Universidad Autónoma de Baja California

Facultad de Ciencias Químicas e Ingeniería

Maestría y Doctorado en Ciencias e Ingeniería

Patrones de Diseño para Sistemas Embebidos Aplicados a Internet de las Cosas

TESIS

Que para obtener el grado de:
Maestro en Ingeniería

Presenta:
Ricardo Castro Gonzáles

Director de Tesis:
Dr. Mauricio Alonso Sánchez Herrera

Co-Director de Tesis:
Dra. Thelma Violeta Ocegueda Miramontes

Tijuana, Baja California, México

Junio 2024

UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA
FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

Folio No.362
Tijuana, B.C., a 11 de junio, 2024

C. RICARDO CASTRO GONZÁLES
Pasante de: Maestría en Ingeniería
Presente

El tema de trabajo y/o tesis para su examen profesional, en la Opción TESIS.

Es propuesto, por los C. Dr. Mauricio Alonso Sánchez Herrera y Dra. Thelma Violeta Ocegueda Miramontes.

Quienes serán los responsables de la calidad del trabajo que usted presente, referido al tema: *"Patrones de Diseño para Sistemas Embebidos Aplicados a Internet de las Cosas".*

El cual deberá usted desarrollar, de acuerdo con el siguiente orden:

I. INTRODUCCIÓN
II. ANTECEDENTES
III. ANÁLISIS DE APLICACIONES IOT
IV. ANÁLISIS Y SELECCIÓN DE PATRONES
V. USO DE PATRONES EN APLICACIONES
VI. PROTOTIPOS Y RESULTADOS
VII. CONCLUSIONES
VIII. TRABAJO FUTURO

M.C. Roberto Alejandro Reyes Martínez
Director

Dra. Ana Alejandra Ramírez Rodríguez
Subdirectora

UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA
POR LA REALIZACIÓN PLENA DEL SER
FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA CAMPUS TIJUANA

Dr. Mauricio Alonso Sánchez Herrera
Director De Tesis

Dra. Thelma Violeta Ocegueda Miramontes
Co-Directora De Tesis

Resumen

La tecnología y el internet han sido de gran impacto para la sociedad actual. Han transformado la comunicación en la sociedad, posibilitando el contacto con personas del otro lado del mundo, e incluso permiten el acceso a dispositivos en lugares remotos con acceso a internet. A consecuencia de esto se origina el Internet de las Cosas (IoT). El IoT es una interconexión de dispositivos que se comunican con el mundo real, estos tienen la capacidad de conectarse a internet, y se caracterizan por ser sistemas embebidos. Se estima que en los próximos años aumente de manera sustancial el número de dispositivos enlazados al IoT. Con este aumento, también incrementa la complejidad de los sistemas que componen, debido a que la funcionalidad se está llevando a cabo mediante software ejecutado en procesadores embebidos de propósito general. El desarrollo de software embebido escasea de estandarización, pues existen numerosas técnicas y herramientas que ayudan en el desarrollo de software, tales como los patrones de diseño, sin embargo estos no son ampliamente utilizados para el software embebido en aplicaciones IoT. El objetivo del presente trabajo fue adaptar patrones de diseño para el desarrollo de software dedicado a sistemas embebidos aplicados a IoT. Se realizó una revisión exhaustiva del estado del arte en el campo de patrones de diseño y aplicaciones de IoT. A continuación se categorizaron los diferentes tipos de aplicaciones de IoT, posteriormente se seleccionaron los patrones de diseño a utilizar y de se adaptaron a estas categorías, para finalizar se elaboraron prototipos por medio de los patrones de diseño y se evaluó su desempeño.

Se consiguió adaptar varios patrones de diferentes categorías, donde en algunos casos se encontró más de un patrón para resolver un problema. Gracias a la categorización de las aplicaciones IoT, se logró utilizar de manera eficaz un patrón de diseño en múltiples aplicaciones. También se detectaron casos en donde varios patrones se utilizan en conjunto para un mayor efecto. Se puede concluir que el utilizar patrones de diseño logra reducir el tiempo requerido para hacer modificaciones en las aplicaciones de sistemas embebidos en IoT. Estos son una herramienta que aunque no ha sido ampliamente adoptada, tiene un alto grado de compatibilidad con los problemas presentados en las aplicaciones de IoT.

Abstract:

Technology and the internet have had a great impact on today's society. They have transformed communication in society, enabling contact with people from the other side of the world, and even allowing access to devices in remote locations with internet access. As a result, the Internet of Things (IoT) arises. The IoT is an interconnection of devices that communicate with the real world, they have the ability to connect to the internet, and they are characterized by being embedded systems. It is estimated that the number of devices linked to the IoT will increase substantially in the coming years. With this increase, the complexity of the systems that make it up also increases, as the functionality is being carried out through software executed on general-purpose embedded processors. The development of embedded software lacks standardization, as there are numerous techniques and tools that assist in software development, such as design patterns, however, these are not widely used for embedded software in IoT applications. The aim of this work was to adapt design patterns for the development of software dedicated to embedded systems applied to IoT. A comprehensive review of the state of the art in the field of design patterns and IoT applications was performed. Then, the different types of IoT applications were categorized, and the design patterns to be used were selected and adapted to these categories. Finally, prototypes were developed using the design patterns, and their performance was evaluated. Several patterns from different categories were successfully adapted, in some cases more than one pattern was found to solve a problem. Thanks to the categorization of IoT applications, a design pattern was effectively used in multiple applications. Cases where several patterns were used together for a greater effect were also detected. It can be concluded that using design patterns reduces the time required to make changes in embedded systems applications in IoT. These are a tool that, although not widely adopted, has a high degree of compatibility with the problems presented in IoT applications.

Índice General

Índice de figuras

Índice de tablas

Capítulo 1: Introducción

1.1 Planteamiento del problema

El Internet ha cambiado la manera en la que las computadoras y la comunicación funcionan en el mundo, ha facilitado una nueva forma de relacionarnos a través de dispositivos electrónicos que otorgan a los usuarios capacidades extendidas para realizar contacto con otros individuos, creando un sistema que estableció una normativa de globalización, una red capaz de conectar cualquier área geográfica mediante dispositivos con la capacidad de comunicarse a dicha red dando cabida al Internet de las Cosas (IoT). IoT representa los dispositivos que podemos conectar a Internet proporcionando acceso a la comunicación expansiva entre las personas, dichos dispositivos son caracterizados por sus sensores embebidos que les permite comunicación entre sí y con el mundo que los rodea, teniendo una amplia gama de aplicaciones que van desde electrodomésticos hasta herramientas industriales[8][9][14].

El IoT proporciona un entorno en el cual los dispositivos se encuentran conscientes de su contexto, para ello hacen uso de sistemas embebidos, lo que facilita el desempeño eficiente y la automatización de procesos. Diferentes autores difieren en el pronóstico de cuántos dispositivos estarán conectados a la red IoT en la siguiente década, pero todos coinciden en que el aumento será significativo, pasando de 8.7 billones en el 2020 a un estimado de entre 25 a 125 billones en el 2030 [11][12].

Además del incremento en la complejidad de las redes IoT debido al aumento de los dispositivos conectados, se debe considerar que los sistemas se han vuelto más complejos con el paso de los años; debido a que cada vez una mayor cantidad de funcionalidades han dejado de ser implementadas en hardware dedicado para hacerlo mediante software ejecutado en procesadores embebidos de propósito general[1] .

Cuando se planea el desarrollo de un sistema embebido, el software se desarrolla tomando en cuenta las características específicas del hardware para el que será desarrollado. Con el paso del tiempo, los métodos, técnicas y herramientas para desarrollar sistemas de software para computadoras grandes, estáticas y de escritorio han demostrado ser ampliamente exitosos en su aplicación. Sin embargo, éstas técnicas no son tan fácilmente aplicables para el desarrollo de sistemas de software de dispositivos embebidos [1].

En el desarrollo de software, los patrones de diseño han demostrado facilitar el mantenimiento y escalabilidad del software [1]. El uso de metodologías de desarrollo permite a los desarrolladores integrarse al proyecto y reduce el tiempo de adaptación [10]. Desafortunadamente, la escasez de metodologías dedicadas al desarrollo de software para sistemas embebidos ha resultado en un notable incremento en la inversión de recursos para el desarrollo de dicho software y en una falta de dirección sistemática [1].

1.2 Justificación

Los patrones de diseño facilitan la escalabilidad, mantenimiento y actualización del software, siendo esto puntos clave para optimizar el proceso de desarrollo [7]. Los patrones de diseño que actualmente son utilizados para desarrollar software, pueden ser adaptados para utilizarlos en sistemas embebidos. Es necesario realizar en ellos las modificaciones pertinentes de manera que puedan ser integrados como parte del proceso de desarrollo de aplicaciones IoT específicas.
La necesidad de estandarización del desarrollo de software dedicado a sistemas embebidos es cada vez mayor debido al incremento de la demanda de éste tipo de sistemas. Una propuesta es la de adaptar los patrones de diseño existentes para satisfacer la demanda de software embebido para IoT.

1.3 Hipótesis

El uso de patrones de diseño para el desarrollo de software dedicado a sistemas embebidos aplicados a IoT facilita el mantenimiento e incrementa la escalabilidad de dichos sistemas.

1.4 Objetivo General

Adaptar patrones de diseño para el desarrollo de software dedicado a sistemas embebidos aplicados a IoT.

1.4.1 Objetivos específicos

- Identificación los patrones de diseño que pueden ser adaptados para el desarrollo de software dedicado a sistemas embebidos.
- Adaptación de patrones de diseño para las distintas aplicaciones clasificadas.

1.5 Metas

1. Revisión del estado del arte.
    a. Identificación de los patrones de diseño existentes.
    b. Identificación de las categorías de los patrones de diseño.
    c. Identificación de los tipos de aplicaciones comunes en IoT.
2. Selección de los patrones de diseño adaptables para el desarrollo de software dedicado a sistemas embebidos.
3. Clasificación de los patrones de diseño generales de sistemas embebidos.
4. Establecer relación entre la clasificación de los patrones de diseños y los tipos de aplicaciones comunes en IoT.
5. Adaptación de patrones de diseño para las distintas aplicaciones clasificadas.
6. Desarrollo de prototipos aplicando los patrones de diseño.

7. Análisis del desempeño de los prototipos.

1.6 Metodología

A continuación se describirá la metodología a emplear que permitirá llevar a cabo el trabajo mencionado anteriormente.

Etapa 1:

- Búsqueda bibliográfica acerca de patrones de diseño de software tanto generales como de sistemas embebidos.

- También se realizará una investigación sobre el tipo de aplicaciones de IoT que son utilizadas actualmente.

- Investigación sobre el uso de patrones de diseño para IoT para obtener una base de patrones que ya se han utilizado previamente y permitirá conocer los diferentes tipos que existen.

Etapa 2:

- Identificación de los distintos tipos de aplicaciones de IoT que existen y su clasificación de acuerdo a sus características.

- Selección de los patrones de diseño que se modificarán.

- Clasificación de patrones de diseño de acuerdo a sus casos de uso.

Etapa 3:

- Categorizar los diferentes tipos de aplicaciones de IoT previamente analizadas de acuerdo a la clasificación de patrones de diseño.

- Adaptar los patrones de diseño a los tipos de aplicaciones de IoT.

- Creación de prototipos utilizando los patrones de diseño previamente adaptados.

Etapa 4:

- Analizar los resultados de los prototipos haciendo uso de métricas que permiten la evaluación de su desempeño.

Capítulo 2: Antecedentes

2.1 Patrones de diseño

Los patrones de diseño fueron propuestos como patrones de arquitectura por Christopher Alexander en 1979, mencionando que mediante el aprendizaje y una serie de patrones, era más fácil hacer edificios de mayor calidad. “Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez” [3]. Los patrones tienen propuestas de soluciones para problemas específicos.

En la ingeniería de software, un patrón de diseño de software es una solución general reusable para un problema que ocurre comúnmente en un contexto específico en el diseño de software. El patrón de diseño no nos da un diseño terminado que podemos directamente convertir en código en el lenguaje de programación que necesitamos. Pero nos da un plano que nos describe una solución general que puede usar en distintas situaciones, y deben ser adaptados para nuestro problema específico[7].

En general, un patrón de diseño cuenta con: un nombre que resume su funcionamiento, describe su propósito, nos dice el problema que resuelve bajo ciertas condiciones, la solución para el problema descrito anteriormente, y las ventajas y desventajas de hacer uso de dicho patrón de diseño, pues nuestro problema pudiera ser resuelto haciendo uso de distintos patrones, y se tienen que considerar las posibles soluciones en caso de tener más de una [4][5].

Comunicarse con otro desarrollador de un equipo usando patrones, no sólo le transmite el nombre de un patrón, si no un grupo completo de cualidades, características y restricciones que el patrón contiene [7], de tal manera que los desarrolladores puedan seleccionar el patrón que más se adapte al problema que

desean resolver. Los patrones se pueden clasificar dependiendo del tipo de problema que resuelven.

2.1.1 Patrones de diseño Creacionales

Estos patrones de diseño abstraen el proceso de instanciamiento. Ayudan a hacer que el sistema sea independiente de la forma en la que sus objetos se crean y se componen. Los patrones creacionales adquieren importancia mientras el sistema evoluciona para depender más en la composición de objetos en vez de herencia de clases. A medida que esto ocurre, el énfasis se aleja de la codificación rígida de un conjunto fijo de comportamientos hacia la definición de un conjunto más reducido de comportamientos fundamentales que pueden combinarse para formar cualquier cantidad de comportamientos más complejos. Estos encapsulan el conocimiento sobre las clases concretas que el sistema usa, y después, ocultan la forma en la que las instancias de dichas clases se crean y se componen. Así, el sistema solo conoce las interfaces de los objetos [18].

