Facultad de Informática

Universidad de Las Palmas de Gran Canaria

Sistema Integrado de Planificación y Control de Misiones con Vehículos Submarinos Autónomos

Eliezer Ramírez Cabrera

Las Palmas de Gran Canaria, MI FECHA

Proyecto fin de carrera de la Facultad de Informática de la Universidad de Las Palmas de Gran Canaria presentado por el alumno:

Eliezer Ramírez Cabrera

Título del Proyecto : Software integrado de planificación y control de misiones de Vehículos Submarinos Autónomos

Tutor : Jorge Cabrera Gámez

Cotutor: José Daniel Hernández Sosa

Cotutor: Antonio Falcón Martel

A mis padres, Fefi y Lorenzo

y hermanos, Pilar y Aitor

Agradecimientos

Quisiera mostrar mi más sincero agradecimiento a los profesores que me han tutorado durante la realización de este proyecto. Éste ha sido un proyecto complejo y en el que muchas veces corría el riesgo de perderme en detalles. Sin su orientación difícilmente hubiera podido acabar este proyecto.

Agradecerles también su constancia y esfuerzo, y el interés que han mostrado por que este proyecto se desarrollara con éxito, sobre todo durante las últimas fases del mismo.

También quisiera hacer una mención especial a Rodrigo Heredero Robayna y Enrique Fernández Perdomo. Emprendimos entre los tres estas líneas de investigación sobre la exploración oceanográfica con ilusión, y compartimos muchos momentos de reuniones, trabajo duro y esparcimiento. Sus contribuciones han sido muy valiosas para el desarrollo de este proyecto.

A mis padres y hermanos, les agradezco que siempre hayan estado ahí, apoyándome y confiando en mis posibilidades, sobre todo en los momentos más críticos de mi proyecto.

Muchas gracias a todos.

Índice

Parte I. Introducción.

Capítulo 1. Introducción

El estado actual del tema de la exploración subacuática está dominado por el uso de vehículos de operación remota (ROV, del inglés Remotely Operated Vehicle). Estos vehículos están teleoperados manualmente desde un barco en la superficie y están conectados mediante un cable donde la información circula en doble sentido hacia el barco y desde el barco.

La idea es que un operador desde el barco sea capaz de comandar a un ROV recibiendo la información que el submarino ha recopilado con cámaras y sensores para realizar una misión en concreto, normalmente de exploración y toma de muestras.

Por otro lado la aparición de los vehículos submarinos autónomos (AUV, del inglés Autonomous Underwater Vehicle) ha supuesto una pequeña revolución en el mundo de la exploración subacuática. La utilización de estos vehículos tiene una serie de ventajas frente a los ROVs. La más evidente es que un AUV no necesita estar conectado, por lo que tiene más libertad de movimiento.

Con los AUVs el usuario solo tendrá que decidir qué tareas va a realizar el submarino a alto nivel. No tiene que teleoperar directamente el submarino, tal y como ocurre con los ROVs, manejando sus actuadores a bajo nivel para que el submarino se mueva de la forma que quiere el usuario.

Además, La duración de la operación se ve reducida con la utilización de AUVs, ya que éstos, además de ser generalmente más rápidos, que los ROVs, no se ven influenciados por las inclemencias del tiempo. El problema con los ROVs es que se ven afectados por el estado del mar y el movimiento del barco. La dependencia con el tiempo en la superficie se reduce, mientras que con el ROV, si el tiempo no es favorable, se deben detener las labores de exploración hasta que el tiempo lo permita.

Otra de las ventajas que se le achaca a un AUV frente a un ROV es que la calidad de los datos que toma es mejor, debido a que un AUV no está conectado a un cable el cual le podría transferir movimientos de la nave o las vibraciones del propio cable, deteriorando el muestreo del vehículo.

Aún así, y como es lógico, no todo son ventajas con los AUV. Una de las desventajas es que el usuario pierde control sobre el vehículo. Los algoritmos de detección de formas son muy costosos y un AUV difícilmente podrá realizar tareas de seguimiento basadas éstas en imágenes. Por ello, hay tareas que un AUV no podrá realizar, como por ejemplo la búsqueda y recuperación de tesoros en barcos hundidos, debido a la complejidad de la búsqueda y la recogida de éstos. Ésta sería una tarea para un ROV. Sin embargo, para la exploración del mar en busca del barco, sería bastante más aconsejable un AUV que rastreara el mar.

1.1 Objetivos

El objetivo de este proyecto es el de estudiar la forma de desarrollar un software integrado de planificación y control, con el que seamos capaces de especificar y monitorizar las misiones que se quieran llevar a cabo con estos vehículos submarinos autónomos.

Para ello habrá que empezar desde el principio. Se habrá de diseñar la forma de especificar al AUV todos los detalles de la misión que el usuario pretende realizar: ruta que debe seguir, medidas que debe tomarlas veces que se debe comunicar con un centro de control etc.

Una vez tenido esto, se debe diseñar un entorno para facilitar al usuario en la medida de lo posible la definición de esta misión. El entorno para desarrollar la misión debe ser un entorno gráfico, que permita definir la misión en función de los requerimientos de usuario, la instrumentación disponible y la autonomía del vehículo.

1.2 Estructura del documento

Esta memoria se ha dividido en 6 partes, cada una de ellas conteniendo uno o dos capítulos. La estructura del documento es la siguiente:

- **Parte I**: Es una introducción al proyecto. Esta compuesto por 2 capítulos:
    - Capítulo 1: Objetivos → Se explican brevemente cuáles son los objetivos del proyecto.
    - Capítulo 2: Estado del arte → Se comenta cuál es la situación actual de los vehículos submarinos no tripulados.
- **Parte II**: Esta parte del proyecto se centrará en el análisis de las misiones y de la aplicación. Está compuesta por dos capítulos:
    - Capítulo 3: Estudio de las misiones → Se estudia todo lo que sea necesario especificar en una misión para un AUV, y en qué planes se puede dividir la información.
    - Capítulo 4: Análisis del sistema → Será un análisis de la aplicación de planificación y control de misiones, y los distintos algoritmos necesarios para implementarla.
- **Parte III**: Diseño del prototipo. Se compondrá de un capítulo:
    - Capítulo 5: Diseño del sistema → Se hablará de todos los detalles del diseño que se ha utilizado para el prototipo.
- **Parte IV**: Implementación. Constará de un capítulo.

- Capítulo 6: Implementación → Se explicará hasta donde se ha podido llegar en la implementación del prototipo, y algunos detalles de la implementación.

-

Capítulo 2. Estado del arte

2.1 ¿Qué es un AUV?

AUV son las iniciales de Autonomous Underwater Vehicle. Se trata de vehículos autónomos submarinos capaces de desarrollar distintos tipos de misiones subacuáticas sin necesidad, durante el desarrollo de la misión, de la intervención de una persona. La idea es tener un submarino, para el que el usuario pueda establecer la misión a realizar estando en tierra, transmitírsela al AUV y que éste sea capaz de realizarlo de forma autónoma una vez se haya introducido en el agua.

Para ello el submarino tendrá que tener cierta autonomía, ser capaz de detectar y evitar peligros y de establecer protocolos de actuación frente a situaciones en las que puede peligrar la misión o su propia integridad. Por ejemplo, si se le agota la batería, el AUV debería ser capaz de regresar hasta la posición final de la misión antes de que se le haya agotado por completo la batería, o, en su defecto, mandar un mensaje de alerta y conservar la suficiente batería como para emitir mensajes cada cierto tiempo dando a conocer su posición.

2.2 Posible equipamiento sensorial de un AUV

A continuación se muestra una lista de los sensores que podrían equipar un AUV, junto con las unidades de medida que se utilizan para tomar los muestreos con esos sensores:

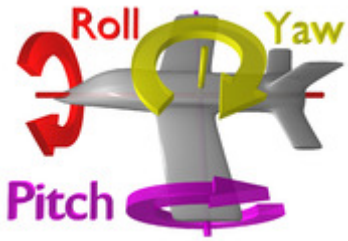
- Sonar: Es un equipo que utiliza la técnica SONAR(Sound Navigation and Ranging). Esta técnica emplea la propagación del sonido bajo el agua, principalmente, para navegar, comunicarse o detectar otros buques.

- Sensores para medir la pose: La pose es la que determina la orientación del vehículo con respecto a su centro de masas. Se definen en inglés como:

    - Roll: Es la rotación que se produce en torno al eje X. El eje X aumenta de forma positiva a través del vector velocidad del submarino.

    - Pitch: Es la rotación que se produce en torno al eje Y. El eje Y tiene su origen en el centro de masa y aumenta hacia la parte derecha del submarino (sistema levógiro).

    - Yaw: Es la rotación que se produce en torno al eje Z.



Existen una serie de sensores que ayudarán al submarino a conocer su pose.

- Magnetómetro (Brújula): Un magnetómetro es un instrumento que se utilizará para medir la intensidad del campo magnético.

- Inclinómetro: Aparato para medir el grado de inclinación del vehículo.

- GPS: Instrumento que se utiliza para conocer la posición en la que se encuentra el submarino. Éste instrumento sólo será útil si el submarino se encuentra en superficie.

- Unidad de Navegación Inercial (INS): El INS constituye una parte importante en de los sistemas de navegación automática en aviones, naves espaciales, misiles teledirigidos, cohetes barcos y submarinos. La información que proporciona el INS a sus ordenadores, les permite registrar las variaciones en la trayectoria y corregir el rumbo.

➢ CTD: Conductivity, Temperature and Depth. El CTD es un tipo de sensor capaz de medir tanto la conductividad, como la salinidad, temperatura, profundidad, densidad del agua, presión. Algunas de estas medidas las toma directamente, y otras, indirectamente, a través de las directas. Los instrumentos CTD miden directamente tres importantes parámetros - conductividad, temperatura y presión. A través de la conductividad se obtiene la salinidad, de la presión se obtiene la profundidad, o de la conductividad, temperatura y presión se puede obtener la densidad.

  - Sensor de presión/profundidad: Servirá para conocer la profundidad en la que se encuentre el AUV.

  - Sensor de temperatura: Aunque con el CTD conseguimos medir la temperatura del agua, sería necesario otro sensor de temperatura en el interior del submarino para controlar la temperatura del hardware.

  - Salinidad: Medir la salinidad del mar es muy útil para los meteorólogos por ejemplo, ya que la salinidad establece la densidad del agua, lo que es un elemento que influye de manera muy importante sobre las corrientes marinas.

- Conductividad: Con la ayuda de la conductividad podremos conocer la salinidad y densidad del agua.

- Sensor de velocidad (DVL, Doppler Velocity Logger): Instrumento que mide la velocidad utilizando el efecto Doppler.

2.3 Estudio de las herramientas existentes.

En esta sección describiremos varios sistemas de planificación de misiones para submarinos autónomos existentes actualmente, y cómo solucionan éstos los problemas derivados de una planificación que puede ser muy compleja y estar abierta a distintas posibilidades y problemas durante el desarrollo de la misión por parte del AUV.

2.3.1 Framework CORAL.

La información ha sido obtenida de **[RADO03]** (ver bibliografía), **[ROMEO]**, **[MARIUS_2]**.

En el Instituto Superior Técnico de Lisboa, Portugal han desarrollado un Framework para el diseño de sistemas de control de misiones. En este centro utilizan el submarino MARIUS, también desarrollado por ellos, para llevar a cabo misiones submarinas.

Este Framework va a permitir desarrollar misiones para cualquier tipo de AUV, no sólo para el MARIUS, por lo que habrá que especificar antes de poder utilizar este submarino en su herramienta de definición de misiones, todas las primitivas que este vehículo es capaz de realizar.

Para definir las primitivas se ha desarrollado un lenguaje de eventos discretos denominado CORAL, al igual que la plataforma en la que es ejecutado. Tanto el lenguaje CORAL como la plataforma en la que se ejecuta se han desarrollado con el objetivo de ser utilizado para el control de vehículos autónomos en tiempo real.

Antes de seguir hay que explicar el concepto de "**Primitiva de vehículo**". Este concepto lo utilizan para definir las acciones que puede llevar a cabo el submarino. Se denomina primitiva de vehículo a una acción atómica y claramente identificable que puede ser ejecutada por el AUV. Las misiones que se diseñen en este entorno tendrán como bloque básico de construcción a las primitivas de los vehículos que participen en la misión.

Por ello, es necesario, antes de definir una misión, conocer las capacidades técnicas del AUV. El usuario deberá definir las primitivas del vehículo antes de definir la misión, para poder diseñar el comportamiento que tendrá éste a lo largo de la misión con ayuda de estas primitivas.

2.3.1.1 Definición de primitivas de vehículos.

En la siguiente imagen se puede ver un esquema que explica cómo funciona el Framework CORAL, para que el usuario pueda definir las primitivas del submarino con él. El usuario dispone de dos opciones para la definición de las distintas primitivas del AUV.

La primera opción es la de un editor de primitivas del vehículo, en el que el usuario utilizaría las redes de Petri para definirlas. Una vez definidas las primitivas del vehículo mediante redes de Petri, el programa se encargará de traducir esas redes de Petri a lenguaje CORAL.

La segunda opción, y la más compleja, sería la de utilizar un editor de texto, en el que el usuario implementaría las primitivas directamente en lenguaje CORAL. El lenguaje CORAL es un lenguaje asíncrono y conducido por eventos, y está clasificado como "Left Recursive" [LR1].

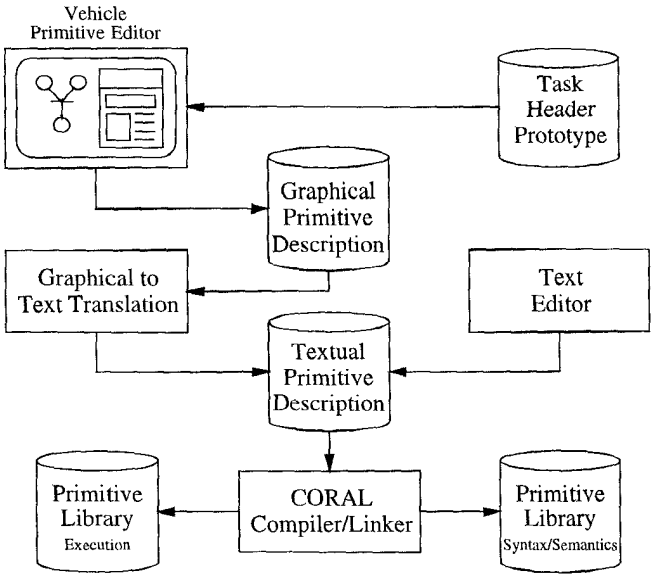


Ilustración 1: Entorno de programación de primitivas de vehículos

Tanto si utiliza una forma u otra de definir las primitivas, al final tendremos un fichero con las primitivas del vehículo definidas en lenguaje CORAL. Las primitivas como una descripción textual en el lenguaje CORAL, se pasan por el compilador CORAL, que producirá dos librerías de primitivas del vehículo. Una será un archivo que contenga las descripciones sintácticas y semánticas de todas las primitivas del vehículo, mientras que la otra va a contener un conjunto de datos necesario para su ejecución.

En principio, una misión simple puede ser embebida en una red de Petri de alto nivel que pueda implementar las estructuras de procedimiento de la misión necesarias. Sin embargo, el análisis de una misión simple usando esta metodología se pude convertir en una tarea muy compleja y tediosa debido a que obtendríamos una red de Petri difícil de manejar. Por ello se necesita definir un entorno específico para el diseño e implementación de una misión.

En la siguiente sección se habla de una misión simple definida mediante redes de Petri.

2.3.1.2 Definiendo una misión mediante una red de Petri.

En esta sección se va a mostrar un ejemplo de implementación de una misión para un AUV en una red de Petri. La misión va a consistir en realizar vueltas con forma cuadrada. Para ello el submarino tendrá que mantener una profundidad y velocidad constante, y la dirección se modificará cada 40 segundos en -90º.

Uno de los procedimientos que se van a utilizar para implementar la misión es el HorizonPath. El procedimiento HorizonPath: comienza por establecer un timer para generar un timeout después de que el tiempo de ejecución requerido haya terminado. Para ejecutar la maniobra, se ayudará de la ejecución de tres primitivas de vehículo en paralelo:

- KeepSpeed con una velocidad “u”.
- KeepDepth con una profundidad “z”.
- KeepHeading con una dirección “φ”.

Tanto la velocidad “u”, como la profundidad “z” y la dirección “φ” son parámetros de la función. Cuando se produce el timeout se termina la ejecución del procedimiento HorizonPath, saliendo de las tres primitivas del vehículo.

En la Ilustración 2 se muestra la implementación de este procedimiento usando el entorno de programación CORAL.

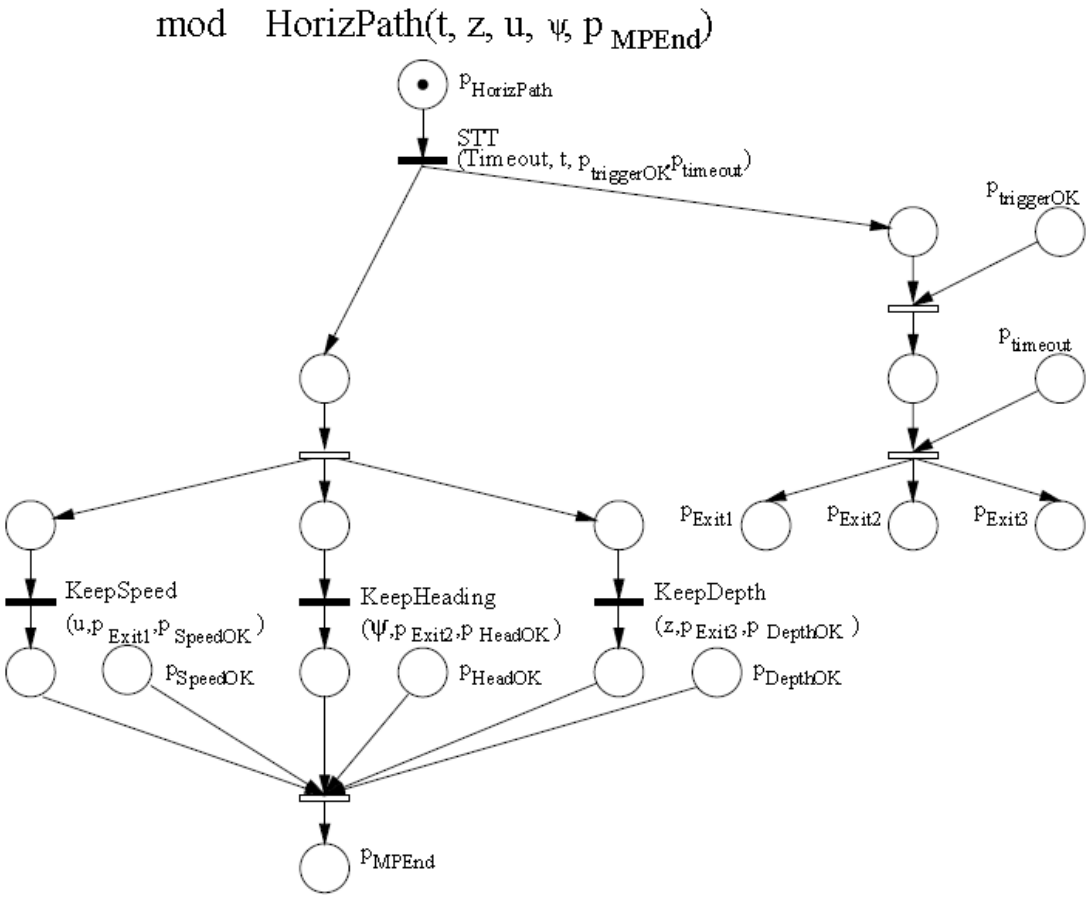


Ilustración 2: Red de Petri. HorizPath

La misión va a constar de cuatro fases diferentes:

- **Fase 1**: Todas las tareas del sistema son inicializadas llamando a la primitiva de vehículo "*Init*".

- **Fase 2**: En esta fase se llama al procedimiento HorizonPath con un periodo de 20 segundos, una velocidad de 2m/s, una profundidad de 1,35 metros y una dirección de 0º. Al final de esta fase el submarino es dirigido al norte y preparado para comenzar la maniobra cuadrada.

- **Fase 3**: Se llama repetidamente al procedimiento HorizonPath con las direcciones 0º, -90º, -180º y -270º, mientras se mantienen los demás parámetros con los mismos valores que en la fase 2. La duración requerida para cada ejecución del procedimiento HorizonPath es de t = 40 s. En paralelo, la primitiva del vehículo "*ControlDatalog*" es ejecutada para comenzar a logear datos del loop de control para su posterior análisis.

- **Fase 4**: Finalmente, en la fase 4 el vehículo pasa a modo manual. La recogida de datos del bucle de control finaliza y la misión termina normalmente si no se han producido errores.

En la Ilustración 3 se puede ver la implementación de esta misión en el entorno CORAL, con sus cuatro fases y todos los pasos necesarios para que el submarino realice esta misión.

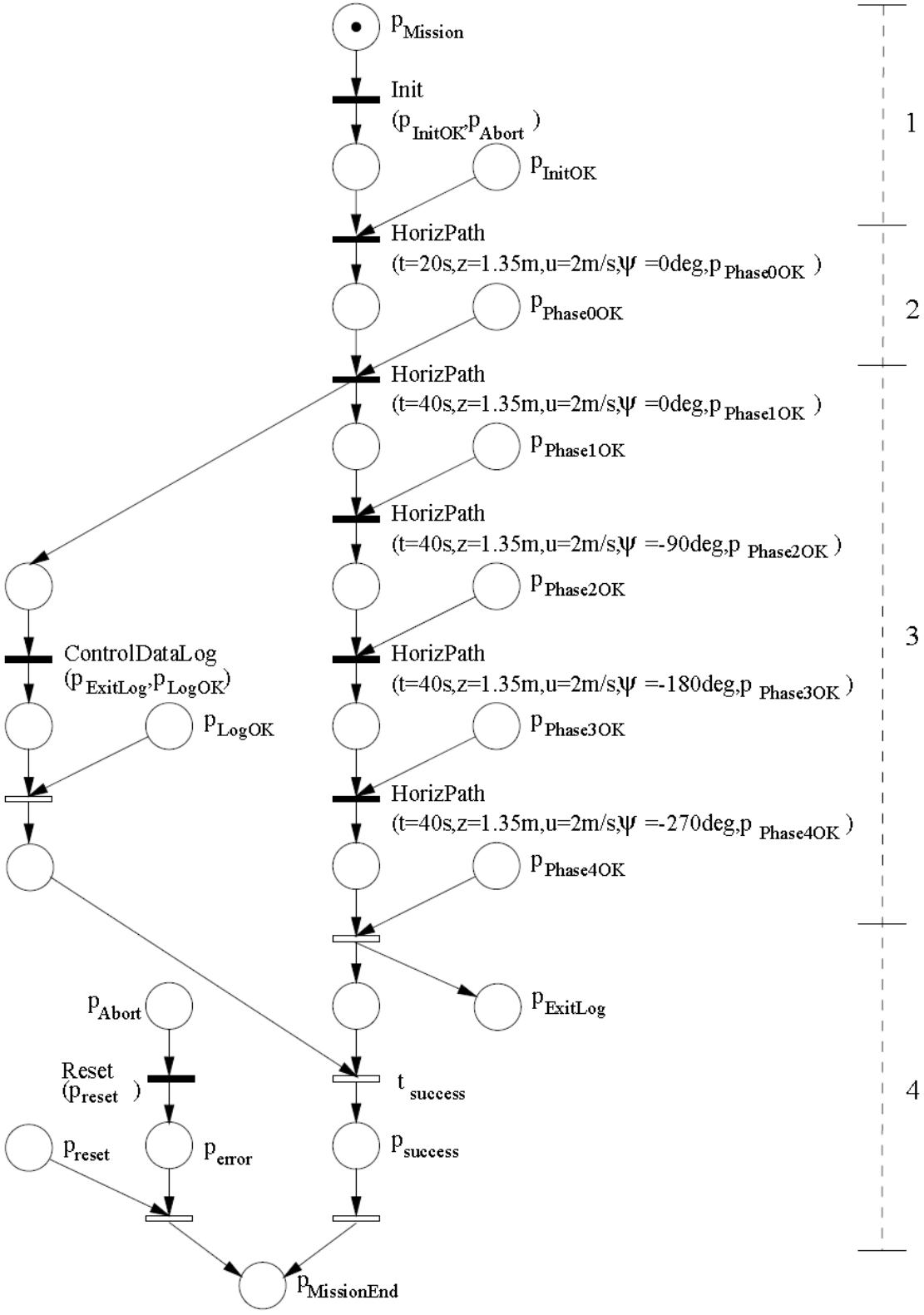


Ilustración 3: Mision en CORAL

El problema es que hacer misiones medianamente complejas en este entorno puede ser una tarea muy laboriosa y compleja. Por ello se ha desarrollado el editor de misiones "Mission Designer". Ésta es una herramienta intuitiva, de fácil comprensión y utilización, pensada para que los usuarios finales no tengan que poseer grandes conocimientos técnicos para poder especificar una misión para un AUV.

El entorno de edición de misiones "Mission Designer" se ha desarrollado en C++ y lo que han pretendido conseguir con este entorno es:

- Presentar un ambiente de edición de misión gráfico y amigable, generando automáticamente código CORAL para ejecutarlo en los motores de CORAL instalado a bordo de los vehículos.
- Ser independiente del vehículo a utilizar.
- Permitir controlar, sea el vehículo que sea, todas las herramientas que éste tenga asociadas.
- Permitir la definición de misiones multiAUV.
- Aprovechar toda la tecnología de edición gráfica para minimizar el tiempo necesario de edición y facilitar la posterior comprensión.
- Ejecutar algoritmos de optimización de trayectorias teniendo en cuenta los obstáculos modelados en el escenario de la misión, con el fin de minimizar la distancia total recorrida por el vehículo.

Para diseñar una misión la aplicación "Mission Designer" necesitará:

- **Información de los obstáculos**: Se definirán en el mapa los obstáculos que deberá evitar el vehículo robótico submarino mediante polígonos. Hay dos formas de insertar la información de un obstáculo, mediante un fichero .god (gps obstacle data), en el cual se especificarán la latitud y longitud de cada uno de los puntos del polígono que forma el obstáculo, o dibujando el obstáculo en el mapa en el editor de misiones.
- **Referencias geográficas**: Las referencias geográficas incluyen la información del escenario en el que se lleva a cabo la misión, como pueden ser mapas, almacenados como imágenes, información topográfica, un fichero de información del escenario, en el que se especificará las coordenadas (latitud y longitud) de los mapas antes mencionados, etc.
- **Modelado del vehículo**: Información que incluirá la descripción, las primitivas y el identificador visual del vehículo.

En la Ilustración 4 se puede ver una captura del programa "Mission Designer":

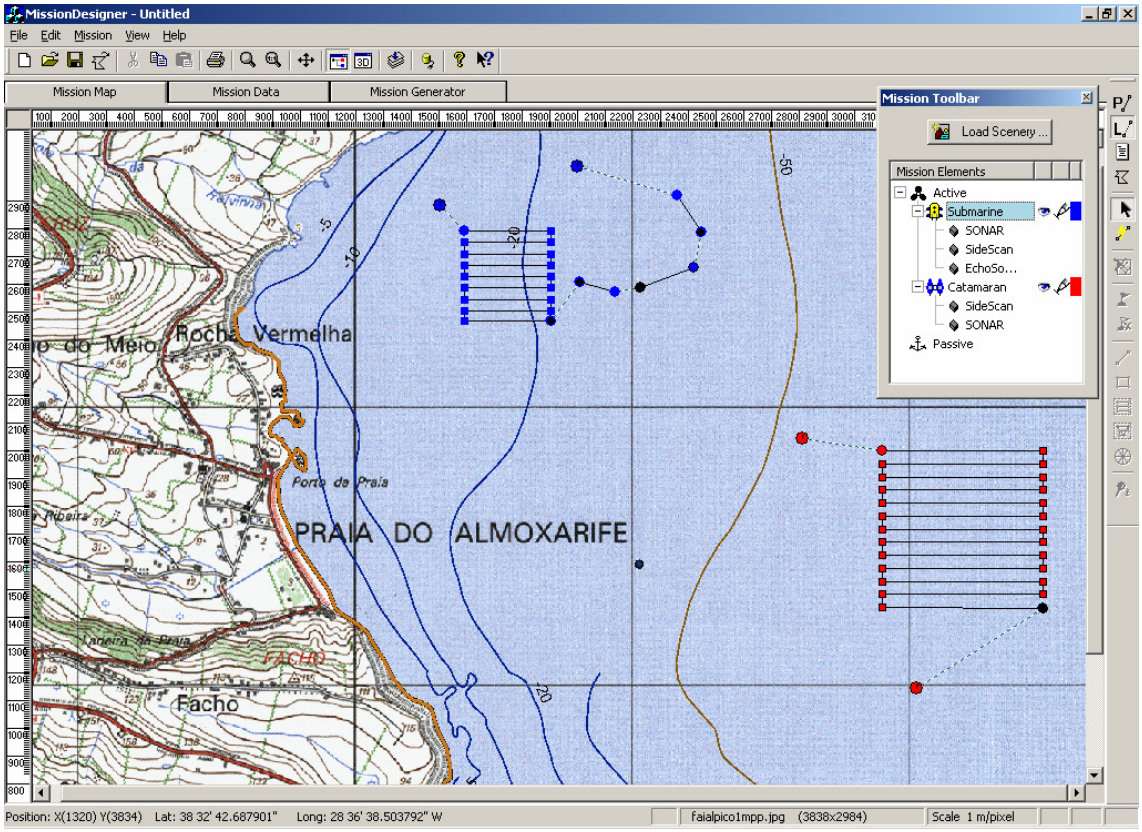
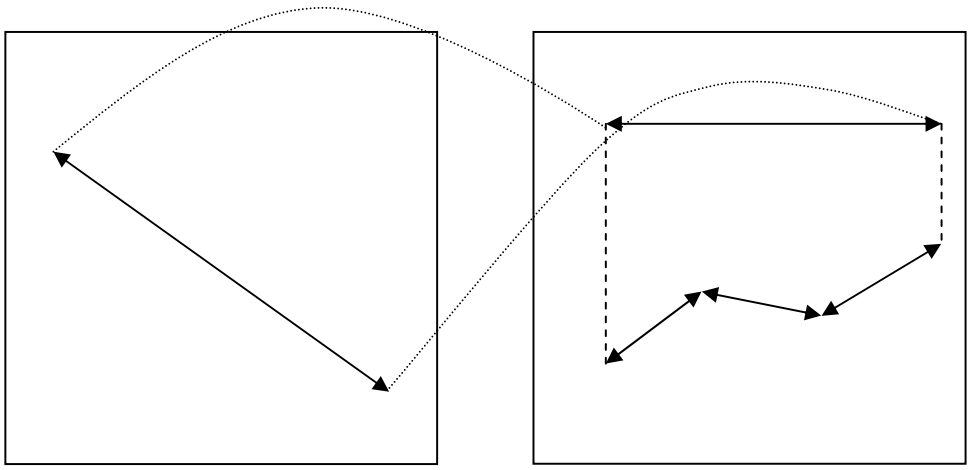


Ilustración 4: Entorno de definición de misión

Para definir una misión, primero definiremos su proyección horizontal, como se ve en laIlustración 4. Sobre el mapa definiremos una serie de segmentos que tendrá que recorrer obligatoriamente el vehículo. Si queremos controlar el submarino en un entorno tridimensional, incluyendo la profundidad a la que se va a mover el submarino en cada punto de su recorrido, podremos especificar para cada segmento xy una serie de "secciones z", es decir, se dividirá a su vez cada segmento, en un plano tridimensional, en varias secciones para la que se definirá una profundidad al inicio de la sección y otra profundidad al final de esa sección. El vehículo se moverá durante el segmento xy subiendo y bajando por esos puntos en los que hemos establecido la profundidad.



Plano horizontal (xy) Profundidad (z)

Ilustración 5: Secciones XY y Z

Una vez hayamos definido los caminos obligatorios por los que va a pasar el submarino, tendremos que definir los caminos auxiliares por los que va a pasar el AUV para unir los distintos caminos obligatorios. Tenemos dos opciones para hacerlo:

- Definir los caminos auxiliares que unirán los caminos obligatorios manualmente.
- El software "Mission Designer" se puede encargar de definir los caminos auxiliares de forma que minimice la distancia final recorrida por el vehículo y evite los obstáculos definidos sobre el mapa.

El usuario, a la hora de definir las primitivas del vehículo en el fichero "AuvLib.def", ha tenido que diferenciar entre las primitivas del módulo XY, las primitivas del módulo Z y "otros módulos":

- **Módulo XY**: Lista de primitivas a invocar para ejecutar el movimiento del vehículo en el plano xy. No es posible invocar simultáneamente más de una primitiva de la lista.
- **Módulo Z**: Lista de primitivas a invocar para ejecutar el movimiento del vehículo en el eje Z. Para estas primitivas tampoco es posible invocar a dos de ellas simultáneamente, y sólo se pueden ejecutar cuando la misión se ha editado en 3D.
- **Otros módulos**: Incluye una lista de primitivas que no están asociadas a ningún tipo de movimiento del vehículo. Estas primitivas pueden indicar funciones auxiliares de los vehículos.

Para modelar el comportamiento del vehículo durante la misión, una vez definida la ruta de navegación, tendremos una serie de secciones xy y éstas, a su vez, divididas en secciones z. A cada una de estas secciones se podrán asignar las primitivas que puede ejecutar el vehículo. Esto definirá las acciones que tendrá que realizar el vehículo en cada punto de la ruta de navegación.

A las secciones XY se asignarán las primitivas definidas para el módulo XY y para “Otros módulos”. A las secciones Z se le asignarán las primitivas definidas para el módulo Z y para “Otros módulos”.

2.3.2 MIMOSA

La información ha sido obtenida de **[MIMOSA]**.

MIMOSA es un gestor de misiones de vehículos submarinos autónomos desarrollado por el IFREMER (Instituto Francés de Investigación para la Explotación del Mar).

Este software dispone de las siguientes funcionalidades:

- Preparación de la misión, planificación y validación.
- Supervisión de la misión en tiempo real.
- Una vez realizada la misión se podrá hacer un análisis y un playback de la misma.
- Gestión de datos de la misión y la navegación.

Cumple el ciclo de vida completo:

- Preparación de la misión
- Programación y validación de la inmersión.
- Supervisión del vehículo en tiempo real.
- Análisis y reporte de la misión una vez realizada.
- Gestión del vehículo y su carga útil.

Este software incluye todas las herramientas necesarias para que el usuario final pueda planificar misiones y llevarlas a cabo.

Una de las principales características de este software es la ayuda que ofrece al usuario para poder planificar de una forma sencilla y rápida una inmersión sin necesidad de que el usuario tenga un profundo conocimiento del vehículo.

MIMOSA almacena todos los datos de la inmersión y es capaz de exportarlos todos o parte de ellos para la explotación científica de estos datos.

En la Ilustración 6 se ve la planificación de una misión sobre un mapa con este software.

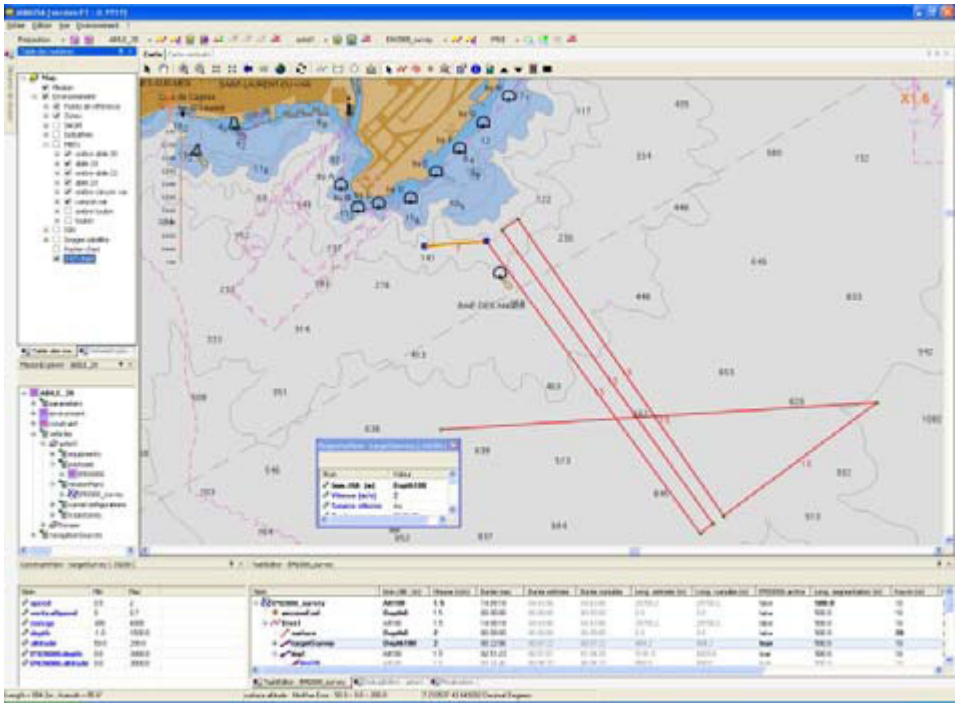


Ilustración 6: Planificando una misión

Una vez realizada la misión, el usuario puede ver, tal y como se puede ver en la Ilustración 7, tanto la ruta planificada como la ruta que ha seguido el vehículo submarino.

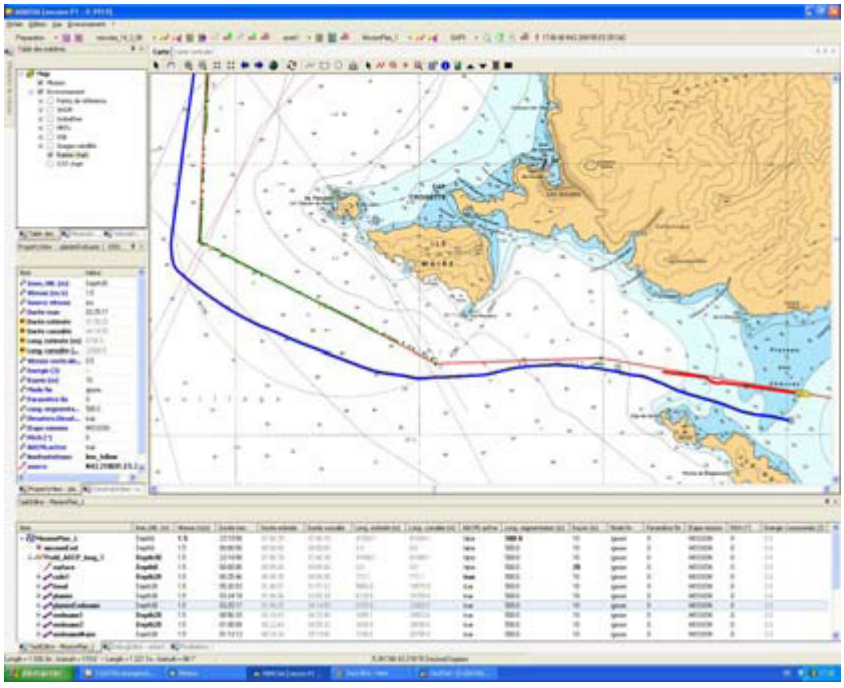


Ilustración 7: Monitorizando una función

Además, el usuario podrá estudiar todos los datos que ha recabado el submarino durante la misión mediante este software, con la ayuda de gráficas que mostrarían los datos de una forma más amena.

Capítulo 3. Metodología y planificación

En este capítulo se hablará sobre la metodología software empleada para el desarrollo del prototipo. Además se mencionarán los recursos necesarios para realizar el proyecto actual.

También se tratará la planificación del trabajo y temporización del mismo.

3.1 Metodología

La metodología empleada para desarrollar este software es la de **aproximación incremental**. Se ha seleccionado este tipo de metodología ya que inicialmente no estaban del todo definidos los requisitos del software, al tener este proyecto una componente bastante importante de investigación.

Cada ciclo constó de un análisis, un diseño, una implementación y pruebas. No se ha llegado a la fase de mantenimiento ya que no se ha llegado a tener una versión final del software.

En el primer ciclo de vida, se analizó y se diseñó unos planes XML para una misión que constaba de tres planes. Estos planes abarcaban la navegación, las comunicaciones y las mediciones que tendría que realizar el submarino durante la misión. Además, durante este ciclo de vida se desarrollaron prototipos en Matlab para soportar este diseño de los planes, siendo finalmente estos prototipos en Matlab desechados para implementar la aplicación completamente en Java.

Durante los siguientes ciclos se fueron agregando planes a la misión (Supervisión y Almacenamiento), y modificando éstos hasta llegar a los especificados en este documento.

En cuanto al prototipo, se fue modificando en cada iteración, a la vez que se modificaba la especificación de los ficheros XML de los planes para adaptarse a los cambios que suponían éstos y para añadir o mejorar funcionalidades con respecto a la versión anterior del prototipo.

3.2 Recursos necesarios

Hardware:

- Portátil: SAMSUNG R70. Intel® Core™2 Duo Processor T7300 (2.0GHz). Memoria RAM, 2.00 GB.

Sistema Operativo: Windows Vista™ Home Premium.

Aplicaciones en general:

- Acrobat Reader: Se ha utilizado para la lectura de documentación en formato PDF.
- Internet Explorer (varias versiones): Navegador Web.
- Mozilla Firefox (varias versiones): Navegador Web
- Google Maps: Servidor de aplicaciones de mapas web.

Aplicaciones para desarrollo:

- Netbeans IDE 6.1: Entorno gráfico de desarrollo de aplicaciones Java.
- JDK 1.5 y JDK 1.6: software que provee de herramientas de desarrollo para la creación de programas en Java.
- Notepad++: Software editor de textos que permite el resaltado de diversos lenguajes de programación, incluyendo el XML.

Aplicaciones para la documentación:

- Microsoft Office Word 2003 y Microsoft Office Word 2007. Software de procesamiento de textos.
- Microsoft Office Visio 2007: Este software se ha utilizado para la realización de diagramas UML.
- Microsoft Office PowerPoint 2003: Software utilizado para la realización de presentaciones.

Librerías:

- JDICplus 0.2.2: Librería usada para incrustar el mapa de Google en al aplicación.
- JDOM 1.1.1: Librería usada para la lectura/escritura de ficheros XML.

3.3 Planificación de trabajo

En este apartado se estimará de forma aproximada el porcentaje de tiempo dedicado en este proyecto a las fases de análisis, diseño, implementación y pruebas.

3.3.1 Análisis

En este proyecto pionero se ha dedicado mucho esfuerzo, durante su arranque, a la recopilación de información sobre AUVs, oceanografía, y software existente para la planificación y control.

Se estima que aproximadamente el 25% del tiempo trabajado en el proyecto ha sido dedicado al análisis, en el que podemos distinguir varias etapas:

- Análisis del estado del arte: Se realizó búsqueda de información con el fin de adentrarnos en el tema de los vehículos submarinos autónomos y todo el mundo que lo rodea (oceanografía, equipamiento, diferentes soluciones a la hora especificar misiones…).

- Análisis de las misiones: Una vez acabado el punto anterior, se realizó un análisis de las especificaciones de este tipo de vehículos, y los requisitos a la hora de definir misiones abarcables por ellos. Este punto fue desarrollado junto con otros dos proyectos paralelos en los que se desarrollaba un sistema de control para un AUV y un simulador de este tipo de robots.

- Análisis de la aplicación: Se analiza cuál puede ser la mejor forma de desarrollar un prototipo en el que se permita al usuario planificar y monitorizar misiones de este tipo.

3.3.2 Diseño

El diseño de esta aplicación incluirá las siguientes etapas:

- Diseño de las misiones: Se diseñan, mediante ficheros XML, los diferentes planes y ficheros que van a formar parte la misión del AUV. Como ya se ah mencionado, este punto se realizó con la colaboración de otros dos proyectos paralelos sobre este tipo de vehículos.

- Diseño de las comunicaciones: Se diseñan una serie de formatos de comunicación entre el AUV y la aplicación.

- Diseño de la aplicación: Es el diseño de la aplicación propiamente dicho, con sus patrones de diseño y los diagrama UML.

3.3.3 Implementación

Este apartado incluye la implementación del prototipo que se ha analizado y diseñado previamente. Será la etapa que más tiempo consumirá, debido a la gran cantidad de clases que se han implementado para que funciones el prototipo.

Se estima que se ha llevado el 35% del tiempo total del proyecto.

3.3.4 Pruebas

Las pruebas consistían en realizar pequeñas misiones y comprobar que se hubieran almacenado correctamente en ficheros XML y que se validen contra un AUV.

Esta etapa es la más corta. Se estima que ha consumido un 5% del tiempo total del proyecto.

3.3.5 Documentación

La documentación ha abarcado aproximadamente el 5% del total de tiempo dedicado al proyecto. A la hora de hacer la documentación se ha podido aprovechar documentos de análisis y diseño previos, por lo que muchos apartado de este documento ya estaban resueltos en las etapas anteriores.

3.4 Temporización

Este proyecto fue asignado el 22 de Enero del 2007. Se prolongó durante 3 años, hasta Enero del 2010 debido a que no se pudo dedicar el 100% del tiempo durante ese periodo al mismo, por diversos motivos.

Hubo varias etapas principalmente durante el desarrollo de este proyecto:

- Una etapa inicial en la que se simultaneaba el proyecto fin de carrera con las últimas asignaturas que quedaban pendientes en la carrera.

- Una segunda etapa de uno o dos meses durante los cuales se pudo dedicar el 100% del tiempo al proyecto, justo después de terminar las asignaturas de la carrera y antes de introducirme en el mundo laboral.

- Una tercera etapa en la que se simultaneaba un trabajo de 40 horas semanales con el proyecto.

- Una cuarta etapa, en la que se aceptó colaborar en un proyecto de boyas oceanográficas en Telecomunicaciones, que aportaría conocimientos sobre sensores y geolocalización. Este proyecto se realizaba a la vez que se dedicaban 40 horas semanales al trabajo. Esta etapa duraría unos 6 meses.

- Una quinta etapa, que se prolongaría hasta el momento actual, en la que se vuelve a simultanear trabajo y proyecto.

Debido a que se prolongó el proyecto durante todo ese tiempo, es complicado establecer una temporización precisa, pero estimo que las horas dedicadas íntegramente al proyecto han sido alrededor de 800.

A su vez, se han realizado trabajos paralelos al proyecto durante estos años, que ha podido ser aprovechados para realizar este documento, como puede ser la beca ya mencionada y algunos trabajos para determinadas asignaturas de la carrera.

Parte II. Análisis.

Capítulo 4. Estudio de las misiones.

Tanto el análisis de las misiones como su diseño, se ha desarrollado con la colaboración de otros dos proyectos paralelos a éste, en los que se trataban de definir un sistema de control para un AUV y un simulador de AUVs.

Con esta colaboración entre proyectos se pretende obtener unas misiones que sean capaces de ser interpretadas uy ejecutadas tanto por un AUV real como un simulador de este tipo de vehículos.

4.1 Análisis de la definición de la misión.

Nuestra idea de AUV ha ido cambiando y evolucionando a lo largo del proyecto. En esta memoria solo mostraremos el estado actual de nuestro análisis y diseño, después de muchas modificaciones y reestructuraciones.

Creemos que es importante dividir el comportamiento del submarino en cinco apartados distintos: Navegación, comunicación, medición, almacenamiento y supervisión de la misión.

