

Grado en Ingeniería Informática

Inteligencia Ambiental Curso

2023/2024



Universidad de Jaén

**Empresa de reparto
usando LEGO**

Rafael Osuna Ventura
rov00005@red.ujaen.es

ÍNDICE

1. INTRODUCCIÓN	1
 1.1. CONTEXTO	1
 1.2. DESCRIPCIÓN DEL PROBLEMA.....	1
 1.3. HERRAMIENTAS	2
 1.4. ALGORITMO BFS	7
2. OBJETIVOS	9
3. REQUISITOS	11
 3.1 REQUISITOS FUNCIONALES	11
 3.2 REQUISITOS NO FUNCIONALES.....	12
4. ARQUITECTURA	12
5. PLANIFICACIÓN	18
6. ANÁLISIS Y DISEÑO	20
 6.1 DIAGRAMA DE CASOS DE USO	20
 6.2 GRAMÁTICA DEL CASO DE USO	21
 6.3 DIAGRAMA DE SECUENCIA	22
 6.4 DIAGRAMA DE ACTIVIDAD	23
 6.5 ALGORITMO BFS	25
 6.6 DIAGRAMA UML.....	26
 6.7 MOCKUPS.....	28
7. IMPLEMENTACIÓN.....	34
 7.1 APLICACIÓN CLIENTE	34
 7.1.1 COMUNICACIONES.....	34
 7.1.2 APARIENCIA	38
 7.2 ROBOT	56
 7.2.1 COMUNICACIONES Y PROCESAMIENTO DE MENSAJES	56
 7.2.2 MOVIMIENTO	62

8. DESPLIEGUE	71
9. RESULTADOS	78
9.1 TIEMPOS	79
9.2 TIEMPOS	81
9.3 DIFICULTADES	83
10. MEJORAS FUTURAS	85
11. REFERENCIAS.....	88

1. INTRODUCCIÓN

1.1. CONTEXTO

Actualmente, gran parte del comercio global son compras online, en las cuales el usuario desde su casa o cualquier sitio busca una interacción fácil y rápida que le permita comprar lo que busca y recibirla con el menor tiempo de espera posible.

Debido a esta nueva necesidad, las empresas de reparto de paquetes buscan constantemente la manera más óptima y eficiente de conseguir este objetivo y satisfacer las demandas de los clientes. Que una empresa cumpla o no este objetivo es un factor determinante en la elección del usuario por una empresa u otra. Por ello, este sector de empresas no deja de invertir en la búsqueda tanto de nuevos mecanismos de reparto, como de nuevas estrategias que mejoren la eficiencia, como en la implementación de nuevas tecnologías y en el desarrollo de nuevos algoritmos de rutas. En este contexto, el proyecto que se nos ha asignado, aunque en una menor escala, responde al mismo reto de estas empresas.

1.2. DESCRIPCIÓN DEL PROBLEMA

Se ha asignado a nuestro equipo la programación de un robot de una famosa empresa de reparto de paquetería a través del uso de la plataforma robótica LEGO Mindstorms EV3. El objetivo es crear un sistema completo y automatizado para la entrega de un paquete desde un punto “a” hasta un punto “b” de la forma más óptima posible. Para ello, se simulará el proceso completo de entrega de paquetes, desde la recogida inicial hasta la entrega final al destinatario, todo ello gestionado por un sistema centralizado el cual poseerá una serie de algoritmos.

Podremos definir 3 etapas en este proceso:

- Recogida: el paquete es recogido por el robot desde el punto de origen. Dicho robot estará correctamente configurado para ejecutar la acción empleando diferentes sensores y actuadores.

- Preparación: implica la asignación de rutas óptimas desde el punto inicial al punto final, teniendo en cuenta los diferentes obstáculos a evitar. Se emplearán algoritmos de inteligencia artificial (BFS [\[1\]](#)) para llevar a cabo todas las operaciones.
- Entrega: el robot tomará la ruta más óptima para entregar el paquete de la manera más rápida y eficiente.

1.3. HERRAMIENTAS

Para la programación del robot “LEGO Mindstorms EV3”, emplearemos la librería Pybricks [\[2\]](#), la cual nos proporciona las herramientas necesarias y específicas para el diseño de las diferentes tareas que vamos a implementar. Esta librería es fácil e intuitiva ya que se basa en el lenguaje MicroPython, este es una versión simplificada y eficiente del lenguaje Python 3. Todo ello será implementado en el entorno de desarrollo Visual Studio Code [\[5\]](#). En dicho entorno, vamos a instalar la extensión LEGO MINDSTORMS EV3 MicroPython. Esta herramienta adicional nos facilita la programación e implementación con el robot haciendo uso de la librería ya comentada anteriormente, a la vez que nos brinda diversas funcionalidades específicas.

Como ya se ha mencionado, para la realización del código que ejecutará el robot “LEGO Mindstorms EV3” se utilizará Visual Studio Code [\[5\]](#). Visual Studio Code (VS Code) [\[5\]](#) es un entorno de desarrollo integrado (IDE) altamente popular desarrollado por Microsoft. Destaca por su versatilidad, ya que está disponible en múltiples plataformas, incluyendo Windows, macOS y Linux, lo que lo convierte en una opción accesible para desarrolladores de diversos sistemas operativos. Su interfaz de usuario intuitiva y su amplia gama de características hacen que sea fácil de usar para principiantes y potente para usuarios avanzados.

Una de las características más destacadas de VS Code [\[5\]](#) es su ecosistema de extensiones, que permite personalizar el IDE para satisfacer las necesidades específicas de desarrollo. Con soporte para una amplia variedad de lenguajes de programación y tecnologías, así como herramientas de integración como Git y herramientas de depuración integradas, VS Code [\[5\]](#) proporciona un entorno de

desarrollo completo y eficiente. Además, su capacidad de colaboración en tiempo real a través de la función Live Share lo convierte en una herramienta invaluable para equipos distribuidos. En resumen, Visual Studio Code [5] es una opción versátil y poderosa para desarrolladores de software de todos los niveles de experiencia.

Para el desarrollo de la aplicación cliente se utilizará el entorno de desarrollo integrado NetBeans [6]. NetBeans [6] es un entorno de desarrollo integrado (IDE) de código abierto y multiplataforma desarrollado principalmente por Apache Software Foundation y patrocinado anteriormente por Sun Microsystems (adquirida por Oracle). Es conocido por su robustez y versatilidad, ofreciendo soporte para una amplia gama de lenguajes de programación como Java [7], C/C++, PHP, y más. Una de las características distintivas de NetBeans [6] es su enfoque en el desarrollo de aplicaciones Java [7], proporcionando una amplia gama de herramientas y características específicas para el desarrollo en este lenguaje, como un editor de código inteligente, herramientas de refactorización, y una interfaz gráfica para el diseño de interfaces de usuario (GUI).

Además de su sólido soporte para Java [7], NetBeans [6] también ofrece capacidades de desarrollo web, incluyendo soporte para HTML, CSS, JavaScript y frameworks web populares como Node.js y AngularJS. Su integración con servidores de aplicaciones y bases de datos, junto con herramientas de depuración y pruebas, hacen que sea una opción atractiva para el desarrollo de aplicaciones empresariales y web. Con una comunidad activa de desarrolladores y una amplia gama de plugins disponibles, NetBeans [6] continúa siendo una opción popular entre los desarrolladores que buscan un entorno de desarrollo completo y robusto para sus proyectos.

La aplicación cliente se programará en el lenguaje Java [7]. Java [7] es un lenguaje de programación de alto nivel y orientado a objetos, desarrollado por Sun Microsystems (ahora propiedad de Oracle) en la década de 1990. Es conocido por su portabilidad, ya que los programas escritos en Java [7] pueden ejecutarse en cualquier plataforma que tenga una máquina virtual Java (JVM) instalada. Esto se debe a que el código Java [7] se compila en bytecode, que luego se ejecuta en la JVM, lo que

proporciona una capa de abstracción entre el código Java [7] y el sistema operativo subyacente.

Java [7] se utiliza ampliamente en una variedad de aplicaciones, desde el desarrollo de aplicaciones empresariales hasta aplicaciones móviles y juegos. Ofrece características como la gestión automática de la memoria a través del recolector de basura, la seguridad integrada a través del modelo de seguridad de Java [7] y una amplia biblioteca estándar que proporciona soporte para diversas tareas de programación.

Para programar la interfaz de usuario se utilizará Java Swing [8], una biblioteca gráfica para el desarrollo de interfaces de usuario (UI) en aplicaciones Java [7]. Forma parte del paquete de bibliotecas de Java Foundation Classes (JFC) y proporciona un conjunto de componentes gráficos que pueden ser utilizados para construir interfaces de usuario interactivas y atractivas. Swing sigue el paradigma de programación orientada a objetos y proporciona componentes como botones, campos de texto, listas y paneles que pueden ser personalizados y combinados para crear interfaces de usuario complejas. Aunque Swing ha sido eclipsado en popularidad por tecnologías más modernas como JavaFX, sigue siendo una opción sólida para el desarrollo de aplicaciones de escritorio en Java [7], especialmente para aplicaciones empresariales y herramientas de productividad.

Para la realización de los mockups se utilizará Marvel App [3]. Marvel App [3] es una plataforma de diseño y prototipado de aplicaciones web y móviles que permite a los equipos de diseño y desarrollo crear y colaborar en prototipos interactivos de forma rápida y sencilla. Con una interfaz intuitiva y herramientas potentes, Marvel App [3] facilita el proceso de diseño desde la conceptualización hasta la implementación.

Una de las características destacadas de Marvel App [3] es su capacidad para crear prototipos interactivos de alta fidelidad sin necesidad de escribir código. Los diseñadores pueden arrastrar y soltar elementos de diseño, como botones, imágenes y campos de texto, para construir interfaces de usuario realistas y funcionales. Además, la plataforma ofrece una variedad de opciones de animación y transición para mejorar la experiencia del usuario y simular el flujo de la aplicación.

Marvel App [3] también facilita la colaboración entre equipos, permitiendo a los miembros del equipo comentar y revisar los prototipos en tiempo real. Los usuarios pueden compartir fácilmente sus diseños con colegas y clientes para obtener comentarios e iterar rápidamente sobre el diseño. Además, Marvel App [3] se integra con herramientas de diseño populares como Sketch y Adobe XD, lo que permite a los diseñadores importar sus diseños existentes y convertirlos en prototipos interactivos sin problemas.

Para la elaboración de la planificación temporal y la división de tareas del proyecto se realizará un diagrama de Gantt utilizando la herramienta GanttPRO [4]. GanttPRO [4] es una aplicación en línea que facilita la creación, gestión y seguimiento de proyectos utilizando diagramas de Gantt. Este software es ideal para equipos de proyectos de cualquier tamaño que necesiten una herramienta para planificar y visualizar las tareas, asignar recursos, establecer dependencias y realizar un seguimiento del progreso del proyecto de manera efectiva.

Una de las características destacadas de GanttPRO [4] es su interfaz intuitiva y fácil de usar, que permite a los usuarios crear rápidamente diagramas de Gantt detallados sin necesidad de conocimientos técnicos especializados. Los usuarios pueden organizar las tareas en una estructura jerárquica, establecer fechas de inicio y finalización, asignar recursos a las tareas y definir dependencias entre ellas.

Además de la planificación inicial, GanttPRO [4] facilita la gestión continua del proyecto mediante la actualización en tiempo real del diagrama de Gantt a medida que se realizan cambios en el proyecto. Los usuarios pueden realizar un seguimiento del progreso del proyecto, controlar los plazos y identificar posibles cuellos de botella o desviaciones del plan original.

GanttPRO [4] también ofrece herramientas de colaboración que permiten a los miembros del equipo trabajar juntos en el mismo proyecto, compartir archivos y comunicarse a través de comentarios y notificaciones. Esto facilita la coordinación entre los miembros del equipo y mejora la eficiencia del proceso de gestión de proyectos.

El diagrama de casos de uso, secuencia, actividades y UML se realizará con Visual Paradigm [9]. Visual Paradigm [9] es una herramienta de modelado y diseño de software líder en la industria que ofrece una amplia gama de características y funcionalidades para ayudar a los equipos de desarrollo de software a visualizar, diseñar y documentar sus proyectos de manera eficiente y efectiva.

Con Visual Paradigm [9], los usuarios pueden crear una variedad de modelos, incluidos modelos de casos de uso, diagramas de clases, diagramas de actividades, diagramas de secuencia y muchos más, utilizando una interfaz intuitiva y fácil de usar. La herramienta también facilita la colaboración entre equipos, permitiendo a los miembros del equipo trabajar juntos en tiempo real en los mismos modelos y proyectos.

Además del modelado de software, Visual Paradigm [9] ofrece capacidades de generación de código y reversión, lo que permite a los desarrolladores generar código a partir de sus modelos y viceversa. Esto ayuda a garantizar la consistencia entre el diseño y la implementación del software.

Visual Paradigm [9] también ofrece herramientas avanzadas de análisis y simulación, lo que permite a los equipos evaluar el rendimiento y la calidad de sus diseños antes de la implementación. Además, la herramienta es altamente personalizable, lo que permite a los usuarios adaptarla a sus necesidades específicas de modelado y diseño de software.

Para la elaboración de las fotos del mapa de la ciudad se ha utilizado Photoshop [10]. Photoshop [10] es una potente herramienta de edición de imágenes desarrollada por Adobe. Con una amplia gama de funciones y capacidades, es ampliamente utilizada por profesionales en diseño gráfico, fotografía, ilustración y otras disciplinas creativas. Permite manipular y retocar imágenes de manera precisa, desde ajustes básicos como recorte y corrección de color, hasta técnicas avanzadas como la composición de varias imágenes, la creación de efectos especiales y la manipulación de texturas. Además, Photoshop [10] cuenta con una interfaz intuitiva que facilita el acceso a sus numerosas herramientas y opciones de personalización,

lo que lo convierte en un software versátil y poderoso para crear y editar imágenes digitales.

Las imágenes explicativas del movimiento del robot se han realizado con Photopea [11]. Photopea [11] es una aplicación en línea de edición de imágenes que ofrece una amplia gama de herramientas y funciones similares a las de Adobe Photoshop [10]. Permite a los usuarios editar imágenes, crear gráficos y manipular fotos directamente desde su navegador web sin necesidad de descargar ningún software adicional. Con una interfaz intuitiva y familiar para aquellos que están acostumbrados a trabajar con Photoshop [10], Photopea [11] ofrece herramientas de selección, capas, ajustes de color, filtros y mucho más. Es una opción popular para aquellos que buscan una alternativa gratuita o económica a Photoshop [10], con la conveniencia de ser accesible desde cualquier dispositivo con conexión a internet.

Los gráficos comparativos de tiempos se han realizado con los gráficos de las hojas de cálculo de Google [12].

1.4. ALGORITMO BFS

El cálculo de la ruta más corta entre cualquier par de posiciones se efectuará mediante el uso del algoritmo BFS [1]. El algoritmo de Búsqueda en Amplitud (BFS, por sus siglas en inglés) [1] es una técnica que permite explorar estructuras jerárquicas, como grafos y árboles. Esta técnica empieza seleccionando un nodo (llamado raíz si es un árbol) y a partir de este visita todos sus nodos vecinos del mismo nivel. Una vez ha visitados todos pasa al nivel inferior.

Para asegurar que el proceso de exploración visita todos los nodos de un nivel, BFS [1] emplea una cola donde guarda los nodos que necesita visitar. Al visitar un nodo, añade a la cola sus vecinos no visitados. Este proceso continúa hasta que se visitan todos los nodos posibles desde el punto de partida, o hasta que la cola se vacía, asegurando que cada capa de la estructura se explore completamente antes de moverse a la siguiente.

Este algoritmo proporciona una serie de ventajas, pero una de ellas, y la cual a nosotros nos interesa y por eso lo hemos empleado, es su habilidad para encontrar el camino más corto en grafos no ponderados. ¿Por qué?

- Al explorar nivel por nivel, BFS [\[1\]](#) llega a cada nodo por el camino con menos aristas desde el origen, asegurando que cualquier camino descubierto primero es el más corto.
- Es ideal para grafos donde todas las aristas tienen el mismo peso, encontrando rutas mínimas basadas en el número de aristas en lugar de en su peso. En nuestro caso, nuestras imágenes representan "calles" donde todas las intersecciones tienen la misma longitud, y el peso total del camino no es relevante. Esto contrasta con algoritmos como Dijkstra o A*, que tienen en cuenta no solo el menor número de aristas, sino también otros factores como la variabilidad del tiempo de viaje.

En un apartado posterior se explicará su implementación.

2. OBJETIVOS

Los principales objetivos que esperamos alcanzar con la finalización de este proyecto dependen en gran medida de la categoría seleccionada, es decir, del nivel de dificultad escogido.

Nuestro equipo ha optado por participar en la categoría C, en la que se espera que el robot pueda recoger un paquete en un punto "a" del mapa y entregarlo en otro punto "b" al menos en dos ocasiones consecutivas. Esto implica que el robot deberá seguir el trazado de las calles, evitando todos los obstáculos que encuentre en el mapa.

Para lograr este objetivo con éxito, el robot deberá cumplir una serie de criterios establecidos en la arquitectura del sistema.

Se diseñarán algoritmos para que el robot siga la ruta óptima de recogida y entrega de los pedidos, con el objetivo de minimizar el tiempo de reparto. Este enfoque busca satisfacer una de las principales demandas de las empresas de reparto: garantizar una experiencia de usuario satisfactoria.

En resumen, los objetivos específicos se desglosan de la siguiente manera:

- Objetivos del robot:
 - El robot será capaz de recoger un paquete en un punto "a".

- El robot será capaz de entregar un paquete de un punto “a” hasta un punto “b”.
 - Este sistema de recogida y entrega se deberá de repetir al menos dos veces consecutivas.
 - El robot deberá de seguir el trazado de las calles evitando los obstáculos impuestos tales como ciudades.
-
- Objetivos de la aplicación cliente:
 - La aplicación permitirá lanzar como mínimo un pedido y poner otro en cola.
 - La aplicación mostrará la odometría (posición y orientación) del robot con una tasa de refresco de 1 s.
 - Para cada pedido se indicará el punto de recogida y entrega de entre las opciones disponibles en el mapa correspondiente.

3. REQUISITOS

3.1. REQUISITOS FUNCIONALES

- Gestión de pedidos:
 - El sistema debe permitir al cliente lanzar tantos pedidos como desee, en el caso de haber más de uno, se ejecutarán por orden de prioridad gracias a una cola de pedidos.
 - El sistema debe de mostrar los datos del pedido que se está ejecutando en el momento.
- Visualización de la odometría:
 - La aplicación cliente debe mostrar la odometría del robot.
 - La odometría debe actualizarse con una frecuencia mínima de 1 Hz para mantener al cliente informado constantemente sobre la ubicación actual del pedido.
- Entrega de Paquetes:
 - El robot debe ser capaz de recoger un paquete en un punto “a” del mapa y entregarlo en un punto “b” a partir de los puntos marcados por el cliente.

3.2. REQUISITOS NO FUNCIONALES

- Eficiencia:
 - La aplicación cliente y el software del robot deben ser eficientes en el uso de recursos para garantizar un rendimiento óptimo del sistema.
- Interfaz de usuario intuitiva:
 - La interfaz de usuario de la aplicación cliente debe ser intuitiva y fácil de usar para garantizar una experiencia positiva para el cliente.
- Rendimiento óptimo:
 - El sistema debe ser capaz de manejar un aumento en el número de pedidos sin empeorar el rendimiento.

4. ARQUITECTURA

La arquitectura propuesta para cumplir con los objetivos y requisitos especificados es la siguiente:

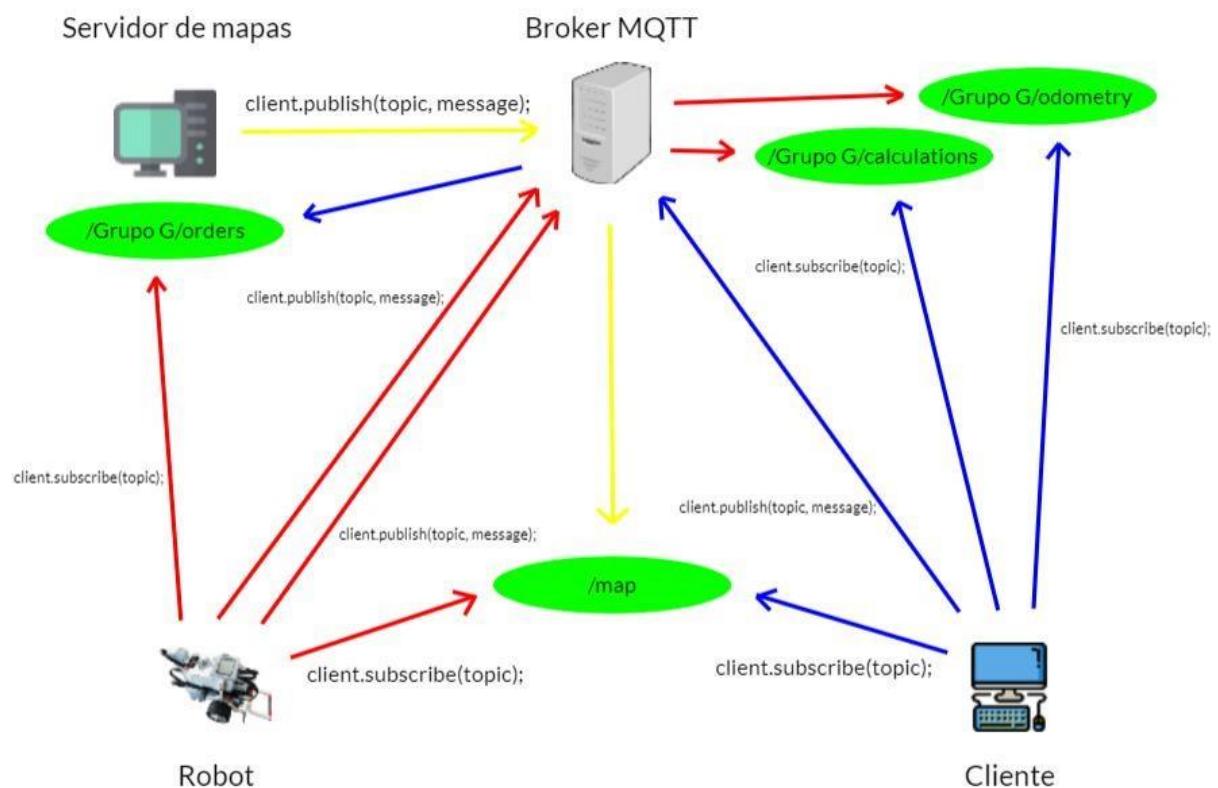


Figura 1. Arquitectura del sistema.

Este diagrama se ha elaborado utilizando la aplicación web Marvel App [\[3\]](#).

Como podemos observar, el sistema consistirá en 4 componentes principales: el broker MQTT, el servidor de mapas, el cliente y el robot.

El broker MQTT es un servidor que recibe y envía mensajes utilizando el protocolo de comunicación MQTT. Este protocolo, conocido como protocolo de publicación-suscripción, opera estableciendo una comunicación con el broker y publicando mensajes asociados a un tema (topic). Todos los clientes interesados en recibir estos mensajes simplemente tienen que establecer una conexión con el broker y suscribirse al tema relevante para recibir los mensajes. Por lo tanto, en este proyecto, el broker MQTT gestionará el envío y la recepción de mensajes entre los distintos equipos involucrados.

El servidor de mapas será el encargado de transmitir el mapa de la ciudad al broker MQTT bajo el tema "/map", permitiendo que todos los clientes interesados puedan acceder a él.

El cliente será un usuario que utiliza una aplicación de escritorio para realizar pedidos. Esta aplicación se suscribirá al tema "/map" para obtener el mapa de la ciudad y representarlo gráficamente. Además, se suscribirá a los temas "/Grupo G/odometry" y "/Grupo G/calculations" para recibir la odometría del robot (ubicación y orientación) y las posibles ubicaciones de recogida y entrega de paquetes, respectivamente. También, publicará los pedidos realizados por el usuario en el tema "/Grupo G/orders" para que el robot pueda recibirlas.

El robot será el encargado de completar los pedidos efectuados por los clientes. Su hardware está basado en la plataforma robótica "LEGO Mindstorms EV3" y queda explicado con el siguiente esquema que hemos creado:

**Servomotores Grandes**

Motor de corriente continua unido a engranajes que amplifican la fuerza y limitan la velocidad a 160-170 rpm. Además, incorpora un encoder óptico que permite medir con precisión la posición y velocidad del eje en incrementos de un grado.

**Servomotor Mediano**

Más compacto y ligero ofrece una menor fuerza tanto en funcionamiento como bloqueo, pero aumenta su velocidad a 240-250 rpm, manteniendo la precisión.

Sensor Color

Consta de emisores de luz en rojo, verde y azul, y un receptor, lo que le permite detectar tanto la cantidad como el color de la luz reflejada. Si se desactivan los emisores, el dispositivo mide la luz ambiental.

**Unidad Principal (Ladrillo)**

Actúa como su cerebro y fuente de poder, contando con un microcontrolador ARM9 con Linux. Al instalar el firmware de Pybricks se activan capacidades avanzadas y conexión SSH, permitiendo controlar elementos como servomotores, sensores, y una interfaz programable, además de incluir opciones de conectividad como Bluetooth y USB.

**Sensor Giro**

Mide la velocidad de rotación hasta 440 grados por segundo y el ángulo de giro total en un eje, con una precisión de ± 3 grados para 90 grados de rotación. El eje de medición es perpendicular a la superficie que muestra el icono de un punto con flechas.



Todos estos elementos más las diferentes piezas formarán nuestro robot



Figura 2. Esquema resumen del robot LEGO Mindstorms EV3.