Los patrones creacionales se componen por:
Fábrica Abstracta (Abstract Factory): Permite la creación de objetos sin especificar su tipo.
Constructor (Builder): Facilita el crear objetos complejos.
Método de fábrica (Factory method): Crea objetos sin especificar el tipo de clase exacta que se debe crear.
Prototipo (Prototype): Crea un nuevo objeto de otro objeto existente.
Singleton (único): Asegura que solo se pueda crear una instancia de un objeto.

2.1.2 Patrones Estructurales

Los patrones estructurales se relacionan con la manera en la que las clases y objetos se componen para formar estructuras más grandes. Estos patrones utilizan

la herencia para componer interfaces o implementaciones. Por ejemplo, en herencia múltiple se combinan dos o más clases para formar una sola, resultando en una clase que combina propiedades de todas sus clases padre. Estos patrones son particularmente útiles para lograr que bibliotecas de clases desarrolladas de manera independiente puedan trabajar en conjunto [18].

En vez de componer interfaces o implementaciones, los patrones estructurales describen la manera en componer objetos para tener nueva funcionalidad. La flexibilidad que se obtiene de la composición de objetos viene de la posibilidad de cambiar la composición en tiempo de ejecución, que es imposible con composición de clase estáticas [18].

Los patrones estructurales se componen por:

Adaptador (Adapter): Permite que dos clases no compatibles trabajen juntas al envolver una interfaz alrededor de una de las clases existentes.

Puente (Bridge): Desacopla la abstracción para que dos clases puedan variar de manera independiente.

Compuesto (Composite): Junta un grupo de objetos en un solo objeto.

Decorador (Decorator): Permite que el comportamiento de un objeto aumente dinámicamente en tiempo de ejecución.

Fachada (Facade): Provee una interfaz simple a un objeto subyacente complejo.

Peso Mosca (Flyweight): Reduce el costo de modelos de objetos complejos.

Proxy (Proxy): Proporciona una interfaz temporal a un objeto subyacente para controlar su acceso, reducir su costo o reducir su complejidad.

2.1.3 Patrones de comportamiento

Los patrones de comportamiento se ocupan de los algoritmos y la asignación de responsabilidades entre objetos. No solo describen los patrones de objetos y clases, sino también la forma en que se comunican entre ellos. Dichos patrones caracterizan el flujo de control complejo que es difícil de seguir en tiempo de ejecución. Estos patrones desvían el enfoque del flujo de control para permitir

concentrarse solo en la forma en que los objetos están interconectados. Utilizan la herencia para distribuir el comportamiento de las clases.

Los patrones de comportamiento se componen por:

Cadena de responsabilidades (Chain of responsibility): Delega comandos a una cadena de objetos de procesamiento.

Command (Comando): Crea objetos que encapsulan acciones y parámetros.

Intérprete (Interpreter): Implementa un lenguaje especializado.

Iterador (Iterator): Accede a elementos de un objeto de manera secuencial sin exponer su representación subyacente.

Mediador (Mediator): Permite acoplamiento flexible entre clases al ser la única clase que tiene información detallada sobre sus métodos.

Recuerdo (Memento): Proporciona la habilidad de restaurar un objeto a un estado previo.

Observador (Observer): Es un patrón de publicar/suscribir que permite a una cantidad de observadores ver un evento.

Estado (State): Permite a un objeto alterar su comportamiento cuando su estado interno cambia.

Estrategia (Strategy): Permite que una familia de algoritmos sea seleccionada sobre la marcha en tiempo de ejecución.

Método plantilla (Template method): Define el esqueleto de un algoritmo como una clase abstracta, permitiendo que las subclases proporcionen comportamiento concreto.

Visitante (Visitor): Separa un algoritmo de una estructura de un objeto al mover la jerarquía de métodos hacia un objeto.

2.2 Sistemas Embebidos

Un sistema embebido es una computadora especializada que usualmente está integrada en un sistema más grande, y consiste en una combinación de hardware y software para formar un motor computacional que realizará un trabajo específico. A

diferencia de una computadora de escritorio convencional que está diseñada para funciones generales, los sistemas embebidos están restringidos a su aplicación [1].

Cuando se crean aplicaciones con sistemas embebidos y de tiempo real no hay lugar para error. La naturaleza de este tipo de sistemas exigen eficiencia y robustez. Las limitaciones de los recursos del procesador y la memoria aumentan la complejidad de estos desafíos [6].

Una parte de un sistema embebido consiste en el hardware que provee el rendimiento necesario para la aplicación y el software que provee la mayoría de las funciones en el sistema [1].

2.2.1 Raspberry Pi Pico W

Raspberry Pi Pico es una tarjeta de desarrollo basada en el microcontrolador RP2040. Cuenta con 2 núcleos ARM a 133 MHz, 2MB de memoria flash y 264KB de RAM. Además, tiene la capacidad de conectarse a Wi-Fi mediante la interfaz de 2.4GHz gracias al chip Infineon CYW43439. Es capaz de operar con voltajes desde ~1.8 a 5.5V. [24]

A continuación se muestran las entradas y salidas del raspberry pi pico w.

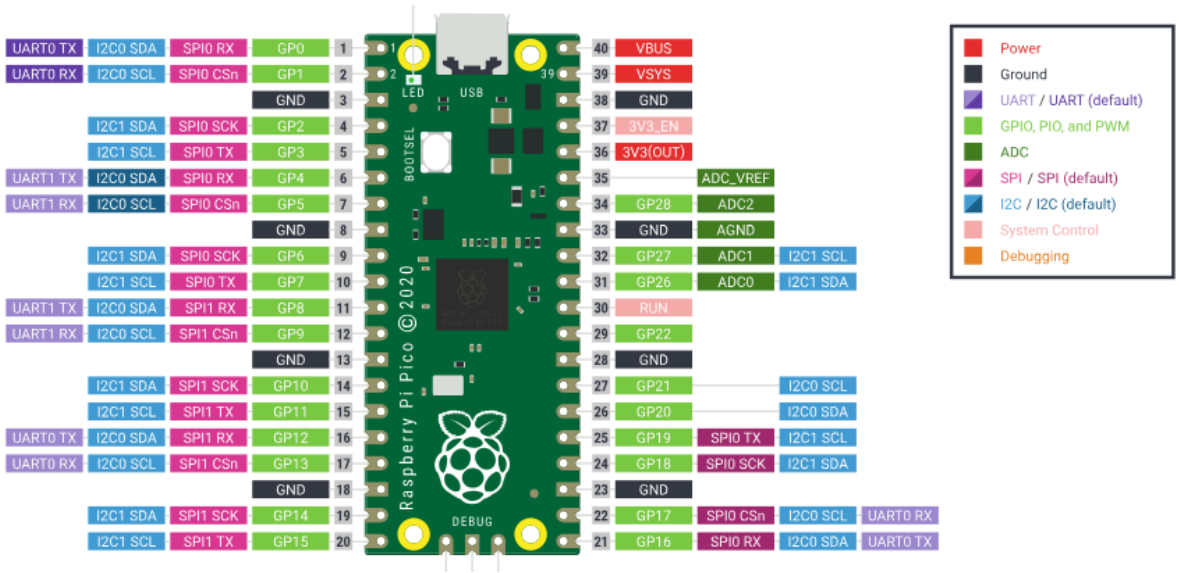

Figura 1, Tarjeta de desarrollo Raspberry pi pico W [24].

2.2.2 ESP32 Wroom DevKit

El esp32 wroom devkit es una tarjeta de desarrollo que cuenta con el SoC ESP32, siendo este una mejora del ESP8266. Cuenta con 2 núcleos que llegan hasta 240 MHz. Cuenta con 448KB de memoria ROM, 520 KB de SRAM y 8 MB de flash. Tiene soporte para Bluetooth 4.2 y Wi-Fi a 2.4 GHz [23].

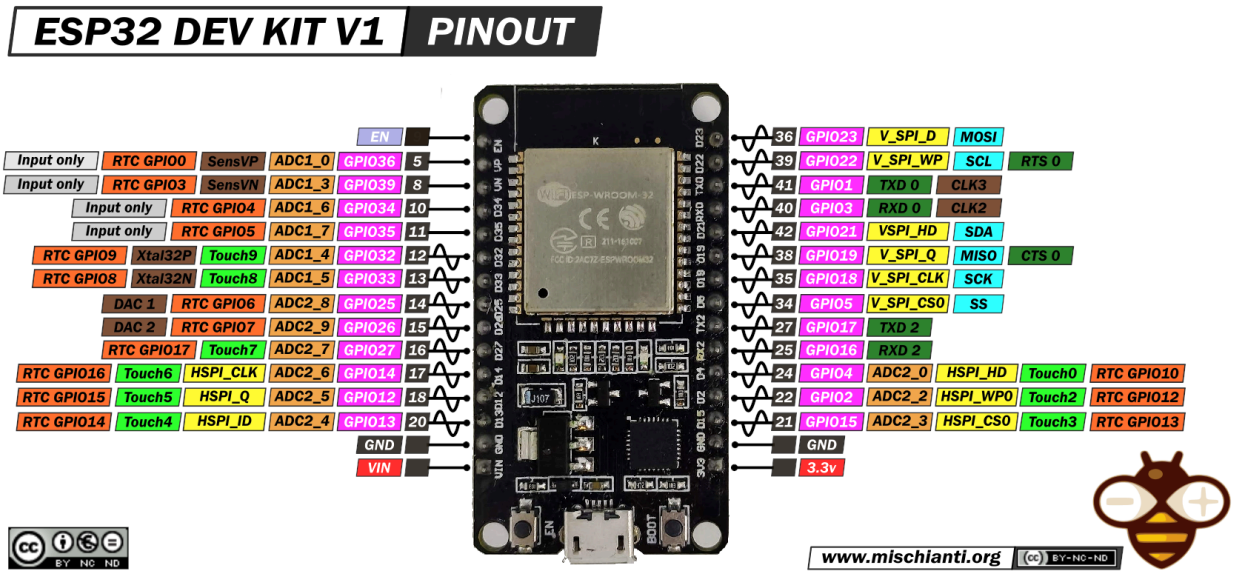

Figura 2, Tarjeta de desarrollo ESP32 Wroom Devkit [23].

2.2.3 Patrones de diseño para sistemas embebidos

Los sistemas embebidos se caracterizan por sus diferentes tipos de hardware, es por eso que existen patrones de diseños que nos ayudan a manejar estos recursos.

- Patrón de diseño de puerto serie: Los sistemas embebidos deben interactuar con diferentes tipos de hardware, el puerto serie siendo de los más comunes. Este patrón de diseño se enfoca en las interfaces del tipo HDLC, RS-232, RS-422, etc.

  Este patrón define una interfaz genérica con un dispositivo de puerto serial. La

intención es encapsular la interfaz con el hardware del puerto serie del dispositivo, lo que hace que las clases que tienen interfaz con el puerto serial no se vean impactados con los cambios de hardware.

El motivo principal para el desarrollo de este patrón fue minimizar la dependencia en el hardware, pues si se llegara a cambiar el tipo de interfaz gracias al costo o por su funcionalidad, no se tendría que cambiar esta parte del software, siendo esta de las principales ventajas [2].

Este patrón es aplicable para todos los dispositivos que involucran transferencias de bytes directas con otros puertos serie.

Este patrón de diseño al ser general para sistemas embebidos ayuda a manejar recursos específicos del hardware, sin embargo es independiente de una aplicación IoT, pues no siempre se utiliza algún puerto serie.

2.3 Internet de las Cosas (IoT)

Un sistema embebido suele ser utilizado en entornos reactivos y con restricciones de tiempo, tal es el caso de las aplicaciones de IoT. El internet de las cosas (IoT) es la composición o agrupación de múltiples dispositivos (cosas) a través de una red donde estos pueden interactuar con el ambiente, entre sí y en tiempo real. Estos dispositivos suelen ser también sistemas embebidos [8].

El éxito del IoT depende en gran medida de las tecnologías y protocolos de comunicación que permiten la interconexión de los dispositivos. Entre los más comunes se encuentran:

Wi-Fi: Ideal para aplicaciones domésticas y de oficina debido a su alta velocidad de transmisión y amplia disponibilidad.

Bluetooth y Bluetooth Low Energy (BLE): Utilizados principalmente en dispositivos portátiles y aplicaciones de corto alcance.

Zigbee: Un protocolo de baja potencia y corto alcance utilizado en sistemas de automatización del hogar y sensores industriales.

LPWAN (Low-Power Wide-Area Network): Incluye tecnologías como LoRa y Sigfox, diseñadas para comunicaciones de largo alcance y baja potencia, ideales para aplicaciones de IoT en áreas rurales y urbanas.

Redes celulares (4G/5G): Ofrecen alta velocidad y cobertura extensa, adecuadas para aplicaciones IoT que requieren movilidad y transmisión de grandes volúmenes de datos.

2.3.1 Aplicaciones

El IoT tiene una amplia gama de aplicaciones que abarcan diferentes sectores:

Domótica: Control y automatización de sistemas domésticos como iluminación, calefacción, seguridad y electrodomésticos.

Salud: Monitorización de pacientes en tiempo real, dispositivos médicos conectados, y gestión de equipos hospitalarios.

Industria: Mantenimiento predictivo, optimización de procesos, y gestión de inventarios en tiempo real.

Agricultura: Monitoreo de cultivos y ganado, gestión del riego, y análisis del suelo.

Ciudades inteligentes: Gestión eficiente del tráfico, iluminación pública, y recolección de residuos.

2.4 Otros Patrones de Diseño

Además de los patrones presentados anteriormente, existe otro amplio rango de patrones de diseño, cada uno para problemas más específicos, pero manteniendo su generalidad para ser reutilizables dentro de su contexto.