Por ello hemos decidido que una misión conste de cinco planes distintos. Cada uno de estos planes se encargará de uno de estos cinco aspectos. A continuación se realiza un análisis de los requisitos que deben cumplir cada uno de estos planes.

4.1.1 Navegación

La navegación, como veremos, va a ser un plan completamente distinto a los demás. En este plan vamos a definir la ruta que seguirá el submarino, de una forma más o menos flexible. A este plan harán referencia los demás planes, ya veremos cuando estemos viendo los demás por qué y cómo lo harán.

Podríamos pensar en varios tipos de planes de navegación que podemos implementar. A continuación enumeraremos los tipos de misiones que estudiaremos para a continuación entrar en más detalles sobre cada uno de ellos.

- Seguimiento de ruta
- Exploración de área
- Seguimiento en base a medidas
  - Gradiente
  - Rango
- Toma de muestras estática
  - Básica
  - Deriva

4.1.2 Seguimiento de Ruta

El usuario necesitará especificar al AUV que tiene que ir desde una ubicación hasta otra, pasando por una serie de puntos intermedios que vamos a denominar waypoints o puntos de paso. Llamaremos ruta a la secuencia de puntos de paso o waypoints. La misión que debería especificar el usuario es algo parecido a lo siguiente.

4.1.2.1 Básica

La misión básica sería un seguimiento de un AUV por una ruta previamente especificada por el usuario. Se le puede suministrar al vehículo determinados datos como pueden ser la batimetría de la zona por la que va a navegar.

Para establecer la ruta por la que debe pasar el submarino, el usuario va a establecer una serie de **waypoints**. Estos puntos serán muy importantes a la hora de establecer medidas, o puntos de comunicación del submarino, para conocer el estado de la misión etc.

```
// Datos de la misión
Ruta ruta1 = [    waypoint1 = {0.54, -0.32, 10},
                  waypoint2 = {0.52, -0.30, 10},
                  waypoint2 = {0.50, -0.28, 10},
                  ]

// Datos de soporte (opcional)
Batimetria batimetria

// Comandos
Seguir(ruta1)
```

Esta misión se puede complicar si además le añadimos una serie de restricciones temporales a la misión. Estas restricciones se pueden especificar estableciendo el tiempo que va a tardar el submarino en llegar desde un waypoint a otro o estableciendo el tiempo que tardará en recorrer por completo la ruta.

Además, quizás sería conveniente establecer el "error", es decir, el submarino va a tardar en recorrer la ruta 1 hora, con un 10% de error que indicará que el submarino puede tardar 6 minutos más o menos de la hora en recorrer la ruta.

Un ejemplo de esta misión se muestra a continuación.

```
// Datos de la misión
Ruta ruta1

// Comandos
Seguir(ruta1)

// Restricciones globales, variables de la misión
TiempoTotal.tiempo = 1 hora
TiempoTotal.holgura = 10%

// Lista de restricciones locales, entre waypoints
//    · Los waypoints deben ser crecientes, evitando solapes
//    · No se permitiría indicar del waypoint 1 al último,
//       pues eso se indicaría con:
```

```
//    TiempoTotal
//    · En el caso de que el usuario definiera un área o zona de
waypoint inicial y final del rango
RangoTiempo.waypoints = [1 4]
RangoTiempo.tiempo = 10 minutos
RangoTiempo.holgura = 1%
```

4.1.3 Exploración de área

Otro tipo de misión que puede ser útil al usuario sería la de exploración de un determinado área. El usuario quiere que el submarino explore un área muestreando un tipo de medidas para después analizarlas.

El usuario deberá especificar el área a explorar y la profundidad o el rango de profundidades entre las que desea que la explore. Además, deberá poder especificar el modo en el que el vehículo recorrerá el área.

4.1.3.1 Básica

```
// Datos de la misión
Area area

// Comandos
Recorrer(area, modo, resolucion)    // Recorre el área en un modo
                                    // (espiral, zig-zag, …), con una
                                    // resolución (o frecuencia)

// Restricciones, como la profundidad
Profundidad.limites = [0 100] metros      // Límite mínimo y máximo
                                          // de profundidad
Profundidad.incremento = 20 metros        // Incremento para volver
                                          // a recorrer área a nueva
                                          // profundidad
```

A este tipo de misión también se le pueden añadir restricciones temporales. En este caso, la restricción abarcaría a todo el área.

```
// Datos de la misión
Area area

// Comandos
Recorrer(area, modo, resolucion, profundidad)

// Restricciones globales, variables de la misión
TiempoTotal.tiempo = 1 hora
TiempoTotal.holgura = 10%
```

4.1.3.2 Seguimiento en base a Medidas (Gradiente, Rango, etc.)

En el apartado anterior veíamos como el submarino puede seguir una determinada ruta delimitada por el usuario mediante un área. Además hemos querido incluir otros tipos de misiones cuyos recorridos no dependerán de localizaciones geográficas, sino de otras medidas que pueda tomar el vehículo subacuático.

4.1.3.2.1 Gradiente

Otro de los tipos de misiones que podría ser importante sería una misión en la que el vehículo subacuático siguiera un gradiente. El AUV haría lecturas de una determinada medida buscando zonas en las que esta medida aumente o disminuya. Trataría de seguir un rumbo en el que consiguiera que esa medida fuera aumentando o disminuyendo.

Además, hay que establecer unos límites para que por encima de unos rangos o por debajo de ellos, el submarino no siguiera gradientes. También se delimitara la zona en la que el submarino se puede mover siguiendo el gradiente mediante un área.

```
// Seguimiento
SeguimientoGradiente.medida = temperatura
SeguimientoGradiente.direccion = Creciente     // Creciente o
                                               //decreciente
SeguimientoGradiente.limites = [0 100] ºC // Límites inferior y
                                          //superior, para que no
                                          //siga gradiente por ellos

// Área límite opcional; fuera de ella no se sigue el gradiente
Area areaLimite   // Área límite (poligonal)
Profundidad.limite = [0 100] metros // Límite mínimo y máximo
                                    //de profundidad
```

4.1.3.2.2 Rango

Otro tipo posible de misión es en la que el usuario decide que el submarino se va a mover entre rangos de una medida. Para ello el usuario puede hacer un plan parecido al siguiente:

```
SeguimientoRango.medida = temperatura
SeguimientoRango.rango = [10 30] ºC //  Rango  mínimo  y  máximo  de
temperatura

// Área límite opcional; fuera de ella no se sigue el gradiente
Area areaLimite   // Área límite (poligonal)
Profundidad.limite = [0 100] metros // Límite mínimo y máximo
                                    // de profundidad
```

Tal y como ocurría con el seguimiento de gradiente, el usuario indicará que el submarino deberá moverse en un área limitada.

Tanto en el seguimiento de gradiente como en el rango de medidas, el usuario podrá incluir unas restricciones temporales dentro de las cuales el submarino deberá cumplir la misión.

```
… (GRADIENTE O RANGO) …

// Restricciones globales, variables de la misión
TiempoTotal.tiempo = 1 hora
TiempoTotal.holgura = 10%
```

4.1.3.3 Toma de muestras estática (Configuración tipo Boya)

4.1.3.3.1 Básica

Si el usuario está interesado en tomar datos como la haría una boya, dejándose arrastrar, podemos utilizar el submarino tal y como si fuera una boya.

Al submarino se le especificará la posición en la que debe mantenerse estático y tomar muestras. En este tipo de misión se puede indicar un área, en lugar de una posición en concreto, un área dentro de la que el submarino se tendrá que mantener. En este caso esta misión se asemejará bastante a la de dejar el submarino a la deriva. Lo único que la diferenciaría sería el tamaño del área, la cual es considerablemente más pequeña que para el tipo de misión para la que el submarino se deja a la deriva.

```
MuetrasEstaticas.tipo = basica.
// Posición a mantener
Posicion.latitud = 10º N
Posicion.longitud = 3º E
Posicion.profundidad = 0 mts

Area areaLimite   // Área límite (poligonal)
```

4.1.3.3.2 Deriva

En el apartado anterior el submarino con ayuda de sus motores intentaba quedarse en la posición indicada, tal y como si fuera una boya anclada. En este caso, dejaremos al submarino a la deriva. Puede ser interesante para estimar corrientes en superficie, por ejemplo.

Podemos establecer el punto inicial tal y como indicábamos antes, con una latitud, longitud y profundidad en la que queremos que el submarino inicie su deriva.

Se podría establecer un área, para que el submarino se deje arrastrar pero solo hasta el límite del área, lugar donde activará los impulsores y volverá a entrar en el área delimitada por el usuario.

```
MuestrasEstaticas.tipo = Deriva.

Area areaLimite   // Área límite (poligonal)
```

Además, también, se podrán añadir restricciones temporales.

```
MuestrasEstaticas.tipo = basica.

// Posición a mantener
Posicion.latitud = 10º N
Posicion.longitud = 3º E
Posicion.profundidad = 0 mts

// Restricciones globales, variables de la misión
TiempoTotal.tiempo = 1 hora
TiempoTotal.holgura = 10%
```

4.1.4 Establecer zonas prohibidas

Debe ser posible, por la seguridad del AUV, establecer zonas por las que se le prohibirá la navegación. El submarino tendrá su propia forma de actuar frente a determinadas situaciones, por ejemplo, que detecte que se va a chocar y esquive el obstáculo. Aún así, sería interesante poder establecer desde tierra las zonas que le estarán prohibidas recorrer durante la misión debido a su peligrosidad.

```
ZonaProhibida = zona
```

4.2 ¿Qué otros planes necesitaríamos?

Una vez tenemos el plan de navegación definido, debemos pensar en plantear qué es lo que se quiere obtener con esta misión. Es decir, necesitamos establecer qué medidas se tomarán durante la misión, y cuándo deben tomarse éstas. A este plan lo llamaremos **Plan de Medidas.**

Sin embargo, también podríamos plantearnos el definir una serie de "puntos de control", en los que el submarino se comunicará con el monitor de la misión para informarle del estado tanto de la misión como del propio AUV, además de poder emitir todos los datos recopilados hasta el momento y permitir al usuario que esté monitorizando la misión emitir nuevos comandos al AUV y cambiar la misión si fuera necesario. Éste será el **Plan de Comunicaciones**.

Hemos querido dar al usuario la posibilidad también de decidir qué medidas se van a almacenar. El usuario no sólo podrá definir qué medidas se toman, sino también qué medidas se van a almacenar. Es decir, que si el usuario pide que se mida un dato, pero no pide que se almacene, éste dato no se almacenará. Con ello conseguiremos que los dos planes sean ortogonales, es decir, que no dependa uno de otro. Se pueden tomar medidas sin necesidad de almacenarlas, y se pueden almacenar medidas que no se hayan requerido que se midan.

Una posible utilidad de este plan sería, por ejemplo, como un log de la temperatura interna del submarino. Ya que ésta temperatura se está midiendo constantemente sin que el usuario tenga que especificar que se mida, se irá almacenando

la temperatura interna y se podrá utilizar posteriormente como log. Éste será el **Plan de Almacenamiento**.

La última de las posibilidades que tiene el usuario, es la de elaborar un plan de supervisión de la misión. Éste plan será capaz de modificar parámetros de los demás planes, he incluso de modificar planes enteros (y con ello la misión). Está pensado como un sustituto básico del usuario en cuanto a monitorización de la misión se refiere, ya que será capaz de realizar las mismas acciones que el monitor de la misión. La idea es que en situaciones en las que el monitor de la misión no puede contactar con el submarino (bajo el agua, por ejemplo), el plan de supervisión sea capaz de modificar el comportamiento del submarino frente a determinadas situaciones, como pueden ser algún fallo en algún sistema del submarino o el agotamiento de la batería. Éste será el **Plan de Supervisión**.

4.2.1 Ideas generales para estos planes.

Todos estos planes mencionados en el apartado anterior (es decir, exceptuando el plan de navegación) tienen un problema en común, y es cuándo lanzar una acción. Para entender esta sección primero explicaré de qué partes se pueden componer todos los planes excepto el de navegación.

Podremos pensar que estos cuatro planes tienen algo en común: se componen de una lista de acciones a realizar que se tienen que producir en un determinado momento. El usuario necesitará especificar cuándo necesita que se lancen cada una de las acciones definidas.

Entonces tendremos dos elementos comunes en cada uno de los planes:

- **Acción**: Acción será lo que deberá hacer el submarino en un momento determinado de la misión. Por ejemplo, si estamos hablando del plan de comunicación, una acción será enviar un tipo de datos al planificador. En cambio, si estamos hablando del plan de medición, la acción consistirá en medir un dato.
- **Disparador**: El disparador nos dirá cuándo debemos ejecutar una acción. Cada acción o grupo de acciones deberán tener su grupo de disparadores.

Los disparadores siempre, para todos los planes, podrán ser los mismos, ya que la forma de especificar cuando se lanza una acción no tiene por qué cambiar. Además, esto nos facilitará el diseño, ya que no tendremos que diseñar disparadores distintos para cada uno de los planes.

4.2.2 Disparadores

Los disparadores deberán ser lo más genéricos posible, pudiendo el usuario establecer cualquier tipo de situación para la que una acción será ejecutada:

- **Lanzar acción puntualmente**: Si se cumplen una serie de condiciones, en la que como datos podemos tomar cualquiera de las medidas que tome el submarino, tanto externas como internas, posición o estimación de la posición,

tiempo transcurrido desde el inicio de la misión, fecha y hora, etc, se ejecutará la acción o acciones que estén relacionadas con este disparador.

- **Lanzar acción en intervalos**: Se permitirá al usuario lanzar una acción en determinados intervalos de tiempo (por ejemplo, que se mida la temperatura, desde el minuto 1 hasta el minuto 300 cada 5 minutos). Aunque quizás no sea tan interesante, también se le permitirá lanzar una acción en determinados intervalos de otras medidas que pueda muestrear el AUV, como pueden ser distancias, temperaturas, etc.
- **Lanzar una acción por una excepción**: Esta última posibilidad permitirá al usuario lanzar acciones cuando ocurra algún tipo de excepción en el submarino. Las excepciones pueden ir desde el agotamiento de la batería hasta la detección de alguna avería en el AUV. Aunque esta parte es más dependiente de la capacidad que tenga el AUV para detectar excepciones, por lo tanto el usuario deberá tener en cuenta que excepciones son soportadas por el submarino que va a llevar a cabo la misión.

Para poder lanzar una acción o un conjunto de acciones, tendrán que estar asociadas a un conjunto de disparadores. Con el fin de que este conjunto de disparadores abarque el mayor número de situaciones posibles, se permitirá que en el mismo conjunto de disparadores que lanza un conjunto de acciones se incluyan los tres tipos de disparadores. Para que se lance la acción o conjunto de acciones se deben cumplir todos los disparadores que contiene el conjunto de disparadores asociado a ese conjunto de acciones. Si hay un sólo disparador que no se cumple no se lanzará la acción. Es decir, la relación entre los distintos disparadores de un mismo conjunto será de tipo "*AND*".

4.3 Medición

En principio el usuario va a desarrollar una misión sin conocer qué submarino lo va a llevar a cabo. No tiene por qué conocer sus especificaciones técnicas. Por ello sería interesante que en el plan de medición se indique qué magnitud es la que desea que se mida, y no el sensor con el que se desea medir.

El AUV debe tener una relación de las magnitudes a medir y relacionarlas con los sensores que tiene equipados. Cuando reciba la orden de medir determinada magnitud, se comprobará internamente qué sensores pueden medir esa magnitud y se mandará la orden de muestrear éstos sensores. El submarino puede decidir si quiere tomar las muestras de un único sensor, o tomarlas de varios sensores.

El usuario puede definir determinadas formas de medición para cada uno de los tipos de navegación que se van a seguir. Si el usuario quiere especificar una medición, lo podrá hacer de la siguiente forma:

```
// Lista de restricciones de muestreo, entre waypoints
ACCION1
      Muestreos.magnitud = temperatura
      Muestreos.muestras = 1000
      Muestreos.frecuencia = 1 Hz
      Muestreos.resolucion = 1ªC
ACCION1
```

```
ACCION2
      Muestreos.magnitud = profundidad
      Muestreos.muestras = 1000
      Muestreos.frecuencia = 1 Hz
      Muestreos.resolucion = 1 metro
ACCION2
```

El usuario debería especificar, para tomar una medida, al menos los siguientes parámetros:

- ✓ **Magnitud**: Es la magnitud que queremos medir. Deberíamos poder especificar una magnitud y no un sensor, ya que se supone que a priori no conocemos que submarino va a realizar la misión. El submarino se encargará de mapear esta magnitud a un sensor capaz de medirla.
- ✓ **Muestras**: El usuario especificará el número de muestras que desea que tome el AUV.
- ✓ **Frecuencia**: Será la frecuencia con la que tomará el AUV las muestras.
- ✓ **Resolución**: Si el usuario necesita tomar medidas con una resolución específica, podrá especificarla en el plan de medidas.

Si a este plan le añadimos los disparadores para saber cuándo se lanzan estas acciones, tendremos un plan de medidas completo.

Un posible esquema gráfico de lo que puede ser el plan de medidas se puede ver en la Ilustración 8.

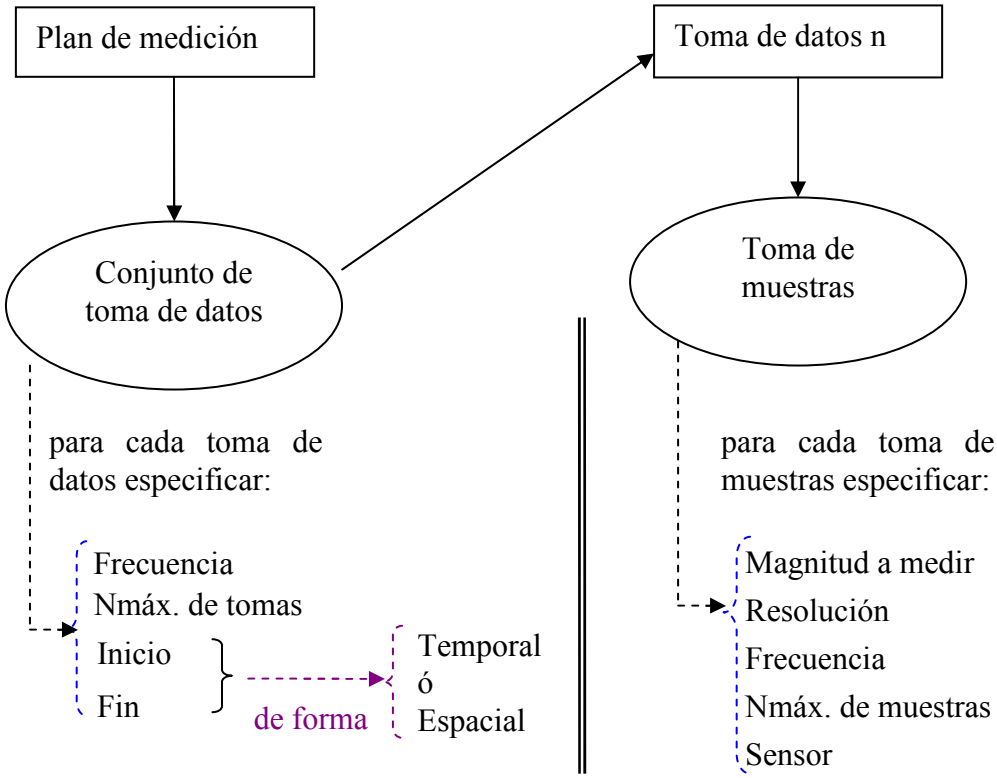


Ilustración 8: Esquema de la medición

Un plan de medición podrá tener muchas tomas de datos. A la hora de definir una medición tendremos que especificar cuándo se tomarán (disparadores, que estarían representados en el diagrama anterior por la especificación de la toma de datos) y una vez se cumpla esta condición, especificar cuántas muestras queremos de qué magnitud y a qué frecuencia, resolución, etc se van a tomar.

Por ejemplo, tenemos en el plan de medidas que se va a realizar la medición de la salinidad cada hora. Esto sería una toma de datos. Ahora hay que especificar cómo se van a tomar los datos cuando se cumpla esta condición (que haya transcurrido una hora), cuántas muestras se van a recoger en esa toma, con qué resolución etc.

Además, si el usuario conoce el submarino, o los sensores con los que se van a llevar a acabo la misión, podrá especificar tanto el sensor, como la configuración que debe tener éste a la hora de tomar medidas.

En este esquema los disparadores encajarían en la toma de datos, mientras que las acciones en sí encajarían en la parte de la toma de muestras (qué medir, cuantas veces, y con qué configuración).

4.4 Comunicaciones

En la comunicación van a participar dos actores principalmente: el AUV y el software planificador y controlador de la misión. Los tipos de comunicaciones que se podrían dar son los siguientes:

- Entre el planificador y el AUV: Se tratará de la comunicación que se establece entre el AUV y el planificador. Puede haber varios objetivos en este tipo de comunicación:
  - Comandar el AUV para que funcione como un ROV.
  - Establecer la misión a realizar.
  - Monitorizar la misión.
  - Modificar la misión en curso.
- Entre dos AUVs: Este tipo de comunicación servirá para que varios AUVs puedan llevar a cabo una misión coordinada. Es decir, una misión multiAUV.
- Entre dos planificadores: Esta comunicación la usaremos para coordinar distintas misiones de AUVs o compartir datos.

La definición de la forma en la que el submarino se va a comunicar con el planificador resulta algo compleja, sobre todo por cómo va a ser llevada a cabo la comunicación. La razón es que el submarino no se va a poder comunicar siempre. Para que se pueda comunicar con el planificador, el AUV va a tener que encontrarse en la superficie.

Una de las ideas aportadas sería la de que el plan de comunicaciones tenga prioridad en su ejecución en el AUV sobre los otros dos planes, el de navegación y el de mediciones. Esto nos aportaría cierta independencia en la definición del plan de comunicaciones por parte del usuario, ya que de otra forma, un plan de comunicaciones sería especificado para un plan de navegación en concreto, que el usuario tendría que

conocer al desarrollar el plan de comunicaciones. Es decir, tendríamos que conocer cuándo está emergido el submarino, para conocer en qué puntos podremos comunicarnos.

A continuación se lista una serie de especificaciones que habría que incluir en el plan de comunicaciones:

- Datos a enviar desde el AUV.
- Datos a recibir por el AUV.
- Fuente/Destino de la información

4.4.1 Qué datos se van a enviar.

De forma muy similar a lo que teníamos con el plan de mediciones, en esta ocasión deberemos dejar que el usuario especifique qué datos desea enviar al centro de control de la misión. Llegados a este punto, hay varias opciones.

- **Emisión de datos de la misión**: La primera de las opciones sería que el usuario decida que quiere que el submarino emita todos los datos que ha ido recogiendo siguiendo la misión especificada por el usuario, al menos lo que ha recogido desde la última comunicación en la que emitió estos datos. Es decir, el AUV necesita almacenar los datos que han ido recogiendo los sensores del submarino, además de datos de posición. Este tipo de comunicación se utiliza para que el usuario pueda monitorizar la misión del submarino.

- **Comunicación de avisos**: El AUV enviará avisos al planificador o a otros AUVs, para indicar algún determinado tipo de aviso. El submarino va a poder avisar del nivel de realización de la misión, enviar SOS en caso de que ocurra algún problema, o simplemente, para avisar de que todo va bien.

- **Sin emisión de datos**: Cabe la posibilidad de que el usuario quiera que el submarino realice simplemente una emersión de control, para que el submarino contraste su ubicación real con la ubicación estimada por él, y dar una señal al controlador de la misión de que sigue activo. También, mientras el submarino se encuentre en línea se puede enviar distintos comandos y datos al submarino para modificar su plan o porque quieren añadirse datos a la información que tiene el submarino.

- **Emisión de datos personalizada**: Este tipo de emisión de datos podría servir para dejar al usuario que especifique los datos que desea que el submarino emita cada vez que emerja. El usuario especificará las medidas que desea que sean enviadas. Hay dos formas de hacerlo:

    - **Especificando un dato**: con dato nos referimos a todo el historial de mediciones tomadas para una medida específica. Si especificamos un dato, se emitirá todo el registro de mediciones.

- **Especificando una medida**: Si especificamos una medida en lugar de un dato, obtendremos valor escalar o atómico de esa medida obtenido por el subsistema sensorial, lo cual incluye sensores simulados.

4.4.2 Qué datos se van a recibir

Durante todos estos tipos de comunicaciones se podrán también emitir datos al submarino o comandos. Por ello, también el usuario podrá especificar el tiempo que deberá encontrarse en línea el submarino en una emersión, para dar tiempo al ordenador de tierra de comunicarle nuevos datos o un nuevo plan.

Los datos que se van a comunicar desde el ordenador central hasta el submarino podrían ser los siguientes:

- **Datos batimétricos**: Si el submarino necesita conocer los datos batimétricos de una zona, los cuales no han sido cargados antes por falta de espacio en disco, el ordenador central le proporcionará esos datos siempre que pueda.

- **Recepción de datos personalizada**: Al igual que lo que ocurría con la emisión de datos, la recepción de datos, también se podrá personalizar de forma que el usuario decida que datos quiere que se puedan recibir. Existen también dos formas de especificar que se quiere recibir, similares a la emisión personalizada de datos:

  - **Especificando un dato**: Al igual que en la emisión de datos, especificando un dato estamos especificando que queremos recibir todo el registro de datos tomados para esa medida.

  - **Especificando una medida**: Implicará que sólo querrá recibir un valor escalar o atómico de esa medida obtenido por el subsistema sensorial, lo cual incluye sensores simulados..

- **Comandos**: El usuario desde el ordenador central emitirá comandos al submarino para que el AUV pueda actuar como un ROV y por teleoperador.

- **Nuevos planes**: El usuario puede especificar nuevos planes al AUV, para que éste modifique la misión.

4.4.3 Fuente/destino de la información.

Para especificar la fuente o el destino de la información, se pueden incluir en el plan de comunicación ristras que identifiquen un receptor/emisor de información mediante una dirección IP a la que el submarino se podrá conectar para recibir/enviar la información especificada en el plan.

4.4.4 Ejemplo de comunicación.

A continuación se mostrarán una serie de ejemplos en el que podremos ver aproximadamente lo que necesitamos del plan de comunicaciones.

```
// Envío de datos
Comunicación.enviar.datos = Registro.temperatura
Comunicación.enviar.datos = Registro.posicion
Comunicación.enviar.datos = Registro.realizacionMision
```

```
// Toma de datos
Comunicación.tomar.datos = posicion
Comunicación.tomar.datos = batimetria
Comunicación.tomar.datos = corrientesMarinas
```

4.5 Almacenamiento

El submarino no tiene por qué almacenar todas las medidas que tome, ya que muchas de ellas servirán para la propia navegación del submarino. Por ello será interesante definir las condiciones en las que el submarino almacenará determinadas medidas. El plan de almacenamiento será el encargado de decidir que datos se van a almacenar en el sistema.

Este plan irá bastante ligado al plan de mediciones y al de comunicaciones. Estará ligado al plan de comunicaciones debido a que sólo se van a poder enviar los datos que hayan sido previamente almacenados por el submarino, por ello los datos a comunicar especificados en el plan de comunicaciones deben haberse especificado también en el plan de almacenamiento si queremos que se envíen correctamente.

Además estará bastante ligado al plan de mediciones debido a que si no se toman unas determinadas medidas no se podrán almacenar. Por ello, el hecho de indicar que se va a almacenar un dato no forzará a que se mida, del mismo modo que el hecho de decir que se mida en el plan de medidas no forzará a que se almacene. De este modo se dispone de planes ortogonales o independientes.

Con ello se consigue la máxima funcionalidad, ya que puede indicarse que simplemente se mida algo sin almacenarlo, o que se almacene algo sin indicar que se mida (en el plan de medidas), o que se mida y almacena. No se considera erróneo o incoherente que se almacene algo que no se mide en el plan de medidas, pues es posible que el usuario quiera que se almacene una medida, ya que en el caso de que el usuario emita un comando al AUV para tomar una medida, ésta se almacenaría.

Una de las diferencias que hay que tener en cuenta con el plan de mediciones, es que este último decidirá la forma en la que se van a tomar los datos. Es decir, en el plan de mediciones el usuario podrá definir la configuración del sensor cuando se vaya a

tomar las muestras. El plan de almacenamiento simplemente va a servir para almacenar, no va a ser posible describir en éste la forma en la que se desea que se tome la medida.

Otra de las diferencias es que podemos solicitar que se almacenen no solo medidas tomadas de sensores, sino también datos internos que maneje el propio submarino para distintos fines como puede ser la estimación de su posición o del estado de su batería.

Lo único que vamos a especificar en cada una de las acciones que definamos en el plan de almacenamiento es qué medida o dato almacenar y cuántas muestras queremos que almacene.

A continuación se muestra un ejemplo de especificación de las acciones en un plan de almacenamiento.

```
ACCION1
      Muestreos.magnitud = temperatura
      Muestreos.muestras = 1000
ACCION1

ACCION2
      Muestreos.magnitud = profundidad
      Muestreos.muestras = 20
ACCION2

ACCION3
      Muestreos.magnitud = posicion
      Muestreos.muestras = 1
ACCION3
```

Con este ejemplo, cada vez que se cumplan las condiciones de los disparadores se va a almacenar el número de muestras indicado para cada una de las medidas especificada.

4.6 Supervisión

El plan de supervisión se encargará, tal y como su nombre indica, de "supervisar" el funcionamiento de la misión.

Cuando hablamos de supervisar el funcionamiento de una misión en un AUV, nos referimos a monitorizar todas las posibles excepciones y situaciones que se den durante la ejecución de ésta y, si es necesario, actuar en consecuencia, ya sea para abortar la misión, para realizar distintos tipos de comandos frente a un imprevisto para ésta, o para cambiar de misión por otra que a su criterio pueda ser más eficaz en determinado tipo de circunstancias.

Las circunstancias en las que va a actuar el plan de supervisión, las definirán los disparadores, tal y como se definen en los demás planes. Las acciones son las que van a ser más específicas para este plan.

Tenemos dos alternativas:

- **Ejecutar plan alternativo**: Podemos especificar que ante unas circunstancias en concreto se cambie la misión que está ejecutando el AUV. Podemos especificar tanto la nueva misión como una lista de parámetros asociados a la misión que queremos que se ejecute.
- **Ejecutar una lista de comandos**: Se puede escoger la opción de ejecutar una lista de comandos cada vez que se produce una circunstancia en particular., en lugar de cambiar un plan completo. Esto nos serviría para evitar o aprovechar alguna situación en un momento determinado, pero sin dejar de ejecutar la misión preestablecida.

Hay un tipo de comando con el que se podrá cambiar, en tiempo de ejecución, algunos parámetros con los que trabajarán los demás planes. Se trata del comando "*CambiarVariableMision*".

Este comando se encargará de modificar una variable definida en cualquiera de las misiones antes mencionadas (Plan de navegación, de medición, de comunicación y almacenamiento). Es decir, el plan de supervisión además podrá modificar el comportamiento de planes ya definidos en la misión.

4.7 Diseño de la misión.

En esta sección se explicará finalmente como se almacenarán los distintos planes que conforman una misión del submarino, en qué ficheros se van a almacenar y qué formato tendrán estos planes.

Hemos elegido para la representación de los distintos planes y de la misión en general que debe seguir el AUV el formato XML [XML-W3C]. XML son las siglas en inglés de E**x**tensible **M**arkup **L**enguaje (lenguaje de marcas extensible). Es un metalenguaje extensible desarrollado por la World Wide Web Consortium(W3C) [W3C].

El metalenguaje XML nos permite realizar un intercambio de información estructurada entre distintos sistemas. En nuestro caso, entre el submarino y el planificador de misiones. Las ventajas que tiene este metalenguaje son:

- Es un lenguaje **extensible**: Si ya estamos trabajando con unas especificaciones iniciales para los documentos XML, es sencillo ampliar éstos para añadir elementos y atributos sin que afecte a los analizadores sintácticos que existan en ese momento de ese fichero XML. Por ejemplo, si tenemos un plan definido en XML y añadimos un elemento más denominado "profundidad" al plan en una nueva especificación del mismo, no afectaría a la implementación actual, ya que simplemente no tendría en cuenta que puede existir en ese plan un elemento con esa denominación y lo ignorará. Las versiones anteriores del analizador sintáctico podrán seguir leyéndolo como hacían hasta ese momento.

- El analizador es un componente estándar. Hay implementadas librerías para los lenguajes de programación más usados para leer y escribir ficheros XML, por lo que además de ahorrarse este problema el usuario, se pueden usar librerías que ya están muy probadas y corregidas.

- Es sencillo para un usuario leer un documento XML, ya que cada elemento y atributo puede tener un nombre fácilmente identificable para una persona. Además la estructura de un documento XML es intuitiva y sencilla.

- Es multiplataforma: es un lenguaje independiente de la plataforma en la que se utilice.

- Fácilmente interpretable. No es un lenguaje indescifrable para un usuario normal. Es bastante autodescriptible y una persona no necesitaría grandes esfuerzos para entender un fichero XML.

Sin embargo este tipo de metalenguaje también tiene una desventaja, y es que no es compacto. Esto es debido a las etiquetas que hay que utilizar para todos los elementos y atributos que contenga el fichero. Esto lo hace quizás más grande de lo necesario, aunque más claro para el usuario que lee el fichero.

4.7.1 Ficheros XML "bien formados" y válidos.

Para que un fichero XML esté bien formado debe cumplir una sintáctica básica que deben cumplir todos los documentos XML. Es decir, que se componga de elementos, atributos y comentarios como XML especifica que se escriban.

Pero además necesitamos que los documentos XML sean válidos. Hay varias formas de definir la semántica de un fichero XML, pero la que hemos usado para los planes son los esquemas XSD**[W3C-XSD]**.

4.7.2 Ficheros que conforman la misión.

Como hemos visto, una misión consta de varios planes. Al menos cinco planes: plan de navegación, plan de medidas, plan de comunicaciones, plan de almacenamiento y plan de supervisión. Cada uno de estos planes va a estar especificado en un fichero XML. Además, tendremos algunos ficheros en los que definiremos algunos parámetros globales a la misión.

En la Tabla 1 se puede ver una relación de ficheros que puede contener la misión:

Directorios	Formatos	Descripción
./planes		Presenta todos los planes que se pueden usar en las misiones
./planes/pda	XML	Presenta todos los planes de almacenamiento empleados en las misiones
./planes/pdm	XML	Presenta todos los planes de medida empleados en las

		misiones
./planes/pdc	XML	Presenta todos los planes de comunicación empleados en las misiones
./planes/pdn	XML	Presenta todos los planes de navegación empleados en las misiones
./planes/pds	XML	Presenta todos los planes de supervisión empleados en las misiones
./misiones	XML	Presenta las misiones, cada una referenciando a cinco planes (anteriores) y de forma opcional a de 0 a varias tablas de parámetros.
./tablasparametros	XML	Se definen conjuntos de parámetros para misión, que actuarán como variables de estado de la misión (sirve para modificar el plan en tiempo de ejecución).
./esquemas	XSD	Se definen el formato de los ficheros xml de planes (1 para cada una de las cinco clases) , el formato del fichero de misión xml y el de la tabla de parámetros.
./esquemas/lib	XSD	Define algunos aspectos auxiliares comunes a todos los planes XML.

Tabla 1: Directorios

En las siguientes secciones se explican cada uno de los ficheros listados en la tabla.

4.7.3 Fichero de misión (<Mision>.xml)

4.7.3.1 Semántica

Este fichero describe la misión a realizar por el AUV. Es una misión no cooperativa, que se estructura en cinco planes que debe cumplir el AUV para completar la misión.

Además, el fichero puede contener o no varias tablas de parámetros, cada una con una lista de parámetros que deben entenderse como una variable de global común y utilizable en todos los planes, y que facilita la modificación de planes en tiempo de ejecución como veremos más adelante.

A este fichero XML Le corresponde el esquema **./esquemas/mision.xsd**.

4.7.3.2 Descripción

La misión, a parte de un identificador y nombre, se forma de cinco elementos obligatorios y únicos primeramente, donde cada elemento tiene el atributo **fichero**, para referenciar al fichero donde se encuentra el plan correspondiente.

Los distintos planes que alberga una misión son:

- **pda:** le corresponde el fichero XML con el plan de almacenamiento.
- **pdc**: el plan XML de comunicaciones.
- **pdm:** el plan XML de medidas.

- **pdn:** el plan XML navegación.
- **pds:** el plan XML de supervisión.

```
<?xml version="1.0" ?>

<mision id="1" nombre="Mision 1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.mision.com"
        xsi:schemaLocation="http://www.mision.com ../esquemas/mision.xsd">

  <pda  fichero="../planes/pda/pda.xml" />
  <pdc  fichero="../planes/pdc/pdc.xml" />
  <pdm  fichero="../planes/pdm/pdm.xml" />
  <pdn  fichero="../planes/pdn/pdn.xml" />
  <pds  fichero="../planes/pds/pds.xml" />

  <tablaParametros  fichero="../tablaparametros/tablaparametros.xml" />

</mision>
```

A continuación de los cinco primeros elementos descritos, se puede dar de 0 a muchos elementos de tipo **tablaParametros**, cada uno referenciando a una tabla de parámetros (en formato xml) en el directorio **./tablasparametros**.

Se sobreentiende (no se comprueba a través del XSD) que si se citan diversas tablas de parámetros deben de ser distintas, y que las variables de estado definidas dentro de cada tabla deben ser distintas entre sí y de otras variables de otras tablas de parámetros citadas en la misma misión.

Aún así, si las variables de estado del conjunto de tablas se repitieran, el efecto sería el de sumarse las referencias (lo veremos más adelante) de cada variable.

4.7.4 Fichero Tabla parámetros (<tablaparametros>.xml)

4.7.4.1 Semántica

Se encarga de definir un conjunto de variables de estado de la misión, (cada una identificable mediante un nombre) y definir todos los puntos en los que son usadas dichas variables por los distintos planes.

Esto permite actualizar los planes de forma dinámica y rápida. Como valor de ciertos atributos de los planes se le puede asignar el valor de una de estas variables globales, de forma que basta con cambiar el valor de una variable de estado dentro de la tabla, para actualizar todos los planes en los que es usada.

Le corresponde el esquema **./esquemas/tablaparametros.xsd**.

4.7.4.2 Descripción

La tabla, a parte de un identificador y nombre, se forma de uno a varios elementos de tipo parámetro. Cada uno define una variable global o de estado.

```
<?xml version="1.0" ?>
<tablaParametros id="1" nombre="Tabla Parámetros 1"
        xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xmlns:parametros="http://www.parametros.com"
        xmlns=http://www.tablaparametros.com
        xsi:schemaLocation="http://www.tablaparametros.com../esquemas/tablaparametros.xsd">

<parametro nombre="SensorTemperatura" valor="TCM345">
<parametros:elemento
nombre="/planDeMedicion[@id='1']/tarea[@id='1']/acciones/medir[@id='1']/sensor[@id='1']/@nombre" />
</parametro>

<parametro nombre="VelocidadCrucero" valor="4">

   <parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='1']/velocidad/posicion/@x" />

   <parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='1']/velocidad/posicion/@y" />

   <parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='1']/velocidad/posicion/@z" />

</parametro>

 </tablaParametros>
```

Cada elemento de tipo **parámetro** (o variable de estado), se define en el fichero XSD llamado **parametros.xsd**, que se encuentra en el directorio **./esquemas/lib**. Es importante ver que en un mismo fichero xml que represente una tabla de parámetros no puede haber dos parámetros con el mismo nombre.

Por tanto, cada **parámetro** tiene:

- un **nombre** por el que será referenciada la variable de estado en los distintos planes.
- un **valor**.
- una lista de uno a varios elementos (llamados **elemento** cada uno, valga la redundancia), donde cada uno referencia algún plan y elemento dentro del plan en que es usada la variable de estado (en el atributo **nombre**). Por tanto habrán tantos **elemento** como veces se use la variable de estado en los distintos planes.

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Parametro		Definición de una variable con una lista de elementos en los		

		que se utiliza éste, un nombre y un valor.	nombre: nombre de la variable	X
			valor: valor de la variable	X
	Elemento	Referencias a los elementos que forman parte de los distintos planes de la misión.		Pueden aparecer varios elementos de este tipo
			nombre: dirección del elemento dentro de un plan	X

Tabla 2: Elemento Parámetro

En cuanto al formato para indicar la dirección del atributo al que se asigna el valor de la variable de estado, se utiliza **xpath [W3C-XPATH]**, como veremos más adelante.

4.7.5 Aspectos generales de los ficheros XML de los distintos planes.

Como vimos en el análisis de los distintos planes, va a haber determinadas estructuras que se repitan en todos los planes de la misión. Por ejemplo, el listado de acciones y disparadores. Aunque cada acción depende del tipo de plan, el esquema disparador → ejecuta → acción se va a repetir y utilizar en todos los planes. Además, los disparadores van a ser los mismos para todos los planes.

4.7.5.1 Disparadores

Los disparadores son una lista de condiciones que van a determinar cuando ejecutar una acción, en el caso de los cuatro planes que no incluyen al plan de navegación, o cuando comenzar a navegar de una determinada forma, en el caso del plan de navegación.

Dentro del elemento "disparadores", puede haber una lista de sub-disparadores (o ninguno, lo que equivale a que la tarea se ejecute constantemente y significa que se activarán todas las acciones y no se desactivarán hasta que el plan se anule o termine), de tres tipos:

- **Condición**: que una determinada magnitud medida por el AUV llegue a un determinado valor. Por ejemplo: "*envía un aviso si la temperatura externa es superior a 15 ºC*."

  Los atributos son:

  - **id**: identificador.
  - **Medida**: medida que se mide o compara, tiempo o variables de estado del AUV como el waypoint, temperatura exterior o velocidad.
  - **Operador**: operador lógico que se aplica, donde **lt** equivale a <, **gt** equivale a > y **eq** equivale a =.
  - **Valor**: umbral de referencia.
  - **Unidad**: en la que se expresa el **valor**.

```
<disparadores:condicion id="1" medida="temperatura exterior" operador="gt"
valor="10" unidad="ºC" />
```

- **Excepción del sistema**. Por ejemplo: "*emerge si se agotan las baterías*". : Solo tiene un atributo (además del identificador), que es el **nombre** de la excepción (existe una lista de excepciones).

```
<disparadores:excepcion id="1" nombre="agotamientoBaterias" />
```

La lista de excepciones no se definirá en ningún fichero de esquema XML, como podría ser el fichero **./esquemas/lib/listas.xsd**, pues se tendría el problema de que el usuario no podría gestionarlas (crear nuevas, eliminar, etc.). En su lugar, la lista de excepciones será parte del equipamiento del AUV, lo cual se explicará en la documentación del equipamiento. Así, se tendrá la lista de excepciones en un fichero XML gestionable por el usuario.

– **Intervalo**: dispara cuando el valor de la medida o del tiempo o variable de estado del AUV se encuentre entre los valores especificados por un rango. Por ejemplo: "*mide temperatura con X frecuencia si estas entre el waypoint 3 y 6, de uno en uno*".

Sus atributos son

- **id:** identificador**.**
- **Medida**: ídem que Condición.
- **Inicio**: inicio del rango a partir del cual se dispara.
- **Fin**: fin del rango a partir del cual deja de dispararse.
- **Periodo**: Es el paso desde inicio a fin.
- **Unidad**: Es la unidad en la que están expresados Inicio, Fin y Periodo (que será la una unidad válida de la magnitud habida en **medida**).

```
<disparadores:intervalo id="2" medida="tiempo" inicio="30" fin="60"
periodo="10" unidad="minuto" />

<disparadores:intervalo id="4" medida="temperatura interior" inicio="2"
fin="INF" periodo="10" unidad="grados" />
```

Alguno de los valores de inicio o fin pueden valer INF o –INF, para indicar que el intervalo sigue al infinito o desde el infinito. Es como indicar que la tarea se ejecute para siempre cada 10 grados (en el último ejemplo por ejemplo).

Sea cual sea el sub-disparador, todos tienen un atributo identificador, y se tiene que cumplir que todos los sub-disparadores de la lista dentro del elemento **disparadores** de una tarea deben tener distinto identificador.

Como ya se dijo, en una tarea, el apartado **disparadores** puede tener una lista de condiciones, excepciones e intervalos en cualquier orden. Combinándose adecuadamente se puede conseguir:

– **AND lógico**: Por ejemplo: "*mide la temperatura si estas entre el waypoint 1 y el 13(**intervalo**), y si la temperatura es inferior a 15 ºC (**condición**)*".

```
<disparadores:disparadores>
   <disparadores:condicion medida="temperatura interna" operador="lt"
valor="15" unidad="grados" />
   <disparadores:intervalo medida="waypoint" inicio="1" fin="13" periodo="1"
unidad="waypoint" />
</disparadores:disparadores>
```

En este caso, se agregan dentro de la misma tarea, en el mismo apartado de disparadores todas las etiquetas unidas por AND lógico:

- **OR lógico**: En este caso, tiene que realizarse la tarea si se cumplen unas condiciones, u otras. Para ello, se ponen dos tareas distintas, cada uno con su condición adecuada en el disparador, y ambas con la mismas acciones, con esto se consigue hacer el or (si se cumple una condición, o la otra, o las dos, entonces, se realiza la acción).

  Por ejemplo: *"Si la temperatura son 10ºC, o la profundidad son 100 metros, envía un SOS"*

```
<tarea id="1" nombre="tarea1">

   <disparadores:disparadores>
      <disparadores:condicion medida="temperatura interna"
         operador="=" valor="10" unidad="grados" />
   </disparadores:disparadores>

   <acciones>
      <enviar mensaje="SOS"/>
   </acciones>

   <periodoInhibicion valor="10" unidad="minuto" />

</tarea>


<tarea id="2" nombre="tarea2">

   <disparadores:disparadores>
      <disparadores:condicion medida="profundidad"
         operador="=" valor="100" unidad="m" />
   </disparadores:disparadores>

   <acciones>
      <enviar mensaje="SOS"/>
   </acciones>

   <periodoInhibicion valor="10" unidad="minuto" />
</tarea>
```

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Disparadores		Lista de disparadores que se necesitará que se activen para ejecutar una acción determinada.		
	Condicion	Especificará una condición determinada.		Pueden aparecer varios elementos de este tipo o ninguno.
			id: identificador	X
			medida: medida sobre la que se realiza la condición.	X
			operador: operador utilizado para la condición.	X
			valor: valor con el que se compara la medida tomada.	X
			unidad: unidad en la que se expresa el valor de la condición.	X
	Intervalo	Especificará un intervalo en el que se deberá ejecutar una acción.		Pueden aparecer varios elementos de este tipo o ninguno.
			id: identificador	X
			medida: medida sobre la que se realiza la comparación.	X
			inicio: inicio del rango a	X

			partir del cual se dispara.	
			fin: fin del rango a partir del cual se dispara.	X
			periodo: paso (frecuencia con el que se ejecuta) desde inicio hasta el fin.	X
			unidad: unidad con la que se expresa el inicio, el fin y el periodo.	X
	Excepcion	Excepción con la que se debe lanzar una determinada acción.		Pueden aparecer varios elementos de este tipo o ninguno.
			id: identificador	X
			nombre: nombre de la excepción.	X

Tabla 3: Elemento Disparadores

4.7.6 Plan de Navegación

4.7.6.1 Semántica

Describe el comportamiento de navegación a seguir por el AUV a medida que avanza la misión. Como vimos durante el análisis de la misión, podemos tener varios tipos de recorridos: el primero sería una ruta fija que el submarino seguiría, marcada por unos waypoints. El segundo tipo constaría en la determinación de un área y la forma en la que el usuario desea que la recorra el submarino (dejándose llevar por la corriente, recorriéndola en zigzag o espiral etc.).

