Universidad Zaragoza
1542

Trabajo Fin de Grado

Rediseño de una Plataforma de robots móviles

Redesign of a mobile robot platform

Autor/es

Daniel Roche García

Director/es

Cristian Mahulea
Joaquín Ezpeleta

Escuela de Ingeniería y Arquitectura

2021

REDISEÑO DE UNA PLATAFORMA DE ROBOTS MÓVILES

RESUMEN

El diseño y optimización de sistemas multi-robot, es una de las principales líneas de investigación, ya que su adaptación a la vida real facilita el trabajo a las personas. En estos sistemas es importante diferenciar entre el hardware y el software, ya que son dos partes necesarias pero diferentes.

El presente Trabajo Fin de Grado, se centra en el rediseño del software de una plataforma multi-robot de bajo coste. Se parte de un sistema complicado de modificar a uno flexible, general y adaptable a diferentes entornos. Para conseguirlo se utiliza la programación orientada a objetos que permitirá optimizar el sistema actual y facilitar la elaboración de nuevos proyectos sobre este.

Partiendo del desarrollo actual, se comprende el comportamiento de la plataforma y se realiza un análisis iterativo de las necesidades de esta para la implementación del nuevo software. Para conseguirlo, se utilizan clases que en algunos casos representan a los objetos físicos que componen la plataforma, por ejemplo, los robots. También, es necesario identificar con clases partes más abstractas de este desarrollo como son la planificación de trayectorias.

El proceso de diseño requiere de sucesivos análisis y pruebas del funcionamiento para conseguir obtener un sistema capaz de ser modificado fácilmente. Este Trabajo Fin de Grado pretende facilitar la adaptación de nuevos agentes, operaciones, etc. en fututos Trabajos Fin de Grado o de Master.

ABSTRACT

The design and optimization of multi-robot system is one of the main research areas because their adjustment to real life makes people work easier. In these systems it is important to distinguish between hardware and software, as they are two important necessary and different parts.

This Final Degree Project focuses on the low-cost platform software redesign. Starting from a system that is complex to modify to achieve a flexible, general, and adaptable system. To get it, object-oriented programming is used to optimize the current system and to make easier the creation of new projects.

Based on the actual development, the platform performance is understanded and an iterative analysis is done to the new software implementation. To achieve this, classes are used that represent the physical components of the platform, for example robots. It is also necessary to identify more abstract parts, like path planning, with classes.

The design process requires consecutive analysis and running checks to create an easier adjustable system. This Final Degree Project intend to make easier the adjustment of new components in future projects.

Tabla de contenido

Capítulo 1

Introducción

A lo largo de este apartado se introducirán los objetivos, así como el alcance del trabajo, para posteriormente, desarrollar más detenidamente las diferentes partes del proyecto.

1.1 Objetivos

Desde un punto de vista técnico, el proyecto tiene como objetivo rediseñar el código de una plataforma de robots móviles con el fin de que adquiera un carácter general y pueda ser reutilizado para otras plataformas. Para ello, se ha seguido un método de diseño orientado a objetos que se explicará a lo largo del trabajo y que permite organizar y hacer más eficiente el código.

Por otro lado, este rediseño permitirá obtener un sistema más flexible al que se le podrán añadir diversas funcionalidades más fácilmente, sin necesidad de conocer el código de una forma exhaustiva. Para poder llevar a cabo este diseño, es necesario conocer el funcionamiento actual de esta y observarlo desde un punto de vista general. Es decir, el proyecto que se lleva a cabo se realiza con el fin de que el código pueda ser reutilizado en otras plataformas, similares o no.

Además de las referencias a las metodologías de programación y la obtención de conocimientos del lenguaje, la plataforma cuenta con la implementación de algoritmos de evitación de bloqueos y de planificación de trayectorias los cuales son objeto de conocimiento y de estudio.

Desde un punto de vista personal, se pretende ahondar en el conocimiento de la ingeniería de sistemas, que ocupa una parte fundamental en el mundo de la ingeniería y con un desarrollo creciente en el mundo Industrial. El conocimiento de lenguajes de programación, así como de metodologías de diseño son imprescindibles hoy en día para cualquier ingeniero. Permiten realizar tareas en muchos casos automatizadas y adaptadas a las necesidades de los diferentes ambientes.

Uno de los objetivos principales, era la introducción personal en los lenguajes de programación y en la ingeniería de sistemas mediante la realización de una aplicación real, en este caso la plataforma multi-robot, con el fin de ampliar los conocimientos en esta rama.

1.2 Contexto y alcance

Este trabajo se realiza dentro del departamento de Informática e Ingeniería de Sistemas donde se dispone de líneas de investigación en robótica y automatización. Para llevar a acabo tareas de docencia e investigación, se ha desarrollado desde el departamento, una plataforma de bajo coste gestionada por un software que el presente trabajo trata de mejorar y que se describe posteriormente.

Con el objetivo ya definido, se va a determinar el alcance del proyecto, es decir, qué se va a llevar a cabo a lo largo de este y qué no. La plataforma multi-robot sobre la que se realiza el proyecto, consta de diferentes métodos de planificación de trayectorias, algoritmos de evitación de bloqueos, comunicación bidireccional con los robots, etc. Por eso, a lo largo de este proyecto se va a adaptar el código con una nueva visión orientada a objetos, realizando modificaciones en el código actual, que posteriormente se comentarán, con el fin de hacer el diseño eficiente, claro y flexible.

La plataforma, actualmente, tiene un diseño monolítico donde es complicado adaptar cualquier cambio o añadir nuevos elementos como tipos de planificaciones, algoritmos de evitación de bloqueos, etc. Además, el diseño actual es complicado de adaptar a otro tipo de plataforma diferente. Por eso en este trabajo se va a realizar un rediseño orientado a objetos que aporta una serie de ventajas que se comentarán posteriormente.

1.3 Metodología

La metodología, es una de las principales partes que se cierran al comienzo del trabajo, ya que permiten conocer como y con qué se va a realizar, así como las líneas que se van a seguir.

En cuanto al código, se utiliza un lenguaje de programación de alto nivel, en este caso C++, que ya había sido utilizado previamente en el desarrollo de la aplicación actual. Su utilización se debe a que es un lenguaje muy apropiado para el diseño orientado a objetos, así como con muchas posibilidades de aplicación.

Por otro lado, es necesario el conocimiento de la programación orientada a objetos para llevar a cabo el rediseño de la plataforma. Desde un punto de vista general, es imprescindible para ver qué es lo que se necesita (objetos) y cómo se puede implementar (clases). Posteriormente, se hablará acerca de este método de programación.

Finalmente, también es necesario conocer el comportamiento de la plataforma actual y tener una visión general de esta, así como de sus principales componentes que más tarde se comentarán.

1.4 Planificación

En este apartado, se va a indicar cuáles han sido las fases por las que ha pasado la realización del proyecto, así como el proceso iterativo que se ha seguido para conseguir el funcionamiento del sistema:

- Adquisición de conocimientos en el lenguaje de programación C++ utilizando documentación escrita y on-line.
- Estudio y conocimiento de la programación orientada a objetos.
- Conocimiento de la lógica que involucra el sistema con el fin de tener una visión general.
- Planificación inicial de las clases necesarias para la orientación a objetos.
- Desarrollo iterativo de las diferentes partes mediante reuniones sistemáticas con los tutores, con la correspondiente resolución de errores y toma de decisiones.
- Implementación de las diferentes clases diseñadas en el programa y comprobación de funcionamiento.
- Obtención de resultados y conclusiones.

A lo largo de todo el proyecto, se han realizado reuniones con los tutores donde se han ido planteando objetivos a corto plazo ya que, al ser un sistema de elevada complejidad y cantidad de código, se optó por ir probando las implementaciones sucesivamente, para posteriormente unirlas como se explicará en el desarrollo más adelante.

Por otro lado, durante la realización, se han ido obteniendo posibles áreas de desarrollo y de mejora de la aplicación, que pueden ser objeto de futuros proyectos, ya que esta es una aplicación con grandes posibilidades para aquellos alumnos que se quieran formar en la ingeniería de sistemas.

1.5 Contenidos

En este apartado, se resumen los capítulos que se van a encontrar a lo largo del trabajo y que lo estructuran.

1. **Introducción:** sirve como resumen de los objetivos tanto personales como técnicos del Trabajo de fin de Grado. También permite situar al lector acerca del tema y su contexto.
2. **Estado del arte:** explica, de forma resumida, los conceptos esenciales necesarios qué más tarde se utilizarán en el desarrollo del trabajo.
3. **Plataforma multi–robot:** se describen los componentes y funcionamiento de la plataforma sobre la que se realiza el trabajo.

4. **Rediseño orientado a objetos:** se explica el proceso para cumplir el objetivo del proyecto, así como las implementaciones.

5. **Conclusiones y líneas futuras:** se plantean las conclusiones y se plantean posibles desarrollos futuros a partir del trabajo realizado

Capítulo 2

Estado del arte

A lo largo de este capítulo se comentan algunas de las partes más importantes del proyecto desde un punto de vista técnico, que más tarde se utilizaran para el desarrollo de la aplicación.

Además, se comentarán aspectos relacionados con los robots-móviles ya que en los últimos años su uso se ha incrementado considerablemente en la industria y en el mundo de la ingeniería. Esto se debe a que facilitan el transporte y permiten llevar a cabo tareas complicadas, peligrosas para las personas o simplemente son más eficientes.

En primer lugar, se explicará el funcionamiento y características de los sistemas multi-robot para dar una idea general acerca de la aplicación y posteriormente se explicarán aspectos relacionados con el diseño y la programación orientada a objetos.

2.1 Sistemas multi-robot

Un sistema multi-robot, es aquel que es capaz de conseguir un objetivo mediante la cooperación de los diferentes agentes que forman el sistema. Los objetivos que plantean pueden ser muy variados y podemos encontrar aplicaciones para operaciones de mapeado en diferentes entornos (Roldán et al. (2016)), para operaciones de vigilancia con el fin de detectar objetos dinámicos (Rodríguez et al. (2014)), así como para el transporte o la obtención de diferentes puntos en un espacio determinado en la que se basa la aplicación de este trabajo.

El uso de sistemas multi-robot ha sufrido un crecimiento importante en los últimos años ya que permite adaptarse a diferentes tareas en las que se necesitan cooperaciones por parte de varios agentes. Además, también se utilizan ya que presentan beneficios respecto a la realización de las mismas tareas con sistemas de un único componente. Una de las preguntas más frecuentes en este contexto es el por que de la utilización de sistemas multi-robot y algunas de sus características ventajosas son (Roldán et al. (2019), Arkin y Balch (1998)): la **efectividad** de estos sistemas gracias a la suma de recursos, la **eficiencia** que presentan debido a que cada robot realiza una tarea individual que puede ser optimizada, la **simultaneidad** de las tareas individuales entre los componentes, la **simplificación** de los sistemas ya que cada componente realizando una tarea simple es más eficaz que un solo elemento realizando múltiples operaciones, la **flexibilidad** para adaptarse a los distintos espacios así como la **tolerancia a fallos.**

En cuanto a la c**lasificación** de estos sistemas existen diferentes taxonomías en función de los autores y en función de los comportamientos de las aplicaciones. No existe una clasificación fija, si no que en la mayor parte de casos se basan en la forma de realizar las actividades, la forma de comunicarse, etc. Un tipo de clasificación que podemos encontrar (Verret, 2005) es la siguiente: en función de la **arquitectura** se pueden encontrar sistemas centralizados o descentralizados dependiendo del tipo de control del sistema; dependiendo del **tipo de comunicación**, que puede ser implícita donde son necesarios sensores para captar estímulos exteriores (Yan et al. (2013)) o

explícita donde los robots se comunican de forma directa entre ellos; en función del **tipo de cooperación** ya que pueden ser cooperativos, cuando los agentes realizan una serie de objetivos individuales para conseguir un objetivo final, o competitivos donde cada uno intenta realizar la tarea lo mejor o más rápidamente posible; dependiendo del **objeto de transporte**, como su utilización para transportes coordinados de objetos a través de la coordinación de varios robots (Trebi et al. (2002)); dependiendo de la l**ocalización** de estos sistemas

2.2 Programación orientada a objetos

Una parte imprescindible de este proyecto es el enfoque de programación orientada a objetos y su relación con la ingeniería del software. A lo largo de este apartado se van a presentar los aspectos más importantes del paradigma orientado a objetos, sus elementos y los pasos que hay que realizar para llevar a cabo un buen diseño.

El paradigma orientado a objetos (Bóveda et al. (1999)) surge ante la necesidad de plantear diferentes modelos, y en algunos casos modelos tangibles en el mundo natural. Durante mucho tiempo, se utilizó para programar ordenadores, pero posteriormente ante la complejidad de los proyectos de software y gracias a sus características, se convirtió en un paradigma ampliamente aceptado y utilizado. Algunas de sus ventajas respecto a otras formas de diseño (Pressman,1998) se presentan en la siguiente tabla:

Ventaja	Descripción
Reutilización	Los objetos se pueden volver a usar indefinidamente
Eficiencia	Se consiguen diseños más eficientes y rápidos
Flexibilidad	La creación de clases permite el uso de estas por varios usuarios sin necesidad de un conocimiento exhaustivo.
Adaptabilidad	Un diseño orientado a objetos es más flexible y se adapta a cualquier entorno.