En [15] se presentan una serie de patrones con un enfoque centrado en sistemas embebidos, se propone una agrupación de la siguiente manera:

- Patrones de Arquitectura: éste tipo de patrones se utiliza para lidiar con problemas de arquitectura. Tienen un alcance más amplio de los problemas y suelen resolver dificultades en la ingeniería.
    - Arquitectura hexagonal (Hexagonal architecture): A partir de este patrón se crean componentes débilmente acoplados que pueden ser conectados entre sí mediante puertos y adaptadores.
    - Arquitectura por capas (Layer architecture): Organiza los componentes en n-niveles o capas, cada una con un rol específico.
    - Microkernel: Crea kernels de menor tamaño con lo necesario para comunicarse con un sistema operativo.
    - Publicar-subscribir (Publish-Subscribe): Desacopla los componentes que publican datos de un tema de los componentes que quieren recibir dichos datos.
    - Arquitectura basada en eventos (Event driven architecture): Crea un sistema alrededor de la producción, detección, consumo y reacción a eventos.
    - Lenguaje de dominio específico (Domain specific language): Es la construcción de un lenguaje o dialecto apto para un dominio de problema específico.

    - Tuberías y filtros (Pipes and filters): Consiste de componentes (filtros) que transforman o filtran información la pasan a otros componentes mediante conectores (tuberías).

- Patrones de software embebido: este tipo de patrones se utilizan en el desarrollo de software para sistemas embebidos, generalmente solucionan problemas relacionados a la utilización de hardware.
    - Software de baja memoria (Small memory software): Son utilizados donde se cuenta con poca memoria para el almacenamiento de datos.
    - *Watchdog timers*: Para el uso adecuado de watchdog timers.
    - Interfaz de hardware (Hardware interfacing): Se encarga de crear una interfaz al hardware mediante software reutilizable, desacoplando el hardware específico.

- Máquinas de estado (State machines): Crea estados que definen el comportamiento de un componente en un momento determinado.
    - *Ultimate Hook*: Provee políticas para manejo de eventos, pero permite al cliente cambiar cada aspecto del comportamiento del sistema.
    - Recordatorio (Reminder): Crea un evento para actuar como recordatorio al mismo.
    - Evento diferido (Deferred event): Simplifica las máquinas de estado modificando la secuencia de los eventos.
    - Componente Ortogonal (Orthogonal component): Utiliza las máquinas de estado como componentes.
    - Transición a historia (Transition to history): Sale de un estado, pero recuerda el último estado activo para poder regresar a él después.
    - Almacenamiento de estado local (State-Local Storage): Reduce el tamaño utilizado en memoria por las máquinas de estado mediante el uso de variables de estado locales dentro del contexto.

- Asignación de memoria: Estrategias de asignación de memoria que se utilizan dependiendo de los requerimientos.
    - Asignación fija (Fixed allocation): Asegura siempre tener memoria disponible pre-asignando estructuras, y evita utilizar asignación de memoria dinámica durante el procesamiento.
    - Asignación variable (Variable allocation): Evita tener memoria vacía utilizando asignación dinámica.
    - Descarte de memoria (Memory discard): Simplifica la desasignación de objetos temporales al ponerlos en un espacio de trabajo temporal y desasigna el espacio de trabajo completo.
    - Asignación agrupada (Pooled allocation): Evita necesitar memoria adicional al pre asignar una cantidad de objetos similares.
    - Compactación (Compaction): Evita fragmentación de memoria al mover a los objetos asignados en memoria para remover espacios fragmentados.
    - Conteo de referencias (Reference counting): Maneja objetos compartidos manteniendo un conteo de las referencias a cada objeto, eliminando cada objeto cuando su cuenta de referencias de 0.
    - Recolección de basura (Garbage collection): Maneja objetos compartidos identificando periódicamente objetos sin referencia y los elimina.

- Manejo de memoria limitada: Hace que cada componente sea responsable de su propio uso de memoria [13].
    - Límite de memoria (Memory limit): Ejerce un límite máximo de memoria que un componente puede asignar.
    - Interfaces pequeñas (Small interfaces): Se crean interfaces pequeñas entre componentes que son diseñados para manejar memoria de manera explícita, reduciendo la memoria requerida para su implementación.

    - Fallo parcial (Partial Failure): Asegura que un componente pueda continuar en “modo reducido”, sin detener su proceso o perder datos existentes cuando no se puede asignar memoria.
    - *Captain Oates*: Mejora el rendimiento general del sistema al entregar memoria utilizada por componentes menos importantes cuando el sistema se está ejecutando con baja memoria.
    - Memoria de sólo lectura (Read-only memory): Utiliza este tipo de memoria para almacenar componentes que no se necesitan modificar, favoreciendo a los que necesitan más memoria principal.
    - Ganchos (Hooks): Permite información almacenada en memoria de sólo lectura aparentar ser cambiada.

- Estructuras de datos pequeñas: Reduce el almacenamiento ocupado por estructuras de datos con distintas técnicas.
- Compresión (Compression): Se utilizan para almacenar datos y código cuando éstos necesitan más almacenamiento del que se tiene disponible.
    - Tabla de compresión (Table compression): Codifica cada elemento en una cantidad de bits variable para que los elementos más comunes requieran menos bits.
    - Codificación de diferencia (Difference coding): Representa secuencias de acuerdo a las diferencias entre éstas.
    - Compresión adaptativa (Adaptive Compression): Utiliza un algoritmo de compresión adaptativo.

- Almacenamiento secundario: Utiliza almacenamiento secundario como memoria adicional durante el tiempo de ejecución.
    - Cambio de aplicación (Application Switching): Divide el programa en ejecutables independientes, sólo uno de ellos se ejecuta a la vez.

- Archivos de datos (Data files): Utiliza el almacenamiento secundario como ubicación para datos inactivos del programa. Estos archivos pueden ser o no visibles para el usuario.
- Archivos de recurso (Resource files): Almacena datos estáticos de sólo lectura. Cuando el programa necesita un recurso (Como un mensaje de error, o la descripción de una ventana), carga el archivo de recurso en memoria temporal, después libera la memoria.
- Paquetes (Packages): Almacena pedazos de código del programa. El programador divide el código en paquetes, que se cargan y descargan como se necesite en tiempo de ejecución.
- Paginamiento (Paging): Arbitrariamente parte el programa en unidades finas (páginas), que después son repartidas automáticamente entre almacenamiento primario y secundario. El paginamiento puede manejar código y datos, y es transparente a la mayoría de los programadores y usuarios.

Sistemas distribuidos: Un sistema distribuido es un entorno en el cual, los componentes están alojados en diferentes computadoras en una red. Dichos componentes se reparten el trabajo y se encargan de realizar una tarea de manera eficiente [17].

Capítulo 3: Análisis de aplicaciones IoT

En este capítulo, se definirán conceptos de IoT que se estarán trabajando.

Una aplicación de IoT puede contar con muchos sistemas que la componen, tales como múltiples dispositivos de sensado, el sistema en la nube que provee la información al usuario, dispositivos para conectarse a dicha nube, entre otros.

Empezaremos acortando en qué parte de la aplicación IoT nos vamos a enfocar:

- Terminal: Dispositivo que tiene una interface para conectarse a la cosa.
- Cosa: Es el sistema embebido que tiene el código encargado del sensado de datos, así como una conexión a internet, con capacidad de enviar información a la nube y tomar decisiones con dicha información.
- Categorías: Las diferentes categorías de la aplicación IoT, algunos ejemplos son IoT de consumidor, militar, médico, entre otros.
- Comunicación: El protocolo por el cuál están conectados al sistema IoT.

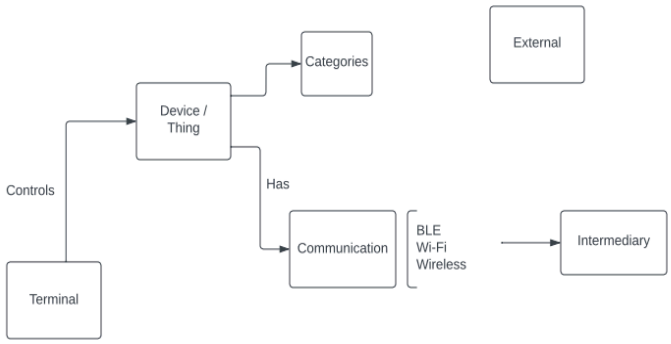

Figura 3, Diagrama de componentes de una aplicación IoT.

Con base al diagrama anterior, podemos indicar en la parte en la que vamos a trabajar, ya que se encuentra tanto en el dispositivo como su protocolo de comunicación. Si bien, un celular es un sistema embebido, no suele ser utilizado como la “cosa” en una aplicación IoT, por lo que no está dentro del alcance de este trabajo.

Para conocer el tipo de patrones que son utilizados comúnmente en las aplicaciones de IoT, se revisaron aplicaciones en la plataforma de github que cumplieran con especificaciones de una aplicación IoT [19][20].

Además se utilizó la clasificación en [21] para distinguir los diferentes tipos de aplicaciones de IoT en un campo más amplio, logrando así encontrar las similitudes entre aplicaciones.

Finalmente con base en [16], se consideraron 4 puntos clave que son recurrentes en la mayoría de aplicaciones de IoT, independientemente de su clasificación, estos puntos claves siendo sensado, procesamiento, comunicación, actuación.

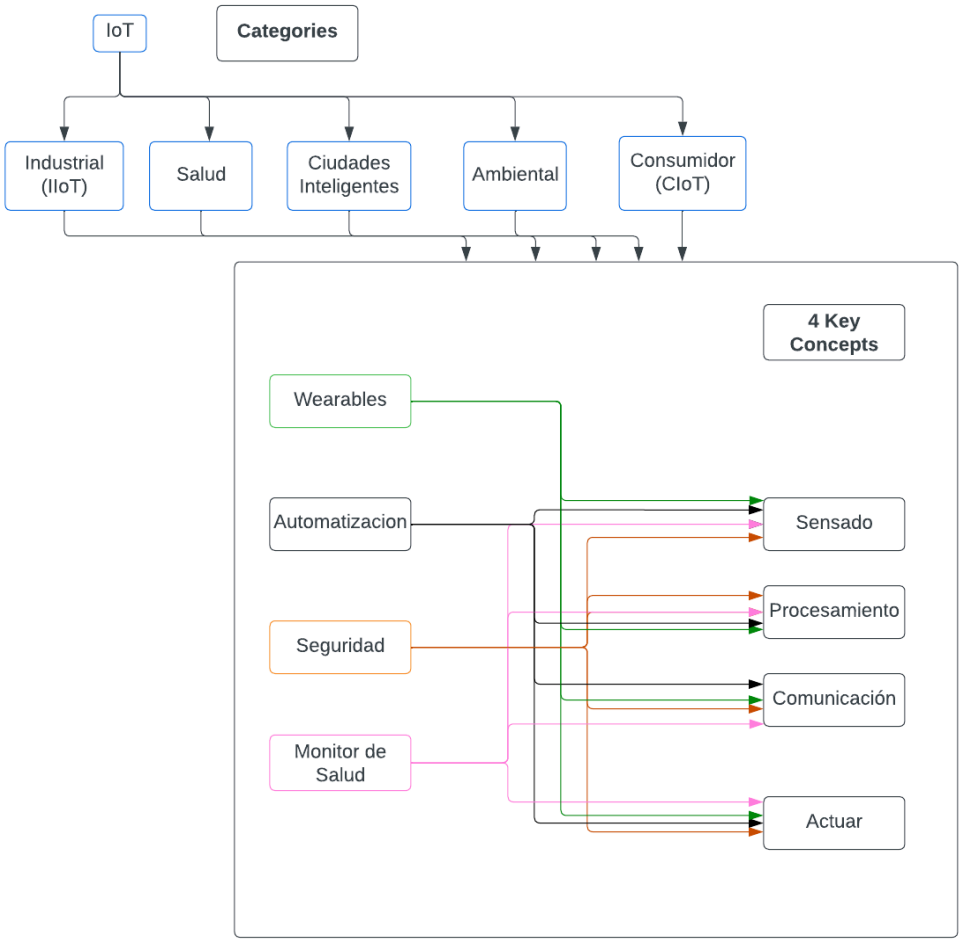

Figura 4, Diagrama de conceptos clave utilizados en IoT.

El sensado es la forma en la que las aplicaciones de IoT se comunican con el entorno, utilizan sensores para saber el estado actual de una puerta, la temperatura de un cuarto, etc.

El procesamiento se puede hacer ya sea en el sistema embebido, tal como promedio o acomodo de datos, o enviarlo a la nube para realizar operaciones que requieran mayor poder.
La comunicación es la forma en la que los sistemas de una aplicación interactúan. Se puede tener un sistema encargado de subir los datos a la nube a través de internet, mientras que otros dispositivos le envían datos a este sistema por medio de bluetooth u otro medio de comunicación.

La actuación es la parte del sistema que se encarga de tomar una decisión con base a los datos obtenidos de los sensores, o bien, de información obtenida de los otros sistemas.

Capítulo 4: Análisis y Selección de patrones

Con base a las aplicaciones de IoT revisadas en el capítulo anterior, se realizó un análisis de patrones de diseño, así como su actual aplicación en sistemas embebidos y en aplicaciones de IoT. En el capítulo siguiente se hablará de su uso de manera detallada en los diferentes tipos de aplicaciones.

Se analizaron los patrones del libro “Patrones de diseño, elementos de software orientado a objetos reutilizable” [18] como base de los patrones de diseño. Cabe destacar que estos patrones se encuentran en el paradigma orientado a objetos. Se utilizó principalmente el lenguaje C++ para el desarrollo de las aplicaciones, ya que este lenguaje tiene la capacidad de ser utilizado con el paradigma orientado a objetos. En [1] se muestra un ejemplo C para demostrar que el patrón no está condicionado por el lenguaje.

4.1 Patrones estructurales

4.1.1 Facade

Propósito

Provee una interfaz unificada a una serie de interfaces en un subsistema. Nos proporciona una interfaz que hace que el subsistema sea más fácil de usar.