También veíamos como sería muy útil tener lo que llamábamos una misión combinada, que consistía en que el plan de navegación no solo estuviera determinado por un área o una ruta únicamente, sino por la mezcla de estas dos formas de recorrer una ubicación determinada.

Por ello, el plan de navegación puede estar compuesto por varios tramos de dos tipos distintos:

- **rutas**: Es un tramo recorrido linealmente a través de transectos unidos por sus vértices, de forma que se define cada uno de los waypoints (con información asociada) por los que debe ir pasando el AUV y los transectos o como recorrer esa lista de waypoints.

- **áreas**: En este caso, se define un volumen en el que el AUV se mueve siguiendo diversos comportamientos configurables. La diferencia es que no hay unos waypoints o puntos por los que pasar de antemano definidos, sino que en función del volumen y comportamiento que se elige para el AUV éste se mueve.

Incluso, para la misma área se pueden definir distintos comportamientos, asociando estos comportamientos a disparadores disjuntos. Así, en función del disparador que se active en cada momento así se comportará el AUV.

Es importante el orden en que se organizan estos dos tipos de tramos en el fichero, pues dos tramos consecutivos indica que el AUV cuando termine el tramo anterior debe recorrer el posterior de forma continuada.

Adicionalmente, se pueden definir las denominadas "Zonas Prohibidas". Estas zonas estarás definidas por un conjunto de vértices que limitarán una zona que no es segura para el submarino. El submarino debe hacer lo posible por evitar estas zonas, ya que a menudo limitarían, por ejemplo, pequeñas islas o arrecifes coralinos.

Las zonas prohibidas se especificarán en el plan de navegación en el mismo nivel que las áreas y las rutas, y no importa el orden ya que el submarino solo las tendrá en cuenta para evitarlas, y no para seguirlas.

Aparte, se puede indicar que todo el plan descrito de la forma que se ha explicado se repita de forma cíclica tantas veces como se requiera.

Se define en el esquema **./esquemas/pdn.xsd**. Veremos que las áreas presentan también disparadores, que se definen como todos los disparadores de todos los planes en **./esquemas/lib/disparadores.xsd**.

4.7.6.2 Descripción

El plan de navegación es el único plan que no sigue la estructura de los otros cuatro planes, como veremos más adelante. Cada uno de esos cuatro planes estaría formado por una lista de tareas que a su vez se diferenciarían en una lista de disparadores y una lista de acciones. Empezaremos explicando el elemento raíz del fichero: es el elemento planDeNavegacion, que presenta los atributos:

- **id**: Será el identificador de la ruta.

- **Nombre**: Nombre de la ruta.

- **repetición**: En este atributo se especifica cuantas veces se quiere que se ejecute todo el plan (consecución de elementos de tipo ruta y de tipo área). Se puede especificar desde 0 (con lo cual se invalida todo el plan, no se ejecuta el plan) hasta INF (se ejecuta constantemente el plan).

- referencia a los esquemas xsd utilizados.

```
- <planDeNavegacion id="1" nombre="PdN" repeticion="1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:disparadores="http://www.disparadores.com"
xmlns="http://www.pdn.com" xsi:schemaLocation="http://www.pdn.com
../../esquemas/pdn.xsd">
```

Dentro del elemento raíz, se puede dar, desde que no tengamos ningún tramo que se pueda recorrer (lo que significa un plan vacío, el AUV no se moverá por no tener plan de navegación) a tener muchos tramos, de tres clases principalmente: **ruta**, **área**, **zonas prohibidas**. No se puede repetir el identificador de estos tramos dentro del mismo elemento raiz, y por tanto, dentro del plan.

Tener en cuenta que el orden en que se colocan rutas y áreas tiene una semántica: una vez el AUV complete el tramo expresado antes en el plan, continuará recorriendo el tramo consecutivo en el fichero (no considerándose las zonas prohibidas en esta norma, pues no "se recorren").

Se describen a continuación los tres tramos que se pueden especificar en el plan de navegación:

4.7.6.2.1 Ruta

```
<ruta id="1" nombre="salida">

  <waypoint id="1">

    <pose>
      <posicion x="30" y="20" z="-10" unidad="m" />
      <orientacion roll="0" pitch="0" yaw="1" unidad="rad" />
       <incertidumbre valor="2" unidad="m" />
    </pose>

    <velocidad>
      <posicion x="1" y="0.2" z="0" unidad="m*s^-1" />
      <orientacion roll="0" pitch="0" yaw="0.1" unidad="rad*s^-1" />
    </velocidad>

    <interpolacion valor="10" unidad="m" />

  </waypoint>

  <waypoint id="2">
    <pose>
      <posicion x="30" y="20" z="-10" unidad="m" />
      <incertidumbre valor="2" unidad="m" />
    </pose>
```

```
      <interpolacion  valor="10" unidad="m" />
   </waypoint>

   <transecto id="1" inicio="1" fin="2">
      <velocidad>
         <posicion x="1" y="0.2" z="0" unidad="m*s^-1" />
         <orientacion roll="0" pitch="0" yaw="0" unidad="rad*s^-1" />
      </velocidad>
   </transecto>

</ruta>
```

En el elemento ruta podemos ver elementos de tipo **waypoint** y **transectos**. Para especificar una lista de transectos necesitamos más de un waypoint, por lo que si tuviéramos un solo waypoint estaríamos especificando una ruta en la que el submarino debe mantener la posición de en ese waypoint.

Por ello, un elemento de tipo ruta debe tener especificados al menos un waypoint. No puede haber dos waypoints con el mismo identificador dentro de la misma ruta. Además, el orden de los waypoints en el fichero no indica nada, ya que el orden de recorrido en realidad está especificado en la lista de transectos.

Elemento Waypoint:

Cada elemento de tipo waypoint presenta internamente un elemento pose, un elemento velocidad y un elemento interpolación. Tanto el elemento pose con el elemento velocidad podrán venir definidos por seis componentes cada uno de ellos, tres especificarán el movimiento lineal del submarino, y los otros tres el movimiento angular (roll, pitch y raw).

- **Elemento pose**: Este elemento es obligatorio, define la posición lineal x,y,z donde se sitúa el waypoint, y la orientación (o posición angular) que debe tener el AUV cuando alcance este waypoint. Además, este elemento definirá la incertidumbre, que será el error permitido al pasar por un waypoint. Se limitará con una circunferencia de radio = incertidumbre y con centro establecido en el elemento posición. La pose se conforma de tres elementos. Lo podemos ver en la siguiente tabla:

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Pose				X
	Posición	Elemento que especificará la posición lineal del waypoint.		X
			x: coordenada x	X
			y: coordenada y	X

			z: coordenada z. Puede indicar profundidad en algunas de las unidades de medida de posición.	X
			unidad: unidad de medida en al que se especifica la posición del waypoint	X
	Orientación	Es la posición angular que deberá tener el submarino en el waypoint.		
			roll (alabeo)	X
			pitch (cabeceo)	X
			yaw (guiñada)	X
			unidad	X
	Incertidumbre			
			valor: Valor del radio de incertidumbre	X
			unidad	X

Tabla 4: Elemento Pose

A continuación se muestra el trozo de XML que representa la pose:

```
<waypoint id="1">
<pose>
  <posicion  x="30" y="20" z="-10" unidad="m" />
  <orientacion  roll="0" pitch="0" yaw="1" unidad="rad" />
  <incertidumbre  valor="2" unidad="m" />
</pose>
```

- **Elemento Velocidad**: Este elemento define la velocidad lineal y la velocidad angular que debe tener el AUV cuando alcance el waypoint. Como se puede ver en la siguiente tabla, los dos elementos que contiene son la posición, que indicará la velocidad lineal, y la orientación, en al que se especificará la velocidad angular.

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Velocidad		Indica la velocidad que debe tener el submarino cuando alcance este waypoint.		
	Posición	Elemento que especificará la velocidad lineal del waypoint.		X
			x: coordenada x	X
			y: coordenada y	X
			z: coordenada z.	X
			unidad: unidad de medida en al que se especifica la velocidad del submarino	X
	Orientación	Es la velocidad angular que deberá tener el submarino en el waypoint.		
			roll (alabeo)	X
			pitch (cabeceo)	X
			yaw (guiñada)	X
			unidad	X

Tabla 5: Elemento Velocidad

- **Elemento Interpolación**: Determina la forma en que se deben interpolar lo transectos de entrada y salida de un waypoint (la pose y velocidad). El atributo valor se entiende como radio de la circunferencia/esfera que determina el arco de giro de un transecto a otro.

Elemento	Descripción	Atributos	Obligatorio
Interpolación	Interpolación de los transectos de entrada y salida de un waypoint (pose y velocidad)		
		Valor	X

		Unidad	X

Tabla 6: Elemento Interpolación

Elemento Transecto:

Además de los waypoints, una ruta puede tener una lista de transectos. No puede haber dos transectos con el mismo identificador.

```
<transecto id="1" inicio="1" fin="2">
  <velocidad>
   <posicion  x="1" y="0.2" z="0" unidad="m*s^-1" />
   <orientacion  roll="0" pitch="0" yaw="0" unidad="rad*s^-1" />
  </velocidad>
</transecto>
```

Los transectos indican el "camino" a seguir por el AUV, de forma que dos transectos consecutivos en el fichero indican al AUV que cuando termine de recorrer el primero, debe seguir con el siguiente transecto en el fichero.

En principio, tampoco lo obliga el xsd, pero es importante que cada waypoint sea utilizado (si un waypoint no esta en un transecto, no se recorre) una y solo una vez en el conjunto de transectos. Esto evita problemas asociados a la reutilización de los waypoints en varios transectos. En caso que se quiera hacer esto último, la solución sería duplicar el waypoint con otro id.

En el elemento transecto se indicará el waypoint inicial, el final para el transecto, además del tiempo en el que se debe recorrer o la velocidad, éstos dos últimos se establecen como parámetros optativos y excluyentes, es decir, si se especifica uno no se debe especificar el otro. En caso contrario, en el que se incluyan los dos elementos, el documento estaría mal formado, ya que se validaría contra el XSD.

La velocidad, se establecerá de la misma forma en la que establecíamos la velocidad en un waypoint, es decir, mediante una velocidad lineal y una angular.

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Transecto		Indicará los waypoints inicial y final que forman el transecto		
			id: Identificador del transecto	X
			inicio: el id del waypoint en el que se inicia el transecto. Se obliga a que señale a un waypoint existente en el fichero. Aunque	X

			el xsd permite reutilizar el mismo Waypoint en distintos transectos, no se considera buena práctica.	
			fin: el id del waypoint en el que acaba el transecto. Se obliga a que señale a un waypoint existente en el fichero. Aunque el xsd permite reutilizar el mismo Waypoint en distintos transectos, no se considera buena práctica.	X
	Velocidad	Elemento que especificará la velocidad lineal del transecto. Ver tabla Elemento Velocidad.		
	Tiempo	Tiempo en el que debe recorrer el submarino el transecto. La velocidad se calculará según el tiempo y el espacio a recorrer, además de otros factores como podrían ser las corrientes marinas.		
			valor	X
			unidad	X

Tabla 7: Elemento Transecto

4.7.6.2.2 Área

Toda área se define mediante una lista de cuatro tipos de elementos consecutivos que deben ir consecutivos:

- **Lista de vértices**: Es la lista de vértices que conforman el área. Debe haber como mínimo tres vértices. Véase que toda área es un prisma. Por lo que con los vértices sólo se define la cara superior del prisma, y mediante la profundidad se define el resto del cuerpo del prisma. Los vértices deben estar especificados secuencialmente para poder formar el prisma. Dos vértices consecutivos comparten arista, y el último vértice comparte arista con el primero.

- **Profundidad**: La profundidad establecerá una profundidad mínima y una máxima que el submarino puede alcanzar dentro del área.

- **Tiempo**: El tiempo máximo que debe permanecer el submarino en el área. Es un parámetro obligatorio.

- **Recorrido:** Habrá una lista de recorridos, que indicará cómo debe recorrer el submarino el área mientras se cumplan una serie de disparadores.

```
<area id="2" nombre="deriva">

  <vertice id="1" x="40" y="20" z="10" unidad="m" />
  <vertice id="2" x="20" y="10" z="20" unidad="m" />
  <vertice id="3" x="30" y="20" z="10" unidad="m" />
  <vertice id="4" x="20" y="20" z="10" unidad="m" />

  <profundidad minima="10" maxima="100" unidad="m" />

  <tiempo valor="30" unidad="minuto" holgura="10" />

  <recorrido id="1">
  <disparadores:disparadores />
  <deriva />
  </recorrido>

  </area>
```

Como se puede ver, un vértice estará compuesto simplemente de una posición indicada mediante las coordenadas *x*, *y* y *z*.

```
<vertice id="1" x="40" y="20" z="10" unidad="m" />
```

Elemento	Descripción	Atributos	Obligatorio
Vértice	Vértice del área que se está definiendo		X Deben haber al menos tres vértices, y uno existe un límite superior.
		x: Coordenada X.	X
		y: Coordenada Y	X
		z: Coordenada Z	X
		unidad: Unidad de medida en la que se expresa la posición.	X

Tabla 8: Elemento Vértice

El elemento profundidad determinará la profundidad máxima y mínima que tendrá el área. Con ella se terminará de definir el prisma que se forma con los vértices definiendo la cara superior, y la profundidad máxima y mínima. Este elemento es opcional,. Si no se especifica, el prisma irá desde profundidad cero hasta la profundidad marina.

```
<profundidad minima="10" maxima="100" unidad="m" />
```

Elemento	Descripción	Atributos	Obligatorio
Profundidad	Profundidad máxima y mínima dentro de un área.		
		minima: profundidad mínima	X
		maxima: profundidad máxima	X
		unidad: Unidad de medida en la que se expresa la profundidad.	X

Tabla 9: Elemento Profundidad

El elemento tiempo define el tiempo que debe invertir el AUV en recorrer el área. Pasado el tiempo debe abandonar el área. Éste es un elemento obligatorio.

```
<tiempo valor="30" unidad="minuto" holgura="10" />
```

Elemento	Descripción	Atributos	Obligatorio
Tiempo	Tiempo máximo que podrá pasar el submarino en el área.		X
		valor	X
		unidad: Unidad de medida en la que se expresa el tiempo.	X
		holgura: es el umbral de error respecto al valor,	

		más menos la holgura.	

Tabla 10: Elemento tiempo

El recorrido define el comportamiento que debe seguir el AUV dentro del área. Al menos debe definirse un recorrido, pero pueden definirse varios, donde cada uno define un comportamiento distinto. En esta lista, todos los recorridos deben tener distinto identificador.

Cada recorrido se define mediante una lista de disparadores, y una acción (o comportamiento). Los disparadores indican cuando el recorrido que los posee debe ejecutarse, y por tanto ese debe ser el comportamiento del AUV, o no.

Por ello, aunque el XSD no lo obliga, es importante que los distintos recorridos que se especifiquen tengan disparadores disjuntos y completos (siempre se cumple la lista de disparadores de algún recorrido, y sólo de uno).

```
<recorrido id="1">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="temperatura" operador="lt"
valor="25" unidad="ºC" />
     </disparadores:disparadores>

    <deriva />
</recorrido>
```

El elemento **disparadores** dentro del recorrido es obligatorio y único. Dentro del elemento **disparadores** se especifican todas las condiciones, siguiendo las mismas reglas que para los otros planes que contienen disparadores. En el caso de que no se especifiquen disparadores implicaría que este recorrido se dispara siempre, o lo que es lo mismo, que es el único comportamiento del AUV en el área.

Para cada elemento recorrido se define un tipo de recorrido. Hay tres tipos de recorridos distintos. Estos tres tipos de recorridos son excluyentes (se ejecutarán o uno u otro) y se corresponderán con tres tipos de comportamientos distintos.

También puede pasar que no se especifique ninguno de los tres, por lo que el AUV no desarrolla ningún comportamiento en el área.

Los tres elementos distintos que se pueden especificar para definir el comportamiento del submarino en el área son:

- **deriva**: Mantenerse a la deriva dentro del área. Sólo será necesario activar los motores para evitar salirse del área. No posee atributos.

```
<recorrido id="1">
    <disparadores:disparadores>
.....
     </disparadores:disparadores>

    <deriva />
</recorrido>
```

- **seguimiento**: misiones de seguimiento de gradientes o de rangos de diversas magnitudes con el área como límite del seguimiento. Dentro de este elemento puede darse uno de estos dos elementos, según como se quiere que se configure el seguimiento:

  - **rango**: se especifica la magnitud y el rango de medidas entre las que debe hacer el seguimiento de la magnitud. Los atributos son:

```
<seguimiento>
   <rango medida="temperatura" minimo="10" maximo="30" unidad="ºC" />
</seguimiento>
```

  - **función**: seguir la medida de acuerdo con una función, como puede ser el gradiente. Los atributos son:

```
<seguimiento>
   <funcion medida="temperatura" modo="gradiente" submodo="ascendente"
/>
</seguimiento>
```

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Seguimiento		Elemento en el que se definirá la forma de recorrer el área. El tipo seguimiento podrá definirse como un seguimiento por rangos o por funciones.		
	Rango	Especifica que se va a hacer un seguimiento en función de un rango de medidas.		X Tiene que haber un elemento Rango o un elemento función en el elemento seguimiento. Seria un o exclusivo, tiene que haber uno y sólo uno.
			medida: la magnitud a seguir.	X
			minimo: valor mínimo que debe seguir.	X
			maximo: valor máximo que debe	X

			seguir.	
			unidad: unidad en que se expresan mínimo y máximo.	X
	Funcion	Se va a seguir la medida según una función.		X Tiene que haber un elemento Rango o un elemento función en el elemento seguimiento. Seria un o exclusivo, tiene que haber uno y sólo uno.
			medida: la magnitud a seguir.	X
			modo: la función de seguimiento de la magnitud.	X
			submodo: parámetro que configura el modo.	X

Tabla 11: Elemento Seguimiento

- **transectos**: Recorrer al área por transectos, es decir, de acuerdo a un patrón de recorrido para cubrir el área. Este recorrido se subdivide conceptualmente en varios transectos lineales, sobre los que se pueden definir acciones concretas.

```
<transectos modo="zigzag" cantidad="10">
  <tiempo valor="5" unidad="minuto" />
  <profundidad valor="10" unidad="m" />
  <angulo valor="20" unidad="grados" />
</transectos>
```

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Transectos		En este tipo de recorrido se subdividirá el área en transectos dependiendo el modo de subdivisión que hay establecido el usuario.		
			modo: Modo de recorrer el área.. se puede especificar, por ejemplo, el modo **zigzag** o el modo **espiral**.	X

			cantidad: Cantidad de transectos en los que se subdividirá el área.	X
			ciclos: Por defecto vale 1. Indica el número de veces que se recorre por completo el área.	
	tiempo	Tiempo en el que se deben recorrer los transectos.		
			valor: Indica tiempo en que debe recorrerse todo el transecto especificado.	X
			unidad: unidad en que se expresa el tiempo.	X
	Profundidad	tiempo en recorrer el transecto.		
			valor: Profundidad a la que debe realizarse el transecto, lo cual permitirá que los recorridos no sean en un plano, sino que al mismo tiempo el AUV se sumerja o emerja. Para no definir la profundidad para cada transecto, se plantea un modelo simplificado en el que se indicará el incremento o decremento de la profundidad..	X
			unidad: unidad en la que se expresa la profundidad.	X
	angulo	tiempo en recorrer el transecto.		
			valor: indica el ángulo inicial que se toma para hacer el recorrido (la orientación del barrido por transectos).	X
			unidad: unidad en la que se expresa el ángulo.	X

Tabla 12: Elemento Transectos

Véase que aunque los tres (tiempo, profundidad y ángulo) son opcionales, si se especifican, éstos deben seguir el orden en que se han mostrado.

4.7.6.2.3 Zona Prohibida

Toda área se define mediante una lista de dos tipos de elementos consecutivos.

```
<zonaProhibida id="30" nombre="triangulo las bermudas">
   <vertice id="1" x="40" y="20" z="10" unidad="m" />
   <vertice id="2" x="20" y="10" z="20" unidad="m" />
   <vertice id="3" x="30" y="20" z="10" unidad="m" />
   <profundidad minima="10" maxima="100" unidad="m" />
</zonaProhibida>
```

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
ZonaProhibida		Elemento en el que se definirá la zona prohibida que debe evitar el submarino.		
			id: Identificador de la zona prohibida	X
			nombre: nombre de la zona prohibida	X
	Vertice	Definirá uno de los vértices que van a componer la cara superior del prisma que va a localizar la zona prohibida para el submarino.		X Debe haber un mínimo de tres vértices, y no hay límite superior.
			id: identificador del vértice	X
			x: Coordenada X.	X
			y: Coordenada Y	X
			z: Coordenada Z	X
			unidad: Unidad de medida en la que se	X

			expresa la posición.	
	Profundidad	Profundidad máxima y mínima dentro de un área.		
			minima: profundidad mínima	X
			maxima: profundidad máxima	X
			unidad: Unidad de medida en la que se expresa la profundidad.	X

Tabla 13: Elemento Zona Prohibida

4.7.7 Plan de comunicaciones

4.7.7.1 Semántica

Este plan se encarga de definir todos los datos que pueden ser comunicados a un destinatario externo al AUV y el momento o condiciones bajo el qué son comunicados.

Permite la emisión o recepción tanto de datos puntuales como de ficheros o simples avisos. En términos propios de los planes, se permitirá enviar y recibir medidas o datos. Las medidas serán datos puntuales o atómicos (muestras de una medida), mientras que los datos serán un conjunto de muestra de la medida homónima, contenidos o almacenados en un fichero.

Se define principalmente en **./esquemas/pdc.xsd** , y en el fichero **./esquemas/lib/disparadores.xsd** donde se definen los disparadores de la forma comentada en la sección "**Aspectos generales de los ficheros XML de los distintos planes**".

4.7.7.2 Descripción

Este apartado se centrará en explicar los elementos acciones de cada tarea (tiene uno y solo un elemento acciones), ya que es en este elemento en el que se diferencian los planes que se guían por la estructura tarea → (disparadores, acciones).

El elemento acciones contiene una lista, de cero a muchos, de cuatro tipos de elementos semánticamente distintos, pero muy similares en su estructura. Los cuatro tipos de elementos que se pueden definir en el elemento acciones son:

- **EnviarMedida**: Especificará una comunicación realizada con el monitorizador de la misión para recibir datos de una medida en concreto.

Elemento	Descripción	Atributos	Obligatorio
EnviarMedida	Acción de comunicación que define el envío de una medida en concreto.		
		id: Identificador	X
		medida: medida que se desea enviar en esta acción.	X
		destinatario: a quien va destinado el dato. Se especificará normalmente mediante la ip del destinatario, o también puede hacerse mediante ip:puerto. Se pueden especificar varios destinatarios (separados por coma), o ninguno, lo que significa un broadcast. Por tanto es un campo opcional	X El atributo debe aparecer en el fichero XML, aunque sea con valor vacío "", para hacer un broadcast.

Tabla 14: Elemento EnviarMedida

- **EnviarDato**: similar al anterior, pero lo que se manda es un fichero (que queda oculto/transparente al usuario y el plan, pues simplemente se indicará el dato). Por tanto, en vez del atributo medida se utiliza el atributo dato.

Elemento	Descripción	Atributos	Obligatorio
EnviarDato	Acción de comunicación que define el envío de un fichero de datos conteniendo la medida especificada.		
		id: Identificador	X
		dato: medida que se desea enviar en esta acción.	X
		destinatario: a quien va destinado el dato. Se especificará normalmente mediante la ip del destinatario, o también puede hacerse mediante ip:puerto. Se pueden	X El atributo debe aparecer en el fichero XML, aunque sea con valor vacío "", para hacer un

		especificar varios destinatarios (separados por coma), o ninguno, lo que significa un broadcast. Por tanto es un campo opcional	broadcast.

Tabla 15: Elemento EnviarDato

- **RecibirMedida**: permite recibir algún dato de una entidad externa, normalmente el controlador de la misión.

Elemento	Descripción	Atributos	Obligatorio
RecibirMedida	Acción de comunicación en la que el submarino se tiene que preparar para la recepción de datos.		
		id: Identificador	X
		medida: la medida a recibir (se recibirán sus muestras).	X
		fuente: se pone la ip de la entidad externa que de la que proviene el dato.	X El atributo debe aparecer en el fichero XML, aunque sea con valor vacío "", para hacer un broadcast.

Tabla 16: Elemento RecibirMedida

- **RecibirDato**: similar al anterior, pero lo que se recibe es un fichero (que queda oculto/transparente al usuario y el plan, pues simplemente se indicará el dato). Por tanto, en vez del atributo medida se utiliza el atributo dato.

Elemento	Descripción	Atributos	Obligatorio
RecibirDato	Acción de comunicación en la que el submarino envía un dato en concreto.		
		id: Identificador	X
		dato: la medida a recibir.	X

		fuente: se pone la ip de la entidad externa que de la que proviene el dato.	X El atributo debe aparecer en el fichero XML, aunque sea con valor vacío “”.

Tabla 17: Elemento RecibirDato

No se exige un orden concreto, ni tienen que aparecer los cuatro tipos de elementos, solo hay que tener en cuenta que todos los elementos que aparezcan dentro del mismo campo de acciones deben tener identificadores distintos.

En esta línea, un elemento acciones puede no tener ninguno de los cuatro elementos, lo que se interpreta como una tarea de depuración o comprobación del sistema. Puede ser útil simplemente para poner en línea al submarino con el operador que está monitorizando la misión.

Un ejemplo de una tarea de este tipo lo podemos ver a continuación:

```
<tarea id="3" nombre="tarea3">
   <disparadores:disparadores>
         ....
   </disparadores:disparadores>
   <acciones>
      <enviarMedida                id="1"                medida="nivelRealizacionMision"
destinatario="172.172.172.1" />
      <enviarDato  id="1" dato="  temperatura exterior" destinatario="172.172.172.2" />
      <recibirMedida  id="1" medida="posicion" fuente="172.172.172.1" />
      <recibirDato  id="2" dato="batimetria" fuente="172.172.172.1" />
   </acciones>
   <periodoInhibicion valor="10" unidad="minuto"/>
</tarea>
```

Por último, todos los elementos de tipo tarea están embebidos en el elemento raíz, de tipo planDeComunicacion, que presenta:

- un id
- un nombre
- referencia a los esquemas xsd utilizados.

```
-                  <planDeComunicacion                id="1"                nombre="PdC"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:disparadores="http://www.disparadores.com"      xmlns="http://www.pdc.com"
xsi:schemaLocation="http://www.pdc.com  ../../esquemas/pdc.xsd">
```

4.7.8 Plan de medidas

4.7.8.1 Semántica

Este plan define todas las medidas que deben ser tomadas desde los sensores de instrumentación y cuando, o bajo que condiciones tienen que ser tomadas. Por ello, representa la misión en sí y es de gran interés cuando el AUV es utilizado para muestreo de parámetros físico-químicos del mar.

La forma de definir las medidas permite distintos niveles de abstracción:

- Especificar sólo la medida (u observable) a medir, sin especificar sensor, con lo que la lógica del AUV resuelve el o los dispositivos que deben servir la medida. También se indicará siempre la frecuencia y resolución de muestreo deseada.

- Especificar además de la medida, el sensor o los sensores desde los que se quieren tomar las medidas.

Además de especificar la medida y sensor, se especifica para cada sensor una configuración concreta. Indicará, por ejemplo, frecuencia con la que se debe medir, resolución, etc. Un sensor podrá tener múltiples paquetes de configuración asociados a éste, y el usuario seleccionará uno para realizar la medición según la frecuencia y resolución que necesite.

Se define principalmente en **./esquemas/pdm.xsd** , y en el fichero **./esquemas/lib/disparadores.xsd** donde se definen los disparadores de la forma comentada en los aspectos generales de los planes(Ver sección 4.7.5.1).

A parte, se utiliza el fichero **./esquemas/lib/equipamiento.xsd**, para la definición de los sensores.

4.7.8.2 Descripción

Se centrará este apartado en explicar los elementos acciones de cada tarea. En este caso, para el plan de medidas, solo tenemos un tipo de acciones para el elemento **acciones**.

El elemento acciones que tenemos en este tipo de plan es el elemento **medir**. Cada elemento **acciones** puede tener ninguno o varios de estos elementos.

```
<tarea id="2" nombre="MedicionComienzoTiempo">
<disparadores:disparadores>
...
</disparadores:disparadores>
<acciones>
<medir id="1" medida="temperatura exterior" muestras="5">
  <frecuencia  valor="1" unidad="Hz" />
  <resolucion  valor="1" unidad="ºC" />
  <sensor  id="1" nombre="brujulaTCM2" />
</medir>
<medir id="2" medida="salinidad">
  <frecuencia  valor="3" unidad="Hz" />
  <resolucion  valor="0.5" unidad="psu" />
</medir>
</acciones>
<periodoInhibicion .... />
  </tarea>
```

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Medir		Acción del plan de medidas en el que se especifica cómo se quiere realizar las medidas.		
			id: identificador de la acción.	X
			medida: medida que se desea tomar.	X
			muestras: número de muestras que se deben tomar. Se puede especificar un número positivo o los caracteres INF para indicar que se tomen muestras indefinidamente.	Opcional. Si se omite el número de muestras a tomar, se tomará como si se especificara el INF (número ilimitado de muestras).
	Frecuencia	Frecuencia con la que se tomarán las medidas.		
			valor: Valor de la frecuencia	X
			unidad: unidad de medida en al que se especifica la frecuencia	X
	Resolución	La resolución con la que se van a tomar los datos.		
			valor: resolución	X
			unidad: unidad de medida en la que se especifica la resolución.	X
	Sensor	Sensor o sensores que se van a poder utilizar para realizar la medida. Ver la tabla “sensor” para más información de este elemento.		Pueden ser o 0, o un número indeterminado de sensores los que se especifiquen en este elemento.

Tabla 18: Elemento medir

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
Sensor		Indica que sensores concretos deben realizar la medida. Si no se especifica, el sistema de control del AUV resuelve el sensor desde el que se realiza la medida.		
			id: identificador de la acción.	X
			nombre: nombre del sensor.	X
	Configuración	Configuración que se le asignará al sensor.		Puede no haber ninguna aparición de este elemento o haber un número indeterminado de ellos.
			configuración: Por ahora, este atributo contendrá el nombre de un fichero que contendrá la configuración que se le asignará al sensor.	

Tabla 19: Elemento Sensor

Cuando se definen los sensores que deberán tomar las medidas, el AUV tendrá que coger éstos para que realicen la toma de muestras. En caso contrario, en el que no se han definido sensores, será el AUV el que tendrá que tomar la decisión de qué sensor utilizar para esa medida.

Por último, todos los elementos de tipo tarea están embebidos en el elemento raíz, de tipo planDeMedicion, que presenta:

- un id
- un nombre
- referencia a los esquemas xsd utilizados.

```
<planDeMedicion id="1" nombre="PdM" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"                    xmlns:disparadores="http://www.disparadores.com"
xmlns:equipamiento="http://www.equipamiento.com"    xmlns="http://www.pdm.com"
xsi:schemaLocation="http://www.pdm.com ../../esquemas/pdm.xsd">
```

4.7.9 Plan de almacenamiento

4.7.9.1 Semántica

Este plan especifica que datos almacenar. Respecto a los datos objeto de este plan, se encuentran medidas del AUV: es decir, toda medida que tome el sistema, tanto las hechas por sensores de misión, instrumentación, como sensores internos, así como las de sensores virtuales que proporcionarán medidas internas (e.g. posición estimada, velocidad estimada, etc.).

En el plan de almacenamiento sólo se indicará la medida a almacenar, pero bajo ningún concepto se forzará a medirla. De este modo, los planes de almacenamiento y de medidas serán ortogonales o independientes. Así, se puede indicar que se almacene una medida, pero que no se mida (no se indicará en el plan de medidas, aunque pudiera forzarse su medición con comandos remotos). Esto permite cubrir los casos en que se quiere almacenar sin medir, medir sin almacenar, y medir y almacenar. El subsistema de almacenamiento simplemente recibirá muestras y si éstas son de una medida a almacenar, se almacenará.

Se define principalmente en **./esquemas/pda.xsd** , y en el fichero **./esquemas/lib/disparadores.xsd** donde se definen los disparadores de la forma comentada en la sección "Aspectos generales de los ficheros XML de los distintos planes".

4.7.9.2 Descripción

El elemento acciones de este plan contiene una lista de elementos de tipo almacenar, donde cada elemento de tipo almacenar especifica una medida (u observable) que se quiere almacenar. Se pueden especificar de cero a muchos elementos almacenar.

```
<tarea id="2" nombre="AlmacenamientoPosicion">
  <disparadores:disparadores>
      ....
  </disparadores:disparadores>
  <acciones>
    <almacenar  id="1" medida="posicion" muestras="5" />

    <almacenar  id="2" medida="orientacion" muestras="4" />

    <almacenar id="3" medida="temperatura interna" muestras="INF"/>

    <almacenar id="4" medida="velocidad lineal"/>

  </acciones>
  <periodoInhibicion  .... />
</tarea>
```

En la siguiente tabla se presenta el elemento **almacenar**.

Elemento	Compuesto de	Descripción	Atributos	Obligatorio

Almacenar		Acción que se encargará de definir los datos a almacenar.		
			id: esto implica que todos los elementos de tipo almacenar dentro de un mismo elemento acciones deben tener distinto ID.	X
			medida: es la medida que se va a almacenar. El dato, por tanto, coincide en nombre con la medida. De este modo, cuando se quiera hacer referencia a un dato se usará el mismo nombre de la medida, pero el dato hará referencia a un dato del inventario, almacenado físicamente en un fichero. Un ejemplo, es el caso de los comandos del plan de comunicación, que podrá enviar medidas o datos.	X
			muestras: indica cuantas muestras son almacenadas. Es un número positivo (incluido cero, en cuyo caso es como si no se pudiera almacenar el elemento) o INF (infinito, para que almacene cualquier cantidad de muestras). Es un atributo opcional, si se omite es igual que poner INF.	

Tabla 20: Elemento Almacenar

Por último, todos los elementos de tipo tarea están embebidos en el elemento raíz, de tipo planDeAlmacenamiento, que presenta:

- un id
- un nombre
- referencia a los esquemas xsd utilizados.

```
<planDeAlmacenamiento id="1" nombre="pda"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:disparadores="http://www.disparadores.com"
xmlns:equipamiento="http://www.equipamiento.com" xmlns="http://www.pda.com"
```

4.7.10 Plan de supervisión

4.7.10.1 Semántica

Este plan describe la forma de actuar del AUV ante excepciones, y por tanto, los planes de contingencia que debe ejecutar el mismo.

También describe ante ciertas condiciones u intervalos de cualquier tipo comandos internos que se deberá realizar para cambiar el modo de actuar del AUV (cambiar configuración de sensores, dispositivos, planes, etc).

Entre los comandos, hay un comando muy especial, que es el comando CambiarVariableMision, que posibilita cambiar:

- las variables declaradas en la misión (en las tablas de parámetros especificadas en el fichero de la misión <mision>.xml)
- cualquier otro atributo, de cualquier otro plan, mediante una referencia con xpath.

Más adelante explicaremos como es posible la modificación de estos atributos y variables en la misión.

Se define principalmente en **./esquemas/pds.xsd** , y en el fichero **./esquemas/lib/disparadores.xsd** donde se definen los disparadores de la forma comentada en la sección "Aspectos generales de los ficheros XML de los distintos planes".

4.7.10.2 Descripción

Este apartado se centrará en explicar los elementos acciones de cada tarea. Este plan podrá tener dos tipos de acciones: **ejecutarPlan** y **ejecutarComando**.

El elemento acciones tendrá una lista de cero a muchos elementos de estas dos clases de acciones. A continuación se muestra un ejemplo de un elemento "tarea" de este plan.

```
<tarea id="1" nombre="tarea1">
  <disparadores:disparadores>

  </disparadores:disparadores>
  <acciones>

    <ejecutarPlan id="1" plan="PonerAUVenSitioSeguro">
      <parametro  id="1" valor="100" />
      <parametro  id="2" valor="0.5" />
    </ejecutarPlan>

    <ejecutarComando id="1" comando="CambiarConfiguracion">
      <parametro  id="1" valor="termometro" />
      <parametro  id="2" valor="configuracionBajoConsumo" />
    </ejecutarComando>
```

```
      <ejecutarPlan id="2" plan="SalvarDatos">
        <parametro  id="1" valor="100" />
        <parametro  id="2" valor="0.5" />
      </ejecutarPlan>

    </acciones>
    <periodoInhibicion  ... />
</tarea>
```

- **ejecutarPlan**: El elemento **ejecutarPlan** permitirá al AUV ejecutar un plan completo.

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
ejecutarPlan		Acción que cambiará el plan que se está ejecutando actualmente por otro especificado en este elemento.		Puede haber de 0 a muchos parámetros.
			id: identificador	X
			plan: Nombre del plan a ejecutar.	X
	parametro	Elemento que especificará un parámetro de entrada para el plan.		Puede haber de 0 a muchos parámetros.
			id: identificador	X
			valor: valor del parámetro	X

Tabla 21: Elemento ejecutarPlan

- **ejecutarComando**: permite al AUV ejecutar comandos concretos ante ciertas situaciones. La diferencia con los planes es que los comandos son mucho más simples y directos. Un plan puede implicar ejecutar muchos comandos. Sus atributos son:

Elemento	Compuesto de	Descripción	Atributos	Obligatorio
ejecutarComando		Acción que ejecutará un comando con los parámetros especificados.		Puede haber de 0 a muchos parámetros.

			id: identificador	X
			comando: Nombre del comando a ejecutar.	X
	parametro	Elemento que especificará un parámetro de entrada para el plan.		Puede haber de 0 a muchos parámetros.
			id: identificador	X
			valor: valor del parámetro	X

Tabla 22: Elemento ejecutarComando

De forma similar al plan, puede contener de cero a muchos elementos de tipo parámetro, para especificar parámetros de entrada al comando, y todos los del mismo elemento comando deben tener distinto identificador.

No se exige un orden concreto entre planes y comandos, ni tienen que aparecer los dos tipos de elementos, solo hay que tener en cuenta que todos los elementos que aparezcan dentro del mismo campo de acciones deben tener identificadores distintos.

En esta línea, un elemento acciones puede no tener ninguno de los dos elementos, lo que se interpreta como una tarea de depuración o comprobación del sistema.

Por último, todos los elementos de tipo tarea están embebidos en el elemento raíz, de tipo planDeSupervision, que presenta:

- un id
- un nombre
- referencia a los esquemas xsd utilizados.

```
<planDeSupervision id="1" nombre="PdS"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:disparadores="http://www.disparadores.com" xmlns="http://www.pds.com"
xsi:schemaLocation="http://www.pds.com ../../esquemas/pds.xsd">
```

4.7.11 Modificación de Planes en Tiempo de Ejecución

Una de las características de los planes, es que puede ser modificable cualquier plan de forma poco costosa. Esto permite tener un AUV con capacidad de relacionarse con su entorno y perfeccionar su plan adaptándose a éste lo mejor posible.

Para este fin, se introduce en todo plan dos formas de acceder a un atributo modificable en tiempo de ejecución:

- **De forma directa**: se indica la "dirección" del atributo, señalando todos los elementos antecesores hasta llegar al elemento que contiene al atributo, usando xpath.

- Con **parámetros de la misión**: se definen en **./tablaparametros**, con ficheros xml, conjuntos de variables de estado de la misión, donde cada variable de estado, con un nombre, se relaciona con uno o más atributos, de uno o varios planes. Esto permite que modificando solo el valor del parámetro, automáticamente se modifiquen todos aquellos atributos a los que se asocia a lo largo de los planes.

La modificación del valor de cualquier atributo, de las dos maneras explicadas, las puede llevar a cabo:

- El **plan de supervisión**, mediante un comando especial de su acción **ejecutarComando** que permite cambiar ciertas variables, y por tanto cambiar los planes selectivamente.

- Comandos enviados por el controlador de la misión al AUV (telecomando).

4.7.11.1 Direccionamiento de atributos: xpath

Para esto, se utiliza el formato **xpath[W3C-XPATH]**: herramienta de XML para referenciar elementos y atributos de un determinado fichero.

La motivación de usar este formato es su utilidad en el **Planificador de la Misión**, pues permite usar consultas **XQuery[W3C-XQUERY]** (muy similares a bases de datos) para rellenar un fichero XML.

Veámoslo con un ejemplo:

```
"/planDeMedicion[@id='7']/tarea[@id='5']/@nombre"
```

Como vemos comenzamos por el elemento raíz del fichero (que del fichero **pdm** es **planDeMedicion**). Especificar el **id** en realidad no es útil, pues un fichero xml contiene un único **planDeMedicion**, pero permitirá referenciar planes habidos en otros ficheros distintos al presente.

Una vez especificado el **plan**, y el **id** (para identificar el fichero xml exacto al que se refiere el cambio), a partir de aquí navegamos por el fichero xml, de manera que

cada vez que nos metemos en un elemento interno, ponemos una / seguida del nombre del elemento, y, si existen varios elementos del mismo tipo, especificamos entre corchetes justo el **id** que tiene que tener el elemento.

Así, hasta que lleguemos al elemento final que queremos modificar. Llegados a el, especificamos el atributo a modificar antecediendo su nombre con una @.

En el ejemplo, queremos modificar el atributo **nombre** de la **tarea** número 5 del **plan de medición** número **7**.

El primer elemento no solo tiene que ser un **plan**, también se puede especificar como elemento raíz /**Mision[@id='3']…**, si se desea modificar el fichero de misión número 3.

4.7.11.2 Ejemplo: Modificación desde el plan de supervisión.

Del set de comandos ejecutables en la acción ejecutarComando, de las acciones del plan de supervisión, se encuentra un comando, el CambiarVariableMision, que permite la modificación de atributos de los planes o de la misión. El primer parámetro del comando es la variable o atributo a modificar, y el segundo el valor. La sintaxis sería tal que así:

```
<tarea id="7" nombre="tarea6">

   <disparadores:disparadores>

      <disparadores:condicion id="1" medida="waypoint"
            operador="eq" valor="10" unidad="waypoint" />

      <disparadores:intervalo id="2" medida="temperatura"
            inicio="1" fin="100" periodo="100" unidad="ºC" />

  </disparadores:disparadores>


   <acciones>

      <ejecutarComando id="3" comando="CambiarVariableMision">
         <parametro id="1" valor="@VelocidadCrucero" />
         <parametro id="2" valor="200" />
      </ejecutarComando>

      <ejecutarComando id="9" comando="CambiarVariableMision">
         <parametro id="1" valor="/planDeMedicion[@id='7']/tarea[@id='5']/@nombre" />
         <parametro id="2" valor="nuevoNombre" />
      </ejecutarComando>

      <ejecutarComando id="4" comando="emerger" />

   </acciones>

   <periodoInhibicion valor="10" unidad="minuto" />

</tarea>
```

En el ejemplo vemos que cuando se cumplan los disparadores de la tarea 7 del plan de supervisión, se procederá a:

- Modificar el parámetro de la misión VelocidadCrucero (y todos los atributos a los que referencia) a valor 200. Para señalar parámetros de la misión se utiliza una arroba antecediendo al nombre de la variable.

  Si observamos la tabla de parámetros donde se define VelocidadCrucero como variable de estado o parámetro de la misión, nos encontramos con esta definición:

```
<parametro nombre="VelocidadCrucero" valor="4">

<parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='12']/velocidad/posicion/@x" />

<parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='12']/velocidad/posicion/@y" />

<parametros:elemento
nombre="/planDeNavegacion[@id='3']/ruta[@id='1']/transecto[@id='21']/velocidad/posicion/@z" />

</parametro>
```

  Por lo tanto, cambiar la variable VelocidadCrucero implica modificar en el plan de navegación 3, en la ruta 1, en el transecto 12, en el elemento velocidad, en el elemento dentro de velocidad posición el atributo x e y, y en el transecto 21 de la misma ruta y plan de navegación, en el elemento velocidad y dentro de este el elemento posición, el atributo z.

- Modificar el nombre de la tarea 5 del plan de medición, por "nuevonombre".

- Emerger.

4.7.12 Diseño del AUV.

En principio habíamos definido una serie de ficheros XML que desarrollaría el usuario y que definiría el AUV. Estos ficheros se enviarían al submarino junto con la misión al completo para que el submarino verificara si realmente posee las capacidades que se especifican en el fichero de definición del AUV. En caso contrario, significaría que el usuario habría realizado unas misiones en el planificador de misiones que podrían ser no aptas para este submarino en cuestión.

Finalmente, se decidió que la definición del AUV en ficheros XML no se tendría en cuenta, ya que el submarino puede ser capaz de verificar la misión que se le ha enviado antes de comenzar la misión. No sería necesario proporcionarle el fichero XML con la definición del AUV para ello.

Aún así, se va a explicar brevemente la definición del AUV en ficheros XML, ya que será útil para que el planificador de la misión pueda almacenar y cargar tanto los datos del submarino en general como de cada uno de sus componentes en particular,

aunque no se va a entrar tan en detalle como con los planes de la misión, los cuales eran mucho más relevantes para este proyecto.

Necesitábamos una definición del submarino que fuera completa, y además, flexible. Vamos a definir, no sólo el submarino, sino cada uno de sus componentes de forma independiente. Esto facilitará al usuario la definición de futuros submarinos, los cuales podrían reutilizar componentes ya usados para otros submarinos ya definidos. Desde los sistemas del submarino, hasta cada uno de los sensores que utiliza, están definidos en distintos ficheros XML y almacenados en distintos subdirectorios.

El AUV estará compuesto por 6 subsistemas, cada uno de ellos se encargarán de un conjunto de tareas en concreto. Los 6 subsistemas que se pueden definir para el submarino son:

- **Sistema de impulsión**: Lo conformarán todos los actuadores del submarino que lo capacitan para desplazarse por el entorno, como pueden ser los motores, que serían los impulsores del AUV, o los mandos, que servirían para dirigirlo.

- **Sistema de alimentación**: Los componentes de este sistema son los que se encargarán de proporcionar la energía necesaria a los demás componentes del submarino. Ejemplos de componentes de este sistema pueden ser baterías o pequeños paneles solares.

- **Sistema de comunicación**: Es el encargado de habilitar las comunicaciones con un software monitorizador de la misión, o con otros submarinos en caso de que sea necesario. Este sistema comprenderá desde componentes para la comunicación con el software monitorizador, como wi-fi o comunicadores vía satélite, hasta sistemas para la comunicación entre submarinos mientras estén sumergidos, como modems acústicos.

- **Sistema sensorial**: Se encargará de monitorizar tanto los parámetros físico-químicos externos como el estado interno del submarino. Se dividirá en tres tipos de sensores distintos: los sensores de la misión (los que se utilizarán para tomar datos interesantes para los científicos durante la misión, como temperatura, salinidad del agua etc), los sensores internos, que se utilizarán para comprobar el estado interno del submarino, como el nivel de estancamiento o la temperatura de sus componentes durante el funcionamiento de éstos, y los sensores denominados instrumentación, que guiarán al submarino, como pueden ser brújulas y GPS.

- **Sistema actuador**: será el encargado de hacer interaccionar al submarino con su entorno para, por ejemplo, extraer muestras o iluminar el entorno, como las luminarias.

- **Sistema de procesamiento**: será el "cerebro" del submarino. Estará compuesto de memorias, procesadores etc.