Tabla 1. Ventajas de la Programación orientada a objetos

2.2.1 Conceptos básicos y proceso de desarrollo

A continuación, se introducen algunos de los elementos que se utilizan para llevar a cabo el rediseño objetivo del proyecto. Los conceptos de estos elementos han sido debatidos por varios autores y caben destacar los siguientes: las **clases y objetos** agrupan un conjunto de características (atributos) y operaciones (métodos) que representan una abstracción de la realidad; los **atributos** están ligados a una clase y almacenan la información característica de esta, suelen ser tipos de datos u otras clases; los **métodos** permiten un tratamiento de datos que se incluyen en los atributos así como los que se pasan como parámetro; la **herencia** es una propiedad propia del diseño

orientado a objetos y permite establecer relaciones entre superclases y subclases (que heredan los atributos y operaciones de un superclase); el **mensaje** hace referencia a la comunicación establecida entre los diferentes objetos que provocan estímulos entre ellos con el fin de activar una operación o realizar alguna modificación.

Se pretende dar una visión general acerca del proceso para desarrollar sistemas orientados a objetos. Este proceso suele seguir un **método iterativo** (Pressman ,1999) que comienza con los requerimientos iniciales por parte del usuario, donde el primer paso es la identificación de las clases esenciales. Posteriormente, será necesario buscar si esas clases ya existen o han sido creadas. Si no es el caso, comienza la creación de las clases donde se deberá seguir la siguiente secuencia: análisis, diseño, programación y pruebas. Para concretar, se van a explicar de forma general las fases de desarrollo de software.

En primer lugar, es necesario **identificar las clases** que van a ser necesarias mediante la observación del problema planteado. Para poder establecer las clases del problema se pueden acudir a diferentes clasificaciones que permiten determinar si un objeto de la realidad puede ser identificado con una clase.

Una de estas fases, es **el análisis orientado a objetos** que cuenta con multitud de métodos y de autores que comentan la mejor forma de realizar el análisis. Entre ellos destaca el método de Booch (Booch ,1999). Sin embargo, todos los métodos siguen un proceso para establecer las relaciones entre las clases, y determinar sus aplicaciones futuras.

El siguiente paso es el **diseño.** En él comienza la construcción del desarrollo de software que continuará con la programación y las pruebas. El objetivo del diseño es obtener una solución para el problema planteado gracias al análisis anterior. Por otro lado, el diseño es un proceso complejo que requiere amplios conocimientos (métodos, tipos de datos, estructuras, etc.).

Finalmente, las **pruebas** son la herramienta más adecuada para la detección de errores cometidos en etapas anteriores de la iteración análisis-diseño. La detección de estos errores ahorra tiempo y esfuerzo debido a la posible definición de clases o subclases innecesarias, así como atributos o métodos que no hayan sido adecuadamente diseñados por falta de entendimiento del problema.

2.3 Modelado de objetos con UML

En los comienzos de la programación orientada a objetos existía una variedad de modelos de representación de software y surgió la necesidad de unificarlos para encontrar uno general y adecuado para todos los diseños (Arlow ,2006).

El nombre se corresponde con "*Unified Modeling Language"* o "*Lenguaje Unificado de modelado"* y surge ante la necesidad de representar gráficamente las abstracciones de la realidad que se realizan mediante el diseño de software. Permite representar un sistema de forma legible para el resto de las personas, así como las relaciones entre las diferentes partes de forma rápida y eficaz.

2.3.1 UML en objetos y clases

En este apartado se pretende dar una idea general acerca de la representación de sistemas orientados a objetos, por medio del uso de UML. Como ya se han comentado antes las partes más importantes de la programación orientada a objetos, esta parte se basa en la representación y las relaciones entre ellas.

En cuanto a la notación que toman las clases en la representación UML se puede ver mediante el siguiente ejemplo:

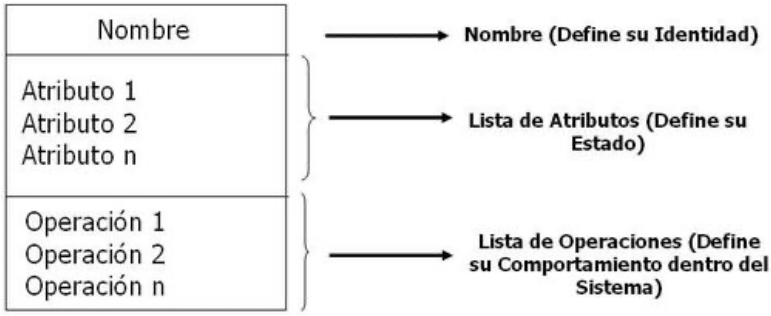


Figura 1. Representación de una clase con UML. Recuperado de: http://programandoenjava.over-blog.es/article-el-uml-o-lenguaje-de-modelado-unificado-como-herramienta-en-el-modelado-de-objetos-53386438.html

Como se puede ver en la figura 1, la representación dispone de tres compartimentos: **compartimento del nombre** donde se introduce el nombre de la clase, **compartimento del atributo**, donde se describen los atributos de la clase y **compartimento de operaciones** donde se definen los métodos.

En lo referido a las relaciones entre las clases u objetos podemos encontrar las que aparecen en la figura 2:

- **Asociación**: se corresponde con los vínculos entre objetos.
- **Herencia o generalización**: el origen se corresponde como una especificación del destino. En POO se utiliza para la herencia, que se comentó su significado previamente.
- **Implementación o realización:** el origen lleva a cabo el trabajo que especifica el origen.
- **Dependencia:** el origen depende del destino y varía en función de las modificaciones del destino.
- **Agregación:** como su nombre indica, el destino es una parte del origen.
- **Composición:** tiene el mismo significado que la agregación, pero es más restrictiva.

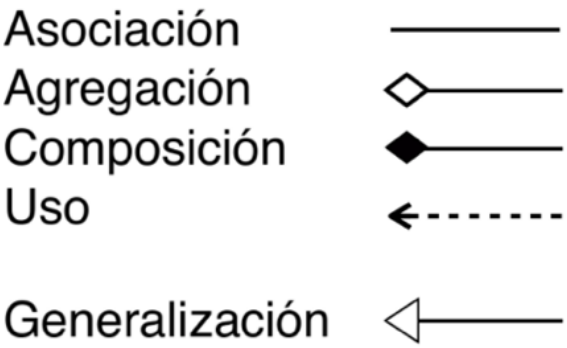


Figura 2. Relaciones en diagramas UML. Recuperado de: https://pertusa.gitbooks.io/programacion-2/content/poo/poo.html

Capítulo 3

Plataforma multi-robot

La plataforma multi-robot sobre la que se realiza el proyecto esta situada en el laboratorio L05.a del edificio Ada Byron en la Escuela de Ingeniería y Arquitectura.

El objetivo de este apartado es comentar su funcionamiento y los elementos que integran el sistema. Se comienza comentando los elementos que la componen:

3.1 Elementos

3.1.1 Robots

El sistema consta de varios robots móviles del tipo “Turtle 2WD Mobile Robot”, pequeños y de bajo coste que se desplazan a través de la plataforma para alcanzar unas configuraciones finales.

Estos robots funcionan con una placa Arduino del tipo DFRobot RoMeo V2.2 , pero que a la hora de programar se comporta como Arduino Leonardo (García et al. (2020)). Dispone a su vez de un microcontrolador que permite el movimiento de este, así como una antena Xbee que permite la comunicación entre los robots y la CPU.

En cuanto a la estructura, dispone de una estructura de aleación de aluminio que se desplaza sobre dos ruedas de goma, ideal para terrenos interiores como los que se van a utilizar. Dispone a su vez de una rueda de metal en la parte inferior que permite el giro sobre sí mismo, así como dos servomotores que aportan acción a las ruedas. Estos motores funcionan gracias a la batería/pilas que se colocan en su parte inferior.

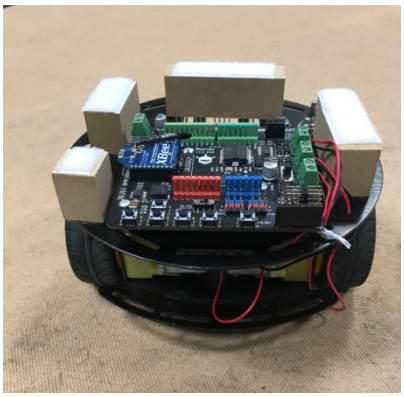


Figura 3. Robot. Vista sin marcador ArUco.

Figura 4. Robot. Vista con marcador.

3.1.2 Plataforma

La estructura sobre la que se mueven los robots consiste en un conjunto de paneles que forman un rectángulo de dimensiones 3.32 x 2.34 m, sobre los que se sitúan los marcadores ArUco, que más tarde comentaremos, y los obstáculos (o regiones de interés en este proyecto) que esquivan los robots.

Como se puede ver en la figura 5 el rectángulo dispone de ocho marcadores, cuatro elevados y cuatro a ras de suelo que permiten calcular la distancia a la que se encuentran los robots (Barrio ,2016). La luz es un factor determinante para la detección de marcadores como se explicará más adelante y se comenta en el trabajo antes mencionado. La plataforma está rodeada por rectángulos de colores que modelan el filtro utilizado para la detección de obstáculos.

Los obstáculos que aparecen tienen el mismo color que los rectángulos de alrededor y como se puede ver hay de varios tipos y como novedad de ese trabajo, su posición se introduce por parte del usuario.

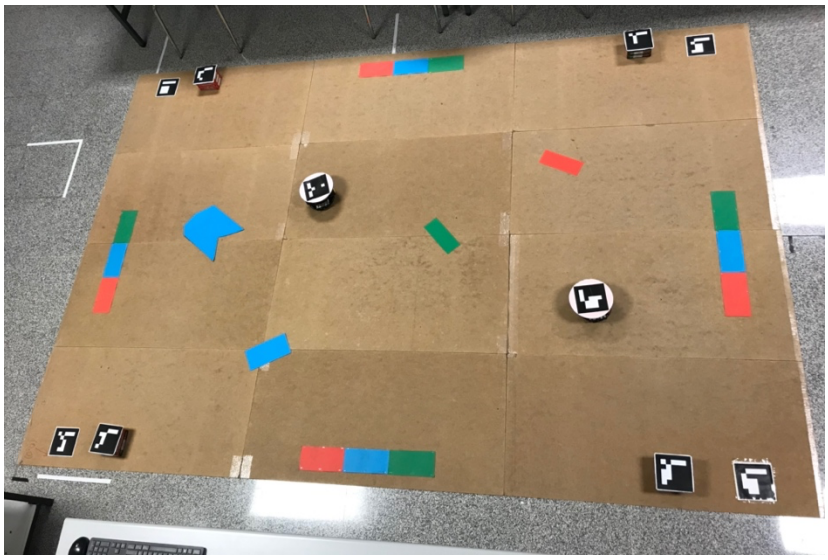


Figura 5. Plataforma montada. Elaboración propia

3.1.3 Cámara

Es una parte imprescindible del sistema, ya que desde una posición cenital y gracias a las librerías de OpenCv para C++ detecta los marcadores ArUco y permite obtener la posición de los robots.

La cámara que se utiliza es Microsoft LifeCam Studio 1080p HD de pequeñas dimensiones y compatible con la mayor parte de sistemas operativos. Dispone de varias resoluciones para grabar que son: 640x480, 1280x720 y 1920x1080. En este trabajo se realiza con la máxima resolución.

En este sistema se trabaja con librerías de OpenCv y ArUco, que permiten el tratamiento de imágenes por fotogramas y son necesarias para la implementación de métodos en el código.

Figura 6. Microsoft LifeCam Studio 1080p. Recuperado de: https://www.microsoft.com/es-xl/accessories/business/lifecam-studio-for-business?activetab=overview%3aprimaryr2

3.1.4 Comunicación

La comunicación se lleva a cabo gracias a la utilización de antenas Xbee que permiten una comunicación inalámbrica entre ellas. El sistema sigue una metodología loT que permite la comunicación entre robots y ordenador.

Aparte de la comunicación inalámbrica es necesaria la comunicación serie, ya que como se comentará más adelante, se envían pequeños paquetes de información que son transmitidos desde la antena al dispositivo inteligente de la misma forma en que se reciben. A continuación, se muestra una imagen de las antenas antes mencionadas.

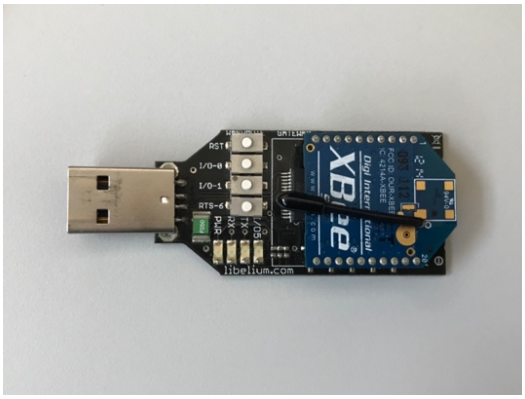


Figura 7. Antena Xbee.

