Código 6.25: Ejemplo de implementación método abrir del módulo Calentando

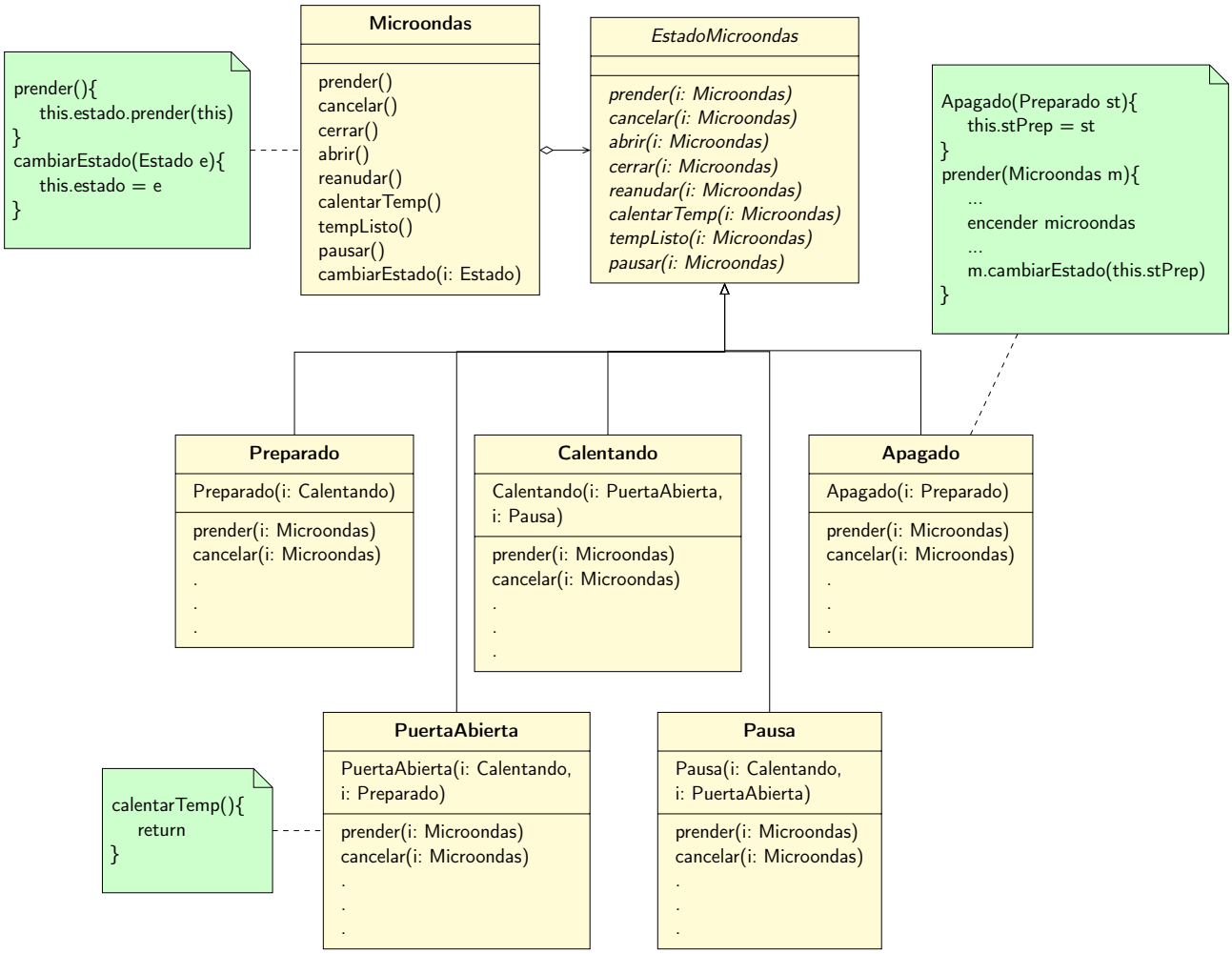
```
abrir(Microondas m){
    ...
    magnetron.apagar()
    m.cambiarEstado(this.stPuertaAbierta)
}
```

Cuando el microondas se encuentra en un estado que no admite un evento específico, el sistema puede simplemente no realizar ninguna acción. Por ejemplo, en el estado PuertaAbierta, el evento *calentarTemp* no puede desencadenar ningún comportamiento. En este caso, no se cumplen las condiciones para encender el magnetrón y calentar la comida. Por lo tanto, el evento es ignorado, lo cual puede implementarse como se muestra en el Código 6.26.

Código 6.26: Ejemplo de implementación método calentar del módulo PuertaAbierta

```
calentar(Microondas m){
    return }
```

Figura 6.13: Módulos participantes del patrón state en el ejemplo del horno microondas.



Como se explicó en la sección Documentación, cada aplicación de un patrón de diseño debe ser acompañada de su correspondiente documentación. En este caso, podemos observarla en la Figura 6.14; en la misma se establece qué módulo cumple la función de cada participante, los cambios previstos y la justificación de su aplicación.

Figura 6.14: Documentación de la aplicación del patrón *State* para el manejo de los estados del microondas.

PatternApp	**Estados del microondas**
based on	Estado (State)
why	**Cambios previstos**: En base a su estado, el microondas responde de diversas maneras a distintos eventos. Es posible modificar, agregar o eliminar estados. **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. Además, 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

Aplicando este patrón para el manejo de estados, se logra una solución más elegante y escalable para manejarlos, superando muchas de las limitaciones del enfoque basado en verificaciones de estado dentro de cada método.

El uso de este patrón mejora significativamente la legibilidad y la modularidad del código. En lugar de manejar el comportamiento de todos los estados en un mismo método, cada estado cuenta con su propio módulo. Esto permite que el código esté mejor organizado y sea más fácil de comprender, ya que cada módulo encapsula el comportamiento específico de su estado, eliminando la necesidad de múltiples verificaciones condicionales y simplificando el flujo de ejecución.

Otro beneficio importante es que facilita la adición de nuevos estados. Cuando se requiere incorporar un nuevo estado, como Descongelando (similar a Calentando, pero con un ciclo de encendido y apagado del magnetrón), basta con definir un nuevo módulo homónimo que herede de EstadoMicroondas. No es necesario modificar la implementación de los demás estados, ya que la nueva funcionalidad es independiente. Únicamente será necesario añadir este nuevo módulo a los constructores de aquellos estados que puedan precederlo. Esta facilidad hace que el sistema sea mucho más flexible y escalable a medida que aumenta su complejidad.

Además, el patrón *State* elimina la duplicación de código, ya que cada módulo de estado gestiona su propio comportamiento. En el enfoque tradicional, muchos métodos deben realizar verificaciones repetidas del estado y aplicar el comportamiento correspondiente, lo que conduce a duplicación e inconsistencias en el código.

Una ventaja clave es que permite cambiar dinámicamente el comportamiento del sistema. A medida que el estado evoluciona, también cambia dinámicamente el comportamiento del módulo principal.

Todo esto contribuye a que el sistema sea más robusto y adaptable. Cuando se necesita modificar el comportamiento de un estado, basta con actualizar la implementación del módulo correspondiente, lo que facilita la localización y modificación del código relevante. Como resultado, el sistema se vuelve más fácil de mantener y se reduce el riesgo de introducir errores al cambiar o extender sus funcionalidades.

Existen, además, otros casos de uso del patrón. Como se mostrará en los ejemplos posteriores, puede emplearse para modificar el comportamiento de ciertas partes del sistema o incluso como mecanismo de configuración del mismo. El concepto central permanece invariable: crear un módulo por cada estado posible y ajustar la implementación para que se adecúe a lo requerido.

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 entre las superficies. Esto genera señales que, de no ser filtradas, pueden causar lecturas erróneas. El comportamiento resultante puede observarse en la Figura 6.15, donde se muestra la señal generada por un botón al ser presionado. Si nuestro sistema, por ejemplo, generara un evento por cada transición, estaríamos contabilizando falsos positivos provocados por el fenómeno descrito.

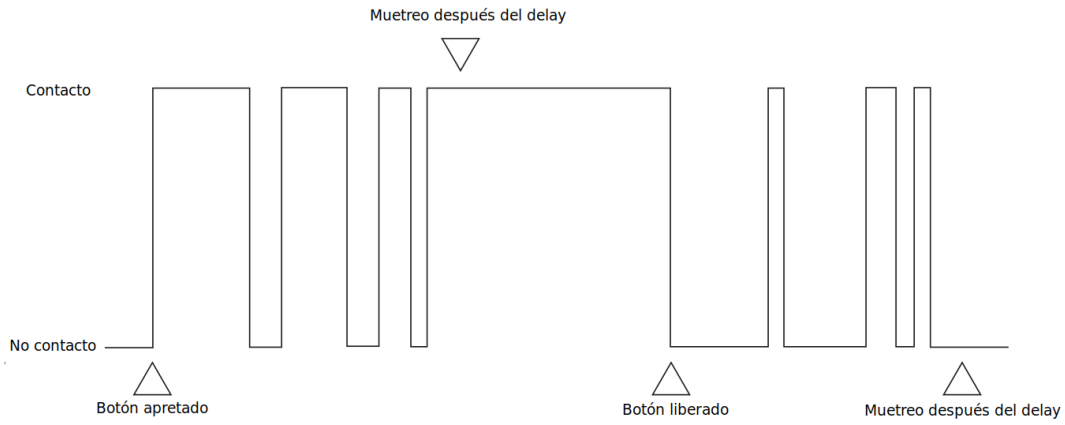


Figura 6.15: Ejemplo de señal de un botón siendo afectada por el fenómeno de rebote, extraída de [Dou11].

En el libro de Douglass [Dou11] encontramos una propuesta de solución más cerca de la implementación que del diseño, algo así como un patron idiomático. En el Código 6.27 se puede ver un extracto del libro, en donde se muestra la función que lleva a cabo la solución del autor. Esta consiste en utilizar un temporizador para permitir 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 6.27: Código ejemplo de una estrategia de detección de rebote extraído de [Dou11].

