Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Organización Industrial

Estudio y aplicación de mejoras sobre la programación de un modelo de fábrica de Industria 4.0

Autor:

Élgar Lloret Pérez

Tutores:

Pedro Sanz Angulo

Jesús Galindo Melero

Departamento de Organización de Empresas y Comercialización e investigación de Mercados

Valladolid, julio de 2022

Resumen

El Departamento de Organización de Empresas y Comercialización e Investigación de Mercados de la Universidad de Valladolid ha adquirido un modelo físico de fábrica de Industria 4.0 con el objetivo de formar a sus alumnos. El modelo incluye una serie de estaciones mediante las cuales se simula el proceso completo de producción de una pieza (identificación, almacenamiento, procesado y clasificación). Para ello la fábrica cuenta con un conjunto de librerías programadas en C/C++ que se cargan sobre seis controladores.

En este trabajo se pretende llevar a cabo un estudio para la aplicación de mejoras sobre la fábrica, con el objetivo de que sirva de punto de partida para que próximos usuarios de la fábrica puedan implementar cambios basados en *Lean Manufacturing*. Para ello se lleva a cabo un estado del arte sobre las herramientas y métodos implementados en la fábrica, se explica la programación necesaria para la comunicación con sensores/actuadores y entre las estaciones (máquinas de estado finito, comunicación con sensores/actuadores y protocolo de comunicación MQTT).

Después, se pasa a explicar la programación de la estación sobre la que se ha centrado el estudio del código; la bahía de almacenamiento sobre la que se aplicarán cambios enfocados principalmente en lograr una mayor eficiencia en el proceso de almacenaje y búsqueda de piezas. Además, se incluyen algunos cambios que pueden resultar útiles para comprender parte del funcionamiento del resto de estaciones. Finalmente, se incluye un estudio económico para poner en valor el trabajo realizado.

Palabras Clave

Learning factory, industria 4.0, máquina de estados finitos, programación orientada a objetos,, protocolo de comunicación.

Abstract

The Department of Business Organization and Market Research of the University of Valladolid has acquired a physical model of an Industry 4.0 factory with the aim of training students. The model includes different stations to carry out a complete production process (identification, storage, processing, and sorting). For this purpose, the factory has a set of libraries programmed in C/C++ that are loaded in 6 controllers.

This work consists of a study in the programming with the aim of making improvements to formalize this work as a starting point for future users to implement methods based on Lean Manufacturing. The work starts with a state of the art of the tools and methods in the factory. After this, the programming of the communication methods (finite state machines, communication with sensors/actuators and MQTT communication protocol) is explained.

Then, the programming of the station in which the code study has been focused, the High Bay Warehouse, is explained. Different changes will be applied, mainly focused on achieving greater efficiency in the process of storing and part searching. In addition, some changes are included that may be useful to understand the other stations. Finally, an economic study is included to give value to the work done.

Keywords

Learning factory, industry 4.0, finite state machine, object oriented programming, comunication protocols

Agradecimientos

Este trabajo fin de grado cierra mi etapa en Valladolid. Una etapa en la que he aprendido y crecido mucho.

Quiero dar las gracias a mis entrenadores de triatlón, Rubén e Israel, y a mi entrenador de atletismo Uriel, por darme la oportunidad de vivir muchos años de práctica deportiva.

A mis compañeros de la residencia Río Esgueva, mi hogar durante ocho años.

A mi madre, Miguel y mi hermano, mis pilares fundamentales. Por acompañarme siempre, en los buenos y en los malos momentos.

A mis tutores, Pedro y Jesús, por su ayuda, paciencia y amabilidad.

Índice del Trabajo

Índice de Figuras

Índice de Código

Abreviaturas

CPS Cyber-Physical System

CDT C/C++ Development Tooling

DPS Delivery and Pickup Station

FIFO First In First Out

FSM Finite State Machine

HBW High-Bay Warehouse

IoT Internet of Things

JSON JavaScript Object Notation

MPO Multi-Processing Station

MQTT Message Queuing Telemetry Transport

NFC Near Field Comuniccation

PLC Programmable Logic Controller

PWM Pulse Width Modulation

POO Programación Orientada a Objetos

RFID Radio Frequency Identification

SLD Sorting Line Detection

SSC Sensor Station With Camera

VGR Vacuum Gripper Robot

Introducción

Justificación

La cuarta revolución industrial ha provocado cambios tecnológicos y humanos a una velocidad vertiginosa, generando la necesidad paralela de una nueva formación más práctica.

Recientemente, en el Departamento de Organización de Empresas y Comercialización e Investigación de Mercados de la Universidad de Valladolid se ha adquirido un modelo físico de fábrica de Industria 4.0. El modelo concreto es el "Learn Fabrik 4.0 9V" de Fischertechnik.

Mediante su utilización se pretende enseñar las tecnologías de la Industria 4.0 de una forma interactiva y en un entorno de aprendizaje que permita desarrollar nuevas ideas.

La fábrica reproduce el proceso completo de una línea de producción. Dicho proceso está completamente automatizado y digitalizado. Esto es posible gracias al uso de sensores y actuadores, controlados por un total de seis controladores que utilizan un protocolo de mensajería (MQTT, *Message Queuing Telemetry Transport*) para enviarse información entre ellos.

La conexión directa de la fábrica a internet, junto con la utilización de una nube propia permite seguir el proceso de fabricación, obtener datos de la estación ambiental y seleccionar los pedidos.

El modelo presenta la posibilidad de utilizar Node-Red, pero hace falta el uso de un dispositivo que soporte esta aplicación como, por ejemplo, una Raspberry Pi. Esta herramienta junto con el protocolo MQTT constituyen las dos grandes herramientas para poder configurar nuestra propia nube. Por ausencia de tiempo y recursos, no se ha podido implementar una interfaz propia mediante la herramienta Node-Red.

La fábrica está programada en C++ y cuenta con una programación compleja y extensa que ha sido desarrollada por programadores expertos de Fischertechnik. La programación del modelo de la fábrica se encuentra accesible en GitHub [1] y cuenta con una gran cantidad de librerías de código. El código viene apoyado de muy pocos comentarios y documentación, exceptuando los diagramas de las FSM (*Finite State Machine*, Máquina de Estados Finitos).

El modelo necesita de una gran cantidad de librerías auxiliares necesarias para el control de dispositivos como: el dispositivo NFC (*Near Field Comunication*, Comunicación de Campo Cercano), para el protocolo de comunicación MQTT; la librerías JSON (*JavaScript Object Notation*, Notación de Objeto de JavaSript) utilizada para codificar datos en formato JSON; la librería OpenCV (*Open Source Computer Vision*, Biblioteca Libre de Visión Artificial), que constituye una potente herramienta para el desarrollo de complejas aplicaciones de procesamiento de imágenes por ordenador; y la librería *spdlog*, utilizada para facilitar la detección de fallos en la depuración del código o *debugging*.

Con este trabajo se pretende sentar las bases para crear una fábrica de aprendizaje basada en el modelo físico, en la que se trabajen las tecnologías habilitadoras de la industria 4.0 (las ya implementadas y otras nuevas como, por ejemplo, un gemelo digital) en las que se integren mejoras basadas en Lean.

Pero antes se necesita conocer la programación, que constituye el primer obstáculo a superar para usuarios no expertos en software, que serán la mayoría de los usuarios de la fábrica. Para un ingeniero, llegar a comprender el código de la fábrica constituye una tarea verdaderamente compleja.

Para llevar a cabo el trabajo se han tenido que refrescar los conocimientos básicos de programación. Sin embargo, la mayor parte del código se ha comprendido a base de "perderse" entre las librerías y aplicar pequeñas modificaciones sobre el código; unas con éxito y otras sin, pero que han permitido marcarse pequeños objetivos y sobre todo mantener la motivación individual y la del equipo.

Es, por lo tanto, uno de los objetivos principales de este trabajo, el de ayudar a comprender más fácilmente la programación, para que los nuevos estudiantes que vayan a utilizar el modelo no partan de cero y tengan un documento en el que apoyarse para poder llegar a comprender gran parte del código principal de la fábrica.

Objetivos

Al principio los objetivos estaban centrados en alcanzar cambios más profundos que permitieran llevar a cabo modificaciones centradas en la metodología Lean, como por

ejemplo la implantación de un método *Heijunka* en la fábrica. Sin embargo, la compleja programación con la que nos hemos encontrado no nos ha permitido completar los objetivos que marcamos al principio y hemos tenido que buscar otros menos ambiciosos, pero que servirán para avanzar hacia otros más complejos en futuros trabajos fin de estudio. Los objetivos del trabajo han sido los siguientes.

Como primer objetivo, se ha planteado realizar un estudio de la programación de la fábrica para conocer sus librerías y funcionamiento. Dentro del estudio se debe prestar especial atención a la bahía de almacenamiento debido a dos motivos principales: constituye un cuello de botella del proceso y es la estación que mayor flexibilidad aporta en la implementación de cambios en la programación que ayuden a comprender su funcionamiento. Este objetivo exige documentar el código para que otros usuarios puedan avanzar más rápidamente en su comprensión.

Como segundo objetivo se propone implementar cambios en la programación que den lugar a mejoras en el proceso productivo y, a su vez, ayuden a comprender mejor el funcionamiento de la fábrica. La mayor parte de los cambios deberán centrarse en la bahía de almacenamiento por los motivos descritos anteriormente.

Como tercer objetivo, se ha considerado llevar a cabo la explicación teórica y práctica de algunas herramientas importantes para la comunicación entre dispositivos (Comunicación sensores/actuadores, comunicación MQTT, FSM (*Finite State Machine*, Máquina de estados finitos), por su utilidad como herramientas clave en la Industria 4.0 que permitan desarrollar nuevas mejoras basadas en *Lean Manufacturing* que puedan implementarse en futuros TFG.

Estructura de la Memoria

La memoria parte de un primer capítulo teórico en el que se explican las tecnologías y herramientas clave para poder entender el funcionamiento y programación de la fábrica.

En el segundo capítulo se lleva a cabo una explicación de algunas de las herramientas utilizadas en la fábrica para llevar a cabo la comunicación con los sensores y actuadores, para la programación de las máquinas de estado de las distintas estaciones de la fábrica y para la comunicación mediante el protocolo MQTT entre las estaciones.

El tercer capítulo es el foco principal del trabajo, ya que consiste en la documentación completa del funcionamiento del almacén y la explicación práctica de los cambios implementados sobre este. En este capítulo se explicarán los métodos que controlan el almacén de manera física y virtual. Después se incluirá la explicación de la máquina de estados del almacén y su comunicación con el robot de aspiración, mediante el protocolo MQTT, para poder llevar a cabo el proceso completo de almacenamiento de piezas. Al final de este capítulo se explicarán los cambios aplicados sobre el almacén, que se enfocarán principalmente, en conseguir una mayor eficiencia en el proceso de almacenamiento. Se añadirán algunos otros cambios sobre otras estaciones que han servido para entender parte del funcionamiento del resto de estaciones.

El último capítulo pretende poner en valor todo el trabajo realizado mediante la elaboración de un estudio económico.

El trabajo finaliza con los apartados de conclusiones y líneas futuras, y con las referencias bibliográficas. Por último, se añade en un anexo aparte, la explicación de la clase de la DPS, debido a que por falta de tiempo no se ha podido terminar la explicación completa de esta estación.

CAPÍTULO 1 Punto de partida

Este capítulo se centra en explicar las tecnologías, las herramientas, los materiales y los métodos que han sido necesarios para desarrollar el trabajo fin de grado.

1 La Industria 4.0 y las tecnologías habilitadoras

El concepto de industria 4.0 nace como un nuevo concepto en el marco académico e industrial por parte de consultoras alemanas en el CeBit de Hannover (2013) como una evolución natural de la Industria 3.0 [2], con el objetivo de reforzar la competitividad del sector manufacturero del país, mediante la integración de las tecnologías de la producción junto con las Tecnologías de Información y Comunicación (TICS). La industria 4.0 no es un solamente un cambio tecnológico, supone un cambio de filosofía y de estructura económica.

Es la industria de la sostenibilidad y las energías limpias, de los nuevos materiales técnicos reciclables, de los diseños ecológicos (ecodiseños) y el diseño colaborativo (codiseño). La industria 4.0 impulsa una integración total de la información y el conocimiento en todas las etapas del ciclo de vida del producto en el entorno de la sociedad digitalizada.

La mayor parte de cambios se produce en los sistemas de producción [2] con la Inteligencia Artificial (IA), la robótica, los sensores y las comunicaciones inalámbricas que, en combinación con el análisis Big Data y la computación en la nube, permiten máquinas autónomas y sistemas inteligentes [3]. En el modelo de Industria 4.0 tanto empresas, que participan en el ciclo del producto, como departamentos internos y funciones de diferentes procesos deben estar plenamente cohesionadas, compartiendo

datos esenciales para optimizar cada etapa de la cadena de valor, mediante procesos inteligentes alojados en la nube (*cloud computing*, computación en la nube) con capacidad de tomar decisiones mediante inteligencia artificial.

Las tecnologías de la Industria 4.0 deben estar presentes en todas las etapas del ciclo vital del producto. El objetivo es llegar a una gestión integral que conecte todas las etapas en un ciclo cerrado que pase por las etapas de diseño, fabricación, distribución, adquisición del producto, uso del producto y reciclaje del producto (Figura 1).

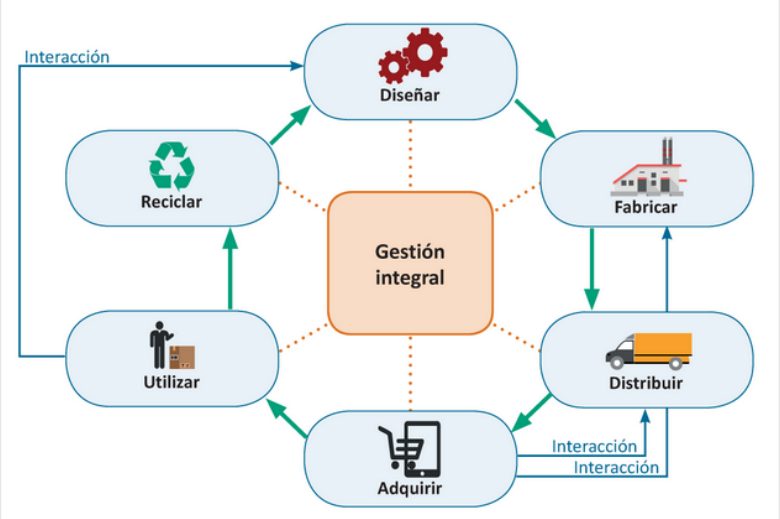


Figura 1. Etapas del ciclo de vida del producto [2].

Los avances en la robótica han sido clave en la revolución de la industria 4.0, y ha cambiado completamente los ambientes de trabajo en las fábricas. La tendencia clara en el ámbito industrial es avanzar en la automatización de los procesos y el incremento de la robótica colaborativa.

Otra de los procesos clave es la utilización de simulaciones en tiempo real de las operaciones de planta; mediante datos capturados en tiempo real se consigue reflejar el mundo físico en un modelo virtual. Además, se ha dado un paso más allá, con la llegada la realidad virtual y la realidad aumentada a las fábricas y su uso como forma de aprendizaje inmersivo de procesos.

En cuanto a la logística interna, está marcada por un fuerte impulso en la automatización de los sistemas de almacenamiento. Entre los requisitos a los que la logística 4.0 debe responder se encuentran [2]: la trazabilidad, tanto de materias primas como del producto final, mediante el uso de etiquetas RFID (*Radio Frequency Identification*, Identificación por Radio Frecuencia) y de Software para la gestión de almacenes, que controle, coordine y optimice los movimientos del almacén [4]. La flexibilidad para adaptar la producción a la demanda cambiante y la eficiencia (minimizando las distancias en el transporte).

En resumen, las principales necesidades a considerar en el marco de la Industria 4.0 son las siguientes [2]:

- Ajustar en tiempo real la planificación de la producción mediante la integración vertical de la información sobre las ventas mediante software CRM (*Customer*

Relationship Management, Gestión de relaciones con clientes), las devoluciones, las existencias en el almacén y los posibles problemas de calidad.
- Gestión automatizada del almacén y en el aprovisionamiento de materias primas.
- Actualización automática de incidencias con los proveedores y en la continua comprobación de la calidad durante la producción.
- Optimización de la logística interna minimizando cuellos de botella (suministros internos de materias primas y piezas) mediante un software adecuado, robots especializados y vehículos autónomos AGV (*Automated Guided Vehicles*, Vehículos de guiado automatizado).
- Siempre que se pueda se implementará la fabricación aditiva (impresoras 3D).
- Los trabajos peligrosos en la cadena de producción los realizarán robots, con lo que también se consigue optimizar las tareas repetitivas. Los robots industriales programables, dotados de visión artificial y todo tipo de sensores permiten aumentar la productividad.
- Se automatizará el control de calidad dotando de capacidades de discriminación de posibles errores mediante sensores ubicados estratégicamente en ciertos puntos de la cadena.
- Se optimizará el mantenimiento preventivo mediante la detección automatizada de las condiciones de trabajo de la maquinaria más crítica, evitando averías y parones en la producción.
- La realidad aumentada para la supervisión de la fabricación, el mantenimiento de las máquinas, el tratamiento de incidencias y para ayudar a nuevos trabajadores a entender los procesos de la fábrica.
- La integración horizontal entre las diferentes etapas del proceso se obtiene mediante el internet de las cosas y la integración vertical entre los diferentes componentes del ERP, mediante la computación en la nube.
- La robotización del embalaje (*packaging*) para la trazabilidad posterior de los productos.

2 La educación para la Industria 4.0

La Industria 4.0 plantea un nuevo escenario, en el que el conocimiento y el capital humano son imprescindibles. Ante un entorno tan rápidamente cambiante se ha de adaptar a nuevas formas de aprendizaje, tanto en las empresas como en el ámbito de la educación, fomentando las habilidades y los incentivos para aprender, y reduciendo las distancias entre las empresas y universidades.

La nueva formación deberá dirigirse en tres ejes [2]. El primero consistirá en una modificación completa del entorno docente, asumiendo la digitalización y la cooperación entre el sistema educativo y el sistema productivo. El segundo eje se refiere a la utilización de las herramientas adecuadas, mediante el uso de sistemas expertos y simuladores. El tercero consistirá en la incorporación de tutores personales sustentados por inteligencia artificial, con el objetico de desarrollar metodologías adaptadas a las características de cada persona.

La necesidad de realizar cambios en los métodos docentes hace necesaria la puesta en práctica de un modelo de aprendizaje activo. Este aprendizaje (Figura 2) engloba un

aprendizaje orientado a la acción, mediante la integración del alumno en el proceso de aprendizaje a través de sus acciones y experiencia; un aprendizaje experimental y basado en juegos; un aprendizaje basado en problemas, en el que partan de un problema como estímulo para resolver una determinada situación; un aprendizaje basado en proyectos, en el que los alumnos trabajen en grupos; y un aprendizaje basado en la investigación [5].

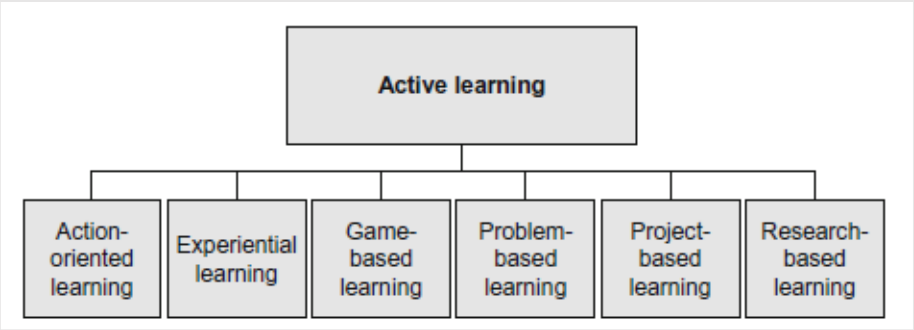


Figura 2. Métodos de Aprendizaje Activo [5].

Tanto en Universidades como en centros de formación se debe tener en cuenta los actuales y futuros perfiles laborales demandados por las empresas [6], sobre todo en entornos innovadores, industriales y tecnológicos, para poder adaptar continuamente la educación a las expectativas del mercado. Las exigencias de la industria se centran en una formación que combine tanto la educación como la experiencia en las fábricas de aprendizaje.

Las fábricas de aprendizaje se centran en que los participantes desarrollen sus capacidades, incluyendo la motivación para hacer frente a situaciones complejas y desconocidas.

3 Las fábricas de aprendizaje de Industria 4.0

Las fábricas de aprendizaje o "Learning Factories" constituyen una herramienta muy valiosa para llevar a cabo proyectos de aprendizaje, investigación y transferencia en el marco de la industria 4.0. Su capacidad para llevar a pequeña escala un sistema complejo para el que se requiere de diversos conocimientos ingenieriles (mecánica, electrónica, software…) hacen de estas una herramienta verdaderamente útil para el estudio práctico de las tecnologías de la Industria 4.0 en laboratorios de Universidades, Formación profesional y en la propia industria. Mediante este tipo de fábricas a pequeña escala se pueden simular, aprender e implementar actividades de digitalización a pequeña escala, antes de aplicarlas a una escala mayor en cualquier empresa.

En 2013, los miembros de la iniciativa de la IELF (*International Association of Learning* Factories, Asociación Internacional de Fábricas de Aprendizaje) se pusieron de acuerdo para establecer una definición sobre el concepto de fábrica de aprendizaje [6]: “Una fábrica de aprendizaje es un entorno de aprendizaje donde los procesos y las tecnologías se basan en un lugar industrial real que permite un acercamiento directo al proceso de creación de productos. Las *Learning Factories* se centran en un concepto didáctico que

enfatiza la experiencia y los problemas basándose en el aprendizaje. La mejora continua se ve facilitada por las acciones propias y la participación interactiva de todos los participantes.”

4 La fábrica de Aprendizaje "Learn Fabrik 4.0 9V" de Fischertecknik

La Industria 4.0 está revolucionando la forma en que las empresas fabrican, mejoran y distribuyen sus productos [7]. Los fabricantes están integrando nuevas tecnologías en sus instalaciones de producción y en todas sus operaciones: IoT (*Internet of Things*, Internet de las Cosas), computación y análisis en la nube, inteligencia artificial y *machine learning*, simulación, big data, … Estas fábricas inteligentes están equipadas con sensores avanzados, software integrado y robótica que recopilan y analizan datos, y permiten una mejor toma de decisiones.

Estas tecnologías digitales conducen a una mayor automatización, mantenimiento predictivo, optimización automática de mejoras de procesos y, sobre todo, un nuevo nivel de eficiencia y capacidad de respuesta a los clientes que antes no era posible [7]. Pero para lograr estas ventajas, es necesario contar, a todos los niveles de producción, con conexiones robustas e información inteligente, lo que se logra a través de una correcta digitalización.

Mediante la fábrica de aprendizaje 4.0 de FischerTechnik (Figura 3) se pueden simular, aprender e implementar estas actividades de digitalización a pequeña escala, antes de aplicarlas a una escala mayor en cualquier empresa [8].

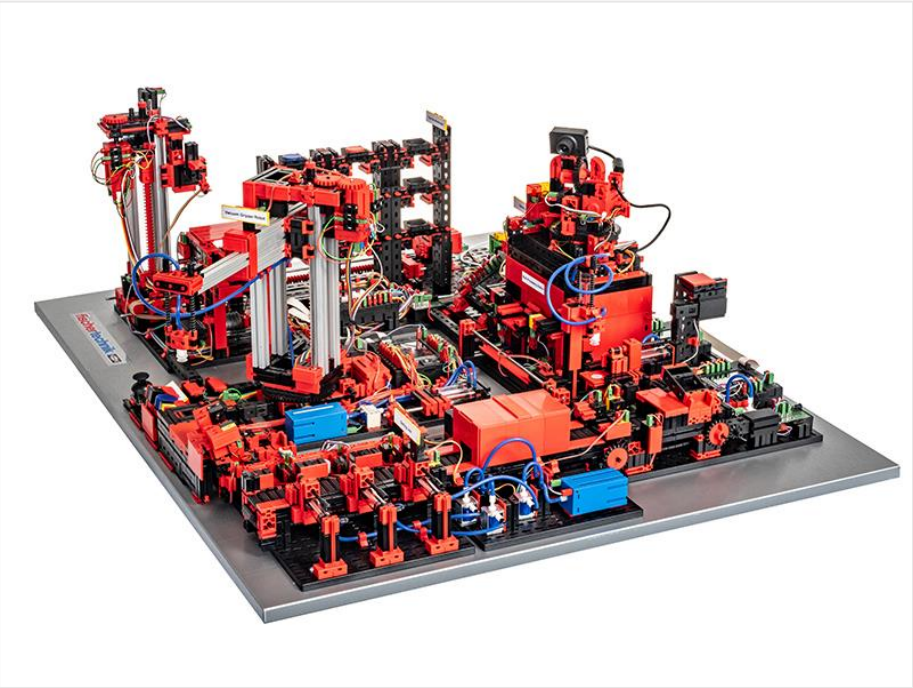


Figura 3. Vista Global de la Fábrica de Aprendizaje de Fishertechnik [8].

En concreto, la fábrica de aprendizaje 4.0 de FischerTechnik permite simular el ciclo de un pedido (realización del pedido, producción y entrega) a través de distintas actividades que están conectadas y digitalizadas. Se trata de un modelo físico de simulación y aprendizaje modular, económico y robusto que se puede utilizar de manera muy práctica.

El modelo de fábrica se compone de varios módulos: estación de entrada y salida, manipulador de aspiración por vacío, estación de almacenamiento y traslado, multiestación de procesamiento con horno de cocción, cinta de clasificación con reconocimiento de color, sensor ambiental y cámara de vigilancia giratoria.

A continuación, vamos a resumir su funcionamiento. En primer lugar, cuando entra una pieza (materia prima) en la fábrica por la estación de entrada, el manipulador la recoge y se la lleva al brazo robotizado del almacén (transelevador), que la almacena en una celda de la estantería.

Si desde el panel de control de la nube se realiza un pedido, la pieza es transportada desde el almacén hasta la estación de multiprocesamiento, donde es sometida una operación de cocción y fresado. A continuación, las piezas procesadas son ordenadas por color en la cinta de clasificación y, finalmente, transportadas al punto de expedición.

Puede seguirse el estado actual de la fábrica en todo momento en el panel de control. Además, el sensor ambiental integrado informa sobre los valores de temperatura, así como de humedad, presión y calidad del aire. También es posible observar la instalación mediante una cámara que puede girar vertical y horizontal, lo que es muy útil para realizar un mantenimiento remoto.

Las piezas de trabajo individuales son rastreadas por medio del dispositivo NFC cada pieza recibe una identificación para poder luego asociarle un tipo y un estado a la pieza. Esto hace posible seguir el estado de la pieza en todo momento.

La fábrica se controla por medio de seis controladores TXT Controller de FisherTechnik, alimentados mediante tres fuentes de 9 V. Estos controladores se comunican entre sí mediante el protocolo de mensajes MQTT que hace posible el intercambio de datos en forma de mensajes entre los dispositivos.

Por último, hay que señalar que las aplicaciones de software están escritas en C/C++ y cargadas en los controladores listas para funcionar. La biblioteca correspondiente C/C++ y la API están disponibles en GitHub [1]. Con la biblioteca es posible escribir programas propios en C/C++ para la fábrica de aprendizaje.

5 Protocolos de Comunicación en Industria 4.0

El IoT es un paradigma basado en la interconexión de dispositivos, para la que es necesaria una conexión a internet. Aunque hay muchas formas de realizar la comunicación M2M (*Machine to Machine*, Máquina a Máquina), en el campo del IoT

existen ciertos requerimientos que son necesarios para poder implementar cualquier protocolo de comunicación, entre los que se encuentran los siguientes [9]:

- La comunicación debe dar **soporte** a gran **cantidad y variedad de dispositivos**, desde pequeños como sensores y actuadores hasta otros más grandes como son los servidores.
- El sistema debe permitir la **escalabilidad**. La conexión y desconexión de dispositivos debe ser dinámica y sin alterar el comportamiento global del sistema.
- Débil **acoplamiento** entre dispositivos. La dependencia entre estos debe ser la menor posible.
- La comunicación debe garantizar la **interoperabilidad**. La conexión debe funcionar con la mayor variedad de dispositivos, sistemas operativos y lenguajes de programación.
- Para garantizar la **rapidez** en la respuesta y un gran número de conexiones simultáneas se requiere de mensajes pequeños y de poco procesamiento.
- Un aspecto clave es la **seguridad**. Los dispositivos están conectados a internet. La transmisión de datos y control de dispositivos se debe efectuar de forma totalmente privada.
- Otro requisito muy importante es la **facilidad** de **acceso** de los dispositivos. Se deberá lidiar con direcciones dinámicas y DHCP.

5.1 Infraestructuras de Servicios en IoT

Entre las metodologías más habituales que podemos encontrar en IoT destacan **Pub/Sub (***Publish/Subscriber*, Publicador/Suscriptor**)** y **rRPC (***Router Remoder Procedure Calls*, Llamada a Procedimiento Remoto).

La metodología *Pub/Sub* es un patrón de mensajería en el que el agente "Publicador" publica mensajes y el agente "Suscriptor" recibe los mensajes mediante un intermediario denominado bróker.

La metodología *rRPC* consiste en un patrón de ejecución remota de procedimientos [10]. En esta metodología, un agente llamado '*Callee*' se comunica con el rúter, el cual proporciona un cierto procedimiento, y el otro agente llamado '*Caller*' puede llamar a este procedimiento. Cuando el rúter invoca el procedimiento en el *Callee* se recoge el resultado del proceso y comunica al *Caller* que lo ha invocado.

Para llevar a cabo estas metodologías existen dos infraestructuras principales: MQ (*Message Queue*, Cola de Mensajes) y MS (*Message Service*, Servicio de Mensajes)

En la infraestructura MQ (Figura 4) el bróker genera una cola de mensajes única para cada uno de los clientes que inician la subscripción y estos mensajes se mantienen en el rúter hasta que el cliente se conecte. El bróker discrimina los mensajes empleando un ID de cliente, aunque también existen mecanismos para distribuir a múltiples clientes.