3.2 Funcionamiento actual

En este apartado, se pretende hacer un resumen acerca del funcionamiento del sistema multi-robot actual con el fin de situar al lector y poder entender los cambios realizados en este proyecto.

Cabe destacar que el sistema ha sido optimizado por varios alumnos y profesores que han dedicado y dedican su tiempo aprendiendo e investigando. Es el caso de los trabajos comentados antes de García (2020) y Barrio (2016), así como otros.

3.2.1 Comunicación con el usuario

El programa se ejecuta y compila en uno de los ordenadores del laboratorio, dedicado a la plataforma y que utiliza el sistema operativo Linux con su distribución Ubuntu, con la versión 16.04. El usuario coloca los robots y los obstáculos en las posiciones iniciales que quiera. Una vez ejecutado el programa el usuario introduce las posiciones que quiere que alcance cada uno de ellos.

La siguiente información que debe aportar el usuario, es el método de planificación de trayectorias y de partición del espacio, así como el algoritmo de evitación de bloqueos (para evitar los obstáculos), como por ejemplo el Algoritmo del Bankero.

Finalmente, el programa va imprimiendo por pantalla las posiciones que ocupan los robots, la comunicación de estos, así como las posiciones de los obstáculos.

3.2.2 Proceso de control

Se establecen dos “threads” de ejecución una vez que se obtienen los puntos por parte del usuario. De esta forma se consigue la comunicación bidireccional con los robots. Uno de estos threads se corresponde con la función “listen_response_WIFI ()” encargada de recibir los datos procedentes de los robots mediante la comunicación serie.

El otro thread responsable del control es la función “ConcurrentNavigationProcess ()”. Estos dos “threads” se ejecutan de forma simultánea y así se facilita la comunicación bidireccional. Dentro de esta función se utilizan los recursos necesarios para que los robots se comuniquen y eviten los obstáculos antes comentados.

3.2.2.1 Sistema de visión

La primera parte de la función se corresponde con la inicialización de las variables locales necesarias. A continuación, mediante **la clase “Eng”** se introducen los puntos correspondientes a los rectángulos que actúan como referencia para los obstáculos, mediante la función “set_Points”.

Una vez situados los puntos, comienza un bucle que se repetirá cada 40 ms si la cámara sigue grabando, donde se detectarán tanto las posiciones de los robots como las de los obstáculos, gracias a las librerías de OpenCv y ArUco, mediante la función de la clase “Eng” con la función “Eng_Process”. Gracias a la lógica introducida en esta

función con los booleanos “aver” y “go” los obstáculos solo se detectan una vez y a partir de ese momento la variable aver pasa a “true”, de forma que el bucle puede continuar. Por otro lado, la posición de los robots se detectará continuamente en el bucle.

3.2.2.2 Planificación de trayectorias y comunicación

El siguiente paso, es utilizar un contador para identificar si es la primera iteración del bucle do-while y se pasa un filtro para los obstáculos, de forma que los que sean inferiores a un área determinada no son considerados como tal. Posteriormente, el tamaño de estos es enviado a los robots mediante la función de comunicación “send_ObstaclesPacket”.

En el caso de que sea la primera iteración, se le pide al usuario que introduce el tipo de partición y de planificación de trayectorias que quiere realizar mediante la función “PlanificationPathProcess ()”, que dispone de varios métodos de partición y de planificación. De esta forma, se obtienen las trayectorias que deben de seguir los robots para alcanzar el punto de destino sin chocar con los obstáculos.

3.2.2.3 Algoritmo de evitación de bloqueos

El siguiente paso también es bloqueante ya que se da la opción al usuario de elegir entre varios algoritmos. García (2020) introduce el Algoritmo del Bankero, para lo que sigue una programación orientada a objetos, creando una biblioteca de clases en “Banker Algorithm”. También está implementado el algoritmo de evitación de bloqueos mediante el cálculo de sifones.

El primer paso es adaptar los datos obtenidos hasta ahora para llevar a cabo el Algoritmo del Bankero mediante una clase “Syst” que modela la ejecución. La ejecución del programa dependerá del número de robots utilizados. Es necesario comprobar que el estado inicial de los robots sea seguro en primer lugar. En el caso de que solo se utilice un robot no será necesario analizar cada cambio de estado, pero en el caso de que haya varios, es necesario un trabajo colaborativo entre todos y para ello se crean las clases necesarias como “nextState” que analizarán los futuros estados de los robots para saber si son seguros y entonces proceder a su movimiento.

También se ha trabajado en el control interno de los robots e introduce mejoras que permiten el movimiento colaborativo de estos permitiendo que, si un robot se está desplazando, el resto pueda ir reorientándose, ahorrando tiempo y haciendo más eficiente el movimiento. Para ello implementa una lógica utilizando más clases como “list_FutureStates”, que permite evitar los tiempos de espera antes comentados. Para ello utiliza las siguientes funciones, ahora en funcionamiento en la plataforma.

- “GetNextStateByBanker ()”
- “control_RobotStateBA ()”
- “checkStatus ()”
- “sendSubPaths ()”
- “sendPosition ()”

Capítulo 4

Rediseño orientado a objetos

En este capítulo se va a desarrollar el proyecto llevado a cabo, cuyo objetivo es el rediseño de la plataforma ya comentada, mediante una programación orientada a objetos. Lo que se pretende es conseguir una abstracción de esta, con el fin de poder aplicar este diseño a cualquier otro sistema multi-robot, y poder reutilizar las clases creadas. El diseño con clases permitirá añadir métodos a estas en futuros proyectos, hacerla más completa y flexible de cara a su uso en otros entornos.

No se pretende poner en funcionamiento otro sistema sino reutilizar lo máximo posible el que está ahora e incluir mejoras que ayuden a un funcionamiento más eficiente. La información que aquí se recoge; se completará más adelante en los anexos.

4.1 Proceso de diseño

En el capítulo 1 se ha comentado la metodología que se ha seguido a lo largo del proyecto, pero aquí, se va a incidir un poco más en los pasos realizados con detalle.

Una vez se tiene conocimiento en C++ y en programación orientada a objetos se van a ir realizando sucesivas reuniones donde se proponen pequeños objetivos. Estas metodologías se conocen como “Metodologías ágiles” y permiten obtener un objetivo en un periodo corto de tiempo y de forma flexible adaptándose a las condiciones del proyecto.

Se realizan pequeñas entregas en un corto periodo de tiempo por parte del equipo de trabajo y se analizan los resultados buscando en el proyecto aspectos susceptibles de mejora (ineficiencias, errores, inconsistencias, etc.). En este proyecto se realizan sucesivas reuniones donde el proceso que se sigue es:

- **Análisis del desarrollo** actual y posible diseño futuro.
- **Objetivos** a corto plazo.
- **Diseño** de estos objetivos y sucesivas **pruebas** de funcionamiento.
- Presentación de resultados, **modificaciones y mejoras**.

Este proceso se ha ido repitiendo a lo largo del proyecto y a continuación se muestra una tabla con las reuniones y un resumen de los objetivos planteados en cada reunión:

Número de reunión	Objetivo
1	Objetivo final del proyecto y conocimiento del sistema actual
2	Presentación inicial de posibles clases
3	Desarrollo conceptual de las clases ideadas
4	Creación de clase Camera e investigación de librería OpenCv
5	Refinamiento de la clase Camera, creación clase Scene, creación del makefile e investigación UML
6	Refinamiento clase Scene, investigación Json, creación clase Robot y ROI
7	Refinamiento de Robot y ROI, creación de clase PlanningPath
8	Creación de Clase Controller

Tabla 2. Secuenciación de reuniones y objetivos. Elaboración propia.

Estas reuniones son indispensables ya que en ellas tanto los directores como el alumno aportan sus ideas, con el fin de encontrar la solución que mejor se adapte a la plataforma en este caso. Como se puede ver en la tabla superior hay varias reuniones en la que los objetivos son más conceptuales ya que entender el funcionamiento de sistema multi-robot requiere de tiempo e información para hacer un buen diseño posterior. Conforme se avanza en el tiempo las reuniones se centran en el análisis y diseño de las clases que posteriormente se explicarán.

El proceso de aprendizaje de programación es lento durante las primeras partes del trabajo ya que los errores cuesta tiempo localizarlos. Una vez se avanza en el trabajo la programación es significativamente más ágil. A su vez la compresión del sistema se va mejorando conforme pasa el tiempo y las reuniones. Por otro lado, como se puede ver en la tabla siempre se produce un refinamiento del código creado, ya que van surgiendo nuevas ideas durante el proyecto o es necesario adaptar el código.

La clase sobre la que más decisiones se han tomado y qué más tiempo ha llevado para programar ha sido “Scene” ya que al final es la que engloba el funcionamiento de los robots, las regiones de interés, etc.

4.2 Visión general

En este apartado se va a presentar mediante un modelo UML una descripción general del sistema y las partes principales del rediseño de la plataforma. Gracias a UML se puede presentar el diseño del sistema de forma inteligible para alguien que no conoce el funcionamiento. En la siguiente figura se puede ver una representación:

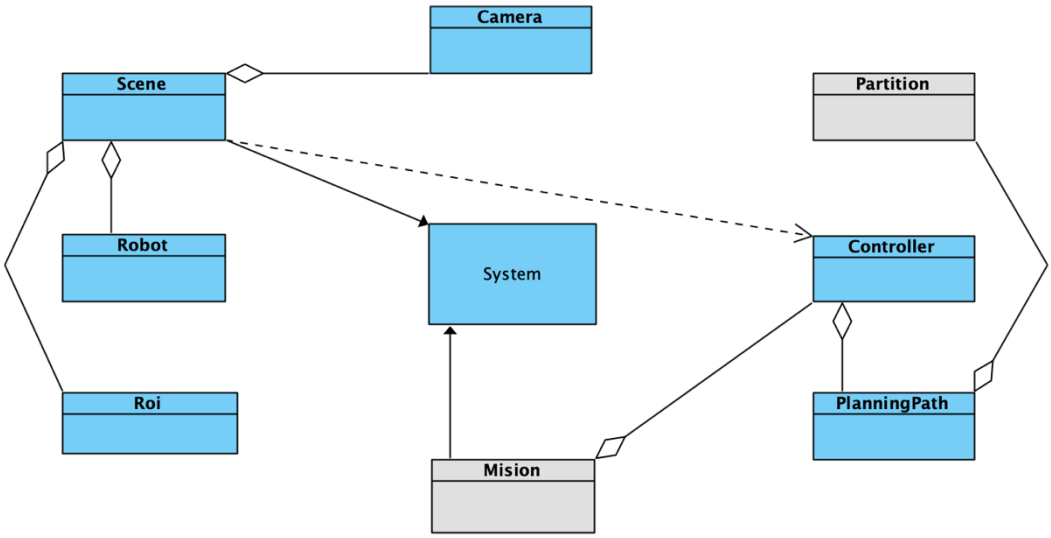


Figura 8. Relaciones entre clases.

En la figura 8, se representan las clases que se han considerado necesarias para el rediseño del sistema, así como sus dependencias y como todas ellas ayudan a la creación del sistema. Se pueden ver dos tipos de colores, el azul se corresponde con aquellas clases que han sido creadas y el resto aportarían una mayor eficiencia al sistema y que pueden ser objeto de trabajos futuros. Sin embargo, la mejora es significativa con las clases ya creadas.

La parte principal de este proyecto se ha basado en el desarrollo de las clases que modelan el sistema. Este proceso consiste en observar el sistema físico y ver como se pueden implementar en el software para conseguir una conexión hardware – software. Por eso, la clase Robot que se ve en la imagen superior hace referencia al robot real utilizado en la plataforma.

4.2.1 Camera

Se crea una clase Camera que representa el sistema de visión de la plataforma donde se incluyen los datos necesarios a almacenar en forma de atributos, que están explicados en el Anexo I, así como los métodos correspondientes. En esta clase es importante la utilización de las librerías de OpenCv y ArUco que están ya instaladas en el ordenador del laboratorio.

Mediante los constructores de la clase se puede crear la instancia inicializando el valor de los atributos de esta que se encuentran en un archivo Json que está explicado en el Anexo II. Esta nueva implementación con los archivos Json permite una lectura más sencilla, pero necesita de la instalación de unas librerías para C++.

Esta clase pretende representar a la cámara que se utiliza en la plataforma, pero que podrían ser muchas en función del sistema y con su constructor podríamos crear las que fueran necesarias si conocemos sus atributos. Una de las posibles mejoras que se podrían realizar sería añadir un método que permitiera la calibración de cada cámara, para hacerlo todavía más general.

En la figura 8, entre la clase “Camera” y la clase “Scene” existe una relación de agregación ya que representa una parte de este. Se podrán almacenar todos los tipos de “Camera” que sean necesarios.

4.2.2 Robot

La creación de la clase robot, es una abstracción del robot físico real que se encuentra en la plataforma. Cuenta con una serie de atributos que lo caracterizan y con unos métodos entre los que se encuentran las funciones de comunicación. Por ejemplo, la clase robot tiene un atributo llamado “Position” donde se almacena la posición de este en las coordenadas x e y que se obtiene a través del escenario. La instancia creada en el programa es capaz de comunicarle al robot físico esa posición a través de la comunicación serie. A su vez dispone de otras funciones de comunicación necesarias para el funcionamiento de la plataforma.