Motivación

Estructurar un sistema en subsistemas ayuda a reducir la complejidad, y una meta común del diseño es minimizar la comunicación y dependencia entre subsistemas. Una forma de cumplir esta meta es con un objeto del tipo fachada que provee una interfaz simple que nos ayude a controlar las partes más generales de un subsistema.

Consiste en una clase que actúa como punto de entrada a un conjunto de clases más complejas, encapsulando sus funcionalidades y ocultando su complejidad interna.

En el ámbito de las aplicaciones de IoT y sistemas embebidos, la motivación detrás del patrón Fachada tiene un enfoque similar. Estos sistemas suelen estar compuestos por una variedad de componentes y dispositivos con funcionalidades diversas y complejas. Por ejemplo, en un sistema de automatización del hogar, podríamos tener una variedad de dispositivos como sensores de temperatura, actuadores para controlar luces y cerraduras inteligentes, y componentes para la gestión de energía.

La complejidad de la interacción y coordinación de estos dispositivos puede volverse abrumadora para los desarrolladores y dificultar la mantenibilidad y la extensibilidad del sistema. Aquí es donde la utilización del patrón Fachada puede proporcionar claridad y simplicidad en el diseño del sistema.

Al proporcionar una interfaz unificada y simplificada a través de una fachada, el patrón permite a los desarrolladores interactuar con el subsistema de manera coherente y sin tener que preocuparse por los detalles internos de implementación de cada componente. Por ejemplo, en el sistema de automatización del hogar mencionado anteriormente, la fachada podría ofrecer métodos como 'encenderLuz()' o 'activarCalefacción()', ocultando la complejidad de la comunicación con los dispositivos subyacentes y facilitando el desarrollo de aplicaciones que controlan el sistema.

Además, el uso del patrón Fachada promueve la modularidad y el acoplamiento débil entre los diferentes componentes del sistema, lo que facilita la reutilización y la evolución del sistema a medida que cambian los requisitos y se introducen nuevas funcionalidades.

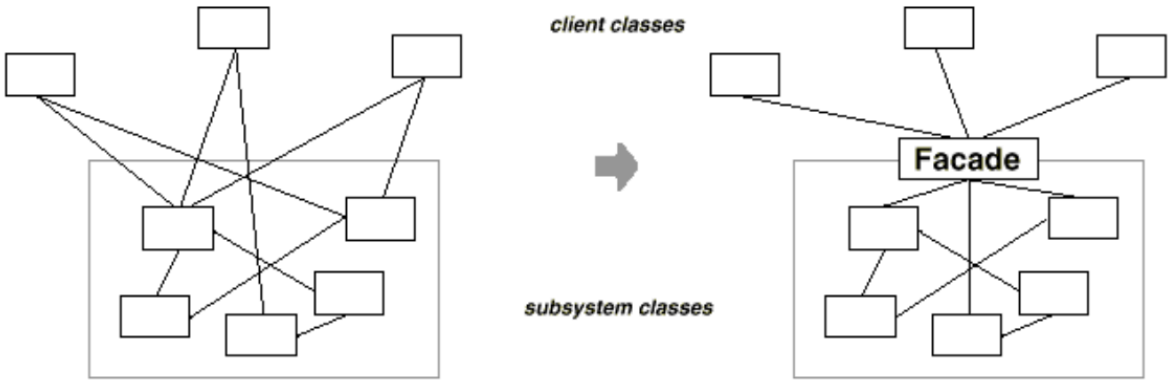

Figura 5, Diagrama OMT del patrón fachada.

4.1.2 Adapter

Propósito

Convierte la interfaz de una clase a otra interfaz esperada por el cliente. Esto permite que clases que no podrían trabajar juntas previamente por tener interfaces incompatibles ahora lo puedan hacer.

Motivación

Consiste en un adaptador que actúa como un intermediario entre dos interfaces, convirtiendo la interfaz de un objeto en otra interfaz que un cliente espera encontrar.

En el contexto de las aplicaciones de IoT y sistemas embebidos, la motivación detrás del patrón Adaptador es crucial debido a la diversidad de dispositivos y protocolos de comunicación que pueden existir en un sistema. Estos sistemas suelen integrar una variedad de dispositivos de hardware y software que pueden tener diferentes interfaces y protocolos de comunicación.

Por ejemplo, en un sistema de monitoreo industrial basado en IoT, pueden utilizarse diferentes tipos de sensores y actuadores que se comunican a través de protocolos de comunicación variados, como UART, SPI, o incluso protocolos personalizados sobre TCP/IP. En este contexto, puede ser necesario unificar la comunicación entre estos dispositivos para facilitar su interacción y control desde una plataforma centralizada.

El patrón Adaptador permite abordar esta diversidad de interfaces y protocolos al proporcionar un mecanismo para convertir la interfaz de un dispositivo en otra interfaz compatible con el sistema. Por ejemplo, un adaptador podría ser utilizado para convertir los datos recibidos de un sensor que utiliza un protocolo UART en un formato compatible con un protocolo de comunicación estándar utilizado por el sistema central.

Además, el uso del patrón Adaptador promueve la reutilización de componentes existentes al permitir la integración de dispositivos heredados o de terceros en el sistema sin necesidad de modificar su código fuente. Esto facilita la escalabilidad y la interoperabilidad del sistema a medida que evoluciona con el tiempo y se agregan nuevos dispositivos y tecnologías.

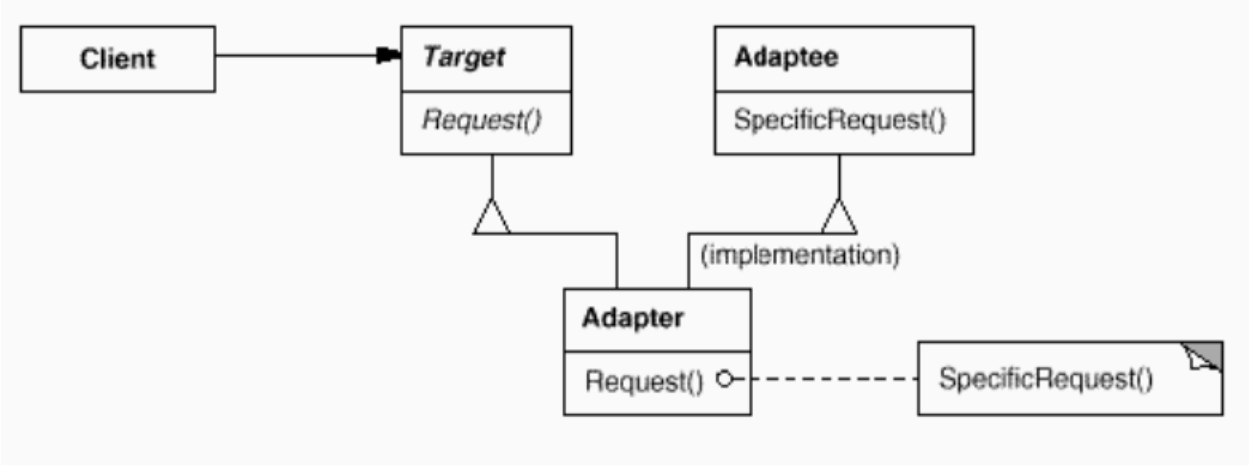

Figura 6, Diagrama OMT del patrón adaptador a nivel de clases.

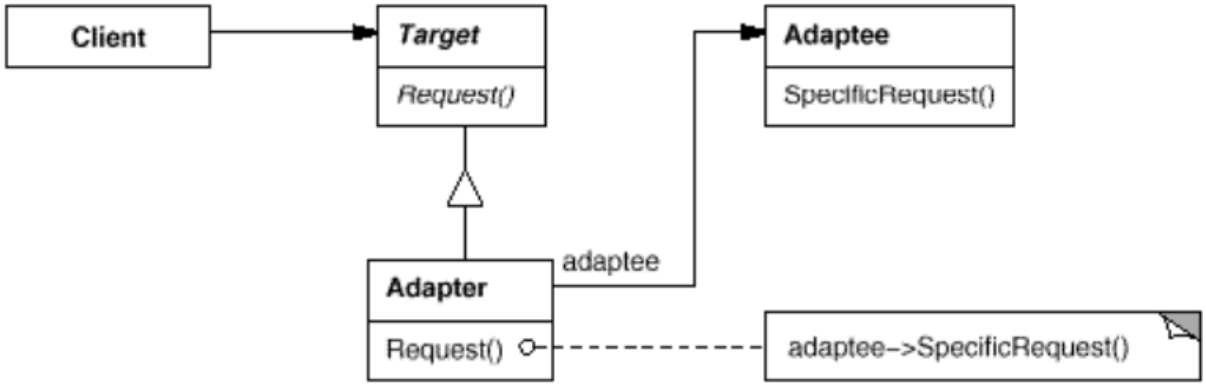

Figura 7, Diagrama OMT del patrón adaptador a nivel de objeto.

Cabe destacar que el patrón Bridge tiene una estructura similar al objeto adaptador, sin embargo la diferencia radica en que el adaptador pretende modificar la interfaz de un objeto ya existente.

4.1.3 Bridge

Propósito

Desacopla la abstracción de la implementación, permitiendo que estas puedan variar de manera independiente.

Motivación

El patrón de diseño Puente desempeña un papel fundamental en la integración de sensores en aplicaciones de Internet de las Cosas (IoT) en sistemas embebidos. En el contexto de la creación de sistemas de monitoreo ambiental, donde la diversidad de sensores es evidente, el uso del patrón Puente se vuelve imperativo.

Los sistemas de monitoreo ambiental a menudo requieren la incorporación de diversos tipos de sensores, cada uno con sus propias interfaces de comunicación, protocolos de datos y requisitos de energía. La aplicación directa del patrón Puente permite crear una abstracción que encapsula la funcionalidad general de un sensor,

como la lectura de datos y el control de configuración, independientemente de su implementación específica.

Por ejemplo, al diseñar un sistema de monitoreo ambiental, es común encontrar sensores de temperatura, humedad y calidad del aire, cada uno con su propio protocolo de comunicación, como I2C, SPI o UART. Al aplicar el patrón Puente, se pueden desarrollar implementaciones específicas para cada tipo de sensor, permitiendo la interacción directa con el hardware del sensor y manejando los detalles específicos de su comunicación y configuración.

La ventaja clave de emplear el patrón Puente radica en la capacidad para desacoplar la lógica de la aplicación de los detalles de implementación de los sensores individuales. Esto simplifica significativamente el desarrollo y el mantenimiento del código, al tiempo que facilita la integración de nuevos tipos de sensores en el futuro. Además, al centralizar la lógica de comunicación y control en una abstracción común, se puede mejorar la escalabilidad del sistema.

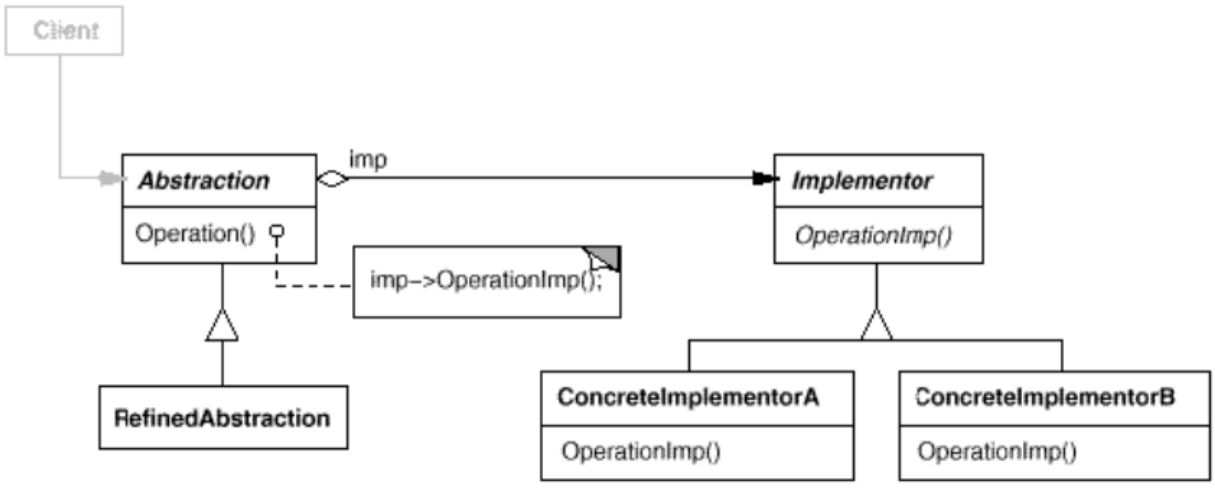

Figura 8, Diagrama OMT del patrón puente.

4.2 Patrones de comportamiento

4.2.1 State

Propósito

Permite que un objeto pueda cambiar su comportamiento cuando su estado interno cambia. El objeto aparenta cambiar su clase.

Motivación

Teniendo una máquina de estados finita, en cualquier momento se puede encontrar en cualquier estado, y tiene la posibilidad de saltar entre estados. Sin embargo, dependiendo del estado actual, se puede mover a solo ciertos estados.

La idea en este patrón es tener una clase abstracta que represente los estados de una conexión. Esta clase declara una interfaz común a las demás clases que presentan diferentes estados.

En el contexto de sistemas embebidos e IoT, se puede aplicar a distintos modos de funcionamiento,

Los sistemas embebidos a menudo operan en diferentes modos (por ejemplo, “normal”, “depuración”, “actualización de firmware”).

El patrón Estado permite definir y gestionar estos modos, adaptando el comportamiento según las necesidades actuales.

Un caso de uso común de este patrón es en la gestión de la conexión de diferentes tipos, tales como Wi-Fi y bluetooth, pues al en el estado de conectado, desconectado o conectando, el sistema se ajusta de tal forma que es claro entender cuál es su funcionalidad y cómo se comportará cuando cambie de un estado a otro.