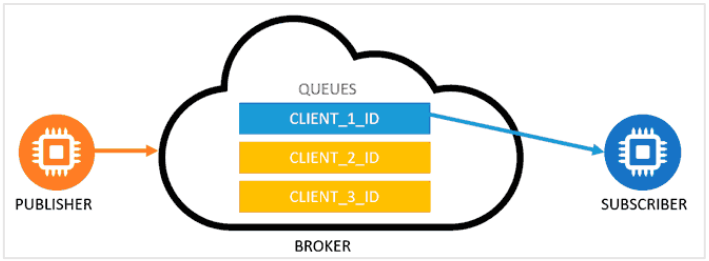


Figura 4. Infraestructura de Message Queue .

En la infraestructura MS (Figura 5) el bróker distribuye los mensajes en el tiempo en el que el cliente está conectado, filtrando los mensajes en base a un criterio como el tema o el contenido de cada mensaje.

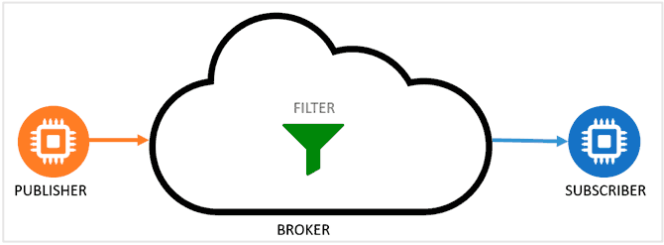


Figura 5. Infraestructura de Message Service.

El MQTT (MQ Telemetry Trasnsport) es un protocolo *Pub/Sub* de MS que actúa sobre el protocolo TCP/IP[1]. Destaca por ser ligero y sencillo en su implementación, resultando apropiado para dispositivos de baja potencia como son los frecuentemente utilizados en IoT.

5.2 Protocolo MQTT

El protocolo MQTT es un protocolo de mensajería con un patrón de diseño publicador/suscriptor (Pub/Sub) [11]. Este patrón presenta un diseño de programación orientado a eventos en el cual hay unas clases que se suscriben (suscriptor) y/o publican (publicador) a un tema (*topic*).

El protocolo MQTT se caracteriza por tener un diseño ligero, simple y de código abierto. Sus características lo hacen idóneo para comunicaciones de tipo M2M y en las conexiones *IoT*.

Una de las características principales del MQTT es el desacoplamiento publicador-suscriptor [12]. Cada cliente puede ser publicador, suscritor o ambos al mismo tiempo [3]. Mediante este desacoplamiento los publicadores no envían los mensajes directamente a un suscriptor concreto, sino que se clasifican en temas y cada suscriptor se suscribe a los temas que le interesan. Este desacoplamiento se produce mediante un bróker intermedio, que es el encargado de filtrar los mensajes y enviarlos a los suscriptores que están suscritos al tema.

[1] TCP (*Transmision Control Protocol*, Protocolo de Control de Transmisión) /IP (Internet Protocol, Protocolo de Internet) es una familia de protocolos de red en los que se basa internet y que permiten la transmisión de datos entre equipos. Existen más de cien protocolos diferentes, entre los que se encuentran FTP, POP, SMTP, DNS, DHCP, …

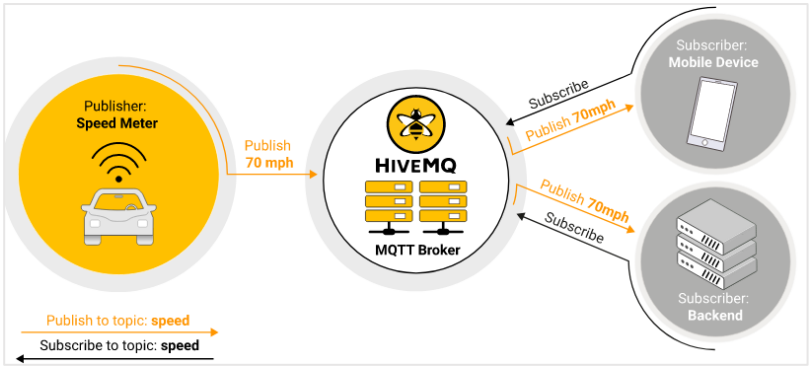


Figura 6. Patrón Publicador Suscriptor [13].

Un cliente MQTT es cualquier dispositivo (desde un pequeño controlador como los de la fábrica hasta un servidor) que implementa una librería MQTT y se conecta con un bróker MQTT a través de una red de internet mediante protocolo TCP/IP [14]. Para publicar o recibir mensajes los clientes necesitan conocer el nombre del host (IP) y el puerto del bróker.

En el protocolo MQTT la comunicación no se realiza directamente entre los clientes, sino que se realiza mediante una conexión intermedia con un bróker. El bróker MQTT es el responsable de recibir los mensajes, filtrarlos (determinando quien está suscrito a cada tipo de mensajes) y enviar los mensajes a aquellos clientes que están suscritos.

Los clientes inician una conexión TCP/IP con el bróker, quien mantiene un registro de los clientes conectados. Una vez establecida la conexión el bróker la mantiene abierta hasta que el cliente la finaliza.

El rango de acciones que puede efectuarse entre el cliente y el bróker son principalmente cinco: conectarse, publicar, suscribirse, ping y desconectar.

- **Conectarse**: El cliente se conecta con el bróker enviando a este un mensaje CONNECT que contiene información necesaria para la conexión como nombre de usuario, contraseña e identificador del cliente. El bróker responde mediante un un mensaje tipo CONNACK con el resultado de la conexión (aceptada, rechazada, etc.).

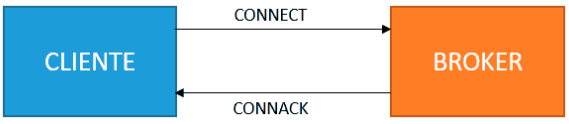


Figura 7. Mensaje MQTT CONNECT/CONNACK [15]

- **Publicar**: Se publica un mensaje sobre un determinado tema. El mensaje puede ser algo tan simple como una indicación de encendido/apagado o un valor de un determinado sensor, como temperatura, presión, etc. Si el tema publicado no existe se crea en el bróker.

Figura 8. Mensaje MQTT PUBLISH [15]

- **Suscribirse**: Un cliente se suscribe a un determinado tema. Se puede suscribir a un tema concreto o suscribirse a toda una rama de temas o a parte de ella. Para suscribirse el cliente envía un mensaje SUBSCRIBE y recibe un paquete SUBACK en respuesta para indicar si ha sido posible la suscripción. Si hay un mensaje retenido para el tema, el nuevo suscriptor también lo recibe.

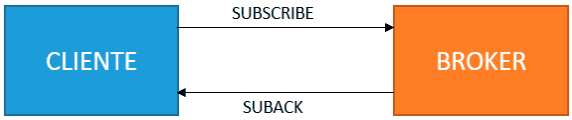


Figura 9. Mensaje MQTT SUBSCRIBE/SUBACK [15]

- **Ping**: Para garantizar que la conexión sigue funcionando el suscriptor envía un paquete PINGREQ y, como respuesta, se recibe un paquete PINGRESP.
- **Desconectar**: Se puede enviar un mensaje de DISCONNECT para desconectar un suscriptor o un editor, informando al bróker de que ya no se necesita enviar o poner a cola mensajes y que ya no recibirá datos de un editor. Este tipo de desconexión permite al cliente volver a conectarse utilizando el mismo identificador de cliente que en ocasiones anteriores. Cuando un cliente se desconecta sin enviar un mensaje de desconexión, se envía su última voluntad y testamento a los suscriptores.

5.3 Estructura de un Mensaje MQTT

La ligereza y agilidad del protocolo MQTT no sería posible sin una definición y topología de mensajes que hagan posible minimizar sus transmisiones mediante mensajes pequeños y bien definidos. Cada mensaje consta de tres partes y su carga útil está limitada a 256 MB.

Cada mensaje cuenta con una **cabecera fija** que ocupa de 2 a 5 bytes y es obligatoria. La cabecera cuenta con un código de control de 1 byte para identificar el tipo, y un código de 1 a 4 bytes para indicar la longitud del mensaje, de los cuales se emplean los 7 primeros bits y el último es un bit de continuidad.

Seguidamente está la **cabecera variable** que es opcional y contiene la información adicional que es necesaria en determinados mensajes o situaciones. Por último está el **contenido** del mensaje (payload)

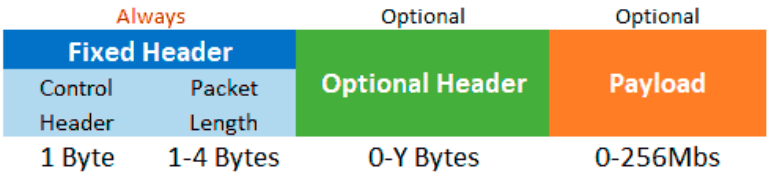


Figura 10. Tipología de los mensajes del protocolo MQTT [15].

Aunque los mensajes más usados son los explicados en el epígrafe anterior el protocolo MQTT tiene 14 tipos de mensajes. Se pueden ver en la siguiente tabla junto a el código de control que identifica el tipo de mensaje.

Message	Code
CONNECT	0x10
CONNACK	0x20
PUBLISH	0x30
PUBACK	0x40
PUBREC	0x50
PUBREL	0x60
PUBCOMP	0x70
SUSCRIBE	0x80
SUBACK	0x90
UNSUSCRIBE	0xA0
UNSUBACK	0xB0
PINGREQ	0xC0
PINGRESP	0xD0
DISCONNECT	0XE0

Tabla 1. Tipos de mensajes MQTT.

6 El Lenguaje de la Fábrica: C++

C++ es un lenguaje de programación que fue desarrollado por Bjarne Stroustrup a principios de los 80 como una versión aumentada del lenguaje C, manteniendo al mismo tiempo la potencia y elegancia de C junto con la inclusión de las características propias de la POO: construcción de clases, herencia y sobrecarga de funciones y operadores entre las más importantes [16]. Es por esto por lo que C++ no es un lenguaje puro de POO .sino que se trata de un lenguaje multiparadigma, pues permite otros tipos de programación como la programación estructurada, procedimental, y genérica [17]. Es lo que hace que este sea realmente un lenguaje muy potente y utilizado hoy en día.

C++ es un lenguaje altamente eficaz para lograr el propósito para el que está diseñado la fábrica. Mediante este tipo de programación es posible crear un código que permita representar problemas del mundo real, mediante un lenguaje cercano y que permite crear jerarquías de objetos con la creación de librerías que permitan después ser usadas por otros programadores. Además, se trata de una programación altamente flexible y ampliable, pues permite la implementación de nuevas características añadiéndose estas a la estructura de un objeto ya existente.

Antes de empezar con la programación de la fábrica se recomienda leer un manual de programación en C++ como, por ejemplo, [18] y [19].

6.1 Programación Orientada a Objetos.

La Programación Orientada a Objetos (POO) es un modelo de programación que consiste en la resolución de problemas en base a la abstracción. La **abstracción** consiste en una propiedad mediante la cual podemos describir un objeto centrándonos en algunos aspectos y olvidando deliberadamente otros aspectos que no son importantes para realizar aquello que queremos hacer.

Todos los lenguajes proporcionan abstracciones [20]. El lenguaje ensamblador es una pequeña abstracción del lenguaje de la máquina subyacente. Los primeros lenguajes de programación (Fortran, Basic…) eran igualmente una abstracción del lenguaje ensamblador pero que aun siendo fruto de importantes mejoras seguían siendo lenguajes de abstracción primaria que requerían un pensamiento en términos de la estructura del ordenador y no del problema a resolver.

Normalmente se piensa en un programa como un conjunto de instrucciones que indican a la máquina qué hacer ante un problema determinado. La POO pretende organizar los programas a imagen y semejanza de la organización de los objetos en el mundo real, entendiéndose cualquier problema como un conjunto de objetos que dialogan entre sí para resolver cualquier tipo de tarea [21].

Un objeto se puede definir como una entidad individual que se caracteriza por un estado y un comportamiento[17]. Los objetos son miembros de **clases**. Se puede decir que la clase es una plantilla que se utiliza para crear un objeto [22]. En el diseño de un programa orientado a objetos se parte del diseño de las clases, mediante su declaración (qué hace) e implementación (cómo lo hace).

De esta forma, la POO permite describir el problema en términos del problema, consiguiéndose crear programas que se adaptan al lenguaje del problema. Este tipo de programación es fundamental para poder entender el lenguaje que utiliza la fábrica para comunicarse, un conjunto de objetos (Almacén, Robot, Estación de Clasificación, …) que realizan una serie de funciones y mantienen una comunicación mediante mensajes que permiten coordinar el conjunto de tareas. Por ello, un objeto no necesita conocer el funcionamiento interno de otro objeto para poder interactuar con este, sino que es suficiente con conocer la forma en que debe enviarle mensajes y la forma en que se recibirá una respuesta [19].

La idea fundamental de la POO es la de combinar en una única unidad o módulo tanto los datos como las funciones que operan sobre estos datos [17]. Las funciones de un objeto, que son la única forma de acceder a los datos de un objeto, reciben el nombre de **funciones miembro** o **métodos**. Los datos de un objeto se conocen también como **atributos** o **variables de instancia.**

Las tres características principales de la POO son la encapsulación, la herencia y el polimorfismo. La **encapsulación** se implementa mediante la creación de tipos definidos por el usuario. Una clase funcionará como una clase totalmente encapsulada de forma que se utilizará como una entidad completa o no se utilizará [22].

Por su parte, la **herencia** permite declarar un nuevo tipo de dato que sea una extensión de uno ya existente: se dice que es una subclase de un tipo existente [22]. Pongamos el caso en el que se desean añadir nuevas funcionalidades sobre una Clase, pero heredándose las funcionalidades que tenía en esencia. Mediante esta característica se permite que distintos objetos que desempeñan funciones parecidas o tienen características similares puedan compartir información evitando la redundancia en la programación

Por último, el **polimorfismo** se refiere a la capacidad de un objeto para realizar comportamientos que no están definidos en esa misma clase, sino en otras clases que se derivan de ella. Una misma clase puede adoptar distintas formas en función de las redefiniciones presentes en las clases que se derivan de ellas.

6.2 Las Clases

En un lenguaje Orientado a Objetos el centro del lenguaje no son las funciones, sino los objetos, que contienen datos y funciones concretas que permiten manipularlos y trabajar sobre ellos.

Un objeto es la instancia de una clase. Por ejemplo, un objeto de tipo 'pieza de trabajo' es una instancia de la clase *TxtWorkPiece*. Lo que diferencia una pieza de otra son los valores que toman sus atributos. Lo que identifica una pieza es un identificador, su tipo (color de la pieza) y su estado (materia prima "RAW", producto final "PROCESSED", pieza rechazada "REJECTED").

El tipo de pieza y el estado son tipos de datos definidos por el usuario. En C++ se utiliza el comando *typedef* para definir una estructura de datos.

La clase TxtWorkPiece tiene tres atributos:

- El primero es el tipo de pieza que almacena su color (Véase Código 1), el cual se almacena en la estructura TxtWPType_t.

```
typedef enum
{
    WP_TYPE_NONE,
    WP_TYPE_WHITE,
    WP_TYPE_RED,
    WP_TYPE_BLUE
} TxtWPType_t;
```

Código 1. Estructura tipo TxtWPType_t.

- El segundo atributo es el identificador de la pieza, que se almacena en una variable de tipo *string*.
- El último atributo es el estado de la pieza según la fase en la que se encuentre del proceso (Ver Código 2) y se almacena en una estructura de tipo TxtWPState_t.

```
typedef enum
{
    //WP_STATE_NONE,
    WP_STATE_RAW,
    WP_STATE_PROCESSED,
    WP_STATE_REJECTED
```

Código 2. Estructura tipo TxtWPState_t.

Normalmente, en C++ se utilizan un archivo para definir las clases y un archivo distinto para la implementación de los métodos de cada clase. La declaración de la clase se encuentra en los archivos de cabecera (fichero de extensión ".h") y su implementación en los archivos de extensión ".cpp".

En el Código 3 se puede comprobar la definición de la clase TxtWorkpiece. Esta clase se encuentra definida en el archivo "*TxtFactoryTypes*.h".

```
class TxtWorkpiece {
public:
    TxtWorkpiece() //constructor por defecto
        : tag_uid(""), type(WP_TYPE_NONE), state(WP_STATE_RAW) {}
    TxtWorkpiece(const TxtWorkpiece& wp) //constructor de copia
        : tag_uid(wp.tag_uid), type(wp.type), state(wp.state) {};
    TxtWorkpiece(std::string tag_uid, TxtWPType_t type, TxtWPState_t state)
        : tag_uid(tag_uid), type(type), state(state) {} //constructor con parámetros
    virtual ~TxtWorkpiece() {} //destructor

    void printDebug() {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "WP uid:{} type:{} "
                "state:{}",tag_uid,toString(type), toString(state));
    }

    std::string tag_uid; //atributo id de etiqueta pieza
    TxtWPType_t type; //atributo tipo de pieza
    TxtWPState_t state; //atributo estado de la pieza
};
```

Código 3. Clase TxtWorkPiece.

La clase TxtWorkPiece es una clase con tres **constructores**, pues es un método con el mismo nombre que la clase. La clase tiene un constructor por defecto, un constructor de copia y un constructor con parámetros. La clase tiene, además, un **destructor** y tres **atributos**. Tanto los métodos de la clase como los atributos están definidos públicamente, pues como se puede observar, están precedidos de la etiqueta **public**.

Un constructor de una clase es una de las funciones miembro cuya tarea es la de inicializar los objetos de su clase. El nombre de esta función es exactamente el mismo que el de la clase y su invocación es necesaria para inicializar atributos de una clase.

Un constructor normalmente tiene una serie de parámetros que permiten inicializar los atributos de una determinada clase, aunque puede darse que no reciba ningún parámetro. Cuando no existe un constructor, C++ crea automáticamente un **constructor por defecto** [17] pero no inicializan los valores los miembros de la clase a un valor previsible. Por ello, siempre es conveniente crear un propio constructor por defecto.

Normalmente, una clase tiene varios constructores y cada uno se invoca en función de con qué tipo o número de parámetros se invoque. Son los **constructores sobrecargados** y su uso es muy frecuente en la POO.

Un constructor puede aceptar como parámetro una referencia a un objeto de su propia clase. Estos constructores son **constructores de copia** y sirven como su nombre indica para crear copias de objetos.

7 Eclipse CDT

7.1 Qué es un IDE

IDE es el acrónimo del término inglés Integrated Development Environment o, lo que es lo mismo, Entorno de Desarrollo Integrado [23]. Un IDE permite al programador aumentar su productividad ya que reúne las distintas actividades necesarias para la creación de software en una sola aplicación, haciendo que la tarea del programador sea más sencilla a través de la incorporación de bibliotecas, compiladores y depuradores.

Las características básicas que debe tener un IDE son las siguientes [23]:

- Editor de código. Editor de texto creado exclusivamente para trabajar con el código fuente de programas informáticos.
- Compilador. Su función es la de traducir las instrucciones en código fuente, escritas en lenguaje de programación, a lenguaje máquina.
- Depurador o *debugger*. Se utiliza para detectar y buscar errores en un programa.
- *Linker*. Se utiliza para combinar los distintos archivos de código fuente y convertirlos en un único fichero ejecutable.
- Refactorización de código. Proceso en el que se recurre a funciones como el reformateo o la encapsulación para mejorar el código fuente.

El abanico de IDE existentes en el mercado es muy amplio, por lo que la elección final dependerá básicamente de las exigencias y necesidades de cada programador, que puede utilizar IDE diferentes para trabajos distintos. En este proyecto se utilizará el **IDE Eclipse** puesto que es la que ha sido utilizada por los programadores de Fischertechnik para la programación de la fábrica.

7.2 IDE y plataforma Eclipse

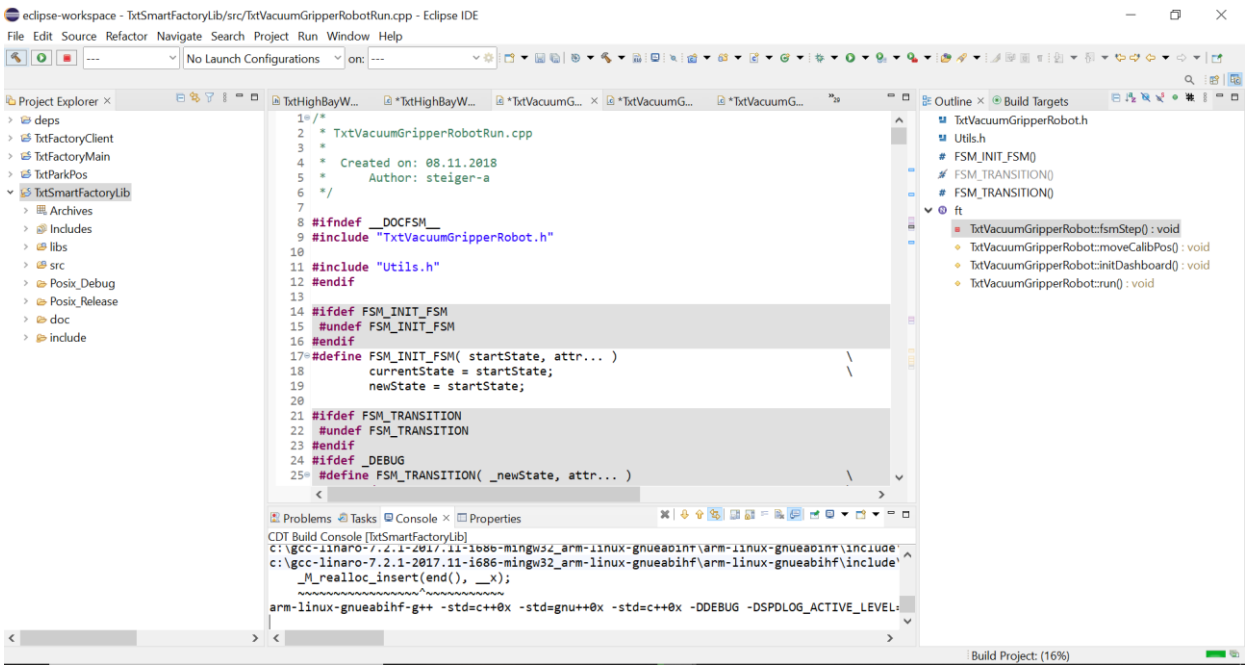
El IDE Eclipse es una de las herramientas que se engloban bajo el denominado Proyecto Eclipse, que aúna tanto el desarrollo del IDE Eclipse como de algunos de los *plugins* más importantes (como el JDT para el lenguaje Java, o el CDT para el lenguaje C/C++).

La base del IDE Eclipse es la plataforma Eclipse, una plataforma de desarrollo integrada de código abierto y multiplataforma, diseñada para ser extensible mediante complementos adicionales [24]. Desarrollada con Java, la plataforma Eclipse se puede utilizar para desarrollar aplicaciones cliente enriquecidas, entornos de desarrollo integrados y otras herramientas que permiten integrar módulos de software independientes en una única aplicación de software. Eclipse se puede utilizar como un IDE para cualquier lenguaje de programación para el que haya un complemento disponible.

El *C/C++ Development Toolkit* (CDT) es un conjunto de *plug-ins* de Eclipse que proporcionan ampliaciones de C y C++ al entorno de trabajo Eclipse. El CDT proporciona un entorno de desarrollo integrado de C/C++ que simplifica muchas de las mismas herramientas que puede utilizar desde la línea de mandatos.

El CDT se abre como la perspectiva C/C++ del entorno de trabajo (*workbench*) Eclipse. Esta perspectiva C/C++ consta de un editor y de las siguientes vistas:

- Explorador de proyectos. Muestra todos los archivos del sistema de archivos bajo su directorio de espacio de trabajo.
- Consola. Visualiza la salida del programa, así como la salida de su construcción y cadena de herramientas externa.
- Editor. La vista del editor de C/C++ proporciona características especializadas para editar archivos relacionados con C/C++.
- Destinos *make*. Permite seleccionar los destinos *make* que desea construir en su espacio de trabajo.
- Esquema. Visualiza la estructura del archivo abierto actualmente en un editor.
- Vista Problemas. Si encuentra errores durante una construcción, los errores se mostrarán en la vista Problemas.
- Propiedades. Muestra los atributos del elemento seleccionado actualmente en una vista o un editor.
- Buscar. Muestra los resultados de búsquedas de archivos o texto.
- Tareas. Lista tareas de las que desea realizar un seguimiento, ya sea como una planificación de cosas a hacer o un historial de cosas ya hechas.



7.3 Primeros Pasos con CDT

Como se ha comentado, el kit de herramientas de desarrollo C/C++ es una colección de funciones basadas en Eclipse que proporciona la capacidad de crear, editar, navegar, construir y depurar proyectos que utilizan C y/o C++ como lenguaje de programación.

La CDT no incluye los compiladores y depuradores necesarios para convertir el código C/C++ en programas ejecutables y para depurar dichos programas, pero sí proporciona los marcos que permiten integrar dichas herramientas de forma coherente. Esto le permite mezclar y combinar dichas herramientas en función de los requisitos del proyecto.

A menudo, las distribuciones comerciales de CDT incluyen las herramientas e integraciones necesarias. Sin embargo, el CDT de la web de Eclipse requiere de un *toolchain* antes de poder construir y depurar cualquier proyecto. El CDT estándar soporta la integración con la cadena de herramientas de GNU, que incluye las utilidades de GNU *make*, el compilador *gcc* y el depurador *gdb*.

Para Windows, las dos principales opciones de plataforma para adquirir la cadena de herramientas de GNU son MinGW y Cygwin. Es importante entender la diferencia entre ellas: Cygwin produce ejecutables que utilizan el tiempo de ejecución POSIX de Cygwin (licencia GPL), mientras que MinGW produce ejecutables nativos de Windows que no requieren un tiempo de ejecución separado.

Para MinGW, se recomienda utilizar el instalador de MinGW, *mingw-get*, para gestionar la instalación de MinGW. Descargue y ejecute el último paquete mingw-get-inst desde la página de descargas de MinGW. La cadena de herramientas de CDT MinGW encontrará esta instalación si se encuentra en el directorio predeterminado C:\MinGW, el directorio bin de MinGW se coloca en su ruta, o si la ubicación de MinGW se almacena en la variable de entorno MINGW_HOME.

Por su parte, Cygwin puede ser instalado desde el sitio de Cygwin en https://www.cygwin.com. Es necesario seleccionar manualmente los paquetes *gcc*, *gdb* y *make* para instalar la cadena de herramientas.

8 La Nube de Fischertechnik

El panel de control de la fábrica de aprendizaje puede gestionarse a través de dispositivos móviles, como tabletas o teléfonos inteligentes, y ordenadores, portátiles o de sobremesa. Este panel de control representa la fábrica desde tres perspectivas diferentes: cliente (amarillo), proveedor (naranja) y producción (verde).

En la **perspectiva-vista del cliente** se muestra una tienda web con un carrito de compras, desde el que se puede encargar una pieza y seguir su estado. Este proceso se muestra en la interfaz del cliente, de forma que esté siempre informado sobre el estado actual de su pedido.

En la **perspectiva-vista del proveedor** se muestra la materia prima existente, a fin de conocer cuándo solicitar más.

Por su parte, en la **perspectiva de producción** se pueden consultar el estado de la fábrica, del proceso productivo, del inventario, del lector NFC, así como los valores del sensor. Además, se puede manejar la cámara de la línea de producción.

- En la vista *estado de fábrica* se muestra el estado de cada módulo mediante una visualización tipo semáforo. Si surge una avería en la producción, una vez eliminada la causa, esta se confirma mediante un botón y la producción continúa.
- En la vista *proceso productivo* se representan cada uno de los pasos de fabricación mediante puntos clave conectados de manera simplificada. El nodo activo (=

módulo de producción) se ilumina en color verde si la fase respectiva del proceso se encuentra en funcionamiento o en rojo si hay un error que debe ser solucionado.

- La vista *inventario* muestra el stock actual de piezas. Esta vista no puede ser editada. La vista *lector NFC/RFID* muestra los datos de la pieza activa en el proceso, aunque también puede usarse para leer manualmente los datos de otras piezas o, incluso, eliminarlos. Cada pieza de trabajo tiene una identificación propia y distintiva que incluye los siguientes datos: estado, color y registro de la fecha y hora desde la producción hasta la entrega.
- La vista de producción también permite controlar la cámara y visualizar los valores recogidos por el sensor medioambiental.

El panel de control de la fábrica de aprendizaje industria 4.0 permite gestionar fácilmente estas vistas y otras funciones de la fábrica. Si se accede al panel de control en la nube de la fábrica se observará una pantalla similar a la que aparece en la Figura 11. Se incluye una numeración sobre la figura para hacer referencia a las ventanas de la captura de pantalla de la nube de Fischertechnik.

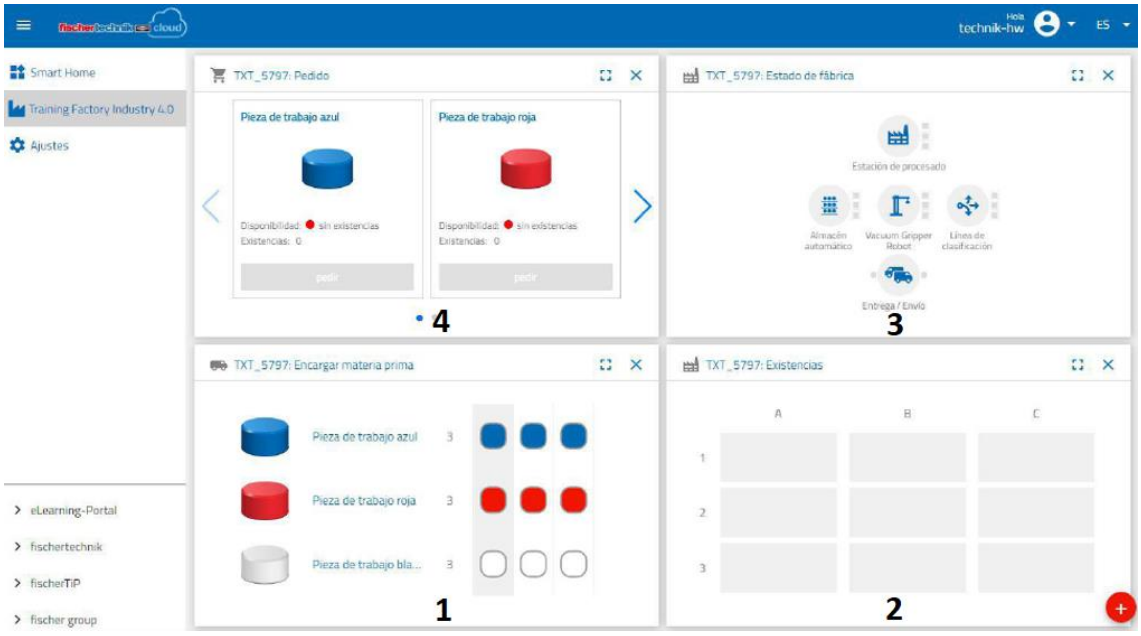


Figura 11. Panel de Control de la Fábrica de Aprendizaje.