5. PLANIFICACIÓN

La planificación elaborada para alcanzar los objetivos y requisitos establecidos se ha realizado mediante un diagrama de Gantt utilizando la aplicación web GanttPRO [4]:

Red | ambientales

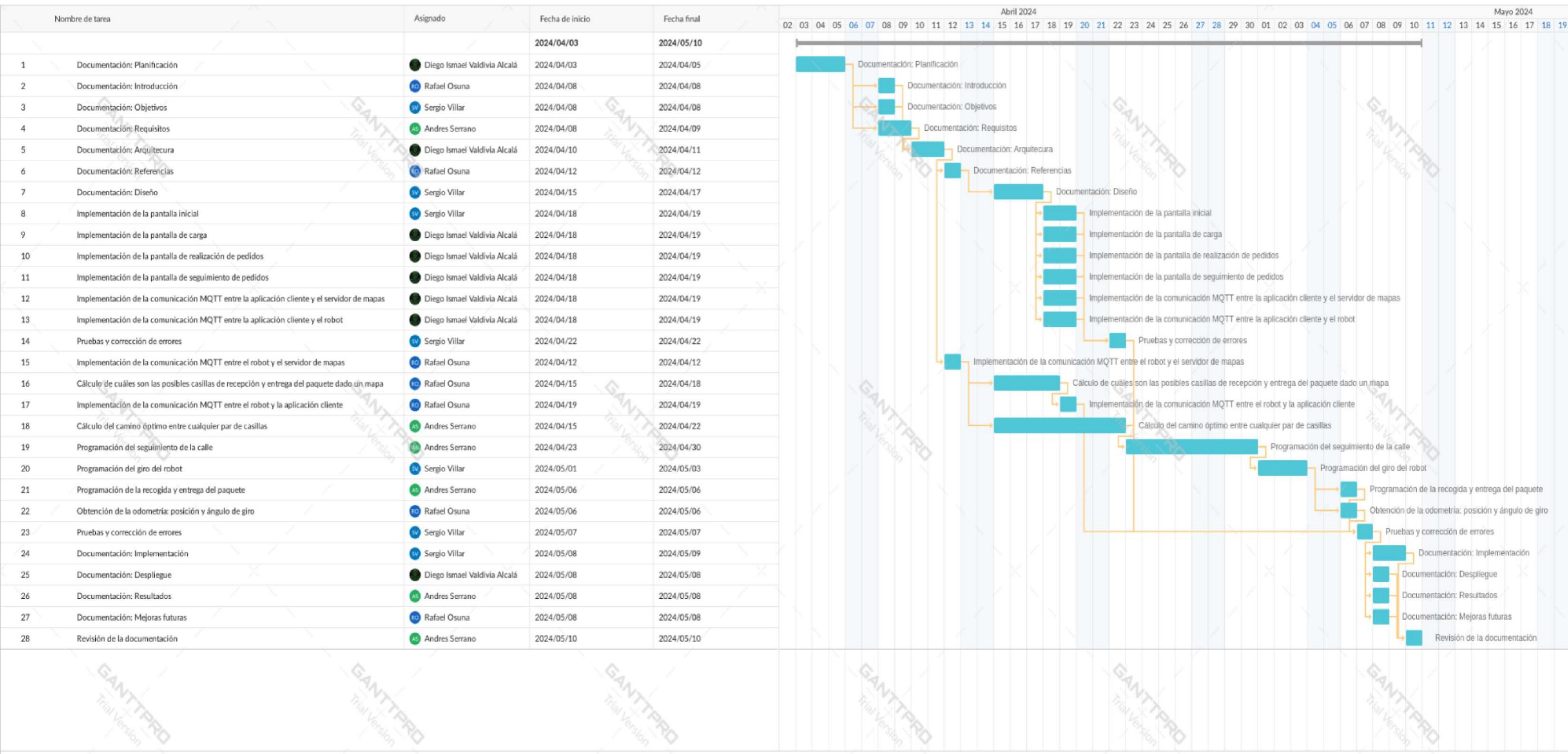


Figura 3. Planificación temporal.

6. ANÁLISIS Y DISEÑO

6.1. DIAGRAMA DE CASOS DE USO

Un diagrama de casos de uso es una herramienta de modelado en el desarrollo de software que representa las interacciones entre usuarios (actores) y un sistema. Ayuda a visualizar cómo los usuarios interactúan con el sistema para lograr ciertos objetivos. En un diagrama de casos de uso, los casos de uso representan las diversas acciones o funciones que un usuario puede realizar en el sistema, mientras que los actores son los roles que interactúan con el sistema. Los casos de uso se presentan como elipses y los actores como figuras externas al sistema, generalmente representados como figuras de palitos. Este tipo de diagrama es muy útil en la etapa de análisis y diseño de sistemas, ya que ayuda a identificar los requisitos funcionales del sistema y a comprender cómo se espera que funcione desde la perspectiva del usuario.

El diagrama de casos de uso donde se refleja el flujo de acciones llevado a cabo en la aplicación es el siguiente:

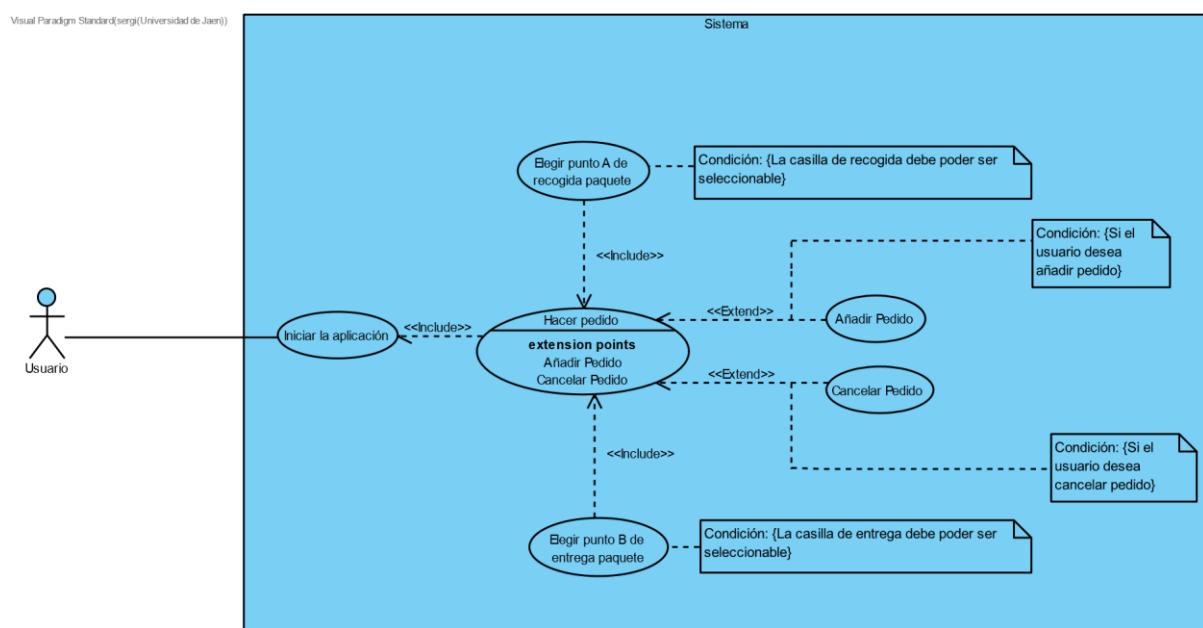


Figura 4. Diagrama de casos de uso.

6.2. GRAMÁTICA DEL CASO DE USO

A continuación, en este apartado, se describirá la gramática de los casos de uso del sistema.

La "gramática de los casos de uso" se refiere a las reglas y convenciones utilizadas para redactar y estructurar los casos de uso en el desarrollo de software y en el diseño de sistemas. Los casos de uso son una técnica para capturar los requisitos funcionales de un sistema, describiendo cómo los usuarios interactúan con este. Se emplean principalmente en el análisis y diseño de sistemas informáticos, pero también pueden utilizarse en cualquier proceso de diseño de interacciones.

En primer lugar, realizaremos una tabla para mostrar el funcionamiento del sistema de la interacción del usuario con la aplicación cliente, mostrando cada una de las acciones que debería de llevar a cabo el usuario para poner en marcha el sistema así como las posibles alternativas para cada caso de uso descrito.

Caso de Uso	Hacer pedido
Actor primario	Usuario
Participantes	Usuario
Sistema	Aplicación de pedidos
Nivel	Usuario
Condición previa	Haber seleccionado algún pedido
Operaciones básicas	
1	Iniciar la aplicación
2	Seleccionar pedido
3	Seleccionar el punto de recogida del paquete
4	Seleccionar el punto de entrega del paquete
5	Añadir pedido
6	Cancelar pedido
Alternativas	
4.A	¿La ruta seleccionada es la correcta?

4.A.1	Si sí, continuar a 5
4.A.2	Si no, continuar a 6
5.A	¿Se va a añadir otro pedido?
5.A.1	Si sí, volver a 3
5.A.2	Si no, salir
6.A	¿Se va a añadir otro pedido?
6.A.1	Si sí, volver a 3
6.A.2	Si no, salir

Tabla 1. Gramática del caso de uso.

6.3. DIAGRAMA DE SECUENCIA

En este apartado, realizaremos un diagrama de secuencia donde intervendrán como objetos principales: la interfaz, el usuario, el servidor de mapas, el broker MQTT y el robot.

Un diagrama de secuencia es una representación gráfica que muestra cómo los objetos interactúan en una aplicación a lo largo del tiempo. Forma parte de la notación UML (Lenguaje Unificado de Modelado), utilizada para modelar sistemas de software. El diagrama de secuencia es muy útil para visualizar el orden de las operaciones y cómo fluye la información entre los componentes del sistema durante un proceso específico.

En el diagrama, se representan los objetos como rectángulos con líneas verticales que cuelgan de ellos, conocidas como líneas de vida. Las interacciones entre los objetos se indican con flechas horizontales que van de un objeto a otro, mostrando el flujo de comunicación mediante mensajes. Esto ayuda a comprender las relaciones dinámicas y la secuencia temporal de las comunicaciones dentro del sistema.

El diagrama de secuencia para este sistema es el siguiente:

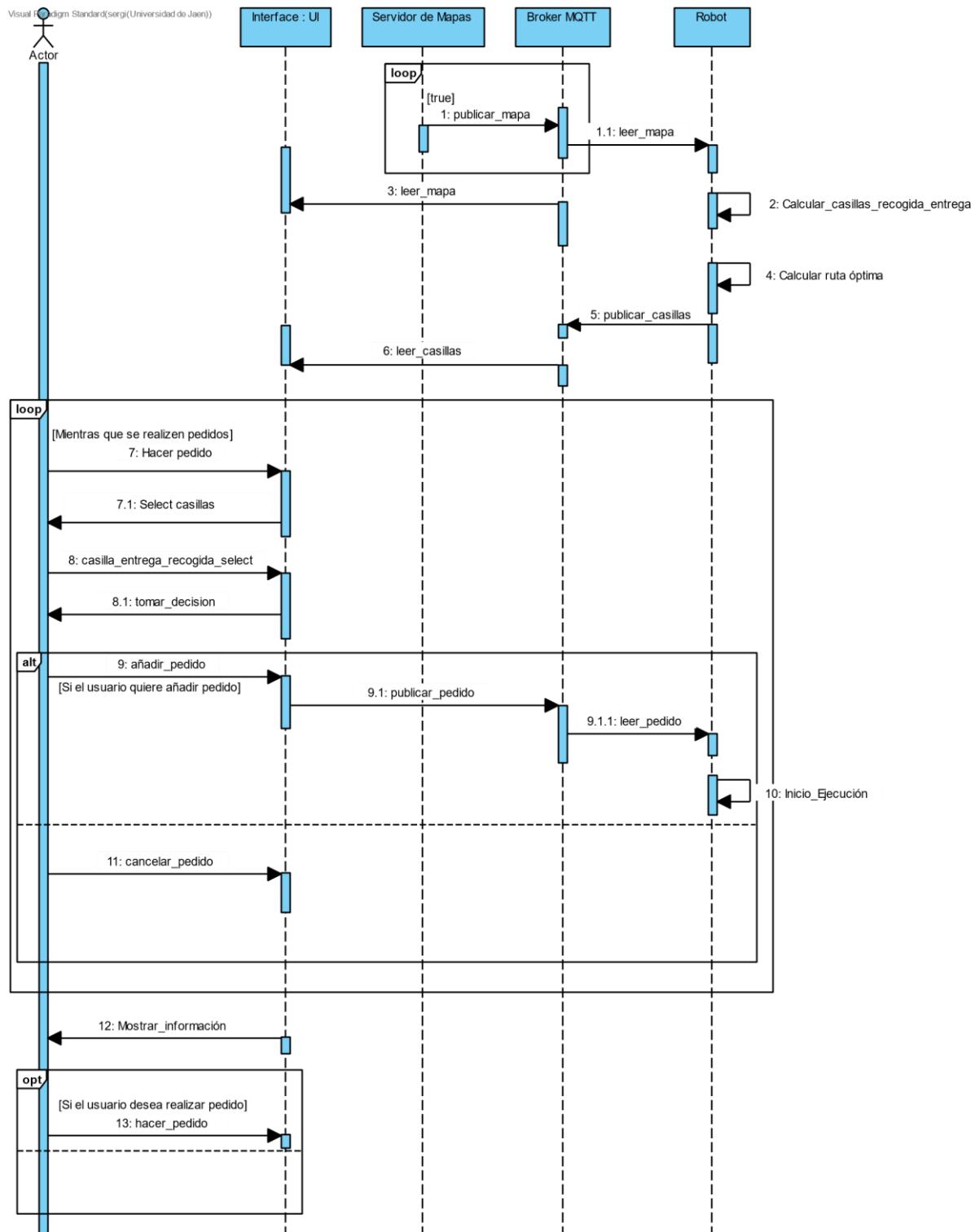


Figura 5. Diagrama de secuencia.

6.4. DIAGRAMA DE ACTIVIDAD

Los diagramas de actividad son una herramienta fundamental en el análisis y diseño de procesos en una variedad de campos, desde la gestión empresarial hasta

el desarrollo de software. Al representar gráficamente el flujo de actividades y decisiones dentro de un proceso, los diagramas de actividad permiten a los equipos visualizar de manera clara y concisa cómo se lleva a cabo un conjunto de tareas. Esto facilita la identificación de áreas de mejora, la optimización de los flujos de trabajo y la comunicación efectiva entre los miembros del equipo.

En un diagrama de actividad, cada actividad se representa como un nodo, mientras que las flechas indican la secuencia de las actividades. Estos diagramas pueden incluir nodos de decisión que bifurcan el flujo de control en diferentes direcciones según ciertas condiciones. Además, los diagramas de actividad pueden mostrar iteraciones, lo que permite representar procesos que involucran repeticiones de actividades hasta que se cumplan ciertas condiciones de salida. Con esta capacidad para capturar tanto la secuencia de actividades como las decisiones que influyen en el proceso, los diagramas de actividad son una herramienta versátil y poderosa para modelar y mejorar procesos en una amplia gama de contextos.

A continuación se presenta el diagrama de actividades correspondiente a este sistema:

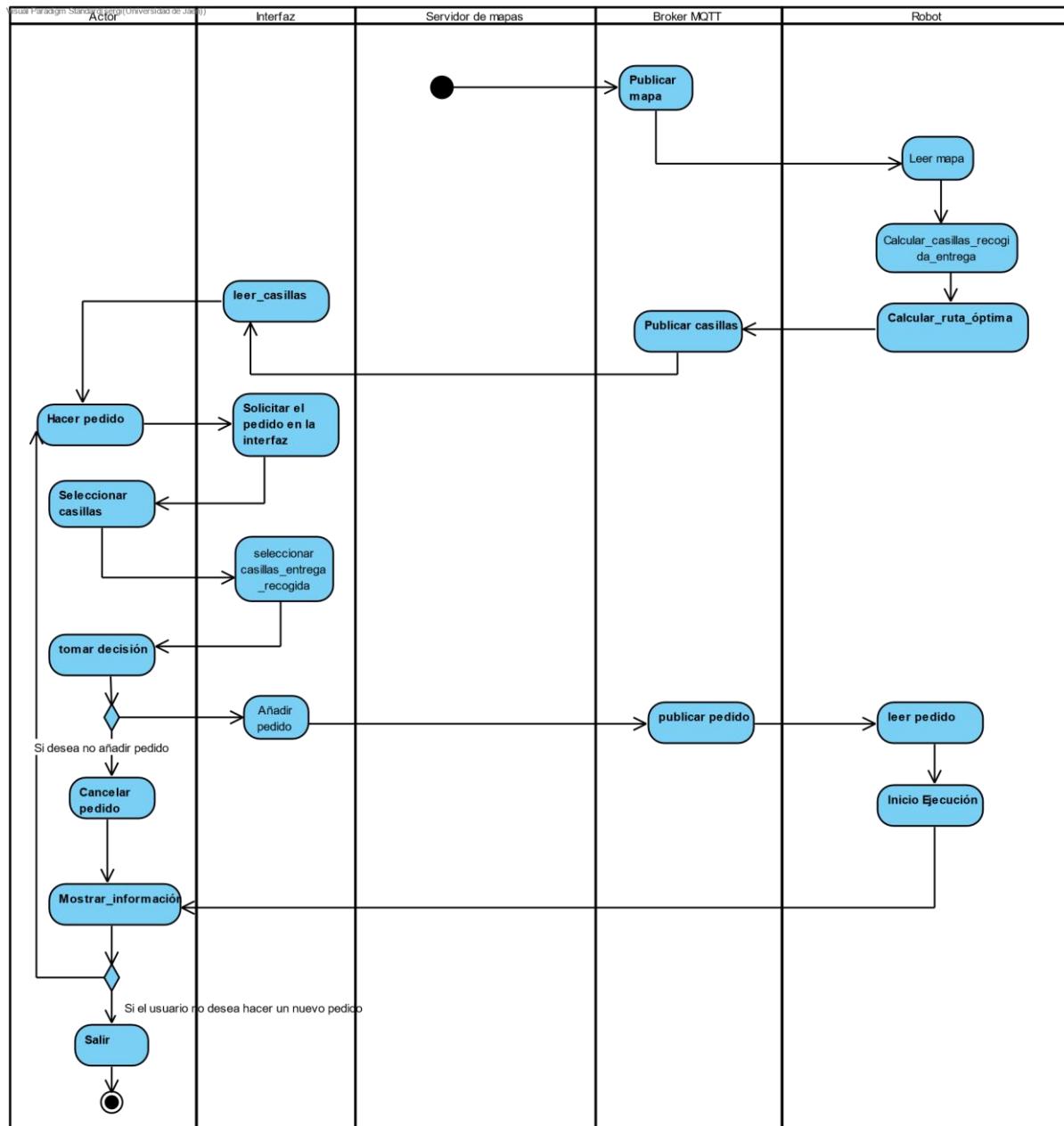


Figura 6. Diagrama de actividad.

6.5. ALGORITMO BFS

Como ya se ha mencionado previamente, el algoritmo que se va a utilizar para calcular las rutas es el BFS [1]. El algoritmo comienza en un nodo inicial y este nodo

se marca como visitado. Es importante llevar un registro de los nodos visitados para evitar procesar el mismo nodo más de una vez y así evitar bucles infinitos en grafos cíclicos. El nodo inicial se añade a una cola, que es fundamental en BFS [1] para gestionar el orden de exploración.

Una vez establecido el nodo inicial, el algoritmo entra en un bucle que sigue mientras haya nodos en la cola. Dentro del bucle, el algoritmo desencola el primer nodo de la cola y realiza alguna operación con él, como registrar su valor o realizar algún cálculo. Luego, se exploran todos los nodos adyacentes de este nodo. Si un nodo adyacente no ha sido visitado aún, se marca como visitado y se encola.

Este proceso se repite hasta que la cola esté vacía, lo que significa que todos los nodos alcanzables desde el nodo inicial han sido visitados y procesados. El uso de una cola asegura que el algoritmo procese los nodos en el orden de su descubrimiento, garantizando que cada nivel se complete antes de pasar al siguiente. El pseudocódigo de este algoritmo es el siguiente:

```
BFS(grafo, nodo_inicial)
    crear cola QUEUE
    marcar nodo_inicial como visitado y encolarlo en QUEUE

    mientras QUEUE no esté vacía
        nodo_actual = QUEUE.desencolar()
        procesar nodo_actual

        para cada nodo_adyacente de nodo_actual
            si nodo_adyacente no ha sido visitado
                marcar nodo_adyacente como visitado
                encolar nodo_adyacente en QUEUE
```

Figura 7. Pseudocódigo del algoritmo BFS.

6.6. DIAGRAMA UML

Presentamos a continuación el diagrama UML diseñado para la aplicación cliente:

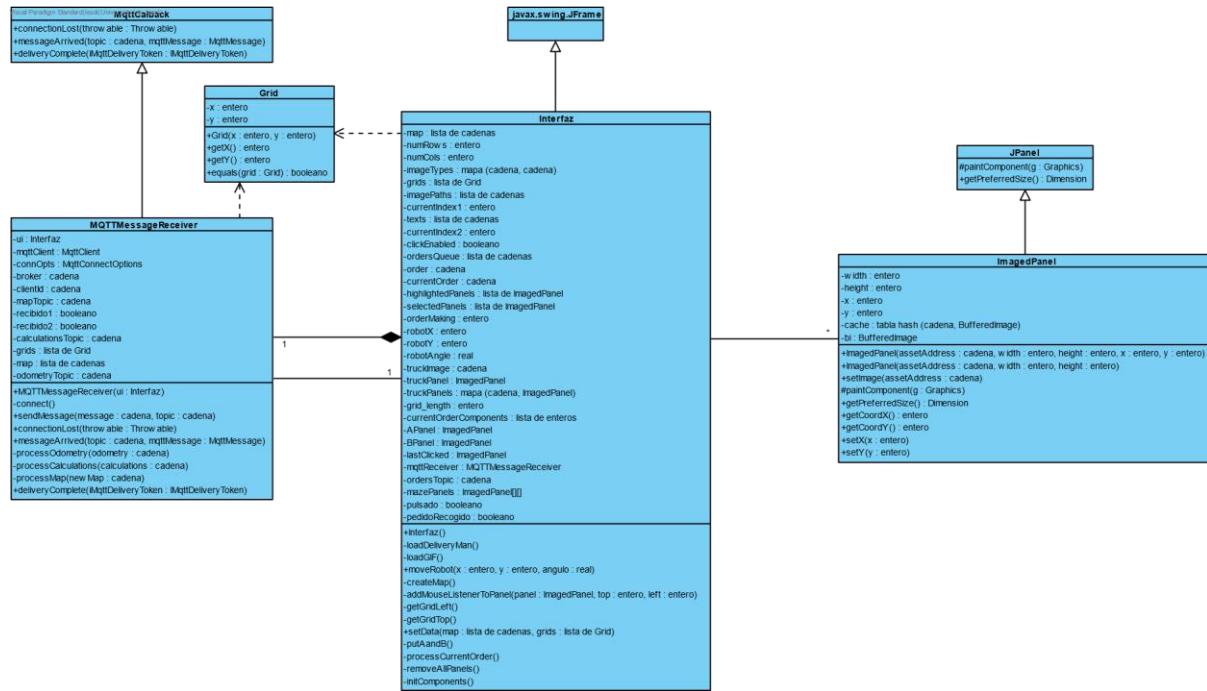


Figura 8. Diagrama UML.

Las principales clases que componen la aplicación cliente son:

- **Interfaz**: subclase de `JFrame` para la creación de ventanas en Java [7]. Representa la interfaz de usuario para realizar y enviar pedidos al robot, mostrando información sobre la odometría y el estado de los pedidos. Se ha establecido una relación de composición con `MQTTMessageReceiver`, ya que la desaparición de la interfaz terminaría las comunicaciones MQTT. Además, hay una relación de asociación con `ImagedPanel` para la representación del mapa, aunque la interfaz podría funcionar sin ella. También existe una dependencia con la clase `Grid` para representar casillas seleccionables.
- **MQTTMessageReceiver**: subclase de `MqttCallback` para manejar comunicaciones MQTT en Java [7]. Gestiona las comunicaciones entre la aplicación cliente y el robot, enviando pedidos y recibiendo/gestionando información. Existe una asociación con `Interfaz`, pero la interfaz seguirá funcionando sin poder recibir mensajes si la interfaz desaparece. También depende de la clase `Grid` para almacenar casillas seleccionables.

- **ImagedPanel:** subclase de JPanel para la representación gráfica del mapa, el robot de reparto y los puntos de recogida y entrega del paquete. Cada panel representa un elemento en el mapa y posee coordenadas para actualizar la representación.
- **Grid:** clase de utilidad para representar casillas seleccionables en el mapa por su posición XY.

6.7. MOCKUPS

Los mockups se han diseñado utilizando la plataforma web Marvel App [3] para implementarlos posteriormente en la apariencia de la aplicación cliente.

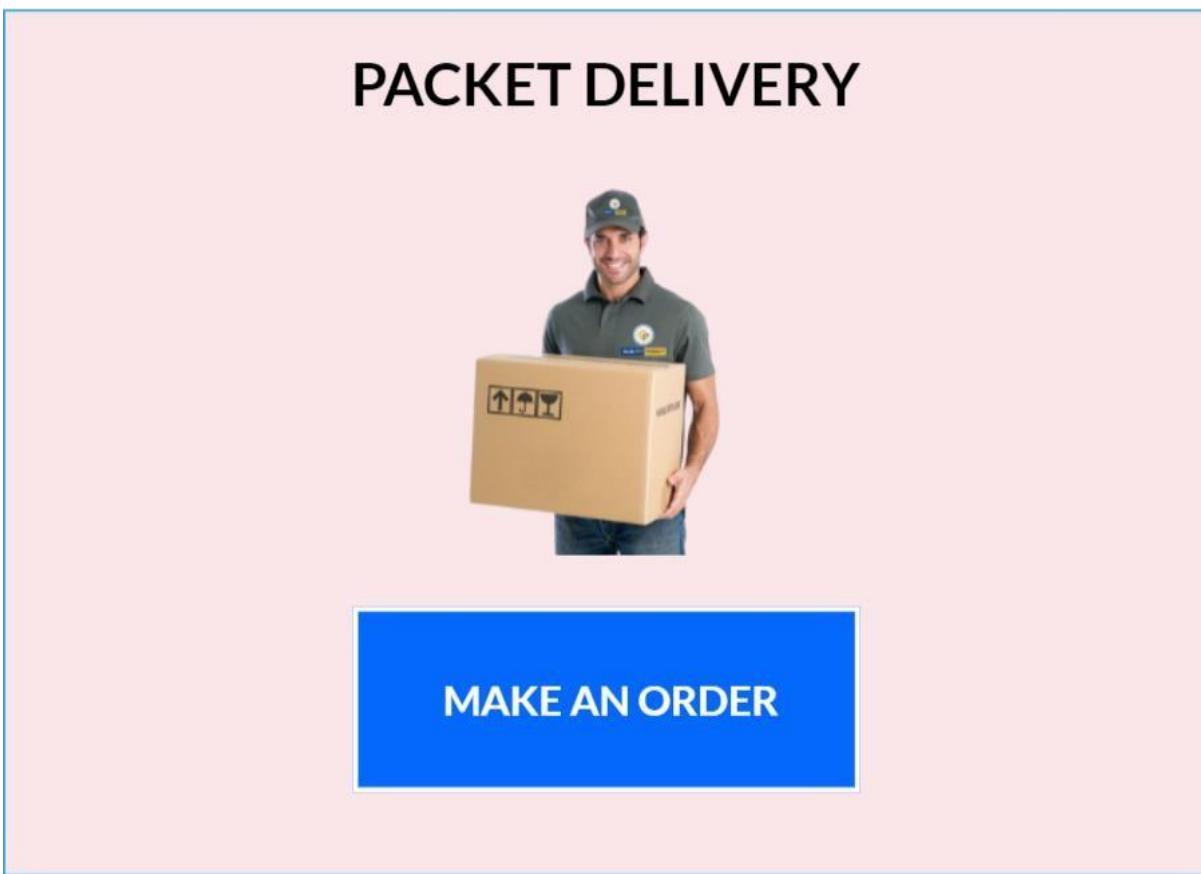


Figura 9. Mockup de la pantalla introductoria.