En cuanto a las funciones de comunicación se plantearon dos formas de diseño: la creación de una sola función que, a través de los parámetros de entrada, decidiera que enviar al robot o implementar las funciones de comunicación de forma separada. Finalmente, se optó por esta última ya que la comunicación con los robots reales se produce de forma reiterada y este diseño es mucho más eficiente ya que reduce los tiempos de procesamiento por parte del programa.

Por otro lado, los constructores de la clase permiten asignar un valor inicial a los atributos a través del archivo Json creado donde se recogen las necesidades del usuario. La creación de estas instancias robot se almacenan en una clase que se comentará posteriormente conocida como “Scene” y que se corresponde con la plataforma.

Como ocurre con la clase “Camera” tiene una relación de agregación con la clase “Scene” ya que las instancias creadas de robots se almacenan en “Scene”. En este caso la plataforma dispone de tres robots, por lo que se crearán tres instancias robot.

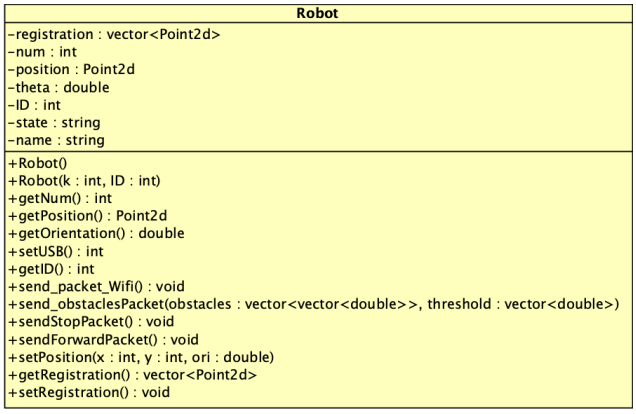


Figura 9. Diseño clase robot UML.

Figura 10. Representación real de la clase robot.

4.2.3 ROI

Se ha cambiado el nombre, ya que hasta ahora se conocían con el nombre de obstáculos. Con regiones de interés se hace la clase más general ya que puede ser utilizada para otro propósito. Esta región modela en este caso las figuras de colores que se obtienen por parte de la visión en el escenario. Como el resto de las clases dispone de sus atributos y funciones que se explican en el Anexo I y todas estas instancias serán almacenadas a su vez en la clase "Scene" como ocurre con los robots y las instancias "camera", de ahí la relación de agregación que se muestra en la figura 8.

En esta clase, se almacenan valores como los puntos que modelan el contorno de la región de interés, su centroide y su área de influencia. Este diseño, permitirá utilizarlos como elementos independientes y no como listas de puntos que se utilizaban hasta ahora.

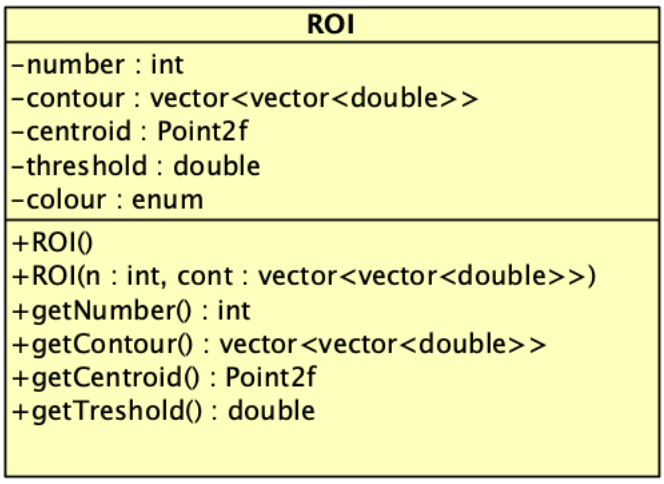


Figura 11. Diseño clase ROI UML.

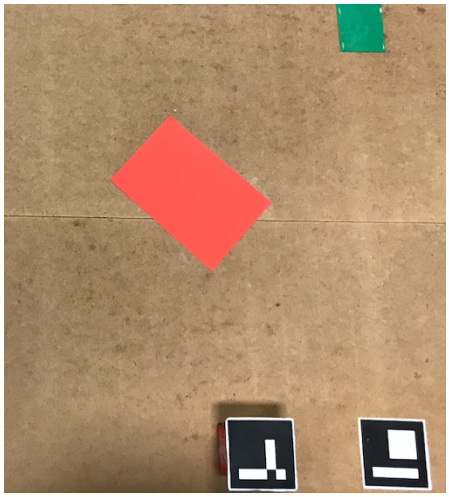


Figura 12. Representación real de la clase ROI.

4.2.4 Scene

Esta clase modela el escenario, que a su vez contará con las clases “Camera”, “Robot” y “Roi” que se almacenan en los atributos con listas de las clases correspondientes. Así, por ejemplo, podríamos tener una plataforma con 4 cámaras, 6 robots y 20 ROI que estarían almacenados con sus atributos correspondientes en la instancia creada de Scene.

Una de las funciones de esta clase es la detección de las regiones de interés. Para su implementación se ha utilizado las funciones de la clase “Eng” donde se han separado la detección de obstáculos de la detección de robots que estaban unidas. Además, se introduce una modificación dentro de la detección de robots, que permite la actualización de la posición de cada robot cada vez que se detecta, lo que mejora la eficiencia gracias al encapsulamiento del código.

De esta forma se consigue un desarrollo más secuencial y estructurado. Para conseguir esto, se han desarrollado pruebas para la detección de los marcadores, los robots y los obstáculos grabando un video mediante la creación de un programa gracias a las librerías de OpenCv. De esta forma, se puede trabajar desde cualquier lugar y no es necesario montar la plataforma cada vez que se quiere hacer una prueba.

Por otro lado, se ha creado una lógica en el constructor de forma que se lee el archivo Json y se van almacenando instancias de la clase “Robot” y de la clase “Camera” en función de los datos introducidos por el usuario. Además, esta clase “Scene” tiene como métodos el filtro de obstáculos, el cálculo del área de influencia de los obstáculos y el cálculo de los centroides de las regiones de interés, entre otros.

Es la clase principal de este diseño ya que es el escenario el encargado de almacenar a objetos de la clase robot, ROI y cámara. Esto se debe a que es él el encargado de detectar sus posiciones y a su vez es conocedor de que elementos componen el sistema. En el programa principal cuando es necesario obtener cualquier agente o dato (la posición de un robot, el número de regiones de interés de la plataforma) es la clase “Scene” la que va a contener esa información.

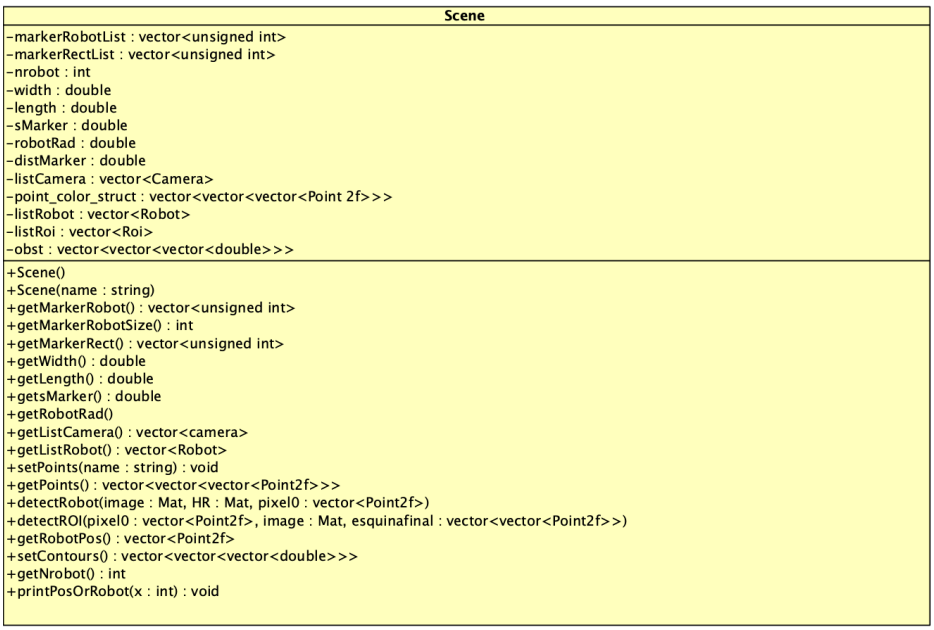


Figura 13. Diseño clase Scene UML

4.2.5 PlanningPath

Una vez definidas las clases que modelan los objetos físicos de la plataforma, es necesario introducir una serie de clases que definen un aspecto más abstracto. Entre ellas está la planificación. Esta clase se ha creado para añadir los diferentes tipos de planificación de trayectorias.

Una vez se han detectado las posiciones iniciales de los robots, las posiciones de los contornos de las regiones de interés y se conocen variables como el ancho, el largo de la plataforma y el destino de los robots, esta clase aplica la planificación conocida como Voronoi (Mahulea et al. (2020)). Esta planificación divide el área en diferentes regiones delimitadas entre ellas, calculando la distancia más cercana entre estas regiones asignando un punto a cada una. De esta forma se calculan las trayectorias que cada robot debe de seguir determinando que regiones y cuáles no puede visitar. De esta forma, cuando se acerca a alguna zona que no puede visitar (en este caso las regiones de interés) circula por el contorno de esta región de forma que nunca entrará en esta. Se almacenan como atributo en la clase y se obtiene cuando sea necesario. Un ejemplo del diagrama de Voronoi es el que aparece en la figura 14.

Una de las ventajas de este diseño es la facilidad para añadir otro tipo de planificación de trayectorias, colocándolos como métodos de la clase. Por otro lado, este diseño reduce considerablemente las líneas de código empleadas en el programa principal mejorando así la eficiencia del sistema.

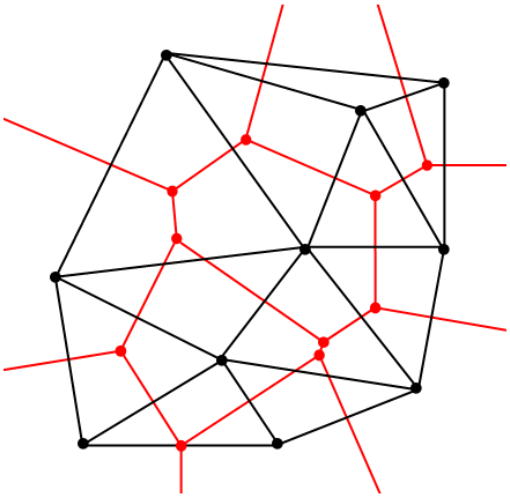


Figura 14. Diagrama de Voronoi. Recuperado de: http://fisicotronica.com/robotica-aplicacion-diagramas-voronoi/.

4.2.5 Controller

Con esta clase, se pretende encapsular las funciones que modelan el control lógico de los robots. En este caso la evitación de colisiones por parte de estos. Se encarga de comprobar el estado de los robots, si sus futuros estados son seguros y cuando tienen que pararse, reorientarse y continuar. Se han encapsulado las funciones que modelan el movimiento de un solo robot por un lado y se ha creado una función que inicia los valores y clases necesarias para la ejecución de esta lógica. Además, se diseña una función que permite reducir las líneas de código del programa principal y que modela el comportamiento colaborativo entre robots.

Como en el caso de la planificación, y del resto de clases, la ventaja es la posibilidad de introducir nuevos métodos que permitan llevar a acabo el control de los robots. Este diseño es más intuitivo y ayuda a futuros desarrollos de programación en esta plataforma.

4.3 Diagrama de flujos

Una vez descritas las clases que modelan el sistema se cree conveniente llevar a cabo un diagrama de flujos que modela la lógica utilizada para el funcionamiento de la plataforma de forma muy general. Se pretende crear una referencia visual y comentar la utilización de las clases creadas a través de este diagrama.