La ventana 1 muestra la materia prima que falta y debe ser encargada.

La ventana 2 representa la materia prima almacenada en cada celda de la estantería elevada. Si se retira o se incluye materia prima, se modifica de acuerdo con la cantidad y el color de la pieza retirada o añadida.

La ventana 3 muestra el estado actual de trabajo de toda la fábrica. Esta ventana permite seguir el proceso de producción tras haber solicitado una pieza. De esta forma, el cliente puede visualizar dónde se encuentra la pieza en la fábrica en cada momento. Por ejemplo, el icono que representa al manipulador de succión se resalta en color azul cuando está funcionando.

En la ventana 4 se indica cuánta materia prima (azul, roja, blanca) hay disponible. Si, por ejemplo, llega una pieza roja a la estantería, se modifica su disponibilidad y se incremente el valor de inventario.

En la ventana 5 (Figura 12) se representa el estado del pedido y en la ventana 6 (Figura 13) se muestra el estado actual del lector NFC y los datos de identificación de la pieza.

Figura 12. Ventana de la nube para el seguimiento del pedido.

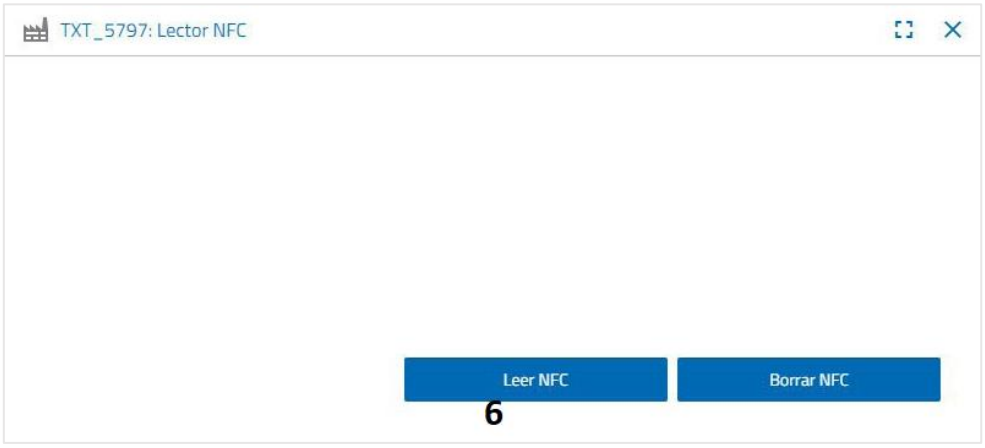


Figura 13. Ventana de la nube que indica el estado actual del NFC.

CAPÍTULO 2 Análisis del código

A lo largo del siguiente capítulo se va a realizar un análisis sobre la programación de algunas de las herramientas clave para llevar a cabo la comunicación entre los dispositivos de la fábrica: las máquinas de estado, el protocolo MQTT y el control y configuración de sensores y actuadores.

En primer lugar, se hablará del esquema de funcionamiento utilizado para la programación de las Máquinas de Estados Finitos en la fábrica. Seguidamente, se explicará el esquema de funcionamiento para la comunicación MQTT en la fábrica. Por último, se explicará la programación para la configuración y control de los sensores y actuadores de la fábrica.

1 Programación de las Máquinas de Estado

La programación de la fábrica consta de cuatro archivos que contienen la programación de las *Finite State Machine* (FSM) o máquina de estados finitos de las distintas estaciones. Los archivos nombrados bajo la sintaxis de "TxtNombreDeLaEstaciónRun.cpp" contienen la programación para las máquinas de estados de cada estación (Figura 14). Se puede comprobar que la estación DPS no tiene una FSM, porque esta estación no funciona independientemente, sino que requiere de los movimientos del VGR; en otras palabras, es el VGR el que controla los estados por los que debe pasar la pieza en la DPS.

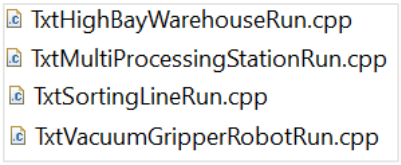


Figura 14. Archivos "Run" para las FSM.

La fábrica puede estar en cuatro estados genéricos que están representados en los diagramas de estado con cuatro colores (rojo, verde, azul y amarillo). Para hacer referencia al nombre de cada estado a lo largo del documento, en los diagramas de estado y en el código se utilizarán letras mayúsculas. En la Figura 15 se puede ver un diagrama elaborado con el objetivo de ayudar a comprender la programación de las FSM de las estaciones. Los cuatro estados genéricos posibles son los siguientes:

- Estado FAULT o estado de error (color rojo).
- Estado IDLE o estado de espera (color verde).
- Estado INIT o estado de trabajo (color azul). Dentro de este estado hay más subestados que son los que controlan los estados por los que pasa cada estación (HBW, VGR, MPS, DPS, SL)
- Estado CALIB o estado de calibración (color naranja). Dentro de este estado hay otros subestados que se encargan de calibrar las distintas estaciones.

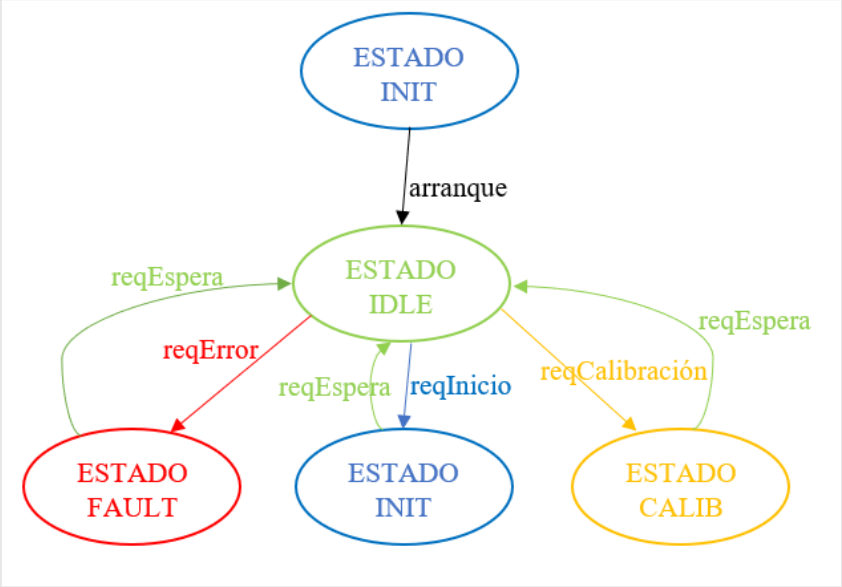


Figura 15. Diagrama general de los estados de la fábrica.

El paso de un estado a otro está marcado por un evento. Cada evento está asociado a una variable *bool* que indica que el evento se ha producido. Las variables que marcan los eventos son del tipo “reqSomething”. Por ejemplo, la variable *reqVGRstore* se pondrá a *true* cuando se necesite llamar al VGR para que vaya a por una pieza para ser almacenada.

Para programar la FSM se utiliza un bloque *switch-case*. La transición entre los estados de la fábrica empieza en el estado INIT, una vez que arranca el programa en cada controlador. De ahí pasa a un estado IDLE de reposo, a la espera de que se produzca algún evento que genere un cambio que marque el paso a un estado de trabajo INIT o a un estado de calibración CALIB (en al caso en el que el usuario desee hacer una calibración).

Una vez que se produce un evento, de los que marca el paso a un estado de trabajo INIT, se empezará a producir la transición entre los subestados INIT de las estaciones. Desde algunos de los estados (se pueden ver en cada diagrama de estados propio de cada estación) se podrá producir un error FAULT (flechas rojas) o se producirá la transición a un estado de reposo IDLE (flechas verdes), en el caso en el que la estación necesite pasar a un estado de reposo a la espera de que se complete alguna tarea en otra estación.

La instrucción necesaria para pasar de un estado a otro es la siguiente:

FSM_TRANSITION (nuevoEstado, color = "idle:green, fault:red, calib:orange, init: blue", label ="etiqueta que marca el evento").

A continuación, se incluye un ejemplo ilustrativo para poder explicar el esquema que se sigue en la programación de las FSM y poder entender cómo se produce la transición entre estados en el código de la fábrica. El diagrama que representa la FSM de ejemplo es el siguiente (Figura 16):

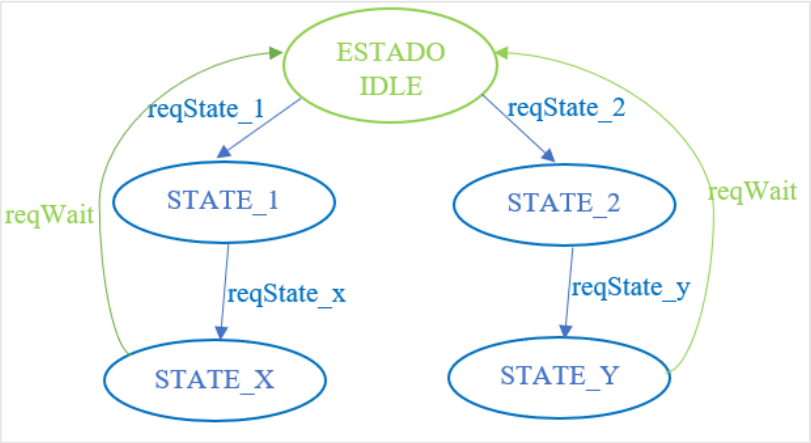


Figura 16. Ejemplo de FSM de invención propia.

En la Figura 17 se muestra el código para llevar a cabo la FSM del diagrama anterior. Se parte de un estado IDLE en el que la máquina se encuentra esperando a que alguna de las dos variables *reqSate*_1 o *reqState*_2 se pongan a *true*. Una vez que una de ellas se active se utiliza la instrucción *FSM_TRANSITION* para pasar del estado de reposo a estado STATE_1 o bien a estado STATE_2.

Tras alcanzar alguno de dichos estados se ejecutarán los distintos métodos que pondrán a *true* la variable que marca la siguiente transición para pasar, respectivamente, al estado STATE_X o al estado STATE_Y. Dentro de estos dos estados, una vez que se ejecuten los métodos correspondientes, volverán al estado IDLE de reposo cuando se active la variable *reqWait*

```
Case IDLE:
            if (reqState_1):
                  FSM_TRANSITION(STATE_1,color="blue",label="reqState_1")
            else if (requtate_2):
                  FSM_TRANSITION (STATE_2,color="blue",label="reqState_2")
-------------------------------------------------------------------------------------------------
Case STATE_1:

      … Mover a pos1 … movePos1()
            if (reqState_x):
                  FSM_TRANSITION (STATE_X, color= "blue",label="reqState_x")
            break;
CASE STATE_X:
      … Hacer X … doX()
            if(reqWait):
                  FSM_TRANSITION (IDLE,color="green",label="wait")
            break;
-------------------------------------------------------------------------------------------------

Case STATE_2:
      … Mover a pos2 … movePos2()
            if (reqState_y):
                  FSM_TRANSITION (STATE_Y,color="blue",label="reqState_y")
            break;
Case STATE_Y:
      … Hacer Y … doY()
            if(reqWait):
                  FSM_TRANSITION (IDLE,color="green",label="wait")
            break;
```

Figura 17. Código FSM.

El funcionamiento de cada FSM no es independiente debido a que, en algunos casos, la transición de un estado a otro en una FSM depende de la finalización de un estado en otra FSM. Es decir, requiere de una comunicación entre estaciones para controlar la finalización de un determinado estado en una de las otras estaciones.

Por ejemplo, es necesaria la comunicación entre el almacén y el VGR. El transelevador del almacén tiene que saber cuándo el VGR ha depositado una pieza sobre el contenedor de la bandeja exterior del almacén para poder iniciar el almacenamiento de la pieza. Para ello es necesario que el VGR mande un mensaje al almacén en el que indique que la pieza ha sido depositada sobre el contenedor. Esto es posible mediante la utilización de un protocolo de comunicación que haga posible el envío de mensajes entre las estaciones.

El protocolo en cuestión es el MQTT y es una herramienta clave dentro del IoT.

2 Programación del protocolo MQTT

El protocolo MQTT es un protocolo de mensajería con un patrón de diseño publicador/suscriptor (Pub/Sub) [11]. Este patrón presenta un diseño de programación orientado a eventos en el cual hay unas clases que se suscriben (suscriptor) y/o publican (publicador) a un tema (*topic*).

El protocolo MQTT se caracteriza por tener un diseño ligero, simple y de código abierto. Sus características lo hacen idóneo para comunicaciones de tipo M2M y en las conexiones IoT.

Una de las características principales del MQTT es el desacoplamiento publicador-suscriptor [12]. Cada cliente puede ser publicador, suscritor o ambos al mismo tiempo [3]. Mediante este desacoplamiento los publicadores no envían los mensajes directamente a un suscriptor concreto, sino que se clasifican en temas y cada suscriptor se suscribe a los temas que le interesan. Este desacoplamiento se produce mediante un bróker intermedio, que es el encargado de filtrar los mensajes y enviarlos a los suscriptores que están suscritos al tema.

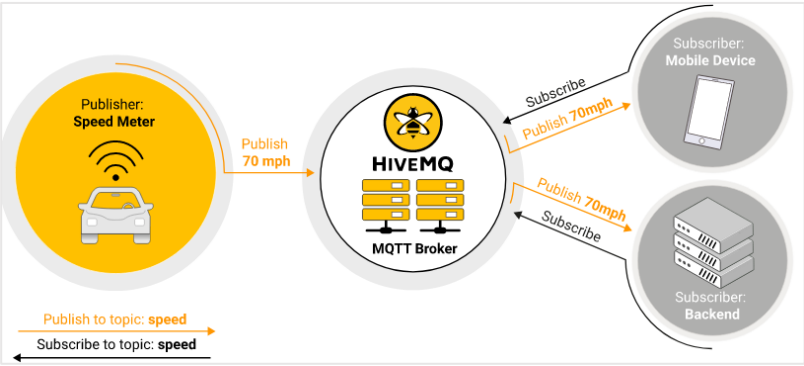


Figura 18. Patrón Publicador Suscriptor [25].

Un cliente MQTT es cualquier dispositivo (desde un pequeño controlador como los de la fábrica hasta un servidor) que implementa una librería MQTT y se conecta con un bróker MQTT a través de una red de internet mediante protocolo TCP/IP [14]. Para publicar o recibir mensajes los clientes necesitan conocer el nombre del host (IP) y el puerto del bróker.

En el protocolo MQTT la comunicación no se realiza directamente entre los clientes, sino que se realiza mediante una conexión intermedia con un bróker. El bróker MQTT es el responsable de recibir los mensajes, filtrarlos (determinando quien está suscrito a cada tipo de mensajes) y enviar los mensajes a aquellos clientes que están suscritos.

Se puede ver en la *Figura 19* el esquema de conexión de los dispositivos [8]. El controlador TXT central (TXT 0, estación SSC) actúa como MQTT Broker y, a su vez, como MQTT cliente ya que transmite los datos de la fábrica de aprendizaje al MQTT bróker remoto (Nube de Fischertechnik). Los clientes MQTT de la fábrica son los controladores TXT 1 (MPO), el TXT 2 (HBW), el TXT 3 (VGR), el TXT 4 (SLD).

Los datos entre ambos MQTT Broker (local y remoto) son encriptados mediante un MQTT bridge (puente) desde el puerto 8883. La comunicación local entre los clientes MQTT se realiza a través del puerto 1883 (flechas naranjas).

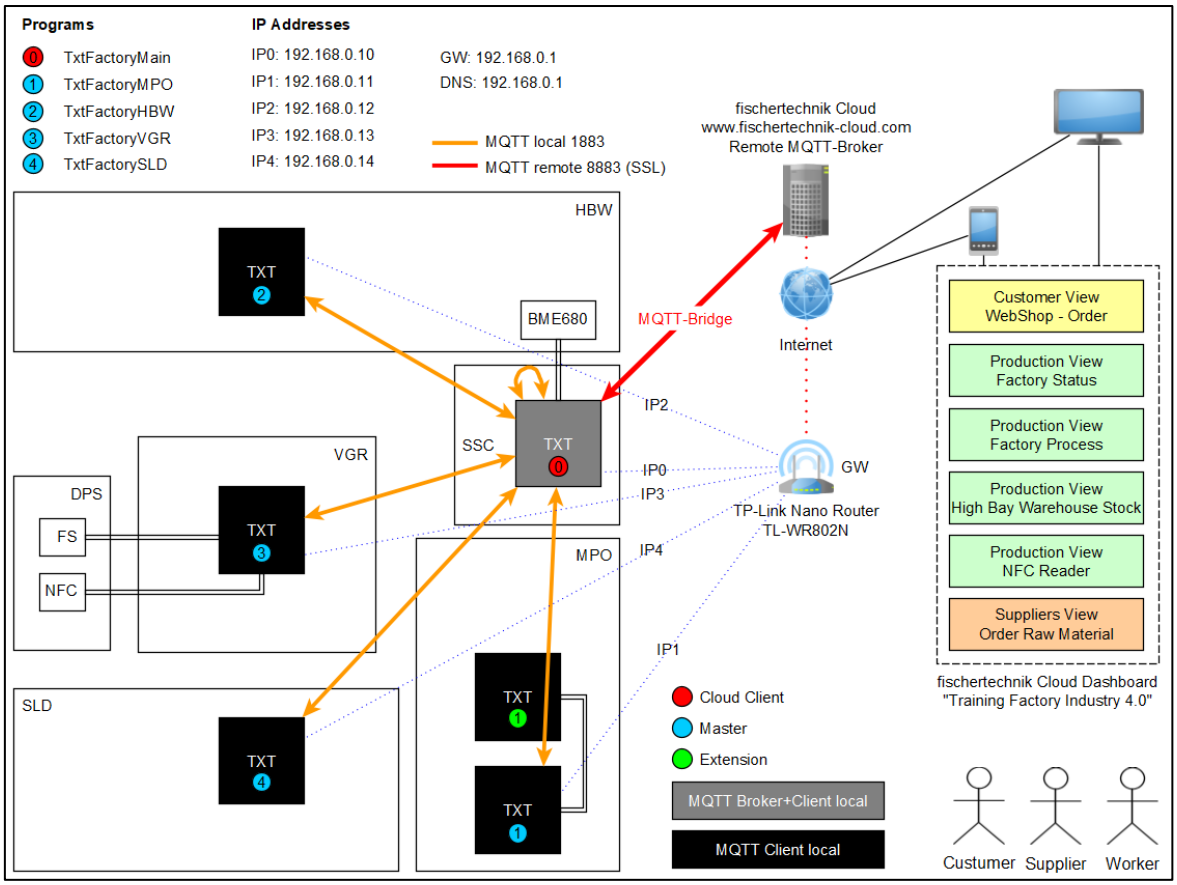


Figura 19. Esquema de conexión MQTT [8].

Un cliente puede publicar mensajes en cuanto se conecte al bróker. Cada mensaje debe contener un *topic* que el bróker utiliza para reenviarlo a los clientes que están suscritos. En MQTT se utiliza la palabra *topic* para referirse a un *string* de tipo UTF-8 [2], en el que se tienen niveles de *topic* separados por barras invertidas en los que con el uso de *subtopic* se añade un mayor nivel de especificación, de igual forma a como se accede a los directorios en el ordenador o en los buscadores web (Figura 20).

La descripción de los *topics* la encontramos en el archivo "TxtMqttFactoryClient.h" junto con la descripción de todos los métodos utilizados para poder establecer los distintos tipos de comunicación en función del tipo de componente publicador o suscriptor. La implementación de estos métodos se encuentra en el archivo "TxtMqttFactoryClient.cpp".

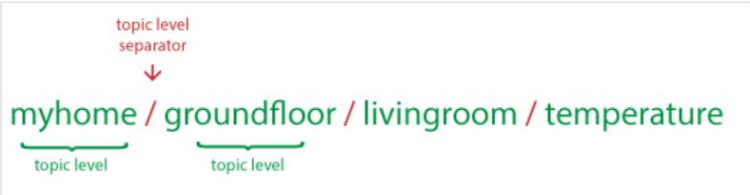


Figura 20. Topic.

[2] UTF-8 son las siglas de «8-bit Unicode Transformation Format», un formato de codificación de caracteres Unicode e ISO 10646 que utiliza símbolos de longitud variable [29].

En la se Tabla 2 se muestran las abreviaturas utilizadas para cada *topic.*

Tabla 2. Abreviaturas topics MQTT.

	Factory Remote
	Factory local
	Configuración
	Entrada (Input)
	Salida (Output)
	Estado
hbw	Bahía de Almacenamiento (High Bay Warehouse)
vgr	Robot de Aspiración (Vacuum Gripper Robot)
mpo	Multi Processing Station with Oven
sld	Estación de Clasificación (Sorting Line Detection)
ssc	Estación Ambiental con cámara de Vigilancia (Sensor Station With Camera)
nfc	Dispositivo NFC

	Sensor DPS entrada piezas
	Sensor DPS salida piezas
	Cámara
	Sensor Estación Ambiental BME680
	Pan Tilt Unit (Sistema de Posicionamiento de Cámara)
	Posición relativa para el sistema de posicionamiento de cámara
	Sensor de Brillo
	Mensaje de Alerta
	Mensaje do
	Acknowledgement (Acuse de recibo de mensaje)
	Stock HBW
	Características de fábrica (hw id, hw model, sw name …)

f dsi
f dso
c cam
i Bme680
o ptu
state pos
ldr
alert
do
ack
stock
broadcast

3 Programación para el control de sensores y actuadores.

Para hacer posible la comunicación entre el cliente y los sensores/actuadores de la fábrica 4.0 se utiliza un área de memoria compartida denominada FischerTechnik Transfer Area (TA). La documentación de la TA se puede encontrar en los archivos de cabecera "*FtShmem*.h" y "*KeLibTxtDI*.h". Parte de la información para explicar la programación ha sido obtenida de GitHub, si se desea conocer mejor el funcionamiento de la Transfer Area puede visitarse [26].

Para establecer la comunicación inicial con el TA en primer lugar se crea un puntero *pTArea* al que se le asigna mediante el método ***`GetKelibTransferArea()`*** la dirección que apunta a la estructura TA. Esto se realiza en el *main* de los programas principales (Cliente, Main y ParkPos)

```
if (StartTxtDownloadProg() == KELIB_ERROR_NONE)
{
    pTArea = GetKeLibTransferAreaMainAddress();

    if (pTArea)
    {
            ft::TxtTransfer T(pTArea);
```

Código 4. Comunicación inicial con Transfer Area.

La Clase TxtTransfer está declarada en la clase TxtAxis (Código 5).

```
class TxtTransfer {
public:
    FISH_X1_TRANSFER* pTArea;
    TxtTransfer(FISH_X1_TRANSFER* pTArea) : pTArea(pTArea) {}
    virtual ~TxtTransfer() {}
};
```

Código 5. Clase TxtTransfer.

Dentro de esta clase se define el puntero (*pTArea*) de tipo FISH_X1_TRANSFER. Esta estructura (Ver Código 6) permite configurar los dispositivos de tipo INPUT (sensores) y de tipo OUTPUT (actuadores), controlar los motores y algunos otros sensores especiales como los de la estación ambiental.

```
typedef struct shm_if_s
{
    FTX1_SHMIFINFO      ftX1info;       // info structure
    FTX1_STATE          ftX1state;      // state structure
    FTX1_CONFIG         ftX1config;     // config structure
    FTX1_INPUT          ftX1in;         // input structure
    FTX1_OUTPUT         ftX1out;        // output structure
    FTX1_DISPLAY        ftX1display;    // display structure

    TRANSFER_STATUS     IFStatus;
    CHANGE_STATE        IFChange;       // change state of Input, Counter, Timer
    RP_TIMER            IFTimer;        // 16-Bit timer variables
    MOTOR               IFMotor;        // motors control
    INPUT_SIM           IFInputSim;     // input simulation
    HOOK_TABLE          IFHookTable;    // hook table with functions pointers

    TXT_SPECIAL_INPUTS  sTxtInputs;     // TXT Special Inputs (Power, Temp, IR...)
    TXT_SPECIAL_OUTPUTS sTxtOutputs;    // TXT Special Outputs (sound, LED)
    TXT_SPECIAL_INPUTS_2 sTxtInputs2;     // TXT Special Inputs (Power, Temp, IR...)

    char                reserved[RESERVE_SIZE];
} FISH_X1_TRANSFER;
```

Código 6. Estructura FISH_X1_TRANSFER.

En caso de que haya dos TXT para el control de una estación, como es el caso de la MPS, en el que un TXT actúa de *master* y el otro de *slave* deberá usarse un área de memoria compartida con dos estructuras (una por cada TXT). El puntero del *master* apuntará a *pTArea* y el otro a *pTArea + 1*. Este caso se puede ver en la configuración de los dispositivos en la MPS (Véase Configuración de los dispositivos).

Se necesita una función *callback* ***`TransferAreaCallbackFunction()`*** mediante la que se realiza el *callback* al método ***`SetTransferAreaCompleteCallback()`***, entre la recepción de la información a través los *inputs* y el envío de información a los *outputs*. La llamada a esta función se realiza en los constructores de las clases de la DPS y la SLD.

```
TxtDeliveryPickupStation::TxtDeliveryPickupStation(TxtTransfer* pT, ft::TxtMqttFactoryClient* mqttclient)
    : TxtSimulationModel(pT, mqttclient),
    lastColorValue(0), activeDSI(false), activeDSO(false), errorDSI(false), errorDSO(false)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "TxtDeliveryPickupStation",0);
    if (!calibData.existCalibFilename()) calibData.saveDefault();
    calibData.load();
    //sound.enable(calibData.sound_enable); //see VGR
    configInputs();
    SetTransferAreaCompleteCallback(DPSTransferAreaCallbackFunction);
}
```

Código 7. Llamada al método setTransferAreaCompleteCallback() en el constructor de la DPS.

3.1 Configuración de los dispositivos.

Antes de comenzar con la lógica de los dispositivos es necesaria su configuración. La configuración se realiza mediante la estructura *ftX1config* que puede ser de tipo *uni* (input) o *cnt* (contador)

```
typedef struct ftX1config
{
    UINT8           pgm_state_req;      // enum PgmState    pgm_state_req;
    BOOL8           old_FtTransfer;
    char            dummy[2];
    BOOL8           motor[IZ_MOTOR];
    UNI_CONFIG      uni[IZ_UNI_INPUT];
    CNT_CONFIG      cnt[IZ_COUNTER];
    INT16           motor_config[IZ_MOTOR][4];
} FTX1_CONFIG;
```

Código 8. Estructura ftx1config.

A continuación, se describe la configuración de los sensores de la MPO realizada en el método *configInputs*. La configuración del fototransistor de detección de piezas del final de la cinta transportadora se configura en MODE_R (esto es para tomar valores de resistencias en Ohms) y en modo digital (*true* es digital "=1" y *false* es analógico "=0"). Para actualizar la configuración de los sensores se emplea la estructura *ftx1state.config_id*.

Dependiendo de qué entrada del TXT se emplee para conectar cada dispositivo, al tratarse de un vector en el que cada posición del vector se corresponde con el orden en el que los dispositivos están conectados en el controlador (se debe comprobar el plan de asignación de cada estación para saber a qué entrada del TXT está conectado), se deben utilizar los corchetes. Como cualquier vector el primer elemento es el 0, y como la numeración de entradas empieza en el 1, hay que sumarle uno al número entre corchetes para saber de qué entrada del TXT se trata.

En este caso si comprobamos el plan de asignación de la MPO (TXT 1) se comprueba que el fototransistor de detección de fin de cinta transportadora está en la entrada 4 del TXT *master*.

```
pT->pTArea->ftX1config.uni[3].mode = MODE_R; // Digital Switch with PullUp resistor
pT->pTArea->ftX1config.uni[3].digital = 1;
//save
pT->pTArea->ftX1state.config_id ++; // Save the new Setup
```

Código 9. Configuración fototransistor final de cinta transportadora.

Seguidamente, se configura el fototransistor que detecta la presencia de una pieza en la bandeja del horno. Se configura de la misma forma que el anterior, salvo que ahora, al estar conectado a una entrada del TXT esclavo (entrada 5 del TXT *esclavo*), se debe tener en cuenta que hay que acceder a la segunda estructura de tipo *pTArea* (*ptArea*+1)

```
(pT->pTArea+1)->ftX1config.uni[4].mode = MODE_R; // Digital Switch with PullUp resistor
(pT->pTArea+1)->ftX1config.uni[4].digital = 1;
//save
(pT->pTArea+1)->ftX1state.config_id ++; // Save the new Setup
```

Código 10. Configuración fototransistor bandeja de piezas del horno.

3.2 Estructura tipo OUTPUT

Mediante este tipo de estructura (Código 11) se van a poder controlar los actuadores: las válvulas que activan los cilindros neumáticos, motores, compresores y las luces LED de la fábrica.

```
typedef struct ftX1output
{
    // Counter reset requests (increment each time by one)
    UINT16          cnt_reset_cmd_id[IZ_COUNTER];
    // If not 0, synchronize this channel with the given channel (1:channel 0, ..)
    UINT8           master[IZ_MOTOR];
    // User program selected motor PWM values
    INT16           duty[IZ_PWM_CHAN];
    // Selected distane (counter value) at which motor shall stop
    UINT16          distance[IZ_MOTOR];
    // Increment by one each time motor_ex settings change
    UINT16          motor_ex_cmd_id[IZ_MOTOR];
} FTX1_OUTPUT;
```

Código 11. Estructura tipo OUTPUT.