Esta sería la pantalla que aparecería al iniciar la aplicación cliente. Su función es servir como introducción a la aplicación y permitirnos seleccionar la opción para realizar un pedido.

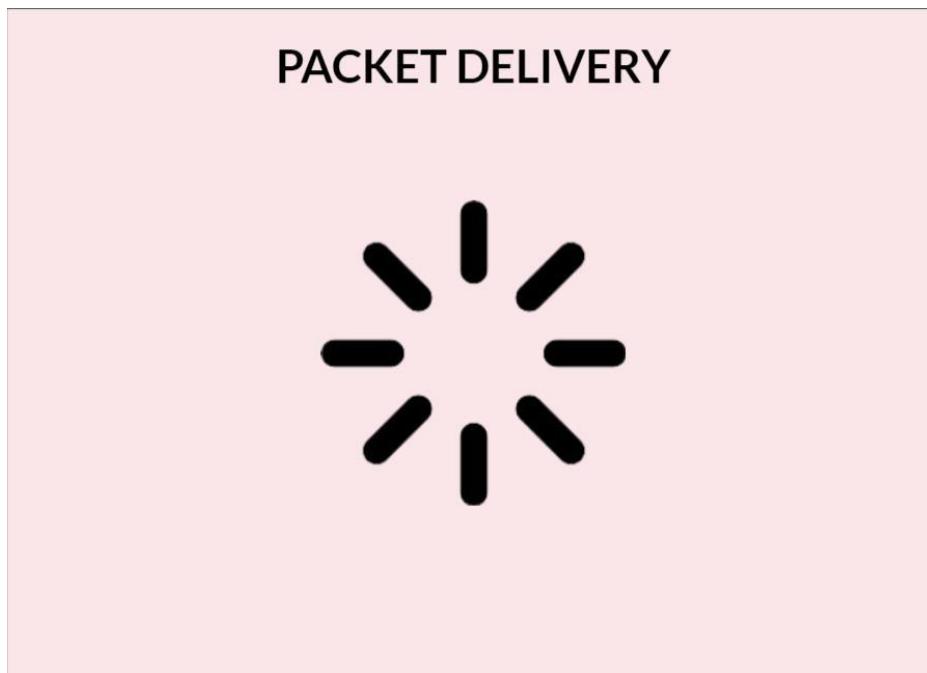
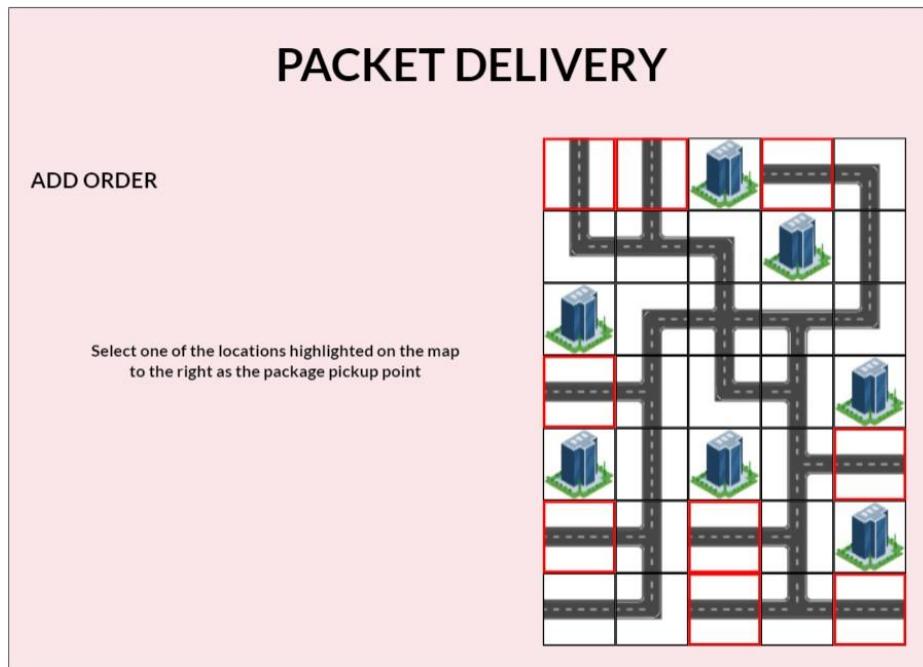


Figura 10. Mockup de la pantalla de carga

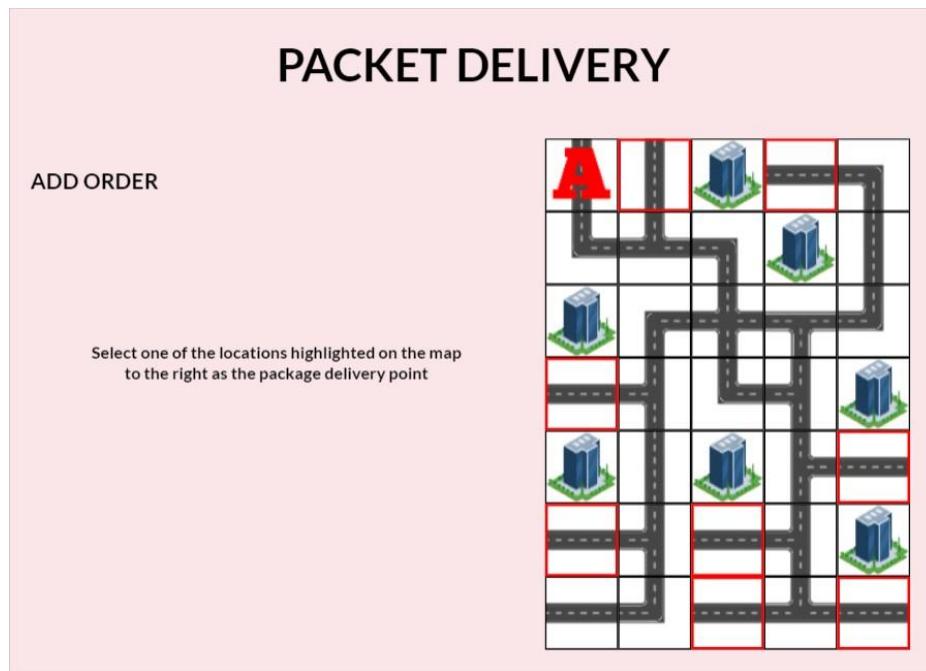
Después de seleccionar la opción para realizar un pedido, nos encontramos con esta pantalla de carga. La aplicación está en espera de recibir ciertos datos, los cuales especificaremos en la sección de implementación.



Figura

11. Mockup de la pantalla de realización de pedidos.

Una vez que la aplicación cliente ha recibido los datos, nos presenta la pantalla de la [Figura 11](#). En esta pantalla, se resaltan varias casillas que representan puntos de recogida del paquete, de entre las cuales podemos seleccionar una.



Figura

12. Mockup de la pantalla de realización de pedidos.

Una vez seleccionado el punto de recogida, la aplicación nos pide que indiquemos el punto de entrega. Es importante tener en cuenta que no nos permite volver a seleccionar el punto de recogida como punto de entrega.

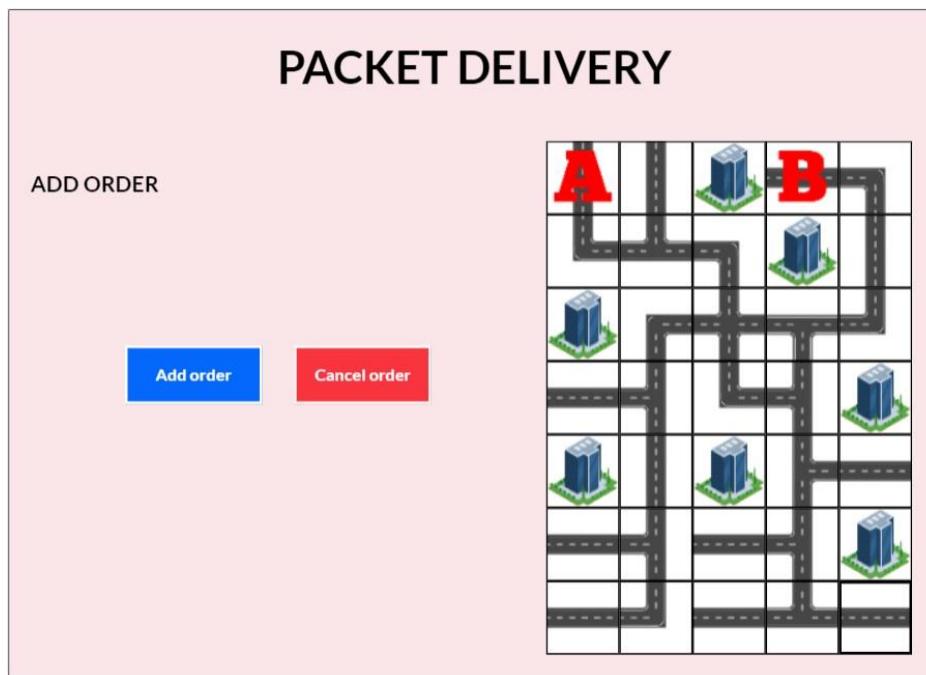


Figura 13. Mockup de la pantalla de realización de pedidos.

Una vez seleccionados los puntos de recogida y entrega, nos encontramos con dos opciones: añadir pedido o cancelar el pedido.

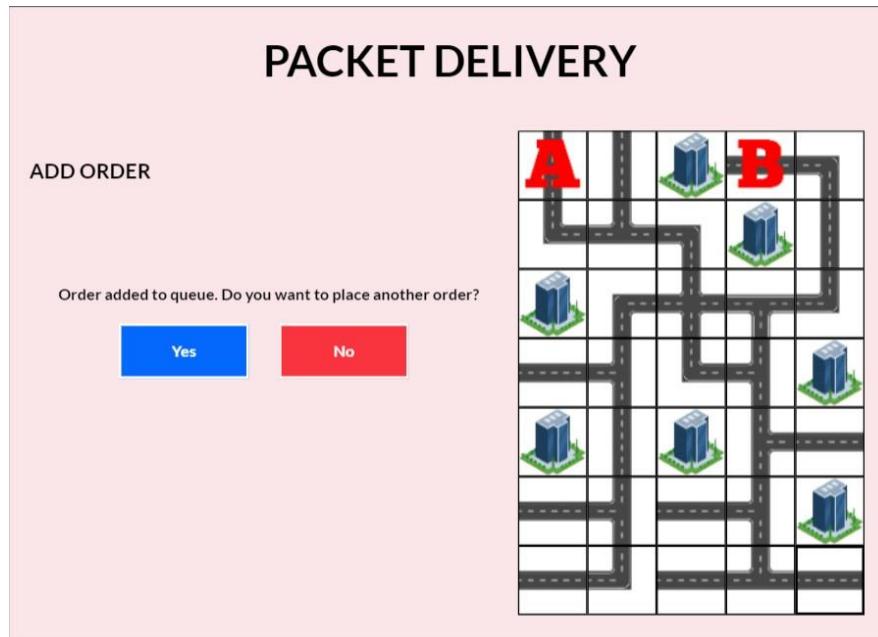


Figura 14. Mockup de la pantalla de realización de pedidos.

Si se añade el pedido, la aplicación nos lo indica con el mensaje correspondiente y nos pregunta si deseamos añadir otro. En caso afirmativo, volveríamos a la pantalla de la [Figura 11](#); de lo contrario, pasaríamos a la pantalla de la [Figura 16](#).

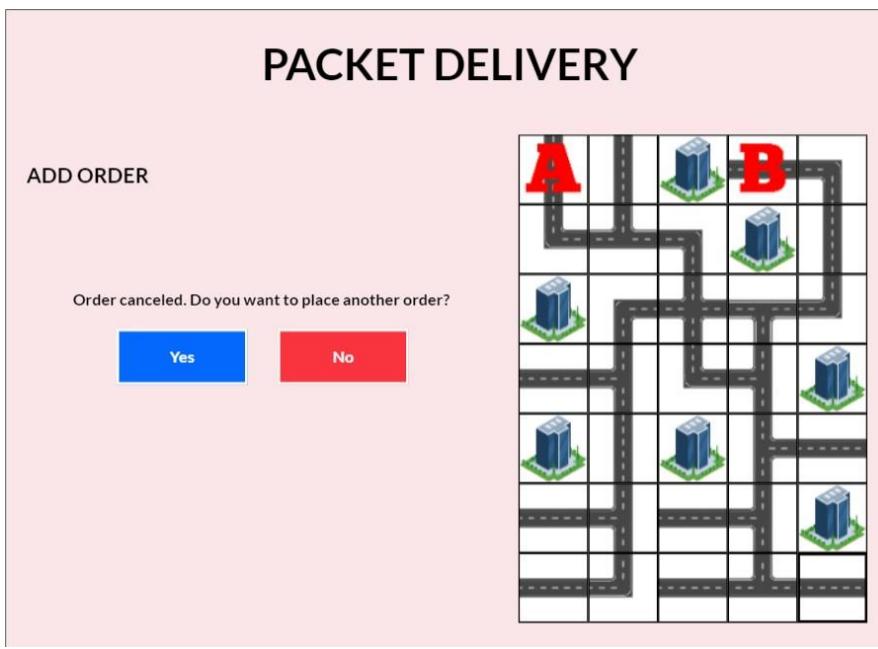


Figura 15. Mockup de la pantalla de realización de pedidos.

Si se cancela el pedido, la aplicación nos muestra el mensaje correspondiente y nos pregunta si deseamos añadir otro pedido. En caso afirmativo, volveríamos a la pantalla de la [Figura 11](#); de lo contrario, pasaríamos a la pantalla de la [Figura 16](#).

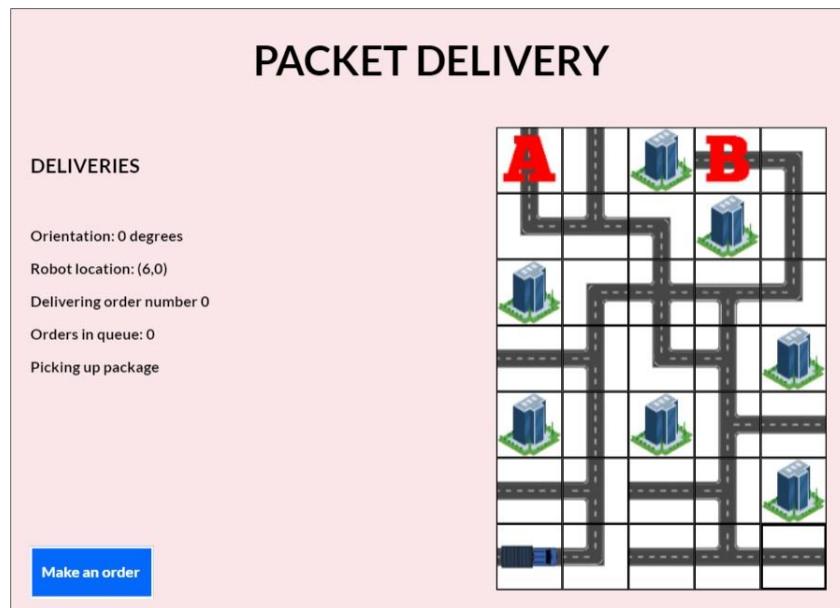


Figura 16. Mockup de la pantalla de entrega de pedidos.

Una vez que hayamos realizado todos los pedidos, accedemos a la pantalla de entrega. En esta pantalla, podemos ver la ubicación en tiempo real del robot repartidor, representado por un camión, junto con los puntos de entrega y recogida. Además, se muestra la orientación del robot, su posición actual, el pedido que está siendo procesado, el número de pedidos en cola y su estado (si está recogiendo o entregando el paquete, inicialmente estaría buscándolo).

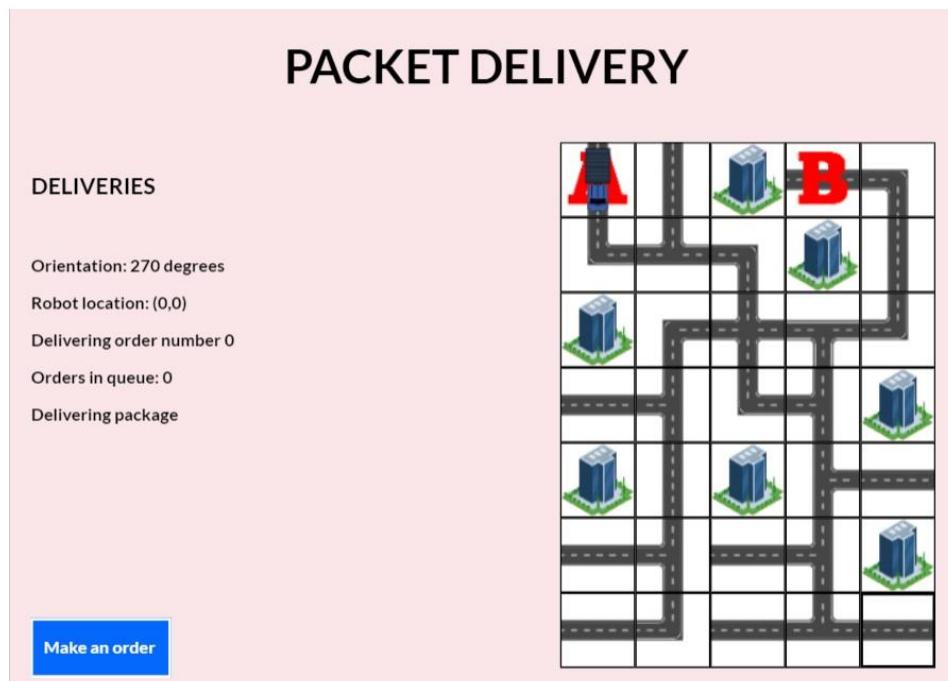


Figura 17. Mockup de la pantalla de entrega de pedidos.

Cuando el robot llega al punto de recogida, la interfaz indica que se está llevando a cabo la entrega del paquete.

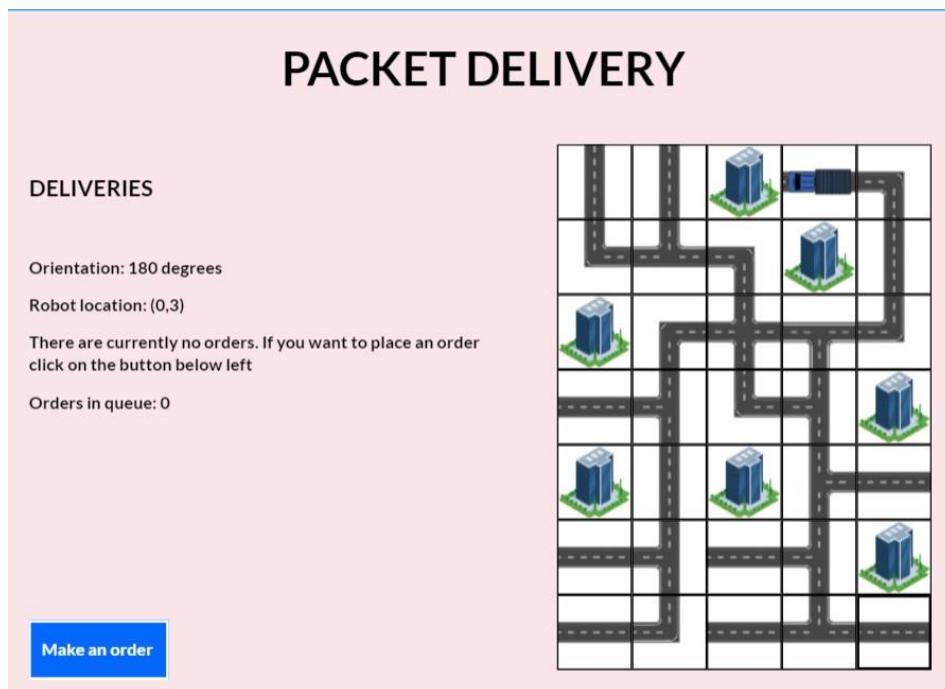


Figura 18. Mockup de la pantalla de entrega de pedidos.

Finalmente, cuando el robot llega al punto de entrega, las letras que indican los puntos de referencia son eliminadas. En este escenario, si no hay más pedidos en cola, se muestra un mensaje indicando que la cola de pedidos está vacía y se ofrece la opción de realizar un nuevo pedido pulsando un botón en la parte inferior. Sin embargo, si hay pedidos pendientes en cola, se mostrarán los puntos de entrega y recogida del próximo pedido.

7. IMPLEMENTACIÓN

7.1. APPLICACIÓN CLIENTE

Como ya se ha mencionado previamente, la aplicación cliente se desarrollará con el entorno de desarrollo NetBeans [6] y utilizando el lenguaje de programación Java [7]. A continuación, se detallarán algunos de los puntos más importantes en su implementación.

7.1.1 COMUNICACIONES

Como se explicó anteriormente en la sección donde se detalla la arquitectura del sistema, todas las comunicaciones entre los distintos componentes se realizarán a través del protocolo MQTT (Message Queuing Telemetry Transport). La aplicación cliente utilizará este protocolo para diversas tareas, como la recepción del mapa del servidor de mapas, las casillas seleccionables de la ciudad, y la odometría por parte del robot, además de enviar los pedidos al mismo.

Para implementar el protocolo MQTT en Java [7], hemos utilizado la librería "org.eclipse.paho.client.mqttv3", desarrollada por Eclipse Paho. Esta librería proporciona a los desarrolladores de Java [7] las herramientas necesarias para conectar y comunicarse con servidores MQTT.

Para gestionar todas estas comunicaciones, hemos implementado la clase MQTTMessageReceiver, que hereda de la clase MqttCallback. Esta clase contiene todos los métodos necesarios para el uso del protocolo MQTT en Java [\[7\]](#).

El código para establecer una conexión con un servidor MQTT es el siguiente:

```
private void connect() throws MqttException {
    mqttClient = new MqttClient(broker, clientId);
    connOpts = new MqttConnectOptions();
    connOpts.setCleanSession(true);
    System.out.println("Conectándose al broker: " + broker);
    mqttClient.connect(connOpts);
    System.out.println("Conexión exitosa");
}
```

Figura 19. Establecimiento de conexión MQTT en Java.

La parte fundamental de este código es la especificación, en forma de cadena de caracteres, tanto de la dirección IP del broker como del identificador del cliente.

Una vez establecida la conexión con el broker, simplemente debemos suscribirnos a los temas que nos interesan para recibir los mensajes y especificar la función que los procesará.

```
mqttClient.setCallback(this);
mqttClient.subscribe(mapTopic);
mqttClient.subscribe(calculationsTopic);
mqttClient.subscribe(odometryTopic);
```

Figura 20. Definición de la función de callback y suscripción.

Dado que la propia clase donde se ejecuta el código hereda de MqttCallback, no es necesario indicar el método que procesará los datos recibidos, ya que se toma por defecto el implementado en la superclase. Por otro lado, podemos observar cómo nos suscribimos a los temas del mapa, las casillas seleccionables y la odometría del robot. Una vez suscritos a los temas necesarios, solo nos queda procesar los datos entrantes. Para ello, redefinimos el método messageArrived() de la clase MqttCallback y procesamos los mensajes entrantes según el tema asociado.

```

@Override
public void messageArrived(String topic, MqttMessage mqttMessage) throws Exception {
    if (topic.equals(mapTopic)) {
        String newMap = new String(mqttMessage.getPayload());
        System.out.println("Received map: " + newMap);
        processMap(newMap);
        mqttClient.unsubscribe(mapTopic);
        recibido1 = true;
        if (recibido2) {
            ui.setData(map, grids);
        }
    } else {
        if (topic.equals(calculationsTopic)) {
            String calculations = new String(mqttMessage.getPayload());
            System.out.println("Calculations: " + calculations);
            processCalculations(calculations);
            mqttClient.unsubscribe(calculationsTopic);
            recibido2 = true;
            if (recibido1) {
                ui.setData(map, grids);
            }
        } else {
            String odometry = new String(mqttMessage.getPayload());
            System.out.println("Odometry: " + odometry);
            processOdometry(odometry);
        }
    }
}
}

```

Figura 21. Gestión de mensajes entrantes.

Una vez que recibamos el mapa de la ciudad, procederemos a procesar la cadena de caracteres que lo representa. Esto implica extraer los elementos de dos en dos para obtener los distintos tipos de casillas que lo componen y almacenarlos en una lista.

```

private void processMap(String newMap) {
    for (int i = 0; i < newMap.length(); i += 2) {
        String substring = newMap.substring(i, i + 2);
        map.add(substring);
    }
}

```

Figura 22. Procesamiento del mapa.