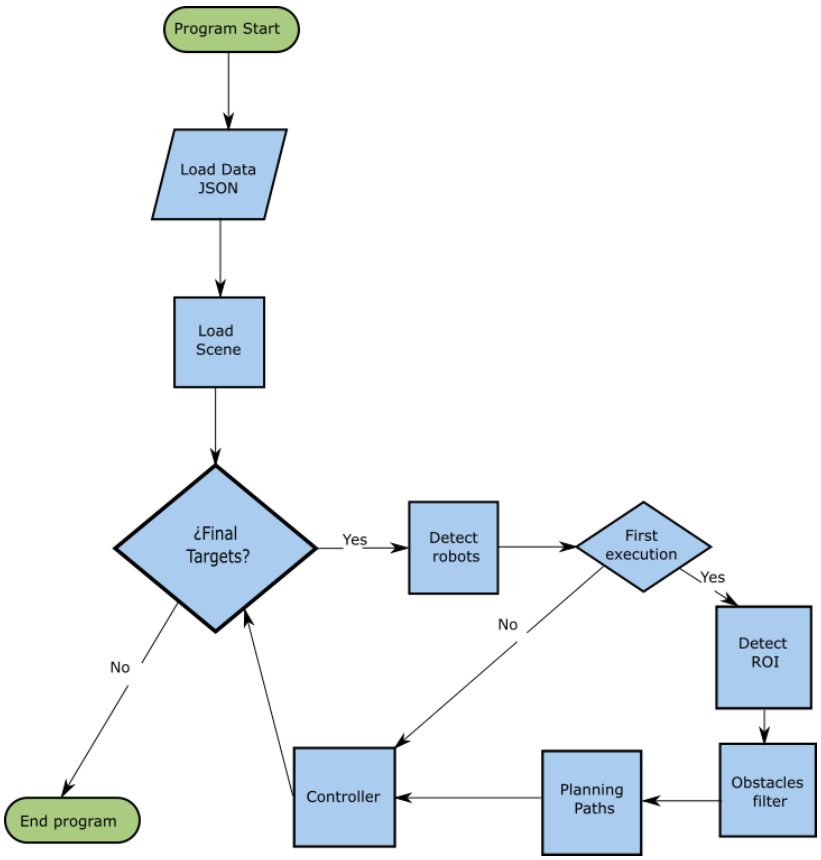


Figura 15. Diagrama de flujos.

El primer paso es la introducción de los datos de la plataforma en el archivo Json creado donde cada variable necesaria tiene asignado un nombre. A continuación, con la lógica implementada en la clase “Scene” se cargan los datos en un objeto de esta y ya están disponibles para ser utilizados por el programa.

El siguiente paso representa una decisión, e índica el inicio de la ejecución del proceso de navegación de los robots y su detección, así como de la detección de las regiones de interés. Una vez que se han alcanzado las posiciones objetivo de los robots este proceso se termina. Existen otras situaciones en las que el programa finaliza, si existe algún error en la planificación o si por ejemplo la posición inicial de los robots es la misma, pero en este caso solo se pretende mostrar una visión general.

Una vez se han detectado las posiciones de los robots se llega a otro punto de decisión, donde es necesario detectar las regiones de interés y utilizar el filtro para determinar si su tamaño es el adecuado como para considerarse regiones de interés. Posteriormente, con la posición inicial de los robots, las regiones de interés y los objetivos, se planifican las trayectorias que seguirán los robots.

Finalmente, es el controlador (utilizando el Algoritmo del Bankero) el que se encarga de evitar las colisiones entre los robots mediante un bucle donde se controla su posición y que alcancen sus puntos de destino. A su vez, se irá comunicando con ellos a través de la comunicación serie controlando su estado, posición alcanzada, etc.

4.4 Desarrollo

Otro aspecto importante del proyecto ha sido el aprendizaje de elementos relacionados con la programación como es el uso de C++ y con el lenguaje como es la compilación o los entornos de desarrollo. En este apartado se van a describir las herramientas que se han utilizado para llevar a cabo el proyecto.

- En primer lugar, para el proceso de programación desde el ordenador personal se decide utilizar **Visual Studio Code** que es un editor de código poco pesado y que permite la programación en multitud de lenguajes de programación mediante la instalación de diferentes paquetes. Una de las ventajas que tiene es que subraya los errores durante el proceso de programación y permite una detección más eficaz y rápida. La compilación se realiza desde la terminal incluida en el propio programa.

- En cuanto al **compilador** que se encarga de traducir el programa, se utiliza g++ la versión 5.4.0 para Ubuntu, ya que en el ordenador del laboratorio se utiliza como sistema operativo la distribución de Linux, Ubuntu con la versión 16.04.

- Para el tratamiento de imágenes se utilizan las **librerías de openCV** ya que se encuentra instalada la versión 3.2.0 en el ordenador del laboratorio. openCV es una librería de libre descarga que permite tratar las imágenes y que dispone de un amplio rango de clases. Se utilizan para dibujar sobre las imágenes, obtener píxeles, etc. En este proyecto, es la clase “Scene” la que utiliza la mayor parte de estas librerías. Por otro lado, como ya se ha comentado, parte de las pruebas se realizan desde el ordenador personal a través del tratamiento de un video por lo que ha sido necesaria la instalación de estas librerías con la misma versión que la utilizada en el laboratorio. A continuación, se muestra un ejemplo de la introducción de las librerías en los diferentes programas:

```
1 #include <opencv2/imgproc/imgproc.hpp>
2 #include <opencv2/core/types_c.h>
3 #include <opencv/cv.h>
4 #include <opencv2/imgcodecs.hpp>
5 #include <opencv2/highgui/highgui.hpp>
```

Código 1.

- Para la detección de los marcadores de la plataforma, es necesario utilizar un módulo adicional para las librerías llamado **ArUco** que permite trabajar con ellos. En este caso está instalado en el ordenador del laboratorio, pero también se ha trabajado con él desde el ordenador personal.

- Para la compilación del programa se ha utilizado un **archivo makefile** que es un archivo de texto y actúa como gestor de proyectos cuando el número de archivos a compilar y sus dependencias son elevadas. En este trabajo, se ha creado un archivo makefile para el ordenador personal y se ha adaptado el ya existente en el ordenador del laboratorio a las necesidades del proyecto, localizando librerías, carpetas, etc.

- Una de las novedades introducidas en este trabajo es la utilización de los **archivos Json,** que son fáciles y rápidos para utilizar. Son archivos que permiten la obtención de datos (Anexo II) de forma sencilla e intuitiva mediante el uso de librerías. En este trabajo, se ha utilizado la librería libre "jsoncpp" que ha sido instalada en el ordenador del laboratorio y con la que se puede trabajar. En el Anexo II se han introducido ejemplos que se han creado para esta plataforma. La utilización de estos archivos Json aportan generalidad al sistema ya que limitan la comunicación de usuario al relleno de unos campos. El programa toma estos datos y los utiliza para llevar a cabo sus objetivos.

- Para la **creación de las clases**, es necesario utilizar dos tipos de archivos, los .h o "headers", que disponen de unos "headers guards" que evitan las redefiniciones de las clases, y los .cpp. Ambos van a estar relacionados con la misma clase, pero los .h van a contener la definición de esta (nombre de atributos, de funciones, etc.) así como las librerías que necesitan los métodos de la clase y los .cpp van a contener el código de las operaciones de la clase. Una vez explicado esto, se aportan las siguientes líneas de código como ejemplo:

```
#include <iostream>
#include <string>
#include <vector>
#include <opencv2/core/types.hpp>
#include <fstream>
#include "../Camera/vision.h"
#include "../Robot/robot.h"
#include "../Roi/roi.h"
#include <opencv2/aruco.hpp>
#include <jsoncpp/json/json.h>
```

Código 2.

Estas líneas de código representan el encabezado del archivo "Scene.h" donde se introducen las librerías necesarias para el funcionamiento y posteriormente se define la clase. Sin embargo, el archivo .cpp dispone del siguiente encabezado:

```
#include "scene.h"
scene::scene(){
}
```

Código 3.

Únicamente se incluye el archivo .h y a continuación aparece el constructor de la clase. Esta es la forma más adecuada y segura para la creación de clases y la que se ha utilizado en este trabajo.
Como el número de archivos es elevado, se ha creado una carpeta para cada una de las clases creadas y en ella se introducen estos archivos .h y .cpp así como otros archivos necesarios donde se utilizaban clases de la versión anterior de la plataforma.

Finalmente, se dispone de un archivo con el nombre “main.cpp” que es sobre el que se realizan las pruebas introduciendo como librerías los .h y cuya compilación crea el ejecutable “plat” que permite ejecutar el programa creado.

Capítulo 5

Conclusiones y líneas futuras

Finalmente, en este capítulo se van a comentar las conclusiones que se obtienen tras realizar el proyecto, tanto técnicas como personales.

El rediseño orientado a objetos permite que sistemas complejos, donde existe mucha cantidad de código, tengan una mayor organización y se puedan programar más fácilmente. Este diseño que se ha creado basado en la utilización de clases le permite adaptarse (con alguna modificación) a otro tipo de sistemas multi-robot con características similares. Por otro lado, las clases se podrán utilizar en otros diseños a gusto del programador.

Otra ventaja que dispone este diseño es que estas clases se pueden completar con diferentes funciones y atributos en función del objetivo que se quiera conseguir. Además, gracias a su uso el código principal se queda reducido a un número de líneas inferior que antes, lo que mejora la eficiencia del programa y facilita el entendimiento de los procesos que se llevan a cabo.

Por otro lado, el sistema adquiere generalidad y la comunicación con el usuario queda reducida a la introducción de las variables necesarias en un archivo. De esta forma se reducen los posibles errores de funcionamiento y no es necesaria la interacción con este.

En cuanto a lo personal, este trabajo ha permitido la adquisición de conocimientos en programación y en diseño de software que en la actualidad son necesarios para cualquier Ingeniero en su carrera profesional. Además, con esta metodología de diseño se aprenden aspectos como: la abstracción de la realidad (paso de algo físico al desarrollo de software), la visión de posibles desarrollos futuros del proyecto y el trabajo guiado, entre otros.

Uno de los objetivos de este proyecto personalmente, era el aprendizaje de programación, así como de aspectos relacionados con la Ingeniería de Sistemas. Tras realizar el proyecto, veo que la mayor parte de estos objetivos se han cumplido y que el aprendizaje de programación comienza con un trabajo duro pero que finalmente es muy satisfactorio.

De cara al futuro, se pueden llevar a cabo diferentes proyectos o mejoras como puede ser la utilización de librerías tanto de openCV como de almacenamiento de datos

para darle un carácter más visual a esta. Por ejemplo, la representación de las trayectorias de los robots a lo largo de la plataforma.

También se pueden mejorar aspectos en cuanto a la detección de los pixeles de los marcadores, realizando medias para conseguir un mayor afinamiento de las posiciones y mejorar así el movimiento de los robots.

En cuanto a las regiones de interés, ahora se tratan como objetos estáticos, pero podrían utilizarse como dinámicos y que aparecieran y desaparecieran en función de los objetivos que se quisieran conseguir obteniendo así un escenario dinámico y modificable.

El tratamiento de imágenes con openCV aporta multitud de posibilidades de mejora de este sistema ya que la cámara es la que detecta la posición de los objetos que están en el escenario y una buena detección es necesaria. Por otro lado, se podrían implantar sensores o cámaras en los modelos de robot y hacer que fueran los robots los que detectan las regiones de interés o lo que se quiera obtener.

Además, gracias a la implementación de las clases, se podrán añadir el resto de los métodos de planificación de trayectorias y se podría realizar un estudio acerca de la eficiencia de estas planificaciones, calculándolas y haciendo que los robots las sigan y poder comprobar su comportamiento.

Capítulo 6

Bibliografía

[1] Aponte, L. E. (2010). *El UML o Lenguaje de Modelado Unificado como herramienta en el modelado de Objetos* [Artículo de blog]. Recuperado de http://programandoenjava.over-blog.es/article-el-uml-o-lenguaje-de-modelado-unificado-como-herramienta-en-el-modelado-de-objetos-53386438.html

[2] Pertusa, A. & Tomás, D., Pérez, C., Aragonés, J., Pérez, J. A. & Moreno, F. *Programación 2 [Libreo de apuntes, Universidad de Alicante].* Recuperado de https://pertusa.gitbooks.io/programacion-2/content/poo/poo.html

[3] Arkin, Ronald C. and Balch, Tucker (1998), Cooperative multiagent robotic systems, In Kortenkamp, David, Bonasso, R.P., and Murphy, R., (Eds.), Artificial Intelligence and Mobile Robots, Cambridge, MA: MIT/AAAI Press

[4] Arlow, J., & Neustadt, I. (2006). UML 2. PROGRAMACION. *Anaya Multimedia-Anaya Interactiva*.

[5] Barrio Arbex, J. & Martínez Montiel, J. M. (2016). *Localización de múltiples robots móviles mediante una cámara cenital.* (Trabajo fin de Grado). Universidad de Zaragoza, Zaragoza, España.

[6] Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Connallen, J., & Houston, K. A. (2008). Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, *33*(5), 29-29.

[7] Bóveda López, M. D. M., Gómez-Méndez, R., González-Pérez, C., Río Pose, J. D., Suárez de Lis, D., & Villanueva Santiago, E. M. (1999). *Tecnologías de la información y patrimonio cultural 1: El paradigma orientado a objetos*. Universidad de Santiago de Compostela.

[8] Coad, P., & Yourdon, E. (1991). *Object-oriented analysis*. Yourdon press.

[9] Esteban Gabriel, M. & Pacienzia, J. (2015). *Metodologías de desarrollo de software* (Tesis final de licenciatura en sistemas y computación). Facultad de Química e Ingeniería “Fray Rogelio Bacon”, Buenos Aires, Argentina.

[10] García Barreto, J. B. & Mahulea, C. & Ezpeleta, J. (2020). *Diseño e implementación de un algoritmo que evite colisiones en un sistema multi-robot utilizando el Modified Banker's Algorithm* (Trabajo fin de Máster). Universidad de Zaragoza, Zaragoza, España.