El control de las válvulas en el programa de la fábrica se realiza mediante la misma estructura que se utiliza para arrancar un motor (*ft1xout.duty)* o una lámpara. La válvula electromagnética se activa con un interruptor controlado por modulación por ancho de pulso (PWM, *Pulse Width Modulation*).

Todos los puertos del controlador están regulados mediante PWM. Esta es la forma de poder regular la cantidad de energía que llega a los actuadores de forma precisa. Lo que hace la modulación PWM es activar y desactivar los interruptores de forma muy rápida. Si el pulso es corto, el *duty cycle* (ciclo de trabajo) es bajo y el motor gira lentamente. Cuando el ancho de pulso es largo y el tiempo de desconexión corto llega más energía a la carga y el motor gira más rápidamente [27].

Mediante la programación de los sensores y actuadores a través de las librerías de la TA podemos regular la cantidad de energía que llega a las cargas. Los controladores permiten ajustar el número de pasos. En la programación se utiliza una configuración de 512 pasos porque permite un control mucho más fino. Se puede ver en la el *duty cycle* en función del número de pasos. Para 511 pasos se tiene un *duty cycle* del 99%, mientras que para 448 pasos el *duty cycle* baja hasta un 61%. Es por esta razón por la que se ha seleccionado un *duty cycle* de 512 pasos.

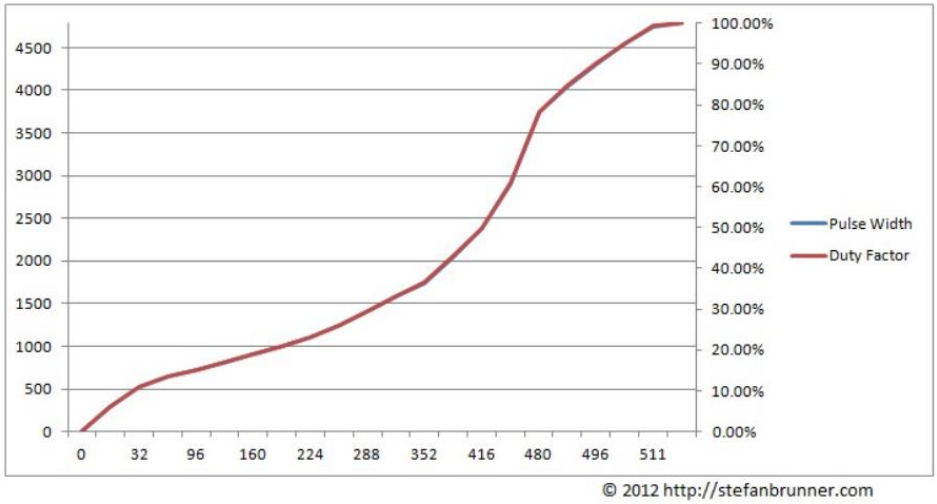


Figura 21. Gráfica relación entre Duty Cycle y número de pasos.

En el siguiente ejemplo se muestra el control de la válvula de aspiración del transportador de piezas de la MPO. El método ***setValveVacuum()*** recibe una variable *bool* y en el caso de que sea true asigna el valor 512 (interruptor *on*) a la estructura tipo output. En el caso de que sea *false* se asigna el valor 0 (interruptor *of*). De esta forma abriríamos o cerraríamos la válvula.

```
void TxtMultiProcessingStation::setValveVacuum(bool on) //Activar valvula de vacio
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "setValveVacuum {}", on);
    assert(pT->pTArea+1);
    (pT->pTArea+1)->ftX1out.duty[4] = on ? 512 : 0; // Switch on with PWM Value 512 (= max speed)
}                                                   //O5 Extention
```

Código 12. Control de la Válvula de Aspiración de la MPO.

En la fábrica para el control de los ejes de los robots se utilizan motores con *encoder*. El motor tiene dos partes: la caja de engranajes y el *encoder*.

Debido al *encoder* el motor debe ser conectado a un puerto de entrada de tipo contador (C) para controlar el número de vueltas y una entrada de tipo motor (M) que lo conforman dos entradas de output (O). Los motores presentan una relación de 3:1 (3 impulsos por cada revolución del eje). Igual que en el ejemplo de la válvula para girar un motor con una velocidad determinada se utiliza la petición *duty* y en función del valor que se le asigne girará más rápido o más despacio (están configurados para que la máxima velocidad sea para 512 pasos).

La petición *duty* se emplea para hacer girar el motor en uno u otro sentido dependiendo del sentido de la corriente. Un motor tiene conectado el polo positivo a una de las clavijas y el polo negativo a la otra, por lo que dependiendo de la intensidad de la corriente girará a la derecha o a la izquierda. Para girar el motor a la izquierda se activa uno de los canales a los que está conectado el motor y se desactiva el otro, y para hacerlo al revés, al contrario. Dentro de la clase TxtAxis se controla el movimiento de los ejes.

Para girar el motor a la izquierda se utiliza el método ***setMotorLeft()*** (Código 13). Se comprueba si *chM* es menor que ocho para detectar si el canal del motor está conectado a un TXT máster o esclavo.

```
if (chM < 8)
{
    assert(pT->pTArea);
    pT->pTArea->ftX1out.duty[chM*2] = speed;
    pT->pTArea->ftX1out.duty[chM*2+1] = 0;
}
else
{
    assert(pT->pTArea+1);
    (pT->pTArea+1)->ftX1out.duty[(chM-8)*2] = speed;
    (pT->pTArea+1)->ftX1out.duty[(chM-8)*2+1] = 0;
}
```

Código 13. Girar motor para control de los ejes a la izquierda.

Para girar el motor a la derecha se utiliza el método ***setMotorRight()*** (Código 14) y se hace al contrario que en el anterior.

```
if (chM < 8)
{
    assert(pT->pTArea);
    pT->pTArea->ftX1out.duty[chM*2] = 0;
    pT->pTArea->ftX1out.duty[chM*2+1] = speed;
}
else
{
    assert(pT->pTArea+1);
    (pT->pTArea+1)->ftX1out.duty[(chM-8)*2] = 0;
    (pT->pTArea+1)->ftX1out.duty[(chM-8)*2+1] = speed;
}
```

Código 14. Girar motor para control de los ejes a la derecha.

Para mover los ejes con *encoder* es necesario utilizar los contadores. La petición *cnt_reset_cmd_id* se utiliza para reiniciar el valor del contador. En la clase TxtAxis1refSwitch el método ***resetCounter()*** (Código 15) resetea todos los contadores de los motores antes de iniciar cada movimiento. Para ello se utiliza el operador de incremento.

```
void TxtAxis1RefSwitch::resetCounter()
{
    SPDLOG_LOGGER_DEBUG(spdlog::get("console_axes"), "{} cnt {}",
            name,chM<8?pT->pTArea->ftX1in.counter[chM]:(pT->pTArea+1)->ftX1in.counter[chM-8]);

    chM<8? pT->pTArea->ftX1out.cnt_reset_cmd_id[chM]++:(pT->pTArea+1)->ftX1out.cnt_reset_cmd_id[chM-8]++;
    //while(chM<8?pT->pTArea->ftX1in.cnt_resetted[chM]:(pT->pTArea+1)->ftX1in.cnt_resetted[chM-8] != 1);
    SPDLOG_LOGGER_DEBUG(spdlog::get("console_axes"), "{} cnt_reset {}",
            name,chM<8?pT->pTArea->ftX1in.counter[chM]:(pT->pTArea+1)->ftX1in.counter[chM-8]);
}
```

Código 15. Resetear contador Clase TxtAxis1RefSwitch.

Para que el motor se mueva un cierto número de pasos se utiliza la petición *distance* (Ver Código 16). El método ***moveRight()*** encargado de controlar el movimiento de los ejes del robot mediante un motor encoder por cada eje mueve un cierto número de pasos que se pasan por parámetro al método en la variable *steps*. Después de realizar el movimiento se utiliza la petición *motor_ex_cmd_id*, que se incrementa en uno cada vez que realizamos alguna petición sobre el motor, para devolver el identificador de la última petición realizada sobre este.

```
if (chM < 8)
{
    pT->pTArea->ftX1out.distance[chM] = steps; // Distance to drive Motor 1 [0]
    pT->pTArea->ftX1out.motor_ex_cmd_id[chM]++; // Set new Distance Value for Motor 1 [0]
}
else
{
    (pT->pTArea+1)->ftX1out.distance[chM-8] = steps; // Distance to drive Motor 1 [0]
    (pT->pTArea+1)->ftX1out.motor_ex_cmd_id[chM-8]++; // Set new Distance Value for Motor 1 [0]
```

Código 16. Mover motor un cierto número de pasos.

3.3 Estructura tipo INPUT

La estructura tipo *input* se utiliza para leer los valores de los sensores y en los motores con *encoder* para controlar los contadores. Para leer los valores de los sensores (pueden ser datos de voltajes, resistencias, cm en los sensores de ultrasonido …) es importante configurarlos antes (Ver Configuración de los dispositivos. Para leer directamente los valores se utiliza la petición *uni*.

```
typedef struct ftX1input
{
    INT16           uni[IZ_UNI_INPUT];
    INT16           cnt_in[IZ_COUNTER];
    INT16           counter[IZ_COUNTER];
    INT16           display_button_left;
    INT16           display_button_right;
    // Set to 1 when last requested counter reset was fulfilled
    BOOL16          cnt_resetted[IZ_COUNTER];
    // Set to 1 by motor control if target position is reached
    BOOL16          motor_ex_reached[IZ_MOTOR];
    // Counter reset command id of the last fulfilled counter reset
    UINT16          cnt_reset_cmd_id[IZ_COUNTER];
    // Motor extended command id of the last fulfilled motor_ex command
    UINT16          motor_ex_cmd_id[IZ_MOTOR];
} FTX1_INPUT;
```

Código 17. Estructura tipo INPUT.

Por ejemplo, para detectar si un fototransistor detecta una pieza en la entrada de la bandeja del horno se utiliza el siguiente método. En este caso, cuando se detecta un valor distinto de uno la variable *ret* se pone a true.

```
bool TxtMultiProcessingStation::isOvenTriggered()
//Devuelve true si el fototransistor detecta pieza en entrada del horno
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "isOvenTriggered", 0);
    assert(pT->pTArea+1);
    bool ret = ((pT->pTArea+1)->ftX1in.uni[4] != 1); //I5 Extention
    return ret;
}
```

Código 18. Detectar pieza en fototransistor bandeja del horno.

Otro ejemplo que funciona igual que el anterior manda parar al motor cuando hace contacto con uno de los interruptores de referencia. Este ejemplo es del método ***setMotorLeft()*** de la clase TxtAxis. Si se toca el interruptor el motor se para. La sentencia controla el canal al que está conectado el switch (*chS1*) independientemente de si está conectado al master (*chS1* menor que ocho) o al esclavo (*chS1* mayor que ocho).

```
if (chS1<8?pT->pTArea->ftX1in.uni[chS1]:(pT->pTArea+1)->ftX1in.uni[chS1-8] == 1)
{
    setMotorOff();
}
```

Código 19. Indicar parada de motor cuando hace contacto con interruptor.

La petición *counter* se utiliza para leer los valores del contador. Se puede ver en el método ***moveLeft()*** de la clase TxtAxis1RefSwitch cómo se utiliza el valor del contador del motor para desplazar el eje.

La estructura tipo *output* también se usa para notificar al usuario en el momento en el que algunas de las acciones asíncronas han finalizado.

La petición *cnt_resetted* se utiliza para indicar que el último contador ha sido reiniciado (se debe volver a poner a 0 una vez usada). Paralelamente *cnt_reset_cmd_id (*counter reset command id) informa al usuario cuál ha sido el último contador que ha sido reiniciado.

Por otro lado, *motor_ex_reached* se pone a *true* si el motor alcanza la posición objetivo. La petición *motor_ex_cmd_id* (motor extended command id) informa al usuario cuál ha sido el motor que ha alcanzado su posición objetivo.

CAPÍTULO 3 Análisis y mejora del Almacén

A lo largo del siguiente capítulo se va a llevar a cabo una completa explicación sobre la programación de la bahía de almacenamiento.

En el epígrafe 1 se explicarán los métodos utilizados para el control de la información del almacén, en el que se incluye la búsqueda de piezas y contenedores y la actualización de la información cada vez que se produce un cambio.

En el epígrafe 2 se explicarán los métodos necesarios para llevar a cabo los movimientos físicos de los ejes del transelevador del almacén, necesarios para buscar contenedores y almacenar piezas.

En el epígrafe 3 se explicará el funcionamiento de la máquina de estados del almacén y su comunicación con el robot de aspiración mediante el protocolo MQTT, necesario para sincronizar las tareas entre ambas estaciones.

En el epígrafe 4 se explicarán el conjunto de cambios propuestos, incluyendo en su mayor parte los implementados sobre el funcionamiento de la bahía de almacenamiento, junto con otros propuestos sobre el resto de las estaciones de la fábrica.

Para llevar a cabo el proceso de almacenamiento de piezas en el almacén se han implementados dos clases. Una de ellas es la clase TxtHighBayWarehouseStorage que se encarga del control del estado del almacén, en cuanto a la información de las piezas y la actualización de cambios. La otra clase es la TxtHighBayWarehouse, utilizada para controlar el movimiento físico de los ejes del transelevador.

El programa principal para el control del flujo de estados en el almacén se encuentra en el archivo "TxtHighBayWarehouseRun.cpp". La programación para el funcionamiento del protocolo MQTT entre las estaciones de la fábrica se encuentra en la clase TxtMqttFactoryClient.

1 Control del estado virtual del almacén

Se incluye la explicación de los métodos de la clase TxtHighBayWarehouseStorage. Para llevar a cabo la actualización de la información en el estado del almacén se utilizan los siguientes métodos:

TxtHighBayWarehouseStorage::TxtHighBayWarehouseStorage

Es el método constructor de la clase. Se inicializan los valores de "currentPos" y "nextFetchPos" a un valor cualquiera (-1). Estas dos variables guardan respectivamente los valores donde el transelevador se encuentra en el momento actual y la siguiente posición a la que debe desplazarse.

```
TxtHighBayWarehouseStorage::TxtHighBayWarehouseStorage()
    : filename("Data/Config.HBW.Storage.json")
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "TxtHighBayWarehouseStorage",0);
    if (!loadStorageState())
    {
        resetStorageState();
    }
    currentPos.x = -1;
    currentPos.y = -1;
    //nextStorePos.x = -1;
    //nextStorePos.y = -1;
    nextFetchPos.x = -1;
    nextFetchPos.y = -1;
    Notify();
    print();
}
```

Código 20. Método Constructor de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::loadStorageState

Se carga un estado virtual del almacén desde un archivo, rellenándose cada uno de los valores de la matriz *wp*.

TxtHighBayWarehouseStorage::saveStorageState

Se guarda el estado del almacén en un archivo.

TxtHighBayWarehouseStorage::resetStorageState

Permite borrar el estado virtual del almacén, eliminando los datos que había almacenados en la matriz de piezas *wp* y en la matriz de contenedores *wpc* (en los métodos explicados posteriormente no se utiliza esta matriz para saber el estado de los contenedores, solo se utiliza la matriz *wp*). Se recorren ambas matrices poniendo a cero los valores de *wp* (para indicar que no hay pieza) y a *true* los de *wpc* (para indicar que hay contenedor vacío). Se llama a este método cuando se pasa el llavero de reseteo por el dispositivo NFC.

```
void TxtHighBayWarehouseStorage::resetStorageState() //
//Cuando borramos el estado del almacen pasando el NFC de la medalla se llama a esta función
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "resetStorageState",0);
    for(int i=0;i<3;i++)// bucle para que los 9 valores de la matriz wp[i][j] aparezcan como vacios
    {
        for(int j=0;j<3;j++)
        {
            wp[i][j] = 0; //Matriz de piezas
            wpc[i][j] = true;//Matriz de contenedores
        }
    }
    Notify();
    saveStorageState(); // llama a esta función para guardar el estado del alamacen
}
```

Código 21. Método resetStrageState() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::storeContainer

Actualiza la matriz de piezas tras haberse devuelto un contenedor vacío. Comprueba que la posición *nextFetchPos* (variable que almacena la posición siguiente que debe alcanzar el transelevador) es una posición válida. En el caso de que sea una posición válida (si las coordenadas están dentro de la matriz 3 x 3) se actualiza esa posición en la matriz de piezas a cero, para indicar que en esa posición ya no hay una pieza. Posteriormente, el método devuelve *true*.

```
bool TxtHighBayWarehouseStorage::storeContainer() //Devolver contenedor vacio al almacen
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "storeContainer",0);

    if (isValidPos(nextFetchPos))
    {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextStorePos",0);
         /*Aqui damos el contendor vacio y el contenido de esa
          celda pasa de un estado W R B  a un estado de 0 o vacio */
        wp[nextFetchPos.x][nextFetchPos.y] = 0;
        Notify();
        print();
        return true;
    }
    return false;
}
```

Código 22. Método storeContainer() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::store

Actualiza el estado del almacén una vez que ha sido almacenada una pieza. El método recibe como parámetro un objeto de tipo WorkPiece (pieza que se desea almacenar) y, tras comprobar que la posición de *nextFetchPos* es válida, se actualiza la posición en la matriz con el objeto WorkPiece. Después, el método devuelve *true*.

```
bool TxtHighBayWarehouseStorage::store(TxtWorkpiece _wp) //Actualizar estado almacén tras almacenamiento
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "store wp:{} {} {}",_wp.tag_uid,_wp.type,_wp.state);
    //nextStorePos.x = -1; //set invalid pos
    //nextStorePos.y = -1;
    if (_wp.type == WP_TYPE_NONE)  /* pieza no identificada en ninguno de los tres colores, devolver
                                      false para rechazar pieza */
    {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "NONE -> return false",0);
        return false;

    }
    if (isValidPos(nextFetchPos)) //Posición dentro de las 9 posibles posiciones
        {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextStorePos type {} ",_wp.type);
        wp[nextFetchPos.x][nextFetchPos.y] = new TxtWorkpiece(_wp); /*se actualiza estado del almacén
                                                                      con nuevo objeto */
        Notify();
        print();
        return true;
    }
    return false;
}
```

Código 23.Método store() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::fetch

Se recorre la matriz de piezas para comprobar dónde hay una pieza del tipo que se está buscando (el método recibe el tipo como parámetro) y se almacena en la variable *nextFetchPos* para que la siguiente posición a la que se dirija el transelevador sea la almacenada en esta variable. Después, se actualiza el estado de la matriz borrándose el objeto *WorkPiece* que había en esa posición y se actualiza al valor cero para indicar que ya no hay una pieza en esa posición. Tras completar estos pasos el método devuelve *true*.

```
    for(int i=2;i>=0 && !found;i--)
    {
        for(int j=0;j<3 && !found;j++)
        {
            StoragePos2 p;
            p.x = i; p.y = j;
            if (wp[i][j] == NULL)
                continue;
            if (wp[i][j]->type == t)
            {
                SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "t {} -> nextFetchPos {} {}",t, p.x, p.y);
                nextFetchPos = p;
                found = true;
            }
        }
    }
if (isValidPos(nextFetchPos))
{
    SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextFetchPos type {} ",t);
    delete wp[nextFetchPos.x][nextFetchPos.y];
    wp[nextFetchPos.x][nextFetchPos.y] = 0;
    Notify();
    print();
    return true;
}
return false;
```

Código 24. Método fetch() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::fetchContainer

Se recorre la matriz de piezas para ir a buscar la primera posición en la que haya un contenedor vacío (valor de la posición de la matriz de piezas igual a cero). La posición se almacena en la variable *nextFetchPos*, variable que se utiliza para almacenar la siguiente posición a alcanzar en todos los métodos en los que el transelevador se debe dirigir a una estantería. Tras completar estos pasos el método devuelve *true*.

```
bool TxtHighBayWarehouseStorage::fetchContainer()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetchContainer",0);
    nextFetchPos.x = -1; //set invalid pos
    nextFetchPos.y = -1;
    bool found = false;
    for(int i=0;i<3 && !found;i++)
        {
            for(int j=2;j>=0 && !found;j--)
            {
                StoragePos2 p;
                p.x = i; p.y = j;
                if (wp[i][j] == 0)
                {
                    SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "cont -> nextFetchPos {} {}",p.x, p.y);
                    nextFetchPos = p;
                    found = true;
                }
            }
        }

        if (isValidPos(nextFetchPos))
        {
            SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextFetchPos cont ",0);
            wp[nextFetchPos.x][nextFetchPos.y] = 0;
            Notify();
            print();
            return true;
        }
        return false;
}
```

Código 25. Método fetchContainer() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::isValidPos

Se comprueba que la posición que se pasa como parámetro al método es una posición válida y se devuelve *true* en caso de que sí lo sea.

```
bool TxtHighBayWarehouseStorage::isValidPos(StoragePos2 p)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "isValidPos {} {}",p.x,p.y);
    bool ret = false;
    if ((p.x >= 0) && (p.x <= 2) && (p.y >= 0) && (p.y <= 2))
    {
        ret = true;
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK",0);
    }
    return ret;
}
```

Código 26. Método isValidPos() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::canColorBeStored

Se comprueba si hay espacio en la matriz de piezas. Para ello se recorre la matriz de piezas y se utiliza un contador (*numc*) que se incrementa en el caso de que en esa posición ya exista una pieza del tipo que se pasa como parámetro al método. Si el contador es menor que 3 se puede almacenar una pieza de ese tipo y, por lo tanto, el método devuelve *true*.

```
bool TxtHighBayWarehouseStorage::canColorBeStored(TxtWPType_t c)//comprueba si hay hueco en el almacen para un color dado
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "canColorBeStored {}", c);
    bool ret = false;
    int numc = 0;
    for(int i=0;i<3;i++) //comprobar que queda espacio en el estante para el color c
    {
        for(int j=2;j>=0;j--)
        {
            if (wp[i][j] != 0){ // si el estante no está ocupado por una pieza de trabajo
                if (wp[i][j]->type == c) numc++; // si la pieza del estante es del tipo c sumar uno numc++
            }
        }
    }
    if (numc < 3)//
    {
        ret = true;
    }
    return ret;
}
```

Código 27.Método canColorBeStored() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::getStockMap

Se crea una variable de tipo *map*[3] para asignar a cada posición (A1, A2, A3…) la posición de la matriz de piezas.

```
/*Da a cada una de las posiciones del almacen (A1 A2... C3) un valor de la matriz wp,
 en la que se va a dar un valor en función de su contenido */
Stock_map_t TxtHighBayWarehouseStorage::getStockMap()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getStockMap",0);
    Stock_map_t map_wps;
    map_wps["A1"] =  wp[0][0];
    map_wps["A2"] =  wp[1][0];
    map_wps["A3"] =  wp[2][0];
    map_wps["B1"] =  wp[0][1];
    map_wps["B2"] =  wp[1][1];
    map_wps["B3"] =  wp[2][1];
    map_wps["C1"] =  wp[0][2];
    map_wps["C2"] =  wp[1][2];
    map_wps["C3"] =  wp[2][2];
    return map_wps;
}
```

Código 28. Método getStockMap() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::charType

Devuelve una variable tipo *char* para indicar el tipo de pieza que hay en una posición concreta. El producto puede ser rojo (“R”), blanco(“W”) o azul (“B”). En caso de almacenar un contenedor vacío se utiliza (“_”). Si la pieza no ha sido identificada dentro de uno de los tres colores (WP_TYPE_NONE) se representaría con una exclamación (“!”).

[3] La clase map se utiliza para almacenar y recuperar datos de una colección de forma que cada elemento es un par que contiene un valor y una clave de ordenación.

```
//Indica cual es el contenido de cada uno de los espacios del almacen (no identificado, blanco, rojo o azul)
char TxtHighBayWarehouseStorage::charType(int x, int y)
{

    char c = '?';
    if (wp[x][y])
    {
        switch(wp[x][y]->type)
        {
        case WP_TYPE_NONE: //0
            c = '!';
            break;
        case WP_TYPE_WHITE://1
            c = 'W';
            break;
        case WP_TYPE_RED://2
            c = 'R';
            break;
        case WP_TYPE_BLUE://3
            c = 'B';
            break;
        default:
            break;
        }
    } else {
        c = '_';
    }
    return c;
}
```

Código 29. Método charType() de la Clase TxtHighBayWarehouseStorage.

TxtHighBayWarehouseStorage::print

Se llama al método anterior con cada posición de la matriz *wp* para mostrar por pantalla el estado del conjunto del almacén.

```
/*Muestra el contenido de cada uno los 9 espacios del almacen,
los valores van del 0 al 2 con x e y.
como hemos visto en la función anterior, el valor charType podrá
ser no identificado, blanco, rojo o azul */
void TxtHighBayWarehouseStorage::print()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "print",0);
    std::cout << charType(0,0) << charType(1,0) << charType(2,0) << std::endl;
    std::cout << charType(0,1) << charType(1,1) << charType(2,1) << std::endl;
    std::cout << charType(0,2) << charType(1,2) << charType(2,2) << std::endl;
}
```

Código 30. Método print() de la Clase TxtHighBayWarehouseStorage

2 Control de los movimientos físicos en el almacén

Los métodos explicados a continuación son los encargados de llevar a cabo el control de los movimientos de los ejes del transelevador para almacenar las piezas e ir a buscarlas cuando sean solicitadas.

TxtHighBayWarehouse::stop

Esta función ordena la parada de los tres ejes de la máquina transportadora. Es la función a la que llamar en caso de emergencia. Si esta subestación tuviera una seta roja, se utilizaría para llamar a esta función.

```
void TxtHighBayWarehouse::stop() //parada
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "stop",0);
    axisX.stop();
    axisY.stop();
    axisZ.stop();
}
```

Código 31. Método stop() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::moveRef

Mueve los tres ejes hacia la posición de referencia. Mientras dura el proceso la fábrica pasa a estado ocupado (SM_BUSY). En este movimiento del transelevador se recoge el brazo, y los ejes x e y se desplazan hasta que tocan los interruptores. Cuando se ha alcanzado la posición de referencia el estado de la fábrica pasa a preparada (SM_RADY).

```
void TxtHighBayWarehouse::moveRef() //mover hasta el lugar de referencia. Indicar estado ocupado
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveRef",0);
    setActStatus(true, SM_BUSY);


    //// Thread para mover todos los ejes paralelamente, ¿se consigue ganar tiempo? PRUEBA 05.05
    //std::thread tz = axisZ.moveS1Thread();
    axisZ.moveS1(); //encoger brazo
    std::thread tx = axisX.moveRefThread(); //Eje X Clase TxtAxis1RefSwitch
    std::thread ty = axisY.moveRefThread();//Eje Y Clase TxtAxis1RefSwitch

    tx.join();
    ty.join();
    //tz.join();
    setActStatus(false, SM_READY); //al finalizar indicador estado Listo
}
```

Código 32. Método moveRef de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::moveJoystick

Esta función permite desplazar el transelevador mediante los *joysticks*. Solo se puede acceder a ella por petición del usuario de la maqueta. En el uso cotidiano de la fábrica no es accesible.

TxtHighBayWarehouse::moveConv

Su función es desplazar el transelevador hacia la bandeja interior de piezas al almacén. Se obtienen las coordenadas alcanzadas por el *encoder* en ambos ejes desde la clase de calibración del almacén.

El método recibe un *bool* (stop). En el caso de que se reciba *true* el brazo del transelevador se recoge y queda recogido sobre la posición de entrada de piezas por la cinta transportadora. En el caso de que se reciba false, el brazo queda extendido sobre esta misma posición. El método devuelve la última posición alcanzada por el *encoder* en los ejes x e y.

```
EncPos2 TxtHighBayWarehouse::moveConv(bool stop) // Movimiento hacia la bandeja interior del almacén
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveConv",0);
    if (!stop)
    {
        axisZ.moveS1(); //eje z dentro si stop es false
    }
    EncPos2 pos2 = calibData.conv; //obtener pulsos de encoder, para mover a bandeja interior, del archivo calib.
    SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "pos:{} {}", pos2.x, pos2.y);
    //Mover ejes x e y paralelamente
    std::thread tx = axisX.moveAbsThread(pos2.x);
    std::thread ty = axisY.moveAbsThread(pos2.y);
    tx.join();
    ty.join();
    return pos2;
}
```

Código 33. Método moveConv() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::moveCR

El método recibe las coordenadas del almacén (fila i, columna j) a las que se debe dirigir para desplazarse hasta un contenedor concreto.

En primer lugar, el transelevador recoge el brazo. Se obtienen las coordenadas de la clase de calibración y con las coordenadas concretas se inicia el movimiento de los ejes x e y. El método devuelve la última posición alcanzada por el *encoder*.

```
EncPos2 TxtHighBayWarehouse::moveCR(int i, int j) //Movimiento a la estanteria (i, j)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveCR idx:{} {}", i, j);
    axisZ.moveS1(); //eje z fuera
    EncPos2 pos2;
    pos2.x = calibData.hbx[i];
    pos2.y = calibData.hby[j];
    SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "pos:{} {}", pos2.x, pos2.y);
    std::thread tx = axisX.moveAbsThread(pos2.x);
    std::thread ty = axisY.moveAbsThread(pos2.y);
    tx.join();
    ty.join();
    return pos2;
}
```

Código 34. Método moveCR() de la Clase TxtHighBayWarehouse

TxtHighBayWarehouse::getCR

El transelevador recibe las instrucciones para desplazarse hasta una posición concreta del almacén (fila i, columna j), en la que haya un contenedor vacío o un contenedor con una pieza, y lo coge. Para ello, el brazo se estira bajo el contenedor, lo eleva ligeramente y se recoge.

```
bool TxtHighBayWarehouse::getCR(int iCol, int iRow) // coger de una posicion
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getCR idx:{} {}", iCol, iRow);
    EncPos2 p2 = moveCR(iCol,iRow); //obtiene pulsos de encoder necesarios y mueve los ejes a esa posición
    axisZ.moveS2(); // eje z dentro
    bool r = axisY.moveAbs(p2.y - ydelta); // eje y bajar 40
    axisZ.moveS1(); // eje z fuera
    return r;
}
```

Código 35. Método getCR() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::putCR

El transelevador recibe las instrucciones para desplazar el brazo al estante que no esté ocupado. Luego, el brazo del almacén deposita el contenedor en el estante y se recoge sobre la posición alcanzada.