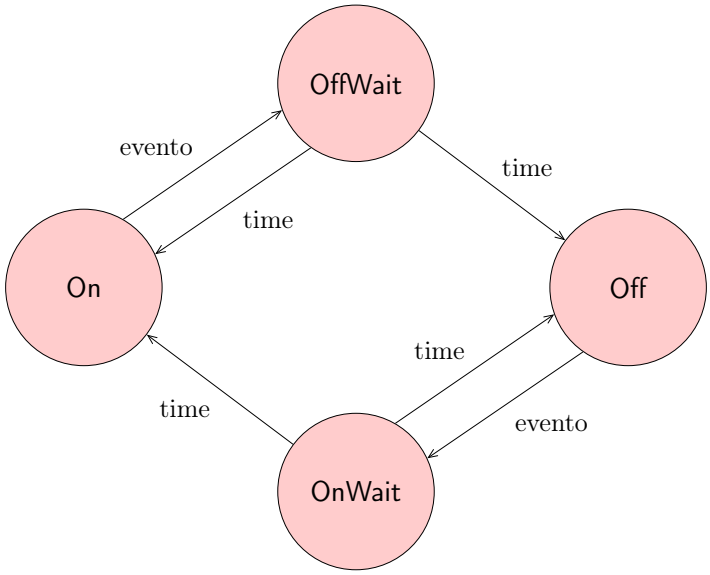
```
void ButtonDriver_eventReceive(ButtonDriver* const me) {
    Timer_delay(me->itsTimer, DEBOUNCE_TIME);
    if (Button_getState(me->itsButton) != me->oldState) {
        /* must be a valid button event */

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

        if (!me->oldState) {

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

Desde el punto de vista de la IS, podemos decir que se está aplicando una mala práctica si queremos anticiparnos al cambio. El estado del botón se almacena en una variable y las funciones modifican su comportamiento mediante sentencias *if*. Como vimos en el Capítulo 6.4, este enfoque puede conllevar múltiples inconvenientes al momento de expandir o modificar los estados. Si se debe agregar uno nuevo, todas las sentencias *if* deben actualizarse, lo que implica modificar la implementación de, en este caso, la función `ButtonDriver_eventReceive`.

Una solución más alineada con los principios de la IS consiste en aplicar el patrón de diseño *State* A.3 de Gamma, como vimos en el Capítulo 6.4. Podemos identificar dos estados principales: *On* y *Off*. Sin embargo, como veremos más adelante, será necesario añadir dos estados adicionales para poder representar el comportamiento deseado.

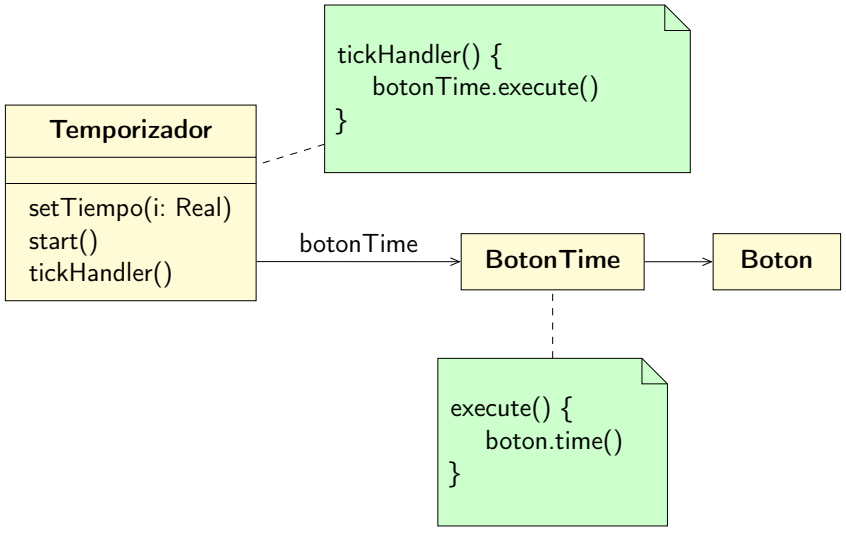
Volviendo al ejemplo del libro, ante la detección de actividad en el botón se recibe un evento. A continuación, se da un tiempo de gracia y luego se vuelve a verificar el valor que adopta la señal proveniente del botón en el registro. Este valor determina si la señal indica que el botón está encendido o no. Haciendo uso de estos recursos, utilizaremos el patrón *State* para representar cuatro posibles estados: On, Off, OnWait y OffWait. Los dos primeros reflejan que el botón se encuentra encendido o apagado, respectivamente, mientras que los otros dos corresponden a la etapa de espera durante el tiempo de gracia, necesaria para determinar si existió o no una transición. Esta máquina de estados es la definida en la Figura 6.16.

Figura 6.16: Máquina de estados propuesta para el control antirebote.



Por otro lado, para gestionar el tiempo de gracia utilizaremos un módulo llamado Temporizador, el cual permite suscribir un método que será invocado luego de transcurridos x segundos. Este módulo y los demas involucrados pueden observarse en el diagrama de la Figura 6.17.

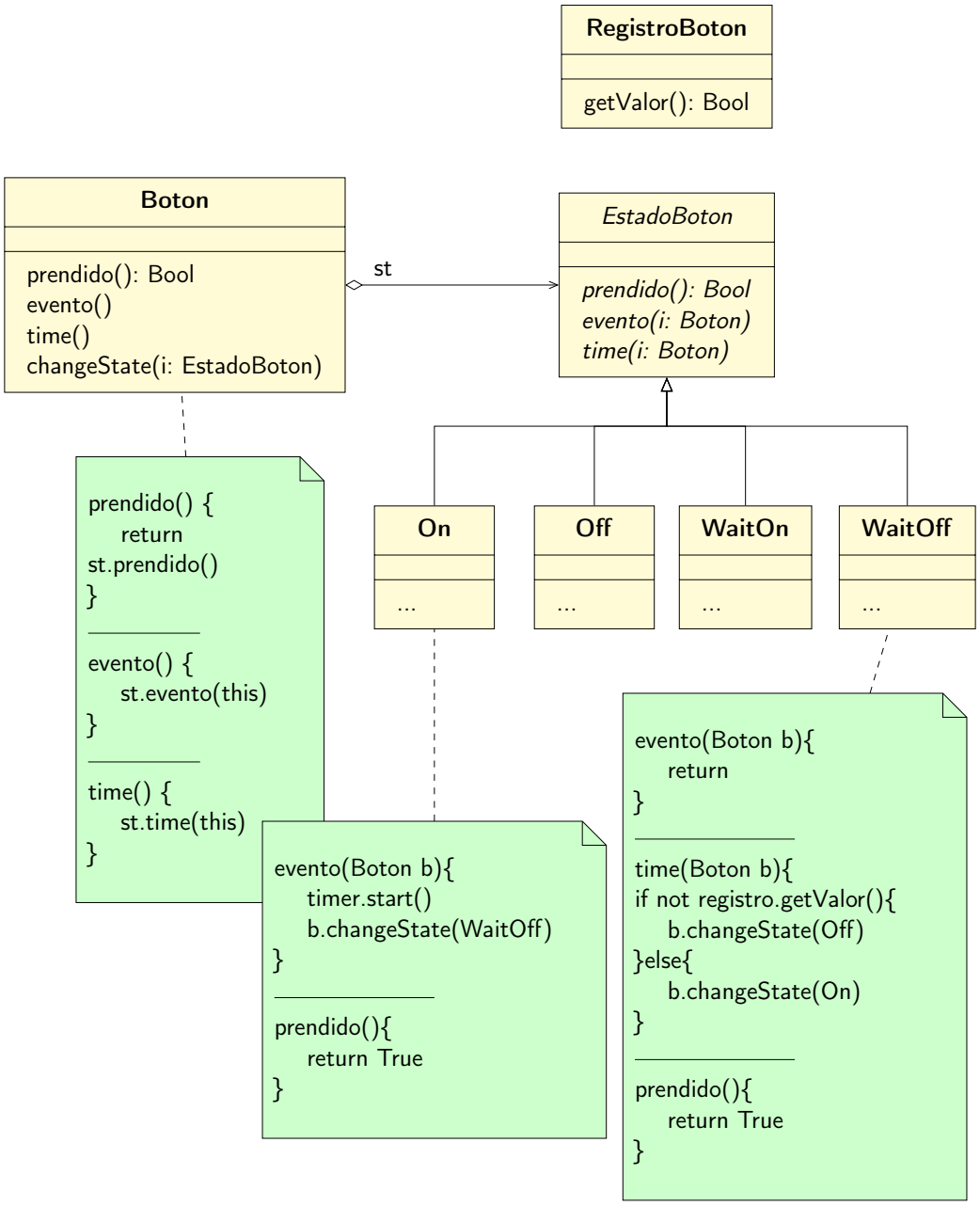
Figura 6.17: Módulos encargados de gestionar el comportamiento del temporizador utilizado en la transisción de estados del módulo Boton.



Resuelto el funcionamiento del temporizador, veamos ahora el módulo Boton, que será el utilizado por los clientes para conocer el estado del botón físico. Para ello, definimos también

el estado y el módulo que encapsula el acceso al registro donde se almacena el valor de la señal proveniente del botón. Todos estos módulos pueden observarse en la Figura 6.18.

Figura 6.18: Diagrama de módulos relacionados al botón.



Repasemos cómo funciona el sistema con la estructura propuesta. Supongamos que el botón se encuentra encendido, por lo tanto, su estado es On. Cuando se produce un evento, es decir, se detecta algún tipo de cambio en la señal proveniente del botón, se ejecuta el método `evento` del módulo Boton. Este último delega su funcionamiento al estado. Como

Figura 6.19: Documentación de la aplicación del patrón *State* para el control del estado del botón.

PatternApp	**Estados del botón**
based on	Estado (State)
why	**Cambios previstos**: Pueden mofificarse los estados de cotrol del botón en base a nuevos requerimientos o cambios en el hardware. **Funcionalidad**: La interfaz de usuario mostrará el botón como encendido o apagado según su estado interno. Para crear un retardo en las transiciones, es necesario añadir estados intermedios que gestionen esa demora.
where	Boton **is** Contexto EstadoBoton **is** Estado On **is** EstadoConcreto Off **is** EstadoConcreto OnWaiting **is** EstadoConcreto OffWaiting **is** EstadoConcreto

dijimos que el botón estaba prendido, el encargado de manejar el evento será On. Este iniciará el temporizador con el tiempo configurado invocando `timer.start()` y cambiará el estado del módulo Boton a OffWait. Estando el botón encendido, solo puede apagarse; es por eso que se transiciona al estado OffWait, el cual decidirá, una vez transcurrido cierto tiempo, si corresponde pasar a Off. Esto lo hace verificando el valor presente en el registro del botón cuando el temporizador ejecuta el método `time()` de Boton. En caso de que el registro muestre que el botón está apagado, efectivamente se cambiará el estado a Off; en caso contrario, se volverá a On, ya que se trató de una detección errónea, probablemente provocada por el efecto rebote.

Todos los estados deben implementar el método `prendido():Bool`, que es el utilizado por los clientes para conocer el verdadero estado del botón. Si el estado es On o OffWait, `prendido():Bool` devolverá `True`; en caso contrario, devolverá `False`.

Dado que la solución propuesta es una aplicación del patrón *State*, debe ser documentada con el método 2MIL, tal como se presenta en la Figura 6.19.

De esta manera, logramos encapsular cada estado en un módulo independiente, que puede ser modificado sin afectar al resto. Además, agregar nuevos estados resulta sencillo: basta con crear un nuevo módulo.

6.6 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**[1] es a través 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 un método que realiza cierto cómputo en base a dos argumentos, `computar(i: int, i: int)`. Existen múltiples posibles implementaciones, estas son algunas:

- Una posible implementación consiste en que la función no realice ninguna verificación y simplemente intente utilizar los valores pasados como argumento, lo cual puede generar un error inmediato si los tipos no coinciden, o un error futuro si el valor está fuera de los parámetros requeridos por el sistema. Por ejemplo, si se pasa un entero negativo, pero el método requiere como precondición que siempre sea positivo, se podría provocar un error división por cero más adelante en la ejecución; esta implementación obliga a que todos los clientes que utilicen este método deban verificar la validez de los valores, lo que no solo genera código repetido, sino que también requiere una actualización manual en múltiples puntos si el requisito cambia, haciendo que la responsabilidad de la verificación de las precondiciones recaiga en el cliente que utiliza la función, en lugar de ser manejada internamente por la misma.

- Otra opción se basa en verificar si el valor es permitido, haciendo que `computar(i: int, i: int)` retorne un valor que indique el resultado de la validación (por ejemplo, 0 en caso afirmativo y -1 en caso contrario). Si bien este enfoque es una mejora respecto a no realizar validación alguna, continua con el acoplamiento entre el cliente de la función y el módulo que la exporta, ya que el cliente debe verificar el valor de retorno para determinar si ha ocurrido un error.

La solución propuesta en el libro de Douglass [Dou11] es más cercana a un patrón idiomático o a una práctica de programación que a un patrón de diseño. Sus conceptos clave son los siguientes:

- Construir tipos de autoverificación 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.

Por lo tanto, la solución planteada no constituye un patrón de diseño general. No obstante, desde la perspectiva de la IS, es posible preparar el sistema para futuros cambios y, al mismo tiempo, realizar las verificaciones necesarias. Para ello, proponemos la aplicación del patrón *Decorator* A.5 de Gamma, el cual permite añadir o eliminar funcionalidades a módulos de manera dinámica. En particular, lo que se busca es agregar la verificación de cada argumento, manteniéndola separada del módulo que implementa la funcionalidad específica.

Para la construcción de la solución, se tomará del libro [Dou11] el concepto de un módulo denominado ErrorHandler, o manejador de errores. Esta es una técnica ampliamente utilizada

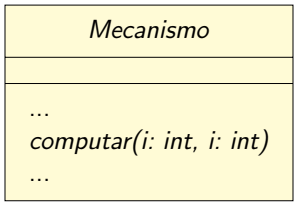
[1] Lenguaje de programación comúnmente utilizado en sistemas embebidos.

e incorporada en ciertos lenguajes de programación, que se basa en delegar la gestión de un error a un módulo especializado. Dicho módulo, basándose en la lógica definida por el desarrollador, resolverá el inconveniente; por ejemplo, puede mostrar un error en pantalla y detener la ejecución, reintentar ejecutar un método o reiniciar ciertos componentes. Existen múltiples enfoques para este problema, algunos de los cuales se estudian en [Gla+].

Lo relevante de esta estrategia es que el módulo proporciona una interfaz clara para el manejo de errores, cuya estructura se detallará a continuación con un ejemplo.

Suponga que se tiene el módulo *Mecanismo* cuya interfaz es la que se muestra en la Figura 6.20. Como se puede ver uno de los métodos es `computar`, el cual para funcionar correctamente y sin errores, requiere que el primer argumento sea siempre mayor a cero y que el segundo sea par. En lugar de agregar el Código 6.28 al inicio de la implementación de `computar`, proponemos usar dos módulos que actúen como decoradores y encapsulen esta funcionalidad.

Figura 6.20: Interfaz del módulo *Mecanismo*.

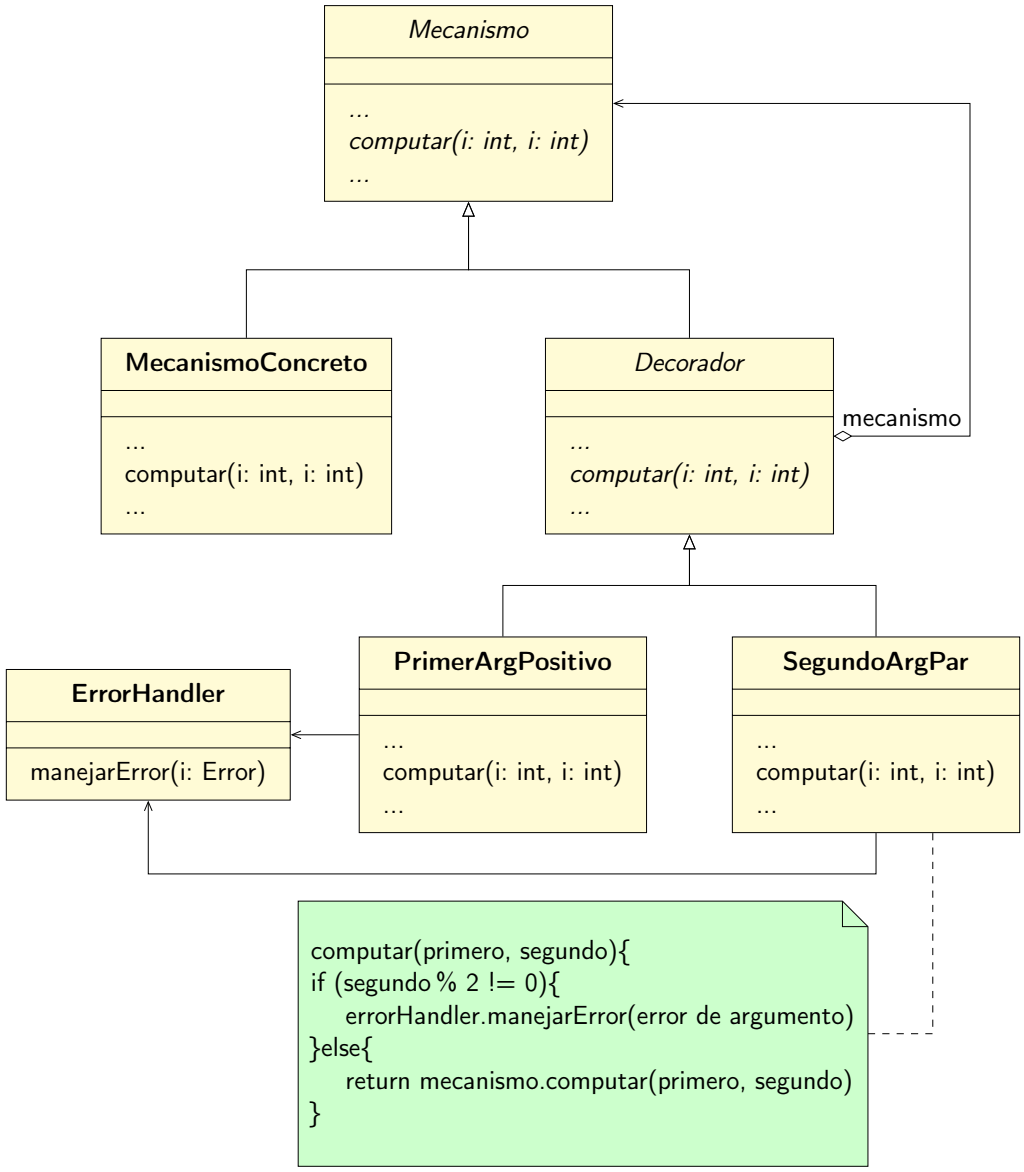


Código 6.28: Verificación de precondiciones en método Computar.