[11] Mahulea, C., Kloetzer, M., & González, R. (2020). *Path Planning of Cooperative Mobile Robots Using Discrete Event Models*. John Wiley & Sons.

[12] McFarland, D. (1994), Towards robot cooperation, In In From Animals to Animats III: Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior, pp. 440–444, Bradford/MIT Press.

[13] Microsoft. *LifeCam Studio 1080p* [imagen]. Recuperado de https://www.microsoft.com/es-xl/accessories/business/lifecam-studio-for-business?activetab=overview%3aprimaryr2

[14] O. (2019). *open-source-parsers/jsoncpp*. GitHub. https://github.com/open-source-parsers/jsoncpp

[15] Pressman, R. S., (1998). *Ingeniería de Software. Un enfoque práctico* (4ª ed.) McGrawHill

[16] Rodríguez-Canosa, G., del Cerro Giner, J., & Barrientos, A. (2014). Detection and tracking of dynamic objects by using a multirobot system: application to critical infrastructures surveillance. *Sensors*, *14*(2), 2911-2943.

[17] Roldán, J. J., Garcia-Aunon, P., Garzón, M., De León, J., Del Cerro, J., & Barrientos, A. (2016). Heterogeneous multi-robot system for mapping environmental variables of greenhouses. *Sensors*, *16*(7), 1018.

[18] Roldán, J. J., Peña-Tapia, E., Garzón-Ramos, D., de León, J., Garzón, M., del Cerro, J., & Barrientos, A. (2019). Multi-robot systems, virtual reality and ROS: developing a new generation of operator interfaces. In *Robot Operating System (ROS)* (pp. 29-64). Springer, Cham.

[19] Trebi-Ollennu, A., Nayar, H. D., Aghazarian, H., Ganino, A., Pirjanian, P., Kennedy, B., ... & Schenker, P. (2002, May). Mars's rover pair cooperatively transporting a long payload. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)* (Vol. 3, pp. 3136-3141). IEEE.

[20] Toledo Díaz, M., (2019, 26 de Mayo). *Robótica y aplicación de diagramas de Voronoi*. Extraído el 22 de jun. de 21 desde http://fisicotronica.com/robotica-aplicacion-diagramas-voronoi/.

[21] Verret, S. (2005). Current state of the art in multirobot systems. *Defence Research and Development Canada-Suffield*, *3*.

[22] Yan, Z., Jouandeau, N., & Cherif, A. A. (2013). A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, *10*(12), 399.

ANEXOS

ANEXO I

Anexo I. Biblioteca de clases

En este anexo se pretende mostrar tanto los atributos como los métodos de las clases antes comentadas, a través de la representación UML con el programa de libre descarga Visual Paradigm.

Camera

La clase Camera, se crea con el objetivo de representar las características de las posibles cámaras que puedan utilizarse en una plataforma. En este caso, solo se utiliza una como se ha comentado en la descripción de los elementos del sistema, pero se realiza con la idea de que se podrían disponer de varios tipos con diferentes características.

Los **atributos** que aparecen en la figura x se explican a continuación:

- **ncamera:** es el número de la cámara que se va a utilizar. Se introduce desde el constructor de la clase.
- **widthRes:** es un entero que almacena el número de píxeles de ancho que utiliza la visión. Como se ha comentado, la cámara utilizada tiene tres resoluciones.
- **heightRes:** se corresponde con el número de píxeles de alto.
- **FPS:** es un entero que representa los “frames per second” de la cámara. Se introduce desde el constructor.
- **Brightness:** se utiliza para la configuración de la clase VideoCapture de openCv.
- **capVideo:** es una clase de la librería openCV que permite abrir el video con los parámetros antes mencionados. Tiene que ser un atributo público para que desde el programa principal se pueda acceder a él.
- **Image:** pertenece a la clase Mat de openCV, donde se almacenan las imágenes que toma la cámara real y que se utiliza para el tratamiento de estas para detectar los obstáculos, marcadores,etc.

En cuanto a las **operaciones** de está clase:

- **Camera ():** constructor de la clase.
- **~Camera ():** destructor.
- **Camera (string, int):** constructor de la clase, que recibe el nombre del archivo Json donde se almacenan los datos y crea la instancia.

```
1  camara::camara(string nombre,int i){
2
3          ifstream datos(nombre);
4          Json::Reader reader;
5          Json::Value j;
6
7          reader.parse(datos,j);
8          ncamera = j["Camera"][i]["NUMBER"].asInt();
9          FPS = j["Camera"][i]["FPS"].asInt();
10         widthRes = j["Camera"][i]["WidthRes"].asInt();
11         heightRes = j["Camera"][i]["HeightRes"].asInt();
12
13         inicVideo();
14
15 }
```

Código 4.

Se utiliza una librería para leer el archivo Json del que se extraen las características de la cámara. Más tarde en el siguiente anexo se explica su funcionamiento.

- **getFPS():** devuelve el valor de los "frames per second" almacenados en la instancia.
- **getHeightRes():** devuelve el valor de la altura de píxeles utilizados en la cámara.
- **getWidthRes():** permite obtener el valor del ancho de píxeles.
- **getNcamera():** devuelve el número de la cámara correspondiente.
- **inicVideo():** permite condigurar la clase VideoCapture con el resto de atributos. Se utiliza en el constructor de la clase, de forma que una vez almacenados los atributos, se configura la clase capVideo y se puede utilizar directamente.

```
1 void camara::inicVideo(){
2     capturadevideo.set(CV_CAP_PROP_FRAME_WIDTH,widthRes);
3     capturadevideo.set(CV_CAP_PROP_FRAME_HEIGHT,heightRes);
4     capturadevideo.set(CV_CAP_PROP_BRIGHTNESS,brightness);
5     capturadevideo.set(CV_CAP_PROP_FPS,FPS);
6 }
```

Código 5.

- **getCap():** devuelve la clase VideoCapture para poder utilizarla en el programa principal.
- **getMat():** devuelve el atributo en el que se almacena la imagen que se toma con la librería openCV.

A continuación, se muestran de forma visual los atributos y las operaciones de la clase, mediante el lenguaje UML:

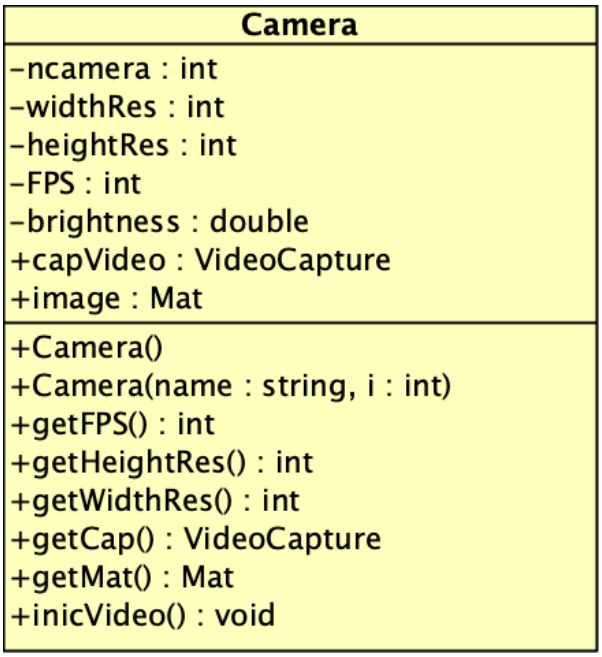


Figura 16. Diseño clase camera UML

ROBOT

Con esta clase, se pretende obtener una abstracción del robot real, tomándolos como objetos independientes. Cada robot, contará con sus funciones de comunicación y almacenará la posición y orientación. El objetivo de esta clase es crear una instancia independiente para cada robot, permitiéndose usar el número de robots que sea necesario. Con solo introducir en el Json el número de robots que se van a utilizar, se crea una instancia para cada uno, permitiendo así un sistema más organizado.

En cuanto a los **atributos:**

- **registration**: es un vector de Point2d que permite almacenar la trayectoria seguida por los robots.
- **num:** entero que almacena el número del robot correspondiente.
- **position:** del tipo Point2d, almacena la posición del robot una vez se ha detectado. Dispone de una posición x e y.
- **theta:** almacena la orientación del robot, cuando se detecta y se realizan los cálculos.
- **ID:** entero que almacena el número de marcador correspondiente a cada robot. Es necesario introducirlo desde el Json.
- **state:** string que almacena el estado de los robots.
- **name:** string que guarda un posible nombre del robot.

Se han creado una serie de operaciones para esta clase y otras, como las de comunicación se han aprovechado del código ya implementado:

- **robot ():** constructor de la clase.
- **~robot ():** destructor de la clase.
- **robot (int, int):** constructor en el que se crea la instancia introduciendo el número del robot y el ID correspondiente. También se inicia el vector registration a cero, dándole a la posición inicial un valor 0 para marcar el inicio.

```
1 robot::robot(int k, int IDs){
2     this-> num = k;
3     this-> ID = IDs;
4        Point2d inic;
5        inic.x = 0;
6        inic.y =0;
7        registration.push_back(inic);
8 }
```

Código 6.

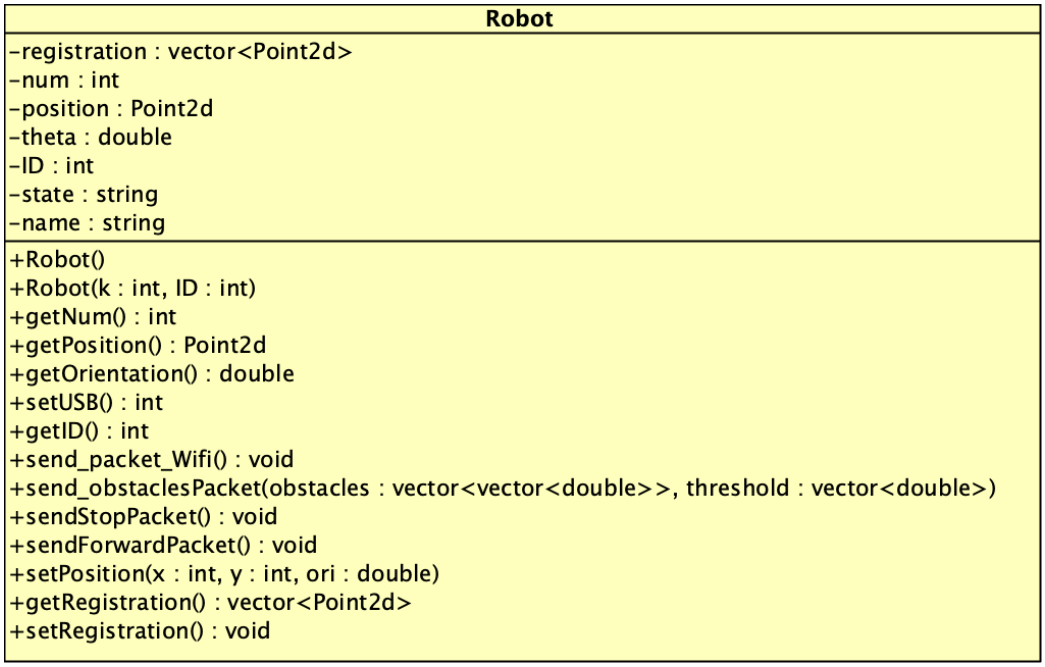
- **getNum ():** devuelve el número del robot.
- **getPosition ():** devuelve la posición actual del robot.
- **getOrientation ():** devuelve la orientación actual del robot.
- **getID ():** devuelve el valor del Id del marcador, que se ha obtenido desde el archivo Json.
- **setNum (int):** asigna al atributo num el número correspondiente, por si es necesario modificarlo o para futuros desarrollos y mejoras.
- **setID (int):** asigna al atributo ID un valor que se introduce como entrada a la función.
- **setPosition (double, double, double):** permite asignar los valores que se introducen a la función, a los atributos de la clase **(Poisition y orientation).**
- **getRegistration ():** permite obtener el vector de posiciones del robot.

A continuación, se muestran las funciones que han sido reutilizadas y las cuales se han adaptado para la creación de la clase:

- **setUSB ():** realiza la configuración del puerto USB y es necesaria cuando se realizan acciones de comunicación con los robots físicos a través de las antenas Xbee.

- **Send_packet_WIFI ():** envía la posición actual, la orientación y el número del robot a través de un string. En vez de tener estos valores como entrada a la función, ahora se cogen desde la propia clase, lo que hace el sistema más eficiente.

- **Send_stopPacket ():** comunica la parada al robot que se está moviendo en el momento que se requiere.



ROI (Region of Interest)

Hasta ahora se conocen con el nombre de obstáculos, pero reciben el nombre de región de interés ya que es más general y se podrá utilizar para futuras aplicaciones. Por ejemplo, en vez de ser lugares para evitar, como ocurre en el actual proyecto, podrán ser lugares para visitar o que aparezcan u desaparezcan creando un espacio dinámico.

Esta clase, se crea con el objetivo de darle una entidad independiente a cada región con sus parámetros más importantes. En cuanto a los atributos que contiene esta clase:

- **number:** entero que guarda el número de obstáculo correspondiente.