Cuando recibamos las casillas de entrega y recogida, procederemos a procesar la cadena recibida, extrayendo los elementos de dos en dos caracteres. Estas casillas serán almacenadas en una lista de objetos de la clase "Grid", la cual representa una casilla con su posición en los ejes X e Y. Por ejemplo, podríamos recibir " 3 ", lo que indicaría que las casillas , 3, son seleccionables tanto para recogida como para entrega.

```
private void processCalculations(String calculations) {
    for (int i = 0; i < calculations.length(); i += 2) {
        int x = Integer.parseInt(calculations.substring(i, i + 1));
        int y = Integer.parseInt(calculations.substring(i + 1, i + 2));
        grids.add(new Grid(x, y));
    }
}
```

Figura 23. Procesamiento de las casillas recibidas.

Finalmente, cuando recibimos la información sobre la odometría del robot, procedemos a dividir la cadena recibida utilizando el delimitador “;”. En la primera parte de la cadena, obtenemos la posición actual del robot, donde los dos primeros caracteres representan la posición en el eje X y el eje Y respectivamente. La segunda parte de la cadena corresponde al ángulo del robot. Por ejemplo, una información recibida podría ser “ ; ”, donde la posición es (0,6) y el ángulo es 0 grados.

```
private void processOdometry(String odometry) {
    String[] split = odometry.split(";");
    int x = Integer.parseInt(split[0].substring(0, 1));
    int y = Integer.parseInt(split[0].substring(1, 2));
    float angle = Float.parseFloat(split[1]);
    ui.moveRobot(x, y, angle);
}
```

Figura 24. Procesamiento de la odometría.

Por último, para enviar al robot los pedidos realizados por el cliente, creamos una función adicional. Esta función genera un mensaje y lo publica en el topic especificado.

```
public void sendMessage(String message, String topic) throws MqttException {
    if (!mqttClient.isConnected()) {
        connect();
    }
    System.out.println("Enviando mensaje: " + message);
    MqttMessage mqttMessage = new MqttMessage(message.getBytes());
    mqttClient.publish(topic, mqttMessage);
    System.out.println("Mensaje enviado");
}
```

Figura 25. Envío de mensajes.

El formato del pedido enviado al robot será una cadena de 4 caracteres. Los dos primeros indican la posición de recogida del paquete, mientras que los dos últimos indican la posición de entrega en los ejes X e Y.

7.1.2 APARIENCIA

Para la implementación de la interfaz de usuario, se ha utilizado Java Swing [8]. Java Swing [8] es una biblioteca gráfica para Java [7] que permite a los desarrolladores crear interfaces de usuario (UI) ricas y dinámicas para aplicaciones de escritorio. Swing proporciona un conjunto de componentes gráficos, como botones, etiquetas, cuadros de texto, menús desplegables y muchos más, que los desarrolladores pueden utilizar para construir interfaces de usuario interactivas. Además, Swing ofrece una amplia gama de funciones de personalización y permite la creación de interfaces de usuario altamente personalizadas y estéticamente atractivas.

A continuación, se presentará una imagen para cada mockup diseñado, indicando los componentes de Java Swing [8] utilizados para replicarlo.

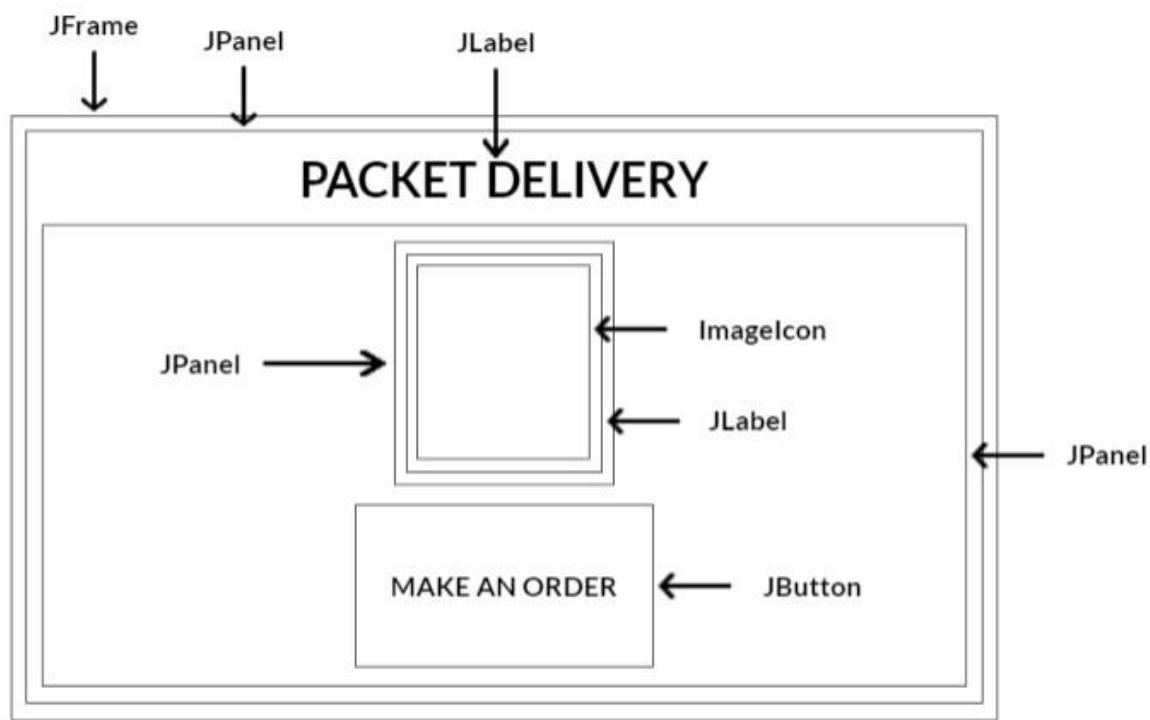


Figura 26. Implementación de la pantalla introductoria.

La base de la interfaz es un JFrame, una clase de Java Swing [8] que representa una ventana en la interfaz gráfica de una aplicación Java [7]. Esta ventana puede contener diversos componentes gráficos, como botones, etiquetas y campos de texto, y ofrece funcionalidades como maximizar, minimizar, redimensionar y cerrar.

la ventana, así como manejar eventos del usuario, como clics del ratón o acciones de teclado. La ventana tiene un tamaño definido de 1550 x 830 píxeles.

Dentro del JFrame, añadimos un JPanel, un contenedor sobre el cual podemos colocar otros componentes como texto, botones y desplegables. Usaremos un JLabel dentro de este JPanel para mostrar el título de la aplicación, que será común a todas las vistas.

Bajo el título de la aplicación, colocaremos otro JPanel cuyo aspecto cambiará según la interacción del usuario. Para lograr esto, lo configuraremos en modo "Card Layout", lo que nos permitirá situar varios paneles encima, de modo que se ajusten a su tamaño y podamos mostrar u ocultar el que necesitemos.

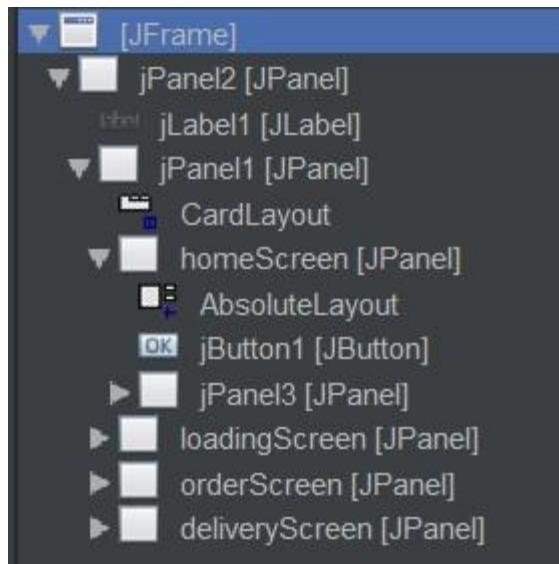


Figura 27. Jerarquía de elementos JPanel.

Dentro del JPanel correspondiente a esta vista, agregaremos otro JPanel donde se mostrará la imagen del repartidor. Utilizaremos un JLabel que cargará el ImageIcon del repartidor para visualizarlo.

El resultado de implementar esto es el siguiente:

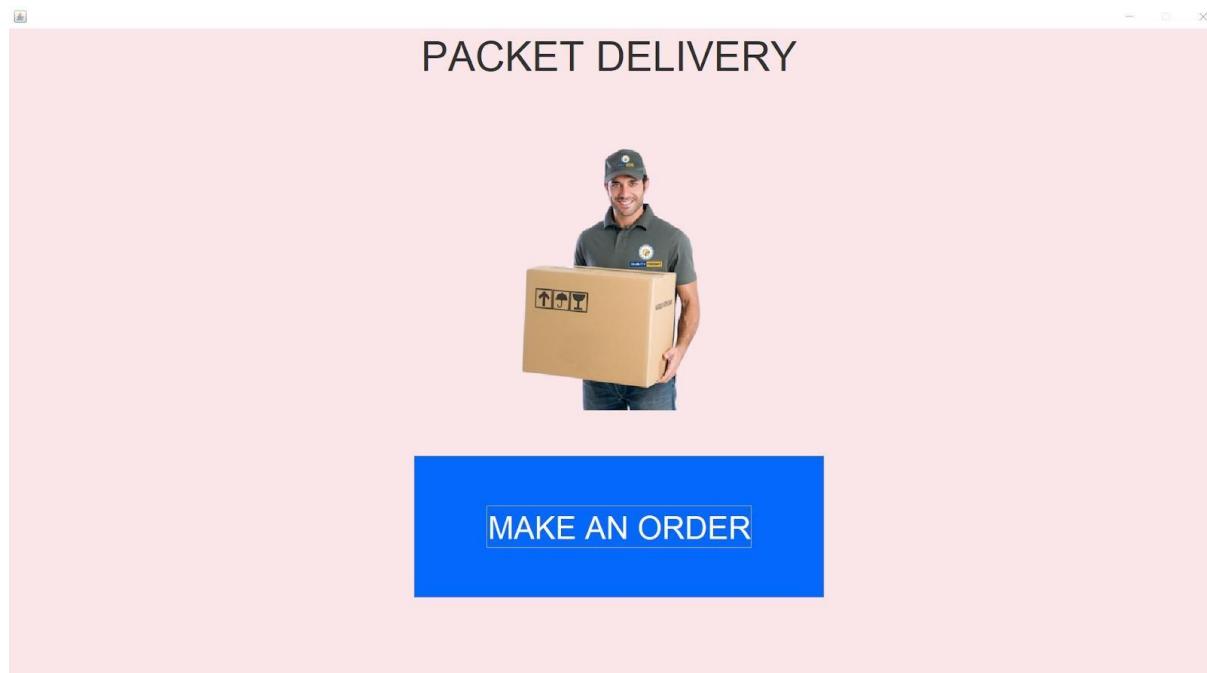


Figura 28. Resultado de la implementación de la pantalla introductoria.

La siguiente pantalla implementada sería la de carga:

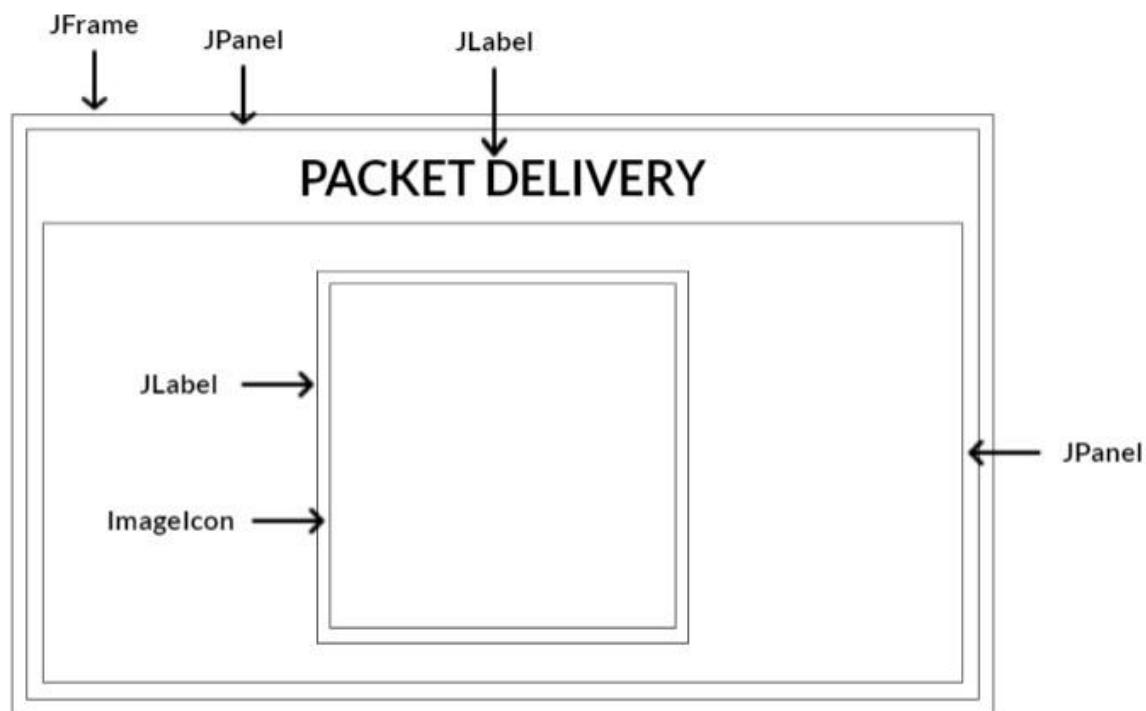


Figura 29. Implementación de la pantalla de carga.

Para mostrar esta vista, simplemente ocultamos el panel de la pantalla inicial y mostramos este nuevo panel.

```
homeScreen.setVisible(false)
```

Figura 30. Ocultación de la pantalla inicial.

```
loadingScreen.setVisible(true)
```

Figura 31. Activación de la pantalla de carga.

En esta vista, se presenta un GIF con un ícono de carga. Utilizamos un JLabel que cargue el ImageIcon del GIF para lograr este efecto.

El resultado de implementar esto es el siguiente:



Figura 32. Resultado de la implementación de la pantalla de carga.

Esta pantalla solo se activará cuando la aplicación cliente esté esperando recibir tanto el mapa de la ciudad como las posibles casillas de recogida y entrega calculadas por el robot. Una vez que ambos conjuntos de datos sean recibidos, se avanzará a la siguiente pantalla. Sin embargo, si estos datos son recibidos antes de que el usuario pulse el botón para realizar un pedido en la pantalla inicial, esta pantalla no se mostrará.

La siguiente vista implementada es la de la realización de un pedido:

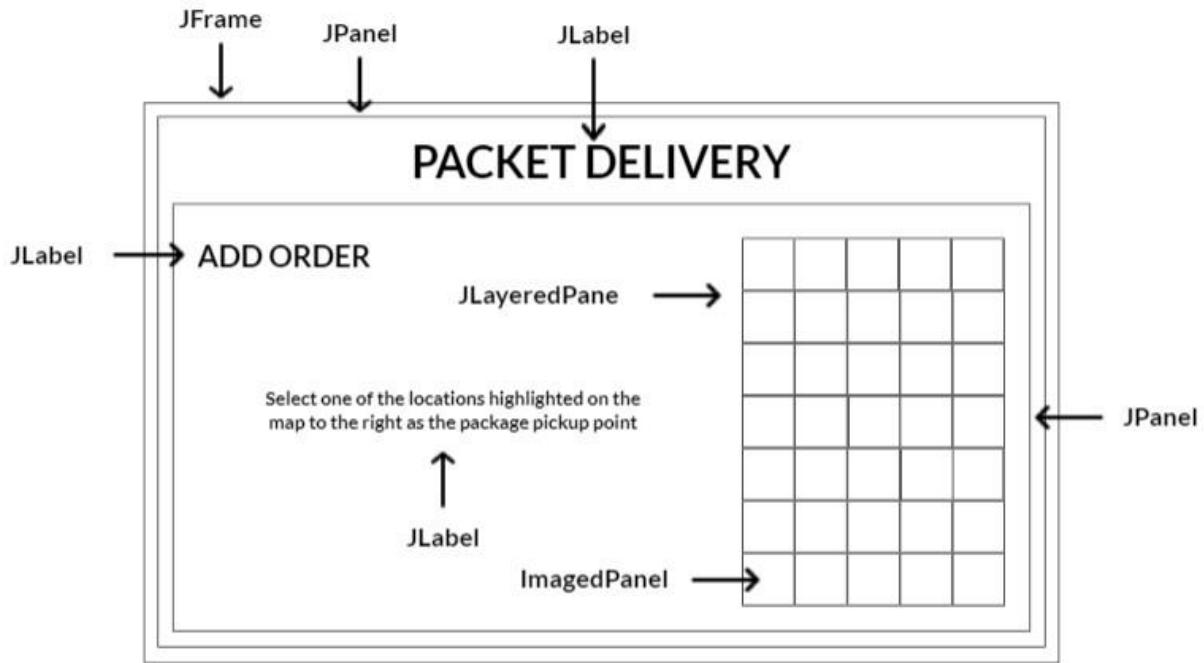


Figura 33. Implementación de la pantalla de realización de pedidos.

Una vez más, para mostrar esta pantalla, ocultamos el último panel activo y activamos el correspondiente. En esta pantalla, todo el texto se presenta utilizando la clase JLabel. Sin embargo, lo más relevante aquí es la representación del mapa de la ciudad. Para lograrlo, utilizamos la clase JLayeredPane, que es un contenedor que permite organizar elementos en diferentes capas a lo largo del eje z para superponerlos.

Para representar el mapa recibido, utilizamos un diccionario de Java [7] que nos proporciona la imagen correspondiente para cada tipo de casilla.

```
imageTypes = new HashMap();
imageTypes.put("00", "assets/image0.jpg");
imageTypes.put("01", "assets/image1.jpg");
imageTypes.put("02", "assets/image2.jpg");
imageTypes.put("03", "assets/image3.jpg");
imageTypes.put("04", "assets/image4.jpg");
imageTypes.put("05", "assets/image5.jpg");
imageTypes.put("06", "assets/image6.jpg");
imageTypes.put("07", "assets/image7.jpg");
imageTypes.put("08", "assets/image8.jpg");
imageTypes.put("09", "assets/image9.jpg");
imageTypes.put("10", "assets/image10.jpg");
imageTypes.put("11", "assets/image11.jpg")
```

Figura 33. Mapa para obtener la imagen de una casilla según su tipo.

Una vez que hemos obtenido la imagen, la colocamos en el JLayeredPane utilizando un ImagePanel. Además, hemos recibido información del robot sobre qué posiciones son seleccionables. Por lo tanto, añadimos un evento de clic del ratón a estos paneles seleccionables para poder saber si se han seleccionado o no y poder almacenar los pedidos. También les hemos aplicado un borde rojo para resaltar que son seleccionables. Todos estos paneles que conforman la ciudad los almacenamos en una matriz para tener una referencia a ellos posteriormente.

```

int i = 0;
for (int x = 0; x < numRows; x++) {
    for (int y = 0; y < numCols; y++) {
        Grid grid = new Grid(x, y);
        String assetAddress = imageTypes.get(map.get(i++));
        ImagedPanel panel1, panel2;
        int top = getGridTop(y);
        int left = getGridLeft(x);
        try {
            panel1 = new ImagedPanel(assetAddress, grid_length, grid_length, x, y);
            panel2 = new ImagedPanel(assetAddress, grid_length, grid_length, x, y);
            panel1.setBounds(top, left, grid_length, grid_length);
            panel2.setBounds(top, left, grid_length, grid_length);
            mazePanels[x][y] = panel2;
            boolean salir = false;
            for (int j = 0; j < grids.size() && !salir; j++) {
                Grid otherGrid = grids.get(j);
                if (otherGrid.equals(grid)) {
                    salir = true;
                }
            }
            if (salir) {
                panel1.setBorder(BorderFactory.createLineBorder(Color.RED, 2));
                highlightedPanels.add(panel1);
                addMouseListenerToPanel(panel1, top, left);
            }
            jLayeredPane1.add(panel1);
            jLayeredPane2.add(panel2);
        } catch (IOException ex) {
        }
    }
}

```

Figura 34. Representación del mapa.

El resultado de implementar esto es el siguiente:

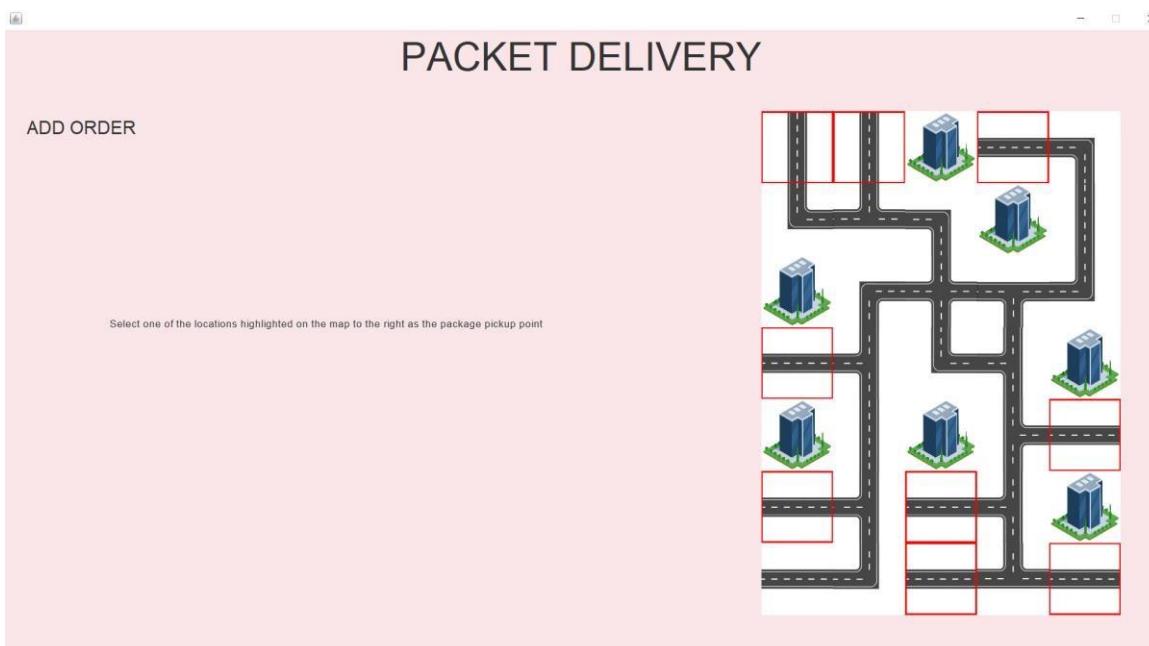


Figura 35. Resultado de la implementación de la pantalla de realización de pedidos.

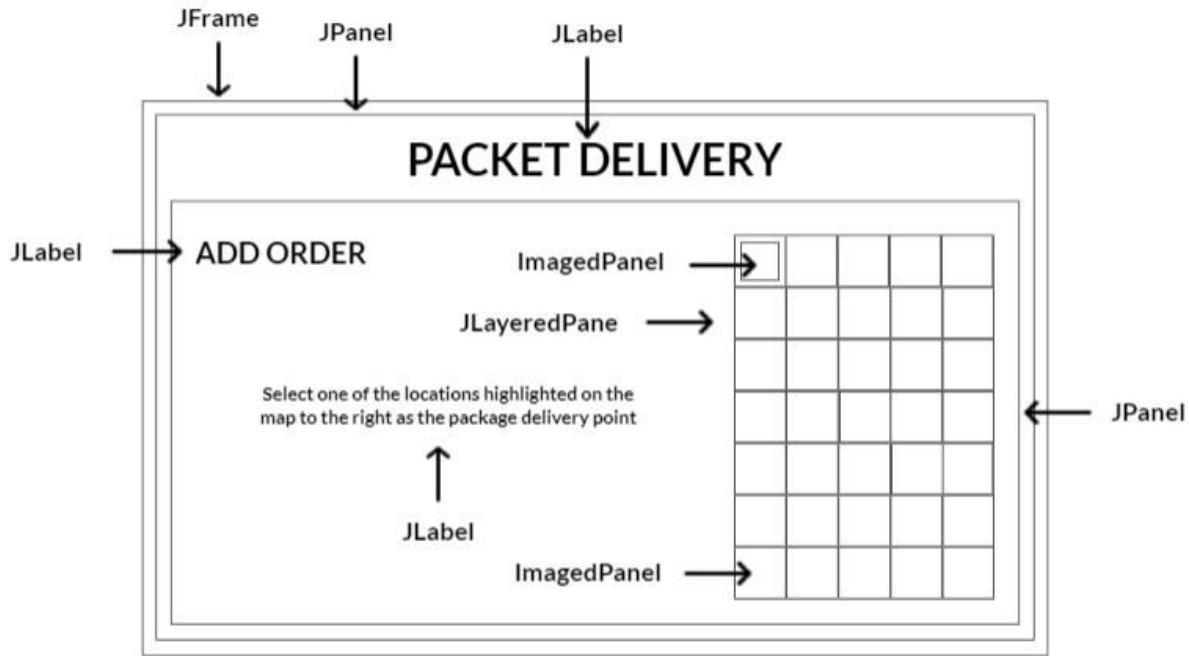


Figura 36. Implementación de la pantalla de realización de pedidos.

Una vez que se selecciona el punto de recogida, procedemos a superponer un ImagedPanel sobre el panel inicialmente colocado. Esto se logra utilizando el método `moveToFront(panel)` de la clase `JLayeredPane`. Cuando una casilla se selecciona, se activa su evento de teclado correspondiente, el cual la hace no seleccionable y elimina su borde rojo. Es importante mencionar que la imagen que colocamos es un PNG con fondo transparente para que la casilla de fondo sea visible.

```
letterPanel = new ImagedPanel(imagePaths.get(currentIndex1), grid_length, grid_length);
currentIndex1 = (currentIndex1 + 1) % imagePaths.size();
if (currentIndex1 == 0) {
    for (ImagedPanel comp : highlightedPanels) {
        comp.setBorder(null);
    }
    lastClicked = null;
    clickEnabled = false;
    jButton2.setVisible(true);
    jButton3.setVisible(true);
}
letterPanel.setOpaque(false);
letterPanel.setBounds(top, left, grid_length, grid_length);
jLayeredPane1.add(letterPanel);
jLayeredPane1.moveToFront(letterPanel);
selectedPanels.add(letterPanel);
```

Figura 37. Implementación de la selección del punto de recogida.

Como se puede observar, se emplea un contador llamado "currentIndex" que distingue si se ha seleccionado el punto de recogida o de entrega.