Para ello, recibe la última posición alcanzada por los ejes del método ***moveCR()***, sube el brazo ligeramente para alcanzar el estante, estira el brazo y baja ligeramente una vez que el contenedor ya está apoyado en el estante. Seguidamente, se recoge el brazo.

```
bool TxtHighBayWarehouse::putCR(int iCol, int iRow)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "putCR idx:{} {}", iCol, iRow);
    EncPos2 p2 = moveCR(iCol,iRow);
    //uint16_t posEndY = axisY.getPosEnd();
    axisY.moveAbs(p2.y - ydelta); //eje y bajar 40
    axisZ.moveS2(); // eje z dentro
    bool r = axisY.moveAbs(p2.y + ydelta); // eje y subir 40
    axisZ.moveS1(); // eje z fuera
    return r;
}
```

Código 36. Método putCR() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::getConv

El método se encarga de dirigir el transelevador hasta la posición de la bandeja interior del almacén y luego coger el contendor (lleno o vacío).

Para ello, se hace desplazar el brazo hacia la bandeja interior mediante la llamada al método ***moveConv()*** y se estira el brazo. Se llama al método ***moveIn()*** de la clase ConvBelt para que desplace el contenedor desde la bandeja exterior a la interior. Con el brazo estirado se hace descender ligeramente para apoyar el contenedor sobre este.

```
bool TxtHighBayWarehouse::getConv(bool stop) //coger pieza de bandeja interior del almacén (cinta transportadora)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getConv",0);
    EncPos2 p2 = moveConv(stop); //mover a cinta transportadora
    axisZ.moveS2(); // eje z dentro
    convBelt.moveIn(); // Mover pieza a bandeja interior meddiante cinta transportadora
    bool r = axisY.moveAbs(p2.y - ydelta); // eje y bajar 40
    if (!stop)
    {
    axisZ.moveS1(); // si stop es false mover eje z fuera, si stop es true eje z dentro
    }
    return r;
}
```

Código 37. Método getConv() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::putConv

El método se encarga de depositar el contenedor en la bandeja interior del almacén y, tras esto, mover la cinta transportadora para dejar el contenedor en la bandeja exterior del almacén.

Se inicia el movimiento hacia la bandeja interior mediante la llamada al método ***moveConv()*** con el brazo recogido. El brazo se extiende y luego baja ligeramente para depositar el contenedor sobre la cinta. Posteriormente, se llama al método ***moveOut()*** de la clase ConvBelt para que saque el contenedor hacia la bandeja exterior.

```
bool TxtHighBayWarehouse::putConv(bool stop) // Depositar pieza en cinta y sacar fuera
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "putConv",0);
    EncPos2 p2 = moveConv(); // mover a cinta transportadora
    axisZ.moveS2(); // eje z dentro
    bool r = axisY.moveAbs(p2.y + ydelta); // eje y subir 40
    convBelt.moveOut(); // cinta transportadora out
    if (!stop)
    {
        axisZ.moveS1(); //eje z fuera si stop es true
    }
    return r;
}
```

Código 38. Método putConv() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::store

Su función es la de almacenar un contenedor lleno en el estante correcto y volver a la posición de referencia.

El método (Código 39) recibe un objeto de tipo WorkPiece y devuelve true tras almacenar el contenedor lleno con la pieza en el almacén. Durante el proceso de almacenamiento la fábrica pasa a estado ocupada. Para ir pasando a los pasos siguientes hay una variable *bool* definida como *r*, que se encarga de que se vayan dando los pasos una vez que cada uno de los métodos se ha efectuado correctamente (la variable solo se pone a *false* si el método encargado de mover los ejes x e y no puede desplazarle hasta la posición deseada). La secuencia de pasos es la siguiente:

1. Se llama al método *store* de la clase TxtHighBayWarehouseStorage y devuelve *true* una vez que ha sido guardada la posición libre sobre la que almacenar la pieza en la variable *nextFetchPos*. Si el método devuelve *true* se realizan los siguientes pasos. En caso contrario la fábrica pasaría a indicar un error.
2. Se accede a *nextFetchPos* para guardar esta posición libre en la variable *p* de tipo *StoragePos2*.
3. Se llama al método ***getConv()*** para ir a por el contenedor lleno a la posición de entrada de piezas.
4. Se llama al método ***putCR()*** con las coordenadas x e y almacenadas en la variable *p* para llevar el contenedor lleno hasta el estante determinado.
5. Se actualiza el estado del almacén y el brazo se desplaza a la posición de referencia.

```
bool TxtHighBayWarehouse::store(TxtWorkpiece wp)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "store {}", wp.type);
    setActStatus(true, SM_BUSY);
    if (storage.store(wp)) // validar posición a almacenar y actualizar matriz
    {
        StoragePos2 p = storage.getNextStorePos(); // Obtener posición nextFetchPos para almacenar
        if (p.x<0 || p.y<0) return false;
        bool r = getConv(true); // Mover translevador a cinta, cinta transportadora in
                                //y coger contenedor de bandeja interior
        if (!r) return false;
        r = putCR(p.x,p.y); // Mover a estanteria y depositar contenedor
        if (!r) return false;
        setActStatus(false, SM_READY);
        storage.saveStorageState(); // actualizar estado del almacén (nube)
        moveRef(); // Mover a referencia
                    /*moveCR(1,1) PRUEBA 20.05 mover a posición (1,1) como posición de referencia
                     * para minimizar distancias*/

        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 39. Método store() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::storeContainer

Su función es la de almacenar un contenedor vacío una vez que la pieza ha salido del almacén.

El método no recibe parámetros. Funciona exactamente igual que el método anterior pero, en este caso, se llama al método ***StoreContainer()*** de la clase TxtHighBayWarehouseStorage para encontrar el lugar del que se ha sacado previamente el contenedor. Después, se actualiza el estado del almacén para indicar que el contenedor vacío ya está en su sitio.

```
bool TxtHighBayWarehouse::storeContainer() //Almacenar un contenedor
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "storeContainer",0);
    setActStatus(true, SM_BUSY); //estado ocupado
    if (storage.storeContainer())
    {

        StoragePos2 p = storage.getNextStorePos(); //obtener posición nextFetchPos
        if (p.x<0 || p.y<0) return false;
        bool r = getConv(true);// Mover translevador a cinta, cinta transportadora in
                               //y coger contenedor de bandeja interior
        if (!r) return false;
        r = putCR(p.x,p.y);// Mover a estanteria y depositar contenedor
        if (!r) return false;
        setActStatus(false, SM_READY);
        storage.saveStorageState();// actualizar estado del almacén (nube)
        moveRef(); // Mover a posición de referencia despues de almacenar contenedor
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 40. Método storeContainer() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::fetch

Su función es la de ir a coger un contendor lleno con la pieza del tipo que se desea y dejar el contenedor con la pieza preparada en la bandeja de salida de piezas.

El método recibe un objeto de tipo WorkPiece y devuelve *true* tras coger un contenedor lleno con la pieza del tipo indicado del almacén. Mediante el método ***getCR()*** el transelevador se dirige a una posición concreta del almacén y coge una pieza. Después, se utiliza el método ***putConv()*** para ir a depositar la pieza en la bandeja interior y saca la pieza fuera mediante la cinta transportadora.

```
bool TxtHighBayWarehouse::fetch(TxtWPType_t t) //Extraer un contenedor lleno
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetch {}", t);
    setActStatus(true, SM_BUSY);
    if (storage.fetch(t))
    {
        StoragePos2 p = storage.getNextFetchPos();
        if (p.x<0 || p.y<0) return false;
        bool r = getCR(p.x, p.y); // Mover a estanteria y coger contenedor
        if (!r) return false;
        r = putConv(true); //depositar pieza en cinta y sacar pieza fuera
        if (!r) return false;
        setActStatus(false, SM_READY);
        storage.saveStorageState(); // actualizar estado del almacén (nube)
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 41. Método fetch() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::fetchContainer

Su función es la de extraer un contenedor vacío. Se comprueba el resultado del método ***fetchContainer()*** de la clase TxtHighBayWarehouseStorage , que devuelve *true* una vez que se ha encontrado la posición del contenedor libre más cercano. Funciona igual que el método ***fetch()*** .

```
bool TxtHighBayWarehouse::fetchContainer() //Extraer un contenedor vacio
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetchContainer", 0);
    setActStatus(true, SM_BUSY);
    if (storage.fetchContainer())
    {
        StoragePos2 p = storage.getNextFetchPos();
        if (p.x<0 || p.y<0) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        bool r = getCR(p.x, p.y); //Mover a estantería y extraer contenedor.
        if (!r) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        r = putConv(true);// depositar contenedor en cinta y sacar fuera
        if (!r) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        setActStatus(false, SM_READY);
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 42. Método fetchContainer() de la Clase TxtHighBayWarehouse.

TxtHighBayWarehouse::canColorBeStored

Llama al método ***canColorBeStored()*** de la clase TxtHighBayWarehouseStorage, que devuelve *true* si existe hueco para almacenar la pieza de un color.

TxtHighBayWarehouse::setSpeed

Recibe la velocidad como parámetro y se la asigna a todos los ejes.

3 La calibración del Almacén.

TxtHighBayWarehouseCalibData::load

Se crean las variables para calibrar el almacén y se cargan sobre un archivo en formato JSON.

TxtHighBayWarehouseCalibData::saveDefault

En esta función se asignan valores por defecto a las posiciones de los ejes x e y, de tal forma que combinadas se obtienen las nueve posibles posiciones de las estanterías del almacén y junto con las dos posibles del eje z (brazo estirado o brazo encogido).

TxtHighBayWarehouseCalibData::save

Se llevan a eventos JSON los valores de las coordenadas de cada posición. Los eventos JSON permiten que software externo comience *workflows*, mediante demandas HTTP.

TxtHighBayWarehouse::fsmStep

Es el método encargado de controlar la transición de estados del almacén.

4 Mensajes MQTT Almacén.

A continuación, se describen los tipos de mensajes MQTT del almacén junto con los métodos asociados para llevar a cabo la comunicación. Los métodos se pueden encontrar en los archivos "TxtMqttFactoryClient". Como la comunicación se produce entre el HBW y el VGR, se añade también una tabla con los tipos de mensajes MQTT que utiliza el VGR.

Para explicar la FSM del almacén nos centramos en la comunicación de modo local (Ver última fila de la Tabla 3 y Tabla 4), que es la que se produce entre los propios clientes (estaciones). El VGR es el que da las órdenes publicando mensajes de tipo "*fl/vgr/do*" al VGR. Por otro lado, el HBW es el que las recibe dando una respuesta cada vez que cumple con la tarea mediante menajes de tipo "*fl/hbw/ack*".

Los mensajes son de tipo PUBLISH, cuya función es la de publicar mensajes sobre aquellos clientes que están suscritos. Hay tres tipos dependiendo del tipo de mensaje que se quiere publicar.

En el Almacén, el primer tipo de mensaje o *topic* sirve para publicar información relativa al estado de la estación, el segundo publica información sobre el estado de las estanterías del almacén y el tercero publica mensajes tipo *ack* (mensajes para publicar que una tarea ha sido completada). En la siguiente tabla se incluye para cada tipo de mensajes el componente publicador (título del tema o *topic* del mensaje), la sintaxis del *topic*, el *payload* (el propio contenido del mensaje) y una pequeña descripción del contenido del *payload*.

Tabla 3. Mensajes MQTT tipo publish del HBW.

Componente Publicador (PUBLISH)	topic	payload	Descripción
Estado de la estación HBW	f/i/state/hbw	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"hbw", "code":0, "description":"text", "active":1, "target":""}	Mensaje para mostrar el estado de la estación HBW (ts, código para luces led, descripción, estado activo, target: estación con la que establece comunicación (VGR))
`void publishStateStation(const std::string station, TxtLEDSCode_t code, const std::string desc, long timeout, int active=-1, const std::string target="");`			
Estado del Almacén	f/i/stock	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "stockItems": [{ "workpiece": { "id":"123456789ABCDE", "type":"<BLUE/WHITE/RED>", "state":"<RAW/PROCESSED>" }, "location":"A1" },{ ... },{ "workpiece":null, "location":"B3" }] }	Mensaje para mostrar estado del almacén, indicando time stamp y características de la pieza (id, tipo, estado y localización).
`void publishStateHBW(TxtLEDSCode_t code, const std::string desc, long timeout, int active=-1, const std::string target="") { publishStateStation("hbw",code,desc,timeout,active,target); }`			
Recibo de Estado del HBW (ack)	fl/hbw/ack	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	Mensaje para indicar evento completado, según el código: 0=HBW_EXIT, 1=HBW_FETCHED, 2=HBW_STORED, 3=HBW_CALIB_NAV, 4=HBW_CALIB_END
`void publishHBW_Ack(TxtHbwAckCode_t code, TxtWorkpiece* wp, long timeout);`			

En el VGR se tienen cuatro temas, descritos en la Tabla 4. El primero publica información sobre el estado de la estación, el segundo publica información relativa al sensor DSI, el tercero publica información del sensor DSO, y el último sirve para enviar órdenes al resto de clientes.

Tabla 4. Mensajes MQTT tipo publish del VGR.

Componente Publicador (PUBLISH)	topic	payload	Descripción
State VGR	f/i/state/vgr	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"vgr", "code":0, "description":"text", "active":1, "target":"hbw"}	Mensaje para mostrar el estado de la estación VGR (ts, código para luces led, descripción, estado activo, target: estación con la que establece comunicación (HBW))
`void publishStateVGR(TxtLEDSCode_t code, const std::string desc, long timeout, int active=-1, const std::string target="" { publishStateStation("vgr",code,desc,timeout,active,target); }`			
State DSI (VGR)	f/i/state/dsi	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"dsi", "code":0, "description":"text", "active":1}	Mensaje para mostrar estado del sensor *dsi* indicando time stamp, estación, código encender color de luces led, descripción y estado activo
`void publishStateDSI(TxtLEDSCode_t code, const std::string desc, long timeout, int active=-1, const std::string target="") { publishStateStation("dsi",code,desc,timeout,active,target); }`			
State DSO (VGR)	f/i/state/dso	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	Mensaje para mostrar estado del sensor *dso* indicando time stamp, estación, código encender color de luces led, descripción y estado activo.
`void publishStateDSO(TxtLEDSCode_t code, const std::string desc, long timeout, int active=-1, const std::string target="") { publishStateStation("dso",code,desc,timeout,active,target); }`			
VGR Trigger	fl/vgr/do	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	Indicar tipo evento producido sobre otra estación según el código: 0=VGR_EXIT, 1=VGR_HBW_FETCHCONTAINER, 2=VGR_HBW_STORE_WP, 3=VGR_HBW_FETCH_WP, 4=VGR_HBW_STORECONTAINER, 5=VGR_HBW_RESETSTORAGE, 6=VGR_HBW_CALIB, 7=VGR_MPO_PRODUCE, 8=VGR_SLD_START
`void publishVGR_Do(TxtVgrDoCode_t code, TxtWorkpiece* wp, long timeout);`			

5 La Máquina de Estados del Almacén.

El diagrama de estados que explica el funcionamiento del Almacén es el de la Figura 22.

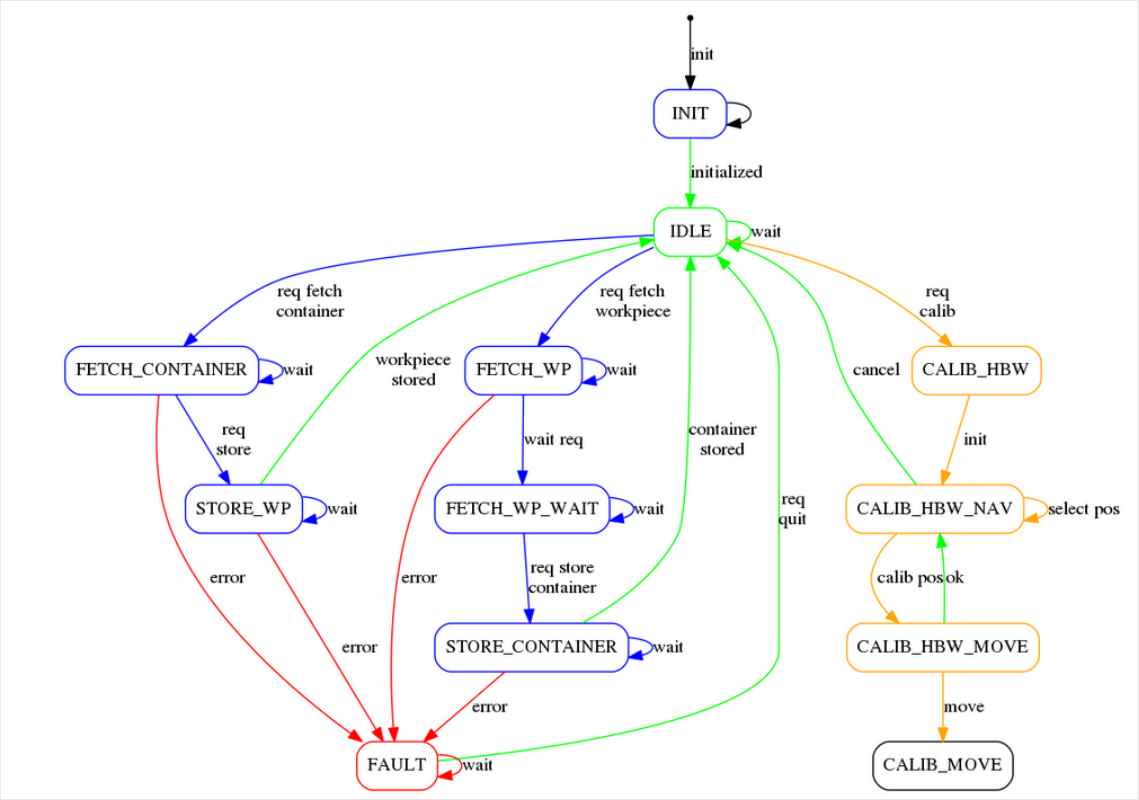


Figura 22. Diagrama de Estados del Almacén.

Las variables encargadas de controlar la transición entre estados (Código 43) junto con los métodos encargados de activar estas variables están definidas dentro del archivo de cabecera de la clase TxtHighBayWarehouse. A continuación, se muestra una figura con las variables encargadas de controlar la transición entre estados del almacén.

```
bool reqQuit;
TxtWorkpiece* reqVGRwp;
bool reqVGRfetchContainer;
bool reqVGRstore;
bool reqVGRfetch;
bool reqVGRstoreContainer;
bool reqVGRcalib;
bool reqVGRresetStorage;
TxtJoysticksData joyData;
bool reqJoyData;
```

Código 43. Variables utilizadas para llevar a cabo la transición de estados del almacén.

A continuación, se muestran los métodos (Código 44) encargados de activar las distintas variables que marcan la transición entre los estados de la FSM del almacén.

```
void requestQuit() {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestQuit",0);
    reqQuit= true;
}
/* local */
void requestExit(const std::string name) {
    std::cout << "program terminated by " << name << std::endl;
    spdlog::get("file_logger")->error("program terminated by {}",name);
    exit(1);
}
void requestVGRfetchContainer(TxtWorkpiece* wp) { //llamar a HBW para que vaya a por contenedor
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRfetchContainer",0);
    reqVGRwp = wp;
    reqVGRfetchContainer= true;
}
void requestVGRstore(TxtWorkpiece* wp) { //almacenar wp
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRstore",0);
    reqVGRwp = wp;
    reqVGRstore= true;
}
void requestVGRfetch(TxtWorkpiece* wp) { //buscar wp
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRfetch",0);
    reqVGRwp = wp;
    reqVGRfetch= true;
}
void requestVGRstoreContainer(TxtWorkpiece* wp) { //almacenar contenedor
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestVGRstoreContainer",0);
    reqVGRwp = wp;
    reqVGRstoreContainer= true;
}
void requestVGRcalib() {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestVGRcalib",0);
    reqVGRcalib= true;
}
void requestVGRresetStorage() {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestVGRresetStorage",0);
    reqVGRresetStorage= true;
}
void requestJoyBut(TxtJoysticksData jd) {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestJoyBut",0);
    joyData = jd;
    reqJoyData = true;
}
```

Código 44. Métodos encargados de activar variables de transición de estados.

El cliente HBW recibe los mensajes y en función del mensaje que se reciba se llama a los métodos encargados de controlar las variables que marcan las transiciones entre estados. La comunicación entre los clientes se realiza dentro del archivo *main* del cliente. Como se ha visto en el epígrafe dedicado a explicar el protocolo MQTT, la comunicación se realiza mediante mensajes que están clasificados en distintos *topics* en función del tipo.

Para poder explicar cómo realiza el almacén los pasos necesarios para llevar a cabo todo el proceso de almacenaje es necesario explicar conjuntamente ciertos pasos de las máquinas de estados del HBW y el VGR, ya que es necesaria la sincronización entre ambos clientes.

A continuación, se explicará paso por paso el funcionamiento y la sincronización entre el VGR y el HBW. Aunque se incluirán imágenes del código en cada paso, la explicación puede resultar un tanto liosa porque se irán comentado la transición entre de ambas máquinas de estados al mismo tiempo. Es por ello por lo que se recomienda comprobar los diagramas de estados del HBW y el VGR al tiempo que se sigue la explicación.

El primer paso que realiza el HBW es ir a por un contenedor vacío. Será el VGR el encargado de lanzar el mensaje al HBW para que vaya a por el contenedor. Es importante conocer en qué momento se produce la comunicación entre ambos clientes. Después de haberse identificado el color de la pieza y justo cuando la pieza vuelve por segunda vez a pasarse por el NFC para ser marcada como RAW, el VGR pasa al estado *STORE_WP_VGR* (Código 45).

```
case STORE_WP_VGR: //Estado almacenar pieza
{
    printState(STORE_WP_VGR);

    assert(mqttclient);
    //Mandar mensaje a HBW para que vaya a por contenedor
    mqttclient->publishVGR_Do(VGR_HBW_FETCHCONTAINER, reqWP_HBW, TIMEOUT_MS_PUBLISH);

    dps.setActiveDSI(false);
    if (dps.getLastColor() == WP_TYPE_NONE)
        //PRUEBA 21. 06 pieza blanca como defectuosa.
    {
        moveWrongRelease(); // llevar a bandeja para piezas defectuosas
        FSM_TRANSITION( FAULT, color=red, label='wrong color' );
        break;
    }

    assert(reqWP_HBW);
    reqWP_HBW->printDebug();
    reqWP_HBW->type = dps.getLastColor(); //Guardar color de pieza en variable reqWP_HBW

    moveToHBW(); //Desplazamiento VGR hacia bandeja exterior almacén.
    FSM_TRANSITION( STORE_WP, color=blue, label='transport to HBW' ); //Pasar a estado STORE_WP
    break;
}
```

Código 45. Estado STORE_WP_VGR de la máquina de estados del VGR.

Será en este estado cuando el VGR lanza el mensaje MQTT al HBW para que vaya a por el contenedor (*PublishVGR_Do: VGR_HBW_FETCHCONTAINER*). El VGR iniciará el desplazamiento hacia la bandeja exterior del HBW para soltar la pieza en el contenedor y pasará al estado *STORE_WP*

Mediante este mensaje se llama al método ***requestVGRfetchContainer()*** (Código 46). Este método pone a *true* la variable *reqVGRfetchContainer*, que será la encargada de hacer que el HBW pase desde el estado IDLE (inactivo) al estado *FETCH_CONTAINER*.

```
case ft::VGR_HBW_FETCHCONTAINER:
    hbw_.requestVGRfetchContainer(wp);
```

```
void requestVGRfetchContainer(TxtWorkpiece* wp) { //llamar a HBW para que vaya a por contenedor
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRfetchContainer",0);
    reqVGRwp = wp;
    reqVGRfetchContainer= true;
```

Código 46. Llamada a método requestVGRfetchContainer()

Pongamos que el HBW ya está en el estado *FETCH_CONTAINER* (Código 47). Desde este estado se llama al método ***fetchContainer()*** de la Clase TxtHighBayWarehouse, que se encarga de extraer un contenedor vacío.

```
case FETCH_CONTAINER:
{
    printState(FETCH_CONTAINER);
    if (fetchContainer()) //Coger contenedor y dejarlo listo en bandeja exterior de piezas.
    {
        assert(mqttclient);
        //Mandar mensaje, contenedor listo
        mqttclient->publishHBW_Ack(HBW_FETCHED, reqVGRwp, TIMEOUT_MS_PUBLISH);
        FSM_TRANSITION( STORE_WP, color=blue, label='req\nstore' ); //Pasar a estado STORE_WP
    }
    else
    {
        //Si fetchContainer devuelve false -> estado FAULT
        FSM_TRANSITION( FAULT, color=red, label='error' );
    }
```

Código 47. Estado FETCH_CONTAINER del HBW.

Una vez que el HBW ha ido a buscar el contenedor y ha sido extraído a la bandeja exterior del almacén para dejarlo preparado para que el VGR suelte la pieza, el método ***fetchContainer()*** devolverá *true*. Llegados a este caso, el HBW publicará un mensaje MQTT (*PublishHBW_Ack: HBW_FETCHED*) para avisar de que el contenedor ya está listo en su sitio. Tras esto el HBW pasará a estado (*STORE_WP*) (Código 48).

```
case ft::HBW_FETCHED:
    vgr_.requestHBWfetched(wp);
```

```
void requestHBWfetched(TxtWorkpiece* wp) { //contenedor o pieza
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestHBWfetched",0);
    reqWP_HBW = wp;
    reqHBWfetched = true;
}
```

Código 48. Llamada a método requestHBWfetched().

El VGR que había pasado al estado *STORE_WP* (Código 49) tras haberse puesto la variable *reqHBWfetched* a *true* soltará la pieza en el contenedor (método ***release()***). Se puede ver en la Figura 23 el momento previo a que el VGR suelte la pieza en el contenedor.

```
case STORE_WP:
{
    printState(STORE_WP);

    if (reqHBWfetched )  // && isDetectedContainer() PRUEBA 22.04
    {

        release(); // soltar pieza de ventosa
        assert(reqWP_HBW);
        reqWP_HBW->printDebug();
        //indiciar timestamp inicio almacenamiento
        proStorage.setTimestampNow(reqWP_HBW->tag_uid, WAREHOUSING_INDEX);
        assert(mqttclient);
        //Mandar mensaje a HBW para almacenar pieza
        mqttclient->publishVGR_Do(VGR_HBW_STORE_WP, reqWP_HBW, TIMEOUT_MS_PUBLISH);
        moveRef(); //mover a referencia
        FSM_TRANSITION( IDLE, color=green, label='fetched' ); //pasar a estado de reposo
        reqHBWfetched = false;
    }
    else
    {
        FSM_TRANSITION( FAULT, color=red, label='no container' );
    }
```

Código 49. Estado STORE_WP de la máquina de estados del VGR.

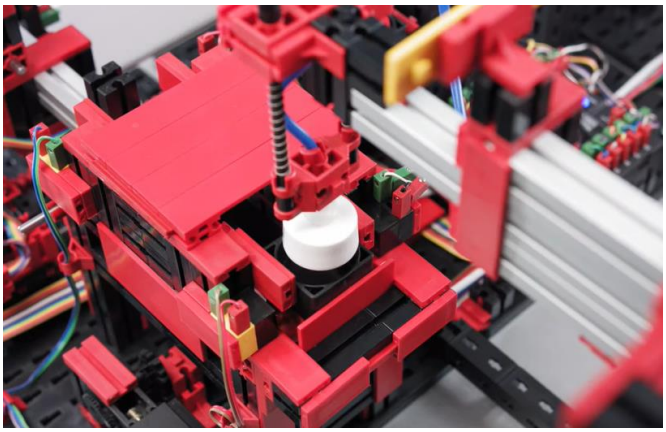


Figura 23. VGR antes de soltar pieza en contenedor de bandeja exterior del almacén.

Una vez soltada la pieza el VGR manda un mensaje al HBW para que inicie el almacenamiento (*PublishVGR_Do: VGR_HBW_STORE_WP*). Justo después el VGR se va a su posición de referencia, apuntando hacia la DPS (Figura 24) y pasa al estado IDLE.

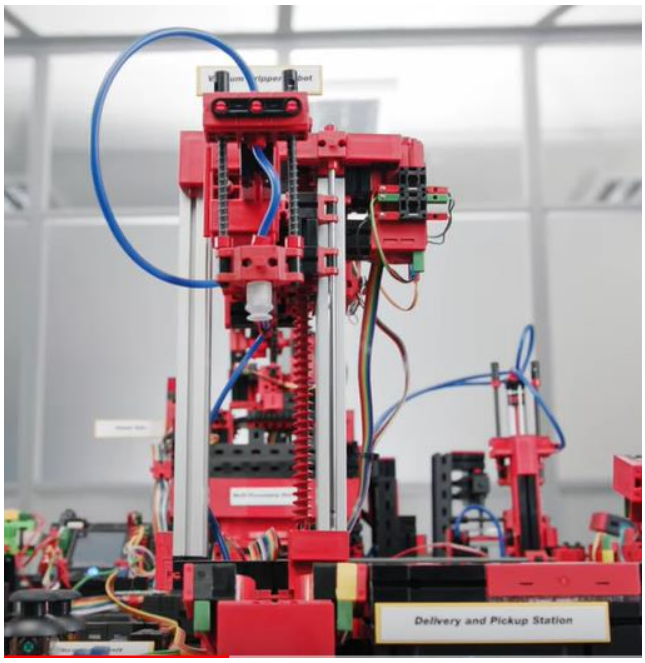


Figura 24. VGR posicionado en posición de referencia.

```
case ft::VGR_HBW_STORE_WP:
    hbw_.requestVGRstore(wp);
```

```
void requestVGRstore(TxtWorkpiece* wp) { //almacenar wp
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRstore",0);
    reqVGRwp = wp;
    reqVGRstore= true;
}
```

Código 50. Llamada a método requestVGRstore().

Ahora el HBW ya está listo para almacenar la pieza. El HBW pasa al estado *STORE_WP* en el que almacenará la pieza (Figura 25) en función del tipo. Una vez que esta ha sido almacenada el HBW pasará al estado de reposo IDLE.