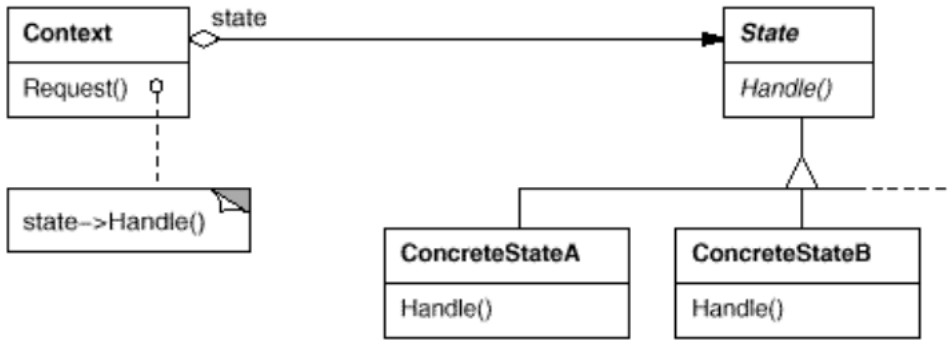

Figura 9, Diagrama OMT del patrón estado.

4.2.2 Observer

Propósito

Define una dependencia de uno a muchos entre objetos, lo que hace que cuando un objeto cambie de estado, todos sus dependientes son notificados y actualizados automáticamente.

Motivación

Se utiliza para definir una relación de uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

En el contexto de las aplicaciones de IoT y sistemas embebidos, la motivación detrás del patrón Observador se hace aún más relevante. Estos sistemas a menudo están compuestos por una variedad de dispositivos y componentes interconectados que necesitan monitorear y responder a cambios en tiempo real. Por ejemplo, considera un sistema de monitoreo ambiental basado en sensores distribuidos. Aquí, múltiples nodos pueden necesitar actualizarse cuando un sensor detecta una fluctuación significativa en los datos ambientales.

La eficiencia energética y la gestión de recursos son consideraciones críticas en el diseño de sistemas IoT y embebidos. Interrogar repetidamente los sensores para obtener actualizaciones consume recursos valiosos, como energía de la batería y ancho de banda de la red, lo que puede ser costoso en términos de rendimiento y escalabilidad del sistema.

Es aquí donde entra en juego el patrón observador. Al establecer una relación de observador-sujeto entre los nodos y los sensores, el patrón permite una propagación automática y eficiente de cambios de estado relevantes. En lugar de que los nodos realicen consultas activas a los sensores, los sensores notifican automáticamente a los nodos cuando detectan un cambio significativo en los datos ambientales. Esto minimiza la sobrecarga de la red y optimiza el consumo de energía al garantizar que los recursos se utilicen de manera eficiente y solo cuando sea necesario.

Por lo tanto, la aplicación del patrón Observador en el contexto de IoT y sistemas embebidos permite una comunicación asíncrona y automatizada entre los componentes del sistema, mejorando la eficiencia operativa y facilitando el desarrollo de sistemas más robustos y escalables.

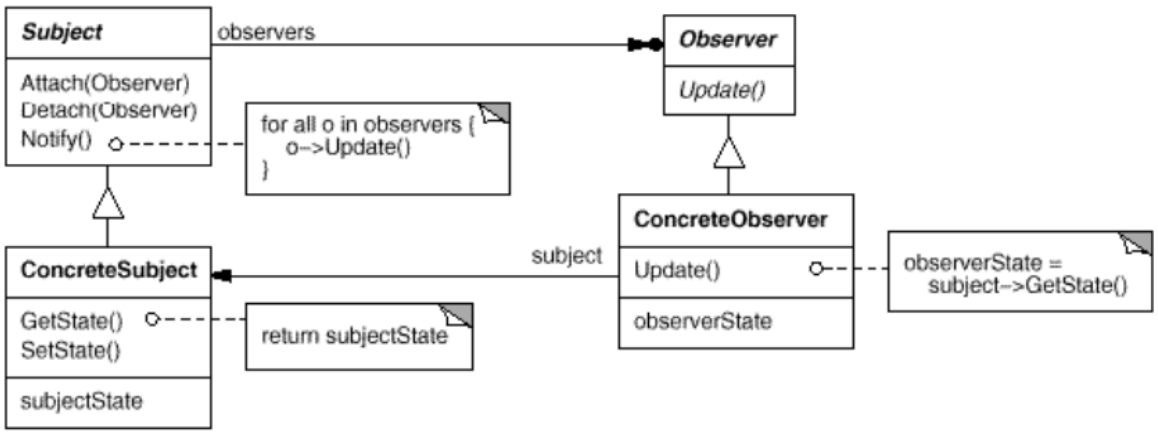

Figura 10, Diagrama OMT del patrón observador.

4.2.3 Strategy

Propósito

Define una familia de algoritmos que se encapsulan en diferentes clases y hace a sus objetos intercambiables.

Motivación

En el desarrollo de sistemas embebidos para aplicaciones de IoT, es común enfrentarse a situaciones en las que un dispositivo debe realizar una misma tarea de diferentes maneras dependiendo del contexto, las condiciones del entorno o incluso las preferencias del usuario. Por ejemplo, un sensor de temperatura podría necesitar procesar sus datos de manera diferente dependiendo de si está en modo de ahorro de energía o en modo de alto rendimiento.

El patrón Strategy proporciona una solución elegante y flexible a este problema al permitir que se definan múltiples algoritmos o comportamientos intercambiables para una misma tarea. Esto se logra encapsulando cada algoritmo en una clase separada y haciendo que las clases del contexto usen estas estrategias de manera intercambiable.

La adopción del patrón Strategy ofrece varias ventajas significativas. Facilita la adición de nuevos comportamientos o algoritmos sin necesidad de modificar el código existente, lo cual es crucial en sistemas embebidos donde la estabilidad y la minimización de errores son esenciales. Al encapsular comportamientos específicos en estrategias separadas, se evita la duplicación de código y se promueve la reutilización. Además, separar los algoritmos en clases distintas simplifica el proceso de mantenimiento y prueba, ya que cada estrategia puede ser desarrollada y verificada de forma independiente.

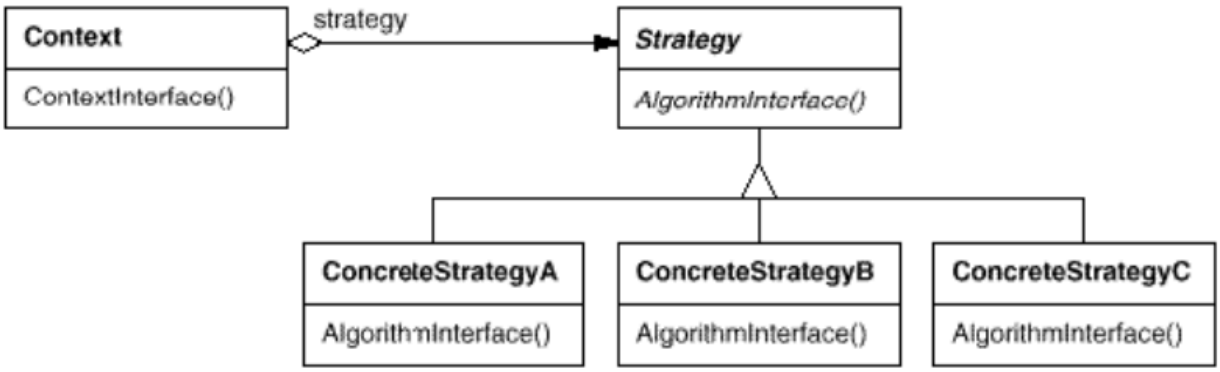

Figura 11, Diagrama OMT del patrón estrategia.

4.3 Patrones Creacionales

4.3.1 Singleton

Propósito

Asegura que una clase solo tiene una instancia, y provee un punto de acceso global a esa instancia.

Motivación

El patrón Singleton es un patrón de diseño creacional que garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a dicha instancia. Este patrón se utiliza comúnmente en situaciones donde se necesita controlar cuidadosamente el acceso a un recurso único o se requiere una única fuente de verdad en el sistema.

En el contexto general del desarrollo de software, el patrón Singleton se utiliza para garantizar que una clase tenga una única instancia en toda la aplicación. Esto es útil en escenarios donde se necesita compartir un recurso único, como una conexión de base de datos, un registro de eventos, o un administrador de configuración.

En sistemas embebidos en aplicaciones de IoT, donde los recursos son limitados y la eficiencia es crucial, el patrón Singleton también encuentra una aplicación significativa. Por ejemplo, en un sistema de control de dispositivos IoT, puede ser necesario utilizar una única instancia de un controlador de comunicación para gestionar la comunicación con dispositivos externos.

Al implementar este controlador como un Singleton, se garantiza que solo exista una instancia activa en todo momento, lo que ayuda a conservar los recursos limitados de memoria y procesamiento en sistemas embebidos. Además, el acceso global a esta instancia única a través de un punto de acceso estático simplifica la integración de la funcionalidad de comunicación en otras partes del sistema, lo que es especialmente beneficioso en entornos donde el espacio y la complejidad del código deben ser cuidadosamente gestionados.

En resumen, el uso del patrón Singleton en sistemas embebidos en aplicaciones de IoT proporciona un mecanismo efectivo para controlar el acceso y la utilización de recursos críticos, garantizando al mismo tiempo la eficiencia y la consistencia del sistema en entornos con recursos limitados. Su capacidad para garantizar una única instancia de una clase y proporcionar un punto de acceso global a esta instancia lo convierte en una herramienta valiosa para el desarrollo de sistemas embebidos en IoT.

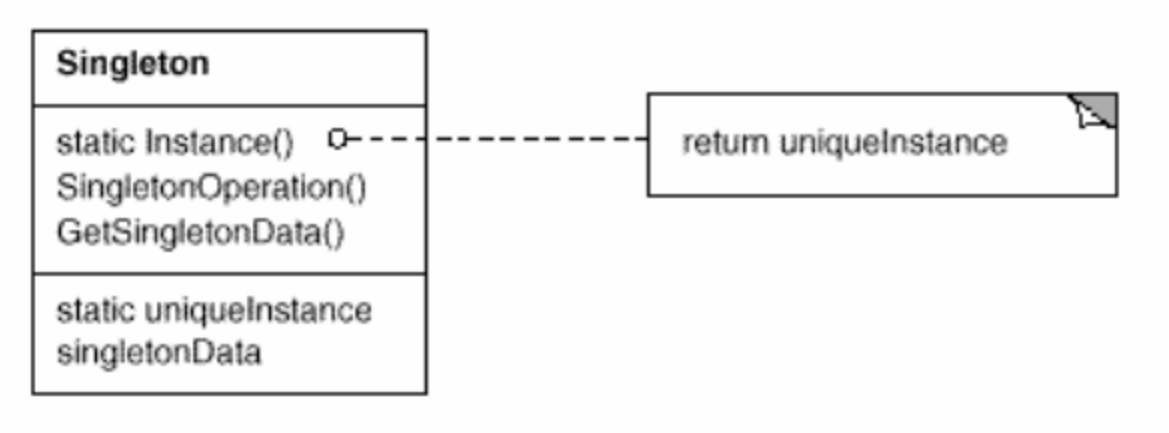

Figura 12, Diagrama OMT del patrón singleton.

4.3.2 Factory

Propósito

Define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase instanciar. El método de fábrica permite que una clase posponga la instanciación a las subclases.

Motivación

El patrón de diseño Factory es un patrón creacional que se utiliza para crear objetos sin especificar la clase exacta del objeto que se creará. En lugar de crear objetos directamente mediante la invocación de su constructor, se utiliza un método de fábrica para crear y devolver instancias de objetos.

En el contexto general del desarrollo de software, el patrón Factory se utiliza para encapsular la creación de objetos y proporcionar una interfaz común para la creación de diferentes tipos de objetos. Esto permite desacoplar el código que utiliza los objetos de las implementaciones concretas de esos objetos, lo que promueve la flexibilidad y la extensibilidad del sistema.

En sistemas embebidos en aplicaciones de IoT, donde la eficiencia y la gestión de recursos son fundamentales, el patrón Factory también encuentra una aplicación significativa. Por ejemplo, en un sistema de control de dispositivos IoT, puede ser necesario crear diferentes tipos de sensores o actuadores según las necesidades del sistema.

Al utilizar el patrón Factory, se puede encapsular la lógica de creación de objetos en una clase de fábrica, lo que permite crear diferentes tipos de objetos según sea necesario. Esto no solo promueve la reutilización del código y la modularidad, sino que también facilita la gestión de recursos en entornos con recursos limitados.

Además, el patrón Factory puede utilizarse para crear objetos de comunicación con diferentes tipos de dispositivos en un entorno IoT. Por ejemplo, se pueden definir diferentes fábricas de comunicación para dispositivos con conexiones Wi-Fi, Bluetooth o de radiofrecuencia, lo que facilita la gestión de la comunicación en sistemas heterogéneos.

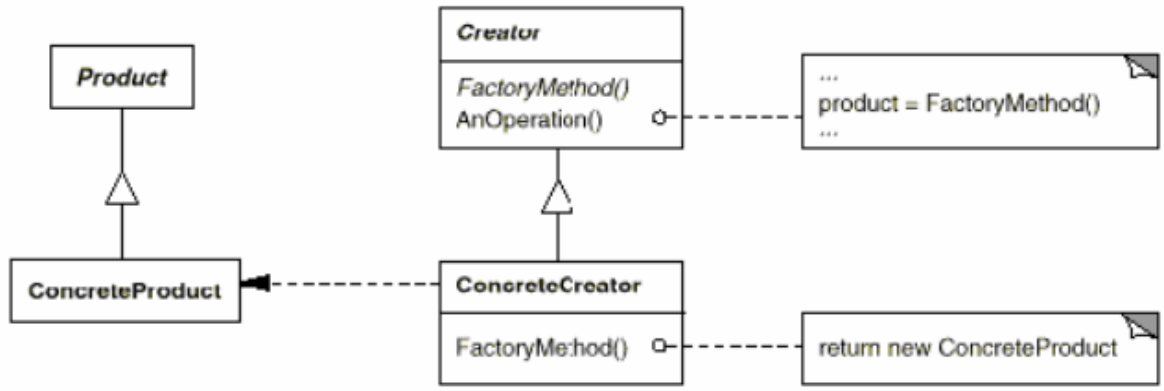

Figura 13, Diagrama OMT del patrón factory.