En la siguiente tabla se puede ver los ficheros que se pueden utilizar para definir un AUV. Más adelante, en el análisis de la interfaz del AUV, se explicará con más detalle cada uno de los componentes del AUV.

Ficheros XML			Descripción
<auv>.xml			Será el que contendrá la definición principal del submarino. Contendrá un listado de todos los ficheros que conforman la definición del AUV.
	./impulsion/ <impulsion>.xml		Definirá el sistema de impulsión del submarino. En este fichero se listarán los ficheros que definen los distintos motores y mandos que conforman el sistema de impulsión del AUV.
		./impulsion/motores/ <motor>.xml	Se especifican las dimensiones y el peso del motor.
		./impulsion/mandos/ <mandos>.xml	Se especifican las dimensiones y el peso del mando.
	./alimentacion/ <alimentacion>.xml		En este fichero se encontrará la lista de ficheros que definen las distintas baterías del submarino.
		./alimentacion/baterias/ <bateria>.xml	Cada uno de estos ficheros especificará un componente que proporciona energía a los sistemas eléctricos del AUV. Se definirán con un peso, una carga y unas dimensiones.
	./comunicacion/ <comunicacion>.xml		Al igual que los demás ficheros que definen sistemas, contendrá una lista de ficheros que definen el sistema de comunicación del

			submarino.
		./comunicacion/dispositivos/ <dispositivo>.xml	Define cada uno de los dispositivos de comunicación. En este fichero se puede especificar su peso, consumo, ancho de banda y dimensiones.
	./sensorial/ <sensorial>.xml		Sistema sensorial del AUV, con una lista de ficheros que definen los sensores de los que está compuesto. Habrá tres tipos, la instrumentación, los sensores internos y los sensores de la misión.
		./sensorial/instrumentación/ <instrumentacion>.xml	Dispositivos que sirven para orientarse navegar. Por ejemplo, una brújula. Se definirá una lista de las medidas que es capaz de tomar este dispositivo, frecuencia, peso, consumo, dimensiones y la ubicación del sensor en el submarino.
		./sensorial/interno/ <sensor>.xml	Sensores internos que medirán el estado del submarino. Se definirán, al igual que con la instrumentación, las medidas que es capaz de tomar (por ejemplo, temperatura), frecuencia, peso, consumo, ubicación y dimensiones.
		./sensorial/mision/ <sensor>.xml	Sensores de la misión. Son los sensores que se utilizarán durante la misión para tomar los datos que necesita la misión que se va a llevar a cabo. Al igual que en los dos casos anteriores, se especifica una lista de medidas, al frecuencia con la que es capaz de tomar los datos, el peso del sensor,

			el consumo, la ubicación y las dimensiones.
	./actuador/ <actuador>.xml		Contendrá un listado con todos los ficheros que definen los elementos que componen el sistema actuador.
		./actuador/actuadores/ <actuador>.xml	Se definirá un actuador con un peso, un consumo, una ubicación y unas dimensiones.
	./procesamiento/ <procesamiento>.xml		Contendrá el listado de ficheros que definen los elementos que compondrán el sistema de procesamiento del submarino. Habrá tres tipos distintos de componentes del sistema de procesamiento: CPUs, memorias y tarjetas.
		./procesamiento/cpus/ <cpu>.xml	Definirá las CPU que conforman el sistema de procesamiento del submarino. Se definirá el peso la frecuencia y el consumo del procesador.
		./procesamiento/memorias/ <memoria>.xml	Definirá los componentes de almacenamiento necesarios. Se especificará el peso, la frecuencia, el consumo y la capacidad.
		./procesamiento/tarjetas/ <tarjeta>.xml	Definirá las tarjetas que formarán parte del sistema de procesamiento. Se especificará en este fichero el peso, la frecuencia y el consumo de la tarjeta.

Tabla 23: Definición XML de un AUV

Se puede ver como cada componente se define en un fichero distinto. Hemos planteado la definición de un AUV de esta forma ya que nos facilita la reutilización de

esos mismos componentes en otros AUVs. Lo normal sería que varios AUVs tengan varios componentes, o incluso sistemas completos similares entre sí, por lo que nos ahorraríamos volverlos a especificar cada vez que queramos definir un nuevo AUV.

Capítulo 5. Análisis del sistema.

En este capítulo se analizarán las distintas interfaces que debe presentar el software de planificación y control para que un usuario de este sistema tenga la posibilidad de definir cualquier tipo de misión que cumpla los requisitos especificados en el capítulo anterior.

También se analizarán algunos algoritmos que servirán para definir planes de navegación sobre un mapa, y planificar las rutas más cortas entre dos puntos.

5.1 Análisis

La interfaz es una parte muy importante en este proyecto, ya que debe permitir al usuario la definición de misiones completas, con cada uno de sus planes y cada uno de ellos a su vez de la forma más detallada posible.

Como se ha visto, los planes que se definen para el submarino pueden ser muy potentes y flexibles. Éstas características no se deben perder al generar las interfaces que los definan. Debido a ello, como veremos, las interfaces quizás no son todo lo ameno que deberían, pero permiten especificar todos los detalles de los planes al usuario.

Por ello, en este capítulo analizaremos cada una de las interfaces necesarias para definir la misión del submarino para que sean lo más completas posibles. Aunque no sean demasiadas amenas, se ha intentado que sean lo más intuitivas posible, y que no sean demasiado tediosas para el usuario.

El tipo de usuario que va a utilizar esta interfaz puede ser científicos medioambientales, biólogos etc. Este tipo de usuario no tiene por qué tener amplios conocimientos de informática.

La interfaz en cuestión, debe poder permitir al usuario usar toda la potencialidad del software de una forma sencilla y fácil de memorizar para poder especificar y monitorizar la misión.

Las partes en las que se podría dividirá este software son las siguientes:

- **Especificación de los planes**: Esta parte del software se encargará de definir cada uno de los planes que se han visto en el apartado anterior (plan de navegación, de medición, de comunicación, de almacenamiento y de supervisión).

- **Especificación de los componentes del AUV**: Éste es el módulo encargado de la definición de los diferentes componentes del AUV. Permitirá definir nuevos componentes para el AUV, sus parámetros de configuración y su lista configuraciones.

- **Especificación de AUV**: Esta parte de la interfaz servirá para definir nuevos AUVs, con todas sus características esenciales, y los componentes que va a contener este AUV para una misión en concreto.

- **Definición de la misión**: Finalmente es necesario disponer de utilidades para componer la misión a partir de elementos previamente definidos, incluyendo planes, un submarino y una instrumentación para realizar esa ruta. Posteriormente se procederá a la validación de la misión, comprobando que todo encaja correctamente, los planes con el submarino y la instrumentación escogida y se crearán los ficheros de misión preparados para ser emitidos al submarino.

- **Monitorización y Control**: Control del desarrollo de las misiones por parte del usuario. Se validará la misión y se comprobará que los planes que la conforman son compatibles entre sí. Además se mostrarán los datos que emita el vehículo que esté siendo monitorizado.

Éstas son las principales utilidades, especificadas de una forma general, que podría tener este software reflejado en su interfaz. Quizás es más correcto definir la funcionalidad de forma distinta. Podemos tener dos funcionalidades principales, consistentes en la parte que podemos denominar "de **definición de la misión**" y la parte de "**control del desarrollo de la misión**". El esquema sería el que se muestra en la Ilustración 9.

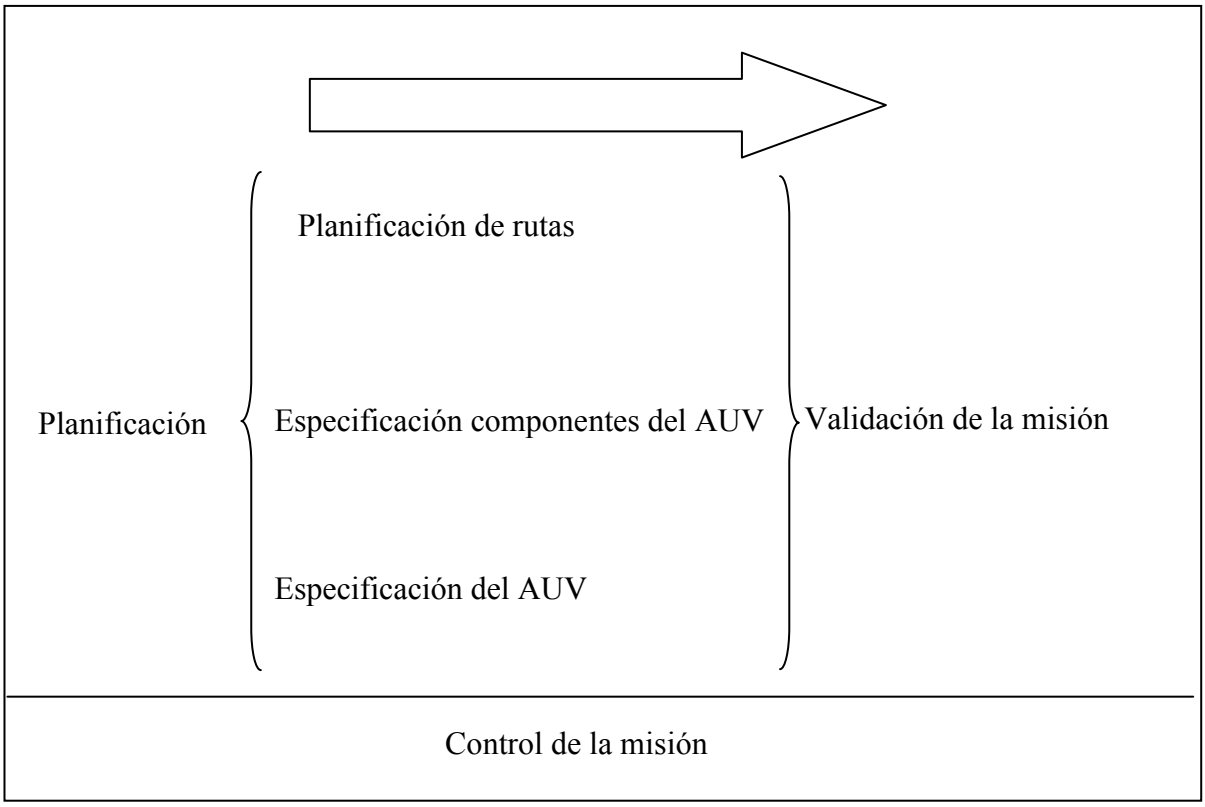


Ilustración 9: Esquema el sistema

La fase de definición incluye una interfaz que permitirá al usuario definir los distintos planes, submarinos, y sus componentes mientras que la de control corresponde a una interfaz en la que el usuario monitorizará la misión del AUV.

La planificación de la misión se puede seguir dividiendo en cada uno de los planes que lo componen, tal y como se muestra en la Ilustración 10.

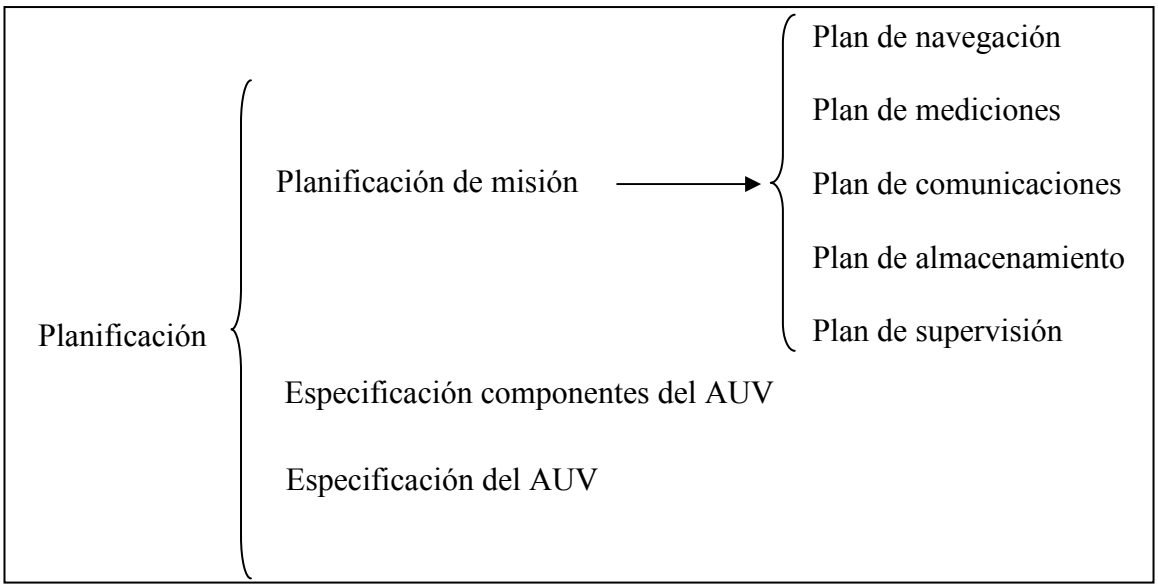


Ilustración 10: Esquema de la interfaz de Planificación

Lo que finalmente se ha implementado del esquema mostrado en la Ilustración 9 son los puntos siguientes:

- **Planificación de misión**: Se ha implementado un prototipo que da soporte a la definición por parte del usuario de cada uno de los cinco planes que componen una misión.
- **Definición de la misión**: El prototipo soporta la definición de una misión a partir de una lista de planes ya definidos.
- **Validador de la misión**: Se valida la misión frente a un AUV definido en código.

5.2 Definición de los planes.

La definición de los planes permite al usuario especificar los diferentes planes que van a formar parte de una misión. Una misión constará de cinco planes. Para definir una misión deberemos especificar cada uno de esos cinco planes.

5.2.1 Plan de navegación

Este plan será el más complejo de los cinco planes ya que habrá que definir sobre un mapa los puntos y áreas a recorrer y las zonas prohibidas para el submarino. En esta sección veremos las distintas formas de definir el plan de navegación y cómo va a poder definir de la forma más sencilla posible el usuario un plan de este tipo.

5.2.1.1 Seguimiento de Ruta

Como se veía en la sección anterior, en el diseño de la misión, en la que se analizaba el plan de navegación, el usuario puede especificar una ruta mediante una serie de puntos denominados waypoints señalados sobre un mapa. Cada uno de los waypoints se define como una tripleta, Latitud, Longitud y profundidad. El usuario en la interfaz, además, deberá definir el orden o prioridad con la que se recorrerán cada uno de los waypoints especificados.

La ventaja de definir prioridades en lugar de una ruta es que dejamos al planificador de misiones que defina la ruta de la forma más eficiente posible. Además, en caso de que el submarino, en un determinado momento de la misión, decidiera que no puede recorrer todos los waypoints (debido a alguna emergencia, como por ejemplo, que no tenga la batería suficiente como para recorrerlos todos), se podría replanificar la misión para recorrer sólo los waypoints con mayor prioridad, ignorando los de menor.

Aunque se pueden definir waypoints con prioridades y áreas, una ruta siempre debe tener un waypoint inicial y uno final. Se ha establecido este requisito ya que el usuario que vaya a echar al mar el vehículo submarino debe conocer de antemano cuál es el punto en el que se va a realizar la inmersión y cuál es el punto en el que se espera recoger el AUV.

Por lo tanto, la forma en la que se podrá definir el usuario en la interfaz la trayectoria a recorrer será:

1) **Establecer orden de recorrido**: Definir todos los waypoints y el orden en los que se tendrá que recorrer cada uno de ellos.

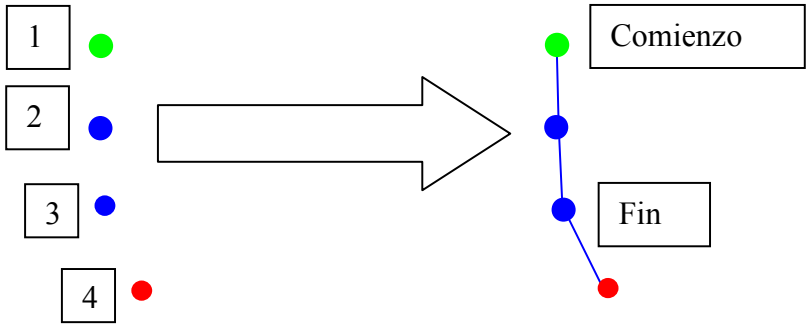


Ilustración 11: Orden de recorrido

En la Ilustración 11 se puede ver como se está definiendo una ruta en la que el AUV tiene que cumplir con esos waypoints y además en ese orden. El planificador no tendrá que hacer nada, ya que el usuario ha definido el orden en el que debe seguir el AUV la ruta.

2) **Establecer prioridades:** Definir todos los waypoints y establecer prioridades (no el orden exacto) con las que se tendrán que recorrer esos waypoints.

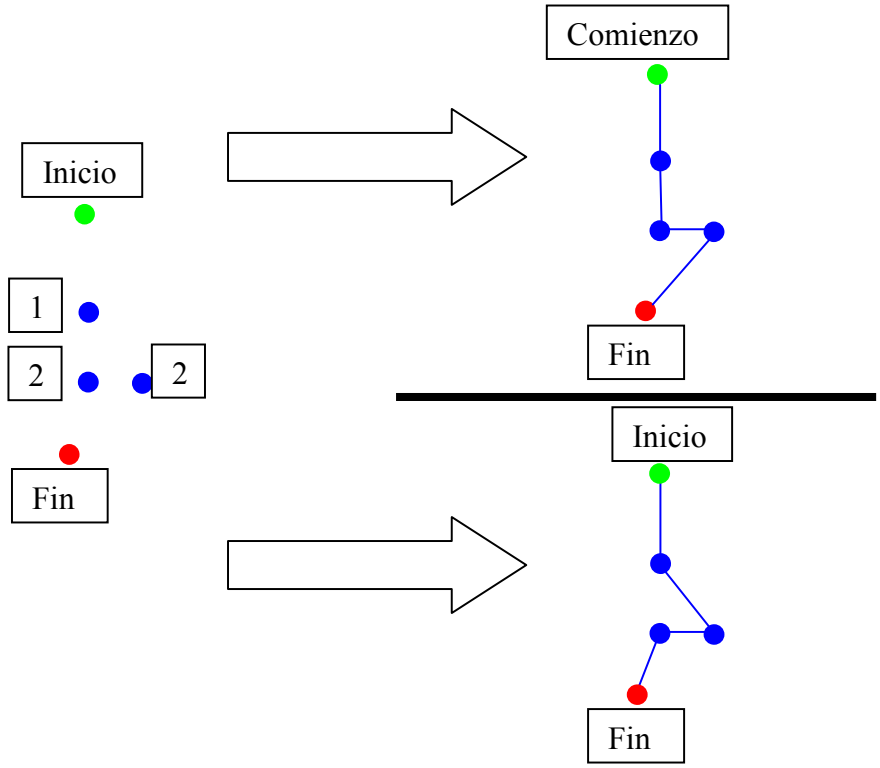


Ilustración 12: Prioridades

Como se puede ver en la Ilustración 12 el usuario tan sólo establecerá prioridades, dejará todo el trabajo al planificador. Establecerá los puntos por los que debe pasar el AUV pero no el orden en el que se debe recorrer (tan solo cuál debe ser el waypoint inicial y cuál el final).

3) **Establecer prioridades y rutas:** Definir una serie de waypoints con la misma prioridad pero algunos waypoints enlazados entre sí que indicarán que si el submarino llega a un waypoint en concreto, deberá ir seguidamente al waypoint que está enlazado a él.

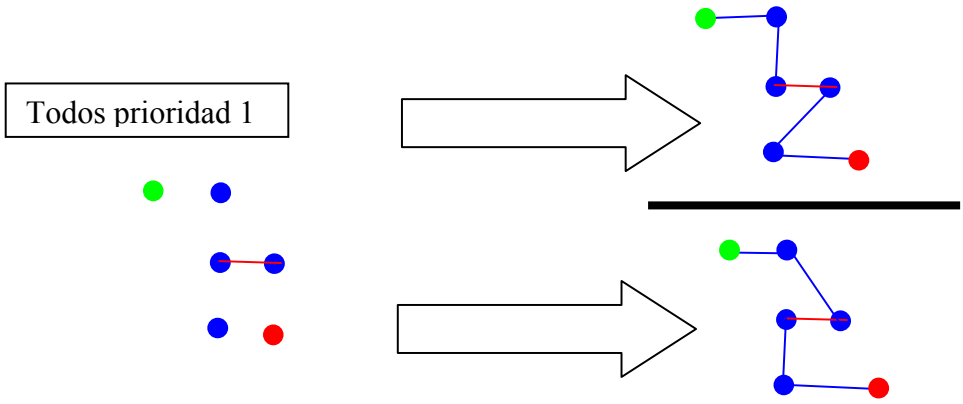


Ilustración 13: Prioridades y rutas

Es similar al descrito en el punto 2), solo que aquí estableceremos además unas rutas de paso obligatorio. Queremos que pase entre el waypoint 2 y el 3, pero que los demás waypoints los recorra de la forma lo más eficiente posible, sin importar el orden. En ese caso hacemos lo que nos muestra la imagen.

4) **Áreas de exploración:** Definir el área de exploración y la forma de recorrerla. En el siguiente apartado se detallan las opciones disponibles.

5.2.1.2 Exploración de área

La exploración del área la podrá definir el usuario en base a las siguientes alternativas: recorrido predefinido, seguimiento en base a medidas, toma de muestras tipo boya y exploración combinada.

Recorrido predefinido.

Definiremos la forma en la que el AUV recorrerá el área. Se dispone de dos opciones de recorrido: en zigzag y en espiral. Para cada una de ellas es posible indicar el número de ciclos que contendrá la trayectoria.

Seguimiento en base a Medidas

En esta alternativa se recorre el área en función de las medidas que se obtengan de una determinada magnitud físico/química. Existen dos modos de búsqueda seleccionables: gradiente y rango. En el modo gradiente, el AUV deberá seguir un parámetro físico/químico a medida que éste aumente o disminuya. En el modo rango lo que se busca es mantener al submarino dentro de unos valores del parámetro físico/químico.

¿Cómo se puede editar una misión en la que se indique que se desea seguir un gradiente determinado, o la detección de determinadas señales?

El usuario podrá especificar una magnitud. A continuación también podrá decidir de que forma quiere que esa magnitud sea seguida, si mediante un gradiente(seguir la magnitud en orden ascendente o descendente) o que siga la magnitud dentro de unos rangos inferiores y superiores dentro de los cuales deberá navegar el submarino y siempre sin salirse del área establecida. Es como si siguiera un camino cuyos límites los marca el rango de medidas especificadas.

Toma de muestras tipo Boya

En este tipo de exploración se persigue la toma de muestras periódica en una zona de interés, pudiendo emplearse una configuración estática o a la deriva:

- *Configuración estática*: En este tipo de configuración el usuario debe ser capaz de definir en el mapa una posición en la que el submarino se tendrá que

mantener con ayuda de los motores para contrarrestar las corrientes que lo arrastren lejos de esa posición.

- *Configuración deriva*: El submarino, a diferencia de con la configuración estática, se dejará llevar por las corrientes partiendo desde un punto inicial definido por el usuario, y, normalmente, estando delimitada la zona por la que se puede mover el submarino por un área.

5.2.1.3 Exploración combinada

La mayoría de los planes de navegación contendrán tipos de exploración combinados. Dejaremos al usuario la capacidad de poder trazar un seguimiento de ruta simple, conformada por waypoints y transectos, unida con la exploración de una u otra forma de diversas áreas. Los disparadores permitirán programar la activación de determinadas formas de recorrer un área en función de determinadas condiciones lógicas.

5.2.1.4 Restricciones temporales

Una vez definida la forma en la que el usuario va a poder definir las zonas a explorar por el submarino, también podrá establecer una serie de restricciones temporales, que servirán al submarino como guía para saber si se está retrasando en una misión o si debe terminar de explorar un área. Las restricciones temporales se podrán especificar una vez se hayan establecido los transectos y áreas que van a formar parte de la ruta. Esto será, según la forma de establecer la ruta que escoja el usuario, antes o después de validar la misión.

A continuación se listan los elementos en el mapa a los que se les podrá asignar una restricción temporal:

- **Transectos**: Cada transecto podrá contener restricciones temporales que orienten al submarino en la determinación de la velocidad que va a tomar en ese transecto o si se está retrasando demasiado.
- **Áreas**: Obligará al submarino a permanecer, como máximo, el tiempo establecido en esta restricción temporal.
- **Ruta**: A la ruta final se le puede establecer una restricción total, que, por supuesto, fuera mayor que la suma de las posibles restricciones temporales asignadas a los elementos que componen la ruta a los que es posible asignarle una restricción temporal (transectos y áreas).

La restricción temporal se puede utilizar tanto como restricción “dura”, como por ejemplo en el caso del área, que si se cumple el tiempo el submarino pase automáticamente al siguiente objetivo, o como una restricción “blanda”, para que el submarino tome conciencia de que se está retrasando más de lo habitual, debido a determinados factores como pueden ser corrientes marinas adversas, etc, y tome medidas al respecto (como por ejemplo, un aumento de velocidad) o simplemente lo avise en la siguiente comunicación que se produzca con el controlador de la misión.

5.2.1.5 Excepciones

Las excepciones que deberán ser controladas desde la planificación pueden ser únicamente las excepciones que conlleven fallos de determinados sensores, ya que las demás excepciones podrán ser manejadas por el submarino y conllevarán probablemente a abortar la misión.

De esta forma, podemos especificar tres grupos:

- Sensores esenciales: Sin los cuales la misión no tiene sentido. Si nos falla un sensor que sea esencial para la misión la solución sería abortar. El vehículo deberá dirigirse hacia el punto final de la misión.
- Sensores importantes: Son los sensores que no son esenciales pero sería recomendable que estuvieran funcionando porque aportan riqueza de datos a la misión. Podríamos decir que si falla un sensor de este tipo, el submarino puede continuar la misión. En cambio si falla más de uno, habría que abortar la misión.
- Sensores poco importantes: Son los sensores que no aportan mucho a la misión que se está definiendo y por lo tanto no nos va a importar quedarnos sin uno o más de éstos sensores. El vehículo continuará a pesar de ellos.

5.2.2 Generalidades sobre los planes.

Como hemos visto en el capítulo de diseño de las misiones, cuando hablábamos de los planes de comunicación, medición, almacenamiento y supervisión, todos ellos van a funcionar con un conjunto de tareas en las cuales va a ver un conjunto de disparadores que van a activar las acciones asociadas a estas tareas. La especificación de los disparadores será común para todos los planes. Los tipos de acciones a tomar cuando éstos se activen serán más específicos a cada plan.

En general, las interfaces para definir estos planes serán más sencillas que la interfaz para definir el plan de navegación, ya que constarán únicamente de casillas en las que el usuario rellenará los datos que se le solicitan. Además se podrá pensar un diseño en el que se pueda reutilizar la mayor parte de código posible para todas estas misiones, debido a su semejanza.

5.2.2.1 Plan de Mediciones

El usuario tendrá que especificar, en el plan de mediciones:

- Medida: Medida a tomar por el submarino. Será el submarino el que decida qué sensor utiliza para tomar la medida.
- Número de muestras a tomar de esa medida.
- Frecuencia con al que se desea tomar esa medida. El usuario podrá fijar la frecuencia y la unidad con la que está especificando ésta.

- Resolución: Al igual que lo que ocurre con la frecuencia, el usuario definirá un valor y su unidad.
- Sensor: Optativamente, se podrá definir el sensor con el que se quiere tomar la medida. En este caso el usuario será consciente de los sensores que lleva a bordo el submarino, y este plan de mediciones solo será útil para submarinos que contengan este sensor entre su equipación.

5.2.2.2 Plan de almacenamiento.

El plan de medidas y el de almacenamiento son muy dependientes entre sí, por lo que el usuario ha de tener muy en cuenta este aspecto antes de comenzar a hacer el plan de almacenamiento.

A la hora de definir el plan de almacenamiento, el usuario deberá especificar las siguientes características:

- Medida que se quiere almacenar.
- Número de muestras que se desea almacenar.

5.2.2.3 Plan de comunicaciones

La interfaz para definir el plan de comunicaciones deberá contener una lista de acciones, en las que el usuario podrá definir cada una de ellas, añadir, eliminar y modificar, tal y como sucedía con el plan de mediciones.

Hay dos tipos de acciones: de recepción de datos (acción en la que el usuario especifica al submarino que va a recibir un dato o medida determinado en esta acción de comunicación) y de emisión de datos (el submarino que se debe preparar para la emisión de datos al controlador de la misión). El usuario podrá definir en esta interfaz, para cada acción, los siguientes datos:

- **Medida o dato a enviar/recibir**.
- **Destinatario o fuente**.

5.2.2.4 Plan de supervisión

Como veíamos, el plan de supervisión tendrá dos tipos de acciones, la ejecución de un plan o la ejecución de un comando. Tanto para uno como para otro, se podrán especificar los siguientes parámetros:

- **Nombre del plan/comando**: Será el nombre del plan completo (como puede ser abortar misión o emerger, para lo que necesita planes nuevos completos que hubieran sido previamente almacenados en el submarino) a ejecutar por el AUV o de un comando en concreto.

- **Lista de parámetros**: Tanto el nuevo plan como el comando pueden tener una lista de parámetros asociados con los que se ejecutarán en el submarino. Cada uno de estos parámetros tendrán un nombre y un valor.

Se puede ver como la definición de este plan es un poco más compleja que los cuatro anteriores, ya que necesitamos, para cada tarea, una lista de acciones para los que a cada uno de ellos se puede asociar una lista de parámetros.

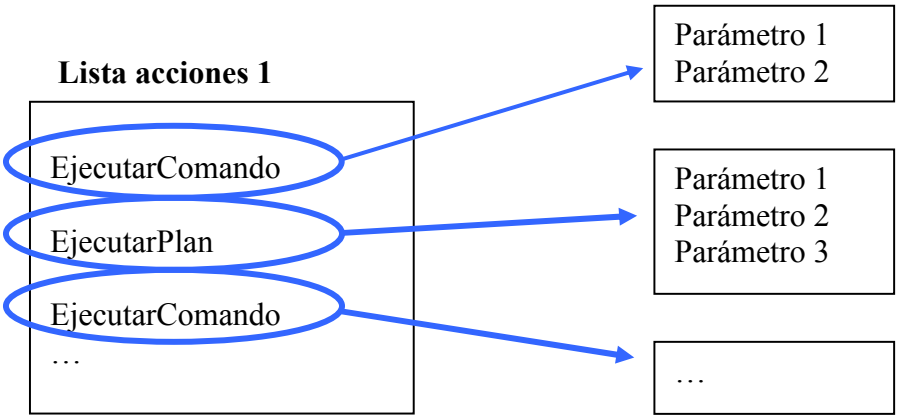


Ilustración 14: Lista de acciones

Como vimos en apartados anteriores, el plan de supervisión iba a poder modificar variables de los demás planes. Por ello, podríamos poner en la interfaz una especie de lista de variables de planes ya definidos que vamos a poder modificar. Por ejemplo, al introducir el usuario el comando "CambiarVariableMisión", que aparezca una ventana con la lista de variables que se hallan definido en los planes ya definidos, para que el usuario seleccione la variable que quiere introducir en este comando, y no tener que definirla a mano, ya que puede ser más engorroso.

5.2.2.5 Definición de los disparadores.

Para que el usuario defina cada plan, tendrá que especificar la lista de disparadores con las que se desea que se lance una acción, y además, la lista de acciones que quiere que se ejecuten para cada lista de disparadores, tal y como se muestra en la Ilustración 15.

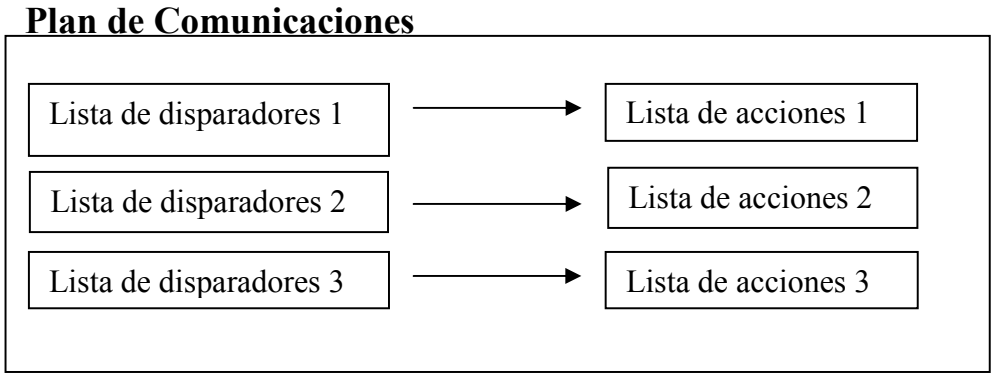


Ilustración 15: Plan de comunicaciones

En el capítulo de diseño de las misiones se explicaba que hay tres tipos de disparadores. El usuario podrá definir listas que incluya los tres tipos en la misma lista.

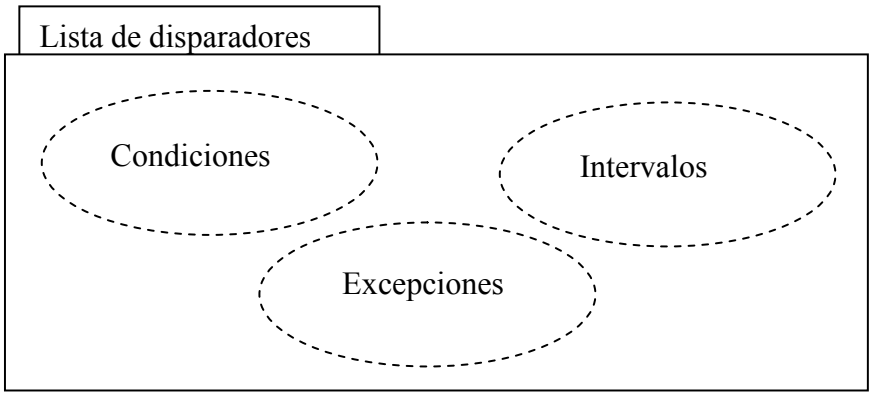


Ilustración 16: Lista de disparadores

- **Condiciones**: Para definir una condición, el usuario tendrá que especificar la medida que desea controlar, la comparación y el valor con el que se compara la medida para saber si se cumple la condición. Por ejemplo, si el usuario quiere como condición que la temperatura sea mayor que 10º tendrá como medida la temperatura, como comparación el "<", y como valor a comparar con la medida el 10.
- **Intervalos**: En un intervalo el usuario definirá un intervalo de valores para una medida, y esta condición se disparará cada x tiempo. Por ejemplo, si el usuario quiere que se ejecute una acción cuando el submarino se encuentre en una profundidad entre los 100 metros y los 500 metros, cada 5 minutos, especificará como medida la profundidad, como valor inicial los 100 metros, como valor final los 500 metros y con un periodo de 5 minutos.
- **Excepciones**: Este tipo de disparadores será el más sencillo de definir, ya que lo único que tendrá que especificar el usuario será la excepción con la que desea que se lance la acción o la lista de acciones asociadas a este disparador.

En la interfaz podremos tener una única ventana que sea similar para todos los planes del submarino (excepto para el plan de navegación) en el que se puedan definir los disparadores. Esto nos permitirá no reimplementar la misma ventana para cada uno de los cuatro planes a definir por el usuario.

5.2.3 Definición de la misión.

Finalmente, el usuario pasa a una ventana en la que se va a decidir de qué planes va a estar compuesta la misión. En este punto, el usuario ha definido todos los planes que necesita. Se recomienda que el usuario haya definido los planes en el siguiente orden:

- **Plan de navegación**: Al definir los demás planes, el usuario se puede ayudar del plan de navegación como base para establecer donde se tomarán las medidas, cuáles de ellas se almacenarán, cuándo se producirán las comunicaciones y por último generar el plan que los supervise a todos los demás.
- **Plan de medidas**: Será recomendable conocer el plan de navegación antes de ponernos a definir cualquier plan posterior. La razón es que si conocemos el plan de navegación podremos establecer los disparadores de las acciones acordes a la posición en el mapa.
- **Plan de almacenamiento**: Este plan es recomendable que se defina después del plan de medidas. Con ello evitaremos especificar que se almacenen medidas que no se han tomado en el plan de mediciones.
- **Plan de comunicaciones**: Sería recomendable conocer, antes de definir este plan, el plan de navegación (para saber cuándo es posible la emisión de datos), el plan de medidas y el plan de almacenamiento (para conocer de qué medidas almacenadas disponemos para emitir).
- **Plan de supervisión**: Este podría definirse el último. Se deberán conocer todas las variables de los demás planes en caso de que se quiera acceder a ellas para modificarlas en cualquier punto de la misión y tener la capacidad de decidir cambiar de planes.

Si el usuario que está definiendo la misión sigue este orden, se minimizarán las posibilidades de encontrar incompatibilidades entre los distintos planes de la misión.

5.2.4 Validación de la misión.

Como hemos visto en el estudio de los distintos planes, intentamos que los planes sean independientes entre ellos, aunque llegado el momento de generar la misión, debemos asegurarnos que son compatibles entre sí y con el AUV que se encargará de realizar la misión.

En esta sección se verá algunos de los puntos que se puede tener en cuenta a la hora de realizar la validación de la misión.

Podemos tener en cuenta que hay determinadas estructuras generales que se cumplen para todos los planes. En este caso, los disparadores. Éstos aparecen en todos

los planes de la misión. Por ello, se puede hacer un análisis de las validaciones que necesitan los disparadores y otro específico para cada plan.

5.2.4.1 Validaciones en los disparadores.

Los disparadores tendrán una lista de condiciones y excepciones que deben ser compatibles con el AUV y con los demás planes de la misión. A continuación se muestran una serie de puntos que se pueden validar de los disparadores con respecto a los demás planes de la misión y al AUV que se haya escogido para esta misión.

- **Medidas**: la mayoría de las condiciones e intervalos de los disparadores se expresarán con ayuda de medidas que tomará el submarino durante la misión. Se deberá validar con el AUV definido para la misión que éste está equipado con los componentes necesarios para tomar las medidas especificadas en los disparadores. Además, se debe comprobar, en caso de medidas para la misión (las medidas que monitorizan el estado interno del AUV), que éstas son tomadas efectivamente en el plan de comunicaciones. En caso contrario, esos disparadores nunca podrán lanzarse debido a que, aunque el submarino está equipado con los sensores adecuados, éstos no se utilizan en la misión.

- **Waypoints, áreas o transectos**: En el plan de navegación de la misión deberán haberse definido esos elementos para los que se está asignando disparadores. No se podrá asociar un disparador, por ejemplo, a un waypoint inexistente en el plan de navegación. Un transecto es un concepto que solo existe en el programa planificador, como ya veremos más adelante. Pero éste se traducirá a un elemento de tipo ruta en el plan de navegación.

- **Disparadores de las áreas**: Los disparadores de recorrido no deben activarse junto con los disparadores definidos para otro tipo de recorrido en la misma área. Por ejemplo, si establecemos dos formas de recorrer un área y a la primera le asignamos un disparador en el que la profundidad > 100, para la otra forma de recorrer el área se deberá establecer el disparador contrario, profundidad ≤ 100.

5.2.4.2 Validaciones en los planes

En los distintos planes que componen la misión se pueden dar las siguientes validaciones:

- PdN → AUV:
  - **Profundidad**: En el AUV se habrá establecido una profundidad máxima a la que puede descender el submarino. En el plan de navegación no puede haberse especificado ningún waypoint que baje más allá de esa profundidad, ya que correríamos el riesgo de perder el AUV.
  - **Velocidad**: El plan de navegación no debe sobrepasar, en ninguno de sus puntos, la velocidad máxima que puede alcanzar

el submarino. Si esto ocurriese, el plan de navegación no podría ser realizable por el submarino en cuestión, al menos en el tiempo establecido.

- **Autonomía**: La distancia máxima que puede recorrer el AUV con su sistema de alimentación pondrá límite a la distancia que se requiere que recorra el submarino en el plan de navegación.

- PdA → AUV:

  - **Medidas**: En el plan de almacenamiento se establecerá que hay unas medidas que deben almacenarse en el submarino para después ser comunicadas al monitorizador de la misión o simplemente almacenarlas hasta que en el final de la misión, el AUV sea recogido y se extraigan esas medidas. El AUV debe ser capaz de tomar esas medidas para su posterior almacenamiento. En caso de que no estuviera equipado con los sensores adecuados, debe producirse un error durante la validación.

- PdC → AUV:

  - **Medidas**: Las medidas que se especifican en las acciones del plan de comunicaciones deben estar representadas en el AUV mediante sensores que sean capaces de tomar esas medidas durante la misión.

- PdC → PdN:

  - **Dispositivos de comunicación**: Si no hay ningún dispositivo de comunicación que permita, por ejemplo, la comunicación submarina, sin necesidad de emerger, definido en el AUV, se debe comprobar que efectivamente el submarino ha emergido cuando se desea realizar la comunicación con el monitorizador de la misión. En el caso de que, debido a como está definido el plan de navegación, sepamos de antemano que la comunicación se producirá mientras el submarino está sumergido, debe producirse un error en la validación de la misión ya que esta comunicación será inviable.

- PdM → AUV:

  - **Medidas**: Al igual que con los anteriores planes, es muy importante comprobar que todas las medidas especificadas en las acciones del plan de medidas puedan ser tomadas por el AUV. De otra forma habrá que informar del error durante la validación de la misión.

- PdA → PdM:

  - **Medidas**: Las medidas que se tomen con los sensores de la misión deben estar definidas en el plan de mediciones. Es decir,

no se puede almacenar una medida si esta no se especifica en el plan de mediciones, ya que no se puede almacenar una medida que no se está tomando. Para los sensores internos y la instrumentación sí que se permitirá, ya que éstos no necesitan de una orden del plan de medidas para que sean monitorizados, ya que el submarino estará constantemente tomando estas muestras para navegar o lanzar alguna excepción (por ejemplo, que la temperatura interna alcance valores peligrosos para el AUV).

- PdC → PdM:
  - **Medidas**: Las medidas que se pretendan comunicar deben estar especificadas en el plan de medidas, de modo que no se pedirá la comunicación de medidas de los sensores de la misión que no sean especificadas en el plan de medidas.

5.3 Definición de AUV

El usuario podrá definir un AUV de dos formas distintas. Primero, podría seleccionar entre una cantidad de submarinos definidos previamente para la misión en cuestión. Otra alternativa sería definir un submarino desde cero, especificando todo lo necesario para el submarino.

5.3.1 Especificación de un submarino.

La especificación de un submarino se deberá realizar cuando el usuario quiera introducir un nuevo submarino en la base de datos de la aplicación. Se especificará desde cero un submarino, dándole nombre y proporcionando al programa los datos que se mencionan a continuación.

Si no tenemos en cuenta las primitivas, el programa debería conocer determinados aspectos del AUV, ya que posteriormente al validar una misión, el programa planificador deberá contrastar los datos de la misión con la capacidad del submarino que se habrá elegido para ser usado en la misión.

Para poder validar la misión, el usuario debería tener que definir los siguientes aspectos:

- **Profundidad máxima del submarino**: El validador de misiones, deberá comprobar si el submarino que se está utilizando en la misión previamente planificada es capaz de alcanzar las profundidades que especifica la misión.
- **Velocidad máxima del submarino**: El validador de misiones utilizará este dato para conocer si es posible que el submarino recorra la distancia especificada en la planificación de la misión en el tiempo especificado.

Además, se exige que se tenga conocimiento de los siguientes datos del submarino:

- **Dimensiones**: Se ha de conocer las dimensiones del submarino para poder ubicar los sensores para los que sea necesarios conocer su ubicación, por ejemplo, un capturador de imágenes.

- **Centro de gravedad**: El centro de gravedad del submarino no será necesario conocerlo en el planificador a la hora de validar la misión, pero sí tendrá que conocerlo a la hora de comunicar la misión al submarino, ya que el submarino real demanda estas características. Será necesario decírselo al submarino ya que el centro de gravedad puede variar según los sensores que se hayan instalado para esa misión en concreto.

- **Imágenes**: Aunque no es imprescindible, puede ser aconsejable tener unas imágenes del submarino para que el usuario vea como es éste.

- **Componentes del submarino**: Además de éstas características se ha de definir la lista de componentes que tiene el submarino. Esto conlleva a todo lo que el submarino en sí tendrá, además de aquellos componentes que se le podrá añadir para una misión en concreto.

Los distintos sistemas que componen el submarino deberán definirse en otra ventana, y está explicada en una de las secciones que siguen. En esta ventana simplemente se escogerá de una lista de sistemas, los que contendrá el submarino, y a su vez, la lista de componentes que conforman estos sistemas.

Habrá varios sistemas en el submarino para los que habrá que especificar todos sus componentes. Los sistemas que tendrán en un principio el submarino serán los siguientes:

- **Impulsión**: El sistema de impulsión serán los motores del submarino. Se especificarán éstos para definir el submarino.

- **Alimentación**: El sistema de alimentación constará de todas las baterías que incluya el submarino.

- **Comunicación**: incluirá a los sistemas de comunicación del submarino.

- **Sensorial**: El sistema sensorial incluirá a todos los sensores del submarino.

- **Actuador**: Los actuadores irán separados conceptualmente de los impulsores, ya que los impulsores son totalmente necesarios para los movimientos del submarino, mientras que los actuadores tratarán de, por ejemplo, tomar muestras. Los actuadores suelen ser añadidos al submarino para misiones específicas.

- **Procesamiento**: Incluirá a las distintas unidades de procesamiento del AUV.

Para cada nuevo componente que el usuario quiera incluir, tendrá que especificar a qué sistema del submarino pertenece.

5.3.2 Elección del submarino.

Cuando el usuario quiera validar una nueva misión para un determinado AUV, tendrá que hacer la elección de un submarino que exista en la base de datos del programa, o bien que haya sido previamente creado por él mismo. El usuario podrá coger el submarino tal cual o modificarlo agregándole a o quitándole componentes. En caso de que el usuario quiera añadirle componentes que no estuvieran definidos en el programa, tendría que definirlos antes de poder asignárselos al submarino.

La selección del AUV será simplemente elegir en una lista donde se mostrará el nombre y, a petición del usuario, más datos del submarino en una determinada ventana de la interfaz del programa.

Una vez seleccionado el submarino, se añadirán los componentes que se crean necesarios para llevar a cabo esa misión, en caso de ser necesario. Lo más probable es que se quiera añadir nuevos componentes a los sistemas sensorial y a los actuadores, ya que añadir componentes a otros sistemas supondría modificar el submarino en sí, modificando sus características (por ejemplo, modificar o añadir impulsores aumentaría la velocidad del submarino, y por tanto, una de sus características intrínsecas, por lo que se podría tomar como un nuevo submarino, o una versión mejorada o amplificada del anterior).

El validador de la misión tendrá que decidir si es posible llevar a cabo la misión o no en base al análisis de las posibilidades de la misión, como puede ser que los sensores que incluye el submarino cubren todas las posibilidades, o que la autonomía del submarino incluyendo todos los instrumentos que tiene éste no es suficiente para realizar el recorrido completo.

5.4 Interfaz de la definición del AUV

La interfaz para la definición de un AUV por tanto deberá constar de dos partes, una en la que el usuario pueda definir por completo un submarino y otra parte, que podría ser parte de la propia planificación de la misión, en la que el usuario hiciera su elección del submarino para la misión que está desarrollando y añada los componentes necesarios para ella.

La interfaz para la definición del AUV será una interfaz sencilla, en la que simplemente se solicitan los datos adecuados para llevar a cabo la definición del submarino. Estos datos son los que se nombraban antes para la definición de un AUV.