```
computar(int primer, int segundo){
    if (primer <= 0)
        return -1
    if (segundo % 2 != 0)
        return -1
    .
    .
    .
```

En la Figura 6.21 se presenta la estructura de módulos conseguida a partir de la aplicando el patrón y el uso del módulo ErrorHandler.

Figura 6.21: Ejemplo de aplicación del patrón *Decorator* A.5 para garantizar el cumplimiento de precondiciones.



Para utilizar el método `computar` del módulo MecanismoConcreto con la verificación de cada argumento, es necesario componerlo, en tiempo de ejecución, con todas las verificaciones de precondiciones deseadas; en este caso, PrimerArgPositivo y SegundoArgPar. De esta manera, el cliente ejecutará el método `computar` de PrimerArgPositivo, el cual realizará su verificación y, si es correcta, invocará el método computar del *Mecanismo* que tiene compuesto, que en este caso es SegundoArgPar. Este último hará lo mismo, pero invocando a `computar` de MecanismoConcreto. Así, se crea un encadenamiento de módulos de forma dinámica, lo que permite agregar y quitar funcionalidades con facilidad.

En la solución aplicamos un patrón de diseño, por lo tanto, debemos presentar la corres-

pondiente documentación para que otros desarrolladores puedan comprender el diseño. Esta la podemos ver en la Figura 6.22.

Figura 6.22: Documentación de la aplicación del patrón *Decorator* al caso de verificación de la integridad de la información.

PatternApp	**Módulo 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 u otras funcionalidades. **Funcionalidad**: El módulo decorador es utilizado para extender la funcionalidad del módulo ComponenteConcreto. Antes de invocar al método de este último se ejecutan la homónima del decorador.
where	Mecanismo **is** Componente MecanismoConcreto **is** ComponenteConcreto Decorador **is** Decorador PrimerArgPositivo **is** DecoradorConcreto SegundoArgPar **is** DecoradorConcreto

Con este diseño se añade una capa de verificación sin modificar el módulo original, lo que ofrece las siguientes ventajas:

- Se desacopla el cliente del módulo invocado, eliminando la necesidad de que el cliente verifique las precondiciones de un método antes de su ejecución.
- La lógica de cada verificación se encapsula en un solo módulo, manteniendo la separación de responsabilidades y facilitando su reutilización. De esta forma, la implementación del módulo original que realiza el comportamiento requerido no necesita ser modificado.
- El patrón *decorator* permite que el sistema pueda evolucionar, permitiendo añadir nuevas funcionalidades o modificar las reglas de verificación sin afectar la implementación original. Esto hace posible, por ejemplo, implementar restricciones de acceso, límites en las operaciones sin alterar el módulo subyacente o verificaciones de integridad de la información mucho más complejas (ver Sección 6.6 a continuación).

Integridad de la información

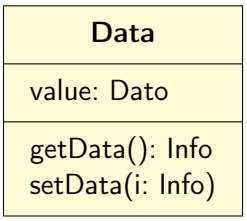
Como se ha señalado en otros *problemas comunes*, el ejemplo propuesto es una versión simplificada, pero es posible agregar funcionalidades mucho más complejas. Una de ellas es la verificación de la integridad de la información almacenada en el sistema.

Douglass, en [Dou11], sostiene que uno de los orígenes más comunes de inconvenientes relacionados con fallas de seguridad o confiabilidad es la corrupción de la información. Un pulso

electromagnético o fallas en el hardware pueden causar daños en los datos, comprometiendo su integridad; por ejemplo, alterando un Bit flip o provocando la pérdida 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 diversas técnicas, desde el cálculo de checksums con el fin de verificar la integridad, hasta el almacenamiento de la información múltiples veces para crear redundancia.

Estas técnicas suelen ser añadidas al código modificando la implementación de los métodos del módulo encargado de almacenar la información, o agregando verificaciones en las partes del código que la utilizan. Es decir, si se utiliza el módulo Data (ver Figura 6.23), se realiza una llamada al método `getData()` para obtener la información almacenada y luego se ejecutan las validaciones necesarias para verificar su integridad. Este enfoque, sin embargo, provoca código repetido en cada llamada a `getData()` y alto acoplamiento entre módulos. Por lo que, ante cualquier cambio en el formato de los datos, la estrategia de validación utilizada podría dejar de ser aplicable, lo que obligaría a modificar múltiples partes del código.

Figura 6.23: Interfaz del módulo Data.



Analizando el patrón *decorator* A.5 aplicado a este *problema común*, se puede intuir que es posible utilizarlo para agregar al módulo Data la funcionalidad de verificar la integridad de la información almacenada. La solución consiste en crear un decorador para el módulo Data que, en primer lugar, obtenga la información y luego aplique algún tipo de verificación de integridad, como el cálculo de un checksum. En el código 6.29 se puede observar una posible implementación de los métodos `getData()` y `setData()` de este nuevo decorador que llamaremos DecoraChecksum, siguiendo el procedimiento mencionado.

Código 6.29: Implementación de los métodos getData y setData del decorador que se encarga de verificar la integridad de la información almacenada en el módulo Data.

```
// Este decorador esta compuesto con el m dulo Data, en este caso
// la instancia est  almacenada en la variable this.data

DecoraChecksum::setData(Info info){
    this.checksum = calcularChecksum(info)
    this.data.setData(info)
}

DecoraChecksum::getData() {
    info = this.data.getData()
    if (calcularChecksum(info) == this.checksum)
        return info
```