- **contour:** vector donde se almacenan los puntos del contorno del obstáculo correspondiente. Se ha decidido almacenarlos ya que estos puntos son una parte representativa del obstáculo y necesarios para el funcionamiento de la plataforma.

- **centroid:** almacena el centroide de la región en un tipo Point2d que tiene componente x e y. Se utiliza ya que es necesario enviárselo a los robots.

- **threshold:** área de influencia representativa del obstáculo que hay que enviar a los robots mediante las funciones de comunicación.

- **colour:** atributo del tipo "enum" que permite almacenar el color de estos, para futuras aplicaciones se puede detectar de que color es y almacenarlo.

Los métodos con los que cuenta esta clase son:

- **roi ():** constructor de la clase.

- **~roi ():** destructor.

- **roi (int, vector<vector<double>>, double, Point2d):** constructor de la clase, que crea una instancia de esta a través de los valores introducidos en la función.

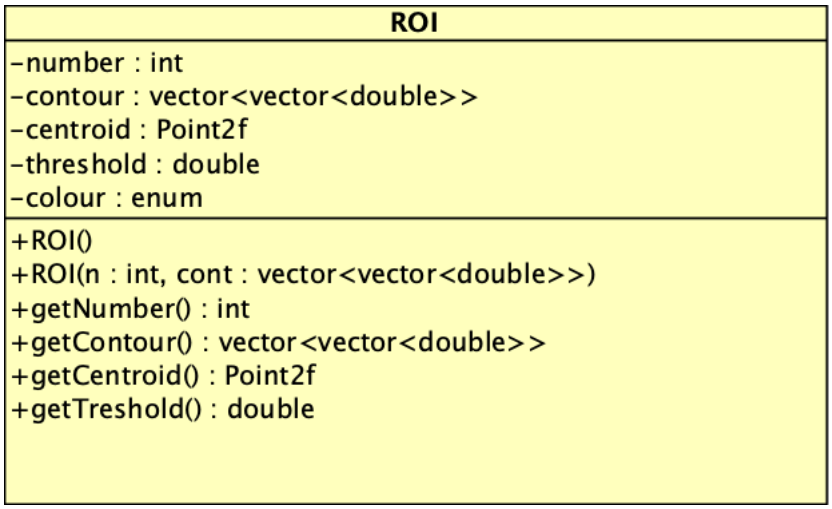
- **getNumber ():** devuelve el número de la región correspondiente.

- **getContour ():** devuelve el vector de puntos del contorno.

- **getCentroid ():** devuelve el punto del centroide de la región.

- **getThreshold ():** devuelve el valor del área de influencia de la región.

Como se comenta en el apartado de posibles desarrollos futuros, lo que se pretende con esta región de interés es dar la posibilidad a muchas mejoras en el sistema. En este caso las regiones de interés se toman como elementos estáticos que se detectan una vez y ya se almacenan, pero en fututos desarrollos se podrían tomar como elementos dinámicos que aparezcan o desaparezcan en función de la trayectoria que se quiera realizar o del objetivo del diseño.



Scene

A través de esta clase se representa el escenario en su conjunto. Es el escenario el que engloba al conjunto de cámaras que se utilizan, los robots que van a ejecutar las trayectorias, las regiones de interés que se van a tratar, etc. El escenario, se va a encargar de realizar las operaciones necesarias para obtener todos estos elementos y para ellos utilizará los recursos disponibles.

Por ejemplo, para la detección de las regiones de interés utilizará la cámara introducida por el usuario, en este caso desde un archivo json, así como para detectar y almacenar las posiciones de los robots. Es una de las partes principales del diseño ya que engloba a parte de los elementos y se encarga de realizar las operaciones necesarias en cada momento para su configuración.

En cuanto a los **atributos** con los que cuenta esta clase:

- **markerRobotList:** vector de enteros donde se almacenan los IDS de los robots introducidos desde el Json.
- **markerRectList:** vector de enteros que guarda los IDS de los marcadores de detección por medio de openCV. En este caso son 8, cuatro elevados y cuatro a ras de suelo.
- **minArea:** valor mínimo necesario para que una región sea reconocida como tal. En el caso de que sea muy pequeña no pasará el filtro y por tanto no será objeto de interés.
- **nrobot:** número de robots utilizados en el sistema que se introduce desde el json.
- **width:** valor del tipo “double” que almacena en metros la anchura de la plataforma.

- **length:** valor del tipo “double” que guarda en metros la longitud de la plataforma. Tanto la anchura como la longitud se introducen desde el json.

- **sMarker:** tamaño de los marcadores necesario para las operaciones de detección.

- **robotRad:** radio que modela el tamaño de los robots desde el punto de vista de la detección.

- **distMarker:** distancia desde el extremo de la pista hasta los marcadores elevados, necesarios para las operaciones de detección.

- **Camera:** atributo de la clase camera creada para su uso en el caso de que el escenario solo utilice una, como ocurre en este proyecto.

- **listCameras:** lista de la clase camera, donde se almacenan todas las cámaras que se pueden utilizar en el sistema. Se completa desde el constructor gracias a la lógica implementada tras obtener los datos del json.

- **point_color_struct:** vector donde se almacenan los puntos de los rectángulos de colores que rodean la plataforma, son necesarios para la detección de las regiones de interés.

- **listRobots:** vector de la clase robot creada, donde se almacenan los robots que se utilizan en el sistema.

- **listRoi:** vector de la clase creada ROI, donde se guardan una vez creadas las regiones de interés antes comentadas.

- **obst:** vector de puntos que guarda el contorno de todas las regiones de interés. Es necesario su almacenamiento para el posterior uso en el controlador.

- **cenr:** vector donde se almacenan los centroides de todas las regiones.

- **thres:** vector donde se guardan las áreas de influencias de todas las regiones de interés.

- **robotcm:** vector de puntos donde se guardan las posiciones de los robots detectadas en el instante actual en cm. Se necesita a la hora de calcular las trayectorias y en el controlador.

Como en los casos anteriores, se ha reutilizado el código empleado en la plataforma y se han creado **funciones** que se comentan a continuación:

- **scene ():** constructor de la clase.

- **~scene ():** destructor de la clase.

- **scene (string):** constructor al que se le introduce el nombre del archivo json donde se encuentran los datos y gracias a la lógica utilizada se almacenan en la instancia. Por otro lado, se almacenan también los puntos correspondientes a los rectángulos de colores que se sitúan en el contorno de la plataforma.

En el código inferior, se representa una parte de la lógica del constructor, donde gracias a los constructores de las otras clases se crea el objeto “Scene”.

```
//Cameras
    int numCam = j["NumCams"].asInt();

    for (int i= 0; i < numCam; i++){
        camara camera(nombre,i);
        listCameras.push_back(camera);
    }

    //Robots
   int numRobots = j["NumRobots"].asInt();

    for (int k = 0; k < numRobots; k++){
        vector<unsigned int> IDS;
        IDS = getMarkerRobot();
        robot robot(k+1,IDS[k]);
        listRobots.push_back(robot);
    }

    set_Points(nombre); //for saving the points that we have
in the platform (the colour rectangles)
}
```

Código 7.

- **getMarkerRobot ():** devuelve la lista de los marcadores de los robots.
- **getMarkerRobotSize ():** devuelve el tamaño de la lista de los marcadores de los robots.
- **getMarkerRect ():** devuelve la lista de los IDS de los marcadores.
- **getMarkerRectSize ():** devuelve el tamaño de la lista de marcadores elevados y a ras de suelo.
- **getWidth ():** devuelve la anchura de la plataforma en metros.
- **getLength ():** devuelve la longitud de la plataforma en metros.
- **getsMarker ():** devuelve el tamaño de los marcadores elevados.
- **getCamera ():** devuelve el atributo de la clase camera creada para utilizarla en el programa principal.
- **getRobotRad ():** devuelve el valor del radio de los robots.

- **getPoints ():** devuelve el vector de puntos de los rectángulos de colores que rodean la plataforma.

- **getListCamera ():** devuelve la lista de cámaras que se utilizan en la plataforma.

- **getListRobot ():** devuelve la lista que contiene objetos de la clase robot creada en este proyecto.

- **setContours (vector<vector<vector<double>>>):** asigna el vector introducido como parámetro al atributo correspondiente de los contornos de las regiones de interés.

- **getListObstacles ():** devuelve la lista de regiones de interés almacenadas en esta clase.

- **getNrobot ():** devuelve el número de robots utilizados en la plataforma.

- **getObstacles ():** devuelve los contornos de las regiones de interés almacenadas como atributo.

- **printPosOrRobot (int):** imprime por pantalla la posición y la orientación del número de robot que se pasa como parámetro.

- **getMinArea ():** devuelve el valor del área mínima que deben tener los obstáculos.

- **configObstacles (vector<vector<vector<double>>>):** función que filtra las regiones de interés por tamaño, calcula su centroide y su área de influencia y los almacena en los atributos del objeto. Para ello se han utilizado otras funciones que se comentarás a continuación.

```
1 void scene::configObstacles(std::vector<std::vector<
2 std::vector<double>>> obstacles){
3     vector<vector< vector<double>>> l = obstacles;
4
5     obstacles_filter(&l);
6     setContours(l);
7     cen = obstacles_centroid(&l);
8     thres = getInfluenceRadiumObstacles(cen,l);
9
  }
```

Código 8.

- **getCentroids ():** devuelve el vector con las posiciones de los centroides de las regiones de interés.

- **getThreshold ():** devuelve un vector con el área de influencia de las regiones de interés.

- **setListObstacles ():** función que permite crear la lista de las regiones de interés con su correspondiente contorno, área de influencia y centroide.

```
1  void scene::setListObstacles(){
2
3      for (int i=0; i< obst.size();i++){
4         Point2d l;
5         l.x = cen[i][0];
6         l.y = cen[i][1];
7
8          roi region(i+1,obst[i],thres[i],l);
9          listRoi.push_back(region);
10     }
11 }
```

Código 9.

- **setRobCm (vector<Point2f>, vector<Point2f>, vector <Point2f>, vector <Point3f>):** almacena la posición y orientación en el atributo robotcm, una vez se han detectado en la función detectRobot la posición de los robots. Para esta función se aprovecha el código presente en la actual plataforma.

- **getRobcm ():** permite obtener el vector que contiene tanto la posición, como la orientación de los robots en cada instante.

A continuación, se han reutilizado algunas de las funciones que se utilizaban en la plataforma actual adaptando los diseños a las necesidades actuales, en las que se han aplicado algunas modificaciones que se van a comentar:

- **set_Points (string):** esta función se encontraba antes en una clase utilizada exclusivamente para la visión, pero se decide colocarla en la clase "Scene". Se encarga de almacenar en un vector, los puntos en los que se encuentran los rectángulos de colores que rodean la plataforma. En la versión anterior, estos puntos se introducían en la misma función. Con el fin de tener un sistema lo más general posible, estos puntos se obtienen de un archivo json que se introduce como parámetro en la función. Como forma parte de la configuración de la plataforma y es una parte estática, se utiliza en el constructor de la clase.

- **detectRobot (cv::Mat, cv::Mat& , vector<Point2f>& pixel0):** en la versión anterior, esta función, estaba unida a la que comentaremos posteriormente (detectROI) mediante una lógica con booleanos que permitía detectar tanto la posición de los robots como la de las regiones de interés.
  En este caso se han separado y esta función se encarga de detectar la posición de los robots únicamente. Por otro lado, se han añadido unas líneas de código que permiten la optimización de las operaciones, de forma que cada vez que se detecta la posición de un robot, se acude a la lista de los robots y se modifican sus posiciones y orientaciones mediante la función de la clase "Robot", setPosition.

- **detectROI(vector<Point2f>, cv::Mat, vector<vector<Point2f>>& , vector<vector<Point2f>>& , vector<vector<vector<double>>>& , bool)** : esta función detecta las regiones de interés y las devuelve en varios vectores donde se encuentran los contornos de estas regiones y sus esquinas. Respecto a la versión anterior se ha eliminado la variable booleana go y se introducen como parámetro un vector con el nombre "píxel 0" que se detecta en la función de detección de los robots y que es necesario para el cálculo de distancias tal y como se había realizado la lógica en el código anterior.

- **obstacles_filter (vector<vector<vector<double>>>):** está función ha sido reutilizada, pero se ha colocado en esta clase, ya que el escenario es el que conoce la posición de los elementos estáticos y el que se encarga de realizar el filtrado de estos. Se pasa como parámetro el vector con los contornos de estos, y realiza un filtro en función del área mínima almacenada en la clase.

- **obstacles_centroid (vector<vector<vector<double>>>):** como en el caso anterior esta función es reutilizada del código anterior con algunas modificaciones y cambio del diseño, pero se sitúa aquí ya que es el escenario el encargado de realizar la detección del centro.

- **getInfluencedRadiumObstacles (vector<vector<double>>, vector<vector<vector<double>>>):** esta función ha sido reutilizada y calcula el área de influencia de los obstáculos para enviársela a los robots.

- **getMaxdistance (vector<double>, vector<vector<double>>) :** función utilizada para el cálculo del área de influencia de cada obstáculo.