Además, en esta definición, se debe dar la posibilidad de invocar ventanas que permitan al usuario definir nuevos componentes, con el fin de que el usuario no tenga que cambiar de ventana para poder definirlos.

La otra ventana nos permitirá seleccionar de entre una lista de submarinos, el que realizará esta misión. En esta interfaz se debe poder visualizar todos los datos referentes al AUV que seleccione el usuario, para que pueda ver si realmente es el AUV que desea seleccionar. Además, al AUV seleccionado se le podrá añadir componentes adicionales para llevar a cabo la misión.

Una vez seleccionado el AUV solo tendremos que realizar la validación de la misión, con los planes de navegación, comunicación, y medición escogidos, el submarino elegido y los componentes añadidos a éste.

5.4.1 Especificación de los componentes de un AUV

Cuando un usuario especifica los componentes para un AUV, tendrá que especificar tan solo aquellas características que sean necesarias, de esos instrumentos, tanto para la planificación como para la comprobación que se debe llevar a cabo en el submarino.

Si obligamos al usuario a definir cada uno de los aparatos que contiene el submarino, con cada una de sus características, estaríamos obligándolo a hacer un trabajo realmente tedioso y aburrido.

Por ello, solo vamos a pedir lo estrictamente necesario de cada elemento que contenga el submarino para realizar la validación de la misión, y para que el submarino sepa qué es lo que necesita tener y realice sus propias comprobaciones.

Para saber qué es lo que necesitamos, vamos a hacer un análisis con cada uno de los elementos que contendrá el submarino y lo que necesitará conocer el planificador sobre él elemento en cuestión.

A continuación se listan cada uno de los sistemas del AUV y se especifica lo que necesitamos conocer de cada uno de sus elementos. Hay que tener en cuenta que para poder almacenar cada uno de los sistemas que el usuario especifique, primero debemos conocer el nombre con el que se desea denominar al sistema en cuestión. Por ello, el nombre será un requisito indispensable.

5.4.1.1 Sistema de procesamiento.

Se puede definir el consumo global aproximado del sistema de procesamiento. El usuario puede hacer una aproximación a este consumo si no quiere especificar elemento a elemento el consumo de cada uno de ellos. En el sistema de procesamiento tenemos tres tipos de elementos que podemos definir:

CPU

- **Frecuencia**: No es un requisito indispensable para la definición del sistema de procesamiento, ya que el planificador no realizará comprobaciones con este dato. Será un dato mas bien informativo para el usuario.

- **Consumo**: Es importante especificar cuál será el consumo de los procesadores, puesto que será un dato importante para obtener la autonomía del submarino. Quizás aquí podríamos especificar consumos según la frecuencia del procesador. Por ello aquí puede ser útil especificar la frecuencia, si se especifica el consumo. No es un dato indispensable siempre y cuando se especifique el consumo del sistema completo.

Memorias

- **Capacidad**: Necesitaremos conocer la capacidad de la memoria, para saber si podemos almacenar la cantidad de datos que se solicitan en el plan de medidas, teniendo en cuenta que éstos datos se pueden eliminar una vez comunicados.

- **Consumo**: Necesitamos conocer el consumo total de la memoria, a no ser que el usuario especifique el consumo total aproximado del sistema de procesamiento.

Tarjeta de adquisición

- **Frecuencia**: La frecuencia de la tarjeta de adquisición nos ofrecerá un dato importante para conocer la capacidad de adquisición de datos que tiene el submarino.

- **Consumo**: Se puede especificar el consumo de la tarjeta de adquisición, o especificar el consumo global aproximado del sistema de procesamiento.

5.4.1.2 Sistema de impulsión

Para el sistema de impulsión también se puede definir un consumo global de energía, aunque es más recomendable definir el consumo energético de cada elemento, con el fin de que el submarino pueda calcular el gasto energético en cada momento, aunque no estén funcionando todos los impulsores que posee.

Los datos que son necesarios definir para el sistema de impulsión son los siguientes:

Motores

- Consumo: Es recomendable definir el consumo energético de cada motor.

Mandos

- Consumo.

5.4.1.3 Sistema de alimentación

En el sistema de alimentación ocurre lo mismo que con los anteriores sistemas, pero al revés. En este caso es recomendable conocer el aporte energético de cada batería, para que el submarino pueda conocer, aunque deba prescindir del aporte energético de alguna batería, cual es la carga que queda aún en las demás baterías.

- **Carga**: Como se ha dicho, es recomendable conocer el aporte energético de cada submarino.

5.4.1.4 Sistema de comunicación

Como siempre, se podría especificar un consumo total para el sistema en cuestión, o el consumo para cada elemento.

- **Ancho de banda**: El ancho de banda de los dispositivos de comunicación puede ser útil para calcular el tiempo que durarán las comunicaciones entre el submarino y el planificador, o entre el submarino y los demás submarinos.

- **Consumo**: Se especificará el consumo de cada dispositivo de comunicación para conocer el coste energético que supondrán las comunicaciones.

5.4.1.5 Sistema de actuadores

Es muy recomendable en este caso, definir el consumo de cada elemento que pertenezca a este sistema por separado, ya que, al igual que lo que ocurre con el sistema de comunicación, no siempre estarán activos todos los actuadores del sistema de actuadores.

5.4.1.6 Sistema Sensorial

Dentro del sistema sensorial se definen tres tipos de sensores:

Sensores internos

Los sensores internos son los que miden el estado interno del submarino. Suelen estar constantemente activos, por lo que para los sensores internos podríamos definir un consumo global aproximado, con el fin de no estar definiendo el consumo de cada uno de los sensores internos del submarino.

- **Magnitud**: El planificador necesita conocer las magnitudes internas que está capacitado a tomar el submarino, para contrastarlo con los datos que necesita el usuario enviar en las comunicaciones.

Instrumentación

La instrumentación será la que esté midiendo el estado externo del submarino. Puede que haya sensores que no estén constantemente midiendo, puesto que podrían estar por duplicado, por lo que lo recomendable en este caso sería definir el consumo de cada sensor.

- **Magnitud**: El planificador necesita conocer las magnitudes que puede tomar el submarino, para contrastarlo con la definición de los planes de comunicación que ha definido el usuario.

Sensores para la misión.

Los sensores para la misión son los sensores que se añaden al submarino para realizar esta misión en concreto. Como siempre, es muy recomendable establecer el consumo de cada sensor de misión por separado, ya que en cualquier momento de la misión es muy probable que no se estén utilizando todos los sensores de la misión.

- **Magnitud**: Es muy importante definir las magnitudes que se van a poder medir con los sensores para la misión, ya que de ello depende tanto el plan de mediciones, como el plan de comunicaciones.

5.5 Etapa de control

La función principal de la interfaz de control sería la de mostrar al usuario el estado de la misión en tiempo lo más cercano al real posible. Se trata de que el usuario pueda conocer, mientras el submarino está en plena misión, los datos recogidos por éste, sus interpretaciones del mundo, el posicionamiento real, etc. Además, este software tendría la capacidad de comandar el AUV, es decir, no actuar sólo como observador de la misión, sino que puede tomar un papel activo en ella.

Los datos que lleguen al usuario dependerán en gran medida de lo que se haya especificado en el plan de comunicaciones. El software se encargará de recibir todos los datos que lleguen a través de las comunicaciones con el submarino, que previamente han sido especificados en el plan de comunicaciones, y transmitirlos al usuario.

Esta interfaz se debe encargar de mostrar de la forma más legible y sencilla posible los datos que transmita el submarino. Una buena idea sería la de mostrar en un mapa la situación exacta del submarino, así como la situación que debería tener cuando realizó la última comunicación el submarino. Con ello nos haríamos una idea del error que está cometiendo el submarino en su trayectoria con respecto al plan original.

El usuario podría seleccionar los datos que quiere ver, con lo que solo tendría que ver los datos que él especificaría, aunque el software esté recibiendo muchos más datos y almacenándolos.

Quizás otra de las utilidades que podría tener este software sería la capacidad de ver tantas veces como desee el usuario los lugares por los que ha ido pasando el submarino y los datos que ha ido recogiendo el submarino en cada uno de esos puntos.

Para ello, el usuario debe tener la posibilidad de almacenar todos los datos que ha ido recogiendo el submarino en la trayectoria (esto se haría siempre por defecto) y cargar estos datos todas las veces que quiera para volver a ver cada uno de los puntos por los que ha pasado el submarino y los datos que ha recogido en cada uno de ellos.

Un ejemplo de este tipo de software se muestra en la Ilustración 17.

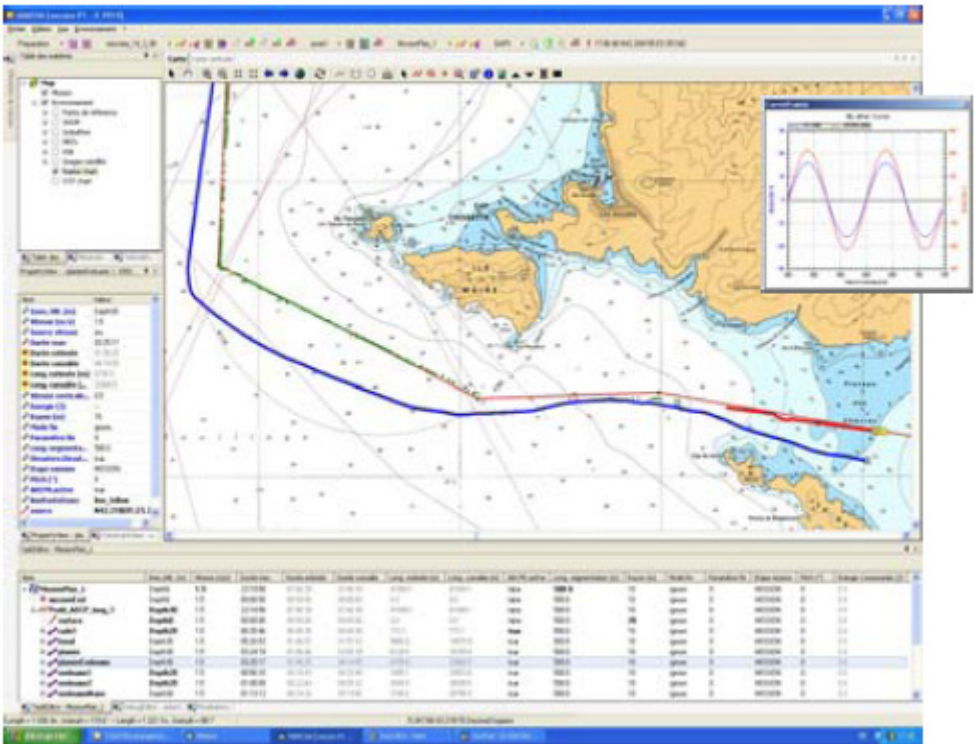


Ilustración 17: Software de control

Se puede ver una línea azul y otra roja. La línea roja es la línea del recorrido previamente planificado, mientras que la línea azul es la línea que realmente ha seguido el submarino.

Resumiendo, los objetivos que persigue este software son los siguientes:

- Control en tiempo real del desarrollo de la misión. Esto incluye mostrar cada uno de los puntos por los que está pasando el submarino y los puntos por los que debería haber pasado según la planificación seguida. También comandar al submarino para modificar su comportamiento si fuera necesario.

- Mostrar los datos que el usuario necesite conocer sobre las muestras recogidas por el submarino.

- Capacidad de playback, es decir, cargar una misión ya realizada y mostrar como ha transcurrido a medida que el submarino iba avanzando.

Para cumplir con el primer objetivo, tal y como se ha dicho antes, se hace necesario mostrar un mapa en el que se dibuje la ruta especificada en el plan y la ruta finalmente seguida por el submarino.

Para el segundo objetivo necesitamos incluir comandos que permitan seleccionar al usuario únicamente algunos datos de los que emite el sensor. Para ello, el usuario seleccionará los sensores que quiere examinar de una lista que incluya únicamente los sensores que tiene el submarino en esta misión.

Para el playback se hacen necesarios comandos para la reproducción de la misión, como pueden ser un comando de avanzar, otro para retroceder, un comando que permita especificar al usuario la velocidad de reproducción, un comando de pause, que detenga el playback justo en el punto en el que se estaba reproduciendo, y un botón de stop que además de detener la reproducción la posicione en el punto inicial.

De esta forma, el usuario podrá ir viendo en cada momento donde se encontraba el submarino y los datos que estaba tomando en un punto en concreto del mapa.

5.6 Replanificando la misión.

Aunque aún no se sabe si se implementarán este tipo de funciones, debido a su complejidad, se está pensando en que el usuario pueda replanificar la misión mientras el submarino la está llevando a cabo.

Para ello, el usuario tendría que definir una nueva misión, y, en la interfaz de control, especificar que se debe cargar esa nueva misión en el submarino la próxima vez que se establezca una comunicación.

El submarino, una vez que reciba esa orden, deberá dejar de seguir el plan inicial y comenzará a realizar el nuevo.

5.7 Análisis de comunicación

El submarino podrá realizar dos tipos de comunicaciones distintas. La primera será la que realizará entre su sistema de procesamiento y el sistema sensorial, para obtener los datos que necesite. A este tipo de comunicación le denominaremos comunicación interna, y para este proyecto no es necesario hablar de ella.

El segundo tipo de comunicaciones y el que tiene que ver realmente con este proyecto es el que se da entre el submarino y el planificador, y el que se da entre el submarino y los demás submarinos. A este tipo de comunicaciones la denominaremos comunicación externa.

De las comunicaciones externas, las que se producen entre los distintos submarinos hará falta conocerla para poder dejar al usuario implementar las comunicaciones entre submarinos de la forma más adecuada posible.

Sin embargo, en este proyecto las comunicaciones que se implementarán serán las que se producen entre el planificador y el submarino, lógicamente.

En caso de que el usuario quiera comunicarse con el simulador desde el planificador, en lugar de comunicarse con el submarino, las comunicaciones deberían ser las mismas. El planificador no tiene por que actuar de forma distinta frente a un simulador que frente a un submarino real. El simulador deberá, a su vez, “simular” ser un submarino real frente al planificador.

En principio la comunicación entre submarino y planificador o entre submarino y otros submarinos se realizará mediante sockets. El protocolo utilizado será el TCP/IP y habrá que establecer unos formatos de paquetes para que en cada comunicación el submarino y el planificador sepan lo que se están enviando uno a otro o entre distintos submarinos.

5.7.1 Formatos de comunicación

Habrá varios formatos de comunicación según lo que se desee emitir y quién sea el emisor de la información y quién el destinatario.

Básicamente, todos los formatos coincidirán en que tendrán un primer campo de metadatos que explicará qué es lo que se está comunicando, y con que formato, y un segundo campo que serán los datos. Este segundo paquete puede ser enviado o no, ya que según el formato del que estemos hablando puede que sea necesario o no.

Formato 1

Para este tipo de formato se da una comunicación entre administrador y submarino. El administrador puede ser tanto el planificador como otro submarino que supervisa la misión. Así mismo, el papel de submarino lo puede jugar también un simulador.

Este tipo de comunicaciones se pueden establecer, por ejemplo, cuando el planificador necesita enviar los datos de la misión al submarino, enviándole los planes y la definición del AUV, cuando se necesita enviar batimetría desde el planificador al AUV, etc.

Habrá una comunicación de tipo Servidor → Cliente en la que el Servidor puede ser el planificador y el cliente el submarino. El formato del mensaje puede ser el siguiente:

Paquete:

- datos de control
    - mision: id de la misión sobre la que se envían los datos.
    - sello temporal
    - tipo de dato: tipo de datos que se envían en el siguiente campo.
    - Checksum: Para errores.
    - Etc.
- datos en bruto. Pueden ser ficheros XML o datos batimétricos.

Los tipos de datos pueden ser:

- **PdN**: es como se denomina al plan de navegación. Será un fichero XML que contendrá el plan de navegación para el submarino.

- **PdC**: plan de comunicaciones. Al igual que el plan de navegación, será un fichero XML, pero éste contendrá especificado dónde se producirán las comunicaciones y qué se comunicará.

- **PdM**: plan de mediciones. Otro fichero XML que contendrá qué medidas se van a tomar y el lugar donde se piensan tomar las medidas.

- **PdS**: plan de supervisión. Será el plan que supervisará el funcionamiento del AUV.

- **Descripción de AUV**: Serán una serie de ficheros XML que definirán el AUV y los sistemas que lo componen.

- **batimetría**: el campo de datos contendrá un fichero con la batimetría que se quiere enviar al submarino.

- **paquetes configuración**: el campo de datos contendrá una lista de paquetes de configuración posibles para un sensor en concreto.

Formato 2

Este tipo de formato de comunicación se utiliza para emitir datos desde el submarino a un administrador. Cuando el submarino quiera establecer una comunicación con el administrador utilizará este formato.

Este paquete consta de varios campos:

- Datos de control: Serán los metadatos del paquete. Contendrá:
  - misión: indicará qué misión está realizando el submarino.
  - tarea de misión: indicará que tarea estaba realizando el submarino dentro de la misión.
  - tarea dentro del plan de comunicación, supervisión, medición: si esta comunicación estaba especificada en alguno de los planes especificados al submarino, especificará a qué comunicación pertenece.
  - origen (AUV): para que el administrador sepa desde qué AUV viene la comunicación.
  - etc.
- Metadato de datos. Fichero CDL(network Common Dataform Language).
- Datos en bruto. Fichero NetCDF **[NETCDF]**.

Como se puede observar, los datos que tenga que enviar el submarino al AUV se almacenarán en un fichero NetCDF. Habrá además un fichero en formato CDL en el que se almacenará información sobre cómo están los datos almacenados en el fichero NetCDF.

En este formato, al igual que en el anterior, tendremos un servidor, que será el administrador, y un cliente que será un AUV o un simulador.

Formato 3.

Este formato es para el envío de comandos de control remoto por teleoperación.

Para enviar comandos de este tipo hace falta previamente haber establecido una sesión con cada uno de los submarinos a los que se quiere comandar. En este caso, el AUV o el simulador harán de servidor, y el administrador de cliente.

Hay un tipo de comando que es el de respuesta, que es el que se produce cuando el submarino tiene que responder a un comando emitido por el administrador. En este caso hay que especificar en el argumento del mensaje a qué comando se le responde. Para realizar esta respuesta habrá que modificar el flujo de información, siendo en ese momento el AUV o el simulador el cliente y el Administrador el servidor.

Desde el administrador o el simulador no se utilizan comandos internos al AUV para comandarlo, sino que empleará comandos externos que el AUV se encargará de traducir en los comandos que necesite.

El paquete tendrá los siguientes campos:

- Nombre de comando: especificará el nombre del comando que se emite desde el administrador al AUV.
- Lista de Argumentos: contendrá la lista de argumentos que se han especificado con el comando.
- Estampa temporal
- etc.

Formato 4.

Este formato se utilizará para la comunicación de avisos. Es similar al formato 4, pero para mandar este tipo de mensajes no hace falta abrir una sesión con el AUV o el simulador.

Este paquete contendrá los siguientes campos:

- Aviso: El aviso que se desea mandar.
- Dirección: A donde se quiere enviar el aviso.

5.8 Análisis de la batimetría

Existen varias definiciones para la palabra relieve según la situación de la que estamos hablando. El relieve contiene información sobre la altitud de cada punto de una zona geográfica.

Cuando hablamos de una zona submarina, a este conocimiento se le denomina Batimetría, mientras que si estamos hablando de una zona con una elevación por encima del nivel del mar, se le denomina topografía.

En este documento nos vamos a centrar en la batimetría, como podemos usar una batimetría conocida y paquetes software que podemos instalar para leer determinadas cartas batimétricas.

5.8.1 ¿Para qué la batimetría en este proyecto?

Aunque la batimetría no va a ser esencial en este proyecto, si que puede jugar un determinado papel a la hora de planificar misiones, puesto que podremos conocer de antemano si una ruta planificada por el usuario puede ser posible realizarla en la realidad.

Por lo tanto, la batimetría en este proyecto se puede utilizar tanto para mostrar al usuario la situación geográfica por la que está planificando la ruta, como para comprobar posteriormente en la validación de la misión si realmente el AUV puede navegar por una determinada ubicación con la profundidad establecida por el usuario.

5.8.2 Formatos actuales para la batimetría

A continuación se muestran una serie de formatos que se usan actualmente para la batimetría.

5.8.2.1 Formato HDF

El formato HDF(Hierarchical Data Format) **[HDFGROUP]** fue desarrollado por la NCSA (National Center for Supercomputer Applications). Éste es un formato autodescriptible, y de libre distribución.

Soporta interfaces en C y en Fortran, además se trata de un formato de datos común sobre el que pueden aplicarse muchas interfaces.

De este formato existen dos versiones:

- **HDF4**: Éste es un formato de datos basado en relaciones jerárquicas entre los datos y las dependencias entre los mismos.

- **HDF5**: Éste formato suministra un modelo de datos mucho más rico que el anterior, con jerarquización, énfasis en la eficiencia del acceso, entrada salida paralela, y soporte para computación de alto rendimiento. En este formato, una representación batimétrica consistiría en tres vectores:

    - ❖ Uno que incluyera las profundidades para cada nodo de la maya

    - ❖ Una imagen de la maya

    - ❖ Una paleta de colores.

Estos dos formatos no son compatibles entre sí, ya que son completamente distintos.

5.8.2.2 Formato CDF

Este formato fue el primero en aparecer. El formato CDF (Common Data Format) es un formato abstracto para matrices multidimensionales auto-descriptivas. Fue diseñado en el NASA/Goddar Space Flight Center. Introdujo la idea de la abstracción de datos para almacenamiento, manipulación, y acceso de datos multidimensionales. Además, también introdujo la idea de la independencia con la máquina.

Más tarde surgiría el formato netCDF, el cual es descrito en la siguiente sección. Aunque sus nombres poseen cierta similitud, no son compatibles, aunque CDF ha ido añadiendo características propias del formato netCDF, como pueden ser el acceso de datos, la representación XDF, la representación en un fichero simple, atributos para variables, etc.

5.8.2.3 Formato netCDF[NETCDF]

Este formato es el que se ha escogido para el proyecto. Por lo tanto es el formato que más se va a detallar.

Una de las ventajas más destacables del formato netCDF es que posee librerías de acceso para C, Fortran, C++, Java, Perl y otros lenguajes. Las cinco características que mejor describen este formato son:

- ❖ **Auto-descripción**: el mismo fichero netCDF contiene información que describen los datos que contiene.

- ❖ **Portable**: netCDF usa una forma extendida del formato XDR para la representación de la información contenida en la cabecera y parte de datos de los ficheros. Éste es un estándar para la codificación de los datos y una librería de funciones para la representación externa de datos. El

usuario, haciendo uso de este estándar, podrá codificar sus estructuras de datos de forma independiente a la máquina.

- **Acceso directo**: Es un formato orientado a vectores, lo que nos proporciona la capacidad de acceder directamente a los datos, sin tener que recorrer secuencialmente el fichero. Además, permite el acceso a vectores enteros o trozos de los mismos como una unidad y de manera directa.

- **Ampliable**: Los datos pueden ser añadidos a un fichero netCDF sin modificar la estructura del mismo.

- **Compartible**: Varios lectores pueden acceder concurrentemente a la vez que accede un escritor a un fichero con este formato.

- **Compatible**: Se garantiza que cualquier versión futura de este formato será compatible con las versiones anteriores.

5.8.2.4 Modelo de datos.

Un fichero netCDF estará compuesto de dimensiones, variables y atributos. Todos tienen un nombre y número de identificación para poder acceder a ellos. Además, los ficheros netCDF contienen una tabla de símbolos para las variables que contienen su nombre, tipo de datos, dimensiones, número de dimensiones, y dirección de disco de comienzo.

A continuación se explica en qué consisten cada uno de los tres conceptos mencionados.

Dimensiones:

Las dimensiones representan una dimensión física real en el mundo científico. Una dimensión tiene un nombre y una longitud. La longitud puede ser un número positivo (que indicará el tamaño de las variables o vectores que usará el dataset) o el descriptor “Unlimited”, que indicaría que el vector puede ser de un tamaño variable.

Variables:

Es un vector de valores del mismo tipo. Poseen un nombre, tipo de datos, forma (o lista de dimensiones que lo conforma) y atributos opcionales que lo describen y que pueden ser añadidos o cambiados tras la creación de la variable o durante la misma.

Los tipos de datos pueden ser: NC_BYTE, NC_CHAR, NC_SHORT, NC_INT, NC_FLOAT, NC_DOUBLE y NC_LONG.

Existen dos tipos de variables:

- **Variables primarias**: contienen los datos del dataset o la información relevante. Si poseen alguna dimensión ilimitada, se le llama *Variable de Registro*.

- **Variables de coordenadas**: tienen el mismo nombre que una dimensión (y solo contiene a dicha dimensión). Definen una coordenada física correspondiente a dicha dimensión.

Atributos:

Almacenan datos relativos a datos o variables. Hay dos tipos de atributos, los atributos globales que se nombrarán con un único nombre, el del atributo, y los atributos que definen una variable, que deben denominarse con el nombre de la variable junto con el nombre del atributo.

Los atributos pueden ser borrados, asignárseles un tipo, longitud, y cambiar sus valores, después de creados, mientras que una variable no puede ser modificada tras su definición.

Organización de los ficheros:

Una base de datos o Dataset en netCDF es almacenada en un solo fichero con dos partes:

- **Una cabecera**: Contiene información de dimensiones, atributos y variables.

- **Datos**: la parte del fichero que contiene lo datos consta de dos partes: compresión de los datos pertenecientes a variables no ilimitadas (Variables de Coordinación) y variables ilimitadas (*Variables de Registro*).

Los ficheros que contienen este tipo de dataset contienen extensiones ***.nc**. Este tipo de ficheros está comprimido, por lo tanto están en binario, no siendo legible por el usuario.

5.8.2.5 Tipos de datos externos

El formato netCDF soporta los siguientes tipos de datos externos:

char	8-bit para la representación de caracteres
byte	8-bit con o sin signo
short	16-bit enteros con signo
int	32-bit enteros con signo
float or real	32-bit IEEE punto flotante
double	64-bit IEEE punto flotante

5.8.2.6 Acceso de Datos

Para acceder a los datos que nos proporciona un fichero en formato netCDF, debemos especificar el fichero del que se desea extraer los datos, la variable netCDF de la que se quiere extraer y un rango, que será el rango que se desea extraer de la variable.

A continuación se listan todos los tipos de accesos a los datos:

- Acceso a todos los elementos de la variable. Es decir, accederemos a todos los elementos del vector.

- Acceso a elementos individuales de una variable. Para ello hará que indicar con un índice qué elemento de la variable es el que necesitamos.

- Acceso a secciones del vector. Para ello tendremos que especificar un vector de índices (que irán desde la posición más cercana al primer índice del vector hasta el más lejano) y un vector de contadores (que indican, para cada uno de los índices pasados en el primer vector el tamaño de la sección a partir del índice que queremos coger).

- Acceso a muestreo de secciones del vector, que es similar a un acceso a secciones del vector, pero especificando un único índice, un contador, y el stride o paso, es decir, cada cuanto se toma un valor. Por ejemplo, si especificamos un índice i, un paso n y un contador c, tomará los datos que se encuentren en i, i+n, i+2n, i+3n, hasta que n=c-1.

- Acceso a secciones mapeadas del vector, donde se especifica, vector, contador, paso, e índice de mapeo del vector, es decir, similar al de muestreo, pero con el índice que indica como los valores asociados a la variable son almacenados en memoria (un desplazamiento respecto al origen de cada valor accedido). Los datos accedidos son los mismos que en el muestreo de secciones del vector.

5.9 Análisis de los algoritmos de planificación

5.9.1 Descripción del problema.

En este apartado se introducirá al problema que supone realizar la planificación de la ruta que ha de seguir un AUV.

Para el proyecto del planificador de misiones, se debe dejar definir por parte del usuario una serie de rutas a seguir por el submarino. Estas rutas van a venir definidas de diferentes formas:

- El usuario definirá el orden en el que tendrá que recorrer los puntos. Éste es el caso más simple, ya que se le impondrá al planificador la forma en la que deberá recorrer el submarino los puntos del mapa.

- El usuario define una serie de puntos de control o waypoints en un mapa por los que debe pasar el submarino. En este caso el usuario no definirá

el orden de recorrido de los puntos. Lo que definirá son unas prioridades de cada punto. Es decir, unos waypoints tendrán mayor prioridad que otros. Por lo tanto, el submarino deberá recorrer éstos waypoints antes de recorrer los waypoints de menor prioridad. Esta será la planificación de mayor complejidad.

- También podremos tener una mezcla de los dos, en el que el usuario define para unos puntos el orden en el que se deberá recorrer (por ejemplo, llegado al waypooint nº 5 ir al waypoint nº7) y dejar a la aplicación decidir el orden en el que se van a recorrer los demás puntos.

El mapa además, contendrá una serie de polígonos bidimensionales que le indicarán al usuario zonas prohibidas para el submarino, debido al alto riesgo de colisión, etc.

El usuario nos planteará un punto inicial y otro final, los cuales son de paso obligatorio para el submarino.

El usuario no tendrá que preocuparse de las restricciones no holométricas, supondremos que el submarino es un robot holométrico, por dos motivos. El primero es que según las especificaciones del planificador, a priori no podemos conocer las características de cada submarino. Una planificación deberá ser válida para distintos modelos de submarinos.

El segundo motivo es que el mapa sobre el que trabajaremos tendrá, relativamente, baja resolución, puesto que tendremos que representar en una ventana de la interfaz un mapa que podría comprende varios kilómetros cuadrados de área. Debido a esto, además podremos suponer que el submarino es un punto, con lo que simplificaremos el algoritmo.

5.9.2 Representación de los distintos elementos en el mapa.

En el caso del planificador de trayectorias, el submarino se representará como un punto en el mapa. El punto inicial y final y los puntos intermedios por los que tendrá que pasar el submarino durante la ejecución de su trayectoria también serán representados por puntos.

Tendremos en el mapa una serie de obstáculos, que en principio podrán ser representados mediante una "mancha" en el mapa, un conjunto de "píxels" en el mapa por los que no deberá pasar el submarino.

Pero para llevar a cabo la discretización del espacio de estados, como veremos más adelante, necesitaremos tener los obstáculos representados por polígonos. Por ello es conveniente tener representados los obstáculos del mapa como polígonos.

Para llegar a esa representación de los obstáculos, podemos implementar un algoritmo con el que podamos obtener el mínimo polígono convexo que inscriba todos los puntos del obstáculo.

Existen ya algoritmos definidos que realizan esta función. Uno de esos algoritmos es el "***algoritmo de Graham***" (Ver **[RADO03]**, Capítulo 5.2). Este

algoritmo cogerá un conjunto de puntos Q (donde el número de puntos incluidos en Q es mayor o igual a 3) y producirá un polígono P que los incluya a todos. De acuerdo con este algoritmo, cada punto de Q es introducido en una pila y los puntos de los vértices son excluidos de la misma. Cuando la ejecución termina, la pila S contiene los vértices del polígono, o sea, el conjunto P.

Con esto habremos solucionado el problema de la representación de los obstáculos.

5.9.3 Discretización del espacio

Hay varias formas de discretizar el espacio. Lo que se busca en este algoritmo es escoger la mejor forma de discretizar este espacio para optimizar la autonomía del submarino. Se busca recorrer el menor espacio posible entre dos waypoints fijados por el usuario.

Las posibles alternativas a la discretización del espacio de estados son:

- Por Puntos: como pueden ser los grafos de Visibilidad o los grafos de Tangentes.

- Por Discretización del espacio: utilizando descomposiciones del espacio en estructuras como pueden ser el Quad-Tree, el BSP-Tree, o una simple discretización en píxeles del mismo tamaño cada uno.

- Por retracción del espacio: Definiendo un subespacio teniendo en cuenta el espacio de evolución y garantizando la continuidad. Un ejemplo de discretización del espacio de esta forma pueden ser los diagramas de Voronoi.

El submarino debe tener la mayor autonomía posible para que pueda cumplir con la mayor parte del plan posible. Además, tendremos que planificar una ruta del submarino sobre un mapa con poca resolución (obstáculos muy grandes y podemos considerar al submarino como un punto en el espacio bidimensional del mapa).

Además, debemos tener en cuenta que podemos suponer que tenemos un robot submarino holométrico, ya que cada píxel del mapa representará una amplia zona real, es decir, cada punto del mapa representará un espacio que perfectamente podría tener varios metros cuadrados de área.

A continuación se explica en que consiste cada una de las formas de discretizar el espacio que hemos nombrado y cuales son las ventajas y desventajas de cada uno de ellos.

5.9.3.1 Grafos de Visibilidad

En este tipo de discretización espacial, los obstáculos están definidos mediante polígonos. Se introduce el concepto de *visibilidad*, por el cual desde un nodo solo es

visible otro nodo si se puede trazar entre ellos una recta que no intersecte con ningún obstáculo.

Se puede considerar a otro nodo como visible, si la recta trazada desde el actual nodo a ese otro nodo toca de forma tangencial un polígono. Por ejemplo, cada uno de los vértices que conforman las aristas de los polígonos serían visibles entre sí si pertenecen a la misma arista.

Si tenemos el punto inicial y el final, los nodos que conformarán el espacio de estados serán estos dos puntos además de los vértices de los polígonos que representan a los obstáculos.

El siguiente paso del algoritmo será unir cada uno de los nodos a los nodos que son visibles desde ellos. Una vez hecho esto, ya tendremos todas las posibles trayectorias a seguir para llegar desde el nodo inicial hasta el final.

La forma en la que seleccionaremos la trayectoria a seguir será mediante el algoritmo que resuelve el problema del "pathfinder", el algoritmo A*. Este algoritmo nos ayudará a seleccionar la trayectoria con el menor costo para llegar desde el punto inicial al final. Otras alternativas a éste algoritmo son el D*, el algoritmo de Dijkstra o el algoritmo de Floyd **[Floyd]** (para éstos dos últimos necesitamos que los grafos sean dirigidos).

Ventajas de los grafos de visibilidad en el planificador de trayectorias para el AUV:

- Es un método que me permitirá llegar desde un waypoint al siguiente que tenga que recorrer el submarino evitando los obstáculos recorriendo la trayectoria mínima posible para ello.

- Es un método simple que encaja con las necesidades del proyecto, ya que los obstáculos serán representados mediante polígonos, que es un método válido para representar los obstáculos en el planificador.

Desventajas de los grafos de visibilidad aplicados en el planificador de trayectorias para el AUV:

- Los vértices de los obstáculos serán nodos por los que pasará el submarino, lo que conlleva un cierto riesgo si el polígono del obstáculo se ha ceñido al borde del obstáculo. Además, el submarino tendrá una estimación de su posición, es decir, que quizás podría introducirse en el área del obstáculo si no estima bien su posición (algo que seguro pasará).

- Es un algoritmo demasiado pesado, ya que, para calcular el grafo de visibilidad, tendremos que calcular los segmentes de cada vértice con

todos los demás vértices de los obstáculos, para después buscar las intersecciones entre todos los segmentos que hayamos obtenido como resultado. Si se producen intersecciones en el segmento establecido entre el vértice del que partíamos y el vértice destino, supondremos que los vértices no son visibles.

Esto puede ser muy peligroso para el submarino. Una posible solución sería el engrosamiento de los obstáculos. Antes de aplicar el grafo de visibilidad aplicaremos un algoritmo de engrosamiento de acuerdo con el error de estima del submarino que creemos que pueda tener.

5.9.3.2 Diagramas de Voronoi

Este método requiere para su correcto funcionamiento de un entorno conocido y de unos obstáculos poligonales. Los obstáculos, como hemos visto antes, conseguimos que sean poligonales mediante el algoritmo de "Graham".

Lo que se intenta con éste método es que el robot pase por una ruta marcada que sea equidistante a los dos obstáculos más próximo. Lo que se conseguirá con esto es conseguir trayectorias en las que el robot pase lo más lejano posible de los obstáculos.

El esquema está formado por dos tipos de segmentos: rectilíneos y parabólicos. Los segmentos rectilíneos se ubicarán en los lugares geométricos que se encuentren situados entre dos aristas de dos obstáculos distintos. En cambio, los segmentos parabólicos se utilizarán en el lugar geométrico en el que se hallan equidistantes un vértice de un obstáculo y una arista del otro obstáculo.

Ventajas de este esquema para el planificador:

La principal ventaja es que es un esquema que nos permite establecer las trayectorias que seguirá el submarino de la forma más distante a los obstáculos posible. De esta forma el submarino correrá el menor peligro posible.

Es un método válido para el proyecto, ya que necesita que los obstáculos estén representados como polígonos, y esa parte la tenemos resuelta.

Desventaja de este esquema para el planificador

Para un robot submarino lo que se busca, entre otras cosas, es que tenga la mayor autonomía posible, para que pueda recorrer la mayor distancia posible. Con este esquema se evita que el submarino corra peligro, pero se incrementa la distancia a recorrer ya que se busca evitar lo máximo posible los obstáculos. El robot se podría desviar bastante de su trayectoria. El ejemplo que se muestra en la Ilustración 18 es una aproximación de lo que podría una trayectoria determinada mediante este método:

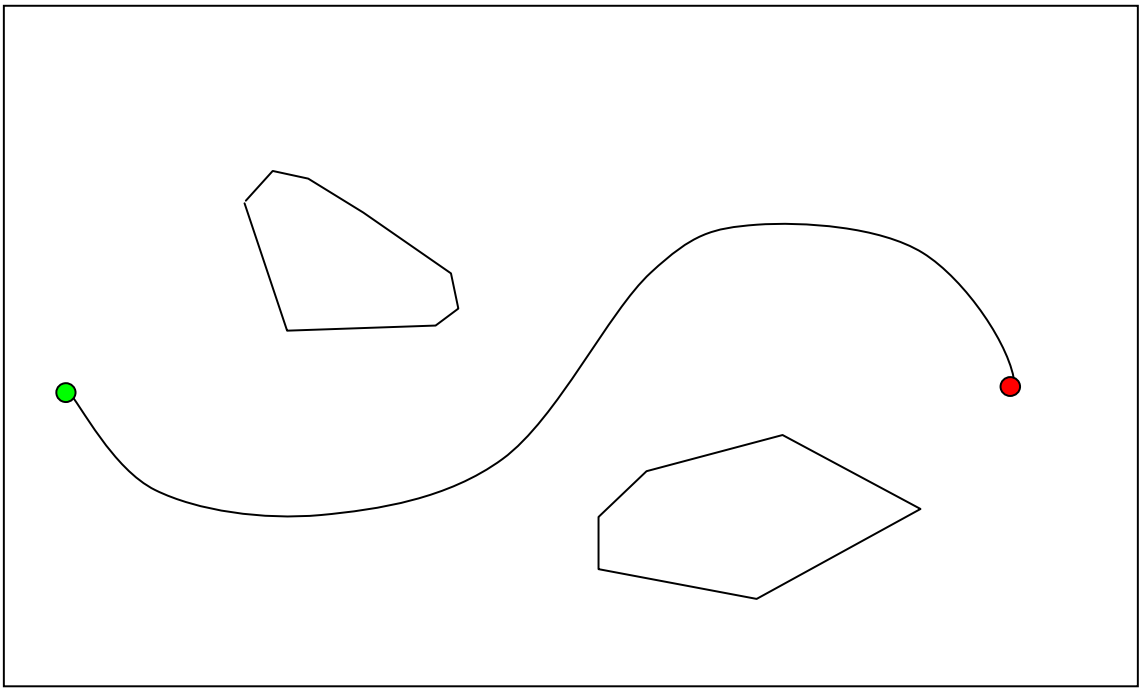


Ilustración 18: Ejemplo de diagrama de Voronoi

Como vemos, una trayectoria que bien podría seguirse en línea recta (en el caso de representar al robot como un punto), mediante este método obtendríamos una trayectoria en zigzag que llevaría al robot a recorrer un mayor camino innecesariamente.

Si hablamos de un submarino autónomo, con esta trayectoria estaríamos limitando considerablemente la autonomía del submarino, siguiendo una trayectoria que no estaría lo suficientemente justificada debido a las dimensiones del mapa comparadas con las dimensiones del submarino.

5.9.3.3 Descomposición en celdas

El esquema de descomposición en celdas se basa en dividir el espacio libre en una serie de celdas. Con ello, la búsqueda de una ruta desde una posición inicial a una final se basa en buscar las celdas consecutivas que se encuentren vacías y que en la celda inicial se encuentre la posición inicial del robot y en la celda final se encuentre el objetivo del robot.

Para aplicar este tipo de algoritmos habría que resolver dos problemas:

- Forma de descomponer el espacio libre en celdas
- Construcción de un grafo de conectividad.

Hay distintas formas de descomponer el espacio libre en celdas, y todas tienen sus pros y sus contras. A continuación se muestran algunas.

5.9.3.3.1 División en celdas iguales

Una de las formas más sencillas consiste en dividir el espacio en celdas del mismo tamaño. A cada una de las celdas le asignaremos el estado ocupado o vacía, de forma que al realizar la planificación de la trayectoria tendremos que comprobar celda por celda si está libre u ocupada.

El principal inconveniente que tiene este algoritmo es que hay que ir celda por celda comprobando si está ocupada o vacía, y, dependiendo de la precisión con la que se haya dividido el mapa, se tendrán un número mayor o menor de celdas en el mapa.

Como vemos, es fácil dividir el espacio de estados en celdas iguales pero es complicado, o costoso, realizar el grafo de conectividad debido al considerable número de los posibles nodos candidatos.

Otra desventaja de este tipo de subdivisión, es que mientras más resolución tengamos, más espacio ocupara el mapa en memoria.

5.9.3.3.2 División mediante Octrees o Cuadtrees

Otra opción es dividir el espacio libre mediante los conocidos métodos de cuadtree y octree. Con divisiones del espacio libre de este tipo, tendremos la solución a la cantidad de candidatos a nodos de los grafos que teníamos en el apartado anterior.

En este caso, solo habrá una mayor resolución donde sea realmente necesario puesto que haya un obstáculo. En el espacio libre las celdas tendrán el mayor tamaño posible.

El grafo se calculará mediante todas las celdas vacías que contenta el mapa.

5.9.3.3.3 Descomposición trapezoidal

Esta descomposición es una descomposición también bastante simple del espacio libre. Consiste en una serie de segmentos paralelos al eje Y en los que cada uno de esos segmentos se encuentra en una posición del eje X en la que hay un vértice.

Habrá tantas celdas como vértices haya más uno. La forma de establecer un grafo en este tipo de esquemas es bastante simple, ya que se escogerá el punto central de cada segmento, teniendo en cuenta que cada segmento vertical comienza o acaba en un vértice. El vértice dividirá en dos el segmento paralelo al eje Y.

Como vemos esta descomposición es bastante sencilla de llevar a cabo. Tanto en la descomposición del espacio libre como la generación del grafo asociado a él es un proceso bastante simple.

5.9.3.4 Algoritmo BUG1

Ver bibliografía **[MOBOT]**. Este algoritmo de evitación de obstáculos es muy sencillo, ya que lo único que tenemos que hacer es bordear el obstáculo por un lado u otro.

Si tenemos un punto origen, p0, un punto final, p1, uno o varios obstáculos entre estos dos puntos, y una lista de caminos S, el algoritmo funcionaría de la siguiente forma:

1) Se busca el obstáculo más cercano al punto p0.
2) Se busca el vértice más cercano del obstáculo al punto p0. Lo llamaremos v0.
3) Se traza una trayectoria desde el punto p0 hasta el vértice v0 y se almacena en al lista S.
4) Seguidamente, se recorre el obstáculo en un sentido o en otro (horario o antihorario) añadiendo los caminos entre los vértices a la lista S.
5) Se deja de recorrer el obstáculo cuando se llega al vértice más cercano al punto objetivo p1. Denominaremos a este vértice el v1.
6) Se comprobará que no hay ningún otro obstáculo entre el vértice v1 y el punto p1.

    a. En caso de que lo haya, se vuelve al inicio del algoritmo, con el v1 como p0, y el p1 sigue siendo el punto objetivo.

    b. En caso de que no haya más obstáculos en medio, se agregará el camino entre el vértice v1 y el puno p1 a la lista S y se finalizará la ejecución del algoritmo.

A continuación se puede ver un ejemplo de evitación de obstáculos en el que tendremos dos obstáculos entre el punto inicial y el final.

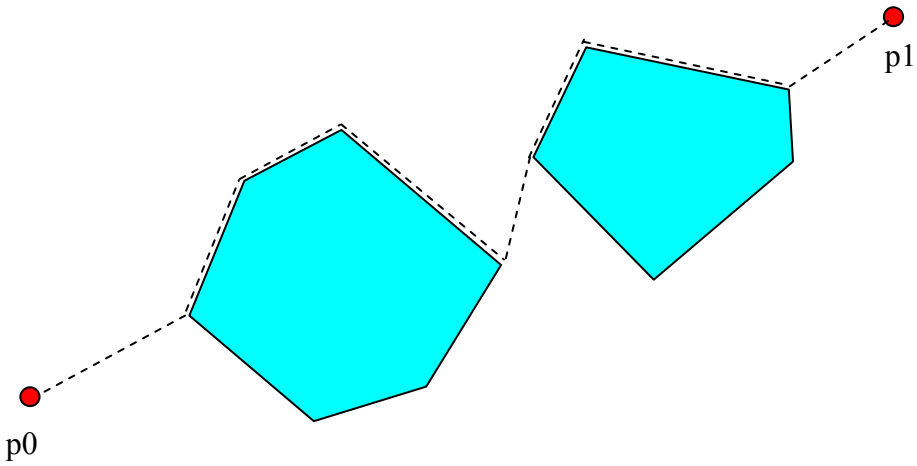


Ilustración 19: Algoritmo BUG1

Como se puede ver, no es una forma eficiente de evitar obstáculos. Se pueden dar casos extremos en los que el submarino tenga que recorrer mucho más camino del que debería debido a que este algoritmo no es óptimo en este sentido.

Definitivamente se ha implementado este algoritmo en la aplicación. Aunque no se ha implementado exactamente como se define aquí, sino que se ha añadido una pequeña modificación.

La modificación consiste en que se bordeará el polígono hasta que se llegue a un vértice desde el que es visible el punto final. Para comprobar si es visible el punto p1 no se tendrán en cuenta los demás obstáculos, solo el que estamos bordeando. Es decir, bordearemos el obstáculo hasta que lleguemos a un vértice desdel que podamos llegar hasta el punto p1, si no tenemos en cuenta los demás obstáculos que hay en el mapa.

Una vez llegados a ese punto comienza el algoritmo de nuevo en el caso de que haya más obstáculos en medio.

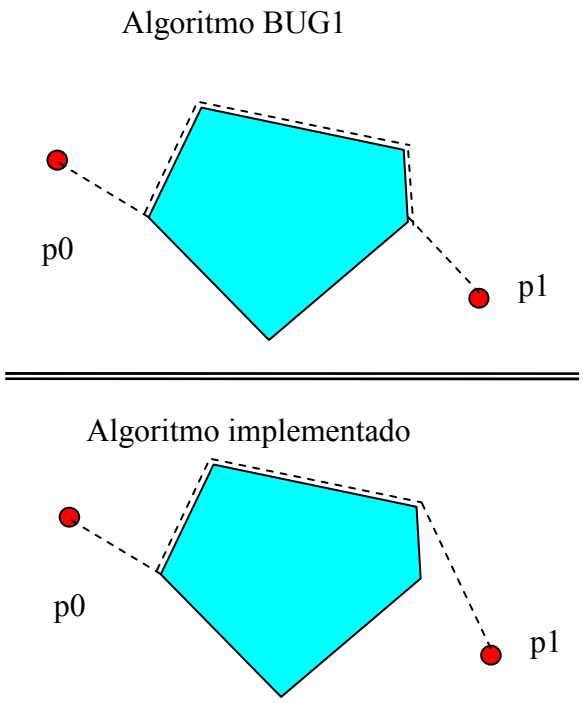


Ilustración 20: Algoritmo implementado (1)

Al algoritmo implementado, comparado con el BUG1, parece ser un poco más óptimo. En la Ilustración 21 se compara con el ejemplo visto anteriormente.