```
13      else
14          errorHandler(error de integridad)
15  }
```

Este es otro ejemplo en el que el patrón *decorator* permite la adición de una funcionalidad a un módulo sin modificarlo.

6.7 Organización de la ejecución

En un sistema robótico embebido se suelen ejecutar múltiples tareas para lograr un cierto objetivo. Ya sea, si se eligió seguir un estilo arquitectónico del tipo "Control de procesos"(recordar Sección 5.1) 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, consiste 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 cómputo.

Una estrategia para mejorar estos puntos consiste en utilizar las interrupciones del microcontrolador, lo que permite atender a todas las señales de entrada y a los eventos del entorno en tiempo real. En particular, en el diseño del robot desmalezador [Pom+24], las interrupciones se utilizan en conjunto con un estilo arquitectónico de control de procesos Sección 5.1. El sistema resultante posee tres tareas principales:

- Recibir información de los sensores y de la PC (por ejemplo, a través del puerto serial).
- Procesar la información recibida y decidir qué 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 deben atender 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 manejador se encargue de realizar todo el proceso de control y cálculo. En el manejo de esa interrupción, se accederá a las 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.

Como se puede deducir, el uso de interrupciones es abundante en sistemas de tiempo real y de control. Por lo tanto, resulta imprescindible tener un buen manejo de ellas. Las interrupciones, como su nombre indica, detienen la ejecución normal de un programa para atender a un evento. Una vez que este manejo finaliza, la ejecución continúa en el punto donde fue interrumpida (más información referida a interrupciones en la Sección 3.2).

Cuando una interrupción es interrumpida por otra, el sistema prioriza la más reciente, ejecutándola primero y reanudando después las anteriores. Un problema común y evitable surge con las interrupciones periódicas: si el tiempo de manejo de una interrupción es mayor que el intervalo de su disparo, por ejemplo, si un temporizador la genera cada x segundos, pero su manejo tarda $x + 2$ segundos, se produce una situación en la que la ejecución nunca finaliza hasta que se desactive el *timer* que lanza las interrupciones. Además de que la tarea no se complete, el sistema puede quedar en un estado inconsistente. Por ejemplo, en un ciclo de control (lectura, procesamiento y actuación), una nueva interrupción puede ocurrir antes de que el ciclo concluya, provocando que la etapa de actuación sobre los actuadores nunca se ejecute y el sistema no realice ninguna acción.

Los problemas de concurrencia de este tipo son difíciles de detectar y corregir, por lo que es útil contar con herramientas para evitarlos. Una forma común de hacerlo es apagando las interrupciones del MCU al comenzar el manejo de una interrupción. No obstante, esto no es ideal, ya que se perderían interrupciones no periódicas que sí se desean manejar. Otra opción es eliminar o desuscribir el manejador de la interrupción al comenzar su ejecución, aunque es crucial recordar volver a agregarlo para que el próximo ciclo se ejecute.

Desde la IS, resulta interesante proponer un diseño para evitar este problema. Para ello, se aplicará el patrón *State* A.3 para representar dos estados posibles en relación con el manejo de una interrupción: *Activo* (disponible para manejar la interrupción) e *Ignorar* (el estado responsable de ignorar interrupciones mientras una está siendo manejada). De este modo, cuando el manejo de una interrupción comience, el estado cambiará a *Ignorar*, lo que hará que cualquier nueva llamada al manejador resulte en un retorno rápido. Una vez que la ejecución del manejo finalice, el estado volverá a cambiar, permitiendo el manejo de nuevas llamadas a la interrupción.

Veamos ahora la estructura de módulos subyacente a esta solución en la Figura 6.24 y en la Figura 6.25 la correspondiente documentación del uso del patrón en este ejemplo.

Figura 6.24: Ejemplo de manejador de interrupciones usando *State* para prevenir inconvenientes en la ejecución.

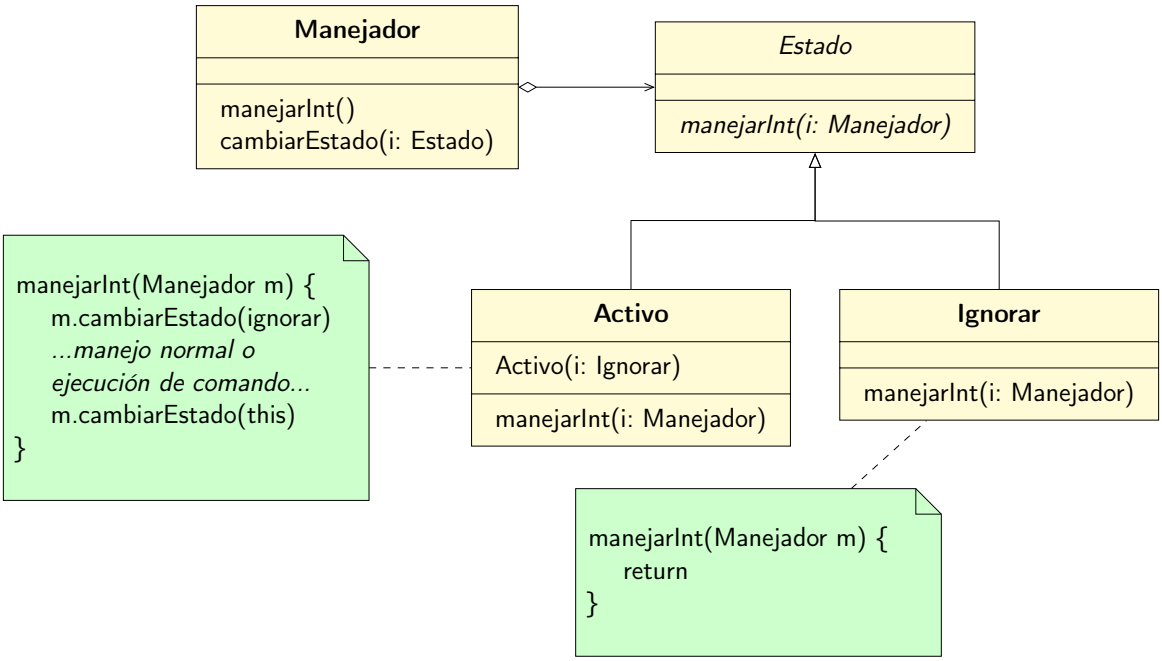


Figura 6.25: Documentación de la aplicación del patron *State* para el ejemplo del manejador de interrupciones.

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 no pause la ejecución de otra del mismo tipo.
where	Manejador **is** Contexto Estado **is** Estado Ignorar **is** EstadoConcreto Activo **is** EstadoConcreto

Al aplicar el uso de interrupciones junto con patrones de diseño como *State*, el sistema embebido de control robótico adquiere una estructura más robusta y mantenible. Se logran reducir el riesgo de problemas de concurrencia provocados por el manejo de interrupciones en tiempo real. Y le quita responsabilidades a los módulos que efectivamente hacen el manejo de la interrupción.

6.8 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 controlar 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 líneas de código, por lo que un diseño orientado al cambio resulta clave.

Enfoque de diseño tradicional

Antes de pasar a explicar la solución preparada para el cambio 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 creado 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 orientado al cambio en [Pom+24].

Estructura y funcionamiento general

El sistema desarrollado controla el siguiente hardware del robot. Se 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 este 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 agrupan 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 operación 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 (recordar Sección 3.3) y terminar la ejecución. Para las primeras dos tareas llama a dos funciones que realizan el trabajo. El ciclo de control se ve representado por un bucle infinito en el cual se realizan las siguientes tareas principales:

- Lectura de entradas, es decir, valores de referencia a alcanzar.
- Lectura de información de los sensores, dirección, velocidad y corriente.
- En base al estado de ejecución actual, se realizan ciertas tareas. Por ejemplo, en el estado `DUTY_REMOTO`, cuando el sistema está recibiendo una orden a través del control remoto, se lleva a cabo el ciclo de control completo. Por otro lado, en el estado `EMERGENCIA`, el robot debe detenerse, por lo que su comportamiento será diferente.
- Se aplican cambios a los actuadores, el ciclo de trabajo (PWM) de los motores y la dirección.

Observaciones

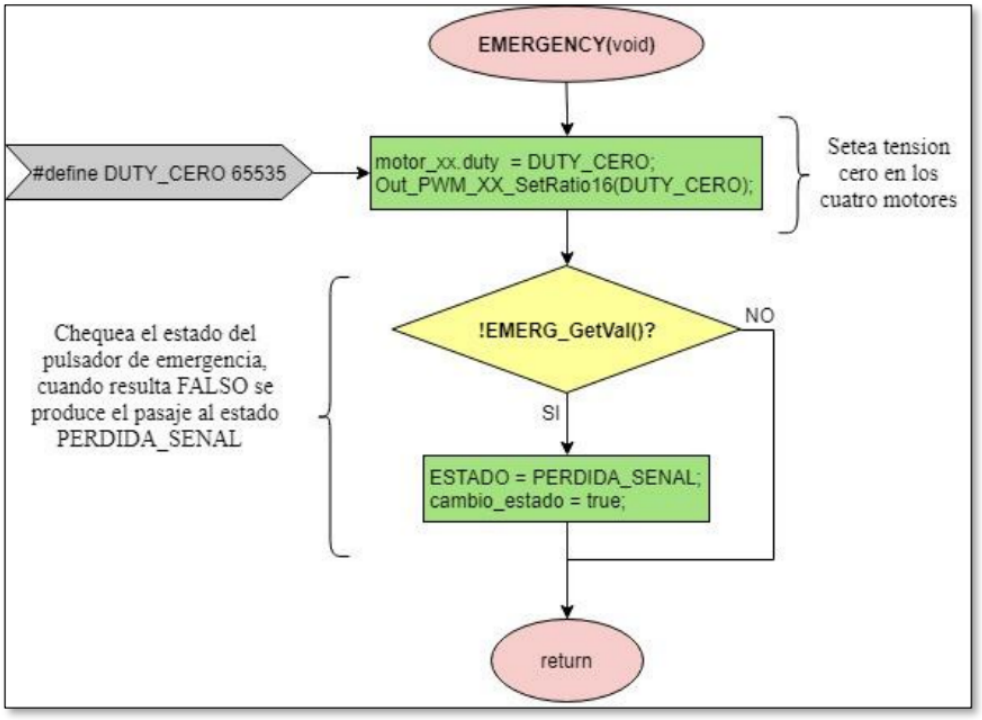
El criterio de división del código parece ser funcional, tanto por el aspecto del código, como por la documentación adjunta en los informes [GIM19, pág. 78-85], [BCD18, pág. 110-149]. El criterio de división funcional, se centra es describir la funcionalidad de cada método y para hacerlo se muestra su comportamiento utilizando diagramas de flujo como el de la Figura 6.26.

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 integradas en la lógica del control (manejo del ciclo de control), lo que indica una baja separación entre la capa de abstracción del hardware y la gestión del ciclo de control. Esto es acompañado con un diseño procedimental, con una serie de pasos secuenciales y un control centralizado en el flujo principal `main`.

Estos son 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 que no son deseadas.
- Los estados se definen en variables y se utilizan sentencias *if* o *switch* para cambiar el comportamiento de las funciones. Como se vio en la Sección 6.4 esto no es una buena práctica cuando se prepara el sistema para el cambio.
- 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. Además, se utilizan múltiples variables globales, las cuales añaden acoplamiento

Figura 6.26: Diagrama de flujo que explica el comportamiento de la función emergency [GIM19, pág. 82].



entre funciones y se debe ser cuidadoso al momento de introducir cambios que las afecten.

- 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. Es decir, no hay una decisión de diseño que contemple la existencia de posibles errores durante la ejecución. 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 no parece diseñada para escalar con nuevas funcionalidades.

Enfoque mejorado pero deficiente

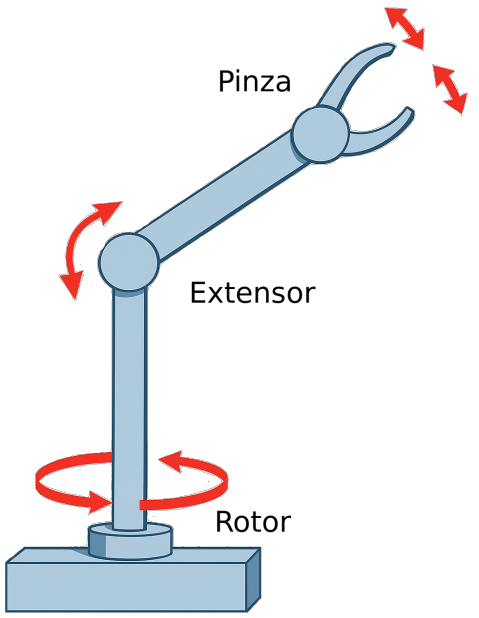
A continuación se describirá un ejemplo extraído de [Dou11, Sección. 3.4], se explicará la solución propuesta en el mismo libro para luego proponer una solución orientada al cambio.

Requisitos del sistema de control de un brazo robótico

Se necesita desarrollar el software de control de un brazo robótico que consta de tres **actuadores**, dos motores paso a paso (uno para rotar en su base y otro para extender o retraer el brazo) y una pinza que se puede cerrar o abrir. En le Figura 6.27 podemos observar un esquema del hardware. Para controlar el comportamiento del brazo, se provee una función compleja la cual toma coordenadas en el espacio y devuelve una secuencia de pasos para que el brazo tome un objeto en la posición determinada por las coordenadas. Cada paso consta de una orden para cada actuador del brazo robótico. Estos se deben ejecutar de manera secuencial, es decir que el paso número dos empezará su ejecución solo si el primero culminó completamente y con éxito. En caso de que las coordenadas sean inalcanzables la función devuelve cero pasos. Si se produce un error durante la ejecución de un paso, se detiene la ejecución del sistema.

Controlar motores paso a paso es más complejo de lo que parece. Habitualmente, se gestionan con dos funciones: una que define el sentido de giro y otra que ordena dar un solo paso. La longitud de cada paso viene determinada por el hardware del motor. Por lo tanto, para lograr movimientos más largos, se deben ejecutar múltiples pasos, lo que requiere un control exhaustivo por parte del software. Este es un requerimiento implícito que el sistema tiene y se deberá atender.