Scene
-markerRobotList : vector<unsigned int> -markerRectList : vector<unsigned int> -nrobot : int -width : double -length : double -sMarker : double -robotRad : double -distMarker : double -listCamera : vector<Camera> -point_color_struct : vector<vector<vector<Point 2f>>> -listRobot : vector<Robot> -listRoi : vector<Roi> -obst : vector<vector<vector<double>>>
+Scene() +Scene(name : string) +getMarkerRobot() : vector<unsigned int> +getMarkerRobotSize() : int +getMarkerRect() : vector<unsigned int> +getWidth() : double +getLength() : double +getsMarker() : double +getRobotRad() +getListCamera() : vector<camera> +getListRobot() : vector<Robot> +setPoints(name : string) : void +getPoints() : vector<vector<vector<Point2f>>> +detectRobot(image : Mat, HR : Mat, pixel0 : vector<Point2f>) +detectROI(pixel0 : vector<Point2f>, image : Mat, esquinafinal : vector<vector<Point2f>>) +getRobotPos() : vector<Point2f> +setContours() : vector<vector<vector<double>>> +getNrobot() : int +printPosOrRobot(x : int) : void

PlanningPath

Una vez detectados los elementos físicos del sistema, como han sido los robots, las regiones de interés, etc. Se crea una clase con el fin de tratar las trayectorias que realizarán los robots para alcanzar los puntos de destino. En esta plataforma, estaban implementados varios tipos de planificación de trayectorias (por ejemplo, Dikstra) y de partición del espacio (celdas cuadradas, triangulares, etc.). Esta clase se centra solo en la planificación y para ello se ha utilizado Voronoi ya que no necesita de partición del espacio, si no que se realiza un diagrama para conocer los puntos que hay que recorrer.

Esta clase, se crea con la idea de que en el fututo se puedan ir encapsulando métodos de planificación de trayectorias para tener una amplia variedad. En este proyecto solo se trabaja con Voronoi.

En cuanto a los **atributos** utilizados en esta clase son:

- **paths_metri:** va a ser un vector en el que se almacenan los puntos de las trayectorias para todos los robots utilizados en la plataforma.

- **selected:** se crea con el fin de determinar, en el caso de que haya varios métodos de planificación, cual es el que se va a utilizar. Esto se podría introducir desde el json y realizar una función en la que se introdujeran los métodos y se seleccionara en función del número (como posible desarrollo futuro).

Para las operaciones de esta clase encontramos:

- **Voronoi (int, int, vector<vector<vector<double>>>, vector<vector<double>>, vector<vector<double>>):** se encarga del cálculo de las trayectorias para todos los robots que se utilicen. Es necesario introducirle como parámetros el alto y el ancho de la plataforma, así como la posición inicial de los robots detectados y los puntos de destino que quieren alcanzar. En este caso se ha modificado a una función tipo “void” ya que antes devolvía las trayectorias. En el proyecto actual, una vez calculadas, se almacenan en la clase.

- **nearest_vertex (Point2D, vector):** es una función que necesita la planificación de Voronoi para obtener las trayectorias.

- **check_vertex (Point2D, vector<Point2D>):** al igual que la anterior es una función que necesita Voronoi para obtener su objetivo.

- **getPaths ():** devuelve las trayectorias almacenadas en los atributos del objeto.

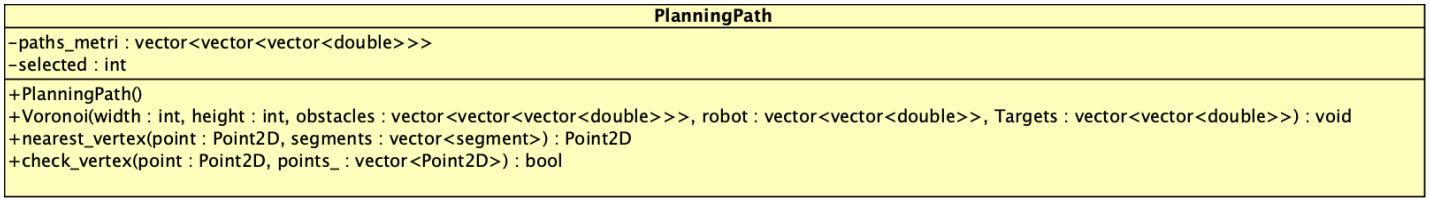

PlanningPath
-paths_metri : vector<vector<vector<double>>>
-selected : int
+PlanningPath()
+Voronoi(width : int, height : int, obstacles : vector<vector<vector<double>>>, robot : vector<vector<double>>, Targets : vector<vector<double>>) : void
+nearest_vertex(point : Point2D, segments : vector) : Point2D
+check_vertex(point : Point2D, points_ : vector<Point2D>) : bool

Figura 17. Diseño clase PlanningPath UML.

Controller

Esta clase se crea con el fin de obtener un controlador para el sistema multirobot. García et al. (2020) realizó un diseño para evitar colisión mediante el Modified Banker's Algorithm. Lo realizó orientado a objetos y en la actualidad la plataforma funciona con este controlador. Lo que se pretende con esta clase es mantener este controlador encapsulando las funciones necesarias.

Se van a presentar las funciones que se han creado, con el fin de encapsular al resto y mantener la lógica actual:

- **setupController (Syst, int, vector<vector<vector<double>>>, cv::Mat, Paths, vector<vector<vector<double>>>, int, int)** : esta función pretende inicializar los parámetros necesarios para seguir la lógica utilizada para el correcto funcionamiento del algoritmo de bloqueo. De esta forma permitirá encapsular las funciones utilizadas en el programa principal hasta ahora.

- **funcOneRobot (Syst, vector<nextState*>) :** en esta función se engloba la lógica correspondiente al funcionamiento con un solo robot. Se introducen como parámetros dos clases de las librerías disponibles.

- **funcCol (Syst, vector<nextState*>, vector<vector<double>, vector<char>):** esta función encapsula las funciones necesarias para el funcionamiento colaborativo de os robots. A continuación, se muestra un ejemplo de código:

```
void controller::funcCol(Syst sistema,std::vector<nextState*>
list_FutureStates,std::vector<std::vector<double> >
robotcm,std::vector<char> ReceivedData){

    getNextStateByBanker(sistema,list_FutureStates);

control_RobotStateBA(sistema,list_FutureStates,robotcm,ReceivedData);
    checkStatus(sistema,list_FutureStates);
    sendSubPaths(sistema,list_FutureStates);
    sendPosition(robotcm,sistema,list_FutureStates);

}
```

Código 10

El resto de las funciones que engloba esta clase son las que se necesitan para ejecutar la lógica que esta explicada en García et al. (2020)

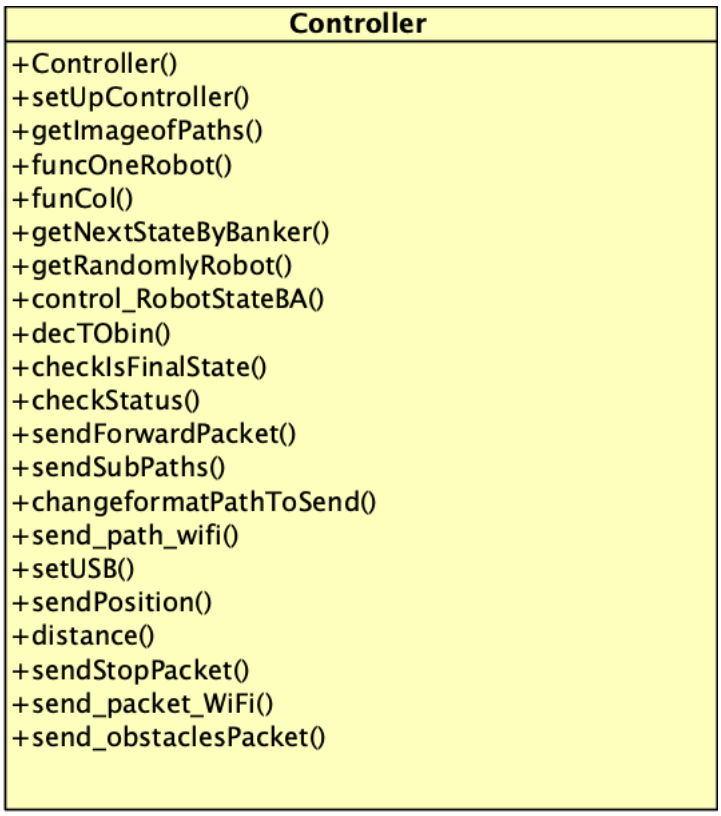


Figura 18. Diseño clase Controller UML.

ANEXO II

Anexo II. Archivos Json

Los archivos Json permiten el intercambio de datos de forma mucho más sencilla y rápida que el resto de los archivos, como por ejemplo el txt. Surgió como complemento del lenguaje de programación JavaScript y en la actualidad casi todos los lenguajes pueden leer los datos que incluye.

Este lenguaje dispone de dos tipos de estructuras como bien se explica en:

1. **Objetos o registros:** donde se representan los pares nombre-valor.

2. **Listas de valores:** que se corresponden con los vectores utilizados en C++, por ejemplo.

Un ejemplo se muestra en las siguientes líneas de código:

```
{
    "NumCams":1,
    "NumRobots":1,
    "Camera":[
      {
        "NUMBER":0,
        "FPS": 60,
        "WidthRes": 1920,
        "HeightRes": 1080
      }

    ]

}
```

Código 11.

En el código superior se tiene un objeto donde hay pares nombre – valor como es el caso de “NumCams”: 1 y listas como ocurre con los calores y los nombres introducidos en el nombre “Camera”. Este texto mostrado es parte del archivo en el que se introducen los datos para el funcionamiento de la plataforma.

Aplicación al proyecto

La idea de utilizar un archivo donde se introducen los datos surge desde el planteamiento del proyecto. Se pretende hacer un diseño general y fácilmente modificable, donde pueden variar los datos de una plataforma a otra. Se utilizan archivos Json, por que son ligeros y su lectura es sencilla utilizando librerías.

En el funcionamiento actual, el programa interacciona con el usuario pidiéndole que le introduzca el número de robots o las trayectorias, por ejemplo. Con este diseño únicamente es necesario modificar el archivo Json donde se introducen los datos en la casilla correspondiente y la lógica implementada, se encarga de usarlos y almacenarlos en cada momento. Por otro lado, en este archivo también se introducen datos necesarios para el funcionamiento de la plataforma. Por ejemplo, con el fin de hacer un diseño general, se introducen desde el archivo de configuración los puntos en los que

se encuentran las esquinas de los rectángulos de colores que rodean la plataforma. De esta forma, si en un futuro se modifican sus posiciones solo habrá que introducir el valor en el archivo de configuración y no introducirse en el código.

Por eso podemos decir que esta implementación aporta los siguientes **beneficios:**

- **Rapidez** de modificación de datos.
- **Facilidad** de lectura de datos.
- **Posibilidad de utilización** en otros sistemas.
- **Comprensión de las variables** principales del programa.

Como se ha comentado, los archivos Json surgieron con el lenguaje JavaScript y en la actualidad se utilizan para lectura en la mayor parte de lenguajes. Para poder utilizar el archivo Json de la plataforma, ha sido necesario encontrar una librería disponible para la versión de Linux correspondiente. En este caso se ha escogido la librería "jsoncpp" que es de libre acceso y que se puede encontrar en O. (2019) donde se explica el funcionamiento de esta.

Existen multitud de librerías que se pueden utilizar, pero se ha escogido esta por compatibilidad y facilidad. A continuación, se presentan algunos ejemplos de uso utilizados en este proyecto:

1. **Constructor de la clase "Camera":**

```
1  camera::camera(string nombre,int i){
2
3          ifstream datos(nombre);
4          Json::Reader reader;
5          Json::Value j;
6
7          reader.parse(datos,j);
8          ncamera = j["Camera"][i]["NUMBER"].asInt();
9          FPS = j["Camera"][i]["FPS"].asInt();
10         widthRes = j["Camera"][i]["WidthRes"].asInt();
11         heightRes = j["Camera"][i]["HeightRes"].asInt();
12
13         inicVideo();
14
15 }
```

Código 12.

2. Parte del constructor de la clase “Scene”:

```
scene::scene(string nombre){
    ifstream datos(nombre);
    Json::Reader reader;
    Json::Value j;

    reader.parse(datos,j);

    //fstream datos(nombre);
   // json j;

    //Attributes
    //datos >> j;
    nrobot = j["NumRobots"].asInt();
    minArea = j["Scene"][0]["MinimumArea"].asDouble();
    sMarker = j["Scene"][0]["MarkerSize"].asDouble();
    width = j["Scene"][0]["Width"].asDouble();
    length = j["Scene"][0]["Length"].asDouble();
    robotRad = j["Scene"][0]["RobotRad"].asDouble();
    distMarker = j["Scene"][0]["Dist"].asDouble();
```

Código 13.

3. Obtención y almacenamiento de una lista en constructor de la clase “Scene”:

```
const Json::Value& books = j["Scene"][0]["RobotIDS"];
    for (Json::ValueConstIterator it = books.begin(); it !=
books.end(); ++it)
    {
        const Json::Value& book = *it;
        markerRobotList.push_back(book.asInt());
    }
```

Código 14.