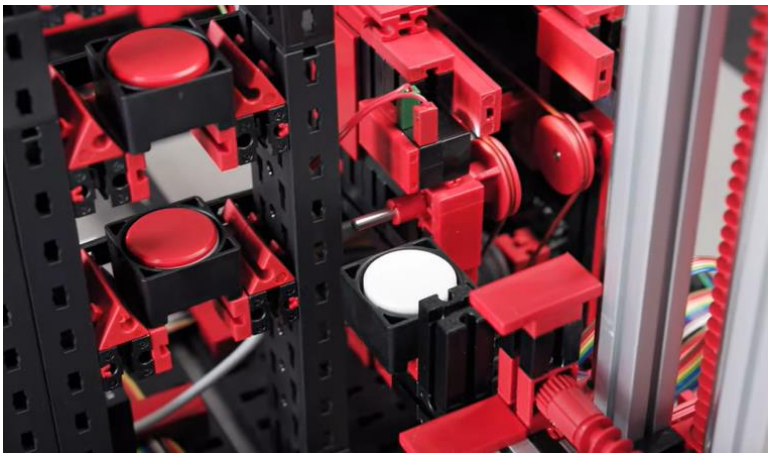


Figura 25.Transelevador del VGR almacenando la pieza.

```
case STORE_WP:
{
    printState(STORE_WP);
    if (reqVGRstore /* && (convBelt.isDetectedContainerOut()or convBelt.isDetectedContainerIn())
    {
        if (reqVGRwp && store(*reqVGRwp))
        {
            FSM_TRANSITION( IDLE, color=green, label='workpiece\nstored' );
        }
        else
        {
            FSM_TRANSITION( FAULT, color=red, label='error' );
        }
        reqVGRstore = false;
    }
```

Código 51. Estado STORE_WP del HBW.

Tanto el VGR como el HBW se encuentran en estado de reposo IDLE a espera del lanzamiento de una orden de pedido desde el panel de control de pedidos de la nube (Figura 26).

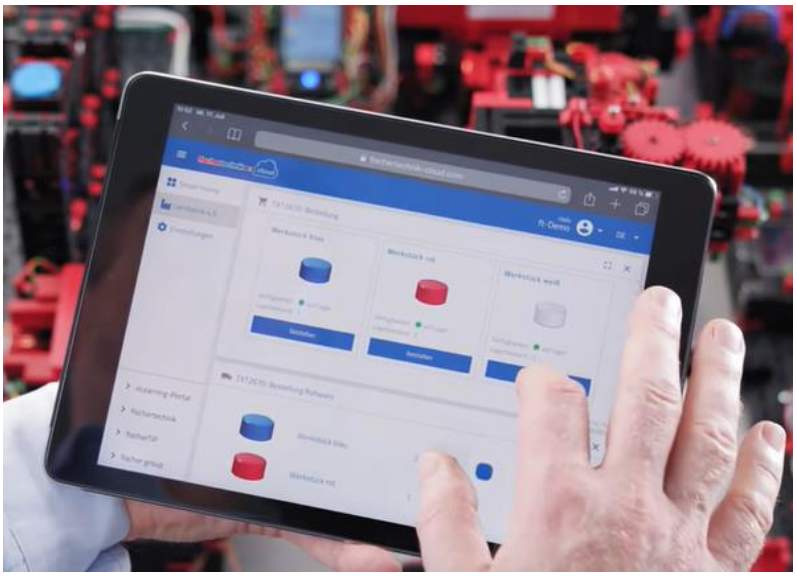


Figura 26. Seleccionar pedido desde la nube de Fischertechnik.

A continuación, se realiza el pedido desde la nube. Una vez que se seleccione el tipo de pieza se mandará un mensaje al VGR mediante el método ***requestOrder()*** con el tipo de pieza. En el método ***requestOrder()*** (Código 52) se guarda el tipo de pieza solicitada en la variable *reqWP_order* y se pone a true la variable *reqOrder*.

```
if (stype == "WHITE")
{
    vgr_.requestOrder(ft::WP_TYPE_WHITE);
} else if(stype == "RED")
{
    vgr_.requestOrder(ft::WP_TYPE_RED);
} else if (stype == "BLUE")
{
    vgr_.requestOrder(ft::WP_TYPE_BLUE);
}
```

```
void requestOrder(TxtWPType_t type) {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestOrder {}",(int)type);
    reqWP_order = ft::TxtWorkpiece("", type, WP_STATE_RAW);
    reqOrder= true;
}
```

Código 52. Llamada a método requestOrder().

Al activarse la variable *reqOrder*, entrará en el *else if* (Código 53), se marcará el estado de pedido como ORDERED. Posteriormente, se mandará un mensaje MQTT de tipo *publishStateOrder* con el estado del pedido y, a continuación, el VGR pasará al estado *FETCH_WP_VGR*.

```
else if (reqOrder)
{
    ord_state.type = reqWP_order.type;
    ord_state.state = ORDERED;
    assert(mqttclient);
    mqttclient->publishStateOrder(ord_state, TIMEOUT_MS_PUBLISH);

    FSM_TRANSITION( FETCH_WP_VGR, color=blue, label='req order' );
    reqOrder = false;
}
```

Código 53. Case IDLE Máquina de Estados del VGR.

Una vez en el estado *FETCH_WP_VGR* (Código 54) se mandará un mensaje al HBW para que inicie el proceso de búsqueda de la pieza en el almacén. El VGR se moverá a la posición de recogida de piezas en la bandeja de salida del almacén (método ***moveFromHBW1()***).

```
case FETCH_WP_VGR:
{
    printState(FETCH_WP_VGR);

    assert(mqttclient);
    reqWP_order.printDebug();
    //Mandar mensaje al HBW para que vaya a buscar pieza solicitada
    mqttclient->publishVGR_Do(VGR_HBW_FETCH_WP, &reqWP_order, TIMEOUT_MS_PUBLISH);
    setTarget("hbw");
    moveFromHBW1(); //Mover a Bandeja exterior del almacén
    //Pasar a estado de espera mientras HBW va a por la pieza.
    FSM_TRANSITION( VGR_WAIT_FETCHED, color=green, label='fetched' );
    break;
}
```

Código 54. Estado FETCH_WP_VGR Máquina de Estados VGR.

El HBW recibirá el mensaje que pondrá la variable *reqVGRfetch* a *true* la pieza. Ahora el HBW pasará al estado *FETCH_WP* y el VGR pasará al estado *VGR_WAIT_FETCHED*.

```
case ft::VGR_HBW_FETCH_WP:
    hbw_.requestVGRfetch(wp);
```

```
void requestVGRfetch(TxtWorkpiece* wp) { //buscar wp
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"reqVGRfetch",0);
    reqVGRwp = wp;
    reqVGRfetch= true;
```

Código 55. Llamada a método requestVGRfetch().

Una vez el HBW está dentro del estado *FETCH_WP* (Código 56) se irá a por la pieza (Figura 27), que será colocada en la bandeja de salida de piezas del almacén; de nuevo, se le comunicará al VGR para que sepa que la pieza ya está disponible mediante el mensaje (*publishHBW_Ack : HBW_FETCHED*).

```
case FETCH_WP:
{
    printState(FETCH_WP);

    if (reqVGRwp && fetch(reqVGRwp->type)) // if (reqVGRwp && fetch(reqVGRwp->type)
    {
        assert(mqttclient);
        mqttclient->publishHBW_Ack(HBW_FETCHED, reqVGRwp, TIMEOUT_MS_PUBLISH);
        FSM_TRANSITION( FETCH_WP_WAIT, color=blue, label='wait req' );
    }
    else
    {
        FSM_TRANSITION( FAULT, color=red, label='error' );
    }
```

Código 56. Estado FETCH_WP Máquina de Estados del HBW.

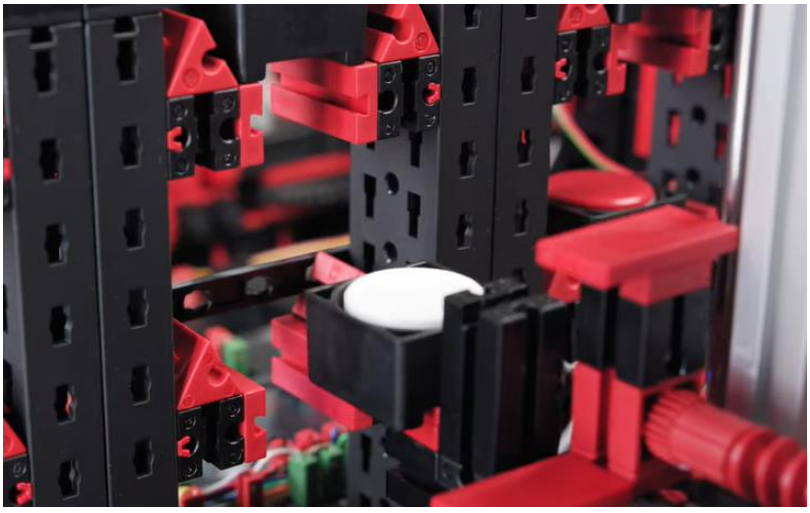


Figura 27. Transelevador recogiendo la pieza solicitada.

```
case ft::HBW_FETCHED:
    vgr_.requestHBWfetched(wp);
```

```
void requestHBWfetched(TxtWorkpiece* wp) { //contenedor o pieza
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestHBWfetched",0);
    reqWP_HBW = wp;
    reqHBWfetched = true;
```

Código 57. Llamada a método requestHBWfetched().

Tras esto el HBW pasará al estado *FETCH_WP_WAIT* (Código 60). El HBW esperará hasta que el VGR publique un mensaje, una vez que haya cogido la pieza, para poder pasar a almacenar el contenedor vacío.

```
case VGR_WAIT_FETCHED:
{
    printState(VGR_WAIT_FETCHED);

    if (reqHBWfetched /*&& isDetectedContainerOut()*/)
    {
        //isDetectedContainerOut();
        moveFromHBW2(); //decender y agarrar pieza

        reqWP_MPO = reqWP_HBW;

        assert(mqttclient);
        //Mandar mensaje al HBW para que devuelva contenedor al almacén
        mqttclient->publishVGR_Do(VGR_HBW_STORECONTAINER, reqWP_MPO, TIMEOUT_MS_PUBLISH);
        assert(reqWP_MPO);
        proStorage.setTimestampNow(reqWP_MPO->tag_uid, OUTSOURCING_INDEX);

        reqHBWfetched = false;
        //pasar a estado mover y soltar pieza en MPO
        FSM_TRANSITION( MOVE_VGR2MPO, color=green, label='transport to MPO' );
    }
```

Código 58. Estado VGR_WAIT_FETCHED de la FSM del VGR.

El VGR una vez que reciba el mensaje se moverá a coger la pieza, agarrándola de la bandeja de salida del almacén (método ***moveFromHBW2()***). Tras coger la pieza, el VGR mandará el mensaje al HBW para que almacene el contenedor vacío (*publishVGR_Do* : *VGR_HBW_STORECONTAINER*). La comunicación pondrá la variable *reqVGRstoreContainer* a *true* que hará que el HBW pase al estado *STORE_CONTAINER*. Una vez aquí se almacenará el contenedor vacío y el HBW pasará al estado IDLE. Una vez en este estado, el robot esperará hasta que vuelva a entrar una nueva pieza por la DPS.

```
case ft::VGR_HBW_STORECONTAINER:
    hbw_.requestVGRstoreContainer(wp);
```

```
void requestVGRstoreContainer(TxtWorkpiece* wp) { //almacenar contenedor
    SPDLOG_LOGGER_TRACE(spdlog::get("console"),"requestVGRstoreContainer",0);
    reqVGRwp = wp;
    reqVGRstoreContainer= true;
```

Código 59. Llamada a método requestVGRstoreContainer().

```
case FETCH_WP_WAIT:
{
    printState(FETCH_WP_WAIT);
    if (reqVGRstoreContainer)
    {
        FSM_TRANSITION( STORE_CONTAINER, color=blue, label='req store\ncontainer' );
        reqVGRstoreContainer = false;
    }
```

Código 60. Estado FETCH_WP_WAIT del HBW.

6 Propuesta de Cambios sobre la programación del Almacén

6.1 Cambios en el proceso de ordenación y búsqueda de pieza

A lo largo de las distintas pruebas sobre la maqueta se ha comprobado que el almacén constituye un cuello de botella en el proceso, debido a la gran cantidad de tiempo invertida por el dispositivo transelevador para almacenar e ir a buscar las piezas. Además, constituye una de las estaciones de la fábrica que mayor flexibilidad ofrece a la hora de proponer formas alternativas en la ordenación y búsqueda de las piezas. Por estos motivos, el almacén ha constituido uno de los principales focos de atención para la propuesta de cambios y mejoras.

6.2 Búsqueda de piezas.

Todos los cambios que se producen en el almacén son guardados en una matriz en la que cada posición refleja el estado de la estantería. Por esta razón todos los cambios requieren de una actualización de la información sobre dicha matriz.

El almacén requiere, por un lado, de control del flujo de información del almacén y, por otro, de un control físico sobre los movimientos de los ejes del transelevador. La actualización del estado de la matriz se realiza mediante un conjunto de métodos de la clase TxtHighBayWarehouseStorage. La clase TxtHighBayWarehouse es la encargada de llevar a cabo los movimientos de los ejes del transelevador. Para llevar a cabo el movimiento del transelevador se dispone de un archivo de calibración "*TxtHighBayWarehouseCalibData*.cpp" en el que se almacenan los valores de referencia absolutos (se pueden ver en Código 63) para llevar a cabo el movimiento de los ejes *x* e

y. Se puede ver una descripción de los métodos de ambas clases en los capítulos dedicados a la descripción de los métodos de cada clase.

```
bool TxtHighBayWarehouseCalibData::saveDefault()//Referencias absolutas por defecto para movimiento del transelevador
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "saveDefault",0);
    //Se indican las coordenadas en valor absoluto del eje X (dirección de la helice)
    //Primera columna a 780, segunda a 1390 y tercera a 1995.
    hbx[0] = 780;
    hbx[1] = 1390;
    hbx[2] = 1995;
    //Se indican cuales son las coordenadas en valor absoluto del eje Y (La altura)
    //Primera altura a 80 (cerca del suelo) segunda a 445 y tercera a 855.
    hby[0] = 80;
    hby[1] = 445;
    hby[2] = 855;
    //Movimiento de los ejes x e y para posicionar transelevador en bandeja de entradada de la cinta.
    conv = EncPos2(20, 720);

    return save();
}
```

Código 63. Método saveDefault() para guardar valores de Calibración del Almacén.

El estado del almacén se guarda en la variable *wp*. La variable *wp* se crea como un vector de dos dimensiones (matriz 3 x 3) en la que se almacenan los objetos de tipo TxtWorkPiece.

En la Figura 29 se han representado las posiciones de la matriz como si se estuviera mirando el almacén desde el exterior. En cada posición de la matriz se almacena el estado de cada estante del almacén. En cada estante puede haber un contenedor vacío o una pieza. La matriz inicialmente tiene todas sus posiciones a cero (esto quiere decir que en esa posición hay un contenedor vacío). Si se almacena una pieza en el contenedor, se guarda un objeto de tipo pieza TxtWorkPiece en esa posición.

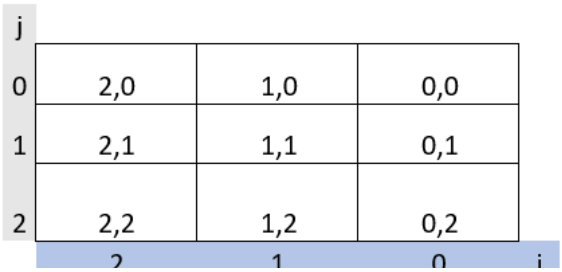


Figura 29. Posiciones de la matriz desde vista exterior del almacén.

Es importante mencionar cómo son los objetos de tipo pieza. Cada objeto TxtWorkPiece tiene tres atributos. Uno de ellos es el tipo de pieza que puede tener los valores *white, red, blue* y *none* (se refiere aquella pieza cuyo color no ha sido identificado como ninguno de los tres colores). Otro de los atributos es el identificador (*tag_uid*), asignado tras la lectura de la pieza en el NFC. El último de los atributos es su estado (*state*), que indica el estado de la pieza y puede ser *RAW* (pieza como materia prima), *PROCESSED* (producto terminado) o REJECTED (pieza defectuosa).

Cuando una pieza entra por la bandeja de la DPS, esta es trasladada al sensor de color y seguidamente al dispositivo NFC para iniciar su identificación. Mediante el método ***nfcDeviceDeleteWriteRawRead()*** de la clase TxtDeliveryPickUpSatation se borran los datos previos de la etiqueta y se escriben los nuevos. Dentro de este método se llama al constructor de clase pasándole como atributos el identificador de la pieza, el color y el

estado. En primer parámetro que se pasa al constructor es el identificados, el segundo el tipo de pieza (en este caso se pasa como parámetro al método tras haber sido identificado el color con el sensor de color) y el estado (en este caso sería de tipo WP_STATE, materia prima).

Cuando la pieza ha pasado todos los procesos, el VGR vuelve a por la pieza a la bandeja correspondiente de la estación de clasificación. Tras esto la vuelve a pasar la pieza por el NFC y se llama al método ***nfcDeviceDeleteWriteProcessedRead()*** que clasifica el estado de la pieza como procesada (PROCESSED)

Por otro lado, la pieza puede ser rechazada en caso de que no se identifique como ninguno de los tres colores (blanco, rojo, azul). En este caso se llama al método ***nfcDeviceDeleteWriteRejectedRead()***.

Tras la identificación de la pieza, el HBW empezará a pasar por los estados necesarios para almacenar las piezas. A continuación, se explicará el proceso de almacenamiento de piezas.

En primer lugar, el transelevador debe alcanzar un contenedor vacío para, una vez que se haya depositado una pieza en él, devolver el contenedor al sitio de donde fue extraído. Para ello se debe comprobar dónde está el contenedor vacío más cercano. La búsqueda de los contenedores vacíos en el almacén se realiza mediante el método ***fetchContainer()*** de la clase TxtHighBayWarehouseStorage (Código 64). A continuación, se expone el código para llevar a cabo la búsqueda de contenedores del código original junto con una imagen (que muestra el orden de búsqueda de los contenedores visto desde el exterior del almacén.

```
bool TxtHighBayWarehouseStorage::fetchContainer()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetchContainer",0);
    nextFetchPos.x = -1; //set invalid pos
    nextFetchPos.y = -1;
    bool found = false;

    for(int i=0;i<3 && !found;i++)
    {
        for(int j=2;j>=0 && !found;j--)
        {
            StoragePos2 p;
            p.x = i; p.y = j;
            if (wp[i][j] == 0)
            {
                SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "cont -> nextFetchPos {} {}",p.x, p.y);
                nextFetchPos = p;
                found = true;
            }
        }
    if (isValidPos(nextFetchPos))
        {
            SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextFetchPos cont ",0);
            wp[nextFetchPos.x][nextFetchPos.y] = 0;
            Notify();
            print();
            return true;
        }
    return false;
```

Código 64. Método fetchContainer() de la clase TxtHighBayWarehouseStorage

.

En el método ***fetchContainer()*** de la clase TxtHighBayWarehouseStrorage (Código 64) se determina la posición a la que debe dirigirse el transelevador, posición que se guarda en la variable *nextFetchPos*. Antes de encontrar el lugar en el que ir a buscar el contenedor, se crea la variable *p* de tipo *StoragePos2* que es una estructura con dos atributos *x* e *y* que guardan la posición (i, j) de la matriz de piezas. Cuando al recorrer la matriz *wp* se encuentra un cero, significa que en esa posición del almacén no hay una pieza almacenada, por lo que ese contenedor está libre para ser extraído. Cuando esto ocurre se asigna la posición *p* a la variable *nextFetchPos*.

El método anterior se llama dentro del método ***fetchContainer()*** de la clase TxtHighBayWarehouse (Código 65), que es la que inicia el movimiento físico del transelevador, cuando en la primera sentencia *if* se comprueba la respuesta del método (devuelve true cuando se ha encontrado la posición para ir a buscar el contenedor). Se puede comprobar que como se está llamando a un método de otra clase, se debe hacer una instancia del objeto de la clase TxtHighBayWaregouseStorage dentro de la clase TxtHighBayWarehouse. Esto se realiza dentro del archivo de cabecera.

Para llamar a cualquier método de otra clase se hará mediante la sentencia de tipo: ***nombreObjeto.metodo()***.

9	6	3
8	5	2
7	4	1

Figura 29. Orden de búsqueda de contenedores del código original visto desde el exterior.

```
bool TxtHighBayWarehouse::fetchContainer() //Extraer un contenedor vacio
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetchContainer", 0);
    setActStatus(true, SM_BUSY);
    if (storage.fetchContainer())
    {
        StoragePos2 p = storage.getNextFetchPos();
        if (p.x<0 || p.y<0) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        bool r = getCR(p.x, p.y);
        if (!r) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        r = putConv(true);
        if (!r) {
            setActStatus(false, SM_ERROR);
            return false;
        }
        setActStatus(false, SM_READY);
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 65. Método fetchContainer() de la clase TxtHighBayWarehouse.

Cada vez que hagamos referencia al objeto de la clase TxtHighBayWaregouseStorage lo haremos mediante el nombre *storage*. Una vez que el método ***fetchContainer()*** devuelve

true comienzan los movimientos del transelevador mediante los métodos ***getCr()*** y ***putConv()***.

Antes debemos acceder a las coordenadas (i, j) almacenadas en la variable *nextFetchPos*. Como es un atributo de otra clase, se debe acceder al contenido de la variable mediante el método ***getNextFetchPos()***.

```
StoragePos2 getNextFetchPos() { return nextFetchPos; }
```

Una vez que se tienen las coordenadas, se le pasan al método ***getCR()*** (Código 66) como parámetros. Este método va a esas coordenadas del almacén y coge la pieza.

```
bool TxtHighBayWarehouse::getCR(int iCol, int iRow) // coger de una posicion
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getCR idx:{} {}", iCol, iRow);
    EncPos2 p2 = moveCR(iCol,iRow); //obtiene pulsos de encoder necesarios y mueve los ejes a esa posición
    axisZ.moveS2(); // eje z dentro
    bool r = axisY.moveAbs(p2.y - ydelta); // eje y bajar 40
    axisZ.moveS1(); // eje z fuera
    return r;
}
```

Código 66. Método getCR() de la clase TxtHighBayWarehouse.

Posteriormente, mediante ***putConv()*** (Código 67) el transelevador deposita la pieza en la bandeja interior y saca la pieza fuera mediante la llamada al método ***moveOut()*** de la clase TxtConveyorBelt. El método ***getCR()*** y ***putConv()*** devuelven *true* una vez que se han realizado los movimientos con éxito.

```
bool TxtHighBayWarehouse::putConv(bool stop) // Depositar pieza en cinta y sacar fuera
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "putConv",0);
    EncPos2 p2 = moveConv(); // mover a cinta transportadora
    axisZ.moveS2(); // eje z dentro
    bool r = axisY.moveAbs(p2.y + ydelta); // eje y subir 40
    convBelt.moveOut(); // cinta transportadora out
    if (!stop)
    {
        axisZ.moveS1(); //eje z fuera si stop es true
    }
    return r;
}
```

Código 67. Método putConv() de la clase TxtHighBayWarehouse.

Seguidamente, una vez que el contenedor ya está en la bandeja exterior se utiliza el método ***store()*** de la clase TxtHighBayWarehouse (Código 68) para iniciar el almacenamiento de la pieza. En el *if* se comprueba la respuesta del método ***store()*** de la clase TxtHighBayWarehouseStorage (Código 69) para, una vez que se ha validado *nextFechPos* y se ha actualizado la matriz de piezas con el nuevo objeto pieza, iniciar los movimientos de los ejes mediante la llamada a los métodos ***getConv()*** y ***putCR()***. Posteriormente, se actualiza el nuevo estado del almacén (método ***saveStorageState()***)

```
bool TxtHighBayWarehouse::store(TxtWorkpiece wp)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "store {}", wp.type);
    setActStatus(true, SM_BUSY);
    if (storage.store(wp)) // validar posición a almacenar y actualizar matriz
    {
        StoragePos2 p = storage.getNextStorePos(); // Obtener posición nextFetchPos para almacenar
        if (p.x<0 || p.y<0) return false;
        bool r = getConv(true); // Mover translevador a cinta, cinta transportadora in
                                //y coger contenedor de bandeja interior
        if (!r) return false;
        r = putCR(p.x,p.y); // Mover a estanteria y depositar contenedor
        if (!r) return false;
        setActStatus(false, SM_READY);
        storage.saveStorageState(); // actualizar estado del almacén (nube)
        moveCR(1,1); // PRUEBA 20.05 mover a posición (1,1) como posición de referencia
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 68. Método store() de la Clase TxtHighBayWarehouse.

```
bool TxtHighBayWarehouseStorage::store(TxtWorkpiece _wp) //Actualizar estado almacén tras almacenamiento
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "store wp:{} {} {}",_wp.tag_uid,_wp.type,_wp.state);
    //nextStorePos.x = -1; //set invalid pos
    //nextStorePos.y = -1;
    if (_wp.type == WP_TYPE_NONE) // pieza no identificada en ninguno de los tres colores, devolver false para rechazar pieza
    {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "NONE -> return false",0);
        return false;

    }
    if (isValidPos(nextFetchPos)) //Posición dentro de las 9 posibles posiciones
        {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "OK -> nextStorePos type {} ",_wp.type);
        wp[nextFetchPos.x][nextFetchPos.y] = new TxtWorkpiece(_wp); //se actualiza estado del almacén con nuevo objeto
        Notify();
        print();
        return true;
    }
    return false;
}
```

Código 69. Método store() de la clase TxtHighBayWarehouseStorage.

Una vez almacenada, la pieza se pasa a los métodos encargados de ir a buscar la pieza seleccionada. La recogida de piezas se realiza mediante el método ***fetch()*** (Código 70).

```
bool TxtHighBayWarehouse::fetch(TxtWPType_t t) //Extraer un contenedor lleno
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "fetch {}", t);
    setActStatus(true, SM_BUSY);
    if (storage.fetch(t))
    {
        StoragePos2 p = storage.getNextFetchPos(); //obtener posición a la que dirigir transelevador
        if (p.x<0 || p.y<0) return false;
        bool r = getCR(p.x, p.y); //ir a coger pieza del almacén
        if (!r) return false;
        r = putConv(true); //Depositar pieza en bandeja interior y extraer
        if (!r) return false;
        setActStatus(false, SM_READY);
        storage.saveStorageState();
        return true;
    }
    setActStatus(false, SM_ERROR);
    return false;
}
```

Código 70. Método fetch() de la Clase TxtHighBayWarehouseStorage.

En el primer *if* de este método se llama al método ***fetch()*** de la Clase TxtHighBayWarehouseStorage para obtener la posición de la pieza que ha seleccionado el usuario. En el método ***fetch()*** de la Clase TxtHighBayWarehouseStorage cada posición de la matriz puede no almacenar una pieza; en tal caso, el valor que se almacena en esa posición es un puntero NULL y el caso en el que exista una pieza almacenada, se almacena en esa posición de la matriz el puntero a un objeto de tipo TxtWorkPiece.

En el método original se recorre la matriz siguiendo el mismo orden que en el método ***fetchContainer()*** (Código 64).

Pongamos ahora que se desea seguir una gestión LIFO del almacén para buscar las piezas de tal forma que cuando se haga el pedido se vaya a buscar la última pieza del color seleccionado que haya sido almacena. Lo que se debe hacer es recorrer la matriz en el orden inverso al realizado en el método ***fetchContainer()***.

Para conseguir que se recorra la matriz en el orden inverso se debe cambiar el bucle *for* para recorrer la matriz *wp* [*i*][*j*] en el orden contrario al método ***fetchContainer()***. Se empezará buscando la pieza por la última fila (j=0) en lugar de por la primera (Código 71).

```
for(int i=2;i>=0 && !found;i--)
{
    for(int j=0;j<3 && !found;j++)
    {
        StoragePos2 p;
        p.x = i; p.y = j;
        if (wp[i][j] == NULL)
            continue;
        if (wp[i][j]->type == t)
        {
            SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "t {} -> nextFetchPos {} {}",t, p.x, p.y);
            nextFetchPos = p;
            found = true;
        }
    }
}
```

Código 71. Bucle for del método fetch() de la Clase TxtHighBayWarehouseStorage.

6.3 Ordenar las piezas por diagonales

Podemos encontrar muchas formas de recorrer la matriz para ir almacenando las piezas en el contendor. Otra forma alternativa de ordenación y búsqueda de piezas en el almacén puede ser siguiendo diagonales (Figura 30)

Para ello se han creado dos vectores, uno almacenará el orden que siguen las *x* en la diagonal, y el otro el orden de las *y*. Con un único bucle *for* se recorrerán ambos vectores simultáneamente, accediéndose a las posiciones de la matriz *wp* en forma de diagonales.

En el método ***fetchContainer()*** incluimos el siguiente código (Código 72). Este código recorre el almacén como se observa en la Figura 30

```
std::vector<int> v1 (9);
v1= {0,1,0,2,1,0,2,1,2};
std::vector<int> v2 (9);
v2={0,0,1,0,1,2,1,2,2};

for(int i=0;i<9 && !found;i++){
    StoragePos2 p;
    p.x = v1[i]; p.y =v2[i];
    if (wp[v1[i]][v2[i]] == 0)
    {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "cont -> nextFetchPos {} {}",p.x, p.y);
        nextFetchPos = p;
        found = true;
    }

}
```

Código 72. Método fetchContainer() modificado para buscar contenedores por diagonales.

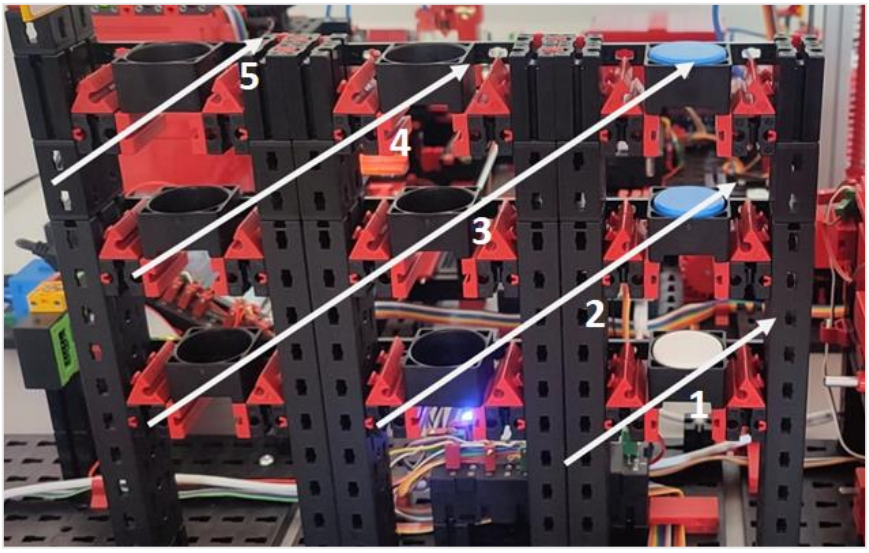


Figura 30. Recorrido del almacén por diagonales.