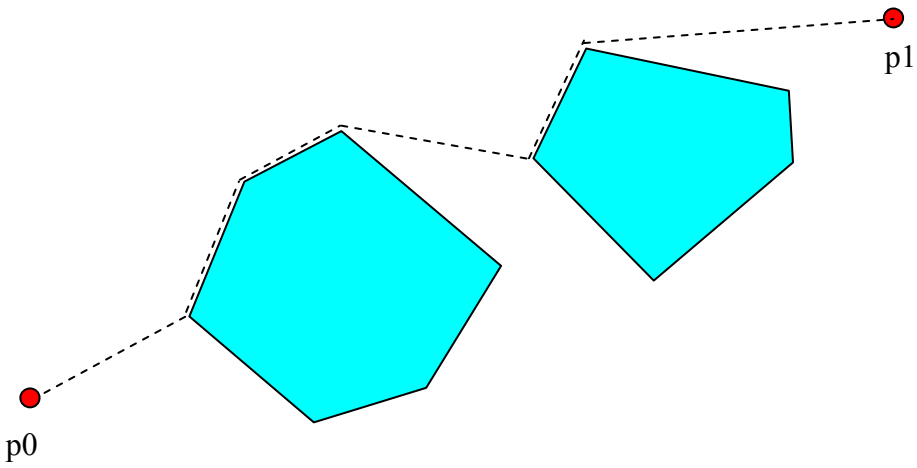


Ilustración 21: Algoritmo implementado (2)

5.9.3.5 Algoritmo BUG2

Este algoritmo es una variante del algoritmo BUG1. En este caso, no iremos de vértice en vértice del polígono, sino que bordearemos el polígono hasta encontrar la zona en la que se cruza con el segmento dibujado entre el punto inicial y el final.

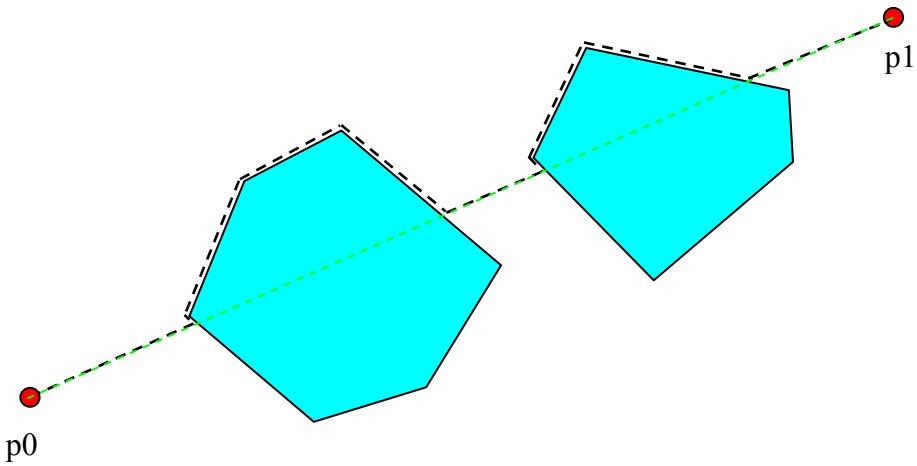


Ilustración 22: Algoritmo BUG2

Para ir dese p0 hasta el primer obstáculo, determinaremos con el algoritmo que calcula las intersecciones, el punto del polígono en el que intercepta el segmento p0p1 al primer polígono. Se añadirá a la lista de caminos esta trayectoria y se rodeará el polígono por un lado o por el otro. Hasta llegar a la intersección más lejana a p0 con el segmento p0p1, donde comenzará de nuevo el algoritmo si hay otro obstáculo en medio.

5.9.3.6 Algoritmo VISBUG

Este algoritmo será similar al anterior, el algoritmo BUG2, con la diferencia que se engrosarán los vértices de los polígonos que representan al obstáculo.

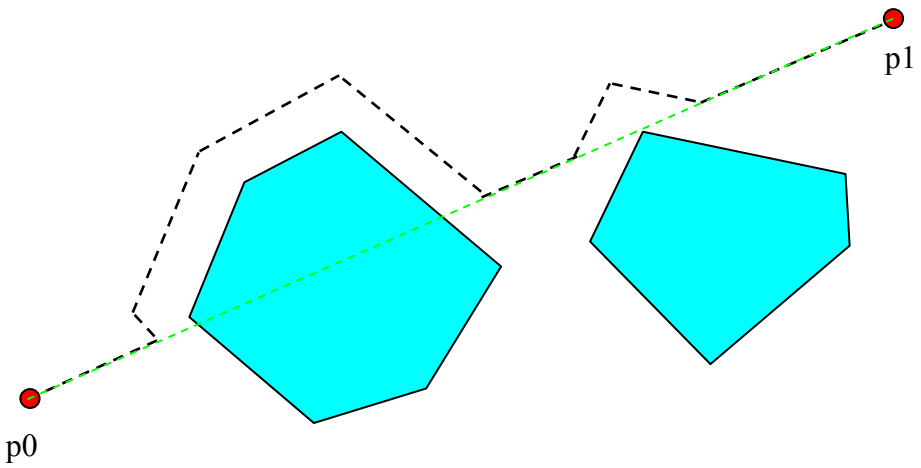


Ilustración 23: Algoritmo VISBUG

Como se puede ver, el robot pasará a una distancia preventiva de los obstáculos, con el fin de evitar posibles colisiones.

5.9.3.7 Conclusión

Hemos visto varias formas de establecer el grafo para evitación de obstáculos. Debemos tener en cuenta que para el planificador del submarino lo que necesitamos es que la trayectoria sea la mínima posible y que además el planificador sea capaz de realizar la evitación de obstáculos de forma que el submarino corra el menor riesgo posible.

Esto nos lleva a pensar que quizás, la mejor forma de representar el espacio de estados es mediante un grafo de visibilidad. Con estos grafos de visibilidad podremos definir algorítmicamente una serie de puntos mediante los cuales podremos llegar de un punto del espacio a otro evitando obstáculos de manera que el submarino recorra el menor espacio posible para rodear los posibles obstáculos que nos podamos encontrar en medio.

El principal problema de este algoritmo sería su poca eficiencia al generar el grafo de visibilidad, ya que hay que calcular las intersecciones entre todos los posibles segmentos que se puedan dar en la ruta, entre los waypoints, entre los waypoints y los vértices de los obstáculos, y entre los propios vértices de los distintos obstáculos.

Por ello, el algoritmo que se ha escogido para implementar, por su eficiencia y su simplicidad, es el algoritmo BUG1.

5.9.4 Algoritmos para el cálculo de intersecciones.

Para comprobar si existen colisiones entre el camino del robot y los obstáculos que hay en el plano, existen varias formas de implementar la comprobación de las intersecciones de los segmentos que conformas las aristas de cada uno de los polígonos que representan a los obstáculos.

Además, hay que tener en cuenta que si escogemos el grafo de visibilidad a la hora de implementar la discretización del espacio de estados, tenemos que comprobar desde cada uno de los vértices de los polígonos los nodos a los que se puede conectar, es decir, los nodos que son visibles desde él.

Por ello la carga computacional del algoritmo deberá ser la mínima posible. La forma más sencilla de comprobar qué segmentos se intersectan en el mapa, es comprobar segmento con segmento cada uno de los demás segmentos que integran el mapa. En ese caso, la carga computacional será máxima, y podrá llegar a ser del orden de $O(n^2)$.

La forma de implementar este algoritmo es la siguiente:

Para cada uno de los segmentos, debemos comprobar con todos los demás segmentos del plano. La forma de comprobar que dos segmentos se intersectan es calculando sus ecuaciones de las rectas, en forma paramétrica, e igualando los puntos de las rectas, ya que tenemos que encontrar qué punto tienen en común las dos rectas. Las fórmulas son las siguientes:

Sabiendo que V es el vértice de la recta y que D es el vector, la fórmula de la recta en forma paramétrica es la siguiente:

$$P_1 = V_1 + \alpha D_1$$
$$P_2 = V_2 + \beta D_2$$

para la recta 1 y para la recta 2. Lo que tenemos que igualar a continuación son los puntos P_1 y P_2, para ver que punto tienen en común las dos rectas. El resultado sería algo como lo siguiente:

$$P_{1x} = V_{1x} + \alpha D_{1x} \quad P_{2x} = V_{2x} + \beta D_{2x}$$
$$P_{1y} = V_{1y} + \alpha D_{1y} \quad P_{2y} = V_{2y} + \beta D_{2y}$$

Igualando las ecuaciones obtendremos:

$$V_{1x} + \alpha D_{1x} = V_{2x} + \beta D_{2x}$$
$$V_{1y} + \alpha D_{1y} = V_{2y} + \beta D_{2y}$$

$$V_{1x}D_{1y} - V_{1y}D_{1x} = V_{2x}D_{1y} - V_{1y}D_{1x} + \beta(D_{2x}D_{1y} - D_{2y}D_{1x})$$

Con esto ya podemos calcular la β, si no son paralelas las rectas, con la que finalmente sustituiremos en la expresión de P_2 y hallaremos definitivamente el punto de intersección entre las dos rectas.

Una vez encontrado ese punto, debemos comprobar si pertenece a los segmentos que se están estudiando. Si pertenece a los dos segmentos, la intersección existe y se produce en ese punto encontrado. En caso contrario no existe intersección.

Si aplicamos este algoritmo a todos los segmentos del plano, puede llegar a ser muy costoso. Por ello hay muchos estudios de distintas implementaciones que nos permiten comprobar las intersecciones de los segmentos de un plano llegando a conseguir una complejidad menor de $O(n^2)$.

Uno de esos algoritmos es el algoritmo "sweep plane". Éste algoritmo intenta calcular tan solo las intersecciones de los segmentos en los que haya una posible intersección, no en todos los segmentos.

Éste algoritmo se explica a continuación.

5.9.4.1 Algoritmo "sweep plane"

Se tiene un conjunto de segmentos S que se van a proyectar sobre el eje y, pudiéndose afirmar que existen varios casos:

- Dos segmentos de recta S_i & S_j se intersectan sobre su proyección en el eje *y*, y que también se intersectan en el plano.

- Las proyecciones en *y* de dos segmentos S_i & S_k no se intersectan. En este caso tampoco habrá intersección en el plano.

- Dos proyecciones de los segmentos S_k & S_l se intersectan en y pero no se intersectan en el plano.

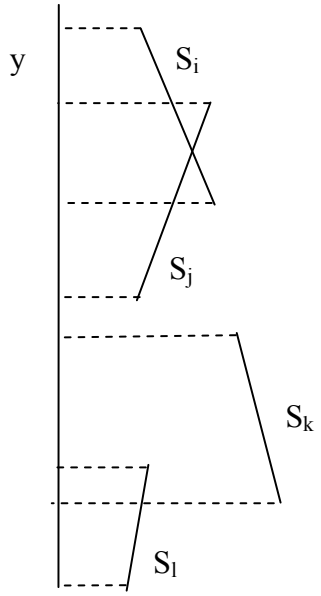


Ilustración 24: Intersecciones

El algoritmo utiliza una línea L de barrido para ejecutar un barrido entre los puntos de mayor y menor coordenada en *y*, comenzando a barrer en el punto de mayor coordenada. Para eso existe una lista de estado T, la cual contiene, a medida que se ejecuta el barrido, los segmentos que interceptan la línea L en la posición en la que esa se encuentra. Se define una acción como un acontecimiento capaz de alterar la lista de estado T. De ese modo, se definen dos acciones distintas:

- Cuando se encuentra el punto con mayor *y* de un segmento, se añade ese segmento a la lista de estado T, y se testea si hay intersecciones entre ese nuevo segmento añadido y los demás segmentos de la lista de estado T.

- Cuando se encuentra el punto con menor coordenada *y* de un segmento, este segmento se quita de la lista de estado T.

Si analizamos este algoritmo nos daremos cuenta de que el número de intersecciones que tienen que ser verificadas puede ser menor que el número de intersecciones que se tenían que verificar en el método exhaustivo.

Aunque si nos ponemos en el peor caso, el número de intersecciones que tendríamos que verificar sería igual al número de intersecciones que comprobábamos con el método exhaustivo.

Por ello, se propuso introducir una modificación al algoritmo. La modificación consistía en, dado un segmento S_i, testear la intersección con los segmentos contenidos en la lista T que se encuentren horizontalmente adyacentes a ese segmento. Es decir, que se ha de testear las intersecciones tan solo con el segmento que se encuentre a la derecha y el que se encuentre a la izquierda de ese segmento S_i.

El algoritmo será ahora capaz de realizar tres acciones:

- **acción tipo I**: Punto de segmento S_i con menor coordenada en *y*.

  Esta acción obliga a la extracción del segmento S_i de la lista de estado T, necesitando a posteriori una verificación de intersección entre los segmentos horizontalmente adyacentes, es decir, la verificación de la intersección entre los segmentos S_l y S_r que se ven en el ejemplo. En caso de que el punto de intersección p' se encuentre debajo, relativamente al eje vertical, del punto de acción p(lo que no pasa en el ejemplo que se muestra a continuación), entonces p' pasa a ser un nuevo punto de acción de intersección tipo III, que veremos más adelante.

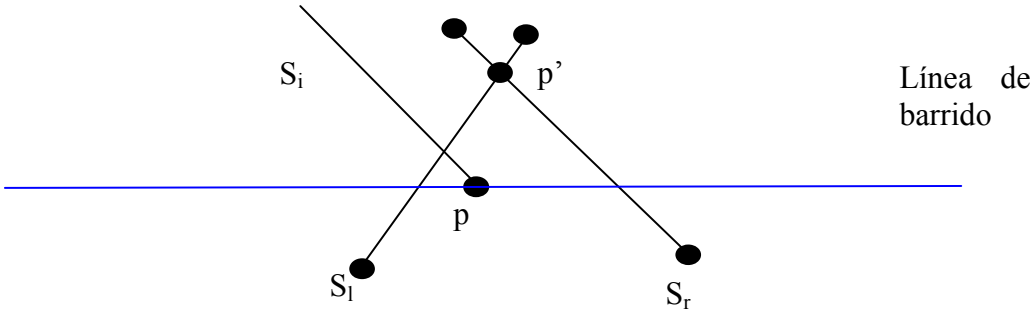


Ilustración 25: Acción tipo I

- **acción tipo II**: Punto de segmento S_i con mayor coordenada *y*. En este caso hay que insertar el segmento S_i en la lista de estado T y comprobar las intersecciones entre los segmentos adyacentes. Al igual que con el anterior, si hay un punto de intersección por debajo de la línea de barrido (tal y como pasa en el ejemplo mostrado a continuación), entonces, habrá que añadir este nuevo punto de intersección como un punto de acción de intersección de tipo III.

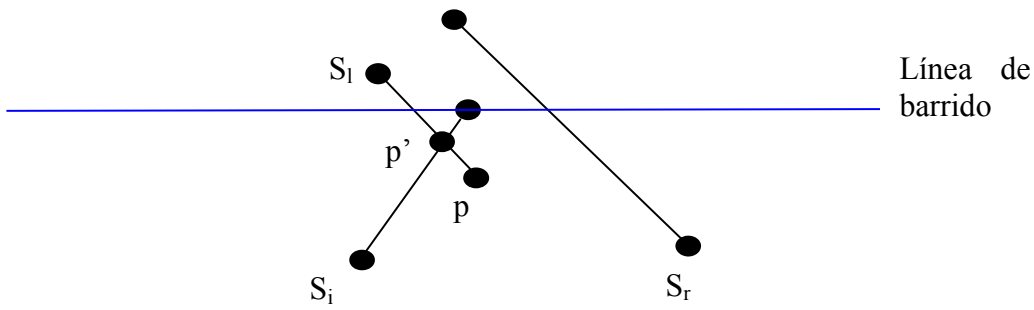


Ilustración 26: Acción tipo II

- **acción de tipo III**: Punto de intersección entre dos segmentos S_i y S_j. Este tipo de acción se ejecuta cuando hay una intersección entre dos segmentos. En el ejemplo se ve como los segmentos S_i y S_j producen una intersección en el punto p. Éste punto p se ha introducido en la lista de punto de acción previamente, cuando se ha insertado uno de los segmentos en la lista de estado T y se ha detectado esta intersección. Cuando la línea de barrido alcanza este punto, se lanza una acción en la que se comprueba las intersecciones del segmento S_j y el S_l (en el ejemplo mostrado a continuación) y la intersección entre el segmento S_i y el S_r. En caso de que haya intersecciones por debajo de la línea de barrido entre estos segmentos, esas intersecciones pasan a ser futuras acciones de intersección.

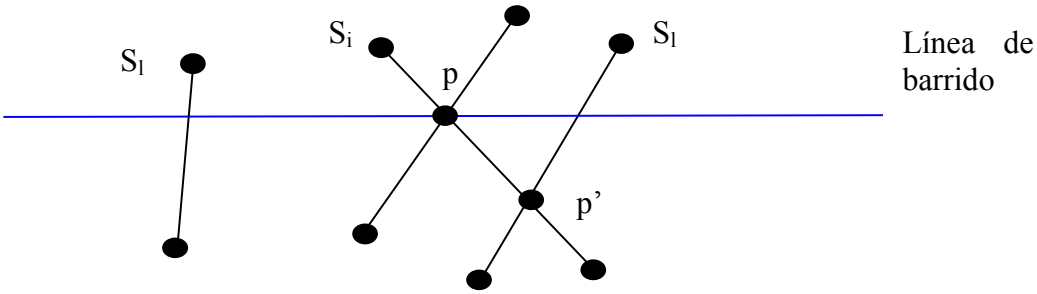


Ilustración 27: Acción de tipo III

Para esta última modificación apuntada, necesitaremos tener otra lista. Una lista que almacene las acciones a ejecutar y en la que vayamos añadiendo los puntos límite de los segmentos y los puntos de intersección que nos vayamos encontrando.

Como se puede ver, no es un algoritmo trivial de implementar, aunque puede ahorrar mucho coste computacional debido a que solo comprueba los segmentos que tengan probabilidades de intersectar con otros segmentos.

El algoritmo se pude definir como se muestra a continuación:

```
listaIntersecciones = SweepPlane(S)

1.    Crear una lista de acciones Q vacía
```

2. Insertar en la lista de acciones todos los puntos terminales de los segmentos contenidos en el conjunto S.

3. Crear una lista de estado T vacía.

4. **Mientras** existan acciones en la lista Q

a – Encontrar el siguiente punto de acción P en la lista de acciones Q

b – Procesar el punto de acción P

– quitar el punto de acción P de la lista de acciones Q.

– actualiza la lista de estados T

– verifica la existencia de intersecciones dependiendo del tipo de acción

– añadir los nuevos puntos de acción de tipo III resultantes de las intersecciones verificadas en la lista de acciones Q y en la variable listaIntersecciones.

5. retornar listaIntersecciones

Este algoritmo se puede realizar de una forma más optimizada aún si tenemos en cuenta para qué lo necesitamos en el entorno de planificación de trayectorias.

Los segmentos "artificiales" que se crean para comprobar la visibilidad entre los distintos nodos del espacio discreto no interfieren en los demás segmentos artificiales de visibilidad. Lo que podemos hacer es crear dos conjuntos de segmentos en los que sabemos que los segmentos de un conjunto no interferirán entre sí pero sí interferirán con los segmentos del otro conjunto.

Por ejemplo, tendremos dos conjuntos, uno con los segmentos de visibilidad, creados artificialmente, y otro con los segmentos de los polígonos que conforman los obstáculos. Si suponemos que los obstáculos no se solapan entre sí, podremos suponer que los segmentos del obstáculo no se intersectan entre sí. Sin embargo, sí que afectarán a los segmentos de visibilidad. Lo mismo ocurrirá con los segmentos de visibilidad, los cuales no debe comprobarse si se intersectan entre ellos, pero sí debe comprobarse si se intersectan con los segmentos del conjunto de segmentos de los obstáculos.

De esta forma la interfaz de la función terminaría siendo de la siguiente forma:

$$listaIntersecciones = SeepPlane(S_{visibilidad}, S_{obstáculos})$$

La complejidad computacional de éste algoritmo depende los algoritmos de ordenación y búsqueda que se implementen para las listas Q y T.

Sobre la lista Q, en la línea a) del paso 4 del algoritmo debe aplicarse un algoritmo de ordenación. Debemos intentar definir un algoritmo de ordenación lo menos complejo computacionalmente posible.

La lista T quizás se pueda implementar a través de árboles binarios balanceados. Necesitamos implementar sobre ella operaciones de inserción, ordenación, eliminación, etc.

5.9.4.2 Conclusiones

Aunque se ha estudiado ampliamente el algoritmo sweep plane, con el objetivo de que una futura implementación del prototipo lo incorpore, algoritmo finalmente implementado para el cálculo de intersecciones es el que utiliza la fórmula de la recta en forma paramétrica para calcular las intersecciones.

5.9.5 Algoritmos planificadores aritméticos de rutas

Se trata de algoritmos capaces de encontrar, dados una lista de nodos, el camino mínimo a recorrer para llegar desde un nodo del grafo (que representará en este caso a un waypoint) a otro nodo final.

En el planificador se tendrá que ejecutar este algoritmo a dos niveles:

- **Optimización de caminos**: El nivel más alto, que será el nivel en el que tendremos solucionado la evitación de obstáculos entre dos waypoints, tendremos que decidir qué waypoints recorrer primero para poder llegar desde el waypoint inicial hasta el final.

- **Evitación de obstáculos**: El nivel que representa la evitación de obstáculos. En este punto tendremos una serie de nodos definidos por el grafo de visibilidad y tendremos que decidir cuál es el menor camino para poder llegar desde un punto al siguiente evitando el obstáculo.

5.9.5.1 Optimización de caminos

Las características especiales que debe tener el algoritmo que implementa la optimización de caminos, y que lo diferencia de los demás problemas de "pathfind", es que en este caso vamos a tener un conjunto de waypoints, que van a hacer las veces de nodos, que podrán estar conectados todos con todos y que tendrán que ser recorridos todos y cada uno de ellos.

No se trata tan solo de encontrar el camino más corto para ir de un nodo inicial hasta un nodo final, sino que además hay que recorrer todos y cada uno de los nodos (waypoints) definidos por el usuario.

5.9.5.2 Algoritmos resolvedores del problema "pathfind".

Hay muchos algoritmos que resuelven este tipo de problemas. A continuación veremos algunos de ellos y veremos sus ventajas e inconvenientes a la hora de resolver este tipo de problemas.

Durante la discretización del espacio, veíamos que muchos algoritmos producían puntos por los que el robot podría evitar obstáculos. Estos algoritmos nos resolvían el problema de la evitación de obstáculos pero no el de optimización de caminos.

Por ejemplo, el Grafo de Visibilidad nos daba un conjunto de caminos a recorrer para evitar el obstáculo, la descomposición en celdas nos daba un conjunto de celdas libres de obstáculos por las que pasar, o el algoritmo BUG, que aunque ya establecía un camino, podíamos esquivar por un lado o por otro, y esta decisión estaba tomada antes de comenzar el algoritmo.

Se podría combinar un algoritmo de tipo BUG, en el que los posibles caminos a recorrer son las aristas de los obstáculos, comenzando en uno u otro vértice, con un algoritmo de optimización de caminos, en el que dados todos los posibles caminos, nos devuelva el más óptimo.

5.9.5.2.1 Algoritmo de Dijkstra

Ver bibliografía **[SRM]**. Es un algoritmo que determina cuál es el camino más corto dado un vértice origen y un conjunto de vértices. El algoritmo de Dijkstra está pensado para un grafo dirigido, por lo tanto, si se usara para el planificador habría que modificar el algoritmo.

La estrategia que sigue este algoritmo es recorrer todos los caminos más cortos para encontrar el camino más corto para llegar de un nodo inicial a cualquier otro nodo. El algoritmo sigue la estrategia de algoritmo voraz, por lo que lo que devolverá al final será un conjunto de trayectorias que hace mínimo el camino hacia cada uno de los nodos del grafo.

La idea de este algoritmo es la de que si tenemos que para llegar desde el origen (nodo **s**) hasta un vértice **u** del grafo un coste d(u) que es mínimo, si podemos llegar desde **u** hasta **v** podremos extender el camino para poder llegar desde **s** hasta v con un coste d(u) + w(u,v), donde w(u,v) es una función que nos indica el coste de ir desde u hasta v. En el caso de que d(u) + w(u,v) sea menor que d(v), estableceremos el nuevo coste d(v) en d(u) + w(u,v), estableciendo el nuevo antecesor para v como el nodo u.

Un algoritmo voraz sigue la siguiente estrategia: Si C es el conjunto de posibles candidatos a solución, y S es el conjunto de candidatos seleccionados por el algoritmo voraz para formar parte de la solución, el algoritmo voraz en cada iteración seleccionará un candidato del conjunto de candidatos de entrada C, al que, por ejemplo, vamos a llamar **a**, el algoritmo voraz comprobará si $\mathbf{a} \cup \mathbf{S}$ puede llevar a una solución. En caso contrario eliminará el candidato **a** y vuelve a seleccionar otro. En caso de que sea factible, añadirá el candidato **a** a la lista de seleccionados S y comprueba si es solución.

En caso de que sea solución, termina el algoritmo. En caso contrario, el algoritmo seguirá iterando.

El algoritmo de Dijkstra tendrá como parámetros de entrada el grafo (G(V,E), conde V es el conjunto del vértices y E es el conjunto de enlaces que existen entre los vértices) además de el nodo inicial del que se quiere partir, al que en este case lO hemos llamado s. El algoritmo de Dijkstra en pseudocódigo es el siguiente:

```
S = Dijkstra(G(V,E),s)
1. S ← s;
2. para i ∈ V hacer
        d(V[i]) = ∞;
        antecedentes(V[i]) = indefinido;
3. d(s) = 0;
4. Q = V[G];
5. mientras no esté vacío el conjunto Q hacer
6.      x = Extraer_Minimo(Q);
7.      S = S ∪ {x}
8.      para cada enlace (x,v) saliente de x hacer:
9.          Si d(x) + w(x,v) < d(v)
                d(v) = d(x) + w(x,v);
                antecesor(v) = x;
10. retornar S.
```

El conjunto Q va a ser el conjunto de vértices que quedan por estudiar, y el conjunto S va a ser el conjunto de vértices escogidos para que pertenezcan a la solución.

Como vemos, lo primero que se hace es inicializar todas las distancias d(v) a infinito. Esto implica que todavía no conocemos el coste que conlleva trasladarnos desde el origen **s** hasta un punto cualquiera del grafo v. La única excepción es la distancia del nodo origen, que será 0.

Seguidamente inicializamos los antecesores. Los antecesores son los que nos van a indicar cuál es el antecesor en el camino de menos coste al nodo actual, cuando se termine de ejecutar el algoritmo. Por ejemplo, poner antecesor(x) = s, nos indicará que

para llegar al nodo x con el coste mínimo, primero debemos llegar al nodo s, del que pasaremos directamente al nodo x.

Se inicializa el conjunto Q a todos los vértices. A continuación, comenzamos a extraer los vértices de Q escogiendo para ello el vértice con menor coste del conjunto Q. La función **Extraer_Minimo** busca el vértice con menor coste de entre los vértices que quedan por extraer del conjunto de vértices Q.

Añadimos el nuevo nodo obtenido x a la lista de vértices escogidos S, y comprobamos todos los enlaces salientes de x, de forma que si encontramos un enlace saliente de x tal que d(x) + w(x,v), es decir, que el camino que habíamos encontrado anteriormente desde el origen hasta el nodo v conlleva un mayor costo que el nuevo camino en el que se pasa por el nodo x, asignamos a d(v) el nuevo valor y sustituimos su antecesor por el nuevo x encontrado.

En cuanto a la complejidad del algoritmo, este algoritmo puede llegar al presentar una complejidad de orden $O(n^3)$ en el peor de los casos. Sin embargo si conseguimos que la función sea una simple búsqueda lineal, podríamos llegar a conseguir que la complejidad disminuya a aproximadamente una complejidad de $O(n^2)$. Esto quiere decir que será un problema con un costo computacional exponencial. No podremos trabajar con él con un número elevado de nodos.

De todas formas, los grafos en la planificación no llegarán a tener un número excesivo de nodos, ya que normalmente no se necesitan demasiados waypoints para establecer la ruta de un submarino.

Sin embargo, la ventaja que conlleva implementar este algoritmo, es que estaremos seguros de determinar cual es el camino de mínimo coste para llegar de un nodo al siguiente, algo que no tendremos para otros algoritmos de menor complejidad computacional.

5.9.5.2.2 Algoritmo de Floyd-Warshall

[FLOYD]. Es una alternativa al algoritmo de Dijkstra. Se puede implementar con tan sólo tres bucles anidados, como se mostrará a continuación.

Este algoritmo permite pesos negativos, aunque se pueden dar problemas con los ciclos con pesos negativos.

Su implementación consiste en una matriz en la que se almacenarán para todos los pares de nodos los pesos de los enlaces que los unen. La matriz sigue el siguiente criterio:

$$A_0[i,j] = \begin{cases} 0 & \text{si } i=j \\ C[i,j] & \text{si } i \neq j \end{cases}$$

$$A_k[i,j] = \min(A_{k-1}[i,j],\ A_{k-1}[i,k] + A_{k-1}[k,j])$$

donde i, j y k serán nodos del grafo.

$A_k[i,j]$ significa el costo del camino más corto que va de la i hasta la j, y que no pasa por algún vértice mayor que k.

El algoritmo en pseudocódigo será el siguiente:

Como se puede observar fácilmente, este algoritmo tendrá una complejidad de $O(n^3)$. Es un algoritmo con complejidad exponencial, por ello no se puede aplicar a un número de nodos alto, ya que el tiempo necesario para resolver el problema crecería exponencialmente.

Otras alternativas a este tipo de algoritmos son las siguientes:

- Algoritmo de Bellman-Ford
- Algoritmo de Prim
- Algoritmo de Kruskal
- Algoritmo de Ford-Fulkerson

5.9.5.2.3 Algoritmo A*.

[**IA**]Este algoritmo es una de las opciones a la hora de implementar una solución que nos permita obtener la menor trayectoria desde un punto al siguiente. Éste método hace uso de heurísticas para calcular el posible costo que se tendría desde un nodo al siguiente, además de una función de costo que nos indica que costo ha tenido movernos hasta el punto actual.

El principal problema a la hora de implementar los anteriores algoritmos surge de que son algoritmos muy pesados, ya que intentan obtener la solución óptima y para ello debe recorrer todos los caminos posibles que se pueden generar desde el inicio hasta el fin.

El algoritmo A* se beneficia de las ventajas de la búsqueda en anchura y de las ventajas de la búsqueda en profundidad en los árboles. Mientras que con la función de coste se tiende a la búsqueda en anchura, con la función heurística se tiende a la búsqueda en profundidad.

Los anteriores métodos solo empleaban para calcular la bondad de un camino la función de coste o la heurística, tan solo una de las dos alternativas. Sin embargo, el algoritmo A* utiliza tanto la heurística como el algoritmo de coste para realizar la estimación de cuanto me ha costado llegar hasta aquí y aproximadamente cuánto me costará llegar al nodo destino. La función de bondad del estado actual que utiliza es la siguiente:

$$f(n) = g(n) + h(n)$$

La complejidad computacional de un algoritmo A* para encontrar el camino más corto depende de la heurística implementada. Mientras mejor sea la heurística implementada menor complejidad computacional tendrá el algoritmo A*. Sin embargo, si tenemos una heurística mala la complejidad computacional del algoritmo A* para la resolución del algoritmo del pathfind puede llegar a ser exponencial. Con la mejor heurística podríamos llegar a conseguir que se ejecutara en un tiempo lineal.

La estrategia que se sigue en este algoritmo es muy sencilla. El algoritmo contiene dos listas:

- **Abierta**: En esta lista se irán insertando los caminos o nodos que quedan por expandir. Estarán ordenados mediante la función de evaluación, de forma que a la hora de expandir el siguiente nodo, se escogerá el primero de ésta lista, el que tenga una mejor función de evaluación, se expandirá sus hijos y se calculará la función de evaluación para cada uno de ellos. Los hijos serán añadidos a la lista abierta para una posterior expansión en caso de que fuera necesaria. El nodo recién expandido se quitará de la lista abierta y pasará a la lista cerrada.

- **Cerrada**: almacena los nodos ya expandidos y el valor para su función de evaluación. En caso de que se detecte que un nuevo nodo expandido coincide, su último nodo de la trayectoria, con otro nodo que se encuentra en la lista cerrada, se eliminará el que presente un valor menos óptimo para la función de evaluación. En caso de que coincida el último nodo de la trayectoria expandida con otras trayectorias en la lista abierta, se eliminarán todas (pasarán a la lista cerrada, donde sólo podrá quedar la solución más optima) excepto la más óptima.

El algoritmo en pseudocódigo puede ser el siguiente:

El algoritmo A* es un algoritmo óptimo y completo si:

- Todo nodo tiene un número finito de sucesores.

- El coste de la aplicación de cada operador (transición) es no negativo (>0)

- La función h(e) es una heurística admisible.

Una heurística es admisible si asegura encontrar una solución óptima si ésta existe. Esto ocurre si nunca se sobreestima el coste de alcanzar la mejor solución desde n:

$$h^*(n) \leq h(n)$$

Las ventajas que nos proporciona este algoritmo al planificador con respecto a los demás algoritmos son:

- Menor complejidad computacional que los algoritmos de Dijkstra y Floyd.

- No necesita que el grafo sea dirigido. Esto implica una modificación menos que hay que aplicarle al grafo. Aunque se sigue manteniendo la modificación para que tenga que pasar de forma obligada por todos los waypoints que marque el usuario.

Las desventajas que aparecen son:

- Este tipo de algoritmo, aunque pueden llegar a tener poca complejidad computacional, el uso de la memoria puede aumentar exponencialmente.

- Una mala elección de la heurística podría provocar que no encontremos la solución óptima, además de aumentar la complejidad computacional.

La solución a la segunda desventaja sería tener especial cuidado a la hora de elegir la heurística. Ésta debe ser lo más informada posible pero que no sobreestime el valor real.

Tanto es así, que si la heurística consiste en calcular la distancia de Manhattan (ver la distancia al destino moviéndonos solo a los cuadrados horizontales y verticales adyacentes al actual), por ejemplo, se podría multiplicar el valor que nos devuelve esta heurística por 10 para que no sobreestime el óptimo real, es decir, que la medida que nos de la heurística de la distancia siempre sea mayor a la distancia real que hay entre ese nodo y el nodo destino. Así, la evaluación de una heurística debe proporcionarnos un valor menos óptimo que el real.

La solución a la primera desventaja es más complicada. Para evitar que este algoritmo consuma una cantidad de memoria excesiva se han ideado una serie de m modificaciones a éste algoritmo. Una de ellas es el algoritmo **A*SRM [IA]**.

5.9.5.2.4 Algoritmo A*SRM

Es el algoritmo A* acotado por la memoria. Se utiliza para trabajar con memoria limitada. Mientras tenga memoria, usara toda la memoria de la que dispone, mediante el procedimiento normal del algoritmo A*.

Si al generar un sucesor falta memoria, se libera espacio de los estados menos prometedores.

5.9.6 Otros algoritmos

5.9.6.1 Implementación del algoritmo de Graham.

Como veíamos antes, el algoritmo de Graham se usa para determinar el menor polígono convexo que contiene a todo un conjunto de puntos definidos en el mapa. Si tenemos un obstáculo definido como un conjunto de puntos en un mapa, este algoritmo nos ayudará a expresar este obstáculo como un polígono convexo.

La idea de este algoritmo es la de comenzar seleccionando el punto con una menor coordenada Y, e introducir a todos los demás puntos en una lista ordenada, según el ángulo que forman entre este punto inicial y la horizontal, ordenándolos de menor a mayor ángulo.

Para seguir iterando, se seleccionará el primer punto de esta lista, que será el que menor ángulo forme con el primer punto escogido, y se tomará como nuevo punto de referencia. En caso de que haya más de un punto que forme el mismo ángulo con el punto inicial, se escogerá el punto que se encuentre más alejado del punto inicial.

Con éste punto seleccionado, el próximo paso es buscar el siguiente punto. El punto seleccionado será el que menor ángulo forme con respecto al punto seleccionado y la horizontal.

Y así sucesivamente, iremos envolviendo el conjunto de puntos con un polígono convexo. En la Ilustración 28 podemos ver un ejemplo de ejecución de éste algoritmo.

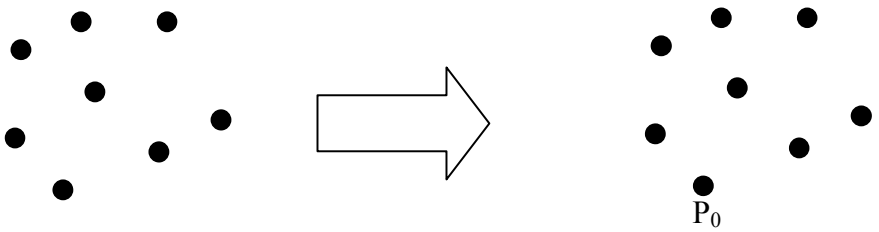


Ilustración 28: Paso 1 Algoritmo de Graham

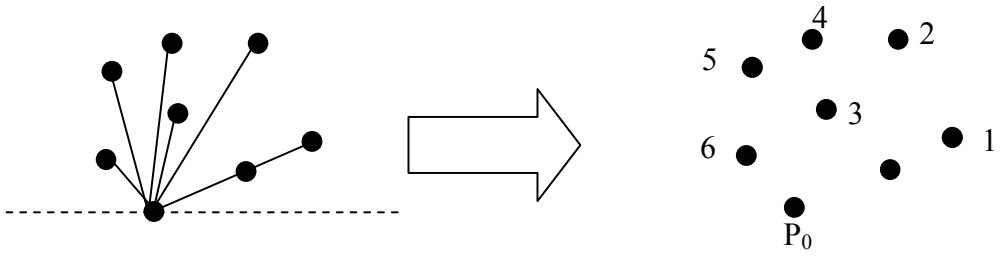


Ilustración 29: Paso 2 Algoritmo de Graham

En la Ilustración 30 se muestra que no se ha numerado uno de los puntos. Esto es debido a que formaba el mismo ángulo que otro punto que estaba más alejado. En ese caso, podemos ignorar los puntos más cercanos al punto de referencia.

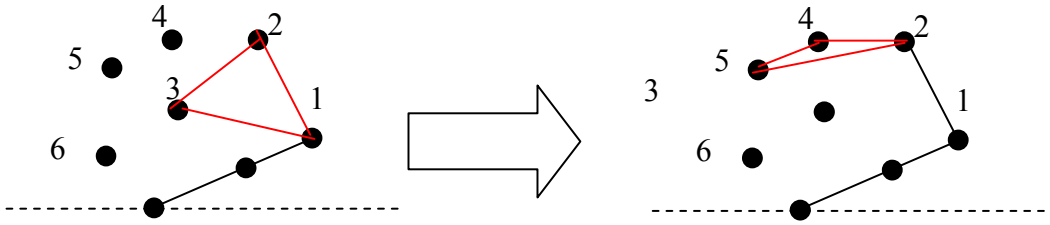


Ilustración 30: Ejecución Algoritmo de Graham

Como se puede ver, se aplica una técnica especial de triangulación que más delante será explicada. Se utiliza para comprobar qué punto forma un menor ángulo con el punto actual.

Se comprueba la triangulación entre el punto 1, el 2 y el 3 para ver cual de los tres puntos forma un menor ángulo con el punto 1 y la horizontal. En este caso se quedará con el punto 2, pero no se quedará definitivamente con el 2 hasta que comprueba que efectivamente, es el punto que genera un menor ángulo, es decir, seguirá comprobando con los demás ángulos de la lista por si hubiera otro que se encontrara formando un menor ángulo con el punto 1 y la horizontal. Seguirá realizando las comprobaciones consecutivamente hasta llegar al último punto de la lista. Al final tendremos algo como lo que se muestra en la Ilustración 31.

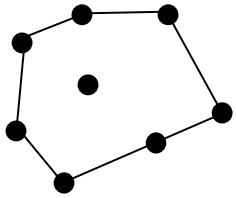


Ilustración 31: Resultado Algoritmo de Graham

El pseudocódigo que implementa el algoritmo de Graham es el siguiente:

S=Graham(Q)

1. definir p_0 como el punto del conjunto Q con menor coordenada y. En caso de que exista más de un punto se cogerá el que tenga una menor coordenada X.

2. ordenar los puntos $\{p_1, p_2, p_3, p_4, p_5, ..., p_m\}$ por el ángulo en sentido antihorario relativamente al punto p_0. En caso de que existan puntos con ángulos iguales, se considera sólo el punto más alejado de p_0, siendo los otros removidos.

3. inicializar(S)

4. apilar(p_0, S)

5. apilar(p_1, S)

6. apilar(p_2, S)

7. **para** i=3 hasta m

    mientras (ángulo formado por (penultimo(S), último(S), p_i) inferior a 180º)

        desapilar(S)

    eapilar(p_i)

8. **retornar** S

Como se puede ver, S será una pila donde iremos apilando o desapilando para conseguir el estado final de la pila con todos los vértices del polígono.

¿Por qué se calcula el ángulo con los tres puntos? Esta implementación del algoritmo de Graham utiliza una técnica para saber qué puntos forman un menor ángulo con el punto actual, que consiste en coger el punto actual y los dos siguientes, y mediante una multiplicación escalar ver qué punto forma un ángulo menor con la horizontal.

Si tenemos tres puntos, p_0, p_1 y p_3, como se muestra en la siguiente figura, podremos hacer el siguiente producto vectorial:

$$(p_1 - p_0)x(p_2 - p_0)$$

En caso de que sea un producto negativo, ocurrirá lo que se muestra en la figura de la derecha de la Ilustración 32. En caso contrario ocurrirá lo que se indica en la figura de la izquierda.

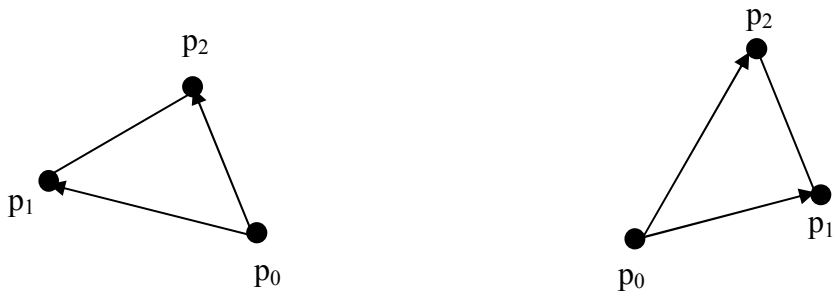


Ilustración 32: Producto de vectores

Otras alternativas al algoritmo de Graham pueden ser el Algoritmo de Jarvis March o el Algoritmo de Quick-hull.

Parte III. Diseño del prototipo

Capítulo 6. Diseño del sistema

6.1 Selección del lenguaje de programación

En esta sección se estudiarán los lenguajes de programación barajados para la implementación del software integrado de planificación y supervisión de misiones de vehículos submarinos autónomos.

Desde un principio se decidió primar la facilidad de programación y la disponibilidad de librerías y herramientas frente al rendimiento en tiempo de ejecución, por lo que se consideraron las alternativas de Matlab y Java.

A continuación se estudian estos dos lenguajes de programación y se muestran las ventajas y desventajas.

6.1.1 Matlab

[MATLAB] En un principio este proyecto estaba pensado para desarrollarse en Matlab. Las ventajas que proporciona este lenguaje y por las que fue considerado un posible candidato son:

- Es un lenguaje de programación potente y es más sencillo implementar complejas ecuaciones matemáticas en este lenguaje que en cualquier otro, como Java o C. Esto es así puesto que este lenguaje permite una sencilla manipulación de matrices, representar gráficamente funciones y datos, implementación de algoritmos, creación de interfaces de usuario, y además permite la ejecución de programas escritos en otros lenguajes como C o Fortran.

- Incluye muchas librerías (toolbox) gratuitas que implementan determinadas funcionalidades y que serían de mucha utilidad. Por ejemplo:

    - Algoritmos de inteligencia artificial que se podrían utilizar para calcular los posibles caminos para un submarino a la hora de planificar una misión en la que éste se desplazara desde un punto inicial del mapa hasta un punto final del mismo.

    - Gestión de ficheros XML.

    - Gestión de formatos batimétricos (como por ejemplo, netCDF).

Desventajas

Las desventajas que tiene Matlab en el marco de nuestro proyecto son las siguientes:

- Es necesario tener instalado Matlab para ejecutar la aplicación implementada en este lenguaje. Esta característica juega mucho en contra

de esta elección, ya que es un entorno muy caro y solo se suele utilizar en ámbitos académicos o de empresas. Además es un entorno muy pesado, y necesita muchos recursos de ordenador para ejecutarse de forma eficiente.

- Matlab sólo incorpora algunos recursos básicos para definir clases, por lo que no se podrían aprovechar todas las ventajas que nos ofrecía la orientación a objetos en una aplicación como la que pensábamos desarrollar. El entorno Matlab, cuando se inició el desarrollo del proyecto, ofrecía una orientación a objetos que obligaba a implementar cada una de las funciones de una clase en un fichero distinto, y todas las funciones de la misma clase incluidas en la misma carpeta, por lo que el uso de objetos en Matlab se hacía tedioso y propenso a errores.

- Las opciones para desarrollar la interfaz con el usuario estaban bastante limitadas en comparación con las opciones que ofrecían otros entornos como Java o C.

6.1.2 Java

Java **[JAVA]** fue la siguiente opción después de descartar Matlab.

Ventajas.

Algunas de las ventajas que ofrece un lenguaje de programación como Java son las siguientes:

- Lenguaje de programación orientado a objetos.

- Independencia de la plataforma. Podremos compilar el programa desarrollado en Java y ejecutarlo en cualquier plataforma con solo instalar el JRE (Java Runtime Environment) correspondiente.

- Java incluye un conjunto de herramientas nativas como AWT y Swing que ofrecen componentes GUI (Graphical User Interface), mecanismos para usarlos y manejar eventos asociados. Debido a ello, con este lenguaje de programación se pueden desarrollar interfaces más complejas de lo que Matlab es capaz de ofrecer.

- Hay implementadas muchas librerías gratuitas que permiten trabajar con documentos XML, el formato batimétrico NetCDF, generar gráficas, incrustar un mapa de Google Maps en la aplicación, etc.

Inconvenientes

Los inconvenientes de utilizar el lenguaje de programación Java en el contexto de este proyecto son:

- Aunque hay muchas librerías gratuitas para Java, es complicado encontrar librerías que implementen algoritmos matemáticos complejos o de inteligencia artificial para esta plataforma.

- Java no tiene de forma nativa librerías que permitan mostrar gráficas de forma sencilla, a diferencia de Matlab.

La comparación entre estos dos lenguajes resultó ser claramente favorable a Java, por lo que fue la alternativa finalmente seleccionada. A continuación se van a explicar las características necesarias que debe tener Java para poder utilizarlo como lenguaje de programación en nuestro proyecto.