El resultado de implementar esto es el siguiente:

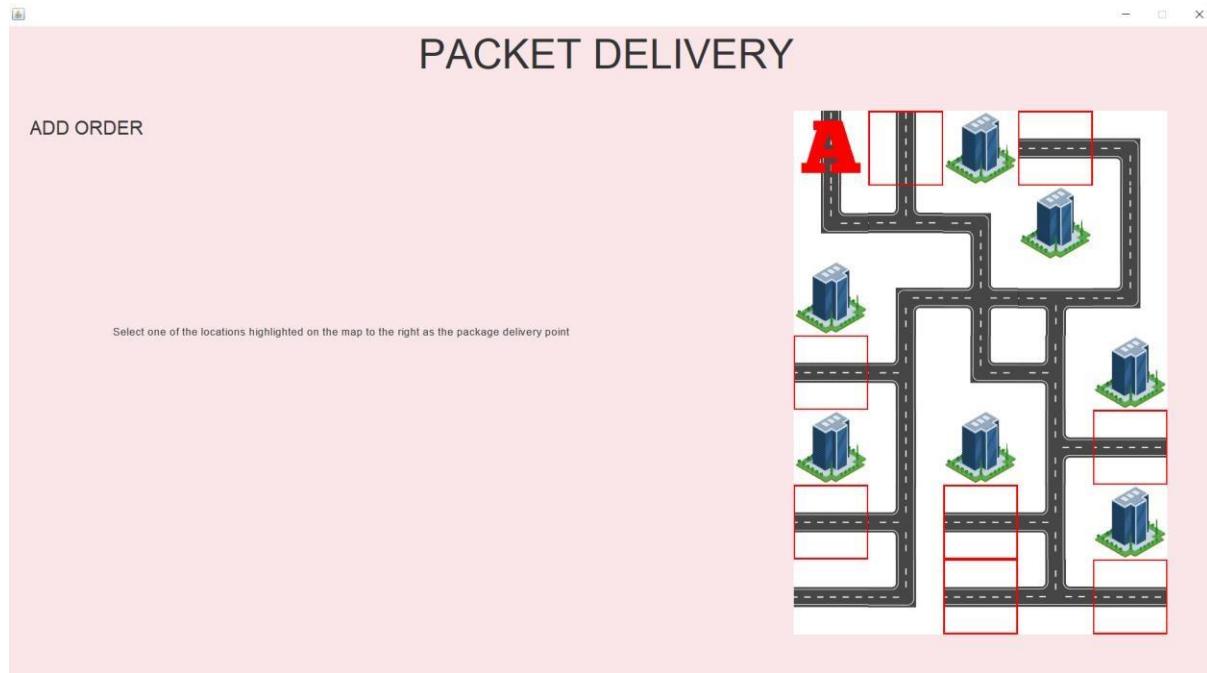


Figura 38. Resultado de la implementación de la pantalla de realización de pedidos.

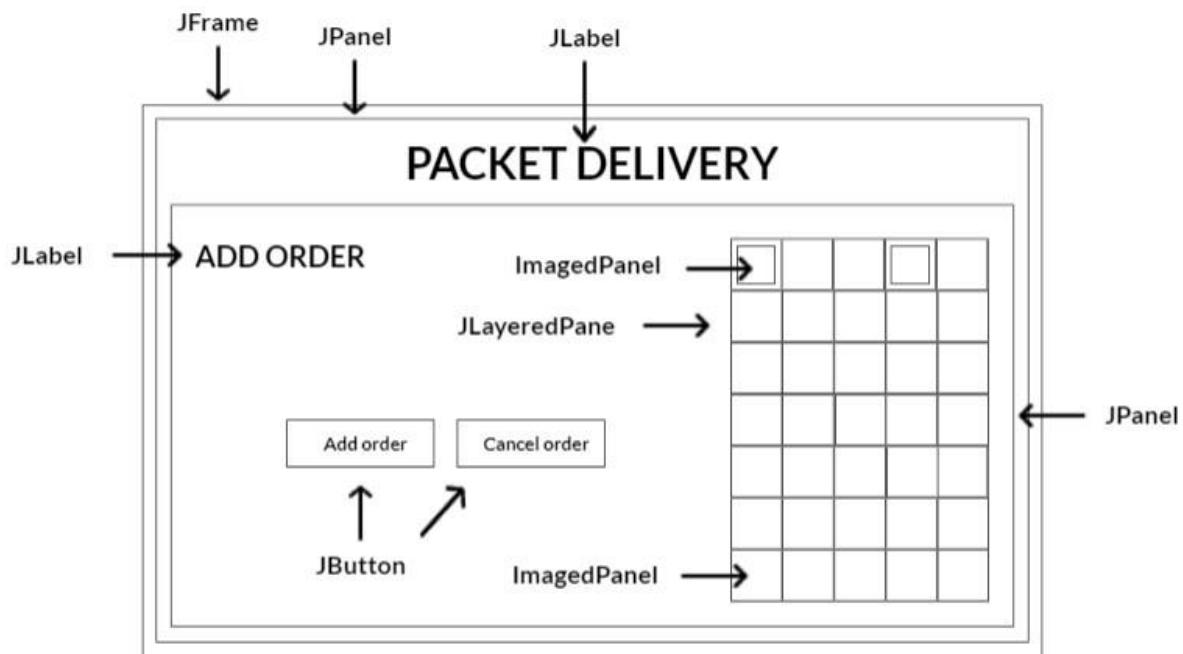


Figura 39. Implementación de la pantalla de realización de pedidos.

Cuando se selecciona el punto de entrega del paquete, se superpone otro icono, como se ha ilustrado en la [Figura 37](#). Además, en este punto se eliminan todos

los bordes rojos de las posibles casillas de recogida o entrega. Esto se logra gracias al uso de la variable “current node”, como se presenta en la [Figura 37](#).

Además, al seleccionar la casilla de entrega, se muestran dos botones (JButton): uno para almacenar y enviar el pedido al robot y otro para cancelarlo. Estos botones estaban ocultos inicialmente, pero aparecen cuando se selecciona la casilla de entrega, como se muestra en la [Figura 37](#).

El resultado de implementar esto es el siguiente:

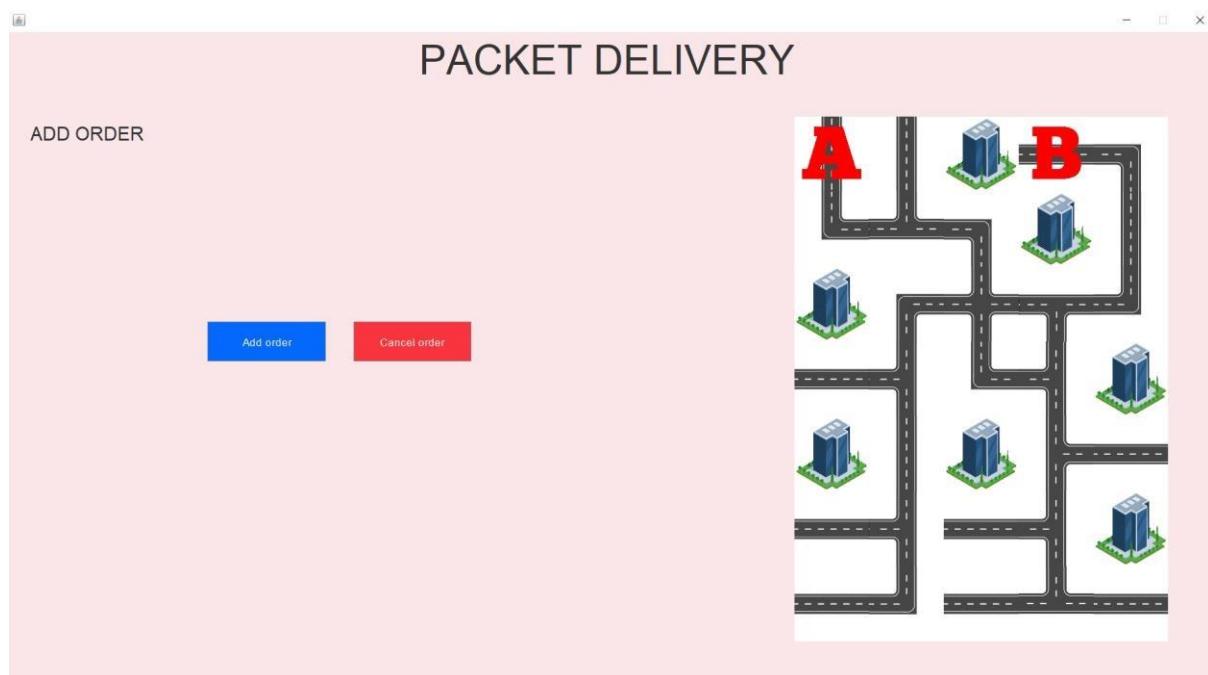


Figura 40. Resultado de la implementación de la pantalla de realización de pedidos.

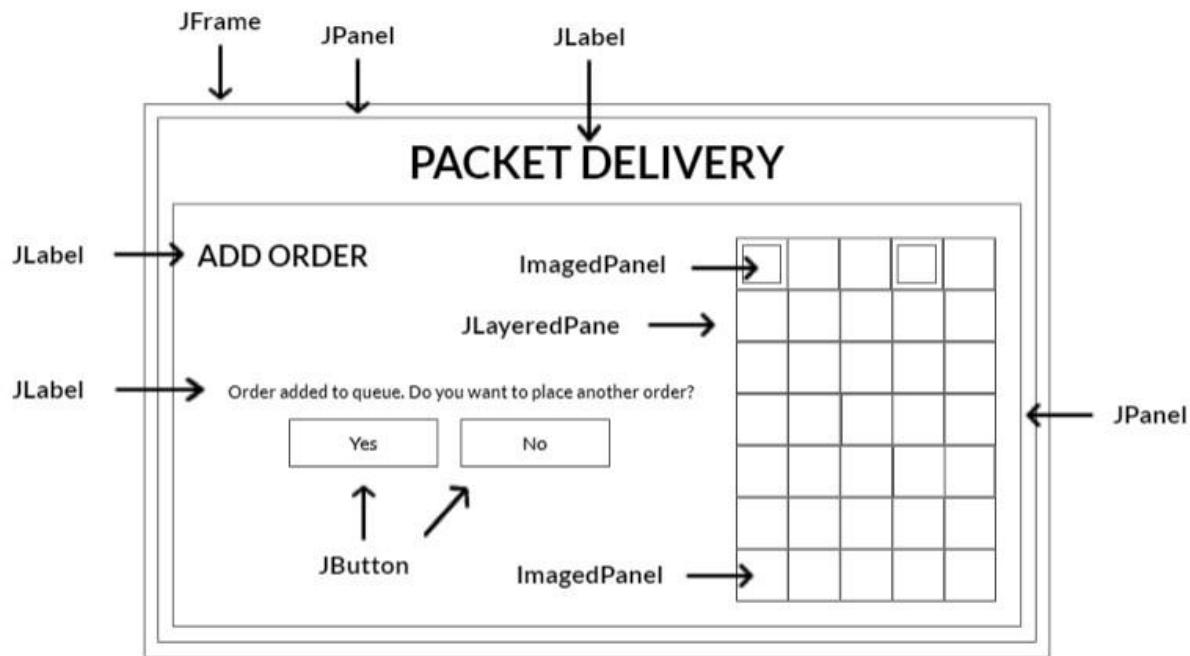


Figura 41. Implementación de la pantalla de realización de pedidos.

Después de añadir el pedido a la cola y enviarlo al robot, los botones se ocultan y aparecen otros para indicar si se desea realizar otro pedido o no.

El aspecto de esto es el siguiente:

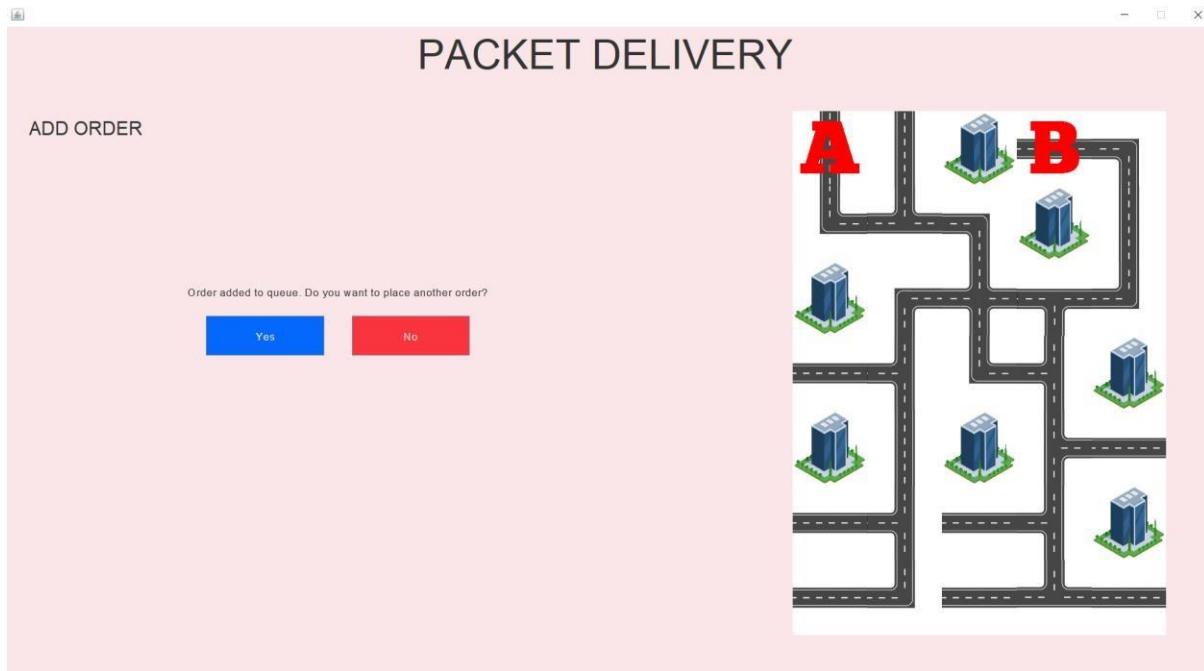


Figura 42. Resultado de la implementación de la pantalla de realización de pedidos.

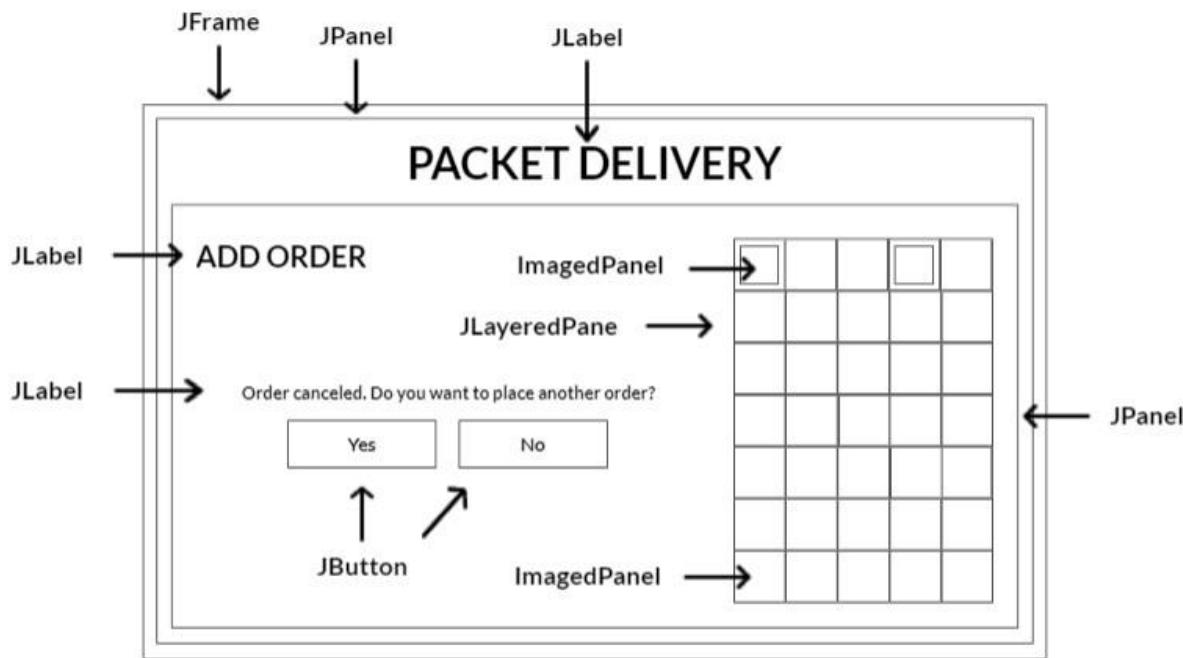


Figura 43. Implementación de la pantalla de realización de pedidos.

Al cancelar el pedido, los botones se ocultan y aparecen otros dos para indicar si se desea realizar un nuevo pedido.

El resultado de esto es el siguiente:

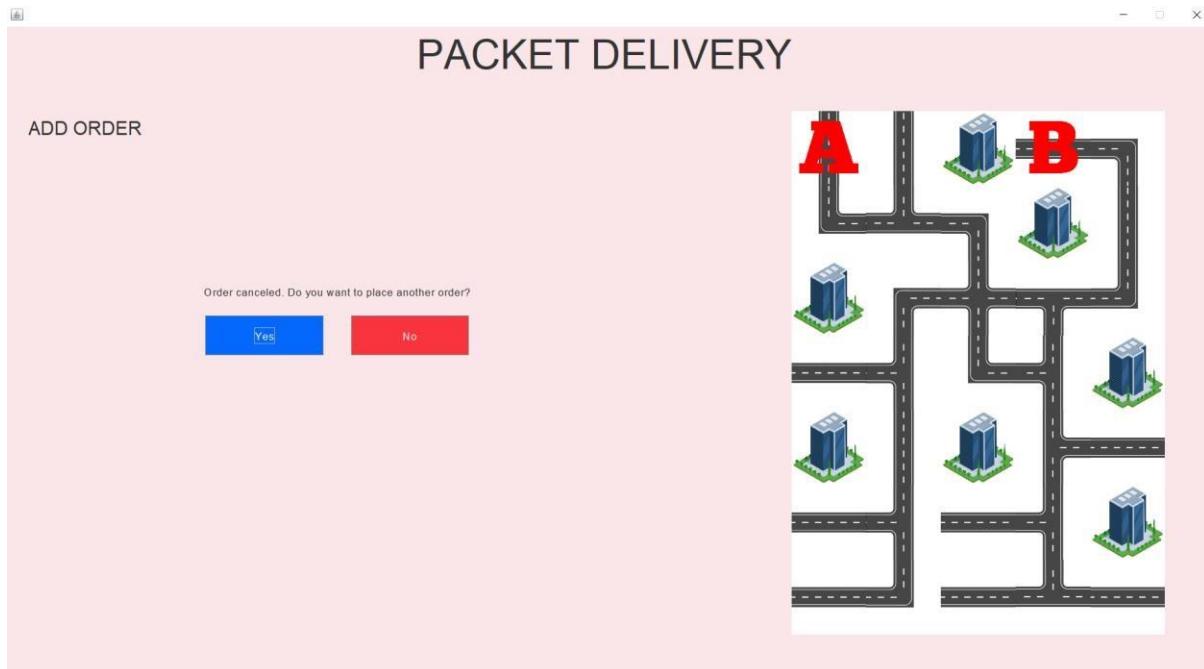


Figura 44. Resultado de la implementación de la pantalla de realización de pedidos.

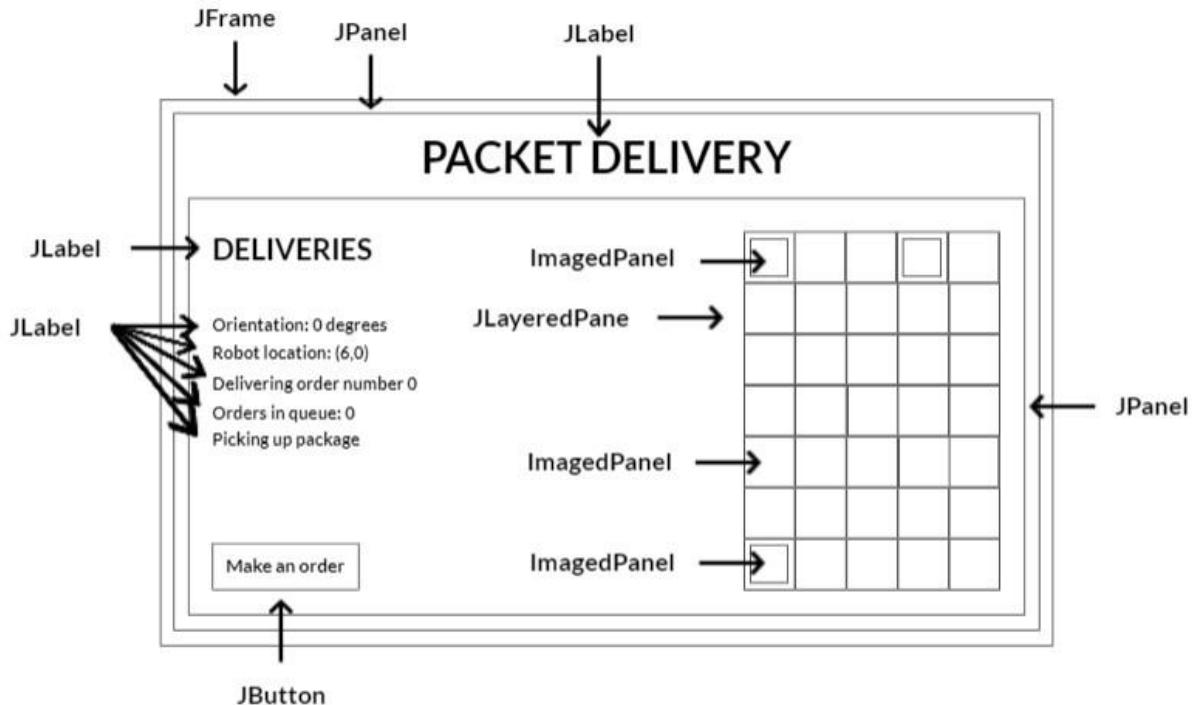


Figura 45. Implementación de la pantalla de entrega de pedidos.

Cuando se presiona el botón que indica que no se desea realizar un nuevo pedido, el panel de la pantalla de realización de pedidos se oculta y aparece el panel de entrega.

```
private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
    orderScreen.setVisible(false);
    if (ordersQueue.isEmpty() && currentOrder == null) {
        jLabel6.setText("There are currently no orders. If you want to");
    } else {
        jLabel6.setText("Delivering order number " + orderMaking);
    }
    jLabel7.setText("Orders in queue: " + ordersQueue.size());
    deliveryScreen.setVisible(true);
    removeAllPanels();
}
```

Figura 46. Activación de la pantalla de visualización de entregas.

En esta pantalla, se mostrará un nuevo JLabeledPane que será una copia del anterior, del cual hemos hablado antes, pero sin elementos seleccionables (por eso es una copia). Aquí se visualizará información en tiempo real sobre el pedido que se está entregando, la odometría del robot, entre otros datos. Dado que el robot envía su odometría cada segundo, actualizamos la información de la interfaz con esta frecuencia.

```

public void moveRobot(int x, int y, float angle) {
    if (angle < 0) {
        angle = 360 + angle;
    } else {
        if (angle > 360) {
            angle = angle % 360;
        }
    }
    if (x != robotX || y != robotY || angle != robotAngle) {
        robotX = x;
        robotY = y;
        if (angle != robotAngle) {
            robotAngle = angle;
            if ((robotAngle >= 315 && robotAngle <= 359) || (robotAngle >= 0
                truckImage = "assets/truck_right.png";
            } else {
                if (robotAngle >= 45 && robotAngle < 135) {
                    truckImage = "assets/truck_up.png";
                } else {
                    if (robotAngle >= 135 && robotAngle < 225) {
                        truckImage = "assets/truck_left.png";
                    } else {
                        truckImage = "assets/truck_down.png";
                    }
                }
            }
        }
    }
    jLabel9.setText("Robot location: (" + robotX + "," + robotY + ")");
    jLabel10.setText("Orientation: " + robotAngle + " degrees");
}

```

Figura 47. Actualización de la odometría en la interfaz.

Como se puede observar, en todo momento realizamos un seguimiento de la posición y orientación del robot. Si la información recibida no coincide con la que tenemos, la actualizamos en la interfaz. Además, seleccionamos una imagen específica del camión que representa al robot según su orientación. Una vez obtenemos la imagen correspondiente a la orientación del robot, la superponemos en el mapa de la ciudad utilizando el método `moveToFront(panel)` de la clase `JLayeredPane`.

```
this.truckPanels = new HashMap();
ImagedPanel rightPanel, upPanel, leftPanel, downPanel;
try {
    rightPanel = new ImagedPanel("assets/truck_right.png", grid_length, grid_length);
    rightPanel.setOpaque(false);
    truckPanels.put("assets/truck_right.png", rightPanel);
    upPanel = new ImagedPanel("assets/truck_up.png", grid_length, grid_length);
    upPanel.setOpaque(false);
    truckPanels.put("assets/truck_up.png", upPanel);
    leftPanel = new ImagedPanel("assets/truck_left.png", grid_length, grid_length);
    leftPanel.setOpaque(false);
    truckPanels.put("assets/truck_left.png", leftPanel);
    downPanel = new ImagedPanel("assets/truck_down.png", grid_length, grid_length);
    downPanel.setOpaque(false);
    truckPanels.put("assets/truck_down.png", downPanel);
```

Figura 48. Mapa con la imagen del camión según su orientación.

Este es un diccionario que nos permite obtener la imagen adecuada del camión según su orientación.

```
truckPanel = truckPanels.get(truckImage);
truckPanel.setX(robotX);
truckPanel.setY(robotY);
int top = getGridTop(robotY);
int left = getGridLeft(robotX);
truckPanel.setBounds(top, left, grid_length, grid_length);
jLayeredPane2.add(truckPanel);
jLayeredPane2.moveToFront(truckPanel);
```

Figura 49. Obtención y colocación de la imagen del camión según su orientación.

Además, queremos resaltar que se ha agregado otro JButton en la parte inferior izquierda de la interfaz para permitir la adición de otro pedido en cualquier momento.