Capítulo 5 Uso de patrones en las aplicaciones

En los capítulos 3 se habló sobre las aplicaciones IoT más comunes y la sección que estaremos trabajando, mientras que en el capítulo 4 se describieron los patrones que más aparecen. En este capítulo se mostrará cómo es que estos patrones son utilizados y algunas otras formas en las que estos patrones pueden aparecer. Debido a que existen distintas diferentes formas de mostrar su uso, se utilizarán métodos distintos.

Para abarcar los mayores casos de uso posibles, se trabajó principalmente con aplicaciones sencillas para permitir mostrar los 4 conceptos clave descritos anteriormente, y algunos patrones se mostrarán en uno o múltiples de estos conceptos.

Teniendo en cuenta las opciones disponibles en cuanto los lenguajes más utilizados para el desarrollo de aplicaciones IoT en sistemas embebidos, los lenguajes más populares son C y C++. El lenguaje Rust; aunque muy nuevo ha tenido grandes avances para su uso en sistemas embebidos, sin embargo continúa siendo una plataforma relativamente nueva, teniendo un nivel de longevidad menor a C y C++. Tomando en cuenta otras cosas como su facilidad de uso, interfaces de bajo nivel y la facilidad de mantenimiento según [22], C++ obtiene una ligera ventaja sobre C cuando se trata del desarrollo de aplicaciones IoT. También se aprovechó el soporte de C++ para la utilización de conceptos de programación Orientada a Objetos, pues los patrones vistos anteriormente nacen bajo este paradigma. Cabe destacar que aunque C no tenga un enfoque orientado a objetos, los patrones también son aplicables en este lenguaje, tomando en cuenta algunas restricciones.

	C	C++	Java	Python / MicroPython	JavaScript/ HTML5/CSS	Rust
Strenghts	Embedded, bare-metal, IoT	Embedded, bare-metal, standalone apps, IoT	Web, cloud	Cloud, data science	Web	Embedded, bare-metal
Ease of development	★★★	★★	★★★	★★★★★	★★	★★★★
Expressive power	★	★★★★	★★	★★★★★	★★★	★★★★
Ease of maintenance	★★★★	★★★★	★★★★	★★★	★	★★★★
Longevity	★★★★★	★★★★	★★★	★★★	★	★
Runtime efficiency	★★★★★	★★★★★	★★★	★★	★	★★★★
Library/module availability	★★★★	★★★★	★★★	★★★★	★★★★	★★
Low-level interface	★★★★★	★★★★★	★★	★★★	★	★★★★
Connectivity support	★★	★★★/★★★★★[1]	★★★★	★★★★★	★★★★★	★★
Graphics support	★★★★	★★★★★	★★★	★★★	★★★★★	★
Developer community	★★★	★★★	★/★★★★[2]	★★★★	★★★★★	★★

★ is the lowest ranking while ★★★★★ is the highest.

Tabla 1, Lenguajes de programación utilizados para el desarrollo de sistemas embebidos con calificación en distintas categorías [22].

5.1 Implementación del patrón state

El patrón estado es un patrón de diseño que comúnmente podemos apreciar en distintas situaciones, sus usos suelen ser a nivel de arquitectura, donde el sistema trabaja con base a una máquina de estados en la cuál todo el comportamiento del sistema depende del estado en el que está, y su otro uso en aplicaciones de IoT se suele ver del lado de la conectividad. Siendo la conectividad una parte fundamental de las aplicaciones IoT, se exploró el uso de este patrón en esta situación. No fue

sorpresa entonces, que este patrón apareciera en muchas aplicaciones diferentes, debido a que su aplicabilidad radica en poder modificar su comportamiento en tiempo de ejecución.

Para demostrar su uso, se implementó el patrón como se muestra en la siguiente figura.

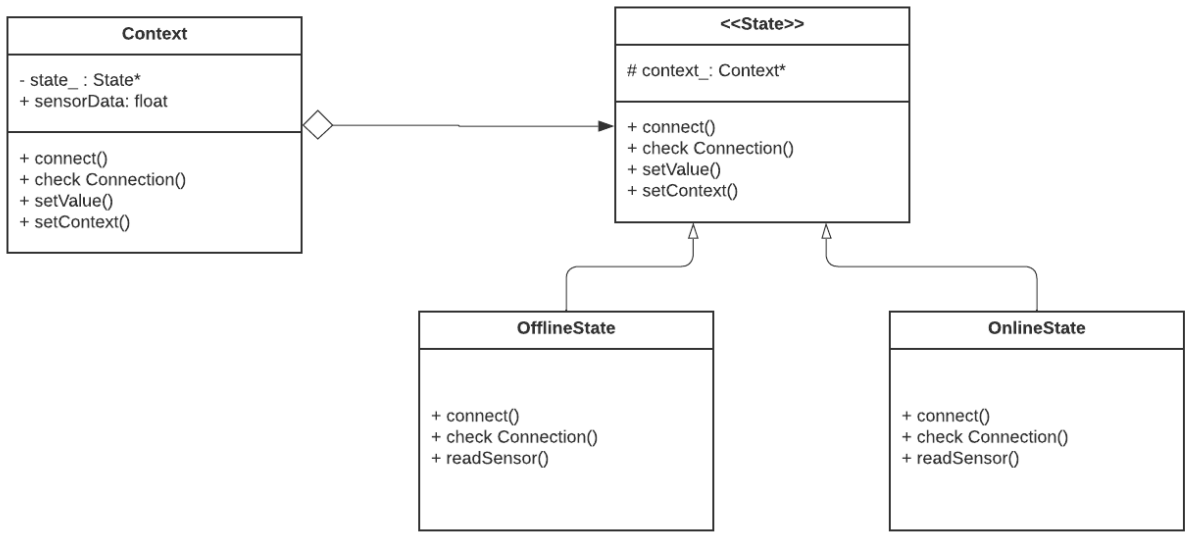

Figura 14, Diagrama del patrón estado para conectividad.

Podemos observar que el estado tiene las funciones Connect(), check Connection() y readSensor(). Estas 3 funciones actuarán de forma distinta dependiendo del estado en el que se encuentre el contexto. Si nos encontramos en OnlineState, connect() simplemente no hará nada, checkConnection() confirma que nos encontramos en línea, por lo que procede a llamar a readSensor() para leer los datos y enviarlos; en este caso, por MQTT.

Si nos encontramos en OfflineState, connect() intentará realizar una conexión, y se verificará con checkConnection(), si checkConnection() nos indica que no nos pudimos conectar la llamada a readSensor() hará una lectura del sensor, para acto seguido almacenarlo en la memoria e intentar enviarlos más tarde.

5.2 Implementación del patrón bridge

El patrón puente no es un patrón que aparece de manera frecuente en estas aplicaciones, sin embargo es un patrón con bastante utilidad, pues permite tener acoplamiento débil entre clases, esto permite extender abstracciones e implementaciones de forma independiente y en tiempo de ejecución. A continuación se mostrará un ejemplo del patrón puente tanto para la comunicación como para el sensado.

En el siguiente código, se tiene una clase para la comunicación y una para el sensado, siendo ambas abstracciones.

```
//Communication
// Implementador: Define la interfaz para los métodos de comunicación
class Communication {
public:
    virtual void sendData(const std::string& data) = 0;
};

// Abstracción: Define la interfaz para los sensores
class Sensor {
protected:
    Communication* communication;
public:
    Sensor(Communication* comm) : communication(comm) {}
    virtual void readAndSendData() = 0;
};
```

Después se observa una implementación concreta para la comunicación WiFi y una para bluetooth.

```
// Implementación concreta A: Comunicación mediante WiFi
class WifiCommunication : public Communication {
public:
    void sendData(const std::string& data) override {
        std::cout << "Enviando datos por WiFi: " << data << std::endl;
    }
```

```
};

// Implementación concreta B: Comunicación mediante Bluetooth
class BluetoothCommunication : public Communication {
public:
    void sendData(const std::string& data) override {
            std::cout << "Enviando datos por Bluetooth: " << data <<
std::endl;
    }
};
```

Posteriormente tenemos la abstracción refinada de 2 sensores, en este caso uno de temperatura y otro de humedad.

```
class TemperatureSensor : public Sensor {
public:
    TemperatureSensor(Communication* comm) : Sensor(comm) {}
    void readAndSendData() override {
        // Simulación de lectura de datos del sensor de temperatura
        std::string data = "25.3 C";
        communication->sendData(data);
    }
};

// Abstracción refinada: Sensor de Humedad
class HumiditySensor : public Sensor {
public:
    HumiditySensor(Communication* comm) : Sensor(comm) {}
    void readAndSendData() override {
        // Simulación de lectura de datos del sensor de humedad
        std::string data = "60%";
        communication->sendData(data);
    }
};
```

Finalmente, dentro de nuestro main los objetos se crean empezando por las instancias, y luego los sensores se crean utilizando las instancias de comunicación.

```
// Crear instancias de los métodos de comunicación
   Communication* wifi = new WifiCommunication();
   Communication* bluetooth = new BluetoothCommunication();

   // Crear sensores con diferentes métodos de comunicación
   Sensor* tempSensorWifi = new TemperatureSensor(wifi);
   Sensor* humiditySensorBluetooth = new HumiditySensor(bluetooth);
   while(1){
       // Leer y enviar datos
       tempSensorWifi->readAndSendData();
       humiditySensorBluetooth->readAndSendData();
   }
```

5.3 Implementación del patrón adapter

El patrón puente, si bien no es muy común posee muchas similitudes con el patrón adaptador, y este aparece con más frecuencia. La diferencia principal radica en que el patrón adaptador adapta las interfaces que se necesiten utilizar, tales como un sensor antiguo, y esto se suele hacer de forma más directa. A continuación, se mostrará un ejemplo del patrón adaptador en un escenario común, que suele ser adaptar una interfaz de un sensor.

En la siguiente figura, se puede observar la clase de sensor de temperatura moderno, que será nuestro objetivo, pues es a lo que queremos llegar. También se puede observar la clase del sensor viejo, que es la clase que se adaptará.

```
// Interfaz objetivo: La que usa nuestra aplicación moderna
class ModernTemperatureSensor {
public:
    virtual float getTemperature() = 0;
};

// Clase Adaptada: El sensor antiguo con una interfaz diferente
class OldTemperatureSensor {
```

```
public:
    float getTempInFahrenheit() {
        // Simulación de lectura de temperatura en Fahrenheit
        return 77.0;
    }
};
```

Después se puede observar la clase adaptador, que es una especie de envoltura, y se encargará de convertir la interfaz antigua por lo que necesitamos.

```
// Adaptador: Convierte la interfaz del sensor antiguo a la interfaz moderna
class TemperatureSensorAdapter : public ModernTemperatureSensor {
private:
    OldTemperatureSensor* oldSensor;
public:
            TemperatureSensorAdapter(OldTemperatureSensor*   sensor)   :
oldSensor(sensor) {}

    float getTemperature() override {
        // Convierte la temperatura de Fahrenheit a Celsius
        return (oldSensor->getTempInFahrenheit() - 32) * 5.0 / 9.0;
    }
};
```

Finalmente, se puede observar su creación y uso en la parte principal del código.

```
// Crear una instancia del sensor antiguo
    OldTemperatureSensor* oldSensor = new OldTemperatureSensor();

    // Crear una instancia del adaptador
                    ModernTemperatureSensor*     adapter     =     new
TemperatureSensorAdapter(oldSensor);

    // Usar el adaptador como si fuera un sensor moderno
    float tempCelsius = adapter->getTemperature();
      std::cout  <<  "La  temperatura  en  Celsius  es:  "  <<  tempCelsius  <<
std::endl;
```

5.4 Implementación de múltiples patrones

Previamente, se mostró el uso de distintos patrones por separado, cada uno atacando un problema en particular. Sin embargo, los patrones también se pueden utilizar en conjunto, se pueden aplicar en conjunto para el problema del sensado, mientras que otro patrón se puede encargar de la comunicación y otro del procesamiento de datos, todo dependiendo de las necesidades del sistema.

En el código del Anexo A, se implementó el uso de factory para la creación de objetos de los sensores, strategy para poder cambiar de forma dinámica el procesamiento de cada sensor, observer para facilitar la notificación de los sensores para que se procesen sus datos, y singleton para una sola instancia de red.

Si bien a primera vista pudiera ser que el código creció innecesariamente y esto puede representar una gran desventaja, en el siguiente capítulo se mencionan algunas de las consecuencias de la utilización de patrones.

Capítulo 6: Prototipos y resultados

Una vez que se implementaron las aplicaciones con los diferentes patrones, se crearon prototipos con comunicación y sensores funcionales. Principalmente se utilizaron el Raspberry pi pico, raspberry pi pico W, ESP32-CAM, ESP32 Devkit y el es8266 NodeMCU,

Los microcontroladores usados para estos prototipos son bastante populares en este ámbito debido a su bajo costo y suficiente potencia para ejecutar estas tareas, ya que en ninguna ocasión nos vimos limitados por la capacidad de éstos.

Para facilitar la prueba de dichos prototipos, el código se desarrolló utilizando la plataforma de Arduino IDE, estos permitió poder crear un programa que fuese capaz de ejecutarse en ambos microcontroladores con mínimos ajustes.