IDEs para el desarrollo de software con Java

Hay varios entornos para trabajar con Java, como pueden ser **Eclipse** (el cual es gratuito), o **JBuilder**(que es software propietario). La mayor experiencia del alumno en el manejo de **Netbeans** decantó la selección hacia esta opción.

Netbeans

Netbeans **[NETBEANS]** es un entorno gráfico gratuito que nos ofrece la posibilidad de implementar una interfaz gráfica en Java de una forma muy fácil y completa. Y no sólo facilita realizar una interfaz gráfica, también posee otras ventajas de este entorno, como puede ser que corrige en tiempo de edición los fallos tanto sintácticos como semánticos del código que estemos escribiendo, muestra todas las funciones accesibles de una clase cuando se escribe el nombre de una variable que representa un objeto de esa clase, para facilitar al usuario la programación, y no obligarle a ojear el manual cada vez que quiere conocer las funciones de una clase, etc.

6.1.2.1 Interface Matlab desde Java.

Se han encontrado distintas implementaciones que nos permiten mantener una interface desde Java hasta Matlab. El problema siempre se encuentra en que no se puede ejecutar directamente código Matlab desde Java, sino que hay que abrir una sesión de Matlab a la que se envían las instrucciones a ejecutar.

Algunas librerías que permiten implementar esta interfaz son las siguientes:

- **JMatLink**: **[JMATLINK]** Es una librería implementada en la que se pueden abrir varias sesiones de Matlab a la vez (a diferencia de JLab, que solo permite abrir una sesión).

- **JLab**: **[JLAB]** Está basada en la librería JMatLink. Implementa las mismas funciones, aunque sólo es capaz de mantener una única sesión con Matlab abierta en un determinado momento. Para instalarlo únicamente es necesario incluir la librería que trae, jlab.jar, a nuestro proyecto, además de agregar el JLab2.dll a la carpeta C:\Windows\system32, en WindowsXP.

- **engine**: **[ENGINE]** esta librería implementa dos formas distintas de acceder a Matlab, denominadas en el archivo engine.zip como approach1 y approach2.

Hay otras de pago, como pueden ser el J-Integra, y por ello en este caso no interesan.

Los inconvenientes que tienen este tipo de librerías son:

- Hay que abrir una sesión en Matlab. Por lo tanto se vuelve un programa muy lento si hay que abrir una sesión de Matlbab cada vez que necesitamos acceder a alguna de sus funciones o representar gráficas. Abrir una sesión en Matlab es una tarea que puede ser muy pesada en memoria y tardar mucho en abrirse.

- No existe bidireccionalidad entre Java y Matlab. Desde Java es imposible conocer las interacciones del usuario con Matlab.

- El principal inconveniente de utilizar Java con Matlab es que se perdería la ventaja de Java de ser multiplataforma, ya que se necesitaría también tener instalado un entorno tan pesado y costoso como es Matlab.

6.1.2.2 XML en Java.

En relación con el uso del XML en Java, en Internet existe abundante documentación. Por ejemplo, en Internet es muy fácil encontrar tutoriales que explican cómo leer y escribir ficheros XML o cómo definir parsers (no fue necesario en este caso).

Algunas alternativas que se encuentran disponibles son las siguientes:

- **DOM**: **[DOM]** es una API de Java que permite tanto leer como escribir ficheros XML. Genera un árbol al leer todo el fichero, y lo construye completo en memoria, esto lo hace más lento y pesado. Al finalizar la lectura del documento XML, tendrá todo el árbol conteniendo toda la información del fichero XML cargado en memoria, y con lo que sólo resta mapear toda esa información a clases de Java.

- **SAX**: **[SAX]** es una API de Java que permite leer ficheros XML, aunque no escribirlos. Es menos pesado y más rápido que el DOM, puesto que a diferencia de éste no carga todo el documento XML, sino que funciona con eventos. Por ejemplo, el evento de encontrar el elemento que buscamos, o el evento del final de fichero.

- **JDOM**: **[JDOM]** El inconveniente de las dos APIs anteriores es que no estaban pensadas para Java, sino de una forma genérica. El JDOM está pensado exclusivamente para Java, por ello es más sencillo e intuitivo de manejar que los dos anteriores.

- **JAXP**: **[JAXP]** Esta es otra alternativa al JDOM, pero tiene el inconveniente de que no todos los parsers tienen soporte a esta API, al contrario de lo que ocurre con DOM y SAX (JDOM incluye el patrón Adapter, que le permite adaptarse a los diferentes parsers).

JDOM es la API finalmente escogido para este proyecto. se puede descargar desde la página www.jdom.org. Para instalarla, se puede compilar, (aunque ya viene con un jdom.jar) y el resultado, la librería jdom.jar se encontrará en la carpeta build. También se puede compilar para obtener los ficheros fuentes, o la documentación sobre el jdom, o incluso para compilar los ficheros de ejemplo que trae consigo (ver fichero README.txt).

Una vez obtenida la librería se puede añadir al proyecto y ya tenemos funcionando el jdom en nuestro proyecto.

6.2 Diseño de la comunicación.

Para la comunicación entre submarino y planificador, teníamos un primer problema, y es que el software embebido en el submarino podría estar implementado en C++ y el software del planificador estar implementado en JAVA. Puede que las dos plataformas tengan tipos de datos distintos, por ello hay que buscar una forma de homogeneizar los datos antes de proceder a la comunicación entre ellos y el envio de datos en la red. Para ello se ha escogido protocolo **XDR**.

6.2.1 Protocolo XDR

XDR son las iniciales de "e**X**ternal **D**ata **R**epresentation". Éste es un protocolo de presentación de datos que permite la transmisión de datos entre máquinas de diferentes arquitecturas y sistemas operativos. Este protocolo se encuentra en el **nivel de presentación**, sobre el **protocolo de sesión** ONC RPC, el cual es un protocolo de llamada a procedimiento remoto desarrollado por Sun Microsystems. Trabaja a nivel de ordenamiento de byte, códigos de caracteres y sintaxis de estructura de datos. El tamaño de los datos es múltiplo de cuatro bytes. En caso de que el tamaño de los datos no sea múltiplo de 4, se rellenará con una cantidad extra de bytes con su contenido nulo.

La representación XDR de los datos es de tipo BIG-ENDIAN (es decir, el bit más significativo primero).

Del XDR usaremos los siguientes tipos básicos:

Descripción	NombreClase
Entero Corto (4 bytes, con signo)	**Short**
Entero (4 bytes) con signo	**Integer**
Entero (4 bytes) sin signo	**Unsigned Int**
Flotante (4 bytes)	**Float**
Flotante doble precisión (8 bytes)	**Double**
Ristras de caracteres	**String**
Vector de ristras de caracteres	**String[]**
Valores lógicos (4 bytes, según XDR)	**Boolean**
Bytes de datos (8 bits, sin signo)	**Byte**

6.2.2 Protocolo de comunicaciones.

El protocolo de comunicación consistirá en el intercambio de una serie de datos encapsulados en una clase denominada Mensaje. Esta clase será serializable, ya que necesitamos que sea intercambiable entre el AUV y el planificador/monitorizador de la misión.

Va a haber varios tipos de comunicación, tal y como veíamos en el análisis de la comunicación. En este caso, vamos a tratar tres tipos de comunicación:

- **Comunicación tipo Telecomando**: puede ser la comunicación que se establece entre el planificador/monitorizador de la misión y el AUV, y servirá para emitir determinados comando al vehículo. Por ejemplo, este tipo de comunicaciones puede servir para usar el AUV como si fuera un ROV, o simplemente para emitir una orden para que le transmita al monitorizador de la misión determinados datos de ésta.

- **Transmisión de datos escalares**: Esta comunicación se puede dar cuando el AUV necesita transmitir datos escalares al monitorizador de la misión. Este tipo de comunicación evita tener que encapsular todos los datos en ficheros cuando lo que se quiere enviar es por ejemplo una muestra en concreto. Este tipo de mensajes se podría dar como respuesta a una petición del monitorizador de enviar todos los datos obtenidos durante la misión.

- **Transmisión de ficheros**: el planificador necesitará emitir al vehículo los datos de la misión, los cuales estarán almacenados en varios ficheros. Este tipo de comunicación servirá para eso. Los tipos de ficheros que más se usarán serán:

  - NetCDF: Será un tipo de fichero que permitirá enviar datos matriciales o batimetría.

  - XML: fichero de configuración, misiones, descripción del AUV y subsistemas.

  - Fichero plano: Se le podría dar otros usos.

6.2.2.1 Modelo de datos de la comunicación

En esta sección se mostrará los modelos de datos usados para la comunicación y cómo se serializarían éstos para ser transmitidos por la red.

6.2.2.1.1 Encabezado estándar del paquete de comunicación.

La clase Mensaje, que será la que se intercambiarán los distintos actores que se dan en este proyecto (AUV, planificador y monitorizador) tiene que tener las siguientes características:

- **Serializable**: Una propiedad esencial de la clase Mensaje es que sea Serializable, es decir, que se pueda convertir en una secuencia de bytes para enviarlo a través de una red.

- **Identificable**: Cada Mensaje debe tener asignado un número que lo identifique unívocamente de los demás mensajes. Por ello, la clase Mensaje debe heredar de la clase Identificable, tal y como se muestra en el diagrama más adelante. Tendrá un único atributo que será un entero sin signo, el id.

- **Temporizable**: Se deberá asignar un sello temporal al mensaje. Por ello, esta clase, de la que heredará la clase Mensaje, tendrá un atributo, de tipo Tiempo, que será un entero sin signo de 64 bits.

- **Comunicable**: La clase comunicable contendrá información sobre cómo debe ser la comunicación. Tendrá los siguientes atributos:

    - Remitente: será la dirección del remitente del mensaje.

    - Destinatarios: un vector con las direcciones de los destinatarios del mensaje.

    - Urgencia: especificará la urgencia del mensaje.

    - Certificado: indicará que se desea una respuesta por parte del receptor del mensaje de que éste le ha llegado y si ha aceptado o denegado la solicitud.

    - Seguimiento: indicará al receptor del mensaje que se desea un seguimiento de la evolución del pedido. Cada vez que este cambie de estado en el receptor del mensaje, se enviará un informe al emisor.

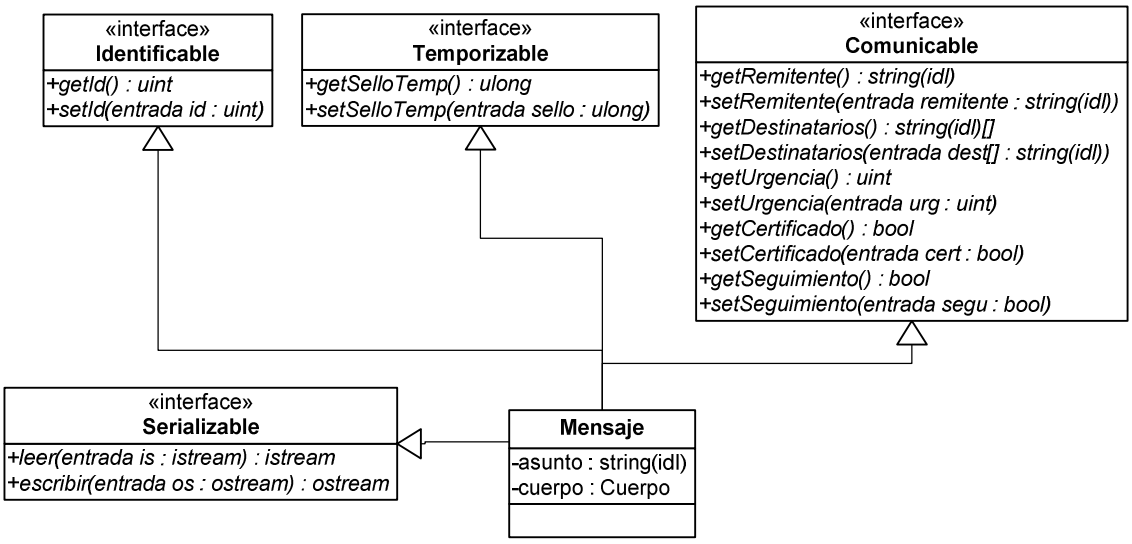


Ilustración 33: Características del Mensaje

Como estamos trabajando con Java, hemos de que tener en cuenta que una clase solo puede heredar de una única clase. Por ello no vamos a poder implementar esto exactamente de esta forma, pero hay alternativas. Por ejemplo, lo que sí podemos hacer es que una clase implemente varias interfaces.

Una interfaz en Java no puede tener atributos, solo funciones, que conformarían la interfaz. Por ello, lo que podemos hacer es una función get y otra set por cada atributo que necesitamos que se cree la clase que herede de la interfaz.

La representación XDR de estas interfaces será la siguiente:

- **Identificable**: Una clase que implemente la interfaz identificable se representará de la siguiente forma:

Serialización	Tipo	Nombre	Descripción
XDR	Unsigned Int	id	Identifica al elemento, y permite asociarlo con otros.

- **Temporizable**: Las clase que implementen esta interfaz deberán tener los siguientes atributos:

Serialización	Tipo	Nombre	Descripción
XDR	**Tiempo**	selloTemporal	Sello temporal del elemento.

Tipo Tiempo:

Serialización	Tipo	Nombre	Descripción
XDR	Unsigned int	segundos	Según la codificación segundos + nanosegundos transcurridos desde el 'epoch' de UNIX (1970).
XDR	Unsigned Int	nanosegundos	Según la codificación segundos + nanosegundos transcurridos desde el 'epoch' de UNIX (1970).

- **Comunicable**: Se necesitan los siguientes atributos para implementar esta interfaz:

Serialización	Tipo	Nombre	Descripción
XDR	String	remitente	Dirección del emisor. Suele ser ip:puerto y para componentes CoolBOT se extiende a ip:puerto:componente:puerto.
XDR	String	destinatarios[]	Vector con las direcciones de destinatarios. Su formato suele ser como el indicado para el remitente.
XDR	Unsigned Int	urgencia	Indica el nivel de urgencia/prioridad.
XDR	Bool	certificado	Indica al receptor, que debe enviar una de aceptación o negación.
XDR	Bool	seguimiento	Activado, indica que el receptor del mensaje enviará informes al cliente cada vez que se produzca algún evento en la solicitud mientras es procesada.

6.2.2.1.2 Mensajes de tipo TeleComando

El Mensaje podrá ser de varios tipos:

- **Solicitud**: La comunicación entre las dos partes se inicia con una solicitud de parte del cliente, que pedirá información de algún tipo al servidor.
- **Informe**: El informe es la respuesta a una solicitud. El servidor proporcionará una clase de tipo informe serializada para responder a la petición del cliente.
- **Tiempo**: Será un tipo de mensaje de sincronización entre el servidor y el cliente.

En el siguiente diagrama de clases UML se ven los tres tipos de mensaje. El tipo de mensaje lo define el cuerpo del mismo.

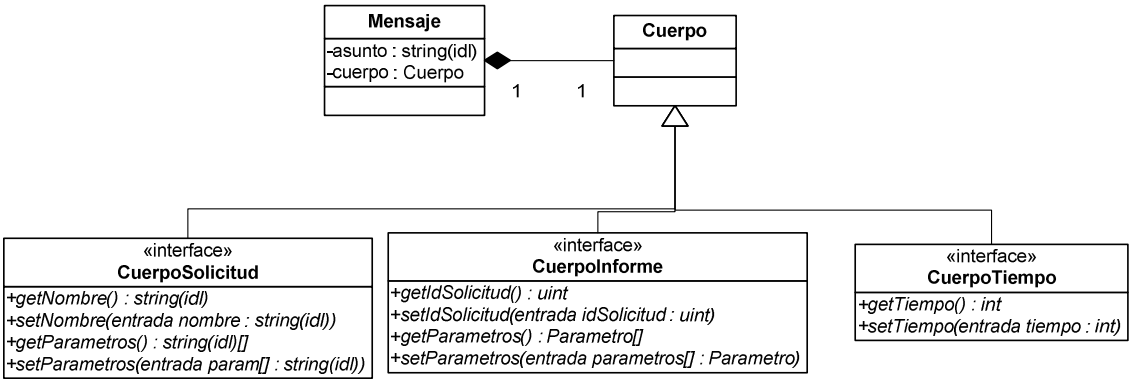


Ilustración 34: Mensaje

Como se puede ver, el cuerpo del mensaje puede ser de tipo CuerpoSolicitud (en solicitudes), de tipo CuerpoInforme (en informes) o de tipo CuerpoTiempo (en mensajes de sincronización).

Siguiendo este esquema, tendremos que el formato del mensaje sería de la siguiente forma:

Serialización	Tipo	Nombre	Descripción
XDR	**Identificable**		Identificación del mensaje.
XDR	**Temporizable**		Sello temporal del mensaje.
XDR	**Comunicable**		Información de comunicación del mensaje.
XDR	String	asunto	Pueden ser tres tipos: **"solicitud"** (elemento de comunicación **PET**), **"informe" (RES), "tiempo" (TIC)**
XDR	***Cuerpo***	cuerpo	Datos adjuntos al mensaje. Dependiendo del **asunto**, el tipo Cuerpo varía.

Como veíamos en el diagrama de clases, la clase Cuerpo puede ser de tres tipos distintos.

- **CuerpoSolicitud**:

Serialización	Tipo	Nombre	Descripción
XDR	String	nombre	Nombre del comando.
XDR	***Parámetro***	parámetros[]	Cada uno de los parámetros necesarios para ejecutar el comando anterior.

- **CuerpoInforme**:

Serialización	Tipo	Nombre	Descripción
XDR	Unsigned Int	idSolicitud	Nombre del comando.
XDR	***Parámetro***	variables[]	Vector con variables que comunica el informe, o variables que son devueltas como respuestas.

- **CuerpoTiempo**:

Serialización	Tipo	Nombre	Descripción
XDR	Integer	Tiempo	Tiempo Marcado en el Time_Clock.

El **TipoParámetro** será de la siguiente forma:

Serialización	Tipo	Nombre	Descripción
XDR	String	nombre	Nombre de la parámetro, i.e. el parámetro formal
XDR	***Empaquetable***	valor	Valor del mismo como empaquetable, de modo que tendrá su propia serialización. Éste sería el parámetro real.

Como vemos, a su vez, el TipoParámetro contiene una clase denominada **Empaquetable**. Esta clase servirá para agrupar todos los tipos de datos que se quieran enviar en el mensaje. Se podrán enviar tanto una lista de medias, como una muestra en concreto o un fichero.

El tipo empaquetable se corresponderá con el diagrama de la Ilustración 35.

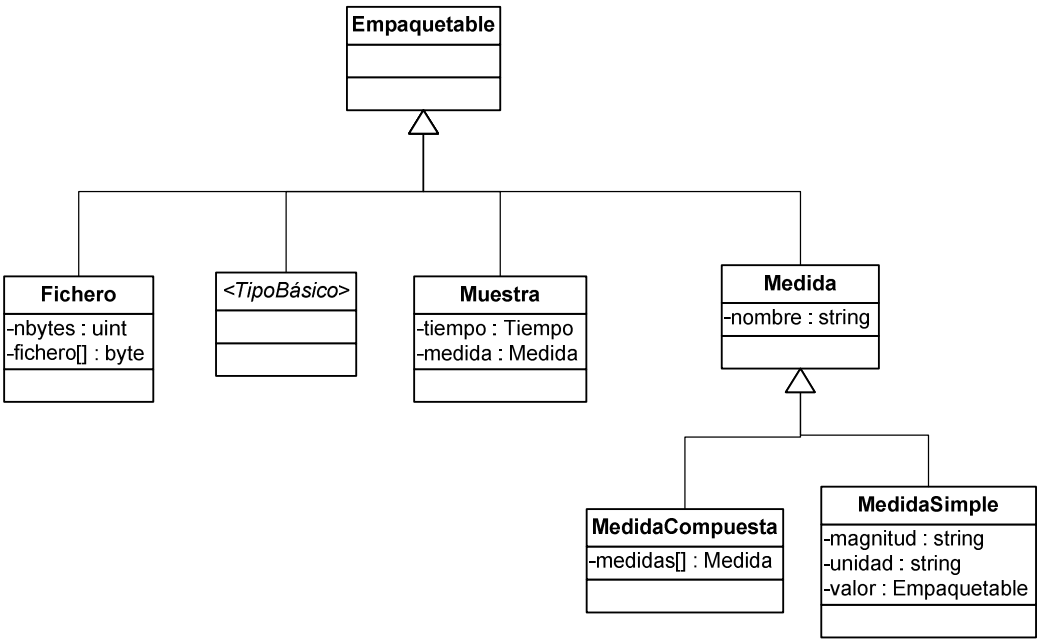


Ilustración 35: Definición del tipo Empaquetable

Puede ser un fichero, un tipo básico, una muestra en concreto o de tipo medida, que a su vez puede ser una medida compuesta (una lista de medidas) o una medida simple.

6.2.2.1.3 Mensajes para transmisión de datos escalares.

Para la transmisión de datos escalares, habrá un tipo de mensaje denominado Paquete, que tendrá como contenido un vector de elementos de tipo Empaquetable.

El formato del paquete será el siguiente:

Serialización	Tipo	Nombre	Descripción
XDR	**Identificable**		Identificación del paquete.
XDR	**Temporizable**		Sello temporal del paquete.
XDR	**Comunicable**		Información de comunicación del paquete.
XDR	**Empaquetable**	contenido[]	Es el vector de contenido. Cada elemento es un empaquetable, con su propia especificación de serialización.

6.2.2.1.4 Mensajes para la transmisión de ficheros.

La clase Empaquetable se serializará obligatoriamente como Fichero. A continuación se muestra el formato de la clase que se ha definido para este tipo de comunicaciones:

Serialización	Tipo	Nombre	Descripción
XDR	**Identificable**		Identificación del mensaje.
XDR	**Temporizable**		Sello temporal del mensaje.
XDR	**Comunicable**		Información de comunicación del mensaje.

XDR	String	tipoFichero	Indica el tipo del fichero, si es "**NetCDF", "xml"**, o **"plano"**.
XDR	String	nombre	Como cada fichero se suele asociar a un servicio (temperatura, batimetría, planes de misión, etc)por lo que el nombre podría ser **Servicio:ip** (la ip del emisor del fichero).
XDR	**Tiempo**	fechaCreacion	Fecha en que fue creado el fichero
XDR	**Tiempo**	fechaUltimoAcceso	Fecha en que fue accedido el fichero por última vez.
XDR	**Tiempo**	fechaModificacion	Fecha última en la que fue modificado el fichero.
XDR	Empaquetable	Fichero	Se envía el/los fichero en sí en modo **raw**, bien en el formato que esté el fichero.

El Empaquetable será serializado de la siguiente forma:

Serialización	Tipo	Nombre	Descripción
XDR	Unsigned Integer	nbytes	Indica el tamaño del fichero. Esto permitirá al receptor leer el fichero por completo sin omitir información.
Binario	raw	nombre	Se envía el fichero en binario.

6.3 Clases generales

Se han implementado algunas clases generales que van a ser utilizadas por distintas clases en todo el modelo de datos. Algunas de ellas se explican a continuación.

6.3.1 Interface ObjetoID.

Esta interfaz se utiliza para toda aquella clase que necesite un identificador, que en este proyecto, son muchas. Tendrá una función para obtener el identificador (getId) y otra par asignarlo (setId).

Todas las clases que necesiten un identificador deberán heredar de esta interfaz, ya que hay determinadas clases, como la ListaElementosID, que veremos más adelante, que sólo aceptan objetos de este tipo entre sus componentes.

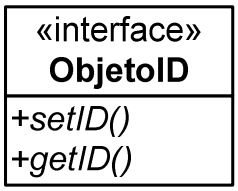


Ilustración 36: Interfaz ObjetoID

6.3.2 Clase Vector3D

Es una clase simple que contiene tres atributos: x, y, z. Estos tres valores son de tipo *double*. Además, para facilitar su gestión, y a diferencia de la mayoría de las clases que se implementan en este proyecto, los atributos son públicos, por lo que se pueden leer y modificar sin necesidad de llamar a funciones de tipo get o set.

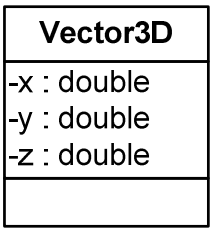


Ilustración 37: Clase Vector3D

6.3.3 Clase Lista.

La clase Lista es, básicamente, una clase que contendrá un vector de datos. No tiene control del tipo de datos que se le asigna, por lo que podría ser una mejora futura añadirle **Generics [GENERICS]** a esta clase.

Esta clase esta implementada ya que se necesitaba una clase que actuara como un vector, y, sin embargo, permitiera detectar las modificaciones que se produjeran sobre éste. Es decir, se necesitaba un vector que fuera de tipo Observable, para poder suscribir clases observadoras a su vista, con el fin de permitir, para cada clase que manejara vectores, controlar todas las inserciones, modificaciones y borrados sobre los vectores que contuviera.

La clase Lista está implementada de forma que contiene las funciones más usadas de la clase Vector. En el fondo siempre está trabajando con el vector, pero notificará las modificaciones que se hagan sobre esta clase.

Un ejemplo de utilización de esta clase en el proyecto como clase Observable se puede tener con las vistas de los disparadores. Como veremos se muestra una tabla para la lista de condiciones y una tabla para la lista de intervalos. Cada tabla estará implementada de forma que sea un observador de la lista que muestra. Una inserción, modificación o borrado de una entrada en la lista es notificada automáticamente a las vistas que se hayan suscrito a la lista correspondiente, y, por lo tanto, éstas son actualizadas.

6.3.4 Clase ListaElementosID.

Esta clase es similar a la anterior. De hecho, como atributo tiene una clase Lista, que será el vector en el que almacenará la lista de objetos. La diferencia con la clase anterior es que necesitaba que esa clase gestionara identificadores únicos para cada objeto que se le asignara. Esta clase tampoco implementa los Generics, aunque sólo admitirá objetos que implementen la interfaz ObjetoID.

Tal y como ocurría con la clase Lista, se puede mejorar esta clase implementando los Generics para controlar los tipos de objetos que se asignan a la lista de forma estática, en tiempo de compilación. Aunque en este caso, muchas veces necesitamos asignar objetos de muy distintos tipos a la lista, como puede ser la lista de componentes del plan de navegación, que puede contener waypoints, minirrutas, áreas y zonas prohibidas (todas ellas implementan la interfaz ObjetoID).

La clase ListaElementosID también heredará de la clase Observable, por lo que se notificarán todas las modificaciones que se hagan sobre ella.

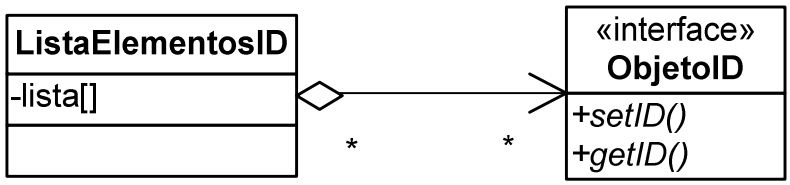


Ilustración 38: Clase ListaElementosID

6.3.5 Clase ListaMedidas

En esta clase se almacenarán, para cada una de las medidas con las que trabaja el software planificador, la lista de unidades que se pueden usar para cada una de ellas.

En principio, todas las medidas y unidades estarán almacenadas en el propio código de la clase. Una posible futura mejora podría ser la de cargar esta lista de medidas y sus unidades de un fichero que el usuario pudiera ampliar y modificar. La complicación sería que también tendría que especificar de alguna forma las fórmulas para pasar de una unidad a otra.

Esta clase tendrá funciones que devolverán la lista de unidades para una medida (getArrayStringUnidades), la lista de medidas para una unidad dada (getArrayStringMedidasUnidad) o simplemente la lista de medidas que tiene almacenada(getArrayStringMedidas).

ListaMedidas
-listaUnidades[med][uni] : String
+getArrayStringMedidas() : String[] +getArrayStringUnidades(entrada magnitud : string) : String[] +getArrayStringMedidasUnidad(entrada unidad : string) : String[]

Ilustración 39: Clase ListaMedidas

6.3.6 Clase MedidaUnidad.

Esta clase va a almacenar dos ristras, una para la medida y otra para la unidad. La unidad que se almacene en esta clase debe pertenecer a la medida almacenada.

La forma de comprobar que una unidad pertenece a una medida determinada (por ejemplo, que los Herzios [unidad] son de frecuencia [medida]) es mediante la clase ListaMedidas. Esta clase gestionará todas las medidas con las unidades que se van a utilizar en el planificador.

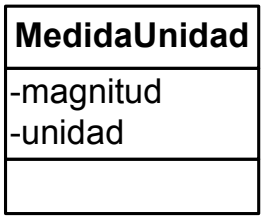


Ilustración 40: Clase MedidaUnidad

6.3.7 Clase ElementoConUnidad.

Es necesario en este proyecto implementar una clase que almacene tanto el valor que se desea almacenar como la Medida-Unidad a la que se refiere. Esta clase va a estar compuesta de un elemento tipo "Generics" y un objeto de tipo MedidaUnidad.

El elemento tipo Generics nos permitirá especificar en el momento de la declaración del objeto de tipo ElementoConUnidad el tipo de objeto que se va a almacenar como valor. Los tipos de objetos más frecuentes en el campo valor serán el tipo Double y el tipo Vector3D.

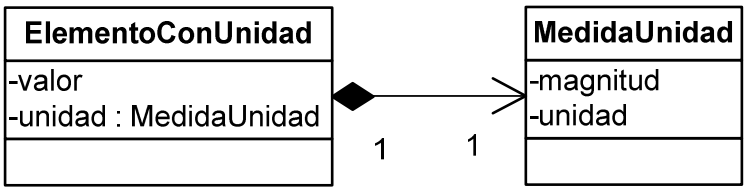


Ilustración 41: Clase ElementoConUnidad

Definición de la clase ElementoConUnidad:

```
public class ElementoConUnidad<T> {

    private T valor;
    private MedidaUnidad unidad;
…
}
```

Declaración de un objeto de tipo ElementoConUnidad:

```
ElementoConUnidad<Double> profMax = new ElementoConUnidad<Double>();
```

De esta forma, en tiempo de compilación podremos saber si estamos usando erróneamente la clase ElementoConUnidad, por ejemplo, si intentamos asignar un valor entero a la variable profMax, como atributo valor.

6.3.8 Interface Vista

Esta interface es muy utilizada en la aplicación, ya que cualquier interfaz que se desarrolle debe implementarla. Más adelante, cuando se expliquen los distintos patrones de diseño implementados, se verá porqué es tan útil esta interfaz. Por ahora se explicará muy resumidamente cuáles son las funciones que declara.

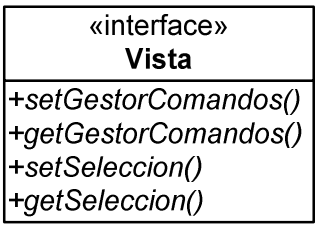


Ilustración 42: Interfaz Vista

Toda vista estará asociada a un gestor de comandos para esa vista y a un objeto de tipo Selección. Esto facilitará un comportamiento estándar de la aplicación a la hora de manejar las vistas.

6.3.9 Clase Selección

La clase selección estará asociada a todas las vistas del sistema. Se utilizará para almacenar cualquier objeto o lista de objetos que estarán seleccionados en la vista.

Este objeto es un observable que contendrá la selección actual de la vista, sobre la que se debe trabajar en caso de evento. Por ello, toda Vista deberá contener un objeto Selección. La modificación de la selección podrá ser detectada por otras vistas que se hayan registrado como observadoras del objeto Selección de esa vista.

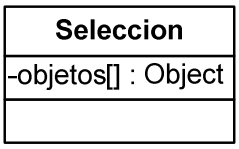


Ilustración 43: Clase Seleccion

6.3.10 Clase Opciones

La clase opciones será una clase que contendrá todas las opciones de visualización de la aplicación almacenadas en sus atributos.

Se podrá acceder a ella desde el controlador principal de la aplicación, y debido a que contiene las opciones de visualización, se podría almacenar cuando finaliza la ejecución de la aplicación para poder tener estas opciones activas cuando el usuario vuelva a ejecutar el programa.

Sólo podrá haber una instancia de la clase Opciones en cada ejecución de la aplicación, es decir, seguirá el patrón Singleton que veremos más adelante.

6.3.11 Clase AccionGenerica

Los componentes que se dibujan en las interfaces de Java, como pueden ser botones, ComboBox, etc, tienen asociados un objeto de tipo AbstractAction. Este objeto almacena algunas características del componente como pueden ser la imagen que muestra, el texto, etc. Incluso implementan una función "actionPerformed" con la que el objeto de tipo AbstractAction recoge el evento del componente y ejecuta esa función.

Esto puede ser útil si tenemos varios botones (por ejemplo, un botón en la ventana y un botón del menú) que ejecutan el mismo código cuando lo pulsa el usuario, ya que se almacenará el código en el mismo sitio, además de mostrar el mismo texto e incluso las misma imágenes en los dos botones definiendo un único objeto de tipo AbstractAction.

Por ello se ha implementado una clase que se denomina AccionGenerica que hereda de la clase abstracta AbstractAction pero en la que la función actionPerformed no está implementada, ya que debido a la gestión de eventos y al patrón Comando, es preferible que el código que gestiona los eventos se encuentre directamente en el controlador.

En la aplicación me se ha hecho un uso intensivo de uno de los atributos de la clase AccionGenerica, el ACTION_COMMAND_KEY, el cual es una string que contiene una especie de código que identificará a la acción sobre la que se ha producido un evento.

6.3.12 Clase ContenedorAcciones

Esta clase va a contener un listado de las acciones que se van a asignar en la vista. Cada una de esas acciones será de tipo AccionGenerica. Para buscar una acción en el contenedor de acciones lo haremos mediante el ACTION_COMMAND_KEY, que será el que diferenciará a cada una de las acciones.

En el patrón Modelo-Vista-Controlador (MVC) veremos cómo usamos esta clase para asignar los eventos al controlador.

6.3.13 Clase Conversor

En esta aplicación se pueden utilizar muchos tipos de medidas y unidades para las diferentes medidas y unidades. Esta clase será la que se encargará de implementar todas las conversiones necesarias entre las distintas unidades de una medida, o incluso entre distintas medidas.

Será, al igual que la clase Opciones, una clase de tipo Singleton, ya que sólo debe haber una instancia de esta clase.

6.4 Patrones de diseño usados.

Ver bibliografía **[GOF]**. Para implementar el planificador de la misión se han utilizado varios patrones de diseño, con el objetivo de que el diseño de la aplicación

quedara de la forma más organizada posible. En esta sección se mostrarán todos los patrones de diseño que se han empleado.

6.4.1 Encapsulamiento

Mencionar, antes de entrar con los patrones de diseño, que se ha intentado, salvo casos excepcionales, que todos los atributos de las clases sean privados. Con esto tendremos un mayor control sobre la modificación de éstos en la propia clase. Por ejemplo, si ocurre una modificación en un atributo de una clase, ésta podrá avisar a las clases observadoras de que se ha modificado su estado. En caso de que el atributo no fuera privado, desde otras clases se podría modificar el atributo del objeto sin que éste se enterara de ello, pudiendo ocasionar problemas de actualizaciones de vistas, o de cualquier otro tipo si ese atributo depende del estado de otros atributos en el propio objeto o en otros.

6.4.2 Patrón Singleton

El objetivo de este patrón será el de evitar que haya más de una instancia de la clase que lo implemente.

Este patrón lo implementarán, por ejemplo, las clases controladoras. Ningún controlador podrá tener más de una instancia en esta aplicación. Además, hay otras clases que sólo podrán tener una instancia durante la ejecución de la aplicación, como pueden ser la clase Opciones o la clase ListaMedidas.

En Java, para implementar este patrón de diseño se han tenido que seguir estos pasos:

- Hay que declarar un atributo de la propia clase, que sea privado y estático. Un atributo es estático cuando sólo se puede dar una única instancia de este atributo para todas las instancias que se declaren para esta clase.

- El constructor de la clase debe ser privado, para que no puedan invocarlo desde otras clases.

- Se van a implementar dos funciones estáticas, una privada, que creará una instancia de la clase en caso de que el atributo antes mencionado no se haya inicializado, y otra pública, que llamará a esta anterior, en caso también de que el atributo no haya sido inicializado. La función privada que crea la instancia va a tener el control de acceso concurrente de java "synchronized", para que sólo pueda estar ejecutándose por un único hilo cada vez, lo que impedirá que dos hilos concurrentes accedan a la vez a la misma función.

A continuación se muestra un ejemplo de la declaración del atributo para el controlador del plan de navegación:

```
public class ControladorPlanNavegacion {
```

```
    private static ControladorPlanNavegacion INSTANCIA = null;
```

El constructor será privado:

```
private ControladorPlanNavegacion() {
      … código para inicializar el controlador …
}
```

Y se definirán las dos funciones que generan la instancia:

```
private synchronized static void crearInstancia(){
    if (INSTANCIA == null){
        INSTANCIA = new ControladorPlanNavegacion();
    }
}

public static ControladorPlanNavegacion getInstancia(){
    if (INSTANCIA == null){
        crearInstancia();
    }
    return INSTANCIA;
}
```

6.4.3 Patrón MVC.

El patrón modelo vista controlador ha determinado en gran medida el aspecto general del diseño de la aplicación. Siguiendo las indicaciones de este patrón de diseño se han diferenciado tres tipos de clases: el modelo, las vistas y el controlador.

- **Modelo**: En el modelo se incluirán todos los planes que puede definir el usuario, el AUV, etc.
- **Vistas**: Las vistas serán las interfaces en Java. En ellas, en la mayoría de los casos, no se encuentra el código de control, siguiendo las recomendaciones de este patrón, sino únicamente será la forma de representar la información y de interactuar el usuario con la aplicación.
- **Controlador**: Son las clases más complejas, junto con algunos modelos, ya que implementan todo el control de la aplicación, y capturan todos los eventos de las vistas para llevar a cabo las modificaciones necesarias en el modelo.

Este patrón, también está mezclado con otros patrones de diseño en la aplicación, como pueden ser el **Observador-Observable** o el patrón **Comando**, que, como veremos más adelante, va a permitir implementar el deshacer y rehacer.

En el diagrama que se muestra en la Ilustración 44 de clases se puede ver un ejemplo de patrón MVC en el código:

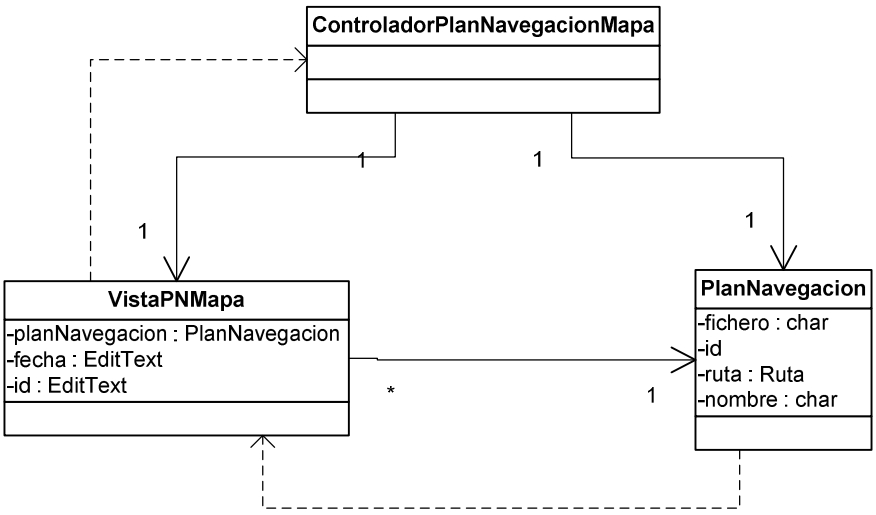


Ilustración 44: Patrón MVC

En este ejemplo, las líneas no punteadas son las asociaciones directas, mientras que las punteadas son asociaciones indirectas mediante el patrón observador observable(línea punteada entre el plan de navegación y su vista) o la captura de eventos (línea punteada entre la vista del plan de navegación y el controlador).

El controlador del ejemplo no es el controlador de la aplicación. Se trata del controlador del plan de navegación. Además vemos la vista del plan de navegación y el modelo del plan de navegación. Los eventos de la VistaPNMapa los captura el ControladorPlanNavegaciónMapa, que se encargará de hacer las modificaciones necesarias en el modelo de datos PlanNavegación. La VistaPNMapa, así como otras vistas que se hayan registrado como observadoras del modelo PlanNavegación (más adelante se explicará el patrón **Observador-Observable**, ya que esta notificación se hace mediante este patrón), recibirán una notificación de un cambio en el plan de navegación, por lo que volverá a leer el modelo de datos y se actualizará en consecuencia para mostrar la última versión de éste.

Como se ha mencionado anteriormente, no se ha utilizado el controlador de la aplicación en el ejemplo, sino el del plan de navegación. En la aplicación no va a haber un único controlador, debido a que sería una clase demasiado grande e inmanejable. El controlador principal delegará en otros controladores que se encargarán de las vistas de los distintos planes. A su vez, puede que estos controladores deleguen en otros que se encarguen, por ejemplo, del dibujo de la ruta en el mapa, debido a su complejidad, o por razones de reusabilidad del propio controlador (como puede ocurrir con el controlador de los disparadores). Se puede ver un ejemplo en el diagrama de clases de la Ilustración 45.

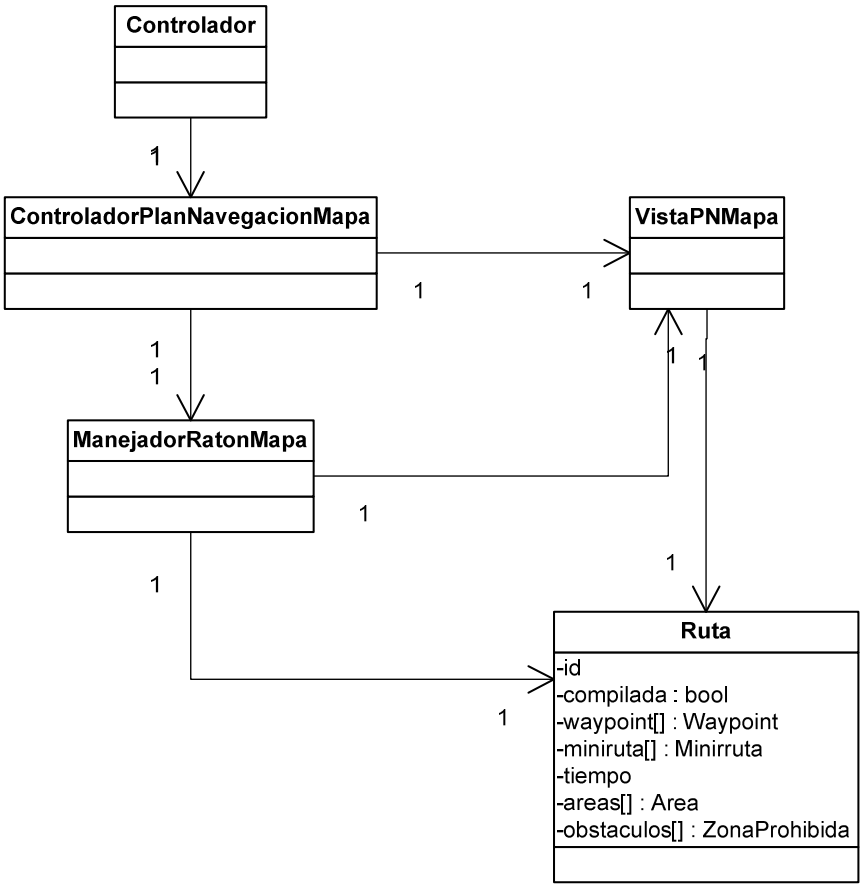


Ilustración 45: Esquema de Controladores

Por problemas de espacio y claridad, este ejemplo se ha simplificado, ya que, como veíamos en el ejemplo anterior, tanto la VistaPNMapa como el ControladorPlanNavegacion van asociados al plan de navegación. Pero se puede ver lo que se intentaba mostrar, como el Controlador de la aplicación delega en el controlador del plan de navegación, y a su vez, este delega en el ManejadorRatonMapa la tarea de ir recogiendo los eventos del usuario en el mapa para dibujar la ruta en la VistaPNMapa. Se puede ver el patrón MVC entre el ManejadorRatonMapa(controlador), la VistaPNMapa(la vista) y la Ruta(el modelo).

6.4.4 Asignación de eventos al controlador

El problema más complicado de resolver en este patrón es la asignación de eventos al controlador. Tenemos una vista y queremos que los eventos que se produzcan en ésta los capture el controlador.

Jugará un papel importante la clase AccionGenerica, explicada anteriormente. El controlador se encargará de rellenar un contenedor de acciones (clase ContenedorAcciones) con todas las acciones que va a controlar para esa vista, y se la pasará a ésta. Cada una de esas acciones tendrá relleno el texto del componente, un icono si es necesario, y el ACTION_COMMAND_KEY. El ACTION_COMMAND_KEY será el utilizado para buscar en el contenedor el objeto de tipo AccionGenerica que queremos asignar a un componente de la interfaz

La clase controladora tendrá definido un atributo de tipo "public static final String" por cada ACTION_COMMAND_KEY que va a utilizar ésta. Es decir, por cada acción que va a implementar. Se ha implementado así para que la vista pueda acceder al atributo y buscar en el contenedor de acciones que ha creado el controlador (y se lo ha pasado a la vista en la función constructora) la acción correspondiente para asignársela al componente que va a disparar ese evento (por ejemplo, un botón).

Vamos a poner como ejemplo, la acción de abrir una ventana de propiedades para un elemento (puede ser un waypoint o un área) del plan de navegación. El ControladorPlanNavegacion va a capturar el evento con el ACTION_COMMAND_KEY "Propiedades". Lo primero que se hará es definir en los atributos del controlador esta String de la siguiente forma:

```
public static final String PROPIEDADES = "Propiedades";
```

Seguidamente, en la función constructora del controlador se va a crear la acción de tipo AccionGenerica para asignársela al contenedor de acciones.

```
AccionGenerica accion = new AccionGenerica("Propiedades", null, "Ver y
modificar las propiedades de este elemento.", 0, PROPIEDADES);

contenedorAcciones.addAction(accion);
```

Al crear la vista, le pasará en su clase constructora el contenedor de acciones. La vista accederá al atributo PROPIEDADES de la clase constructora, buscará en el contenedor de acciones la acción de tipo AccionGenerica que le corresponda a ese ACTION_COMMAND_KEY y se la asignará a los menú Items y botones de la interfaz que vayan a implementar este comando de la siguiente forma:

```
bttnPropiedades.setAction(
                  contenedorAcciones.getAction(
                              ControladorPlanNavegacion.PROPIEDADES));
```

La variable bttnPropiedades contiene el botón al que se le va a asignar la acción.

Además de asignarle la acción, habrá que poner al controlador como "listener" del botón. Por ello el controlador ha de implementar las interfaces necesarias (MouseListener, ActionListener, etc). La asignación del evento se puede hacer con la siguiente instrucción en la vista:

```
bttnPropiedades.addActionListener(controlador);
```

Si le asignamos el evento ActionPerformed (con la función addActionListener), y no un evento de ratón o cualquier otro tipo de evento al controlador será más sencillo, ya que en el evento se almacena el ACTION_COMMAND_KEY del componente que lo ha generado. Podremos obtener en ese caso el command key del evento y, al ver que es "Propiedades", ejecutar el código correspondiente. El siguiente código se ubicaría en el controlador:

```
public void actionPerformed(ActionEvent e){
    String accion = e.getActionCommand();
    if (accion.equals(PROPIEDADES)){
```

```
            …
            Código para ejecutar las propiedades.
            …
        }
}
```

6.4.5 Patrón Observador-Observable

Hemos nombrado varias veces el patrón Observador-Observable. En esta sección se explicará cómo se implementa, con las clases Observer y Observable de Java, este patrón.