Figura 6.27: Esquema del brazo robótico.



Solución propuesta en el libro de Douglass

Como solución, el autor propone la creación de un módulo llamado **RobotArmManager**, cuya función es gestionar los actuadores y coordinar su comportamiento. Además, para cada tipo de 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 establecer 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 la función que se mencionó en los requerimientos. En este caso es llamada `graspAt(i: Coordenadas)`.

La solución propuesta parece estar un nivel de abstracción por encima de lo que se esperaría para este caso. Aunque no hay suficiente información sobre el hardware del brazo robótico, sabemos que utilizar motores paso a paso no es tan simple como invocar un método. Este proceso suele requerir control continuo, operando mediante pulsos que hacen que el motor avance de a un paso. Por lo tanto, los módulos encargados de los movimientos probablemente tengan más responsabilidades de las que se plantean en el libro. Esto va en contra de las prácticas de la IS, que establecen que un módulo debe ocultar un solo elemento de cambio. 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, Douglass aplica el patrón *Mediator* A.4 de Gamma. De manera que **RobotArmManager** cumple el rol de **Mediador** y cada módulo que se encarga de manejar cada actuador cumple el rol de **Colega**. Los autores en [Gam+95] establecen que el patrón es aplicable en los siguientes casos:

- Un conjunto de módulos se comunica de maneras bien definidas pero complejas. Las interdependencias resultantes son poco estructuradas 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 construida por Douglass es similar al patrón y logra el objetivo de reducir el acoplamiento entre colegas, evitando que los módulos se refieran directamente entre ellos. Podemos decir desde la perspectiva de la IS que la aplicación del patrón ayuda a preparar el sistema ante un cambio en los colegas o su forma de comunicación.

Por otro lado, un ítem de cambio común son las estructuras de datos, como se mencionó en la Sección 4.3. Por ello, establecer el uso de una lista directamente en la interfaz de un módulo no responde a una buena práctica.

Es posible que el problema haya sido simplificado con fines didácticos. Con el objetivo de explicar la aplicación del patrón *Mediator*. La solución planteada parece alejarse de una implementación realista, dejando requisitos menos específicos que podrían dar lugar a diferentes interpretaciones.

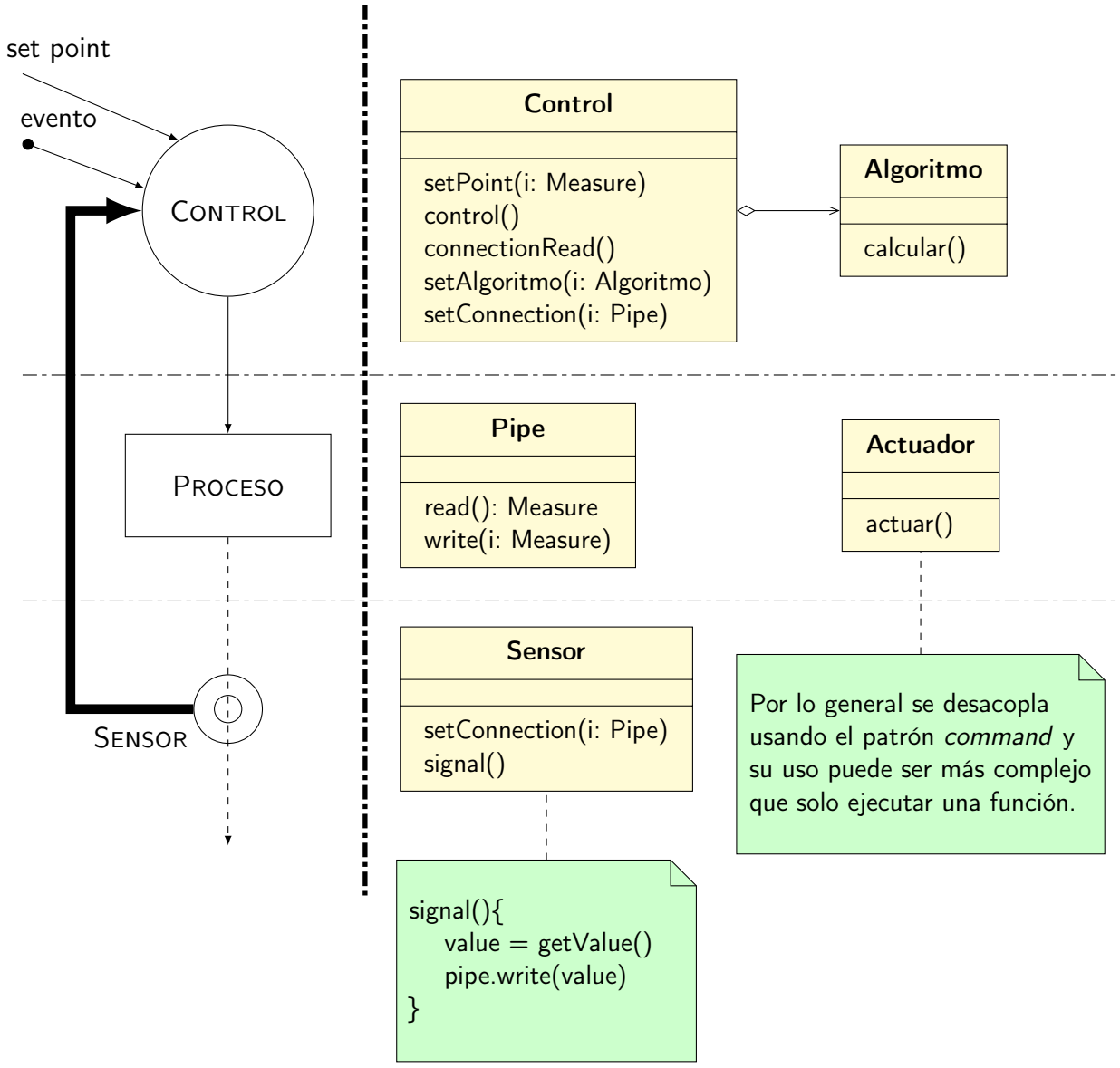
Enfoque para el cambio: Subsistemas de control

Para dar la solución orientada al cambio y completa que se propone en este trabajo, primero se introducirá el concepto de lo que llamaremos **Subsistema de control**. El cual fue extraído del trabajo realizado en el robot desmalezador [Pom+24] y se utilizará como diseño de referencia.

En un sistema complejo donde se controlan múltiples propiedades físicas, el sistema se divide en subsistemas. Cada subsistema controla **una** propiedad específica y se organiza según el estilo arquitectónico de control de procesos (Sección 5.1). De esta manera, se logra independencia entre el control de cada propiedad, lo que prepara al sistema para cambios individuales en el control de cada una de ellas. Además, se promueve la reutilización del código, ya que un subsistema de control diseñado para manejar una propiedad específica puede aplicarse en diferentes escenarios o sistemas.

Estos subsistemas de control son una estructura de módulos relacionados mediante herencia, composición o invocación. Cuyo trabajo es lograr que cierta propiedad física alcance el valor deseado. Para hacerlo interactúa con la variable a manipular y las variables medidas relacionadas (recordar Tabla 5.1). En la Figura 6.28 se puede observar de manera concreta los módulos que forman parte de cada componente de un subsistema. Para permitir la interacción con un cliente externo deben proveer una interfaz que permita establecer un valor al que se quiera llevar la propiedad (ver *setPoint* en la Tabla 5.1) y que indique el comienzo de la tarea de control.

Figura 6.28: Módulos de un subsistema de control y el componente arquitectónico al que pertenecen.



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.

- Pipe: módulo que se ubica entre Control y Sensor tiene un funcionamiento sencillo, ofrece dos métodos: uno para escribir y otro para leer. De esta manera, un módulo escribe y otro lee, lo que permite comunicación entre ellos sin necesidad de invocación directa. Encapsula la comunicación entre módulos desacoplandolos.

Una vez decidido el valor deseado o *set-point* al cual se busca aproximar la variable física, se puede comenzar a ejecutar el ciclo de control. Este será invocado de manera periódica mediante un temporizador. Siguiendo el estilo arquitectónico de control de procesos, se define que un ciclo de control consta de los pasos mostrados en el Código 6.30. En primer lugar, se establece el set-point utilizando el método `setPoint(i: Measure)` de Control. Luego, se indica a Sensor que escriba en Pipe el valor de la variable medida. En la línea 3, se solicita a Control que lea dicho valor y, una vez recuperado, pueda decidir qué cambios aplicar en Actuador. De esta manera, luego de una serie de iteraciones se espera que la variable física se encuentre más cerca del *set-point*.

Código 6.30: Ejemplo de uso del subsistema.

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

Notar que en la Figura 6.28 aparece un módulo que no fue mencionado anteriormente, Algoritmo el cual se encarga de los cálculos necesarios para determinar de qué manera accionar sobre el o los actuadores. Control delega en Algoritmo la tarea de realizar los cómputos. Si el algoritmo cambia (recordar que es un ítem de cambio probable Sección 4.3) solo ese módulo se ve afectado. Si se definió correctamente la interfaz, solo se agrega otro módulo con el nuevo algoritmo. No es necesario modificar el módulo Control. Esta representa una de las aplicaciones más directas y sencillas del patrón *Strategy* A.8.

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 él, por ejemplo, los controladores PID[Min22] (controlador proporcional, integral y derivativo). Este tipo de técnicas puede ser usada siguiendo la estructura modular propuesta y, en particular, los cálculos asociados se definirían en este módulo Algoritmo.

Como se mencionó previamente, cuando se presentó el ejemplo extraído de [Dou11], los motores paso a paso requieren de un seguimiento exaustivo para cumplir su cometido. Por ejemplo, suponga que Control decidió que es necesario que el motor gire 30°, y que un paso del motor representa solo 1°. En ese caso, deberíamos aplicar múltiples cambios en el actuador (motor paso a paso) para alcanzar el objetivo. No es posible ejecutar todos los pulsos al mismo tiempo, sino que debemos respetar los tiempos de actuación del motor.

Por otro lado, si el motor requiere menos tiempo que el intervalo entre ciclos de control para completar un paso, estaríamos desperdiciando una parte significativa de tiempo. Es por esto que se presenta una modificación del módulo Control para manejar actuadores que

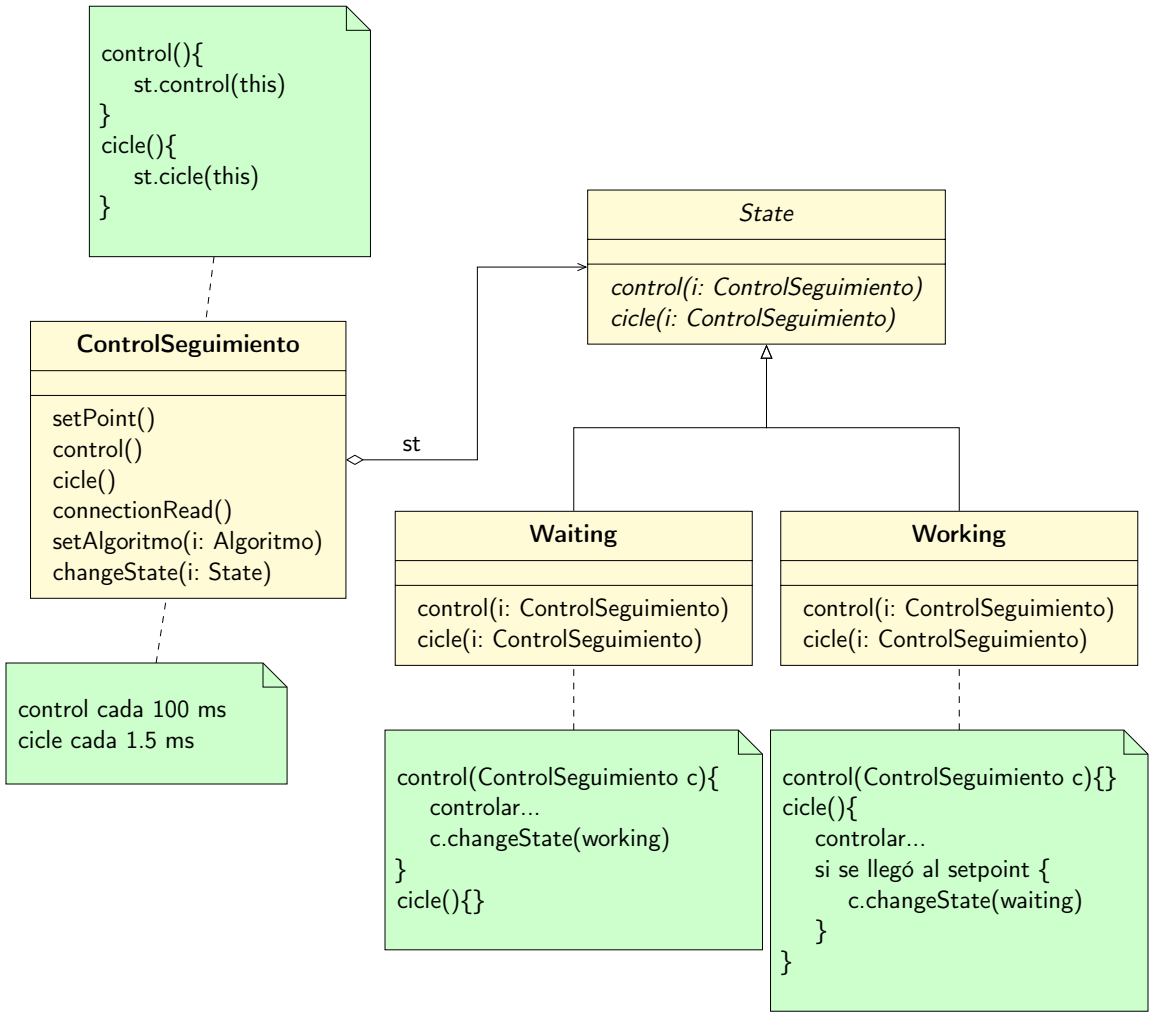
[2]La estabilización es crucial para evitar oscilaciones, reducir el tiempo de respuesta y minimizar sobrepasos, garantizando un control preciso y eficiente. Un sistema bien ajustado responde de manera estable ante perturbaciones externas y optimiza el consumo energético, mejorando la fiabilidad y el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

necesitan un seguimiento entre ciclos de control. A este módulo extendido lo llamaremos ControlSeguimiento.

Por un lado, se agrega una nueva interrupción periódica que invocará al método `cicle` del módulo ControlSeguimiento cada cierto intervalo de tiempo. Este período es menor al del ciclo de control y puede variar en función del actuador que se utilice.

Además, es necesario dotar al módulo de un estado, de modo que existan dos estados básicos: trabajando (*working*), cuando en el ciclo de control se ha definido un *set-point* y se está actuando activamente para alcanzarlo; y en espera (*waiting*), cuando el *set-point* fue alcanzado o no se estableció. Para lograrlo, se incorpora un nuevo método y se aplica el patrón *State*[3]. La estructura modular resultante de aplicar el patrón y extender Control puede observarse en la Figura 6.29.

Figura 6.29: Estructura módulo ControlSeguimiento.



[3]El uso de este patrón se explica en la Sección Máquinas de estado.

En consecuencia, cuando se ejecuta un ciclo de control por medio de la invocación de `Waiting::control()` y se determina la necesidad de aplicar un cambio en los actuadores controlados por ControlSeguimiento, su estado interno cambiará a Working. De este modo, comenzará a atender la interrupción periódica de menor periodo. Esta interrupción invoca a `Working::cicle()` método que verificará nuevamente si se ha alcanzado el *set-point*. De ser así, el sistema transicionará al estado de espera Waiting, donde se ignorará la interrupción de menor período (`Working::cicle()`), pero se continuará atendiendo la interrupción que marca el inicio del ciclo de control (`Waiting::control()`). En caso contrario, aplicará el cambio en el actuador sin transicionar, con el fin de que en una invocación futura de `Working::cicle()` la variable física se encuentre cerca del valor esperado. La estructura presentada permite un seguimiento preciso en la aplicación de cambios a los actuadores que lo requieran, como los motores paso a paso.

La documentación de la aplicación del patrón *State* en este caso está presente en la Figura 6.30.

Figura 6.30: Documentación 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	ControlSeguimiento **is** Contexto State **is** Estado Waiting **is** EstadoConcreto Working **is** EstadoConcreto