El resultado de implementar esto es el siguiente:

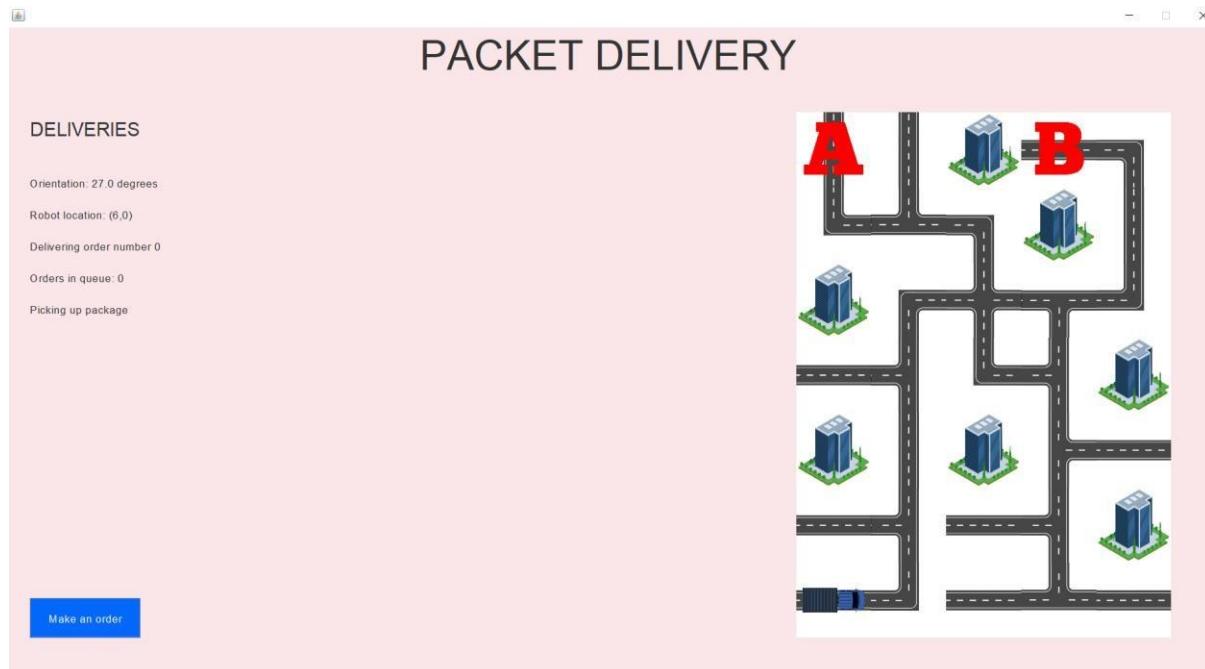


Figura 50. Resultado de la implementación de la pantalla de entrega de pedidos.

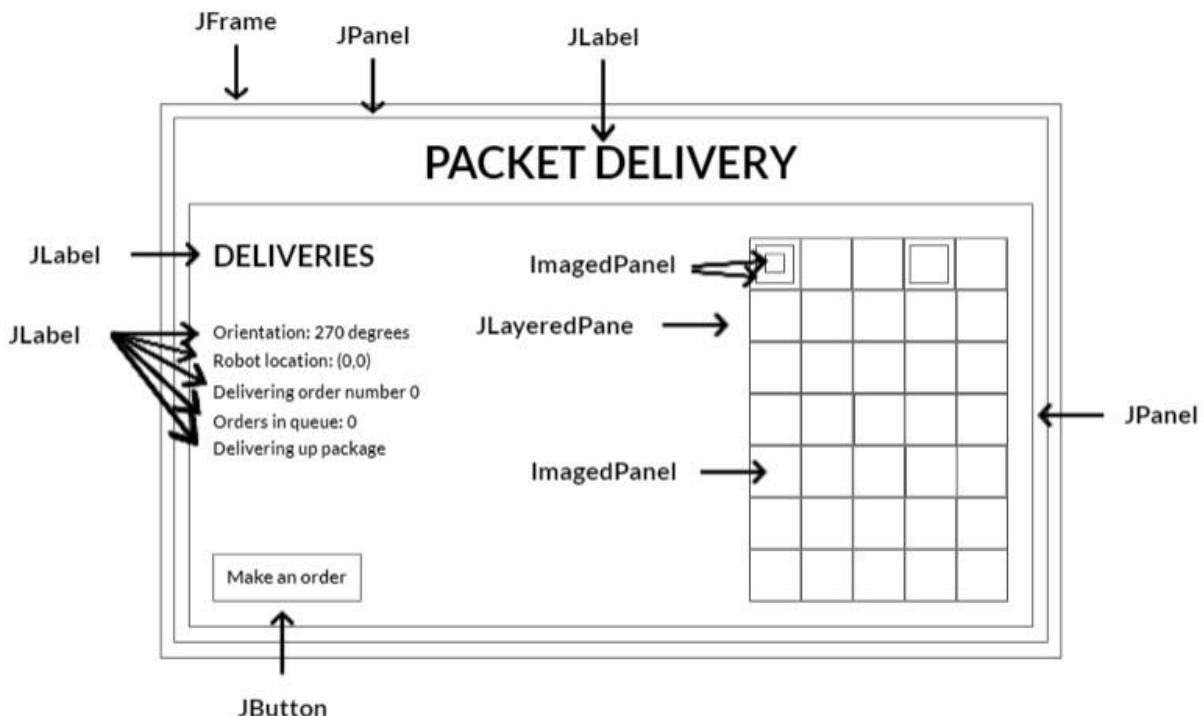


Figura 51. Implementación de la pantalla de entrega de pedidos.

En esta parte, se observa que cuando el robot llega a la casilla de recogida, se indica en la interfaz que comienza la entrega del paquete (se acaba de recoger). Además, el robot se encuentra sobre la casilla recogida, por lo que es necesario superponerlo en dos paneles: el de la ciudad y el indicativo de la casilla de recogida.

Esto se logra nuevamente con el método `moveToFront(panel)` de la clase `JLayeredPane`.

El resultado de implementar esto es el siguiente:

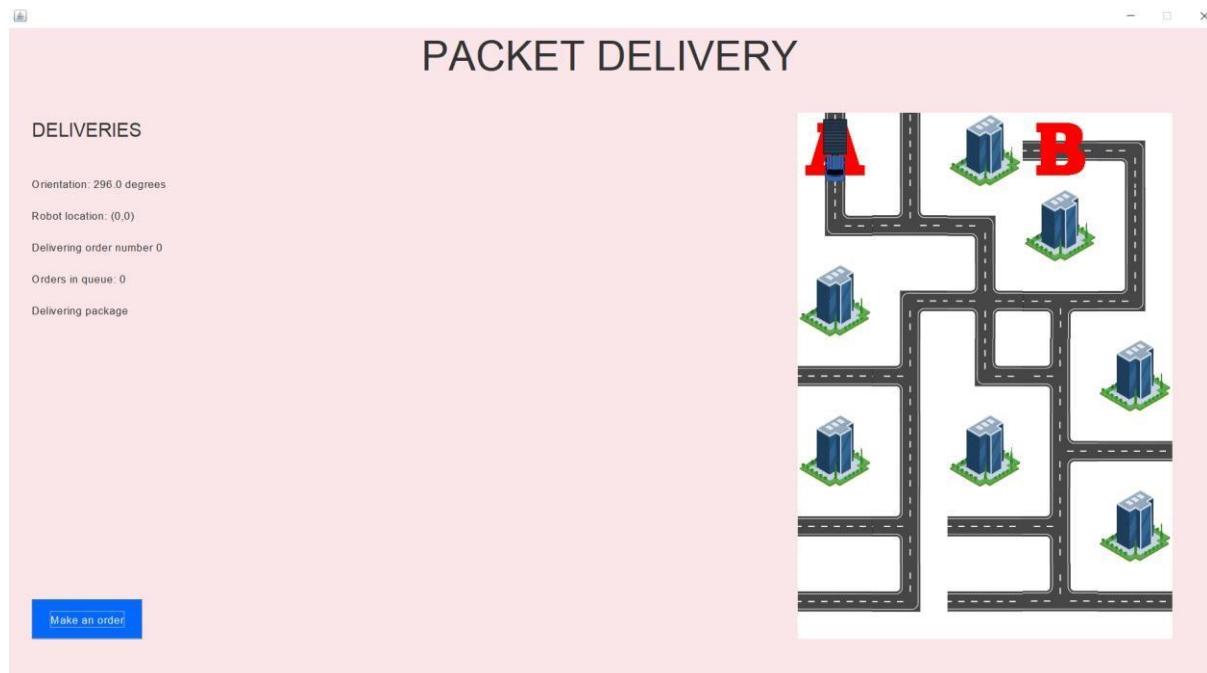


Figura 52. Resultado de la implementación de la pantalla de entrega de pedidos.

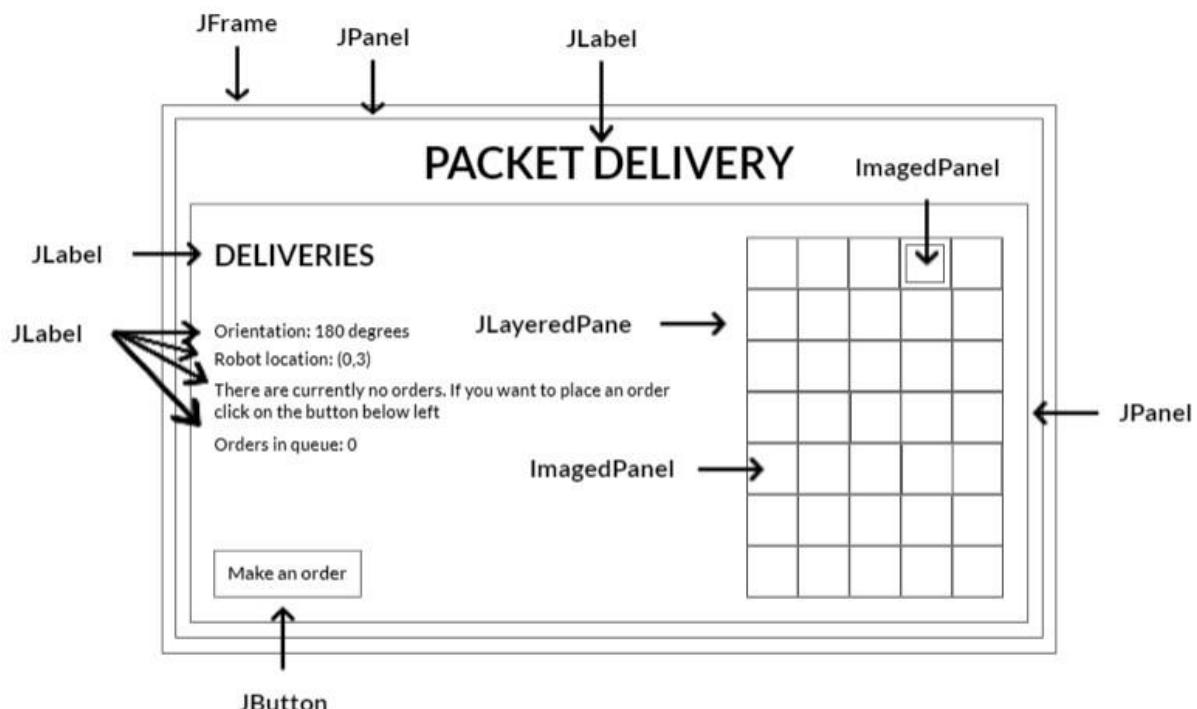


Figura 53. Implementación de la pantalla de entrega de pedidos.

Finalmente, cuando el robot llega a la casilla de entrega, es necesario eliminar los paneles que indicaban el recorrido. Para ello, obtenemos la referencia a estos paneles, los eliminamos y movemos al frente los paneles del mapa de la ciudad en esas posiciones. Sin embargo, dado que queremos que el panel del camión siga siendo visible, volvemos a mover hacia atrás los paneles del mapa de la ciudad.

```
jLayeredPane2.remove(APanel);
ImagePanel panel = mazePanels[APanel.getCoordX()][APanel.getCoordY()];
jLayeredPane2.moveToFront(panel);
jLayeredPane2.moveToBack(panel);
jLayeredPane2.remove(BPanel);
panel = mazePanels[BPanel.getCoordX()][BPanel.getCoordY()];
jLayeredPane2.moveToFront(panel);
jLayeredPane2.moveToBack(panel);
```

Figura 54. Eliminación de elementos en el mapa.

El resultado de implementar esto es el siguiente:

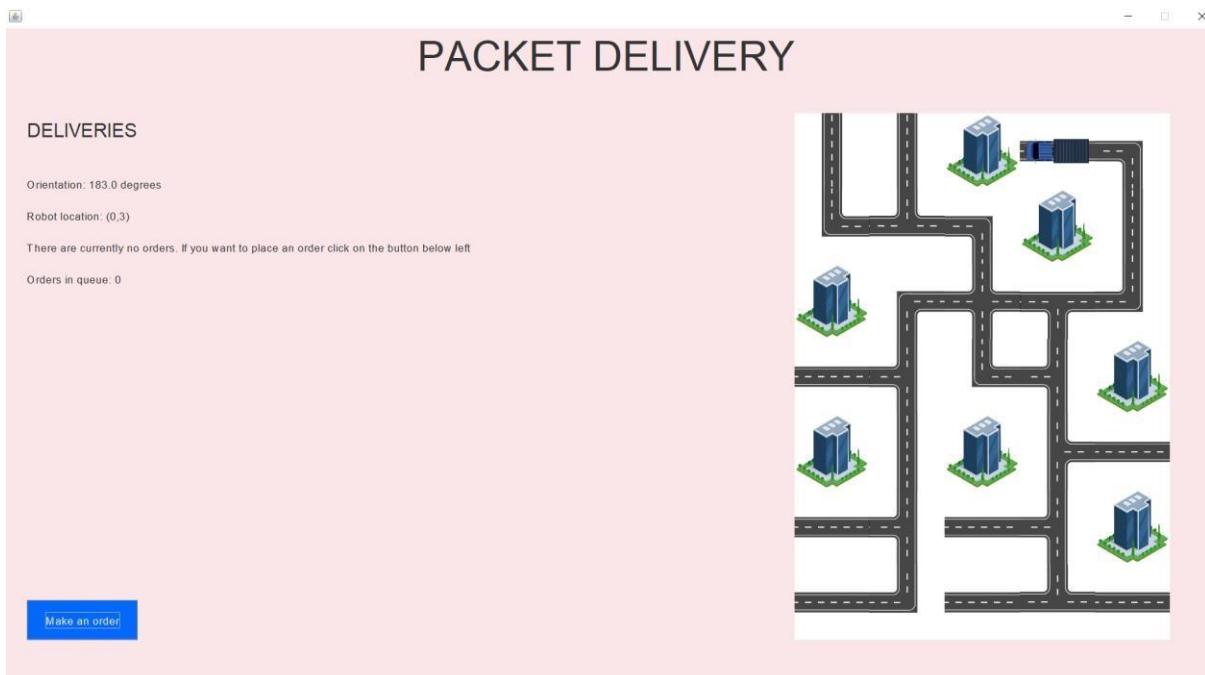


Figura 55. Resultado de la implementación de la pantalla de entrega de pedidos.

Para más detalles sobre la implementación de la aplicación cliente, visite el siguiente repositorio de GitHub:

<https://github.com/diva0001/Inteligencia-Ambiental>

7.2. ROBOT

Como se ha mencionado con anterioridad, el código para el robot "LEGO Mindstorms EV3" se desarrollará utilizando el entorno de desarrollo Visual Studio Code [5] y el lenguaje de programación Python. Además, se empleará la librería Pybricks [2]. A continuación, se detallarán algunos de los puntos más importantes de su implementación.

7.2.1 COMUNICACIONES Y PROCESAMIENTO DE MENSAJES

El robot establecerá comunicación con los diversos componentes del sistema mediante el protocolo MQTT. Esto permitirá adquirir el mapa de la ciudad y los pedidos, así como publicar información sobre las casillas de entrega y recogida, además de la odometría.

Para poder llevar esto a cabo, es necesario importar un cliente MQTT desde la librería "umqtt.robust".

```
from umqtt.robust import MQTTClient
```

Figura 56. Importación del cliente MQTT.

A continuación, debemos declarar el cliente MQTT especificando su nombre y la dirección IP del broker al que se conectará. Luego, llamaremos al método connect(), el cual establecerá la conexión con el broker. Además, es esencial definir una función de callback que procesará los mensajes recibidos a través del protocolo MQTT. Posteriormente, procederemos a suscribirnos a los topics relevantes. Por último, ponemos el cliente MQTT a la escucha de mensajes.

```
client = MQTTClient("Robot", MQTT_Broker)
client.connect()
client.set_callback(getmessages)
client.subscribe(map_Topic)
client.subscribe(orders_Topic)
```

Figura 57. Declaración del cliente MQTT, conexión y suscripción.

```
while True:
    client.check_msg()
```

Figura 58. Cliente MQTT a la escucha.

En la función getmessages(), encargada de procesar los mensajes recibidos por MQTT, el primer paso es decodificar el contenido del mensaje. Si el mensaje recibido es el mapa y es la primera vez que se recibe, se procede a procesarlo para obtener las posibles casillas de recogida y entrega. Además, dependiendo del tipo de casilla desde la que parte el robot, se establece la orientación del robot mediante la función reiniciar_odometria(). Posteriormente, las casillas de recogida y entrega se envían al broker.

En caso de que no sea la primera vez que se recibe el mapa de la ciudad, lo que indica que las casillas ya han sido calculadas previamente, simplemente se reenvían sin necesidad de recalcularlas.

Si el mensaje recibido es un pedido, este se procesa y se almacena en una cola de pedidos.

```
def getmessages(topic, msg):
    global primera
    topic = topic.decode()
    try:
        content = msg.decode()
        if primera and topic == map_Topic:
            primera = False
            processMap(content)
            reiniciar_odometria(angulo)
            client.publish(calculations_Topic, highlightedPanels)
        elif not primera and topic == map_Topic:
            client.publish(calculations_Topic, highlightedPanels)
        elif topic == orders_Topic:
            processOrder(content)
    except Exception as ex:
        print(ex)
```

Figura 59. Gestión de los mensajes recibidos por MQTT.

Para el procesamiento del mapa, la cadena recibida, que contiene los tipos de casillas, se almacena en una lista donde cada posición representa un identificador.

Luego, se crea una matriz con las dimensiones del mapa y se recorre la lista para colocar cada identificador en la posición correspondiente de la matriz.

Además, durante este proceso, se aprovecha para determinar la orientación inicial del robot en función del tipo de casilla de partida. Esto se logra observando si la casilla inicial es horizontal o vertical, lo que proporciona información sobre la orientación inicial del robot.

```
def processMap(city):
    global highlightedPanels, roads, angulo
    mapImages = [city[i: i + 2] for i in range(0, len(city), 2)]
    mapMatrix = [[0] * 5 for _ in range(7)]
    k = 0
    for i in range(7):
        for j in range(5):
            if i == 6 and j == 0:
                if mapImages[k] == "01":
                    angulo = 0
                else:
                    angulo = 90
            mapMatrix[i][j] = dic[mapImages[k]]
            k += 1
```

Figura 60. Almacenamiento de los identificadores de casilla en una matriz.

A continuación, procederemos a calcular las casillas de recogida y entrega. Para este propósito, hemos definido un diccionario que nos indicará los posibles movimientos según el tipo de casilla.

```
dic = {"00": [],
       "01": ["izquierda", "derecha"],
       "02": ["arriba", "abajo"],
       "03": ["arriba", "derecha"],
       "04": ["derecha", "abajo"],
       "05": ["abajo", "izquierda"],
       "06": ["izquierda", "arriba"],
       "07": ["izquierda", "arriba", "derecha"],
       "08": ["arriba", "derecha", "abajo"],
       "09": ["derecha", "abajo", "izquierda"],
       "10": ["abajo", "izquierda", "arriba"],
       "11": ["arriba", "derecha", "abajo", "izquierda"]}
```

Figura 61. Almacenamiento de los identificadores de casilla en una matriz.

Ahora procedemos a recorrer la matriz creada previamente y a verificar sus movimientos posibles. Por ejemplo, si la casilla (0,0) tiene un movimiento hacia la derecha, debemos comprobar si la casilla a su derecha tiene un movimiento hacia la izquierda, lo que indicaría que podemos movernos entre ellas. El objetivo principal es contar cuántos movimientos posibles se pueden realizar desde cada casilla. En el caso de que solo sea posible realizar un movimiento, almacenamos esa casilla como de recogida y entrega.

Es importante tener en cuenta que la casilla de partida no se considera como de recogida y entrega, aunque cumpla con los criterios mencionados anteriormente.

```
highlightedPanelsInteger = []
for i in range(7):
    for j in range(5):
        if i != 6 or j != 0:
            cont = 0
            movements = mapMatrix[i][j]
            for mov in movements:
                if mov == "derecha" and (j + 1) < 5:
                    if "izquierda" in mapMatrix[i][j + 1]:
                        cont += 1
                elif mov == "arriba" and (i - 1) >= 0:
                    if "abajo" in mapMatrix[i - 1][j]:
                        cont += 1
                elif mov == "izquierda" and (j - 1) >= 0:
                    if "derecha" in mapMatrix[i][j - 1]:
                        cont += 1
                elif mov == "abajo" and (i + 1) < 7:
                    if "arriba" in mapMatrix[i + 1][j]:
                        cont += 1
            if cont == 2:
                break
        if cont == 1:
            highlightedPanels = highlightedPanels + str(i) + str(j)
            highlightedPanelsInteger.append((i, j))
```

Figura 62. Cálculo de las casillas de recogida y entrega.

A continuación, procederemos a calcular el camino más corto entre cualquier par de casillas en ambos sentidos utilizando el algoritmo BFS [1], ahora que conocemos las casillas donde se pueden colocar los paquetes. Estos caminos se almacenarán en un diccionario, donde la clave será el punto de origen y destino, y el valor será una lista

de casillas que conforman la ruta, junto con la lista de movimientos necesarios para recorrer dicha ruta.

```

for orig in highlightedPanelsInteger:
    for destino in highlightedPanelsInteger:
        if orig != destino:
            road = shortest_road(mapMatrix, orig[0], orig[1], destino[0], destino[1])
            roads[((orig[0], orig[1]), (destino[0], destino[1]))] = (road, getMovements(road))
            reverse_road = road[::-1]
            road.pop(0)
            reverse_road.reverse()
            roads[((destino[0], destino[1]), (orig[0], orig[1]))] = (reverse_road, getMovements(reverse_road))
            reverse_road.pop(0)
for orig in highlightedPanelsInteger:
    road = shortest_road(mapMatrix, orig[0], orig[1], 6, 0)
    roads[((orig[0], orig[1]), (6, 0))] = (road, getMovements(road))
    reverse_road = road[::-1]
    road.pop(0)
    reverse_road.reverse()
    roads[((6, 0), (orig[0], orig[1]))] = (reverse_road, getMovements(reverse_road))
    reverse_road.pop(0)

```

Figura 63. Cálculo del camino óptimo entre las casillas de recogida y entrega.

La función `getMovements()` tiene como objetivo traducir un camino dado en una lista de movimientos. Para lograr esto, recorremos la ruta por pares de origen y destino, identificando si el movimiento es hacia arriba, abajo, izquierda o derecha. Una vez obtenidos estos movimientos, detectamos los giros en la ruta. Por ejemplo, si el camino implica primero un movimiento hacia arriba y luego hacia la derecha, sabemos que se realiza un giro a la derecha.

```

def getMovements(road):
    lista_movimientos = []
    for i in range(len(road) - 1):
        origen = road[i]
        destino = road[i + 1]
        if origen[0] > destino[0]:
            lista_movimientos.append("arriba")
        elif origen[0] < destino[0]:
            lista_movimientos.append("abajo")
        elif origen[1] > destino[1]:
            lista_movimientos.append("izquierda")
        else:
            lista_movimientos.append("derecha")
    nueva_lista_movimientos = []
    for i in range(len(lista_movimientos) - 1):
        origen = lista_movimientos[i]
        destino = lista_movimientos[i + 1]
        movimiento = None
        if (origen == "izquierda" and destino == "arriba") or (origen == "arriba" and destino == "derecha") or \
           (origen == "derecha" and destino == "abajo") or (origen == "abajo" and destino == "izquierda"):
            movimiento = "giro derecha"
        elif (origen == "derecha" and destino == "arriba") or (origen == "arriba" and destino == "izquierda") or \
              (origen == "izquierda" and destino == "abajo") or (origen == "abajo" and destino == "derecha"):
            movimiento = "giro izquierda"
        if movimiento:
            nueva_lista_movimientos.append(movimiento)
        else:
            nueva_lista_movimientos.append(origen)
    return nueva_lista_movimientos

```

Figura 64. Obtención de la lista de movimientos dado un camino.

A continuación, presentamos la implementación del algoritmo BFS [1] para la búsqueda de rutas, tal y como se ha explicado en un apartado previo:

```
def shortest_road(mapMatrix, start_row, start_col, end_row, end_col):
    rows, cols = 7, 5
    visited = [[False] * cols for _ in range(rows)]
    queue = deque([(start_row, start_col, 0, [1])])

    while queue:
        row, col, distance, road = queue.popleft()
        visited[row][col] = True

        if row == end_row and col == end_col:
            return road + [(row, col)]

        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < rows and 0 <= new_col < cols:
                if (dr == -1 and "arriba" in mapMatrix[row][col] and "abajo" in mapMatrix[new_row][new_col] and not visited[new_row][new_col]) or \
                    (dr == 1 and "abajo" in mapMatrix[row][col] and "arriba" in mapMatrix[new_row][new_col] and not visited[new_row][new_col]) or \
                    (dc == -1 and "izquierda" in mapMatrix[row][col] and "derecha" in mapMatrix[new_row][new_col] and not visited[new_row][new_col]) or \
                    (dc == 1 and "derecha" in mapMatrix[row][col] and "izquierda" in mapMatrix[new_row][new_col] and not visited[new_row][new_col]):
                    new_road = road + [(row, col)]
                    queue.append((new_row, new_col, distance + 1, new_road))
                    visited[new_row][new_col] = True
```

Figura 65. Implementación del algoritmo BFS.