Pasar del código de implementación de los patrones a un ejecutable para el microcontrolador se realizó sin mucho problema, fue aquí el momento en el que se comenzó a ver la utilidad de algunos patrones que permiten la modificación de una parte del código sin afectar al resto. Ya que, por ejemplo, para agregar la funcionalidad real del sensor de humedad DHT11, sólo se tuvo que actualizar la implementación concreta de dicho sensor, y el resto del código permaneció sin cambios.

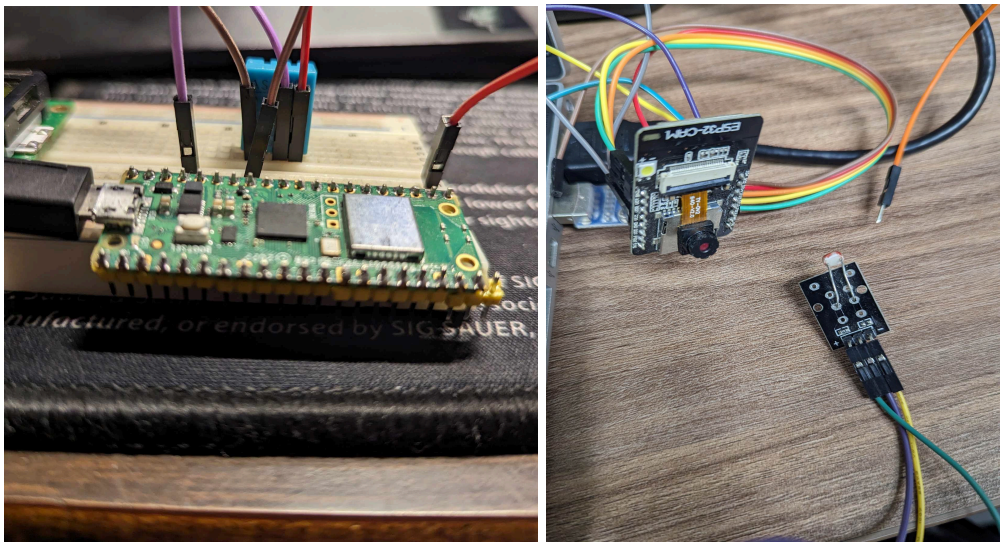

Figura 15, Se observan un par de prototipos utilizados, siendo un raspberry pi pico W con un sensor DHT11 y un ESP32-CAM con sensor de luminosidad.

Para poder evaluar los resultados obtenidos, es importante tener en cuenta cosas como, las consecuencias de utilizar el patrón, las prioridades de nuestro sistema y si vale la pena el porcentaje de mejora que se obtendrá a cambio de usar dicha solución.

A continuación, tenemos una tabla que nos ayuda a medir de forma cuantitativa la eficiencia de nuestro programa, que se le llama tabla de compensación.

	Design Criteria					
	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Criterion 5	
	Weight = 7	Weight = 5	Weight = 3	Weight = 2	Weight = 1.5	Total Weighted
Design Solution	Score	Score	Score	Score	Score	Score
Alternative 1	7	3	6	9	4	106
Alternative 2	4	8	5	3	4	95
Alternative 3	10	2	4	8	8	**120**
Alternative 4	2	4	9	7	6	84

Tabla 2, Compensación utilizada para obtener resultados cuantitativos [1].

Tomando en cuenta dicha tabla, de manera crítica, podemos asignar pesos a ciertos criterios. Tomemos el ejemplo de la aplicación que lee el sensor de temperatura y luminosidad y envía los datos por MQTT.

Al ser una aplicación sencilla y que no requiere de mucha memoria, podemos asignarle un valor bajo, que va de 0 a no importante hasta 10 muy importante, por lo tanto se le asigna un 3. Después, tendríamos velocidad de respuesta del sistema, si todo lo que hace el sistema es leer datos de sensor y enviarlos nos interesa que lo haga lo más rápido posible, pero tampoco es crítico si se atrasa por unos cuantos milisegundos, así que le asignaremos un 5. Finalmente podemos también considerar la escalabilidad, tomando en cuenta si en un futuro quisiéramos agregar otro sensor o algún otro tipo de comunicación, para efectos de demostración podemos decir que la escalabilidad es importante, pero no es la prioridad por lo que le asignamos un puntaje de 7.

En la siguiente imagen, se muestra el tiempo de ejecución y la cantidad de RAM libre en el sistema. Del lado izquierdo se tiene el programa programado con tipo super loop, y del lado derecho tenemos la combinación de 4 patrones de diseño.

Se puede observar que tanto el tiempo de ejecución como la memoria utilizada son menores para el programa que hace uso de patrones, a pesar de que se podría creer que utilizar patrones agregaría algún tipo de overhead que podría ser significativo. La diferencia en tiempo de ejecución es que a nuestro programa con patrones de diseño le toma ⅛ del tiempo de lo que necesita el otro programa, mientras que la diferencia en memoria es solamente de alrededor de un 10%.

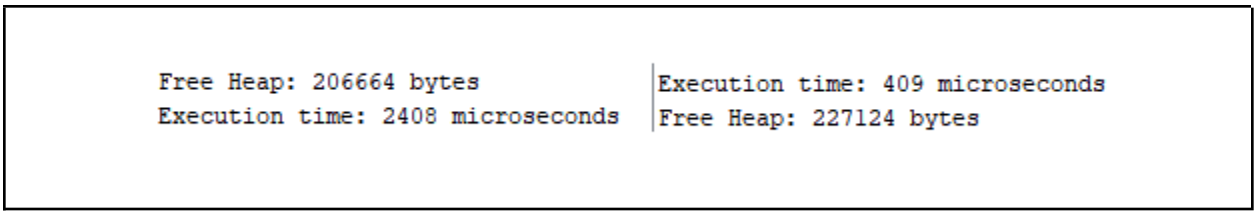

Figura 16, Métricas de ejecución de un programa que no hace uso de patrones (izq.) comparado con uno que sí (der.).

Finalmente la escalabilidad del sistema, dado que el programa que hace uso de patrones es naturalmente más escalable, lo consideraremos como una ligera ventaja.

Una vez revisado esto, le asignaremos un valor de 10 puntos a la velocidad de ejecución del programa de patrones, y 2 puntos al otro. Debido a la muy ligera diferencia en la memoria que utilizan, ambos reciben un 5. Para concluir, se le asignará un puntaje de 7 al programa hecho con patrones de diseño, pues los patrones utilizados no están enfocados en escalabilidad, pero si la mejoran un poco, y se le asignan 5 puntos al otro programa. En la siguiente tabla, se mostrará la alternativa 1 como la solución sin patrones y la 2 la solución con patrones. Los criterios se enumeran por peso, donde el criterio 1 es escalabilidad, el criterio 2 es la velocidad y el 3 el uso de memoria.

	Criterion 1	Criterion 2	Criterion 3	
	Weight = 7	Weight = 5	Weight = 3	Total Weighted
Design Solution	Score	Score	Score	Score
Alternative 1	5	2	5	60
Alternative 2	7	10	5	114

Tabla 3, compensación utilizando uno de los prototipos.

Con nuestro problema correctamente planteado y nuestras prioridades identificadas, estos resultados nos ayudan a decidir si el hacer uso de patrones o no nos dará una ventaja demostrable, además de que con el paso del tiempo, ya no es necesario hacer este tipo de tablas, pues rápidamente se pueden asignar valores a las cosas que queremos optimizar y recordar las cosas que han funcionado antes.

Es importante destacar que esta no es la única forma de estimar el rendimiento de una solución, en [1] se menciona que este tipo de análisis también se hace de

manera informal al pensarlo, o se puede traducir a tablas u otras herramientas como se mostró anteriormente.
De esto nos podemos llevar que nunca existirá una solución perfecta, siempre vamos a sacrificar rendimiento de algo por otro, y lo importante es tener un equilibrio entre optimizar lo que más nos interesa sin elegir otras características.

Evaluar todos y cada uno de los prototipos utilizando el método anteriormente mostrado con la tabla de compensación no sería prudente para el espacio de este trabajo académico, por lo que se resumirán en el siguiente capítulo.

Capítulo 7: Conclusiones

Hay mucho que nos podemos llevar de trabajar con patrones de diseño, podemos constantemente evaluar su posible impacto, desempeño e incluso el saber en dónde se van a encontrar.
Hubo algunos patrones que no fueron mencionados en este trabajo, y esto es porque puede que no se hayan utilizado casi ninguna vez, o el problema que resuelven no ocurre tan seguido dentro del área de aplicaciones de IoT en sistemas embebidos.
Sin embargo, hay patrones como el decorador, que aparecen en áreas como sensado y procesamiento si se requiere algún tipo de ajuste dinámico. El patrón fachada es un patrón que aparece de manera frecuente, pero su uso a veces pasa desapercibido, pero no es necesario que sea utilizado en sistemas muy simples. El patrón observador es uno de los que más vemos utilizarse, aunque también pasa desapercibido, pues el protocolo MQTT trabaja directamente de forma publish-subscribe. El iterador es otro patrón que aparece con frecuencia en muchas situaciones diferentes, sin embargo sus implementaciones varían dependiendo de la aplicación y los tipos de datos que se manejan. El patrón comando es útil para enviar información. El patrón composite también se puede encontrar en cuando hay objetos que se representan con algún tipo de jerarquía.

El tema de los patrones de diseño con IoT es algo que aún tiene mucho por madurar, y es un área que durante la investigación, se logró ver crecer mientras aprendíamos más de estas herramientas.

Durante la investigación se aprendió sobre el concepto de sistemas de patrones, que prácticamente es ir un paso más allá de la simple utilización y buscar combinarlos de diferentes maneras de tal forma que llegas a un sistema en el cuál, eliges y cambias patrones sabiendo qué es lo que se busca optimizar. Sin embargo, otro punto importante a considerar es el tener cuidado con querer utilizar

demasiados patrones sólo porque se puede, o utilizarlos en aplicaciones muy sencillas que no van a requerir ningún tipo de escalabilidad en el futuro, pues esto puede perjudicarnos en vez de darnos todos los potenciales beneficios que conlleva un correcto uso de ellos. A veces, la solución más sencilla es la mejor solución.

Finalmente, se ratifican los beneficios que tiene el uso de los patrones de diseño cuando se trata de las aplicaciones IoT en sistemas embebidos cuando son aplicados de manera correcta en las situaciones adecuadas.

Capítulo 8: Trabajo Futuro

Un trabajo a futuro potencial sería explorar el tema de los sistemas de patrones. Un sistema de patrones no es simplemente un conjunto de patrones, ya que se debe especificar la organización de estos para guiar a los usuarios a elegir los patrones correctos para aplicar, además de asegurarse que el sistema de patrones tiene posibilidades de modificarse y crecer [1].

Otro posible caso de estudio interesante sería el de explorar el uso de patrones de diseño en sistemas embebidos e IoT que implementen algún tipo de red neuronal o IA, ya que debido a la creciente popularidad de introducir IA en sistemas de bajo poder, debería haber una gran variedad de aplicaciones para analizar.

Referencias Bibliográficas

[1] Robert Oshana, (2013). *Software Engineering of Embedded and Real-Time systems*. Elsevier.

[2] *Design Patterns*. (2020). Event Helix. https://www.eventhelix.com/design-patterns/

[3] Christopher Alexander (1979). *The timeless way of building. Vol. 1*. New York: Oxford University Press.

[4] 1&1 IONOS Inc. (2021). What are design patterns? IONOS Digitalguide. https://www.ionos.com/digitalguide/websites/web-development/what-are-design-patterns/

[5] Design patterns. (2021). Refactoring Guru. https://refactoring.guru/es/design-patterns

[6] Douglass, B. (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (PAP/CDR ed.). Addison-Wesley Professional.

[7] Freeman, E., Bates, B., Sierra, K., & Robson, E. (2004). *Head First Design Patterns: A Brain-Friendly Guide* (1st ed.). O'Reilly Media.

[8] *What is the Internet of Things (IoT)?* (2021). Oracle. http://oracle.com/internet-of-things/what-is-iot/

[9] Clark, J. (2020, August 28). *What is the Internet of Things, and how does it work?* IBM Business Operations Blog. https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/

[10] *Patrón de diseño de software - EcuRed*. (2019). EcuRed. https://www.ecured.cu/Patr%C3%B3n_de_dise%C3%B1o_de_software

[11] Statista. (2021, October 19). Number of IoT connected devices worldwide 2019–2030. Retrieved October 19, 2021, from https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[12] G., N. (2021, October 2). How Many IoT Devices Are There in 2021? [All You Need To Know]. TechJury. Retrieved October 19, 2021, from http://techjury.net/blog/how-many-iot-devices-are-there/

[13] Weir, C. W., & Noble, J. N. (2000). *A Programmer's Introduction to Small Memory*.

[14] Leiner, B. M. L., Cerf, V. G. C., Clark, D. D. C., Kahn, R. E. K., Kleinrock, L. K., Lynch, D. C. L., Postel, J. P., Roberts, L. G. R., & Wolff, S. W. (2021, June 1). *Brief History of the Internet*. Internet Society. https://www.internetsociety.org/internet/history-internet/brief-history-internet/

[15] Johnston, P. (2023, December 22). Design Pattern Catalogue - embedded Artistry. Embedded Artistry. https://embeddedartistry.com/fieldatlas/design-pattern-catalogue/

[16] Moyer, B. (2020, August 3). Categories. Inside the IoT. https://www.insidetheiot.com/categories/

[17] Distributed systems explained | Splunk. (n.d.). Splunk. https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html

[18] Gamma, E., Johnson, R., Helm, R., Johnson, R. E. ., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH.