El método ***fecthContainer()*** determina el orden en el quedan almacenadas las piezas. El método ***fetch()*** determina el orden en el que se buscarán las piezas cuando se haga el pedido.

Si se desean buscar las piezas siguiendo diagonales se crearán en el método ***fetch()*** los mismos vectores del método ***fetchContainer()***. Estos vectores se podrán recorrer ahora en el mismo orden, y por lo tanto se seguirá un método FIFO (primera pieza que se almacena, primera pieza que se va a buscar) (Figura 30) o se podrán recorrer en sentido

contrario, por lo que se seguirá un método LIFO (última pieza que se almacena, primera pieza que se va buscar) (Figura 31)

```
std::vector<int> v1 (9);
v1= {0,1,0,2,1,0,2,1,2};
std::vector<int> v2 (9);
v2={0,0,1,0,1,2,1,2,2};

for(int i=9;i>=0 && !found;i--){ //LIFO (diagonal)
    StoragePos2 p;
    p.x = v1[i]; p.y =v2[i];
    if (wp[v1[i],v2[i]] == NULL)
    {
        continue;
    }
    if (wp[v1[i]][v2[i]]->type == t)
    {
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "cont -> nextFetchPos {} {}",p.x, p.y);
        nextFetchPos = p;
        found = true;

    }
}
```

Código 73. Método fetch() modificado para recorrer diagonales en orden inverso

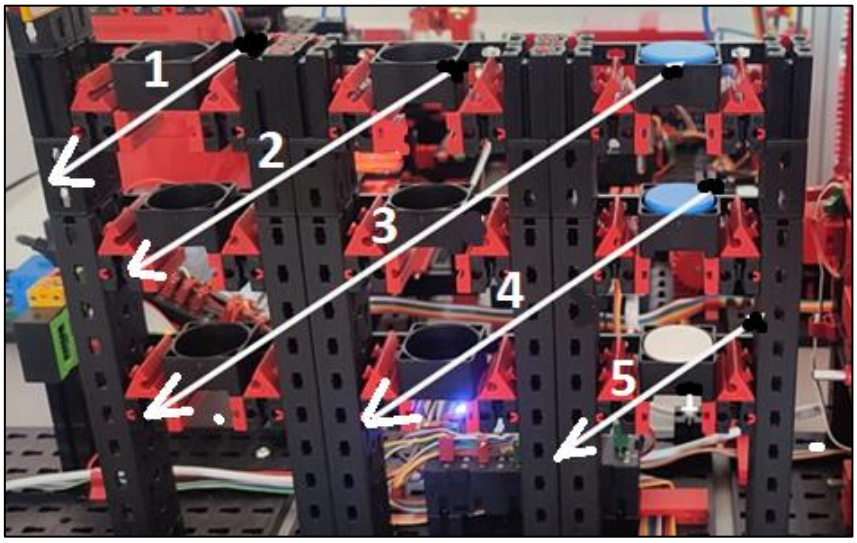


Figura 31. Recorrido de almacén por diagonales en el método fecth() según LIFO.

6.4 Ordenar las piezas por colores en el almacén.

El objetivo es el de almacenar las piezas por colores, de forma que queden todas las de un mismo color en una fila. Igual que en los cambios anteriores las modificaciones se realizan dentro de la clase TxtHighBayWarehouseStorage.

En el código propuesto se pretende ordenar las piezas en el almacén en función del color, de forma que cada fila almacene piezas de un mismo color. Para ello se hace una pequeña modificación sobre el código original (Código 74).

```
bool found = false;
StoragePos2 p;
switch(_wp.type)
{
case WP_TYPE_WHITE: p.y=2; break;
case WP_TYPE_BLUE: p.y=1; break;
case WP_TYPE_RED: p.y=0; break;
}
for (int i=0;i<3&& !found;i++){
    p.x=i;
    if(wp[i][p.y]==0){
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "0 -> nextStorePos {} {}",p.x, p.y);
        nextFetchPos = p;
        found = true;
    }
}
```

Código 74. Ordenar las piezas por colores en método store().

El método ***store()*** recibe como parámetro un objeto de tipo WorkPiece y devuelve *true* o *false* en función de si se ha encontrado una posición libre para almacenar ese tipo de pieza. Dentro del método ***store()*** se va a buscar la posición en la que se desea almacenar la pieza. Para ello se crea una variable booleana que se pone a *true* en caso de que se cumpla que esa es la posición en la que se quiere almacenar ese tipo de pieza y, además, hay un hueco libre en esa posición.

Para ello, dependiendo del tipo de pieza, se hace un *switch* para asignar en función del color la fila en la que se desea almacenar ese tipo de pieza. Las blancas quedarán almacenadas en la fila de arriba, las rojas en la fila intermedia y las azules en la fila de abajo.

6.5 Cambiar posición de referencia del Almacén.

Durante la realización de las prácticas llevadas a cabo para probar los cambios nos dimos cuenta de que el transelevador del almacén realizaba movimientos innecesarios. Cuando el HBW está en estado de reposo IDLE acude a la posición de referencia, la cual se alcanza cuando los ejes hacen contacto con los interruptores (Figura 32).

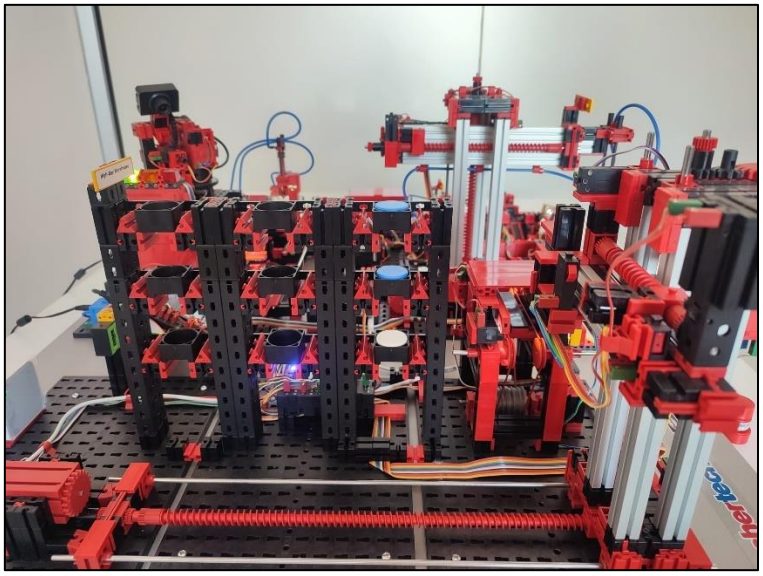


Figura 32. Transelevador en posición de referencia.

Cuando llega una pieza al almacén el transelevador la coge y la deposita en el estante correspondiente de la bahía para almacenarla hasta que se solicite. Tras almacenarse el transelevador se dirige a la posición de referencia. Desde esta posición de reposo, el transelevador puede tener que ir a buscar una pieza solicitada o puede tener que dirigirse a la bandeja interior de piezas del almacén para almacenar una nueva pieza.

Cuando entra una nueva pieza por la DPS, el VGR necesita un tiempo para identificarla y llevarla hasta la bandeja exterior del almacén. Cuando se solicita una pieza el tiempo que tarda en ir a buscarla es el tiempo que tarda desde la posición de referencia hasta la bandeja correspondiente.

Para minimizar tiempos en el proceso de almacenaje se ha modificado la posición de referencia en la que espera el transelevador durante el estado de reposo IDLE. Si se modifica la posición de referencia para ubicar al transelevador en una posición central (posición (1,1)) estaremos reduciendo los tiempos de recogida de piezas. En el Código 75 se puede ver la modificación relativa al cambio de referencia a la posición central mediante el método ***moveCR(1,1)***.

Por un lado, si el transelevador debe dirigirse a la bandeja interior de piezas del HBW para almacenar una nueva pieza, el VGR habrá avisado al HBW de la entrada de una nueva pieza. Por lo tanto, ambas operaciones podrán transcurrir simultáneamente sin incurrir en tiempos de espera por parte del HBW para posicionarse en la bandeja interior de piezas. En el segundo caso, se está reduciendo el tiempo pues se puede observar, al comparar la Figura 33 y Figura 34, que las distancias a recorrer con la posición de referencia central son menores.

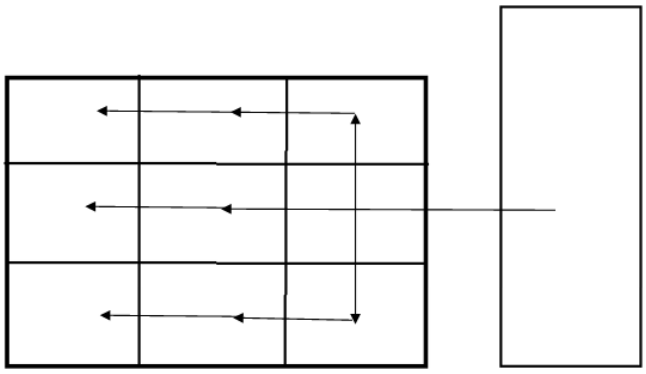


Figura 34. Posición de referencia original.

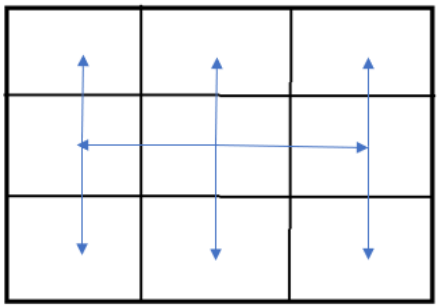


Figura 33. Posición de referencia central (1,1)

```
case IDLE: //Estado de reposo del HBW
{
    printEntryState(IDLE);
    setSpeed(512); // SetSpeed recibe un valor AQUI
    //En el estado de reposo del almacén llevamos transelevador a pos (1,1)
    moveCR(1,1); //Código original: moveRef()
    setActStatus(false, SM_READY);
    publishStorage();
    break;
}
```

Código 75. Modificación para cambiar posición de referencia del transelevador del HBW.

6.6 Detección de contenedor en las bandejas del Almacén.

Tras comprobar que los fototransistores que comunican la entrada y salida de piezas en el almacén no desempeñan ninguna función, se ha considerado la posibilidad de comprobar la presencia de un contenedor antes de depositar una pieza con el objetivo de poder detectar un fallo de este tipo. Además, estos cambios son de utilidad para aprender a configurar los sensores y obtener su información, y realizar cambios sobre la FSM.

Para ello, se han planteado una serie de cambios con el objetivo de dar uso a los sensores de luz (fototransistor interior y exterior) ubicados en la cinta transportadora de la bandeja de entrada y salida de piezas en el almacén (Entradas I1 e I4 del TXT 2). Estos sensores están conectados al TXT pero no están programados para desempeñar ninguna función y podrían ser de utilidad para detectar fallos. Un fallo podría ser el caso en el que usuario de la fábrica no ha ubicado los contenedores en sus posiciones y, por lo tanto, cuando el VGR vaya a depositar la pieza no haya un contenedor en la bandeja exterior. En la Figura 35 se puede ver el fototransistor de la bandeja exterior del almacén rodeado de un círculo negro.

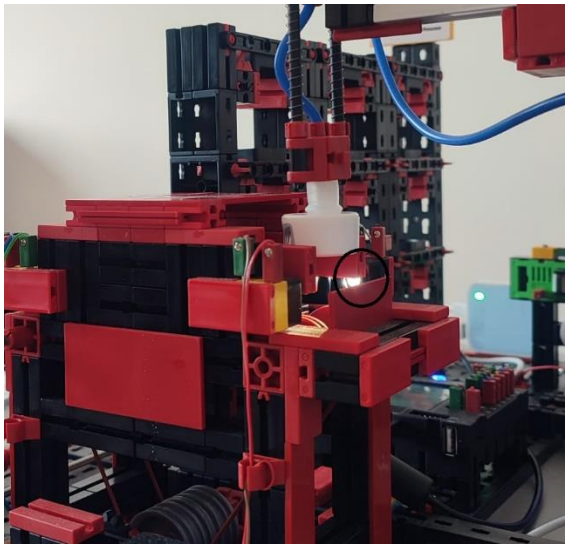


Figura 35. Fototransistor exterior del HBW.

Pongamos el caso en el que el transelevador va a por un contenedor vacío para almacenar una pieza y no hay un contenedor. Como esto no se detecta de ninguna forma, cuando el VGR vaya a soltar la pieza (Figura 35) sobre la bandeja exterior no habrá ningún contenedor. Este caso podría producirse algún desperfecto sobre el sistema. Por este motivo es conveniente tratar este posible error.

Para detectar este fallo se puede hacer uso del fototransistor exterior. Nuestro objetivo es que el fototransistor exterior detecte un contenedor, para que en el caso en que no haya ninguno se produzca una parada del sistema. Para ello hay dos clases que ya están implementadas para controlar las cintas transportadoras de la fábrica: la clase TxtConveyorBelt y TxtConveyorBeltLightBarriers. Dentro de estas clases se controla el la velocidad y el movimiento y parada de los motores.

En primer lugar, se crea el método ***configInput()*** (Código 76) para configurar los fototransistores (Ver el apartado Configuración de los dispositivos.)

```
void TxtConveyorBeltLightBarriers::configInputs(){
    //Config Inputs
    assert (pT->pTArea); //detectar errores en Debug

    pT->pTArea->ftX1config.uni[chL1].mode = MODE_R; // Digital Switch with PullUp resistor
    pT->pTArea->ftX1config.uni[chL1].digital = 1;

    pT->pTArea->ftX1config.uni[chL2].mode = MODE_R; // Digital Switch with PullUp resistor
    pT->pTArea->ftX1config.uni[chL2].digital = 1;

    // actualizar estado
    pT->pTArea->ftX1state.config_id ++;

}
```

Código 76. Método configInputs() creado para detección de contenedor.

En segundo lugar, creamos dos métodos (Código 77) para controlar el fototransistor interior y exterior, y poder detectar el contenedor tanto en la bandeja interna como en la bandeja externa del almacén. El método ***isDetectedContainerOut()*** detecta si hay un contenedor en el exterior del almacén y el método ***isDetectedContainerIn()*** en el interior. Para detectar el fallo solo utilizaremos el primer método.

```
bool TxtConveyorBeltLightBarriers::isDetectedContainerOut() {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "isDetectedContainer",0);
    assert(pT->pTArea); //detectar errores en Debug
    return (pT->pTArea->ftX1in.uni[chL1] != 1);

}
bool TxtConveyorBeltLightBarriers::isDetectedContainerIn() {
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "isDetectedContainer",0);
    assert(pT->pTArea); //detectar errores en Debug
    return (pT->pTArea->ftX1in.uni[chL2] != 1);

}
```

Código 77. Método isDetectedContaionerOut() y método isDetectedContainerIn().

En la inicialización del constructor de la clase TxtHighBayWarehouse (Código 78) se comprueba que ya está inicializado el objeto convBelt con sus parámetros correspondientes para poder controlar los fototransistores y que después de la calibración se debe llamar al método ***configInputs()***. El primer parámetro es el puntero *pT*, el segundo parámetro es el canal del motor *chM*, el tercero es el fototransistor exterior *chL1* y el cuarto el fototransistor interior *chL2*)

```
TxtHighBayWarehouse::TxtHighBayWarehouse(TxtTransfer* pT, ft::TxtMqttFactoryClient* mqttclient)
    : TxtSimulationModel(pT, mqttclient),
    currentState(__NO_STATE), newState(__NO_STATE),
    calibPos(HBWCALIB_CV),
    axisX("HBW_X", pT, 1, 4, 2050),
    axisY("HBW_Y", pT, 3, 7, 1050),
    axisZ("HBW_Z", pT, 2, 5, 6),
    convBelt(pT, 0, 0, 3),
    reqQuit(false),
    reqVGRwp(0), reqVGRfetchContainer(false), reqVGRstore(false),
    reqVGRfetch(false), reqVGRstoreContainer(false), reqVGRcalib(false), reqVGRresetStorage(false),
    joyData(), reqJoyData(false),
    obs_hbw(0), obs_storage(0)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "TxtHighBayWarehouse",0);
    if (!calibData.existCalibFilename()) calibData.saveDefault();
    calibData.load();
    convBelt.configInputs(); // prueba fototransistores
}
```

Código 78. Inicialización del Constructor del Almacacén.

Finalmente, quedaría configurar el estado *STORE_WP* de la máquina de estados del HBW (Código 79). Para ello se llamaría al método ***isDetectedContainerOut()*** para que en el momento en el que el HBW va a pasar a almacenar la pieza se compruebe que se detecta un contenedor antes de iniciar el almacenamiento. En el caso en el que no se detecte el contenedor la fábrica lanzaría un error pasando a un estado FAULT de parada, con la indicación de fallo mediante un sonido y las luces rojas parpadeantes de la estación de vigilancia.

```
if (reqVGRstore  && convBelt.isDetectedContainerOut()) //PRUEBA 21.06
{
    if (reqVGRwp && store(*reqVGRwp))
    {
        FSM_TRANSITION( IDLE, color=green, label='workpiece\nstored' );
    }
else
{

        FSM_TRANSITION( FAULT, color=red, label='error' );
}
reqVGRstore = false;
}
```

Código 79. Estado STORE_WP del HBW.

7 Otros cambios propuestos sobre otras estaciones.

7.1 Propuesta de cambio para engañar al sensor de color de la estación (SLD).

En el siguiente cambio se ha "engañado" al sistema con la detección del color por parte del sensor de color en la estación de clasificación de piezas. Para ello se ha realizado una pequeña modificación en el método ***getDetectedColor()*** de la clase TxtSortingLine. Este método devuelve el color detectado *TxtWpType_t* en función de los valores detectados por el sensor de luz y los valores almacenados en el vector *CalibData.color_th*.

Cuando en realidad la pieza es de tipo blanco y debería identificarse como blanca, se ha modificado el tipo para que se identifique como roja (WP_TYPE_RED).

```
ft::TxtWPType_t TxtSortingLine::getDetectedColor()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getDetectedColor", 0);
    if ((detectedColorValue >= 200)&&(detectedColorValue < calibData.color_th[0]))
    {
        return WP_TYPE_RED;// WP_TYPE_WHITE PRUEBA 19.04
        //La pieza se reconoce como roja y se produce su eyeccion en el lugar de las rojas
    }
    else if ((detectedColorValue >= calibData.color_th[0])&&(detectedColorValue < calibData.color_th[1]))
    {
        return WP_TYPE_RED;
    }
    else if ((detectedColorValue >= calibData.color_th[1])&&(detectedColorValue < 2000))
    {
        return WP_TYPE_BLUE;
    }
    return WP_TYPE_NONE;
}
```

Código 80. Método getDetectedColor() modificado.

Cuando se produce la transición al estado *CHECK_COUNT* en la máquina de estados de la estación de clasificación, se cuentan el número de impulsos necesarios para expulsar la pieza en el lugar adecuado, mediante el *encoder* que registra la rotación de la rueda dentada que mueve la cinta. Comparando este dato con los valores de la calibración se producirá la eyección en la bandeja correspondiente. En este caso, la pieza se identifica como roja cuando es blanca. Por lo tanto, la eyección se realiza sobre la bandeja de las rojas.

```
case CHECK_COUNT:
{
    printState(CHECK_COUNT);
    switch (getDetectedColor())
    {
    case WP_TYPE_WHITE:
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "WHITE",0);
        if (u16Counter >= calibData.count_white)
        {
            FSM_TRANSITION( EJECTION_WHITE, color=blue, label='counter\nw' );
        }
        break;
    case WP_TYPE_RED:
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "RED",0);
        if (u16Counter >= calibData.count_red)
        {
            FSM_TRANSITION( EJECTION_RED, color=blue, label='counter\nr' );
        }
        break;
    case WP_TYPE_BLUE:
        SPDLOG_LOGGER_DEBUG(spdlog::get("console"), "BLUE",0);
        if (u16Counter >= calibData.count_blue)
        {
            FSM_TRANSITION( EJECTION_BLUE, color=blue, label='counter\nb' );
        }
        break;
```

Código 81. Estado CHECK_COUNT de la FSM de la Sorting Line.

7.2 Serrados distintos en función del tipo de pieza (MPO)

El objetivo es dar distintas órdenes a la sierra, para simular un proceso de fresado diferente, en función del tipo de pieza. Los cambios se han implementado mediante la creación de un nuevo método ***sawMoreTimes()*** en la clase TxtMultiProcessingStation.

El método ***sawMoreTimes()*** recibe el tipo de pieza (color) y llama al método ***setSawRight()*** o ***setSawLeft()*** en distinto orden en función del color. Si la pieza es blanca girar a la derecha, luego parar, girar a la izquierda y volver a parar. Si la pieza es azul al contrario y si es roja girar a la izquierda, parar y volver a girar a la izquierda.

```
void TxtMultiProcessingStation::sawMoreTimes(TxtWPType_t t){

    if(t==WP_TYPE_WHITE) {
        setSawRight();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
        setSawLeft();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
    if (t==WP_TYPE_BLUE) {
        setSawLeft();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
        setSawRight();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
    if (t==WP_TYPE_RED){
        setSawLeft();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
        setSawLeft();
            std::this_thread::sleep_for(std::chrono::milliseconds(2500));
        setSawOff();
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}
```

Código 82. Método sawMoreTimes() de la Clase TxtMultiProcessingStation.

La llamada al método ***sawMoreTimes()*** se realiza desde la máquina de estados de la MPO, cuando pasa al estado *TABLE_SAW*. En la variable *reqVGRwp* se almacena la pieza que está siendo procesada. Recordemos que un objeto pieza tiene tres atributos, y como nos interesa el tipo, accedemos a este mediante `reqVGRwp->type` que se pasa como parámetro al método.

```
case TABLE_SAW:
{
    printState(TABLE_SAW);
    axisRotTable.moveS2();
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    sawMoreTimes(reqVGRwp->type);// PRUEBA 19.04: Serrar x numero de veces en funcion del tipo de pieza
    FSM_TRANSITION( TABLE_BELT, color=blue, label='processed' );
    break;
}
```

Código 83. Estado SAW de la FSM de la MPO.

7.3 Envío de pieza defectuosa a bandeja de piezas defectuosas (VGR).

En el código original, cuando no se detecta ninguno de los tres colores principales en una pieza, la pieza se identifica como WP_TYPE_NONE. En este caso, las piezas son enviadas a la bandeja de piezas defectuosas ubicada en la DPS.

La detección de piezas se efectúa a través del sensor de color de la DPS. Como no se tienen piezas de otro color que las que vienen con el modelo de la fábrica se ha realizado

un cambio para comprobar cómo de desecharía una pieza. En este caso, como ejemplo, se ha simulado el caso en el que se identifique una pieza blanca como defectuosa.

Para ello, en la máquina de estados del VGR, en el estado siguiente a identificar la pieza en el NFC, justo antes de iniciar el desplazamiento del robot hacia el almacén (*STORE_WP_VGR*) se comprueba si la pieza es blanca para mover la pieza hacia la bandeja de piezas defectuosas mediante la llamada al método ***moveWrongRelease().***

```
if (dps.getLastColor() == WP_TYPE_WHITE) //pieza defectuosa. Se puede probar con otro color PRUEBA 14.06.22
{
    moveWrongRelease(); // llevar a bandeja para piezas defectuosas
    FSM_TRANSITION( FAULT, color=red, label='wrong color' );
    break;
}
```

Código 84. Llamada al método moveWrongRelease() en la FSM del VGR.

CAPÍTULO 4 Estudio Económico

1 Introducción

El presente capítulo pretende poner en valor el trabajo realizado para llevar a cabo este proyecto. Se dará un valor aproximado del trabajo desempeñado por dos alumnos y dos tutores del proyecto junto a los costes asociados. Es importante subrayar que esta estimación considera el esfuerzo hecho por el equipo, no tiene en cuenta el beneficio económico facturable o de ahorro que se daría en una empresa privada.

En este estudio se tendrá en cuenta los recursos utilizados (ordenador, electricidad, acceso a internet, etc…) para poder desarrollar el proyecto y las horas dedicadas a cada una de las etapas del proyecto. También se considerarán elementos ficticios o virtuales, como los sueldos que los trabajadores tendrían en una empresa en caso de trabajar en una actividad como esta.

De esta manera y solo como ejercicio en este estudio económico el alumno estará considerado como portador de varios roles o papeles con funciones diferentes que tendrían distintas personas encargadas de un proyecto. Primero se definirá en qué consisten esos roles y cómo se divide el trabajo en las distintas etapas que forman el proyecto, por último, se describirá qué roles toman parte en cada una de las etapas. En último lugar se establecerán los salarios ficticios de cada rol y la carga de trabajo que ha supuesto cada una de las tareas o actividades para cada uno de los roles. De esta forma podremos calcular el coste de cada etapa y finalmente el coste total.

2 Profesionales que intervienen en el proyecto

Se considera el caso en el que una supuesta empresa industrial ha adquirido un modelo de fábrica de aprendizaje de Industria 4.0 con el objetivo de formar a los ingenieros de nueva incorporación. La empresa asigna distintos proyectos a grupos de dos o tres personas con el objetivo de aplicar nuevos métodos sobre la fábrica que puedan ser probados antes de poder aplicarse a los procesos reales.

Se ha llevado a cabo un estudio de tiempos en los procesos de fabricación y se ha llegado a la conclusión de que el proceso de almacenaje constituye un cuello de botella. El almacén no se encuentra completamente automatizado y se producen muchos movimientos de material innecesarios. El jefe de la sección de logística ha planteado un proyecto para la automatización completa de la bahía de almacenamiento y la implementación de modificaciones que minimicen los tiempos de almacenaje y búsqueda de material. Para ello se deberán proponer cambios e implantarse sobre la fábrica de aprendizaje.

Partiendo de esta situación, los profesionales que formarán parte de esta labor serán los siguientes: un director del proyecto, un ingeniero senior y un ingeniero junior (Figura).

El director del proyecto es el responsable de dirigir y coordinar las actividades de los proyectos a su cargo. Tiene entre otras tareas: la planificación del proyecto (definiendo alcance, plazos de entrega y asegurando su cumplimiento), comprensión de su viabilidad basándose en su experiencia previa, el control económico del proyecto y la comunicación con el jefe de logística. Deberá mantener reuniones de forma periódica con el jefe de logística para atender sus necesidades y reportar el estado del proyecto, así como de negociar su lanzamiento y ajustar el alcance en caso de que fuera necesario. [1]

El Ingeniero senior, cuenta con una notoria experiencia en el área de programación e implantación 4.0, ha participado en proyectos similares, se encarga de la comunicación entre la instalación y el programa cliente, la revisión del código, la innovación materializada en puntos de mejora y supervisión del equipo, así como del buen estado de los entregables y su corrección, así como del contenido a presentar y la redacción de los apartados más exigentes.

El ingeniero junior tiene como labora cumplir con las tareas que el ingeniero senior considere oportunas entre las que estarán la revisión, estudio e inclusión de comentarios sobre el código, implementación de puntos de mejora, correcciones en la compilación del código y redacción de la mayor parte de los entregables, así como la impresión y entrega de los documentos.

Es importante tener en cuenta que suponemos que estos trabajadores no dedican el total de su jornada laboral a este proyecto en cuestión, en lugar de eso, deberán compaginarlo con otros proyectos y actividades dentro de la empresa. De esta forma estamos reflejando en este modelo las circunstancias reales que hemos encontrado, en la que el alumno debe compaginar el desarrollo de este proyecto con otras asignaturas, trabajo u otras cuestiones.

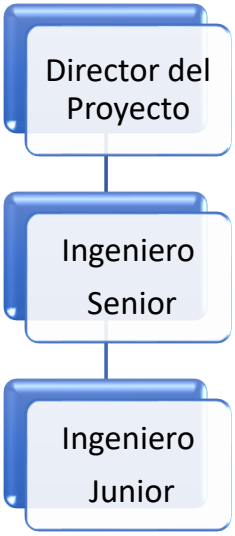


Figura 36. Organigrama del Proyecto.

3 Definición de las etapas del proyecto

El proyecto en el que trabajamos está orientado a resolver mediante ingeniería Inversa el código con el que la instalación automatizada del almacén de la fábrica de aprendizaje se encuentra programada, así como la materialización de este ejercicio en cambios y optimizaciones sobre el proceso real. Para finalizar se deberá entregar al cliente una memoria detallando en qué consisten las funciones y objetos del código y como se han llevado a cabo dichos cambios. Este informe se entrega y presenta al jefe de la sección de logística de la fábrica.

La primera etapa del proyecto será la puesta en marcha. Aquí el director presentará el nuevo proyecto a su equipo, estableciendo cual es el alcance, el reparto de tareas, documentación a entregar, así como la planificación inicial, que constituye la parte más crítica puesto que incluye los plazos, hitos del proyecto, periodicidad de revisiones del proyecto y recursos de los que dispondrá el equipo. El recurso más importante será la disposición de personal, tal y como ya hemos dicho, los profesionales que tomarán parte en el proyecto serán un director de proyecto, un ingeniero senior y un ingeniero junior. En esta etapa el director del proyecto también compartirá los programas y código necesarios para comenzar el trabajo.

La fase número dos consiste en la recogida de información para la elaboración de un estado del arte. El Ingeniero Senior llevará a cabo una búsqueda de información de las tecnologías y métodos más utilizados en los procesos de almacenamiento.

Figura 37. Fases del Proyecto.

En la tercera fase los ingenieros junior y senior llevarán a cabo un estudio del código, en el que analizarán la programación ayudándose de la implementación de pequeñas modificaciones. Se comentará sobre el código los distintos métodos y las modificaciones.