Por otro lado, como en la [Figura 60](#) hemos obtenido la orientación inicial del robot, lo que hacemos es reiniciar los ángulos tanto del giroscopio como de los motores.

```
def reiniciar_odometria(angulo):
    motor_izquierdo.reset_angle(angulo)
    motor_derecho.reset_angle(angulo)
    giroscopio.reset_angle(angulo)
```

Figura 66. Inicialización de los ángulos.

Para procesar los pedidos recibidos, que se presentan en forma de una cadena con el formato “x₁y₁x₂y₂”, simplemente leemos la cadena carácter por carácter, convertimos cada uno en un número entero y luego agregamos el pedido a la cola.

```
def processOrder(order):
    global ordersQueue
    x1 = int(order[0])
    y1 = int(order[1])
    x2 = int(order[2])
    y2 = int(order[3])
    ordersQueue.append(((x1, y1), (x2, y2)))
```

Figura 67. Procesamiento de un pedido.

Para publicar la odometría, hemos definido la función enviar_datos(), que envía la posición y orientación actual del robot en una cadena en el formato “xy;ángulo”.

```
def enviar_datos(celda_x, celda_y, angulo):
    data = str(celda_x) + str(celda_y) + ";" + str(angulo)
    client.publish(odometry_Topic, data)
```

Figura 68. Envío de la odometría del robot.

Esta función se invoca cada segundo. Para medir el tiempo se ha importado la clase StopWatch del módulo tools de la biblioteca Pybricks [2].

```
from pybricks.tools import Stopwatch
```

Figura 69. Importación de la clase StopWatch para medir el tiempo.

Esta clase mide el tiempo transcurrido en milisegundos desde que se declara el objeto hasta que se llama al método time(). Por lo tanto, verificamos si han transcurrido más de 1000 milisegundos (1 segundo), momento en el cual enviamos los datos y reseteamos el contador, equivalente a reiniciarlo para comenzar a contar nuevamente.

```
reloj = Stopwatch()

while True:
    client.check_msg()

    if reloj.time() >= 1000:
        enviar_datos(celda_x, celda_y, giroscopio.angle())
        reloj.reset()
```

Figura 70. Envío de datos cada segundo.

7.2.2 MOVIMIENTO

La estrategia para implementar el movimiento del robot sigue esta secuencia: comienza con la pala levantada, esperando un pedido. Una vez que recibe un pedido, calcula la ruta para recogerlo y la ejecuta. Al llegar al lugar del paquete, baja la pala, determina la ruta hacia el punto de entrega, se da la vuelta y se dirige hacia allí. Una

vez en el punto de entrega, levanta la pala. Si hay más pedidos en la cola, repite este proceso. En caso contrario, permanece en espera hasta recibir otro pedido.

Para implementar el seguimiento de las calles, nuestra estrategia principal consiste en seguir el verde siempre que sea posible. En primer lugar, medimos el valor del verde objetivo utilizando el sensor de color en modo RGB, el cual nos proporciona tres valores entre 0 y 100: uno para el rojo, otro para el verde y otro para el azul. A medida que el robot se mueve, continúa leyendo los colores y los compara con el verde objetivo. Si el color leído coincide con nuestro verde objetivo, el robot se ajustará ligeramente hacia la derecha; si es diferente, se ajustará hacia la izquierda. Este enfoque garantiza que el robot siempre se ubique a la derecha de la calle. Sin embargo, uno de los desafíos es que el valor del verde no es uniforme en todo el mapa, por lo que es necesario establecer un umbral por encima y por debajo del color objetivo para considerarlo como verde.

```
rojo = 31
verde = 54
azul = 20
lado = 10
```

Figura 71. Definición del verde objetivo y el umbral.

```
def seguirVerde(color_actual, rojo, verde, azul, lado):
    if color_actual[0] >= (rojo - (lado/2)) and color_actual[0] <= (rojo + (lado/2)) and color_actual[1] >= (verde - (lado/2)) and \
        color_actual[1] <= (verde + (lado/2)) and color_actual[2] >= (azul - (lado/2)) and color_actual[2] <= (azul + (lado/2)):
        turn_rate = -23
        robot.drive(DRIVE_SPEED, turn_rate)
    else:
        turn_rate = 23
        robot.drive(DRIVE_SPEED, turn_rate)
```

Figura 72. Seguimiento del verde usando umbrales.

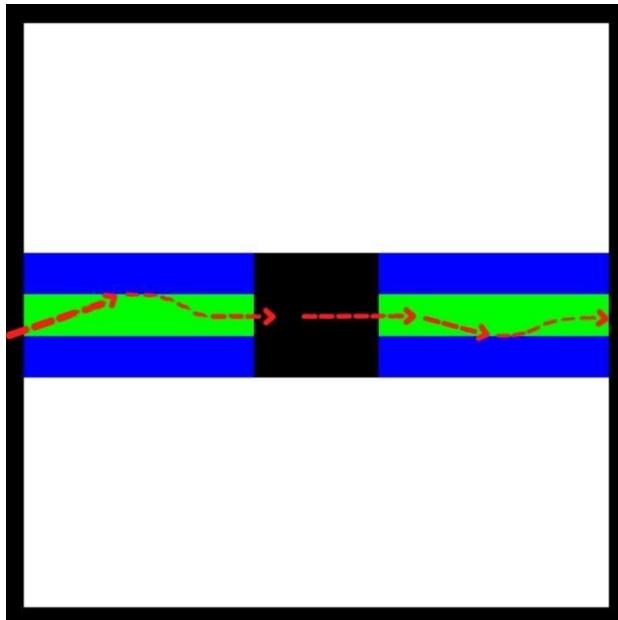


Figura 73. Seguimiento del verde.

El problema que surge es que el robot puede encontrarse con tramos de negro, como la línea que delimita dos casillas o el cuadrado en el centro de la casilla. Por esta razón, inicialmente consideramos que cuando el robot detectara el color negro, su movimiento debería ser recto.

```
def estaEnNegro(color_actual):
    return color_actual[0] >= 0 and color_actual[1] <= 25 and color_actual[1] >= 0 and color_actual[1] <= 25 and \
           color_actual[2] >= 0 and color_actual[2] <= 25
```

Figura 74. Detección del color negro.

Sin embargo, esto también plantea un inconveniente: dado que el robot ajusta su trayectoria al seguir el verde con giros a la izquierda y a la derecha, puede ocurrir que al llegar a una línea negra o al cuadrado en el centro de la casilla, el robot quede ligeramente girado hacia la izquierda. Entonces, al avanzar recto para salir del negro, puede ocurrir que ya no encuentre el verde, girando aún más hacia la izquierda, lo cual queremos evitar.

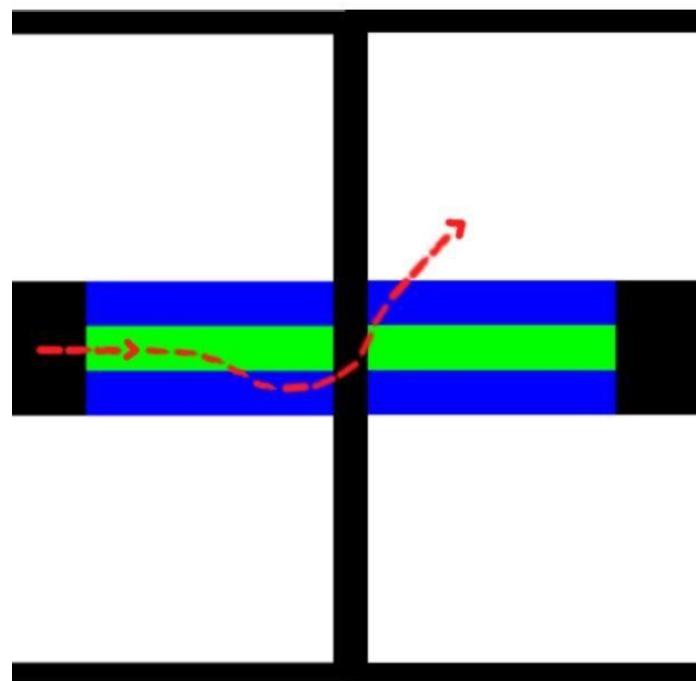


Figura 75. Dificultad al avanzar recto cuando se detecta el color negro.

Para solucionar esto, hemos implementado que cuando el robot detecte negro, en lugar de avanzar en línea recta, avance girando un poco hacia la derecha. De esta manera, nos aseguramos de que el robot se ubique correctamente en la calle.

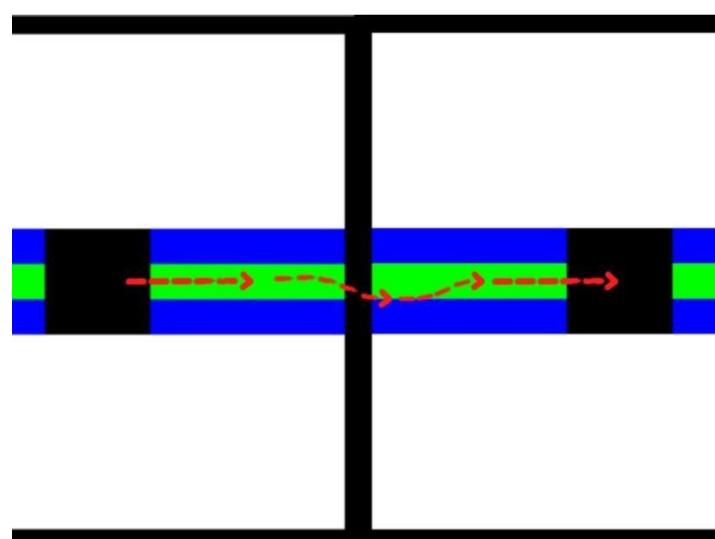


Figura 76. Ajuste a la derecha cuando se detecta el color negro y hay que avanzar recto.

```

def salirDeNegro(primerNegro, lista_movimientos):
    turn_rate = 0
    drive_speed = DRIVE_SPEED
    if primerNegro:
        movimiento = lista_movimientos[1][0]
        if movimiento == "giro derecha":
            drive_speed = 150
            turn_rate = -65
        elif movimiento == "giro izquierda":
            drive_speed = 150
            turn_rate = 65
        else:
            turn_rate = -20
    else:
        turn_rate = -40
    robot.drive(drive_speed, turn_rate)

```

Figura 77. Código de ajuste a la derecha cuando se detecta el color negro y hay que avanzar recto.

En este caso, observamos que al encontrar un cubo cuadrado negro, el robot gira 20 grados a la derecha, mientras que al detectar una línea negra, gira 40 grados.

Como era de esperar, es necesario llevar un registro de qué tipo de negro estamos detectando, ya sea la línea que delimita dos casillas o el cuadrado central. Para lograr esto, utilizamos una variable booleana llamada “primerNegro”, la cual cambia su valor cada vez que detectamos que hemos salido del color negro. De esta manera, podemos determinar cuándo el robot cambia de casilla y actualizar en consecuencia su posición.

Para utilizar la lista de movimientos, cada vez que detectamos un cuadrado negro (cuando “primerNegro” es True), extraemos el primer movimiento de la lista. Si el movimiento no implica un giro, permitimos que el robot avance según lo explicado previamente. Si el movimiento implica un giro, aplicamos un ángulo de giro mayor para asegurarnos de que el robot salga del negro y quede orientado correctamente para el giro. El objetivo de este proceso es que todo este código se ejecute hasta que el robot salga del cubo negro y quede correctamente orientado, ya sea para girar o para seguir la línea verde.

```

if negro:
    if primerNegro and len(lista_movimientos[1]) == 0:
        subir_pala()
        pedidoEntregado = True
    else:
        salirDeNegro(primerNegro, lista_movimientos)
        seguirLinea = False

```

Figura 78. Código que se ejecuta cuando se detecta el color negro.

```

def salirDeNegro(primerNegro, lista_movimientos):
    turn_rate = 0
    drive_speed = DRIVE_SPEED
    if primerNegro:
        movimiento = lista_movimientos[1][0]
        if movimiento == "giro derecha":
            drive_speed = 150
            turn_rate = -65
        elif movimiento == "giro izquierda":
            drive_speed = 150
            turn_rate = 65
        else:
            turn_rate = -20
    else:
        turn_rate = -40
    robot.drive(drive_speed, turn_rate)

```

Figura 79. Código para salir del color negro.

Una vez que el robot abandona el cuadrado negro, procedemos a obtener el siguiente movimiento de la lista. Si el movimiento es recto, seguimos la línea verde y luego eliminamos el movimiento de la lista. En caso de que sea un giro, aplicamos una rotación al robot con un ángulo adecuado para el tipo de giro detectado, y posteriormente eliminamos el movimiento de la lista.

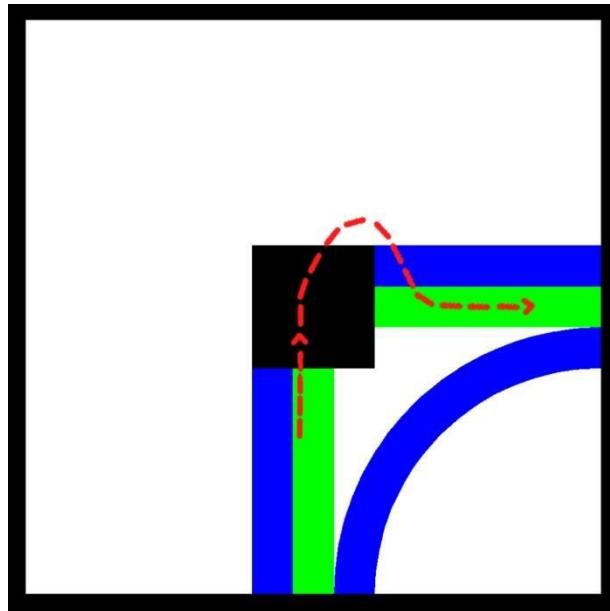


Figura 80. Recolocación y aplicación del giro.

```
if primerNegro and not seguirLinea:
    movimiento_completado = aplicarMovimiento(lista_movimientos, rojo, verde, azul, color_actual)
    if movimiento_completado:
        seguirLinea = True
        primerNegro = False
```

Figura 81. Código para aplicar el movimiento tras pasar un cuadrado negro.

```
def aplicarMovimiento(lista_movimientos, rojo, verde, azul, color_actual):
    movimiento = lista_movimientos[1][0]
    if movimiento in ["giro derecha", "giro izquierda"]:
        turn_rate = -80 if movimiento == "giro derecha" else 75
        lista_movimientos[1].pop(0)
        robot.turn(turn_rate)
        return True
    lista_movimientos[1].pop(0)
    seguirVerde(color_actual, rojo, verde, azul, lado)
    return True
```

Figura 82. Aplicación del movimiento al salir del cuadrado negro.

Por otro lado, resta explicar el proceso de recogida y entrega del paquete. La recogida del paquete ocurre cuando la lista de movimientos del robot está vacía, lo que indica que ha llegado a su destino. En este punto, al detectar negro, en lugar de extraer el siguiente movimiento, el robot bajará la pala. Del mismo modo, durante la entrega, al detectar el cuadrado negro de la casilla, el robot elevará la pala. Este proceso se ilustra en la [Figura 78](#).

```
def subir_pala():
    motor_pala.run_target(500, PALA_ARRIBA, then = Stop.HOLD, wait = True)

def bajar_pala():
    motor_pala.run_target(500, PALA_ABAJO, then = Stop.HOLD, wait = True)
```

Figura 83. Subida y bajada de la pala.

Para que el robot dé la vuelta, ya sea para ir a buscar el siguiente paquete o para entregarlo una vez recogido, aplicamos un giro de aproximadamente 180 grados hacia la derecha. Sin embargo, debido a que el giro del robot no es exacto, hemos indicado un ángulo de giro mayor para asegurar que gire como deseamos.

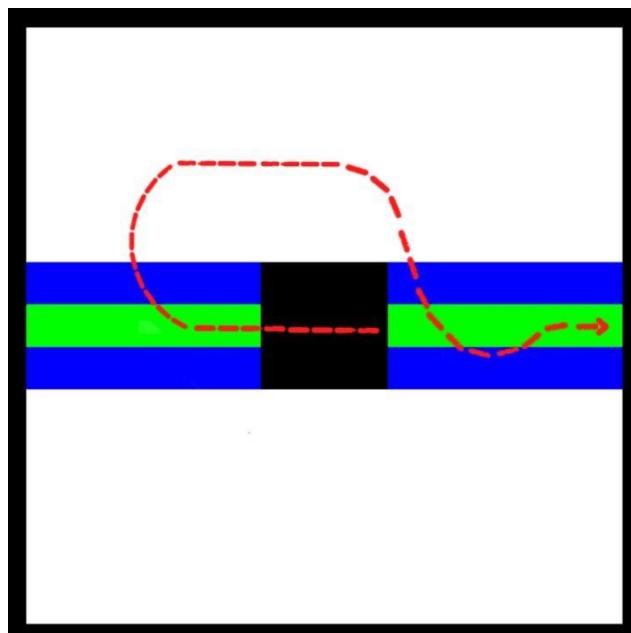


Figura 83. Giro del robot.

```
def vuelta():
    robot.turn(-226)
```

Figura 84. Código de giro del robot.

El código del robot se divide en dos partes principales: la recogida y la entrega del paquete. Para lograr esto, se han desarrollado dos funciones que se presentan a continuación:

```

def recogerPedido(vueltaInicial, celda_x, celda_y, lista_movimientos, primerNegro, seguirLinea, rojo, verde, azul, lado):
    pedidoRecogido = False
    color_actual = color_sensor.rgb()
    if not vueltaInicial:
        vuelta()
        vueltaInicial = True
    else:
        negro = estaEnNegro(color_actual)

    if negro:
        if primerNegro and (lista_movimientos == None or len(lista_movimientos[1]) == 0):
            bajar_pala()
            robot.stop()
            pedidoRecogido = True
        else:
            salirDeNegro(primerNegro, lista_movimientos)
            seguirLinea = False
    else:
        if primerNegro and not seguirLinea:
            movimiento_completado = aplicarMovimiento(lista_movimientos, rojo, verde, azul, color_actual)
            if movimiento_completado:
                print("fuera de primer negro")
                seguirLinea = True
                primerNegro = False
            else:
                if not seguirLinea:
                    print("fuera de la linea negra")
                    posicion = lista_movimientos[0].pop(0)
                    celda_x, celda_y = posicion
                    primerNegro = True
                    seguirLinea = True
                seguirVerde(color_actual, rojo, verde, azul, lado)

    return vueltaInicial, celda_x, celda_y, primerNegro, seguirLinea, pedidoRecogido

```

Figura 85. Código de recogida del paquete.

```

def entregarPedido(vueltaInicial, celda_x, celda_y, lista_movimientos, primerNegro, seguirLinea, rojo, verde, azul, lado):
    pedidoEntregado = False
    color_actual = color_sensor.rgb()
    if not vueltaInicial:
        vuelta()
        vueltaInicial = True
    else:
        negro = estaEnNegro(color_actual)

    if negro:
        if primerNegro and len(lista_movimientos[1]) == 0:
            subir_pala()
            pedidoEntregado = True
        else:
            salirDeNegro(primerNegro, lista_movimientos)
            seguirLinea = False
    else:
        if primerNegro and not seguirLinea:
            movimiento_completado = aplicarMovimiento(lista_movimientos, rojo, verde, azul, color_actual)
            if movimiento_completado:
                seguirLinea = True
                primerNegro = False
            else:
                if not seguirLinea:
                    posicion = lista_movimientos[0].pop(0)
                    celda_x, celda_y = posicion
                    primerNegro = True
                    seguirLinea = True
                seguirVerde(color_actual, rojo, verde, azul, lado)

    return vueltaInicial, celda_x, celda_y, primerNegro, seguirLinea, pedidoEntregado

```

Figura 86. Código de entrega del paquete.

Para más detalles sobre la implementación del robot, visite el siguiente repositorio de GitHub:

<https://github.com/diva0001/Inteligencia-Ambiental>

8. DESPLIEGUE

Para poder probar el funcionamiento del proyecto, en primer lugar hay que instalar los siguientes entornos de desarrollo:

- **Visual Studio Code:** <https://code.visualstudio.com/download>

- **NetBeans:** <https://netbeans.apache.org/front/main/download/>

A continuación, es necesario acceder al siguiente repositorio de GitHub y descargar el código fuente:

<https://github.com/diva0001/Inteligencia-Ambiental/tree/master>

Después de descargar el código, abra Visual Studio Code [5] y proceda a instalar la extensión "EV3 MicroPython", siguiendo los pasos que se muestran en la siguiente imagen:

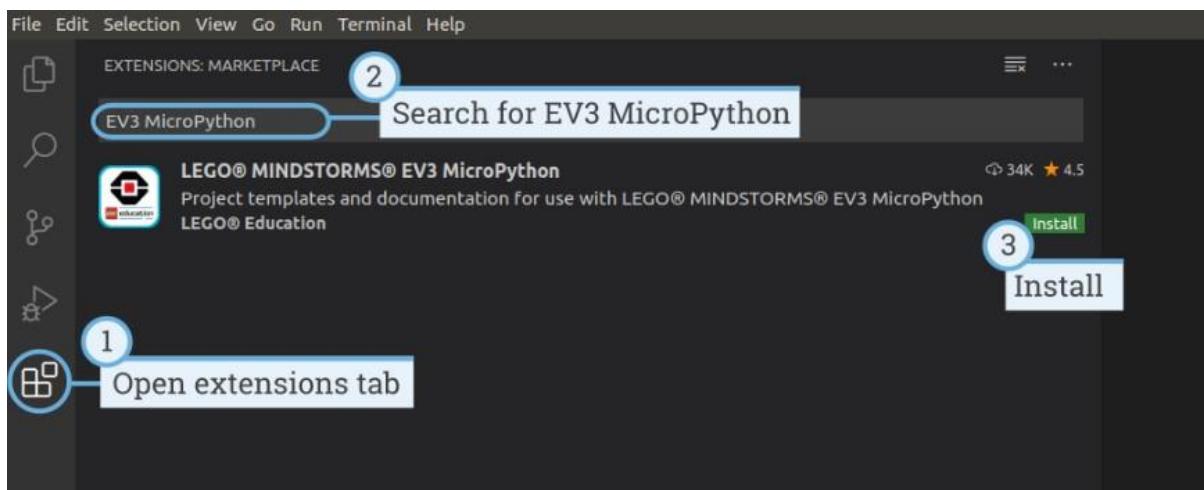


Figura 87. Descarga de la extensión.

Descomprima la carpeta descargada del repositorio y abra la carpeta "robot" que se encuentra dentro de ella desde Visual Studio Code [5].



Figura 88. Apertura del proyecto.

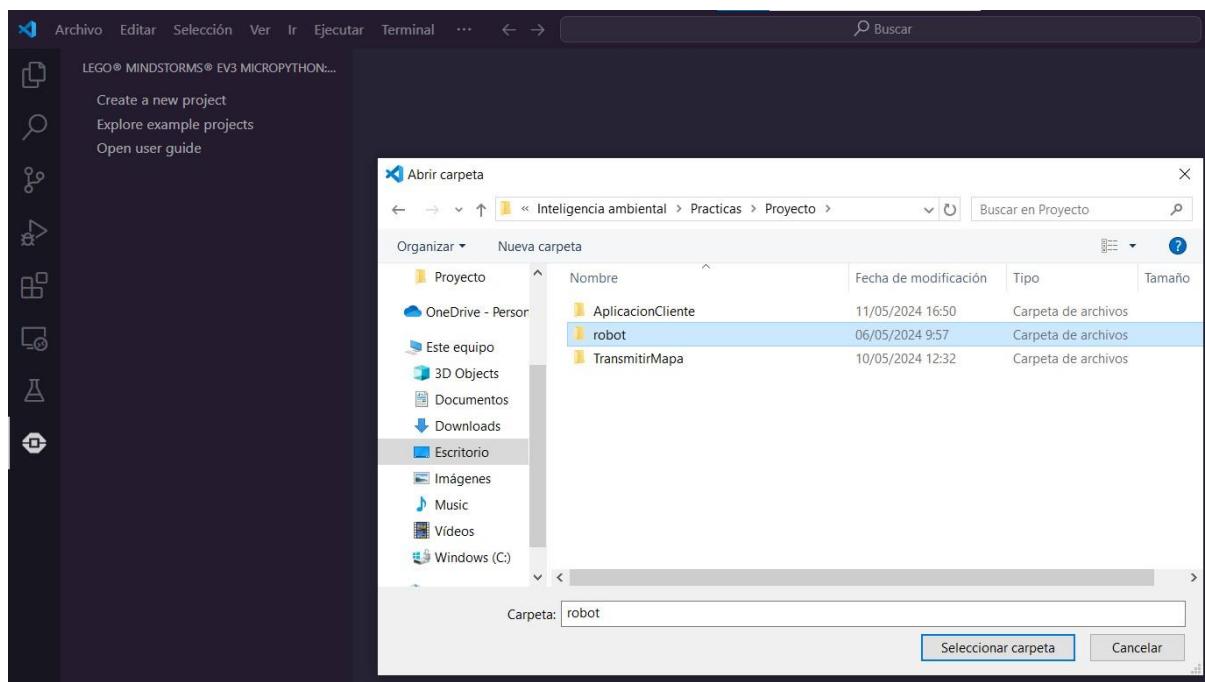


Figura 89. Apertura del proyecto.

Luego, abra el archivo "main.py".

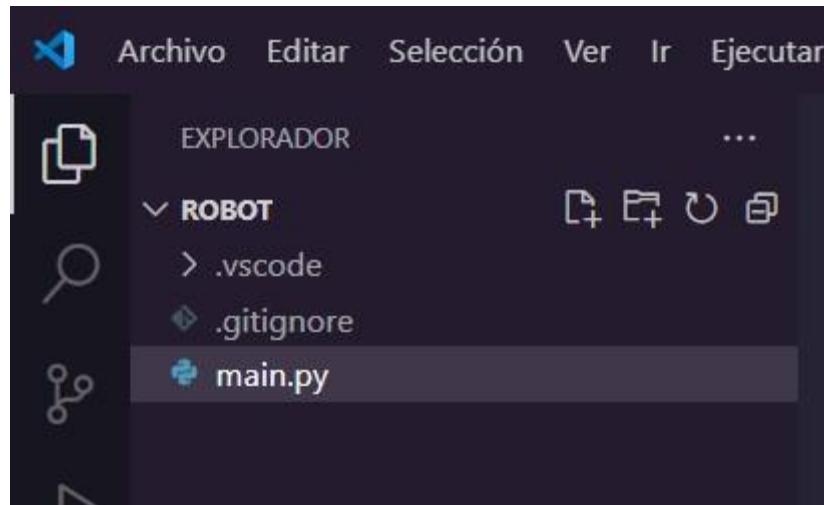


Figura 90. Apertura del código del robot.