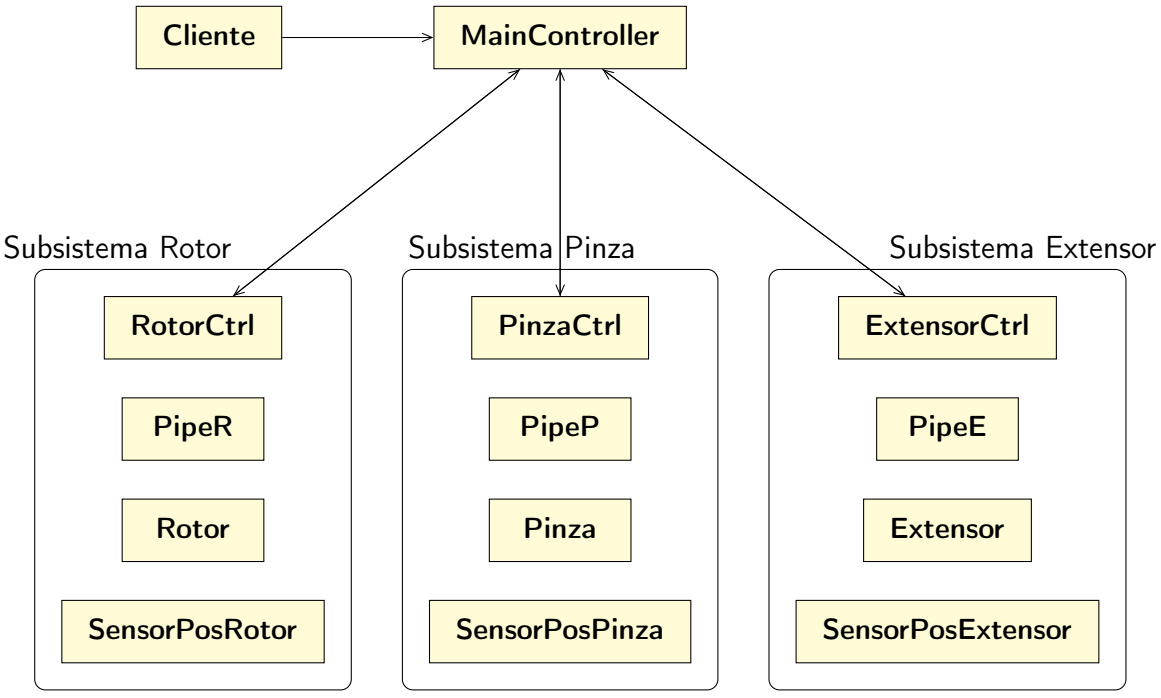
Por último, repasemos los beneficios de utilizar subsistemas de control. Este diseño permite el control de una variable específica del proceso, como la velocidad de giro de una rueda o la posición de un motor paso a paso. Si se requiere controlar otra propiedad de forma independiente, se hará con otro subsistema. El enfoque principal es encapsular el control de cada propiedad de manera individual. Esto permite que cada una pueda cambiar de forma aislada sin afectar al resto del sistema. Además, facilita la extensibilidad, ya que añadir nuevas propiedades físicas con sus respectivos actuadores y sensores solo requiere la creación de nuevos módulos. Este diseño proporciona una capa de abstracción que permite construir sobre los subsistemas un controlador general, el cual se encarga de involucrar y sincronizar a cada uno para llevar a cabo comportamientos más complejos.

Solución para el cambio

Esta solución se basa en aquella dada al problema del robot desmalezador en [Pom+24], pero aplicada al brazo robótico descripto en 6.8. Por lo tanto, se aplica el concepto de subsistemas que se introdujo previamente. Para ello primero se debe identificar las propiedades del mundo físico a controlar. Lo que se requiere es modificar la posición y el estado (abierto o cerrado) de la pinza del brazo. Para hacerlo se cuenta con distintos actuadores que intervienen diferentes propiedades físicas. En particular dos motores paso a paso y la pinza. Además, en los requisitos se especifica que se cuenta con una función que genera una lista de pasos con órdenes para cada actuador. De esta manera se define un subsistema de control por cada actuador. Estos susbistemas serán los encargados de llevar a cabo las órdenes generadas. Un ejemplo de orden es rotar brazo a 30°, esta hace referencia a actuar sobre el motor paso a paso de la base del brazo. En particular, estableciendo un *set-point* de posición igual a 30°.

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 6.31. PinzaCtrl

Figura 6.31: Diagrama de los componentes del sistema brazo robótico.



Se puede pensar que el gráfico guarda cierta similitud conceptual con el patrón *mediator* A.4. Este patrón encapsula cómo interactúan los módulos entre sí, con el fin de permitir variar su interacción de manera independiente. Un módulo Mediator coordina el trabajo de los College, en este caso los subsistemas. Los autores, en [Gam+95], indican que el patrón puede aplicarse cuando se tiene un conjunto de módulos que se comunican de manera compleja pero

fija. En este caso, la comunicación no es demasiado compleja, pero sí lo suficiente como para justificar el uso del patrón. En particular, el MainController establece los *set-points* de cada subsistema y desencadena el proceso de control en cada uno. A la inversa, los subsistemas informan cuándo alcanzan el *set-point* o cuándo encuentran un error. A su vez, los subsistemas no se comunican directamente entre sí: toda la interacción pasa por el MainController. Esto permite, por ejemplo, que la pinza pueda cerrarse únicamente cuando el brazo esté extendido, sin necesidad de que los módulos se comuniquen entre sí. Tradicionalmente, el módulo encargado de manejar el rotor le avisaría al módulo encargado de controlar la pinza que finalizó su tarea. Esto, sin embargo, genera un acoplamiento entre módulos, problema que ya se comentó previamente en esta tesina. En cambio, con este patrón, no sabe ni tiene noción de la existencia de PinzaCtrl o ExtensorCtrl. Toda la comunicación pasa a través de MainController, lo que desacopla los módulos controladores.

Ahora se detallarán los módulos que conforman cada componente, comenzando por los módulos básicos que se crearán para representar el hardware. Los módulos de la Figura 6.32 representan motores paso a paso, los cuales, para ser controlados, deben habilitarse mediante el métodos `enable` y luego establecer su dirección invocando los métodos `right` o `left`. Finalmente, es posible avanzar un paso utilizando el método `pulse`. Para alcanzar la posición deseada, puede ser necesario invocar varias veces dicho método. Este detalle será relevante al diseñar el subsistema encargado de controlar cada dispositivo. En cambio, para la pinza (ver Figura 6.33) se utiliza un dispositivo que posee únicamente dos estados: abierto o cerrado.

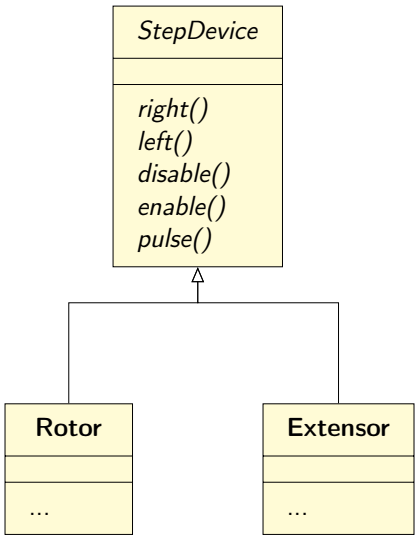


Figura 6.32: Actuadores paso a paso.

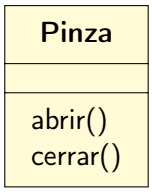
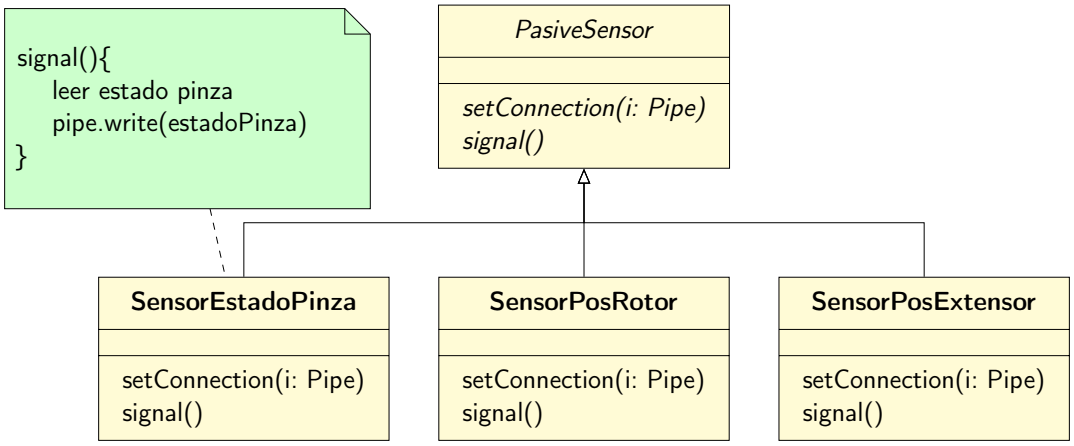


Figura 6.33: Interfaz módulo Pinza.

En la Figura 6.34 se encuentran definidos los sensores asociados a cada actuador, estos heredan del módulo sensor pasivo el cual provee dos métodos, `setConnection(i: Pipe)` el cual configura el Pipe por el cual se enviará la información obtenida del sensor cuando el otro método, `signal()` sea invocado.

Para completar los módulos que conforman un subsistema de control, falta mostrar cómo se define el módulo encargado de llevar el control de cada uno. En la Sección 6.8 mencionamos dos tipos de controladores: uno básico y otro más complejo que realiza un

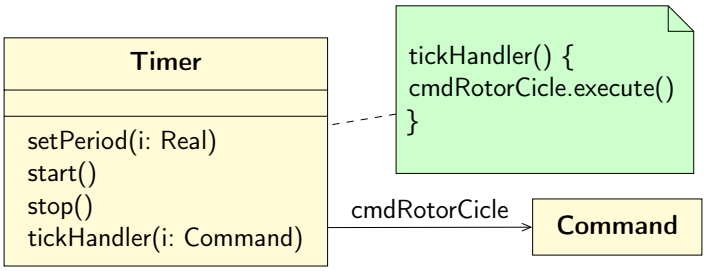
Figura 6.34: Sensores del brazo robótico.



seguimiento, utilizado cuando los actuadores involucrados en ese subsistema requieren un control exhaustivo, como los motores paso a paso. En este ejemplo contamos con dos motores paso a paso: uno en el rotor de la base y otro en el extensor del brazo. Además, tenemos la pinza, que no necesita un seguimiento exhaustivo para su operación; simplemente se le indica la posición a adoptar y el hardware se encarga de ejecutarla. A continuación, trataremos primero el módulo controlador que utilizaremos en el subsistema encargado del rotor RotorCtrl (Figura 6.39), siendo el del extensor de estructura similar.

Como tambien fue mencionado en la Sección 6.8, es necesario añadir una interrupción con un periodo menor al del ciclo de control y que está definido por el hardware a controlar. Por ejemplo, para el control del motor paso a paso que maneja la dirección del robot desmalezador en [Pom+24] se eligió $1.5ms$. Esta interrupción es lanzada por un dispositivo de hardware externo que actúa como temporizador. El módulo que encapsula su comportamiento es Timer y tiene la estructura mostrada en la Figura 6.35.

Figura 6.35: Módulo Timer.



El método `tickHandler()` ejecutará comandos siguiendo el patrón *Command* A.2 por cada subsistema que lo necesite. En particular, el comando es utilizada para desacoplar el Timer del módulo que controla el subsistema. Este uso del patrón fue explicado en la

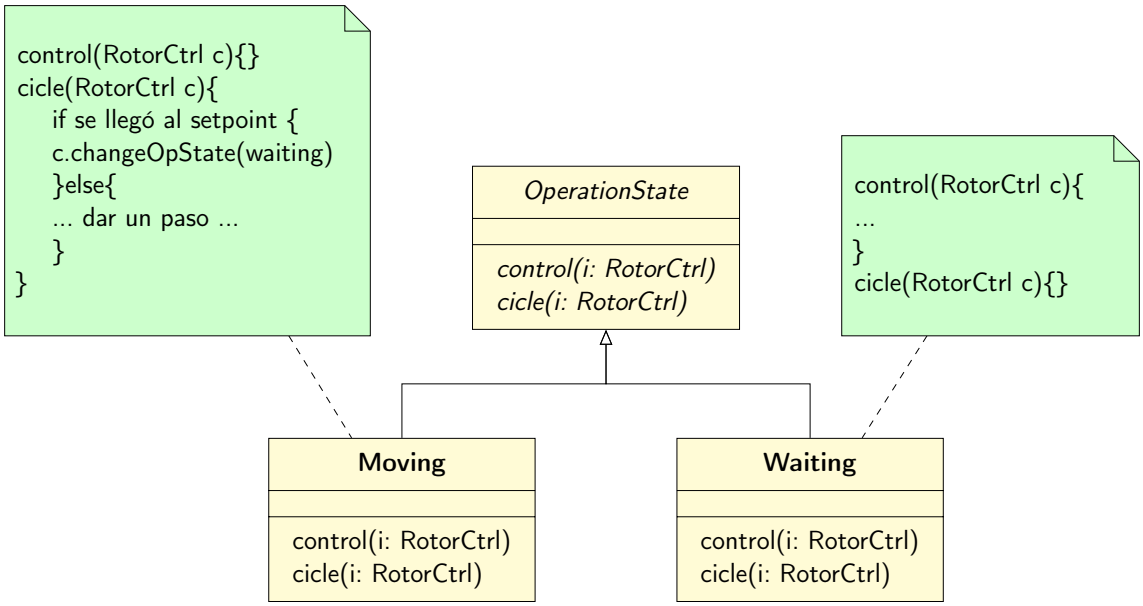
Sección 5.2. En la Figura 6.36 se puede observar la documentación de la aplicación del patrón *Command* para este caso.

Figura 6.36: Documentación de la aplicación del patrón *Command* para el desacople de ejecuciones que invoca 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 interrupción 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

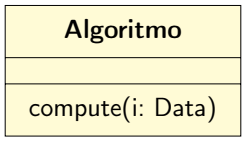
Una vez solucionada la invocación, introduciremos un nuevo módulo que será utilizado en RotorCtrl. Como vimos en la Sección 6.8, es necesario agregar un estado al controlador; para ello hacemos uso del patrón *State*. Este estado define los modos de operación de RotorCtrl: uno para atender el ciclo de control y otro para atender la interrupción de menor período, que utilizamos para dar seguimiento a la aplicación del cambio en el motor paso a paso. Para implementarlo, construimos los módulos mostrados en la Figura 6.37.

Figura 6.37: Módulos que forman parte del patrón *State* que son necesarios para complementar al módulo RotorCtrl.



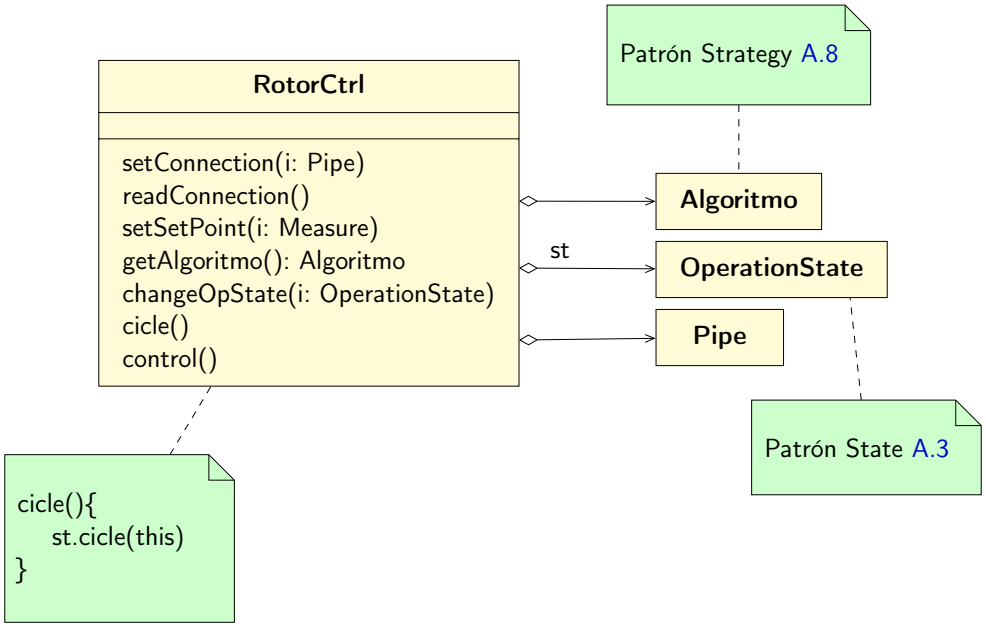
Siguiendo con la propuesta de creación de subsistemas, también se define el módulo Algoritmo que encapsula los cálculos necesarios para determinar cómo accionar sobre los actuadores. Su interfaz se puede observar en la Figura 6.38. Se puede aplicar el patrón *Strategy* A.8 de Gamma para prepararse ante cambios en el algoritmo. Es por eso que compondremos el módulo Algoritmo en el controlador.

Figura 6.38: Módulo complementario a RotorCtrl.



Con todos los módulos necesarios ya definidos, veamos ahora en la Figura 6.39 el diagrama de RotorCtrl.

Figura 6.39: Diagrama del módulo RotorCtrl.



Suponga que un cliente desea utilizar este subsistema que controla el rotor, ¿qué métodos debe ejecutar si ya se encuentra inicializado? En primer lugar, debe indicarle al sensor de posición que escriba el valor de lectura en Pipe. Para ello, se ejecuta el método `signal` del módulo . A continuación, RotorCtrl debe leer esta información desde Pipe y almacenarla; para esto se utiliza el método `readConnection`. Luego, se establece el *set-point* deseado y, finalmente, se invoca el método `control`, el cual delega su funcionamiento al `control` de su estado interno. Si el estado actual es Waiting, el método `control` tomará los valores actuales junto con el *set-point* y decidirá, utilizando el módulo Algoritmo, qué cambios aplicar en los actuadores, en este caso sobre Rotor. Dado que agregar todo el código para manejar un motor paso a paso puede requerir la ejecución de múltiples métodos, se puede construir un módulo que implemente el patrón *Command* A.2, encapsulando qué métodos deben invocarse. Este comando permitirá tanto cambiar la dirección de giro (ej. `StepDevice::left()`, Figura 6.32) como avanzar un paso mediante la ejecución de `StepDevice::pulse()` (Figura 6.32). Como el motor puede girar en ambos sentidos, es posible definir dos comandos: uno para cada dirección (y). Un ejemplo del método `control()` para el caso del subsistema del rotor en el estado de operación Waiting se muestra en el Código 6.31 (recordando que el *set-point* del subsistema del rotor corresponde a una medida en grados).

Código 6.31: Ejemplo de implementación del método control del módulo Waiting.