En la cuarta etapa se llevará a cabo la elaboración del entregable, en el que se incluirá toda la información requerida para lograr los pasos anteriores explicándose punto por punto las distintas mejoras realizadas y la documentación del programa del almacén.

Para finalizar, en la cuarta etapa, se presentarán los resultados, con el código comentado, demostrado por los cambios en la instalación y los puntos de optimización llevados a cabo en la instalación. Esta presentación la llevará a cabo el propio director del proyecto, en esta presentación el jefe de logística participará recibiendo la información sobre el código, su funcionamiento y compartiendo impresiones sobre su satisfacción con el proyecto.

4 Coste de elaboración del proyecto

Primero calculamos las horas y semanas efectivas anuales por cada miembro del proyecto. Después teniendo en cuenta los salarios de cada profesional que participa en el proyecto, calcularemos su coste por semana y hora para cada uno. En esta tabla hemos considerado media hora de descanso repartida durante la jornada laboral de 8 horas.

Tabla 5. Cálculo del tiempo efectivo.

Concepto	Tiempo
Duración año medio	365,25 días
Sábados y domingos	-104,36 días
Duración de vacaciones	-23 días
Duración festivos reconocidos	-12 días
Media de tiempo perdido por enfermedad	-15 días
Duración año jornada laboral año medio	210,89 días
Tiempo total efectivo en horas	1581,675 horas

Los salarios de los participantes del proyecto se han establecido según los salarios medios registrados en [28]. Se ha considerado que los participantes tenían las siguientes experiencias: El director de proyecto contaba con más de 20 años de experiencia, el Ingeniero senior contaba con más de 10 años de experiencia y el ingeniero junior recién egresado o sin experiencia.

Tabla 6. Coste asociado a salarios.

Concepto	Director de proyectos	Ingeniero Senior	Ingeniero Junior
Salario Bruto	89.000,00 €	69.000,00 €	20.500,00 €
Seguridad Social Empresa	31.150,00 €	24.150,00 €	7.175,00 €
Total	120.150,00 €	93.150,00 €	27.675,00 €
Coste diario	569,73 €	441,70 €	131,23 €
Coste horario	75,96 €	58,89 €	17,50 €

4.1 Cálculo de las Amortizaciones para el equipo informático.

Hoy en día en cualquier proyecto de ingeniería resulta muy importante considerar el impacto económico proveniente del equipo informático, tanto hardware como software. Para este cálculo hemos tomado un período de amortización de 8 años para los equipos de información y 5 años para el software de los equipos. En ambos casos con una amortización con cuota lineal y con valor residual nulo. La amortización debida a este proyecto la calcularemos dividiendo del total el periodo correspondiente a la duración del proyecto.

Tabla 7. Coste de Amortizaciones del Equipo Informático.

Concepto	Coste	Cantidad	Coste Total
Pórtatil HP Pavillion 4-dv1004ns	600,00 €	1	600,00 €
Teléfono móvil +línea	300,00 €	1	300,00 €
Software - paquete Office 365	345,00 €	1	345,00 €
Software - Antivirus	70,00 €	1	70,00 €
Impresora	75,00 €	1	75,00 €
Total a amortizar			**1.390,00 €**
	Tipo	Período	Amortización
	Anual	5	**278,00 €**
	Semanal	260	5,35 €
	Diaria	1.054	1,32 €
	Horaria	7.908	**0,18 €**

4.2 Coste del material Consumible

Este proyecto siempre se ha llevado en cabo teniendo en mente prácticas sostenibles como evitar las fotocopias innecesarias y la utilización de material no digital. Pese a ello tenemos un residual en su mayor parte debido a la impresión y encuadernación del informe de resultados. Los costes expresados en la siguiente tabla se corresponden exclusivamente al coste de dicha impresión y encuadernación.

Tabla 8. Coste de Material Consumible.

Concepto	Coste
Tóner	30,00 €
Papel	20,00 €
Material de oficina	5,00 €
Servicio reprografía de encuadernación.	15,00 €
Coste del proyecto en material consumible	70,00 €

4.3 Costes Indirectos

En este apartado incluimos gastos de electricidad, línea de teléfono y oficinas entre otros. Reflejamos estos costes y su valor anual por persona. Para este ejercicio hemos considerado que el alquiler de una oficina en Valladolid para un espacio de unos 130 metros cuadrados es de unos 600 €.

Tabla 9. Costes Indirectos.

Concepto	Coste
Alquiler	600 € /mes
Internet y teléfono fijo	100 € /mes
Electricidad	45 € /mes
Otros	50 € /mes
Coste anual de oficina	9.540,00 €
Coste anual por persona	954,00 €
Coste horario por persona	0,60 €

4.4 Costes asignados a cada fase del proyecto

En la siguiente tabla se incluye una estimación sobre cuál ha sido la dedicación de los participantes a cada una de las fases del proyecto.

Tabla 10. Número de horas según persona y fase del proyecto.

	Etapas					
Personal	1	2	3	4	5	Total
Director	14	5	-	-	2	21
Ingeniero Senior	5	15	60	5	3	83
Ingeniero Junior	-	85	200	40	10	295

Después de haber considerado todas las horas necesarias y costes podemos hacer el desglose del coste de cada una de las etapas. En cada una de las fases vamos a calcular el coste del personal como producto de las horas empleadas por cada participante por el coste horario asumido a cada persona. Además, se sumará el gasto de amortización de equipos y de costes indirectos, siempre proporcionales al número de horas dedicadas al proyecto por cada uno de los profesionales.

Costes asignados a la fase 1 de Puesta en Marcha del Proyecto

Tabla 11. Coste asociado a fase 1 del proyecto.

Concepto		Horas	Coste horario	Coste total
Personal	Director	14	75,96 €	1.063,49 €
	Ingeniero Senior	5	58,89 €	294,47 €
Amortización de equipos		19	0,18 €	3,42 €
Costes Indirectos		19	0,60 €	11,40 €
Coste total fase 1:				**1.372,78 €**

Costes asignados a la fase 2 de Recopilación de Información

Tabla 12. Coste asociado a Fase 2 del proyecto.

Concepto		Horas	Coste horario	Coste total
Personal	Director	5	75,96 €	379,82 €
	Ingeniero Senior	15	58,89 €	883,40 €
	Ingeniero Junior	85	17,50 €	1.487,27 €
Amortización de equipos		105	0,18 €	18,90 €
Costes Indirectos		105	0,60 €	63,00 €
Coste total fase 2:				**2.832,39 €**

Costes asignados a la fase 2 de Análisis de Código

Tabla 13. Coste asociado a Fase 3 del proyecto.

Concepto		Horas	Coste horario	Coste total
Personal	Ingeniero Senior	60	58,89 €	3.533,60 €
	Ingeniero Junior	200	17,50 €	3.499,45 €
Amortización de equipos		105	0,18 €	18,90 €
Costes Indirectos		105	0,60 €	63,00 €
Coste total fase 3:				**7.114,95 €**

Costes asignados a la fase 4 de Elaboración del Informe

Tabla 14. Coste asociado a Fase 4 del proyecto.

Concepto		Horas	Coste horario	Coste total
Personal	Ingeniero Senior	5	75,00 €	375,00 €
	Ingeniero Junior	40	17,50 €	699,89 €
Amortización de equipos		45	0,18 €	8,10 €
Costes Indirectos		45	0,60 €	27,00 €
Coste material consumible				70,00 €
Coste total fase 4:				**1.179,99 €**

Costes asignados a la fase 5 de Presentación de Resultados

Tabla 15. Coste asociado a Fase 5 del proyecto.

Concepto		Horas	Coste horario	Coste total
Personal	Director	2	75,96 €	151,93 €
	Ingeniero Senior	3	58,89 €	176,68 €
	Ingeniero Junior	10	17,50 €	174,97 €
Amortización de equipos		15	0,18 €	2,70 €
Costes Indirectos		15	0,60 €	9,00 €
Coste total fase 5:				**515,28 €**

4.5 Coste Total del Proyecto

El coste total del proyecto lo obtenemos sumando el coste total de cada una de las fases que forman parte del proyecto. El desglose del coste del proyecto por fases se muestra en la Figura 38 y el desglose por tipo de coste se muestra en la Figura 39.

Tabla 16. Coste Total del Proyecto por fases.

Concepto	Coste
Fase 1	1.372,78 €
Fase 2	2.832,39 €
Fase 3	7.114,95 €
Fase 4	1.099,46 €
Fase 5	515,28 €
Coste Total	**12.934,85 €**

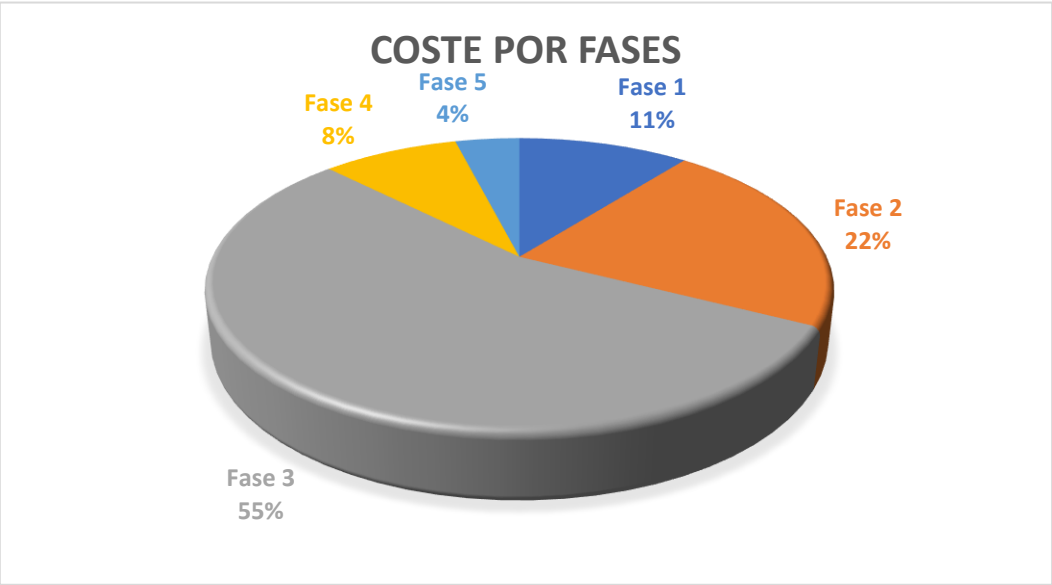


Figura 38. Coste del proyecto por fases.

Figura 39. Coste del proyecto por conceptos.

Conclusiones y líneas futuras

El proyecto partía con el objetivo de poder aplicar cambios más profundos sobre la programación de la fábrica que pudieran dar lugar a la aplicación de conceptos del Lean Manufacturing. Sin embargo, se ha podido comprobar que la comprensión del código y la aplicación de mejoras no es un proceso sencillo; los cambios se deben compilar, ver que no hay ningún error, cargar los programas sobre los controladores y ver la respuesta sobre el modelo. Algunas veces los cambios se han realizado con éxito y en otros casos no se han podido aplicar.

El gran obstáculo con el que nos hemos encontrado ha sido la elevada complejidad de la programación. Es por ello por lo que se han empleado gran cantidad de horas para la comprensión del código, con la dificultad añadida de ser un código con muy pocos comentarios y para el que no existe ningún tipo de documentación, salvo los diagramas de estado de los distintos módulos de la fábrica que se pueden encontrar en la página de GitHub. Además, el modelo de fábrica con el que se cuenta para el estudio no incluye un PLC por lo que se hace difícil observar el funcionamiento de sensores y actuadores en tiempo real. El proceso de adaptación al manejo de la gran cantidad de librerías ha sido complejo y la forma de adaptarse al manejo de las librerías ha sido la de llevar a cabo modificaciones y mediante la revisión e interpretación del código. Además, el proceso de comprobación de los cambios propuestos es lento ya que para ello es necesario reunirse, comprobar que funciona y proponer nuevos objetivos.

En este trabajo se ha llevado a cabo una completa explicación de la programación del almacén junto con una serie de cambios propuestos centrados principalmente en esta estación. Aunque se han aplicado cambios en otras estaciones, por falta de tiempo, no se ha podido detallar la explicación de todas las estaciones. Aun así, una vez que se conoce cómo funciona una máquina de estados y cómo se lleva a cabo la comunicación MQTT entre estaciones resultará mucho más fácil llegar a comprender el funcionamiento de todas las FSM y las comunicaciones MQTT, las cuales constituyen el eje principal de la programación de la fábrica. Este trabajo podría servir de base para llevar a cabo la explicación completa de todas las estaciones. Sería interesante completar una explicación del robot de aspiración (VGR), ya que constituye el motor de la fábrica y es esta estación

la que más posibilidades puede ofrecer en la aplicación de modificaciones, como el procesamiento simultáneo de varias piezas, la reducción de tiempos, la detección de errores y otros posibles cambios dentro de la metodología Lean junto con la introducción de nuevas herramientas que permitan obtener más datos de la fabricación a través de los sensores y actuadores.

Aun así, con todas las dificultades, se ha conseguido entender gran parte del funcionamiento de los programas principales de la fábrica que han permitido aplicar cambios que ayudan a entender el funcionamiento de la fábrica. El trabajo pretende ser de utilidad para usuarios futuros de la fábrica, y ser un punto de partida para la mejora de la eficiencia de los procesos, para la aplicación de conceptos de *Lean Manufacturing* y la implementación de una interfaz y un posible gemelo digital de la fábrica en un futuro.

El modelo de fábrica de industria 4.0 puede resultar una herramienta muy útil en la formación de alumnos. Un modo de trabajo mediante el cual podrían trabajar los alumnos podría ser mediante pequeños proyectos centrados en pequeñas áreas de la fábrica. Se podrían llevar a cabo tormentas de ideas con el objetivo de poder aplicar nuevas mejoras. Además, se podrían realizar talleres en distintas asignaturas del área de ingeniería en los que se pusiera en práctica el concepto de aprendizaje activo mediante el uso de la fábrica.

La utilización de la herramienta Node-Red podría ser de gran utilidad para la implementación de una nube propia, y para poder programar el protocolo MQTT de una forma más fácil que directamente sobre la programación en C++. Para poder utilizar esta herramienta es necesario el uso de un dispositivo que soporte Node-Red como por ejemplo una *Raspberri pi* (no viene incorporada con el modelo). Mediante esta herramienta se podrían obtener indicadores o KPIs del proceso de producción, especialmente orientados a la toma de tiempos, utilizando los *time stamp* se podrían obtener tiempos de ciclo.

Además, se podría añadir la utilización de un autómata programable o PLC. La programación de la fábrica mediante estos dispositivos podría resultar un área de programación más familiar para los ingenieros, especialmente los del área de ingeniería electrónica. La implantación de un PLC junto a la utilización de un software de simulación como *Witness*, por ejemplo, sería un gran avance previo a la creación del gemelo digital. Este software es el que se usa en la universidad para los talleres de simulación y por tanto los alumnos ya están familiarizados con esta herramienta.

Referencias

[1] Alexander Steiger, "FischertechnikTraining Factory Industry 4.0 (9V)." https://github.com/fischertechnik/txt_training_factory (accessed Jul. 04, 2022).

[2] A. Garrell Guiu and L. Guilera Agüera, *La industria 4.0 en la sociedad digital*. Marge Books, 2019.

[3] K. De Backer, C. Menon, I. Desnoyers-James, and L. Moussiegt, "Reshoring: Myth or Reality?," no. 27, 2016, doi: https://doi.org/https://doi.org/10.1787/5jm56frbm38s-en.

[4] "¿Qué es un SGA? - Mecalux.es." https://www.mecalux.es/manual-almacen/almacen/que-es-un-sga (accessed Jul. 04, 2022).

[5] E. Abele *et al.*, "Learning factories for research, education, and training," in *Procedia CIRP*, 2015, vol. 32, pp. 1–6, doi: 10.1016/j.procir.2015.02.187.

[6] L. A. Laguna, "TFM Análisis de las 'Lean Learning Factories' del mundo," 2021.

[7] "¿Qué es la Industria 4.0 y cómo funciona? | IBM." https://www.ibm.com/es-es/topics/industry-4-0 (accessed Jul. 11, 2022).

[8] H. Weininger, *Manual de Instrucciones de la Fábrica de Formación de Industria 4.0.*, 1ª. Waldachtal, 2019.

[9] Luis Llamas, "Protocolos de comunicación para IoT," 2019. https://www.luisllamas.es/protocolos-de-comunicacion-para-iot/ (accessed Jul. 04, 2022).

[10] "Protocolos de comunicación para IoT." https://www.luisllamas.es/protocolos-de-comunicacion-para-iot/ (accessed Jul. 05, 2022).

[11] OASIS Message Queuing Telemetry Transport (MQTT) TC, "MQTT Version 3.1.1," *OASIS Standard Incorporating Approved Errata 01*, 2015. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (accessed Jun. 30, 2022).

[12] Microsoft Docs, "Patrón de publicador y suscriptor - Azure Architecture Center | Microsoft Docs." https://docs.microsoft.com/es-es/azure/architecture/patterns/publisher-subscriber (accessed Jun. 30, 2022).

[13] The HiveMQ Team, "Publish & Subscribe - MQTT Essentials: Part 2," 2015. https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/ (accessed Jun. 30, 2022).

[14] The HiveMQ Team, "MQTT Client and Broker and MQTT Server and Connection Establishment Explained - MQTT Essentials: Part 3," 2019. https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/ (accessed Jun. 30, 2022).

[15] "¿Qué es MQTT? Su importancia como protocolo IoT," 2019. https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/ (accessed Jul. 12, 2022).

[16] E. Balagurusamy, *Programación orientada a objetos con C++*. McGraw-Hill, 2006.

[17] L. J. Aguilar, *Programación en C++. Algoritmos, estructuras de datos y obsjetos*. McGraw-Hill Interamericana de España S.L., 2006.

[18] P. Bustamante, I. Aguinaga, M. Aybar, L. Oloizola, and I. Lazacano, *Aprenda C++ Básico como si estuviera en primero*. San Sebastián: Campus Tecnológico de la Universidad de Navarra., 2010.

[19] P. Bustamante, I. Aguinaga, M. Aybar, L. Olaizola, and I. Lazcano, "Aprenda C++ Avanzado como si estuviera en primero," *Campus Tecnológico la Univ. Navarra*, 2004.

[20] B. Eckel and C. Allison, *Thinking in C++*, no. v. 1. Prentice Hall, 2000.

[21] F. J. C. Sierra, *Programación orientada a objetos con C++*, vol. 3. Grupo Editorial RA-MA, 2007.

[22] J. Liberty and R. Cadenhead, *Programación C++*. ANAYA MULTIMEDIA, 2011.

[23] UNIR La Universidad de Internet, "¿Qué es un IDE en programación? | UNIR." https://www.unir.net/ingenieria/revista/ide-programacion/ (accessed Jun. 30, 2022).

[24] "Visión general de CDT - Documentación de IBM." https://www.ibm.com/docs/es/developer-for-zos/9.1.1?topic=concepts-cdt-overview (accessed Jul. 10, 2022).

[25] HIVEMQ, "MQTT Topics, Wildcards, & Best Practices - MQTT Essentials: Part 5," 2019. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-

practices/ (accessed Jun. 30, 2022).

[26] C .van Leeuwen, “The Fischertechnik Transfer Area,” 2020. https://github.com/fischertechnik/txt_demo_c_download/blob/master/TransferArea.md (accessed Jul. 04, 2022).

[27] Stefan Brunner, “Fischertechnik TX Outputs - Stefan Brunner,” 2012. http://stefanbrunner.com/fischertechnik-tx-outputs/ (accessed Jul. 04, 2022).

[28] “Sueldo de un ingeniero en España.” https://www.jobted.es/salario/ingeniero (accessed Jul. 13, 2022).

[29] Fernando García de Zúñiga, “Qué son los caracteres UTF-8 y qué ventajas tienen en la web - Blog de arsys.es,” 2021. https://www.arsys.es/blog/utf8 (accessed Jul. 02, 2022).

ANEXOS

Funcionamiento de la estación de entrada y salida de piezas

La DPS cuenta con 6 áreas de trabajo: una estación de calibración, la bandeja de entrada de entrada y la bandeja de salida de piezas con fototransistor, la bandeja de piezas defectuosas, el lector NFC y el sensor de color.

Cada vez que una pieza nueva entra por esta estación, un fototransistor detecta la pieza y se manda un mensaje al VGR para que se dirija a por ella. El VGR se dirige al sensor de color para clasificar el color y tras esto se dirige al dispositivo NFC para marcar los nuevos datos. Si se identifica como uno de los tres colores la materia prima pasa el control de calidad y en caso de que no, el VGR desplazará la pieza a la bandeja de piezas defectuosas. Después de pasar por el resto de las estaciones de la fábrica para llevar a cabo el procesado y la clasificación de la pieza en la estación de clasificación, el VGR vuelve a coger la pieza para volver a llevarla hasta el dispositivo NFC de la DPS para marcar la pieza como procesada y dejarla después en la bandeja DOUT de salida de piezas. Las funciones de la DPS por lo tanto son las siguientes:

- Detectar pieza en bandeja DIN de entrada de piezas.
- Detectar color de la pieza en el sensor de color.
- Borrar datos previos de la pieza e identificar la pieza como materia prima.
- Borrar datos previos e identificar la pieza como procesada.
- Detectar pieza en bandeja DOUT de salida de piezas.

La DPS no cuenta con una FSM propia porque es el VGR el que lleva el control de los procesos por los que pasa la pieza en esta estación.

La Clase DeliveryPickUpStation

Los métodos para llevar a cabo los procesos son los siguientes:

DeliveryPickUpStation::nfcDeviceDeleteWriteRawRead

Se realiza el proceso de identificación de la pieza previo a su almacenaje. Este método identifica la pieza, asignando un id y borra los datos previamente marcados. Se le asigna a esta pieza el estado RAW para marcarla como si fuera materia prima.

Se crea un objeto de tipo *WorkPiece* (mediante la llamada al constructor) con: un id vacío (“ ”), el tipo de pieza que recibe como parámetro el método y se marca como materia prima (estado RAW). Después el NFC escribe los nuevos datos sobre la pieza y se lee el identificador de la pieza, el cual es devuelto al terminar el método. Así se tiene la pieza de materia prima identificada para llevarse al almacén.

```
std::string TxtDeliveryPickupStation::nfcDeviceDeleteWriteRawRead(ft::TxtWPType_t c, std::vector<int64_t> vts, uint8_t mask_ts)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveNFCDeviceDeleteWriteRawRead", 0);
    nfcDelete();
    TxtWorkpiece wp("", c, WP_STATE_RAW);
    nfcWrite(wp, vts, mask_ts);
    std::string sid = nfcRead();
    return sid;
}
```

Figura 40. Método nfcDeleteWriteRawRead de la clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::nfcDeviceWriteProducedRead

Se realiza la identificación de la pieza una vez que ha sido procesada. Se borran los datos NFC anteriores y se marcan los nuevos para indicar que ha sido procesada. Funciona exactamente igual que el método anterior pero ahora, al llamar al constructor del objeto de tipo pieza, recibe el estado de PROCESSED.

```
std::string TxtDeliveryPickupStation::nfcDeviceWriteProducedRead(ft::TxtWPType_t c, std::vector<int64_t> vts, uint8_t mask_ts)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveNFCDeviceWriteProducedRead", 0);
    TxtWorkpiece wp("", c, WP_STATE_PROCESSED);
    nfcWrite(wp, vts, mask_ts);
    std::string sid = nfcRead();
    return sid;
}
```

DeliveryPickUpStation::nfcDeviceWriteRejectedRead

Este método se llama en el caso de que el color haya sido detectado erróneamente en el sensor de color para marcar esta pieza como rechazada. Funciona exactamente igual que los dos métodos anteriores, pero ahora, al llamar al constructor del objeto de tipo pieza, recibe el estado REJECTED.

```
std::string TxtDeliveryPickupStation::nfcDeviceWriteRejectedRead(ft::TxtWPType_t c, std::vector<int64_t> vts, uint8_t mask_ts)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "moveNFCDeviceWriteRejectedRead", 0);
    nfcDelete();
    TxtWorkpiece wp("", c, WP_STATE_REJECTED);
    nfcWrite(wp, vts, mask_ts);
    std::string sid = nfcRead();
    return sid;
}
```

DeliveryPickUpStation::configInputs

Método para configurar los sensores de la DPS: fototransistor de entrada, fototransistor de salida y sensor de color.

```
void TxtDeliveryPickupStation::configInputs()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "configInputs", 0);
    assert(pT->pTArea);
    //Confirgurar DIN
    pT->pTArea->ftX1config.uni[6].mode = MODE_R; //Entrada fototransistor I7  Digital Switch with PullUp resistor
    pT->pTArea->ftX1config.uni[6].digital = 1;
    //trigger first read (workaround first value is wrong)
    u16LastStateDIN = pT->pTArea->ftX1in.uni[6];
    //Configurar DOUT
    pT->pTArea->ftX1config.cnt[3].mode = MODE_R; // Entrada fototransistor C4  Digital Switch with PullUp resistor
    //trigger first read (workaround first value is wrong)
    u16LastStateDOUT = pT->pTArea->ftX1in.cnt_in[3];
    //Configurar ColorSensor
    pT->pTArea->ftX1config.uni[7].mode = MODE_U; //Entrada sensor de color I8 Analog Voltage
    pT->pTArea->ftX1config.uni[7].digital = 0;
    //Actualizar configuración
    pT->pTArea->ftX1state.config_id ++; // Save the new Setup
}
```

Código 85. Método configInputs() de la clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::is_DIN

Este método se encarga de detectar la entrada de una pieza en la bandeja de entrada mediante el fototransistor, que devuelve true en el caso en el que el fototransistor de entrada detecte la presencia de una pieza.

```
bool TxtDeliveryPickupStation::is_DIN()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "is_DIN", 0);
    assert(pT->pTArea);
    return (pT->pTArea->ftX1in.uni[6] == 1); //Devuelve TRUE si fototransistor entrada I7 detecta pieza
}
```

Código 86. Método is_DIN() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::is_DOUT

Devuelve true en el caso en el que el fototransistor de salida de piezas detecte una pieza. Detecta que la pieza ya ha sido procesada y está disponible para salir de la fábrica.

```
bool TxtDeliveryPickupStation::is_DOUT()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "is_DOUT", 0);
    assert(pT->pTArea);
    return (pT->pTArea->ftX1in.cnt_in[3] == 1); //Devuelve TRUE si fototransistor entrada I4 deetcta pieza
}
```

Código 87. Método is_DOUT() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::readColorValue

Lee al valor asociado al color detectado mediante el sensor de color y se lo asigna a la variable *lastColorValue*.

```
int TxtDeliveryPickupStation::readColorValue()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getColorValue", 0);
    assert(pT->pTArea);
    lastColorValue = pT->pTArea->ftX1in.uni[7];//Toma el valor del Voltaje (0-9000 mv) captado por fototransistor
    return lastColorValue;
}
```

Código 88. Método readColorValue() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::getLastColor

Compara el valor de la variable *lastColorValue* con los valores almacenados en la clase de calibración de la DPS. Dependiendo del rango de valores en el que se sitúe la variable *lastColorValue* el método devuelve un tipo de pieza u otro.

```
// Esta es la función que lee el color de la pieza que entra en el sistema.
ft::TxtWPType_t TxtDeliveryPickupStation::getLastColor()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "getColor", 0);
    if ((lastColorValue >= 200)&&(lastColorValue < calibData.color_th[0])) //color_th[0]=850
    {

        return WP_TYPE_WHITE; //PRUEBA introduir una pieza blanca, interprete que es roja. return WP_TYPE_RED;
    }
    else if ((lastColorValue >= calibData.color_th[0])&&(lastColorValue < calibData.color_th[1])) //color_th[0]=850;
                                                                                                  //color_th[1]=1550
    {
        return WP_TYPE_RED;
    }
    else if ((lastColorValue >= calibData.color_th[1])&&(lastColorValue < 2000))
    {
        return WP_TYPE_BLUE;
    }
    return WP_TYPE_NONE;
}
```

Código 89. Método getLastColor() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::nfcRead

Mediante el dispositivo NFC se lee toda la información de la etiqueta NFC y se emite el sonido de lectura.

```
std::string TxtDeliveryPickupStation::nfcRead()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "nfcRead", 0);
    sound.info1();
    return nfc.readTags(); // Lee información de la etiqueta NFC
}
```

Código 90. Método nfcRead() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::nfcDelete

Mediante el dispositivo NFC se borra la información de la etiqueta.

DeliveryPickUpStation::nfcReadUID

Mediante el dispositivo NFC se lee el identificador de la pieza mediante el método *readTagsGetUID()* de la clase *TxtNfcDevice*.

```
std::string TxtDeliveryPickupStation::nfcReadUID()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "nfcReadUID", 0);
    //sound.info1();
    return nfc.readTagsGetUID(); // Lee id de la pieza.
}
```

Código 91. Método nfcReadUID() de la Clase TxtDeliveryPickUpStation.

DeliveryPickUpStation::nfcWrite

Mediante el dispositivo NFC se escriben los datos de la pieza sobre la etiqueta y se emite sonido para indicar que ya se ha procesado la escritura de los datos.

```
bool TxtDeliveryPickupStation::nfcWrite(TxtWorkpiece wp, std::vector<int64_t> vts, uint8_t mask_ts)
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "nfcWrite {} {} {}", wp.state, wp.type, vts.size());
    std::vector<uTS> vuTS;
    for(unsigned int i = 0; i < vts.size(); i++)
    {
        uTS uts;
        uts.s64 = vts[i];
        vuTS.push_back(uts);
    }
    sound.info1();
    return nfc.writeTags(wp, vuTS, mask_ts);
}
```

Código 92. Método nfcWrite() de la Clase TxtDeliveryPickUpStation

DeliveryPickUpStation::run

Se notifica a los observers y se actualizan las variables *reqUpdateDIN* y *reqUpdateDOUT* a false para actualizar los estados de petición de entrada y salida de piezas.

```
void TxtDeliveryPickupStation::run()
{
    SPDLOG_LOGGER_TRACE(spdlog::get("console"), "run",0);
    while (!m_stoprequested)
    {
        if (reqUpdateDIN)
        {
            Notify();
            reqUpdateDIN = false;
        }
        if (reqUpdateDOUT)
        {
            Notify();
            reqUpdateDOUT = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}
```

Código 93. Método run() de la Clase TxtDeliveryPickUpStation.