[19] Build software better, together. (n.d.). GitHub. https://github.com/topics/iot?l=c

[20] Build software better, together. (n.d.). GitHub. https://github.com/topics/iot?l=c%2B%2B

[21] Asghari, P., Rahmani, A. M., & Javadi, H. H. S. (2019). Internet of Things applications: A systematic review. Computer Networks, 148, 241–261. https://doi.org/10.1016/j.comnet.2018.12.008

[22]Qt Company. (n.d.). Embedded Software Programming Languages: Pros, Cons, and Comparisons of Popular Languages. https://www.qt.io/embedded-development-talk/embedded-software-programming-languages-pros-cons-and-comparisons-of-popular-languages

[23] Mischianti, R. (2023, December 12). DOIT ESP32 DEV KIT v1: high resolution pinout and specs. Renzo Mischianti. https://mischianti.org/doit-esp32-dev-kit-v1-high-resolution-pinout-and-specs/

[24] Bhuiyan, R. (2023, December 30). A complete pinout guide of Raspberry Pi Pico and Pico W: GPIOs explanation. Embedded There. https://embeddedthere.com/a-complete-pinout-guide-of-raspberry-pi-pico-and-pico-w-gpios-explanation/

Anexo A

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// Interfaz del observador
class Observer {
public:
    virtual void update(const std::string& data) = 0;
};

// Clase sujeto que notifica a los observadores
class Subject {
private:
    std::vector<Observer*> observers;
public:
    void addObserver(Observer* observer) {
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) {
           observers.erase(std::remove(observers.begin(), observers.end(),
observer), observers.end());
    }

    void notifyObservers(const std::string& data) {
        for (Observer* observer : observers) {
            observer->update(data);
        }
    }
};

// Singleton para la gestión de la red
class NetworkManager {
private:
    static NetworkManager* instance;
    NetworkManager() {}
```

```
public:
    static NetworkManager* getInstance() {
        if (instance == nullptr) {
            instance = new NetworkManager();
        }
        return instance;
    }

    void connect() {
        std::cout << "Conectando a la red..." << std::endl;
    }

    void disconnect() {
        std::cout << "Desconectando de la red..." << std::endl;
    }
};

// Inicialización del puntero estático
NetworkManager* NetworkManager::instance = nullptr;

// Interfaz de la estrategia de procesamiento de datos
class DataProcessingStrategy {
public:
    virtual void processData(const std::string& data) = 0;
};

// Implementaciones concretas de estrategias
class JsonDataProcessing : public DataProcessingStrategy {
public:
    void processData(const std::string& data) override {
            std::cout << "Procesando datos en formato JSON: " << data <<
std::endl;
    }
};

class XmlDataProcessing : public DataProcessingStrategy {
public:
    void processData(const std::string& data) override {
```

```
            std::cout << "Procesando datos en formato XML: " << data <<
std::endl;
    }
};

// Interfaz del sensor
class Sensor : public Subject {
protected:
    DataProcessingStrategy* strategy;
public:
    Sensor(DataProcessingStrategy* strategy) : strategy(strategy) {}

    void setStrategy(DataProcessingStrategy* newStrategy) {
        strategy = newStrategy;
    }

    virtual void readAndSendData() = 0;
};

// Implementaciones concretas de sensores
class TemperatureSensor : public Sensor {
public:
    TemperatureSensor(DataProcessingStrategy* strategy) : Sensor(strategy)
{}

    void readAndSendData() override {
        std::string data = "25.3 C"; // Simulación de lectura de datos
        strategy->processData(data);
        notifyObservers(data);
    }
};

class HumiditySensor : public Sensor {
public:
    HumiditySensor(DataProcessingStrategy* strategy) : Sensor(strategy) {}

    void readAndSendData() override {
        std::string data = "60%"; // Simulación de lectura de datos
        strategy->processData(data);
        notifyObservers(data);
```

```
    }
};

// Factory para crear sensores
class SensorFactory {
public:
             static    Sensor*    createSensor(const    std::string&    type,
DataProcessingStrategy* strategy) {
        if (type == "temperature") {
            return new TemperatureSensor(strategy);
        } else if (type == "humidity") {
            return new HumiditySensor(strategy);
        } else {
            return nullptr;
        }
    }
};

// Implementaciones concretas de observadores
class DataLogger : public Observer {
public:
    void update(const std::string& data) override {
        std::cout << "Registrando datos: " << data << std::endl;
    }
};

class DataAnalyzer : public Observer {
public:
    void update(const std::string& data) override {
        std::cout << "Analizando datos: " << data << std::endl;
    }
};

// Uso de todos los patrones combinados


void setup(){
    NetworkManager* network = NetworkManager::getInstance();
    network->connect();
```

```
}

void loop(){
    DataProcessingStrategy* jsonStrategy = new JsonDataProcessing();
    DataProcessingStrategy* xmlStrategy = new XmlDataProcessing();

        Sensor*  tempSensor  =  SensorFactory::createSensor("temperature",
jsonStrategy);
       Sensor*  humiditySensor  =  SensorFactory::createSensor("humidity",
xmlStrategy);

    DataLogger logger;
    DataAnalyzer analyzer;

    tempSensor->addObserver(&logger);
    tempSensor->addObserver(&analyzer);

    humiditySensor->addObserver(&logger);
    humiditySensor->addObserver(&analyzer);

    while(1){
      tempSensor->readAndSendData();
      humiditySensor->readAndSendData();
    }
}
```

Anexo B

```
// File: IObserver.h
#ifndef IOBSERVER_H
#define IOBSERVER_H

class IObserver {
public:
  virtual void update(float data) = 0;
};

#endif
```

```
// File: regularmqtt.ino
#include <Arduino.h>
#include "SensorDHT.h"
#include "SensorLight.h"
#include "SensorReader.h" // Include the new SensorReader.h
#include "SerialObserver.h"
#include <list>
#include <WiFi.h>
#include <AsyncMqttClient.h>

#define WIFI_SSID ""
#define WIFI_PASSWORD ""
#define MQTT_HOST IPAddress(192, 168, 0, 25)
#define MQTT_PORT 1

#define MQTT_PUB_TEMP "esp32/dht/temperature"
#define MQTT_PUB_HUM "esp32/dht/humidity"
#define MQTT_PUB_LIGHT "esp32/light"

std::list<SensorReader *> SensorReaders; // list of SensorReader objects
instead of ISensor

AsyncMqttClient mqttClient;

void connectToWifi() {
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void connectToMqtt() {
  mqttClient.connect();
}

void setup() {
  Serial.begin(115200);
  delay(1000);
  Serial.println("\nStart\n");

  // Create SensorReader objects for each sensor
```

```
   SensorReader *tempReader = new SensorReader(new SensorDHT(DHT11, 14),
"Temperature");
  SensorReader *lightReader = new SensorReader(new SensorLight(LIGHT, 26),
"Light");

  // Create and add observers
  SerialObserver *serialObserver = new SerialObserver();
  tempReader->addObserver(serialObserver);
  lightReader->addObserver(serialObserver);

  SensorReaders.push_back(tempReader);
  SensorReaders.push_back(lightReader);

  WiFi.onEvent([](WiFiEvent_t event) {
    switch (event) {
      case SYSTEM_EVENT_STA_GOT_IP:
        connectToMqtt();
        break;
      case SYSTEM_EVENT_STA_DISCONNECTED:
        if (mqttClient.connected()) {
          mqttClient.disconnect();
        }
        connectToWifi();
        break;
    }
  });

  mqttClient.setServer(MQTT_HOST, MQTT_PORT);
  mqttClient.setCredentials("twice", "twice");
  connectToWifi();
}

void loop() {
  static unsigned long previousMillis = 0;
  const long interval = 10000; // 10 seconds interval
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
```

```
    unsigned long startMicros = micros();
    unsigned long startHeap = ESP.getFreeHeap();


    Serial.println("Im Alive");

    for (SensorReader *reader : SensorReaders) {
      reader->readSensor();
           //Serial.println("Current  "  +  reader->getName()  +  ":  "  +
String(reader->getData(0)));

      // Publish sensor data to MQTT
      if (reader->getName() == "Temperature") {
                           mqttClient.publish(MQTT_PUB_TEMP,    1,    true,
String(reader->getData(0)).c_str());
                            mqttClient.publish(MQTT_PUB_HUM,    1,    true,
String(reader->getData(1)).c_str());
      } else if (reader->getName() == "Light") {
                          mqttClient.publish(MQTT_PUB_LIGHT,    1,    true,
String(reader->getData(0)).c_str());
      }
    }
    unsigned long endMicros = micros();
    unsigned long endHeap = ESP.getFreeHeap();

    Serial.print("Execution time: ");
    Serial.print(endMicros - startMicros);
    Serial.println(" microseconds");

    Serial.print("Free Heap: ");
    Serial.print(endHeap);
    Serial.println(" bytes");
  }

}

// File: Sensor.h
#ifndef ISENSOR_h
#define ISENSOR_h
```

```
#include <inttypes.h>
#define DHT11 0
#define DHT22 1
#define PRESSURE 2
#define LIGHT 3

#define DEFAULT 0
#define TEMP 0
#define HUM 1

class ISensor {
 public:
  virtual void read_sensor() = 0;
  virtual float get_data(int index) = 0;
  virtual byte get_type() = 0;
};

#endif
// File: SensorDHT.h
// File: SensorDHT.h
#include "Sensor.h"

#define DATA_SIZE 2

// TIPO DE DATO PARA RECIBIR EL FRAME DE DATOS DEL SENSOR
typedef union {
  uint64_t data;
  uint8_t frame[8];
} Frame;

class SensorDHT : public ISensor {
  private:
    volatile uint32_t duracion, tiempo_Fin, tiempo_Ini;
    volatile uint8_t i;
    Frame data_frame;
    float data[DATA_SIZE]; // 0:Temperatura  1:Humedad
    float temp, hum;
    float divisor;
    int flag; /*BANDERA DE VALIDACION DE DATOS*/
    int gpio_connect;
```

```
  byte sensor;

public:
  SensorDHT(byte sensor, int gpio_connect) {
    this->sensor = sensor;
    this->gpio_connect = gpio_connect;
  }

  void read_sensor1() {
    i = 0;
    data_frame.data = 0;
    delay(1500);
    start_host();
    get_frame();
  }
  void read_sensor(){
    data[0] = 1;

  }

  void get_frame(){
    int y = 8;
  }
  void start_host() {
    pinMode(gpio_connect, OUTPUT);
    digitalWrite(gpio_connect, LOW);
    delay(18);
    digitalWrite(gpio_connect, HIGH);
    delayMicroseconds(22);
  }

  void get_frame1() {
    pinMode(gpio_connect, INPUT);
    while (digitalRead(gpio_connect));
    while (!digitalRead(gpio_connect));
    while (digitalRead(gpio_connect));
    while (i < 40) {
      while (!digitalRead(gpio_connect));
      tiempo_Ini = micros();
      while (digitalRead(gpio_connect));
```

```
        tiempo_Fin = micros();
        duracion = tiempo_Fin - tiempo_Ini;
        data_frame.data <<= 1;
        if (duracion >= 10 && duracion <= 35) {
          //0
        } else {
          if (duracion > 35 && duracion <= 110) {
            data_frame.data += 1;
          } else {
            Serial.print("frame error");
          }
        }
        i++;
      }
      write_data();
    }

    void write_data() {
                if   (data_frame.frame[0]   ==   ((data_frame.frame[1]   +
data_frame.frame[2] + data_frame.frame[3] + data_frame.frame[4]) & 0xFF))
{
        if (sensor == DHT11) {
          flag = 1;
          hum = (float)(data_frame.frame[4]);
          temp = (data_frame.frame[2]);
          if ((hum < 10.0) || (hum > 90.0)) {
            flag = 0;
          }
          if (temp > 60.0) {
            flag = 0;
          }
        }
        if (sensor == DHT22) {
          flag = 1;
                        hum   =   ((float)((data_frame.frame[4]   <<   8)   +
data_frame.frame[3])) / 10.0;
          divisor = (data_frame.frame[2] & 128) ? -10.0 : 10.0;
                      temp   =   (((data_frame.frame[2]   &   127)   <<   8)   +
data_frame.frame[1]) / divisor;
          if (hum > 110.0) {
```

```
            flag = 0;
          }
          if ((temp < -50.0) || (temp > 135.0)) {
            flag = 0;
          }
        }
        if (flag) {
          data[0] = (float)temp;
          data[1] = (float)hum;
        }
      } else {
        // FAIL CHECKSUM
      }
    }

    float get_data(int index) {
      if (index < DATA_SIZE) {
        return data[index];
      }
      return 0;
    }

    byte get_type() {
      return sensor;
    }
};

// File: SensorLight.h
#include "Sensor.h"

class SensorLight : public ISensor {
  private:
    float data;
    int gpio_connect;
    byte sensor;

  public:
    SensorLight(byte sensor, int gpio_connect) {
      this->sensor = sensor;
      this->gpio_connect = gpio_connect;
```

```
    }

    void read_sensor() {
      data = (analogRead(gpio_connect)) / 4;
    }

    float get_data(int index) {
      if (index < 1) {
        return data;
      }
      return 0;
    }

    byte get_type() {
      return sensor;
    }
};
// File: SensorReader.h
#include <list>
#include "IObserver.h"

class SensorReader {
  private:
    ISensor *sensor;
    String name;
    std::list<IObserver *> observers; // Lista de observadores

  public:
        SensorReader(ISensor *sensor, String name) : sensor(sensor),
name(name) {}

    void addObserver(IObserver *observer) {
      observers.push_back(observer);
    }

    void removeObserver(IObserver *observer) {
      observers.remove(observer);
    }

    void notifyObservers(float data) {
```

```
      for (IObserver *observer : observers) {
        observer->update(data);
      }
    }

    void readSensor() {
      sensor->read_sensor();
      float data = sensor->get_data(0);
         notifyObservers(data); // Notificar a los observadores con los
nuevos datos
    }

    float getData(int index) {
      return sensor->get_data(index);
    }

    String getName() {
      return name;
    }
};

// File: SerialObserver.h
#include "IObserver.h"

class SerialObserver : public IObserver {
public:
  void update(float data) override {
    Serial.println("Updated sensor data: " + String(data));
  }
};
```