```
control(Controller controller) {
    dif = ...se usa Algoritmo para computar diferencias entre setpoint y valor
        actual...
    if abs(dif) > LIMIT_ACCEPT and dif > 0 {
        ...se configura sentido de giro a la derecha...
    }
    if abs(dif) > LIMIT_ACCEPT and dif < 0 {
        ...se configura sentido de giro a la izquierda  ...
    }
    ...se envia pulso para que el motor de un paso...

    //se cambia al estado Moving ejecutando:
    controller.changeOperationState(moving)
}
```

Con la presentación del módulo finalizamos la definición del subsistema rotor. Además de este, deben crearse los módulos correspondientes al subsistema extensor, el cual resulta similar al del rotor.

Por último, se debe definir el subsistema pinza, que es más simple que los anteriores, ya que no requiere de un seguimiento entre ejecuciones del ciclo de control. En consecuencia, puede emplear el controlador básico presentado en la Sección 6.8, sin necesidad de incorporar estados, nuevas interrupciones ni el método `cicle`.

Ya definidos los subsistemas, podemos continuar con la construcción de los módulos que conforman la Figura 6.31. En este punto, resta abordar el módulo MainController, encargado de proveer una interfaz al cliente y, al mismo tiempo, coordinar el funcionamiento de cada subsistema. Recordando los requerimientos, sabemos que debe existir una función capaz de generar una secuencia de pasos, donde cada paso implique una acción sobre los tres actuadores. ¿Cómo diseñar este comportamiento? Una alternativa consiste en utilizar un iterador siguiendo el patrón *Iterator* A.7 para recorrer los pasos generados por la función, y combinarlo con el patrón *Command* A.2 para la ejecución de cada uno de ellos. En particular, se adopta una de las variantes mencionadas por Gamma [Gam+95], en la cual la ejecución de un comando desencadena automáticamente la ejecución de los siguientes, permitiendo encadenar acciones de manera ordenada y desacoplada.

Para esto, debemos definir el iterador, al que llamaremos Steps, cuya interfaz se muestra en la Figura 6.40. Su correspondiente documentación se encuentra en la Figura 6.41. En la Figura 6.42 se observa la estructura de módulos resultante de aplicar el patrón *Command* en su variante. Además, en la Figura 6.43 se presenta la documentación asociada, y en la Figura 6.32 se incluye un ejemplo de una posible implementación del módulo OrdenRotor.

Figura 6.40: Interfaz módulo Steps.

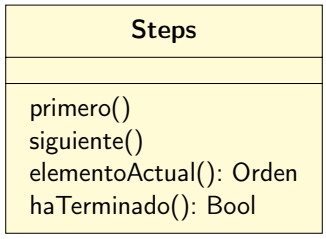
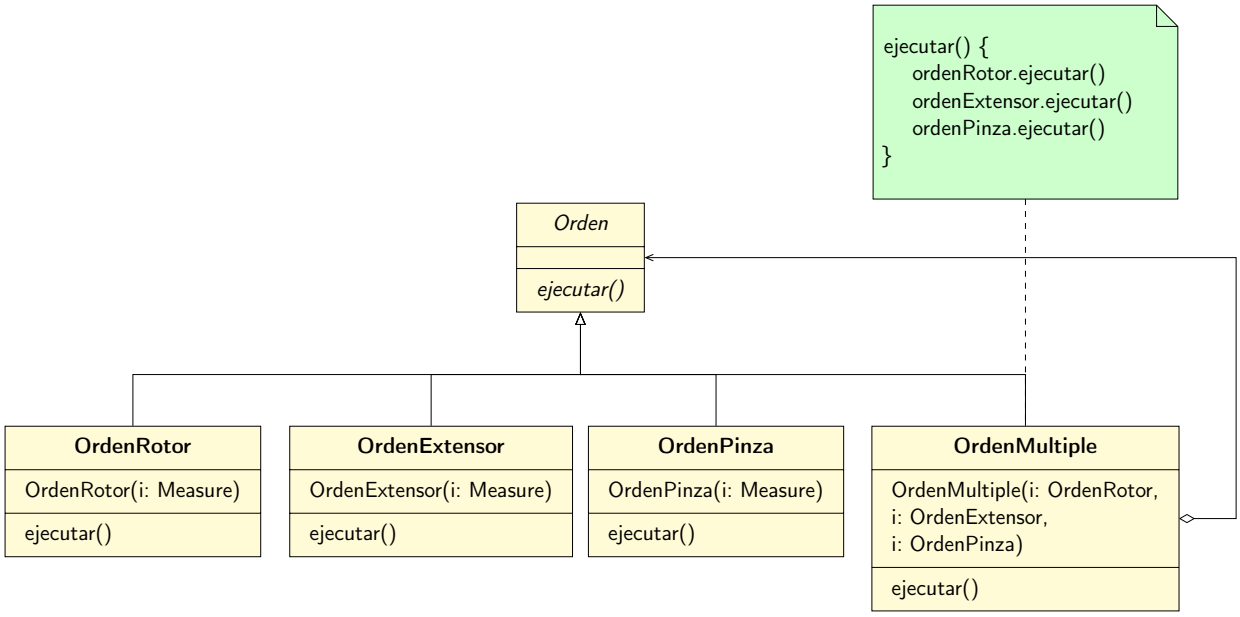


Figura 6.41: Documentación de aplicación del patrón *Iterator*.

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

Figura 6.42: Interfaces de las ordenes de ejecución para cada subsistema.



Código 6.32: Ejemplo de implementación del módulo OrdenRotor

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

Figura 6.43: Documentación de la aplicación del patrón *Command* para el desacople de órdenes 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

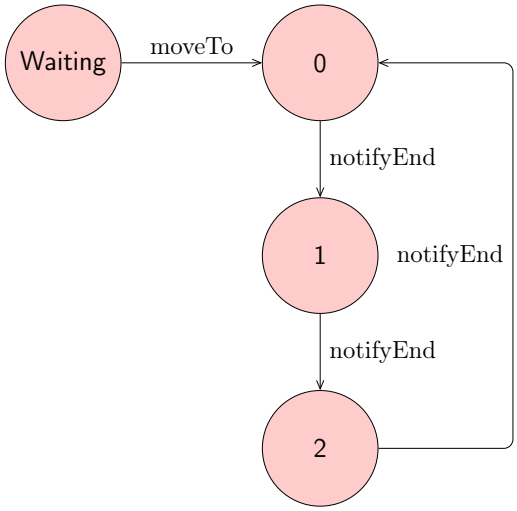
Siguiendo este diseño se obtuvo un iterador (Figura 6.40) que almacena todas las órdenes generadas por la función provista `graspAt(i: Coordenadas)`. Además, se permite recorrerlas de manera secuencial y haciendo uso de la orden múltiple, se logra fácilmente ejecutar un paso en todos los actuadores. Logramos, almacenar el procedimiento a realizar y desacoplar cómo se pone en marcha el ciclo de control en cada subsistema.

Hasta ahora, todo lo que se ha hecho fue aplicar el concepto de subsistemas. Falta entonces introducir, finalmente, el módulo MainController. Sin embargo, para poder cumplir con los requerimientos particulares del ejemplo, es necesario explicar una última solución. Dado que se debe ejecutar un paso a la vez, es preciso esperar a que todas las órdenes enviadas en el paso anterior a los subsistemas se encuentren finalizadas.

Para ello se hace uso del patrón *state* (véase la documentación de su aplicación en la Figura 6.46), al cual el MainController delegará el comportamiento de los métodos `moveTo` y `notifyReady`. La idea es modificar el comportamiento de estos métodos según el estado en el que se encuentre el sistema. Es decir, por cada orden enviada y completada se cambia de estado, llevando un registro de la cantidad de acciones finalizadas. Solo cuando se hayan completado las tres órdenes será posible ejecutar un nuevo paso. Este funcionamiento puede

observarse en el gráfico de estados de la Figura 6.44.

Figura 6.44: Transiciones de estados del MainController



`moveTo` y `notifyOrder` son dos métodos de la interfaz de MainController (que presentaremos más adelante en la Figura 6.45). El primero es utilizado por los clientes del sistema: recibe las coordenadas en las que se desea posicionar el brazo y desencadena el funcionamiento de todos los subsistemas. El segundo, en cambio, es un método empleado por los subsistemas para comunicar que alcanzaron el *set-point*. Teniendo en cuenta la función de cada método, veamos cómo varía su comportamiento según el estado.

El estado Waiting no implementa `notifyOrder`, mientras que los estados 0, 1 y 2 no implementan `moveTo` (no tiene sentido iniciar un nuevo movimiento si otro está en curso). En cambio, Waiting sí implementa `moveTo`: este calcula y genera el iterador de pasos, ejecuta el primero y cambia el estado a 0. Posteriormente, con cada `notifyEnd` los subsistemas informan al MainController que finalizaron la ejecución de la orden, y en cada notificación se transiciona a un nuevo estado. Una vez ejecutado `moveTo`, se pasa al estado 0; al recibir un `notifyEnd` se transiciona al estado 1, y de la misma forma hasta llegar al estado 2. Cuando se está en el estado 2 y se recibe un `notifyEnd`, se ejecuta un nuevo paso si aún quedan pasos en el iterador; en caso contrario, se transiciona nuevamente a Waiting, dado que la secuencia de ejecución ha finalizado.

Está claro que, para sostener este comportamiento, los subsistemas deben responder a estas necesidades. Es decir, cada uno debe invocar el método `notifyEnd` del MainController como un *callback* únicamente cuando hayan terminado de ejecutar la orden, es decir, cuando alcancen el *set-point*. Los subsistemas son responsables de determinar cuándo un paso se ha completado. En el caso de aquellos que requieren un seguimiento exhaustivo (los que usan un ControlSeguimiento), al llegar al *set-point* se transiciona del estado Moving al estado Waiting; en ese punto puede incluirse la llamada a `notifyEnd`. En el caso del subsistema de la pinza, que no requiere seguimiento, la llamada puede realizarse inmediatamente después de aplicar el cambio en el módulo Pinza.

La estructura de módulos necesaria para la inclusión de estos estados en el MainController puede observarse en la Figura 6.45, y la aplicación correspondiente del patrón *state*, en la Figura 6.46.

Figura 6.45: Diagrama del módulo MainController.

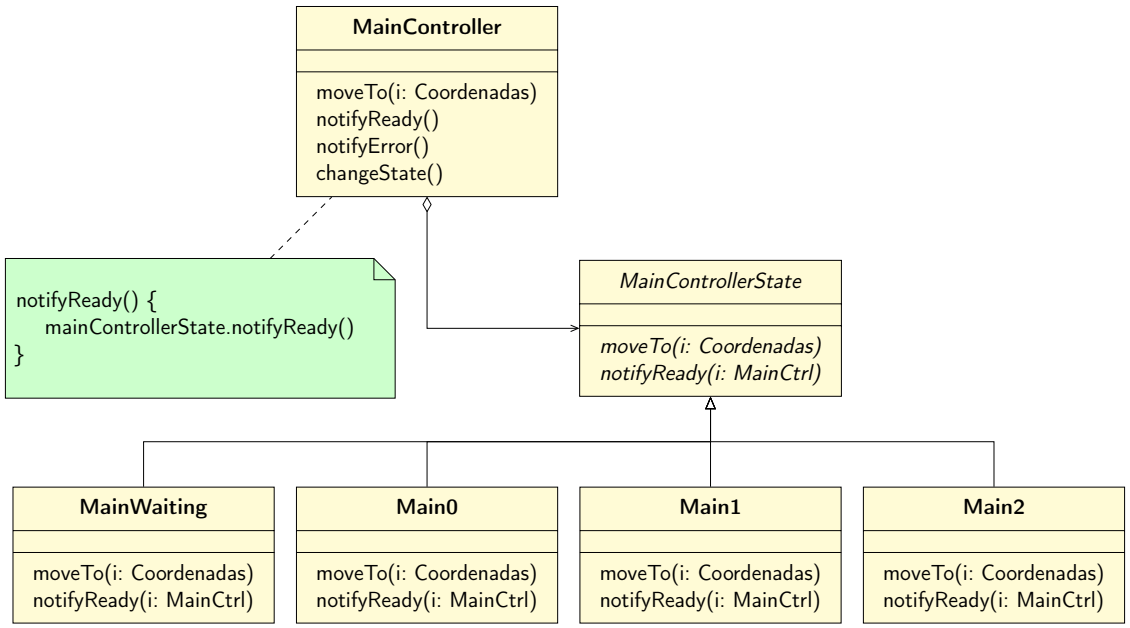


Figura 6.46: Documentación de la aplicación del patrón *State* para el manejo de ejecución de pasos completos.

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

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 probables están encapsulados, por lo que el control de calidad se reduce a verificar módulos individuales y no secciones de código enteras.

Capítulo 7

Conclusión

A lo largo de este trabajo se exploró un conjunto de problemáticas recurrentes en el desarrollo de sistemas embebidos de control, poniendo en evidencia cómo la falta de preparación ante el cambio puede derivar en diseños rígidos, difíciles de mantener y costosos de evolucionar. Estos problemas, comunes en la práctica profesional, no se limitan a cuestiones de bajo nivel como el manejo de periféricos o la eficiencia en tiempo de ejecución, sino que se relacionan con la organización misma del software y con la capacidad de anticiparse a los cambios en el hardware, en los requerimientos funcionales o en el entorno de uso.

Frente a este panorama, se adoptó la perspectiva de la IS centrada en el diseño para el cambio, siguiendo las ideas fundamentales propuestas por David L. Parnas. Esta visión propone identificar explícitamente los ítems de cambio y encapsularlos dentro de módulos bien definidos, de modo que una modificación futura impacte en la menor cantidad posible de componentes. Bajo este enfoque, se analizaron diferentes ejemplos tomados tanto de la literatura [Dou11] como del trabajo realizado sobre el robot desmalezado [Pom+24], y se propusieron soluciones basadas en estructuras modulares y patrones de diseño.

En particular, se mostró cómo patrones clásicos como *State*, *Strategy*, *Adapter*, *Command*, *Decorator* y *Mediator* permiten abordar de forma sistemática problemas habituales en el software embebido. El patrón *State* resultó útil para representar transiciones claras entre estados de un dispositivo físico, evitando estructuras rígidas con múltiples sentencias condicionales. *Strategy* ofreció un mecanismo para variar criterios de decisión sin afectar la estructura global del sistema. *Adapter* permitió integrar librerías de distintos fabricantes sin necesidad de reescribir todo el código cliente, *Command* brindó una manera ordenada de manejar eventos e interrupciones. *Decorator* posibilitó añadir funcionalidades dinámicamente a módulos y *Mediator* organizó la coordinación entre subsistemas para lograr objetivos comunes. Estos ejemplos evidencian que los patrones de diseño, originalmente concebidos para software de propósito general, pueden trasladarse con éxito al ámbito embebido siempre que se adapten a sus restricciones particulares de recursos y desempeño.

Otro aspecto importante del trabajo fue la discusión del estilo arquitectónico de control de procesos, el cual permitió enmarcar el diseño de los sistemas embebidos de control en una visión más amplia, donde las actividades de adquisición de datos, procesamiento y actuación se organizan en una estructura sistemática. Este estilo arquitectónico ofrece un marco conceptual que facilita la integración de nuevos dispositivos y algoritmos sin afectar la lógica central de control. Asimismo, la estructura modular para la obtención de información se destacó como

una estrategia clave para desacoplar el hardware de los mecanismos de cómputo, permitiendo que los datos se capturen, almacenen y procesen de manera flexible y reutilizable.

El aporte central de esta tesina puede interpretarse como la construcción de una caja de herramientas conceptual para desarrolladores de sistemas embebidos que no tengan mucha experiencia en el diseño de software. Esta caja de herramientas no pretende ofrecer recetas únicas ni soluciones rígidas, sino marcos de referencia que permitan razonar ante la necesidad de cambio. La idea es que, frente a un problema práctico, el diseñador disponga de un repertorio de enfoques probados que le permitan seleccionar, adaptar y combinar las técnicas más adecuadas a cada contexto. Así, el diseño para el cambio se transforma en un hábito metodológico que guía las decisiones y no en una actividad reactiva que aparece recién cuando los problemas ya se han manifestado.