Seguidamente, establezca la dirección IP del broker MQTT:

```
MQTT_Broker = "192.168.0.129"
```

Figura 91. Establecimiento de la dirección IP del broker MQTT.

Luego, conecte el robot a su equipo con el cable y acceda a él.

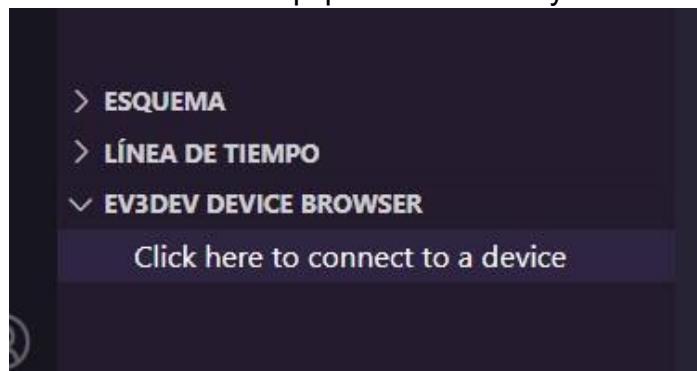


Figura 92. Conexión al robot.

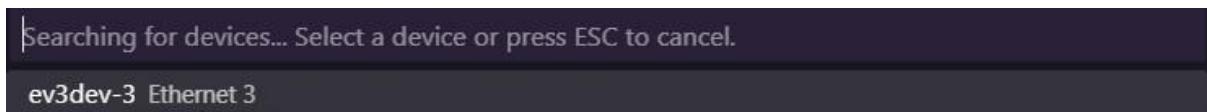


Figura 93. Selección del dispositivo.

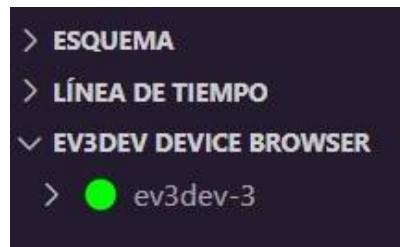


Figura 93. Robot conectado.

A continuación, coloque el sensor de color del robot sobre el color verde del mapa para calibrarlo. Después, busque el siguiente bloque de código, descoméntelo y súbalo al robot presionando la tecla F5 para iniciar su ejecución.

```
# while True:  
#     color = color_sensor.rgb()  
#     print(color)
```

Figura 94. Calibración del color verde.

Mientras el código se ejecuta, veremos algo similar a (31, 54, 20), que son los valores RGB del color verde. Estos valores son los que debemos introducir en las variables:

```
rojo = 31  
verde = 54  
azul = 20
```

Figura 95. Calibración del color verde.

¡Cuidado! Hay que poner los valores que nos salgan, no los indicados en la imagen.

Después, una vez calibrado el verde, volvemos a comentar el código mencionado y lo volvemos a subir al robot con la tecla F5. En ese momento comienza la ejecución del robot.

Ahora pasamos a ejecutar la aplicación cliente. Se puede ejecutar tanto desde un archivo .jar como desde el entorno de desarrollo NetBeans [6]. Dado que la dirección IP del broker puede cambiar, vamos a explicar cómo ejecutarla desde NetBeans [6].

En primer lugar, abra NetBeans [6] seleccione la opción de “ abrir proyecto”.

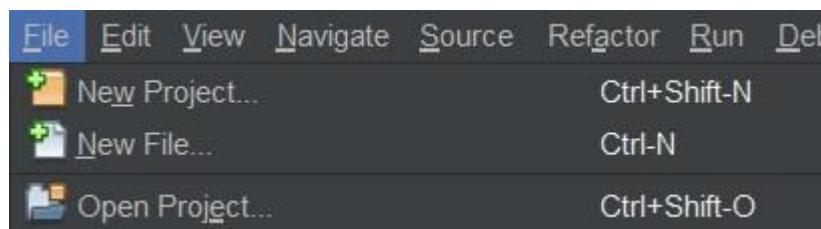


Figura 96. Apertura del proyecto.

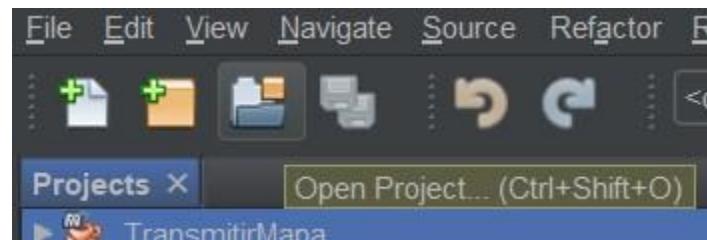


Figura 97. Apertura del proyecto.

En segundo lugar, abrir la aplicación cliente.

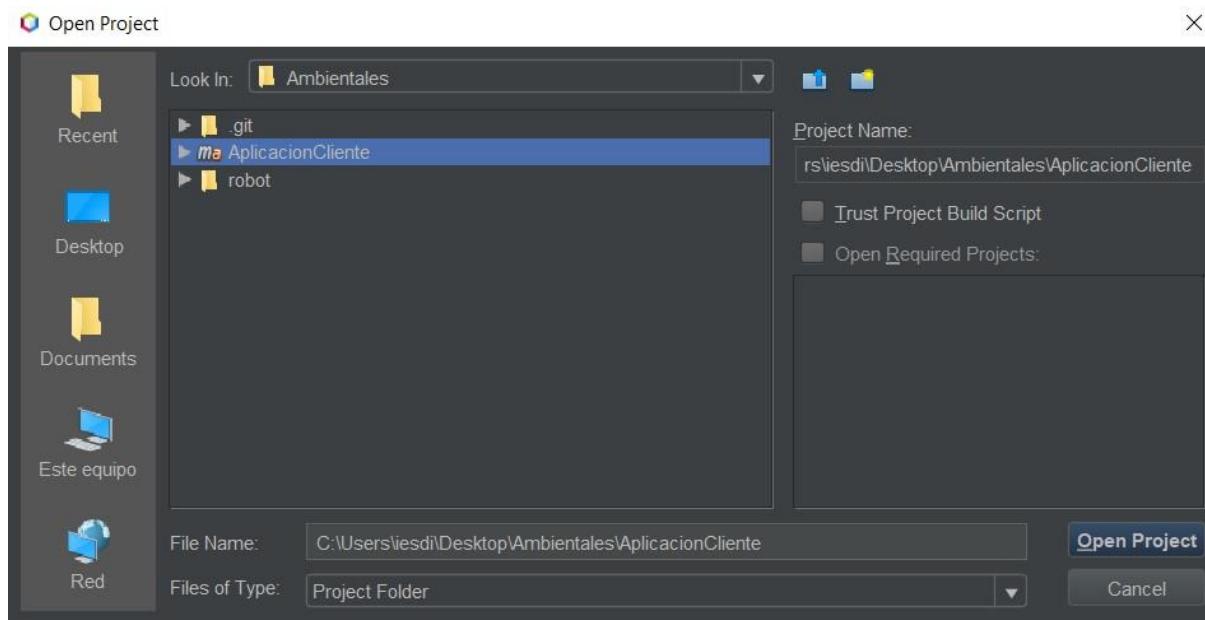


Figura 98. Apertura del proyecto.

Una vez hecho esto, el proyecto nos aparecerá en el panel de la izquierda.

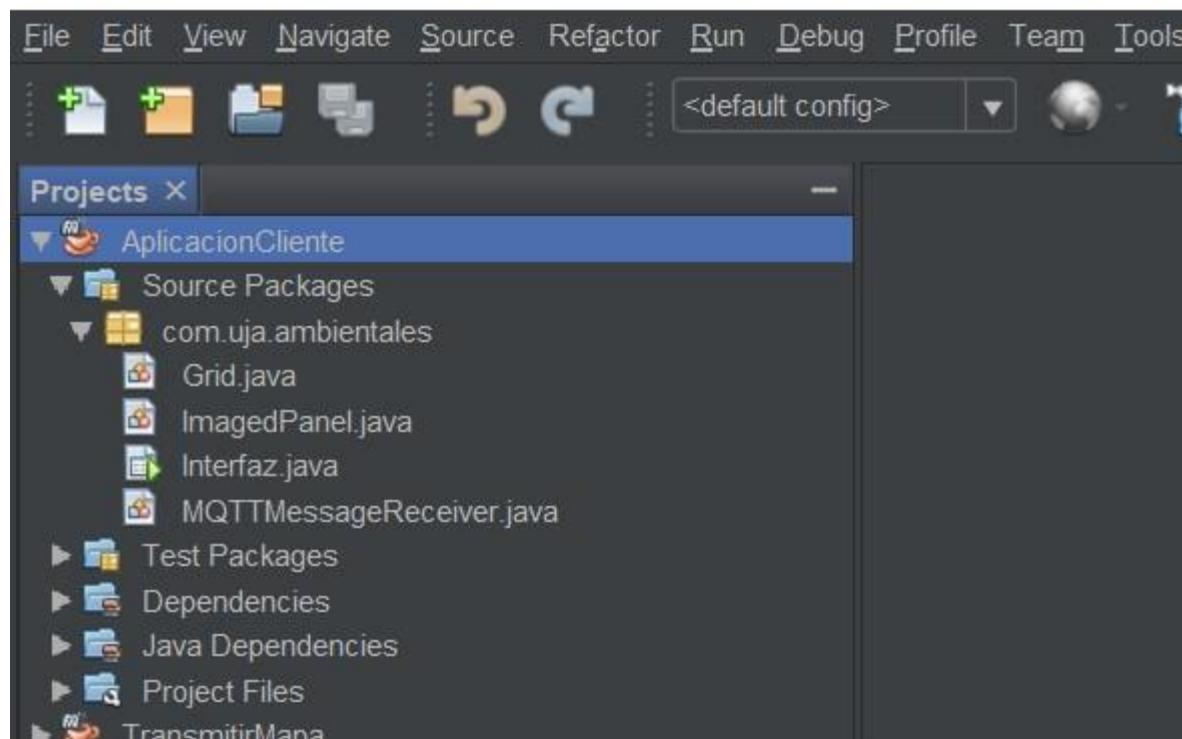


Figura 99. Proyecto abierto.

bra el fichero "MessageReceiver.java" establezca la dirección IP del broker.

```
this.broker = "tcp://192.168.0.129:1883";
```

Figura 100. Establecimiento de la dirección del broker MQTT.

Por último, solo nos queda lanzar la aplicación dándole al botón de ejecución.

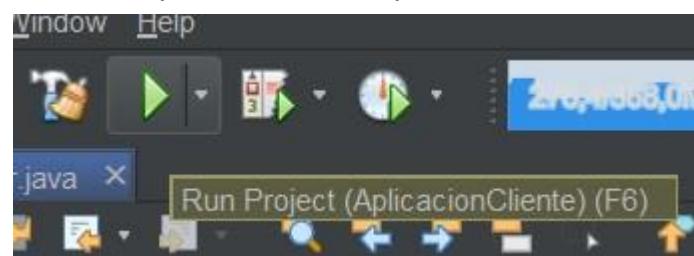


Figura 101. Ejecución de la aplicación cliente.

Si hemos seguido correctamente todos estos pasos, podremos utilizar el sistema sin problemas.

Es importante destacar que si deseamos ejecutar la aplicación cliente desde el archivo .jar, simplemente debemos hacer doble clic en él. Sin embargo, es crucial que la dirección IP del broker sea la correcta antes de generar el .jar. Por defecto, se considerará que la dirección IP del broker es 192.168.0.100, tal como se indica en el guión del proyecto. El fichero ejecutable se encontrará dentro de la carpeta "software".

Además, es importante tener en cuenta que si el broker tiene una dirección IP privada, debemos estar en la misma red que él para poder utilizarlo.

9. RESULTADOS

En este apartado se mostrarán las métricas de tiempo que se han obtenido con la variación de ciertos parámetros importantes para el funcionamiento del robot. También incluye enlaces a vídeos para una mejor comprensión de estos.

9.1. TIEMPOS

Antes de poder comparar los distintos resultados de los tiempos conseguidos por el robot, se necesita explicar las distintas variables que se han ido modificando para calibrar el robot a lo largo de las entregas exitosas.

- **Velocidad lineal:** Es la velocidad por defecto del robot. Usará esta velocidad mientras intenta detectar el color verde del suelo.
- **Ángulo de corrección verde:** Es el ángulo que girará el robot para mantenerse sobre el camino sin desviarse. Si el color detectado es verde, el ángulo será negativo y girará a la derecha, si el color detectado es distinto de verde, el ángulo será positivo y girará a la izquierda.
- **Ángulo de giro negro:** Es el ángulo que girará el robot cuando llegue al bloque negro, girará a la derecha o hacia la izquierda dependiendo del movimiento almacenado por el robot en la lista de movimientos. Para ello, avanzará un poco recto con un pequeño giro y cuando se salga, realizará el giro hasta orientarse y seguir su curso.
- **Ángulo para ir recto negro:** Es el ángulo que girará el robot cuando llegue al cubo negro y el siguiente movimiento sea recto. Girará un poco hacia la derecha al igual que en la línea negra para que no se salga del camino.
- **Ángulo línea negra:** Es el ángulo de corrección hacia la derecha que se aplicará cuando el robot detecte la línea negra para no salirse del camino, es decir, como el robot debe de orientarse entre la línea verde y la línea azul de la derecha, aplicamos un ángulo hacia la derecha al llegar a la línea negra, para tratar de que no se desvíe a la línea azul de la izquierda.
- **Grados para el giro vuelta:** Grados que girará el robot una vez recogido o entregado el paquete para darse la vuelta.

Se ha conseguido mejorar el tiempo de entrega y recogida de los paquetes un total de 4 veces, dándonos como mejor tiempo: 1 minuto y 27 segundos. Los tiempos en cada mejora son los siguientes:

- **Tiempo 1: 3 minutos y 35 segundos.**

- Velocidad lineal: 60.
- Ángulo de corrección verde: 70.
- Ángulo de giro negro: 15.
- Ángulo para ir recto negro: -10.
- Ángulo línea negra: 20.
- Grados para el giro vuelta: -210.

[Enlace al video.](#)

- **Tiempo 2: 2 minutos y 4 segundos.**

- Velocidad lineal: 100.
- Ángulo de corrección verde: 25.
- Ángulo de giro negro: 22.
- Ángulo para ir recto negro: -13.
- Ángulo línea negra: -23.
- Grados para el giro vuelta: -230.

[Enlace al video.](#)

- **Tiempo 3: 1 minutos y 37 segundos.**

- Velocidad lineal: 150.
- Ángulo de corrección verde: 30.
- Ángulo de giro negro: 22.
- Ángulo para ir recto negro: -20.
- Ángulo línea negra: -35.
- Grados para el giro vuelta: -230.

[Enlace al video.](#)

- **Tiempo 4: 1 minutos y 27 segundos.**

- Velocidad lineal: 150.

- Ángulo de corrección verde: 23.
- Ángulo de giro negro: 65.
- Ángulo para ir recto negro: -20.
- Ángulo línea negra: -40. ○ Grados para el giro vuelta: -226.

[Enlace al video.](#)

9.2. TIEMPOS

La siguiente tabla resumirá los tiempos del robot.

TIEMPOS	PAQUETE1	PAQUETE2	TOTAL
Tiempo 1	1 min y 57 segundos	1 min y 38 segundos	3 min y 35 segundos
Tiempo 2	1 min y 7 segundos	57 segundos	2 min y 4 segundos
Tiempo 3	51 segundos	46 segundos	1 min y 37 segundos
Tiempo 4	46 segundos	41 segundos	1 min y 27 segundos

Tabla 2. Comparativa de tiempos.

Se presenta un gráfico que muestra los tiempos de cada paquete completado por el robot.

Tiempos de los paquetes en segundos

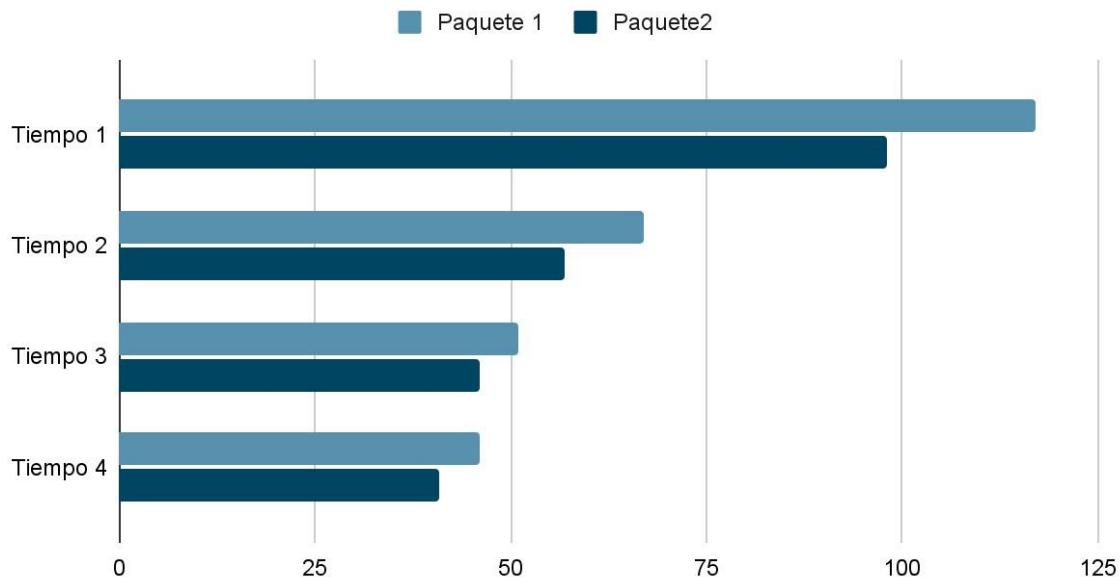


Figura 102. Comparación de tiempos entre paquetes.

Gráfico del tiempo total que se ha conseguido optimizar:

Tiempo final en segundos



Figura 103. Comparación de tiempos.

En el gráfico de tiempos finales se puede observar una curva descendente y que empieza a mantenerse constante cerca de los últimos valores, mostrando así la dificultad para optimizar los tiempos del robot.

En el gráfico de los tiempos de cada paquete se puede ver una diferencia entre el tiempo de ambos paquetes de un mismo tiempo ejecutado, esto se debe a que para realizar la entrega y recogida del paquete 2 no hace falta hacer tanto recorrido como en el primer paquete. Aun así, en los últimos tiempos conseguidos (tiempos 3 y 4), el primer pedido consigue optimizar más su marca a como estaba inicialmente con respecto al paquete 2.

Olvidando la diferencia entre ambos paquetes, también se puede ver que cada tiempo final consigue optimizarse con respecto al tiempo de recogida y entrega anterior, quedándose algo más constantes en los últimos tiempos.

9.3. DIFICULTADES

Aquí se explicarán los fallos más comunes y las variables que más se han tenido que modificar para poder solventar estos problemas. Además se agregarán videos para facilitar la comprensión de estos fallos.

En primer lugar resaltaremos, uno de nuestros problemas y fallos que más tiempo nos han ocasionado de arreglar.

- **Variable ángulo de corrección.**

Hemos tenido que realizar muchísimas pruebas para ajustar esta variable puesto que esta, se encarga principalmente del movimiento en línea recta del robot, realizando un pequeño giro a la derecha cuando esté en el verde, y un giro a la izquierda cuando el color sea distinto de verde.

Nos ha ocasionado muchos problemas, debido a que cuando aumentamos la velocidad, en diversas situaciones, el robot no corregía lo suficiente rápido y se salía del camino hacia la línea azul de la izquierda y por lo tanto, al estar en azul, gira la derecha, pero de esta forma, se saldría hacia el blanco y perdería por lo tanto, la orientación el robot.

Este problema, surgió a raíz del movimiento de realizar la vuelta, puesto que cuando realizaba la vuelta de 180 grados, tocaba casi la línea negra y al no estar el robot totalmente recto, tocaba la línea negra y como en la línea negra tenemos definido de que el robot siga recto, llegaba al azul de la izquierda y por lo tanto, se salía de la trayectoria.

Para arreglar este fallo, ajustamos la vuelta para que se oriente lo más recto posible.

Además también, este problema se arregló, ajustando un ángulo de giro negativo hacia la derecha cuando el robot tocaba la línea negra, para que así, asegurarnos de que el robot nunca tocaría la línea azul de la izquierda al igual que cuando pasaba por los bloques negros y el movimiento realizado era seguir recto.

Para ello, todas las variables, de giro, se fueron ajustando realizando muchas pruebas hasta encontrar la configuración óptima con el ajuste de todos los parámetros de giro y velocidades.

- **Giro.**

Otro de los problemas comunes que también nos llevó un período de tiempo extenso en arreglarlo fue a la hora de realizar los giros, ya que casi nunca se realizaban de forma exacta, es decir, cuando se llamaba a la función `robot.drive(velocidad_giro, angulo_giro)`, el giro nunca era lo suficientemente exacto.

- **Calibración del color.**

Por último, otro de los problemas fue la calibración de colores, puesto que según la iluminación de luz en el aula, los colores RGB varían y por lo tanto, debíamos de modificar el umbral de los colores de la variable del verde y del negro. Aunque cabe destacar que los colores RGB del negro no hicieron faltar casi nunca calibrarlos puesto que aunque cambiaría la iluminación de luz, su variación era ínfima.

Video de un fallo en los parámetros del robot, la variable de giro para darse la vuelta en la finalización de la entrega del paquete 1 no era suficiente:

[Enlace al video](#)

Video de un fallo por la velocidad del robot que provocaba que leyese 2 veces el color negro en el suelo cuando no debía de hacerlo. También podía fallar al leer el color blanco del suelo por los rayajos que había en el mapa:

[Enlace al video](#)

10. MEJORAS FUTURAS

Tras la exitosa culminación de este proyecto, en el cual hemos logrado realizar todas las tareas requeridas, es importante señalar algunos puntos que quedaron

pendientes y que podrían haber mejorado aún más los resultados obtenidos con la configuración y estructura utilizada.

Como se ha comentado anteriormente en puntos anteriores, el funcionamiento del robot se basaba principalmente en el seguimiento del color verde, teniendo en cuenta que, si se encontraba en el verde, se le aplicaría un ángulo de corrección negativo hacia la derecha para desplazarse hacia el color azul y en caso contrario, es decir, si es distinto de verde, se desplazaría hacia el color verde con un ángulo de corrección positivo hacia la izquierda.

Una mejora significativa que podría haber reducido el tiempo de recorrido de la ruta dada habría sido la implementación del control proporcional.

El control proporcional es una estrategia de control comúnmente utilizada en sistemas de control automatizado, como en robots. La idea básica detrás del control proporcional es corregir el error entre un valor deseado y un valor medido utilizando un término proporcional al error.

Esta incorporación trata de realizar la siguiente operación:

1. Se calculan en primer lugar, los colores RGB del verde y azul.
2. A continuación, se detecta el color RGB que está leyendo el sensor del color actualmente.
3. Una vez calculados todos los colores RGB, se calcula la distancia que existen entre ellos, con la raíz cuadrada de la suma de la diferencia de los colores RGB al cuadrado.

$$\text{Visto de forma matemática: } d = \sqrt{(RV - RS)^2 + (GV - GS)^2 + (BV - BS)^2}$$

donde:

Rv es el rojo objetivo.

Rs es el rojo leído en cada momento.

Gg el verde objetivo.

Gs es el verde leído en cada momento.

Bv es el azul objetivo.

Bs es el azul leído en cada momento.

- Una vez que se haya calculado la distancia, aplicaremos una fórmula, la cual nos dará la w (ángulo de corrección que se tendrá que desplazar el robot). Este valor, se aplicará directamente sobre la variable de giro, es decir, el turn_rate.

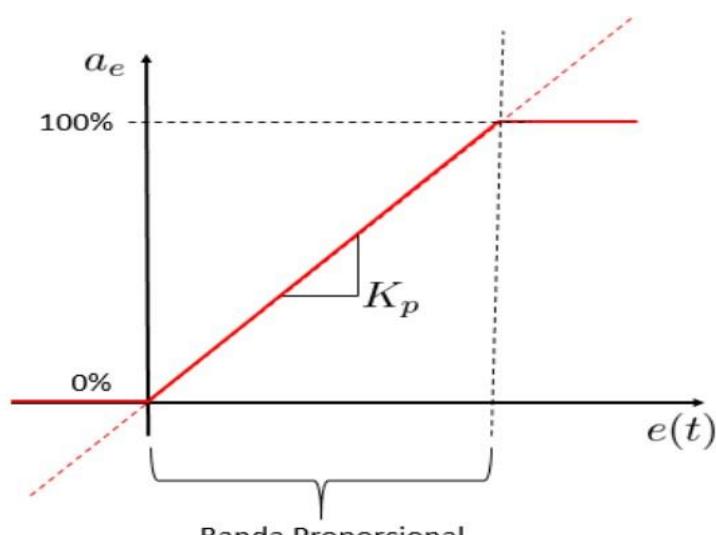
La fórmula a aplicar es la siguiente:

$$w = \frac{2w_{max}}{d_{max} \times d} - w_{max}$$

donde:

w_{max}: es el ángulo máximo que puede girar. d_{max}: es la distancia máxima que puede moverse en el giro.

En resumen, el control proporcional busca calcular la distancia que delimita los colores con el objetivo de calcular el ángulo de corrección que se aplicará en cada momento. Con ello, se notará una mejor suavidad en la corrección del giro y en consecuencia, una mejora del tiempo recorrido debido a una mayor precisión de dicha corrección.



Figura

104. Control proporcional.

11. REFERENCIAS

- [1] [BFS](#)
- [2] [Librería Pybricks](#)
- [3] [Marvel App](#)
- [4] [GanttPRO](#)
- [5] [Visual Studio Code](#)
- [6] [NetBeans](#)
- [7] [Java](#)
- [8] [Java Swing](#)
- [9] [Visual Paradigm](#)
- [10] [Photoshop](#)
- [11] [Photopea](#)

[12] [Hojas de cálculo de Google](#)