El trabajo también resalta la importancia de integrar prácticas de la IS en el campo de los sistemas embebidos. Tradicionalmente, el diseño en este ámbito ha estado orientado de manera casi exclusiva a la eficiencia y al aprovechamiento máximo del hardware disponible. Sin embargo, los ejemplos desarrollados demuestran que un diseño flexible y mantenible no solo no va en contra de la eficiencia, sino que puede convertirse en una ventaja competitiva al prolongar la vida útil de los productos, facilitar la actualización tecnológica y reducir los costos de mantenimiento a largo plazo.

En síntesis, vimos que el diseño para el cambio constituye un enfoque valioso en el desarrollo de software embebido. Su aplicación consciente, junto con el apoyo de estilos arquitectónicos como el control de procesos y estructuras modulares, contribuye a construir sistemas más robustos, escalables y preparados para la evolución constante del entorno tecnológico. Al mismo tiempo, ofrece a los desarrolladores una forma de organizar su trabajo con mayor previsión, reduciendo el riesgo de introducir errores y favoreciendo la reutilización de componentes.

Un aspecto que merece ser destacado es que, durante el análisis, muchos de los supuestos patrones de diseño encontrados en la literatura no respondían en realidad a los criterios de la IS, sino que resultaban ser patrones idiomáticos, ligados a características específicas de implementación o a convenciones de un lenguaje en particular. Esto muestra que, en la práctica, aquello que inicialmente considerábamos "patrones de diseño" no siempre cumplía con los principios de modularidad, ocultación de información y preparación para el cambio propuestos por la IS. Reconocer esta diferencia no es un detalle menor: permitió replantear los objetivos de la tesina, orientar el trabajo hacia soluciones que sí se alineen con las buenas prácticas de la disciplina y, al mismo tiempo, resalta la necesidad de que el diseño en sistemas embebidos no se quede en soluciones idiomáticas de bajo nivel, sino que adopte un enfoque más robusto y fundamentado en la Ingeniería de Software.

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.

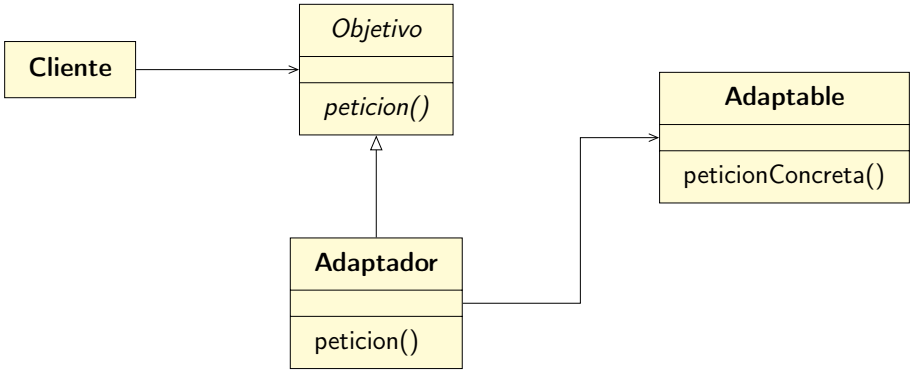
Participantes

- 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 replicarse 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 re-ejecutarlos mediante la operación **Execute**.

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

Participantes

- *Orden*
  Declara una interfaz para ejecutar una operación.

- OrdenConcreta
  Define una asociación entre un módulo receptor (Receiver) y una acción. Implementa el método Execute invocando las operaciones correspondientes en el receptor.

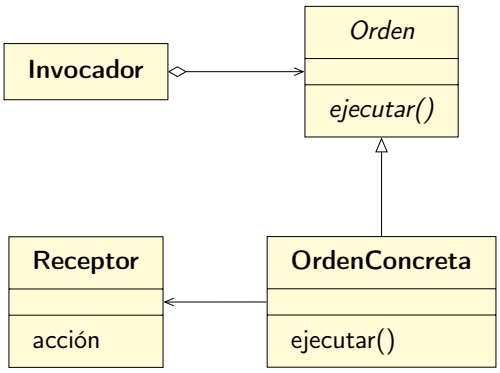
- Cliente
  Utiliza OrdenConcreta y configura su receptor.

- Invocador
  Solicita al comando que lleve a cabo la solicitud.

- Receptor
  Conoce cómo realizar las operaciones asociadas con la ejecución de una solicitud. Cualquier clase puede actuar como un receptor.

Estructura

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



A.3 State

Intención

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

Aplicabilidad

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

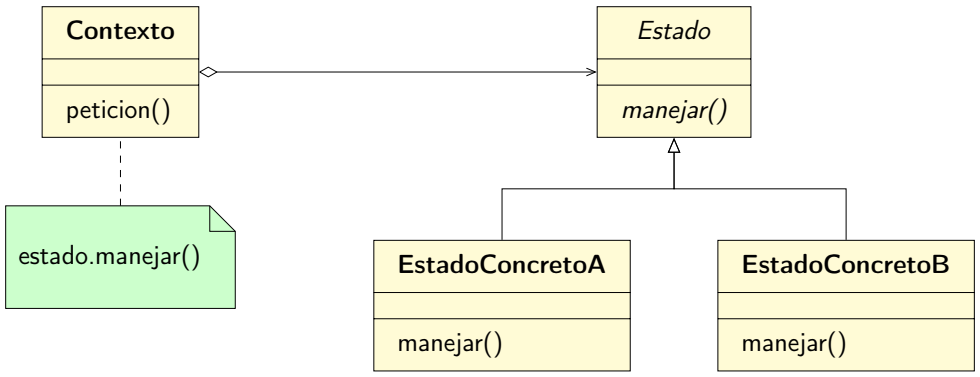
Participantes

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

- EstadoConcreto
  Cada subclase implementa un comportamiento asociado con un estado del contexto.

Estructura

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



A.4 Mediator

Intención

Define un módulo 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.

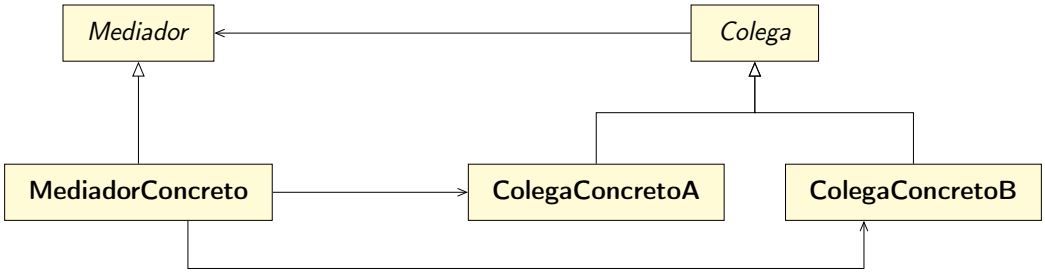
Participantes

- 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 sé 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.

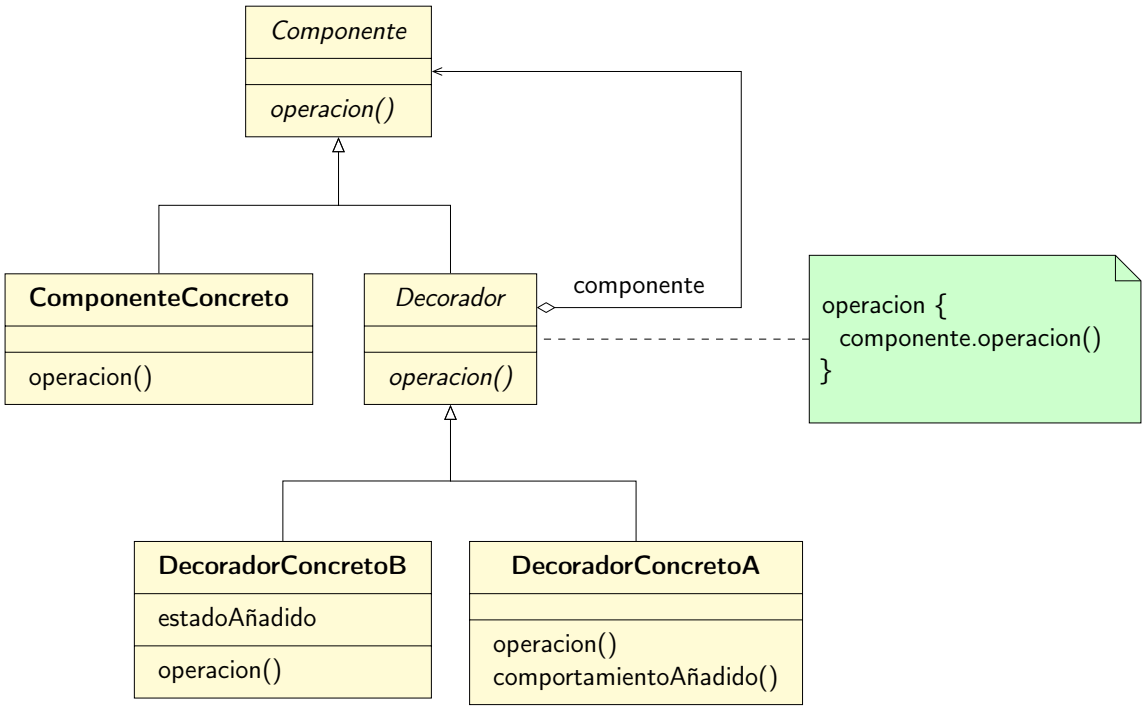
Participantes

- 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:

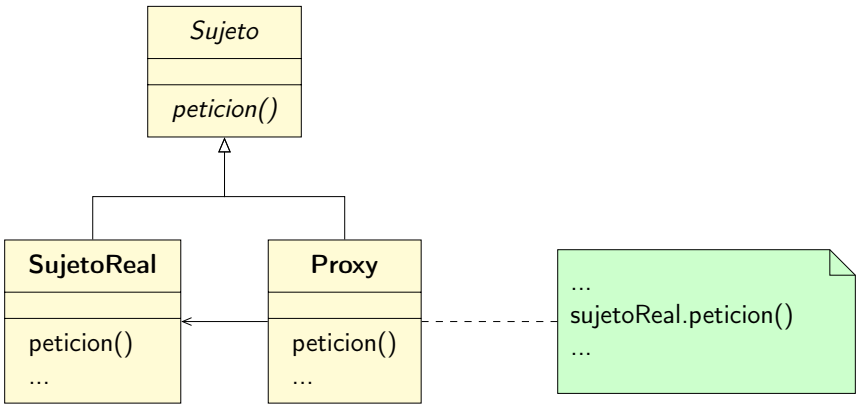
   - Contar el número de referencias a la instancia real para que pueda ser liberado automáticamente cuando no queden más referencias (también llamado punteros inteligentes).
   - Cargar una instancia persistente en memoria cuando se referencia por primera vez.
   - Verificar que la instancia real esté bloqueado antes de acceder a ella, para asegurar que ninguna 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 si las interfaces de 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 y **Proxy**, de modo que un Proxy se pueda usar en cualquier lugar donde se espere un .
- RealSubject Define el objeto real que el proxy representa.

Estructura

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



A.7 Iterator

Intención

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

Aplicabilidad

Úsese el patrón Iterador:

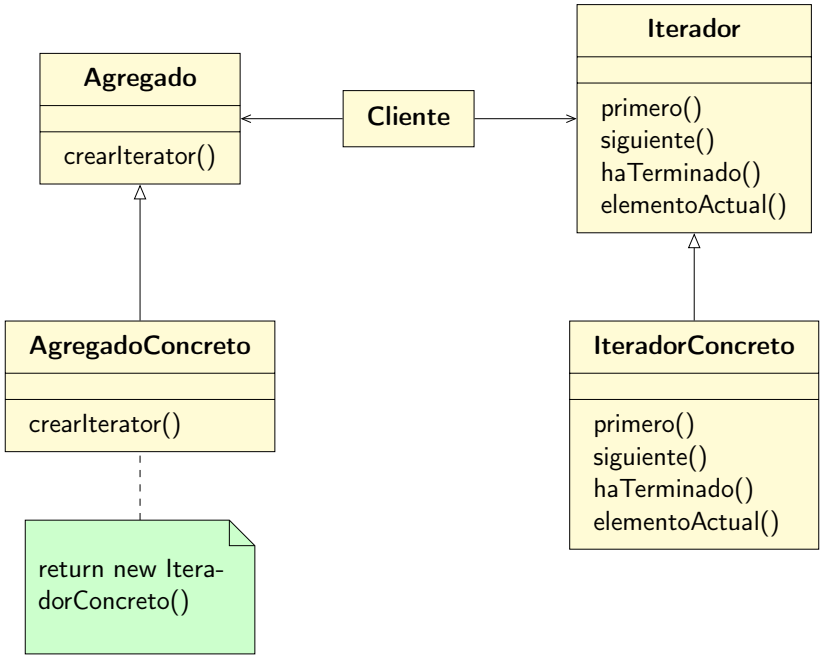
- Para acceder al contenido de un módulo agregado sin exponer su representación interna.
- Para permitir varios recorridos sobre objetos agregados.
- Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la iteración polimórfica).

Participantes

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

Estructura

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



A.8 Strategy

Intención

Define una familia de algoritmos, encapsula cada uno de ellos y hace que sean intercambiables. Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.

Aplicabilidad

Use el patrón Strategy cuando:

- Muchos módulos relacionados que solo difieren en su comportamiento. Las estrategias ofrecen una manera de configurar una clase con uno de varios comportamientos.
- Cuando se necesita diferentes variantes de un algoritmo. Por ejemplo, se podría definir algoritmos que reflejen diferentes compromisos entre espacio y tiempo. Las estrategias pueden usarse cuando estas variantes se implementan como una jerarquía de clases de algoritmos.
- Un algoritmo utiliza datos que los clientes no deberían conocer. El patrón evita exponer estructuras de datos complejas y específicas del algoritmo.

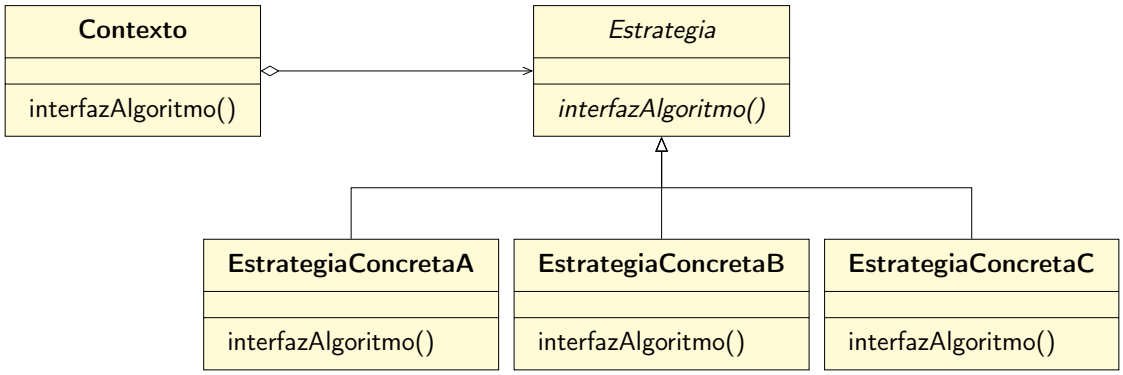
- Un módulo define muchos comportamientos que aparecen como múltiples sentencias condicionales en sus operaciones.

Participantes

- Estrategia Declara una interfaz común para todos los algoritmos soportados. El Contexto utiliza esta interfaz para invocar el algoritmo definido por una EstrategiaConcreta.
- EstrategiaConcreta Implementa el algoritmo utilizando la interfaz Estrategia.
- Contexto
  - Está compuesto con un módulo EstrategiaConcreta
  - Mantiene una referencia a un módulo Estrategia.
  - Puede definir una interfaz que permita a Estrategia acceder a sus datos.

Estructura

Figura A.8: Estructura patrón **Estrategia**



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. 51, 65

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. 48

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. 86, 89

ADC El ADC (acrónimo de convertidor analógico digital) es un dispositivo electrónico que convierte una señal analógica en una señal digital. 25

API Interfaz de Programación de Aplicaciones. 33

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.. 82

callback Es una función A que se usa como argumento de otra función B. De esta forma, al llamar a B, esta ejecutará A.. 112

checksum Un checksum, o suma de verificación, es un valor numérico calculado a partir de un conjunto de datos que se utiliza para verificar la integridad de la información durante su transmisión o almacenamiento. Su propósito es asegurar que los datos no se hayan alterado o dañado de forma accidental.. 82

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. 25

DBOI Diseño basado en ocultación de la información. 31, 32, 54

DC Corriente continua. 48, 54

DRV8838 Controlador para un único motor de corriente continua (DC) con escobillas. 48–50, 53, 54

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). 19

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. 20

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. 25

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. 61, 63, 64, 86

IS Ingeniería de Software. 13–15, 17, 19–21, 29, 34, 47, 56, 59, 62, 74, 78, 84, 90, 109, 110

ISR Interrupt Service Routine. 26

magnetrón Un magnetrón es un tubo electrónico al vacío que genera microondas mediante la interacción de electrones acelerados en un campo eléctrico y confinados por un campo magnético. Es el componente central de los hornos de microondas, siendo el responsable de producir las ondas electromagnéticas que calientan los alimentos. También se utiliza en equipos de radar para generar pulsos de alta potencia.. 67, 70, 72

MCU Unidad de Microcontrolador. 13, 14, 25, 26, 32, 48–50, 84

microcontrolador Circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria . 49

motor paso a paso Un motor paso a paso es un tipo de motor eléctrico que convierte señales eléctricas en movimientos mecánicos discretos y precisos. A diferencia de los motores tradicionales, se mueve en pasos fijos en lugar de girar continuamente, lo que permite controlar con exactitud el ángulo de rotación sin necesidad de sensores. Esto lo hace ideal para aplicaciones que requieren posicionamiento preciso, como impresoras 3D, robótica, CNC y automatización industrial.. 44, 89, 90, 93, 95–98, 101

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. 23, 24

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. 27, 93

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. 25, 87

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. 20, 21

RPM revoluciones por minuto. 40, 61

servomecanismos Un servomecanismo es un sistema de control automático que utiliza retroalimentación para regular con precisión la posición, velocidad o aceleración de un actuador, como un motor. Está compuesto típicamente por un motor, un sensor (como un potenciómetro o encoder) y un controlador que ajusta la señal de entrada en función del error entre la posición deseada y la real. Los servomecanismos se usan ampliamente en robótica, automatización industrial y sistemas embebidos donde se requiere movimiento controlado y exacto.. 27

Referencias

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

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

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

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

[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.

[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.

[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).

[GJM03] 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.

[Tam+12] M. Tamura, T. Kamiyama, T. Soeda, M. Yoo y T. Yokoyama. "A Model Transformation Environment for Embedded Control Software Design with Simulink Models and UML Models". En: (2012).

[TBJ19] M. Tkáčik, A. Březina y S. Jadlovská. "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).

[Her+13] C. Hernández, J. Bermejo-Alonso, I. López y R. Sanz. "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.

[MRT03] M. Montemerlo, N. Roy y S. Thrun. "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. URL: https://www.fceia.unr.edu.ar/ingsoft/diseno-a.pdf.

[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.

[Pom25] L. Pomponio. *Introducción a 2MIL - Lenguaje de Interconexión de M´odulos*. 2025. URL: https://www.fceia.unr.edu.ar/ingsoft/intro2MIL.pdf.

[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.

[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).

[GIM18] G. Giacomino, J. F. Imsand y S. Martinez. "Desarrollo e implementación de funciones de navegación autónoma para un prototipo de robot pulverizador". En: (2018).

[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)*.

[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).

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