Este patrón de diseño se utiliza para separar completamente el modelo de la Vista. Éste tendrá la capacidad de notificar a todas las vistas que se han suscrito a su lista de observadores cualquier cambio para que éstas se actualicen con la nueva información. El modelo no sabrá nada de las vistas, ya que sólo actuará con éstas a través de la clase (interfaz en este caso) Observer.

Lo que intentamos evitar con este patrón es lo que se muestra en la Ilustración 46.

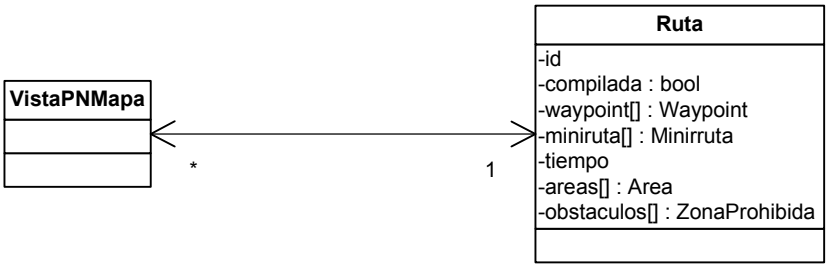


Ilustración 46: Bidireccionalidad Modelo-Vista

El modelo (en este caso la clase Ruta) no debe conocer la interfaz (en este caso la clase VistaPNMapa).

En Java, como ya se ha comentado, no se permite la herencia múltiple de clases, pero sí de interfaces (clases abstractas).

Normalmente las clases que representan vistas heredan de otras clases propias de java (JFrame, JPanel …), por lo que no es posible que hereden además de una clase Observer. Pero esto no es un problema, ya que Java dispone de una implementación del patrón Observador-Observable en la que para que una clase sea observadora, tendrá que implementar la interfaz **Observer**.

El modelo que queremos que sea observable sí que tenemos que ponerlo como clase hijo de la clase **Observable** de Java. Por lo que si implementamos una clase que hereda de otra, tendremos que hacer que la clase padre herede a su vez de la clase Observable de Java.

Se quedaría algo parecido a lo que se muestra en la Ilustración 47.

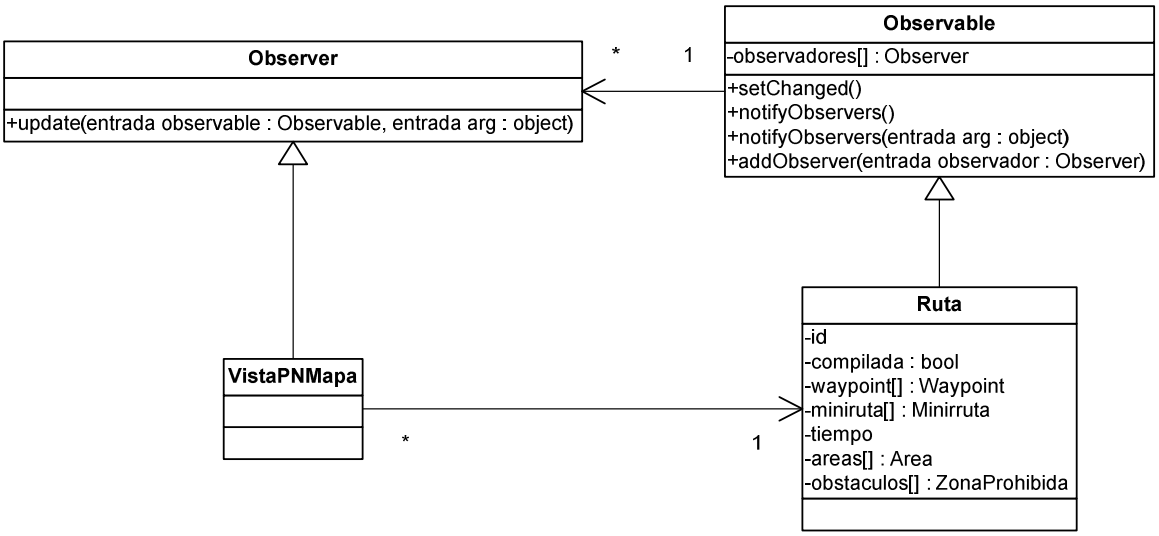


Ilustración 47: Patrón Observador-Observable

El modelo Ruta no ve a la vista VistaPNMapa, pero mediante su padre Observable puede notificar a sus observadores que se ha modificado su estado. La clase Observable implementa las funciones setChanged, para activar el flag que indica que ha sido modificada, notifyObservers, que recorrerá la lista de observadores notificando del cambio producido, y la función addObserver, que añadirá a la lista de observadores de la clase observable la nueva clase observadora.

Así mismo, la interfaz Obsever tendrá una función update, a la que se le pasará un observable y un argumento. El observable que se le pasa por parámetro será el que ha generado la notificación, y el argumento es el que le ha pasado el observable a la función "*notifyObservers(arg)*". La vista volverá a leer el estado del modelo y se actualizará.

El patrón MVC quedaría de la forma que se muestra en el esquema de la Ilustración 48.

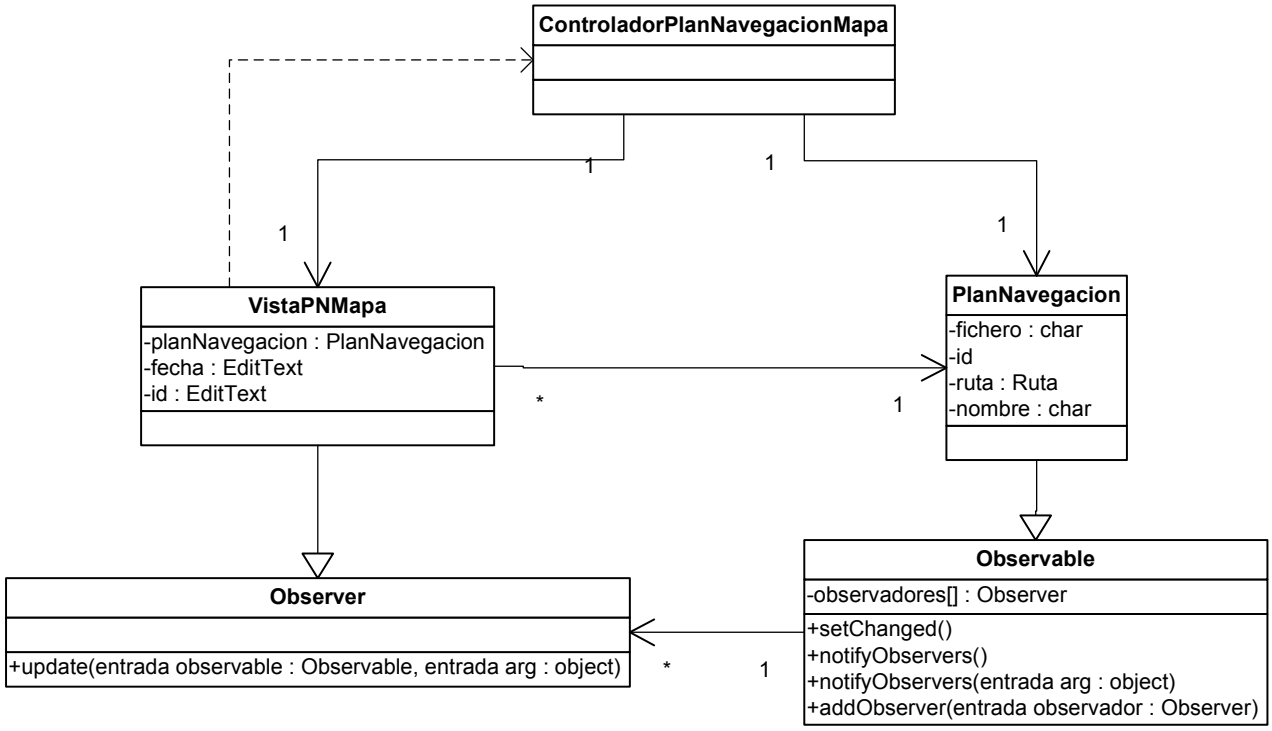


Ilustración 48: Patrón MVC con Observador-Observable

6.4.6 Patrón Comando

El patrón de diseño comando se utiliza para encapsular las acciones a realizar en objetos.

Las ventajas de aplicar este patrón de diseño son:

- Permite manipular y ampliar comandos al tratarlos como objetos.
- Permite añadir nuevos comandos fácilmente sin afectar a las demás clases.
- Permite reutilizar código, ya que un comando puede estar compuesto de varios comandos a su vez.
- Se pueden implementar añadidos para conseguir que los comandos puedan ejecutar acciones y deshacer su efecto posteriormente (undo y redo).

El controlador, cuando captura un evento y decide qué acción se va a realizar, creará un objeto comando que resuelva esa acción, pasándole todos los datos necesarios a la función constructora de la clase. Seguidamente almacenará el comando en una clase gestora de comandos, que será la que se encargará de contener una lista de comandos y de ejecutarlos en orden, pudiendo ejecutar el deshacer y el rehacer si lo permitiera el comando.

El diagrama de clases del patrón comando se puede ver en la Ilustración 49.

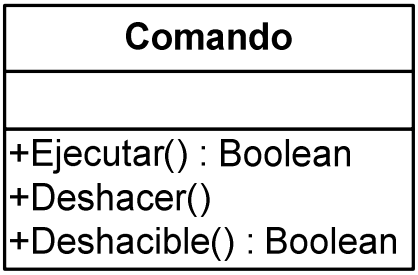


Ilustración 49: Comando

Se ha implementado la clase abstracta Comando, de la que heredarán todas las clases Comando. Esta clase se ha implementado para que todos los comandos tengan la misma interfaz. Las clases que hereden de la clase comando deberán implementar las funciones "Ejecutar", "Deshacer" y "Deshacible".

Además, se ha implementado la clase "GestorComandos". Esta clase se encargará de gestionar la lista de comandos que se van ejecutando para cada vista, para dar la posibilidad al usuario de deshacer una acción ejecutada o rehacerla si lo desea.

Como ya vimos con la interface Vista, cada clase que herede de esta interfaz deberá tener asociado un gestor de comandos. Habrá uno por cada vista, ya que el usuario puede realizar distintas acciones en cada vista, y cuando cambia de vista, al darle al botón deshacer o rehacer, no desea que se deshaga la acción asociada a la vista que ya cerró, sino sólo las acciones realizadas en la nueva vista. En resumen, cada vista tendrá su propia gestión independiente de comandos para que no se deshagan acciones de otras vistas con el botón deshacer de la vista actual.

La clase GestorComandos se muestra en el diagrama de clases de la Ilustración 50.

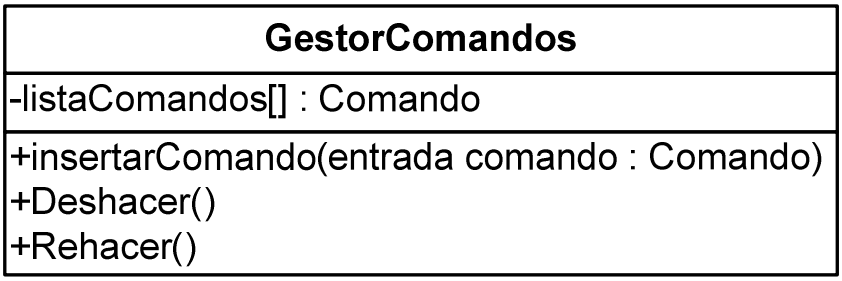


Ilustración 50: GestorComandos

Las funciones que se implementan en el gestor de comandos son:

- **insertarComando**: Esta función se encargará de insertar un nuevo comando en la lista de comandos del GestorComandos. Además, al insertar un nuevo comando, también se llamará a la función Ejecutar de este comando, para que ejecute la acción correspondiente.

- **Deshacer**: Esta clase mantendrá un índice en el último comando de la lista ejecutado, y será este comando el que se deshaga. Llamará a la función deshacer del comando y restará una posición al índice.

- **Rehacer**: el índice se sumará en uno. En caso de que haya un comando en esa posición de la lista, ejecutará su función Rehacer.

En el diagrama de clases de la Ilustración 51 se muestra un ejemplo de utilización de este patrón en la aplicación:

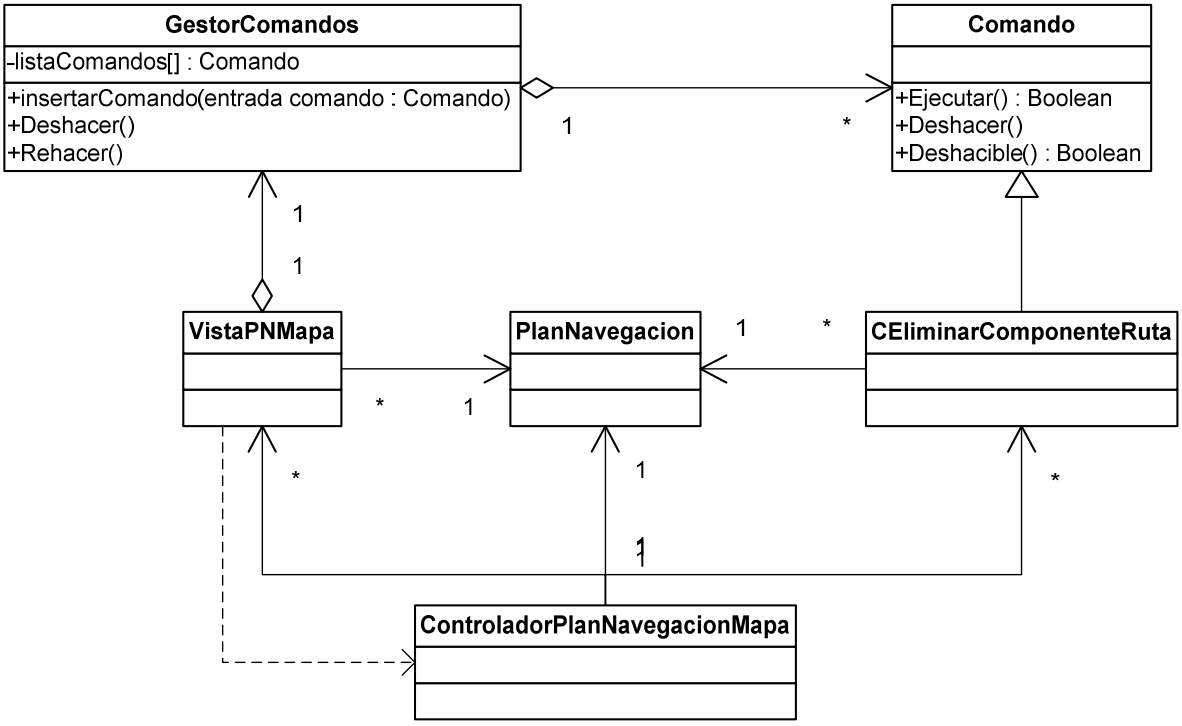


Ilustración 51: Patrón Comando

Este esquema muestra la estructura que permitirá ejecutar un comando para eliminar un componente de la ruta del plan de navegación (un waypoint, un área, etc). El controlador del plan de navegación, capturará un evento de la vista plan de navegación para eliminar un componente de la ruta del plan de navegación. El controlador deberá obtener el objeto Selección de la VistaPNMapa para conocer qué objeto está seleccionado.

El controlador no ejecutará el código para eliminar el componente, sino que creará un comando de tipo CEliminarComponenteRuta, y le pasará en el constructor todos los datos necesarios para realizar esa acción. El controlador accederá al gestor de comandos de la vista del plan de navegación a través de ésta, y ejecutará su función "insertarComando", pasándole como parámetro el comando creado.

El gestor de comandos, ejecutará el código del comando y, en caso de que lo pueda ejecutar correctamente, lo almacenará en su lista de comandos.

6.5 Modelos de datos para las misiones

En esta sección se muestran los modelos de datos definidos para representar los distintos planes. En primer lugar, se describirán los planes de comunicación, de almacenamiento, de mediciones, y de supervisión, ya que todos estos planes tienen muchos aspectos en común y se ha diseñado de forma que se reutilice la mayor cantidad de código posible.

Finalmente se describirá el plan de navegación, puesto que es bastante distinto a los anteriores.

6.5.1 Modelos de datos para los planes con lista de tareas

Tendremos una clase **Misión** que contendrá un plan de almacenamiento, un plan de supervisión, un plan de medidas, un plan de comunicación y un plan de navegación. Además, contendrá los atributos propios de la misión.

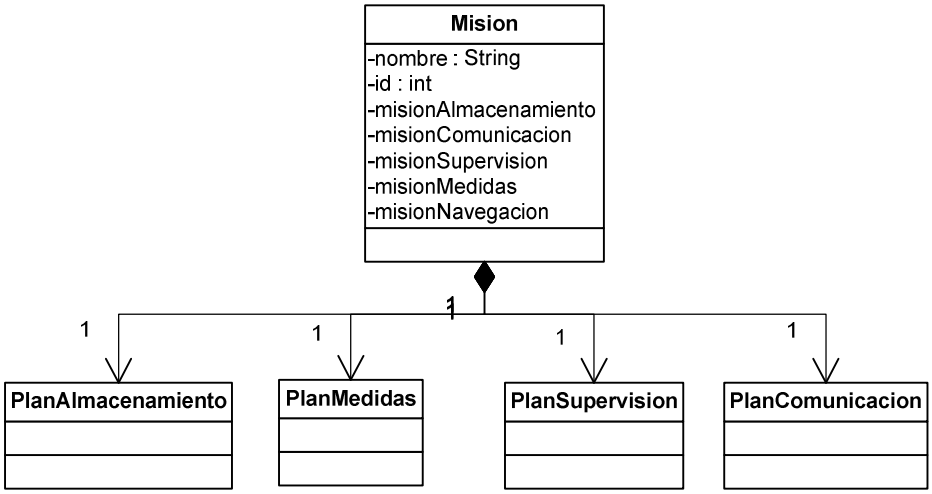


Ilustración 52: Planes con lista de tareas

Tal y como se muestra en el siguiente esquema, tenemos los cuatro planes mencionados, el plan de almacenamiento, de comunicación, de supervisión y de medidas. Estos planes heredan de una clase denominada **EltoMision**, que contiene los atributos comunes que los definen.

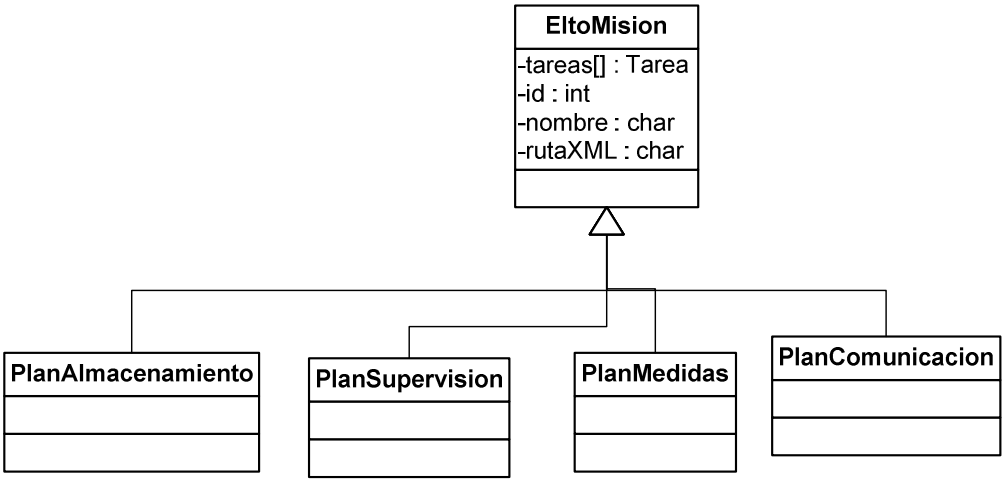


Ilustración 53: Clase EltoMision

Esta clase padre, contendrá una lista de tareas, cada una de ellas dividida en acciones y disparadores, además de un identificador, un nombre y una ruta donde está almacenado el fichero XML que lo representa.

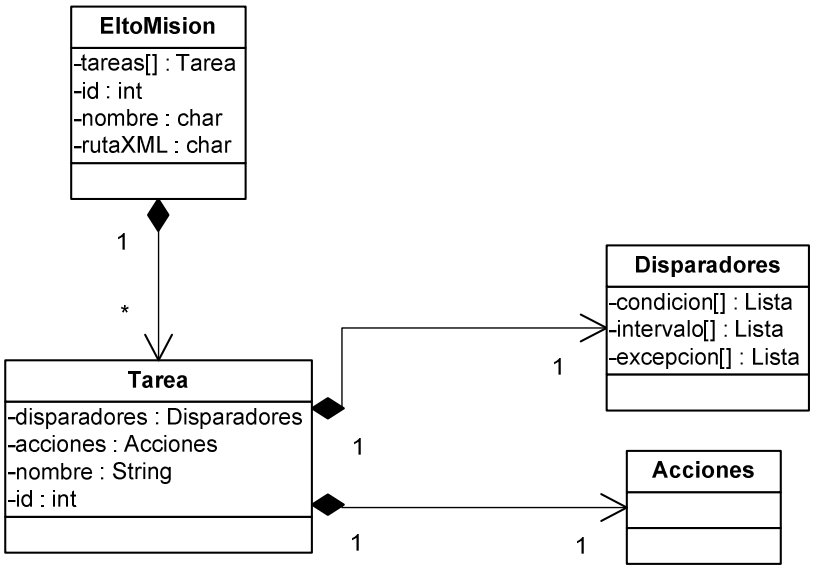


Ilustración 54: Tareas

La clase **Disparadores** albergará el conjunto de disparadores necesarios para lanzar la ejecución de las acciones de la tarea. La clase Acciones será una clase genérica. Representa todos los tipos de acciones que se pueden almacenar para cada tipo de plan distinto.

6.5.1.1 Clase Disparadores

La clase Disparadores se usa en los cinco planes que componen la misión. Su diagrama de clases se muestra en la Ilustración 55.

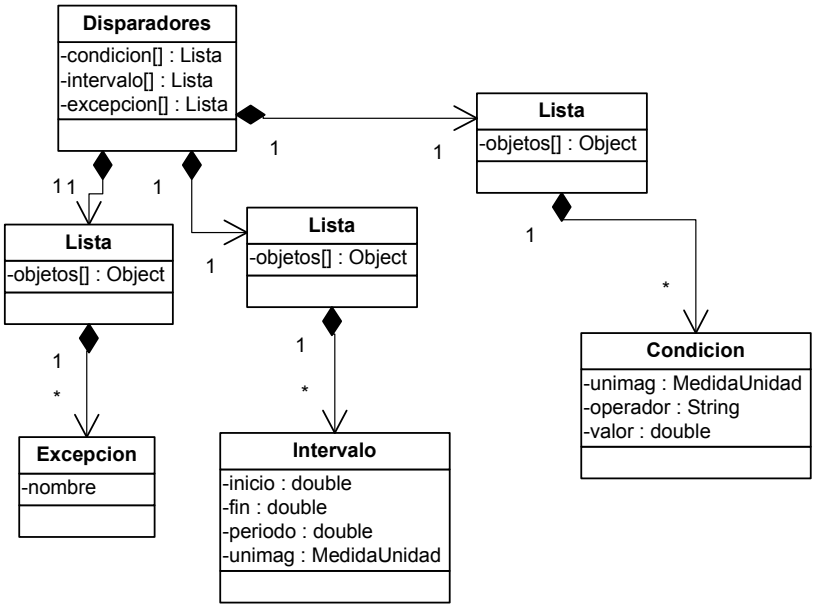


Ilustración 55: Clase Disparadores

La clase **Disparador** tendrá tres clases de tipo lista. Esto es así ya que es más fácil controlar la modificación de las listas de esta forma. Se tiene una clase **Lista** que avisa cada vez que es modificada, por lo que no hay que repetir este comportamiento para todas las listas que contenga la clase **Disparadores**, ya que está implícito en la propia clase **Lista**.

Si se utiliza un atributo tipo Vector, por ejemplo, para cada una de las listas a almacenar en la clase **Disparadores**, sería mucho más complejo controlar las actualizaciones realizadas sobre estos vectores, ya que, o se implementan funciones para actualizar estas variables en la clase, con el fin de controlar la notificación a las vistas de que el objeto de tipo Disparadores ha sido modificado, o se devuelve el vector con una función get, perdiendo la capacidad de notificación de cambios, ya que el objeto que obtenga control sobre esos vectores podrá modificarlos sin que la clase Disparadores tenga conocimiento de ello.

En resumen, la **Lista** es un Observable al que se registrarán las vistas que lo deseen, pudiendo suscribirse únicamente a las modificaciones producidas en la lista de condiciones, la lista de intervalos o la lista de excepciones, según lo que estén mostrando.

6.5.1.2 Clase Acciones

Habrá una lista de acciones distinta para cada tipo de plan, tal y como se puede ver en la Ilustración 56.

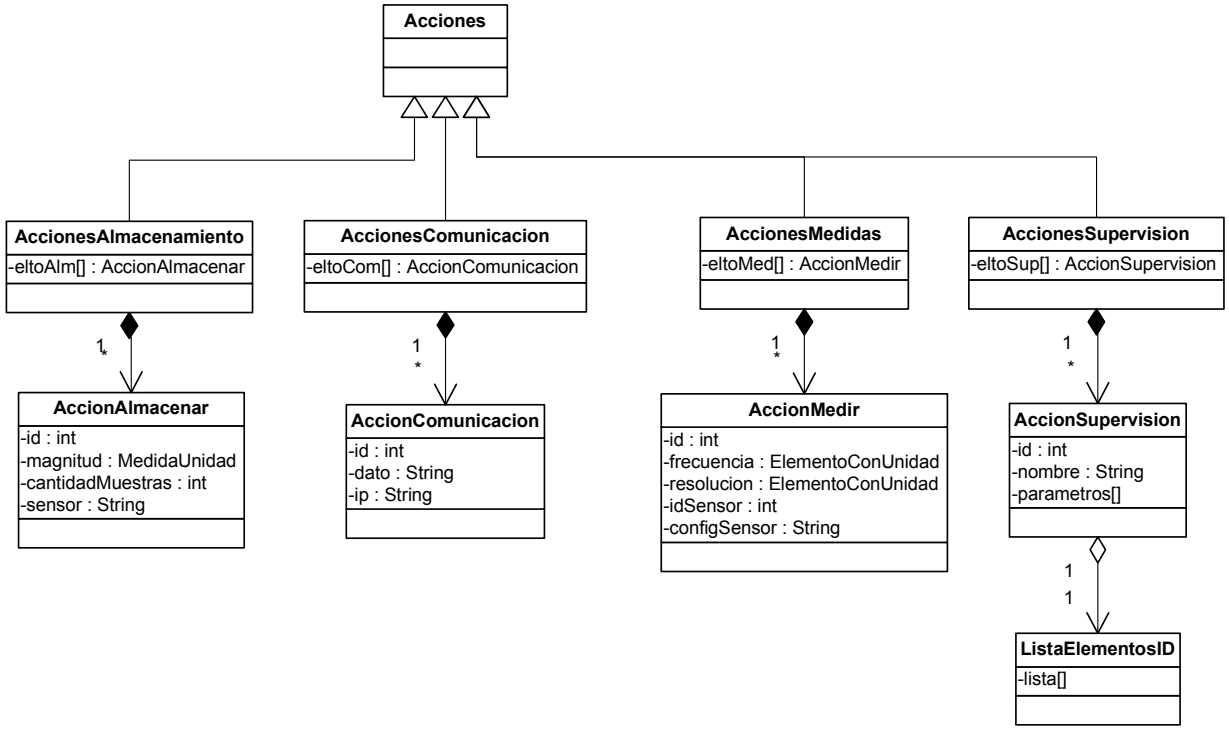


Ilustración 56: Acciones

Por lo tanto, la clase Acciones puede ser de tipo AccionesAlmacenamiento, AccionesComunicacion, AccionesMedidas, o AccionesSupervision, según el plan que estén almacenando. Si estructuramos las clases acciones de esta forma, podremos reutilizar bastante código, ya que, como hemos visto, podremos tener una única clase Tarea asociada a una clase Acciones, que será de un tipo u otro según el plan con el que se esté trabajando. No tendremos que crearnos varias clases Tarea, una para cada plan, o una función para añadir, modificar, insertar, etc para cada tipo de acción en la clase Tarea.

La clase Acciones será una clase Observable, es decir, que para cada función que se haya implementado en la clase Acciones que modifique su estado, habrá que notificar a los observadores que esta clase se ha modificado.

6.5.2 Modelo de datos para el plan de navegación

Este modelo de datos puede albergar tanto un plan de navegación en edición como un plan de navegación cuya edición haya finalizado. Su elemento principal, que contendrá todo el plan de navegación, será la clase PlanNavegación, que estará asociada a la clase Ruta, que será la que contendrá todo el recorrido que debe hacer el AUV.

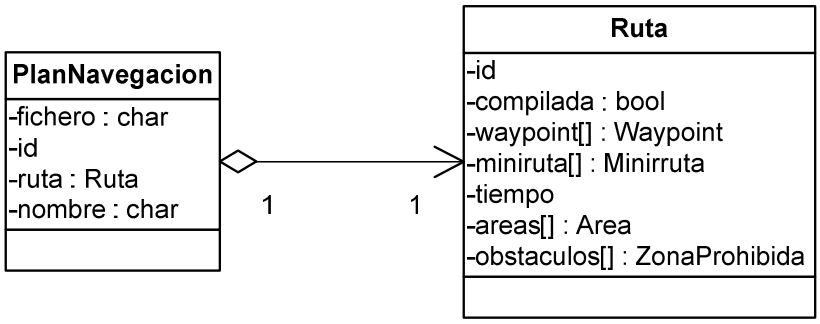


Ilustración 57: Clase PlanNavegacion

La clase Ruta será una clase bastante grande y compleja, ya que deberá gestionar todas las modificaciones que se hagan durante el tiempo de edición en la ruta del AUV. Tendrá asociada una lista de waypoints, una lista de minirrutas, una lista de áreas y una lista de zonas prohibidas. Cada vez que el usuario inserte, elimine o modifique uno de estos elementos, la clase ruta gestiona todos los cambios necesarios que se deben hacer en la ruta completa, siguiendo una serie de restricciones para esos cambios.

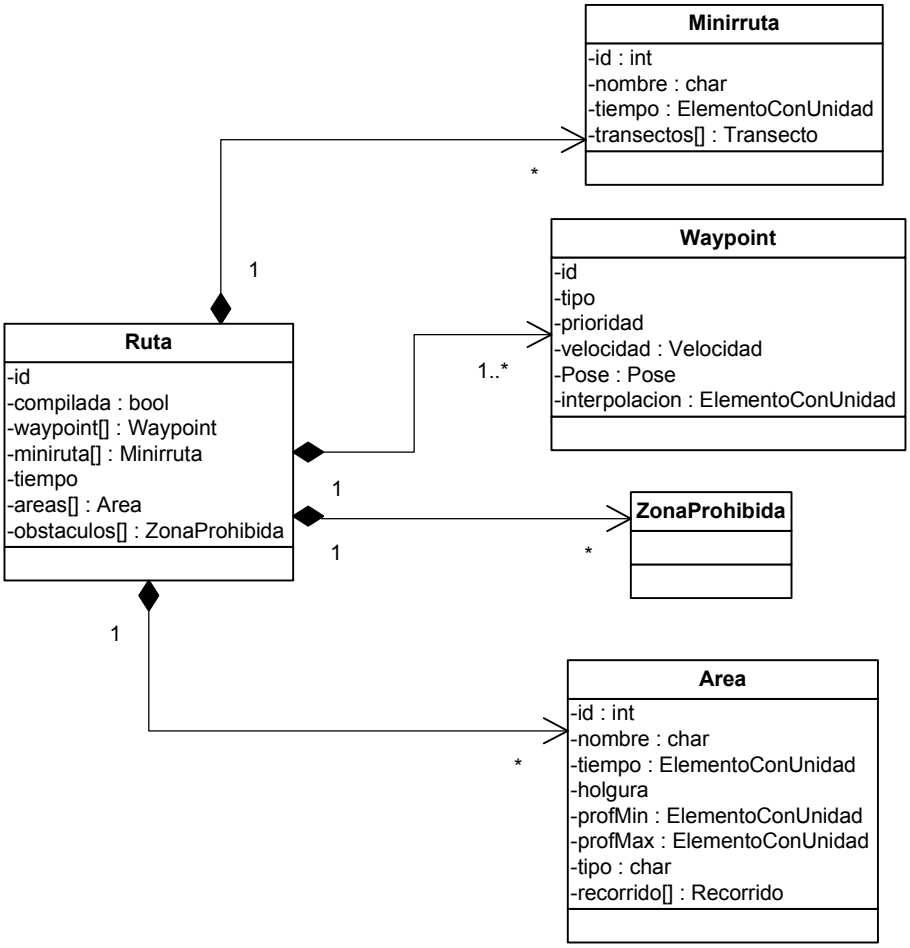


Ilustración 58: Clase Ruta

En un fichero xml que defina un plan de navegación, una ruta puede tener muchos transectos y waypoints. En el diseño de la aplicación, la clase Ruta no se identifica con ese tipo de ruta, sino que es una clase que contendrá a todos los elementos que va a recorrer el vehículo durante el plan de navegación.

Si tuviéramos que hacer un símil, un elemento ruta del fichero XML se correspondería con una Minirruta, que será la que contenga una serie de Transectos y Waypoints (estos últimos de forma indirecta a través de los transectos) asociados.

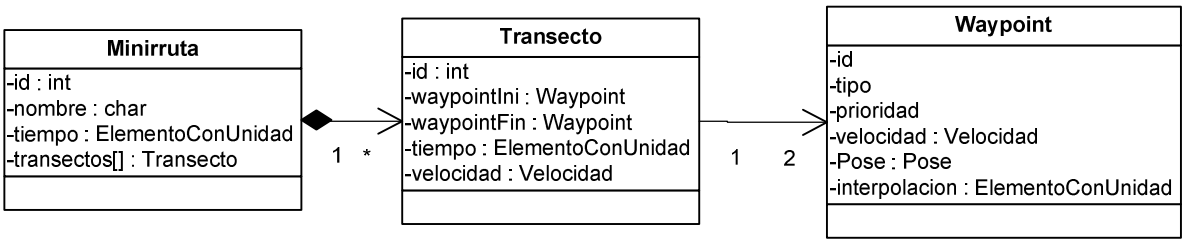


Ilustración 59: Minirruta

La minirruta contiene una lista de transectos, y éstos a su vez deben tener inicialmente dos waypoints. Opcionalmente, la clase Transecto contendrá la trayectoria a seguir en caso de que los dos waypoints no sean visibles entre sí, es decir, que haya un obstáculo en medio. En ese caso, se almacenará una lista de vértices del obstáculo que será la trayectoria que deberá seguir el AUV para ir desde el waypoint inicial al final evitando el obstáculo.

Debido a las similitudes que hay entre la clase Area y la clase ZonaProhibida, se ha implementado una clase Zona, que será la clase padre de las clases Area y ZonaProhibida, como se puede ver en la Ilustración 60.

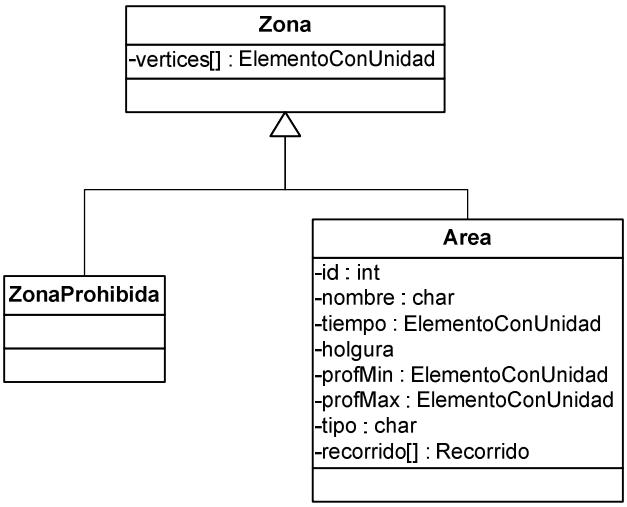


Ilustración 60: Clase Zona

La clase Zona se encargará de almacenar la lista de vértices que pertenecen a la zona, además de implementar funciones y características propias de las áreas. Por ejemplo, el algoritmo de Graham, como veremos más adelante.

La clase Zona tendrá asociada una lista de vértices. La clase Vertice implementará la interfaz del ObjetoID, y cada objeto de tipo Vertice tendrá un identificador único dentro de la zona.

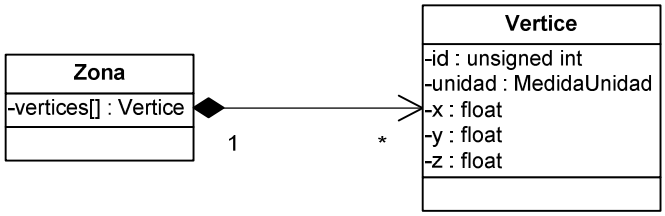


Ilustración 61: Clase Zona

La clase ZonaProhibida es una clase mucho más simple que la clase Area, ya que tan sólo tendrá que contener una lista de vértices que la conforman. Sin embargo, como veíamos durante la definición de un área para el plan de navegación, la clase Area tendrá que albergar distintas formas de recorrerla, tiempo máximo, etc.

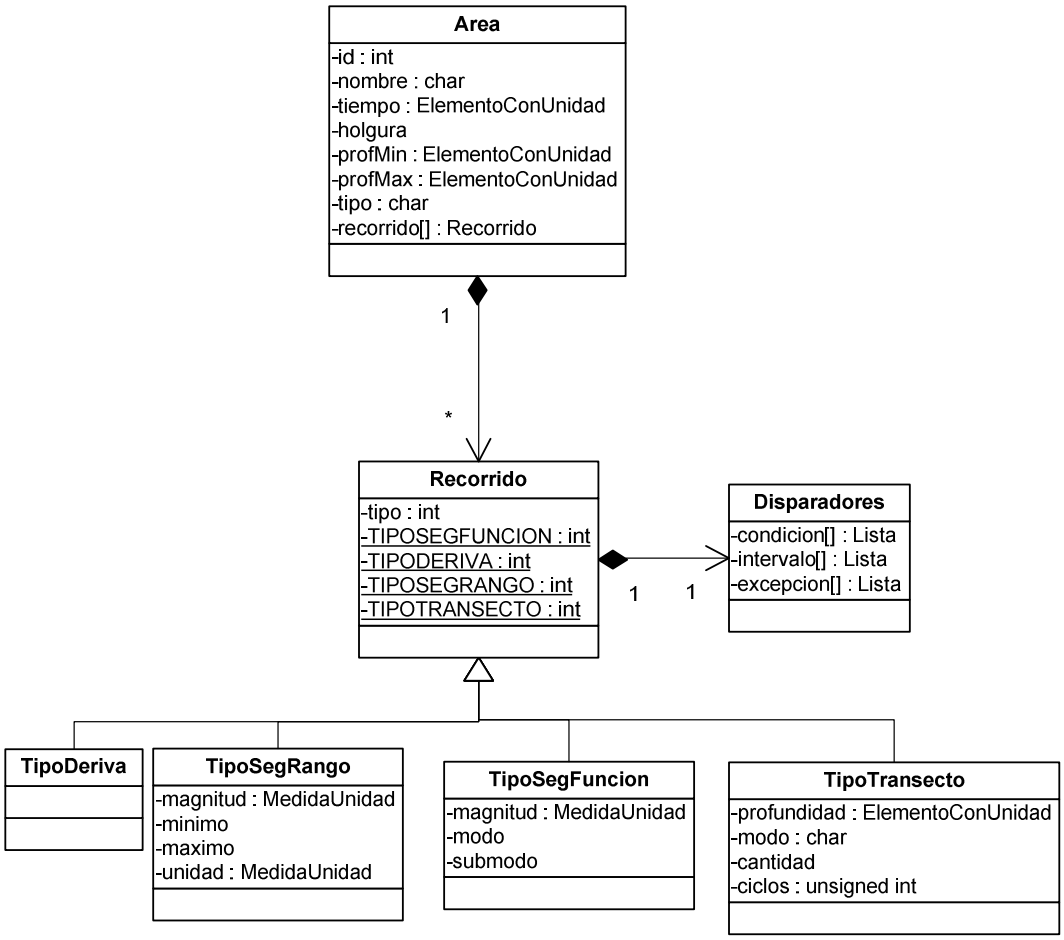


Ilustración 62: Área

Un objeto de tipo Area puede contener una lista de objetos de tipo Recorrido. Esto es debido a que, tal y como veíamos en la definición de un área, ésta puede ser recorrida de distintas formas según se activen unos disparadores u otros. Se puede ver un diagrama de clases de la clase Area en la Ilustración 62.

Hay cuatro clases que heredan de la clase Recorrido, ya que hay cuatro formas de recorrer un área: deriva, rango, función y transecto. Cada una de esas clases representará una forma distinta de recorrer el área.

La clase Recorrido, además, tendrá asociada una clase Disparadores, que indicará cuándo debe estar activo ese recorrido en concreto.

6.5.2.1 Restricciones en la edición de un plan de navegación

Cuando el usuario esté editando un plan de navegación en la ventana correspondiente, el modelo de datos del plan de navegación debe tener una serie de restricciones en cuenta para que la generación posterior del plan de navegación sea correcta. A continuación se listan las restricciones que controlará el modelo de datos para su correcto funcionamiento, y qué clase controlará la restricción y de qué forma:

Restricción 1:

Waypoints, Áreas, Minirrutas y Zonas prohibidas deben tener identificadores únicos.

Control de la restricción:

La clase Ruta será la encargada de evitar que se rompa esta regla.

La clase Ruta mantiene una única lista con estos cuatro componentes, además de otras cuatro listas, una para cada uno de estos elementos. El objetivo de la lista única es el de expresar en ésta el orden de recorrido de los distintos componentes, además de mantener un identificador único para cada elemento que se inserte en la lista. Se intenta cumplir con esta lista con la restricción del plan de navegación que se enviará al submarino, que establece que un identificador no se debe repite entre los distintos componentes del primer nivel (rutas, áreas y zonas prohibidas) de este fichero XML.

En el fichero XML los identificadores de los waypoints sólo deben identificarlos a estos entre ellos, y no con los demás componentes de la ruta. Pero en la edición de la misión necesitamos incluir a los waypoints en la lista ordenada, ya que no siempre estarán asociados a una minirruta, al menos hasta que el usuario los asocie a una, y la generación del plan de navegación lo haga. Por ello, al menos durante la edición del plan, los waypoints que no estén asociados a minirrutas estarán al mismo nivel que las minirrutas, áreas y zonas prohibidas. Al ser asociados a una minirruta se quitarán de esta lista y al ser desasociados de una minirruta se volverá a asociar a esta lista.

El problema viene cuando tenemos un waypoint asociado a una minirruta y queremos desasociarlo, por cualquier motivo. En ese caso, al desasociar un waypoint de su minirruta, tendremos que incluirlo en la lista ordenada junto con los otros tres tipos de componentes de la ruta. En el caso de que el waypoint tenga un identificador que coincida con el de cualquier elemento de la lista ordenada de la ruta, no se insertará, ya que esto lo controla la clase ListaElementosID, explicada anteriormente, y perderemos el waypoint.

Por ello esta lista no asignará un identificador ya usado previamente para otro componente(es la clase ListaElementosID la que se encarga de asignar nuevos identificadores a los nuevos componentes asignados) a los nuevos componentes que se asignen a esta lista. Siempre generará un identificador nuevo, con la ayuda de un contador que no podrá hacer más que aumentar con cada nuevo componente asignado a la lista. Entonces, cada waypoint, aunque esté asociado a una minirruta, seguirá conservando un identificador que será unívoco si y le identificará frente a los demás componentes de la ruta.

Restricción 2:

Una minirruta en el plan de navegación ya editado debe cumplir que todos sus transectos estén asociados a dos Waypoints.

Control de la restricción:

De esta restricción se encargará la clase Ruta. Cuando se genera una minirrutala clase Ruta se encargará de generarla de forma que cada transecto tenga dos waypoints y sólo dos asociados.

Sin embargo, durante la edición del plan de navegación, no tiene por qué cumplirse esta premisa, ya que el usuario estará editando las minirrutas y por tanto, en su generación, tan sólo tendrá un waypoint asociado.

Restricción 3:

Un Waypoint no debe estar contenido en más de una minirruta.

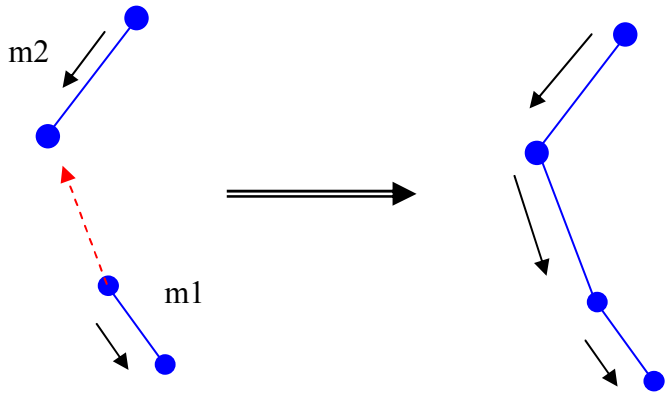
Control de la restricción:

Para evitar esto, no se permitirá al usuario, en la ampliación de una minirruta, añadir a ésta un waypoint que ya esté contenido en otra minirruta cualquiera (o incluso en la propia minirruta). Si es estrictamente necesario que un waypoint pase dos veces por el mismo punto, el usuario debe generar dos waypoints distintos posicionados en las mismas coordenadas.

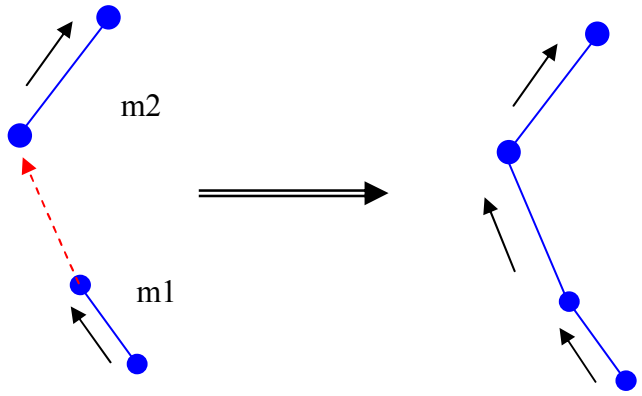
Para controlar esta restricción se ha implementado en la clase Ruta una serie de funciones que se encargarán de evitar que se añada un waypoint que ya está asignado a otra minirruta a la nueva minirruta que se está generando. Esto ocurre con waypoints intermedios, sin embargo, cuando el waypoint se encuentre en una posición inicial o final de otra minirruta, las dos minirrutas se fusionarán en una. El orden que ocupe cada minirruta dependerá de que se trate de puntos iniciales o finales.

Teniendo en cuenta que el usuario tiene la opción de decidir añadir por la parte inicial o por la final de la minirruta nuevos waypoints, tendremos los siguientes casos a la hora de fusionar minirrutas. Se denominará m1 a la minirruta a la que se le están añadiendo waypoints y m2 a la minirruta en al que se encuentra el waypoint a añadir:

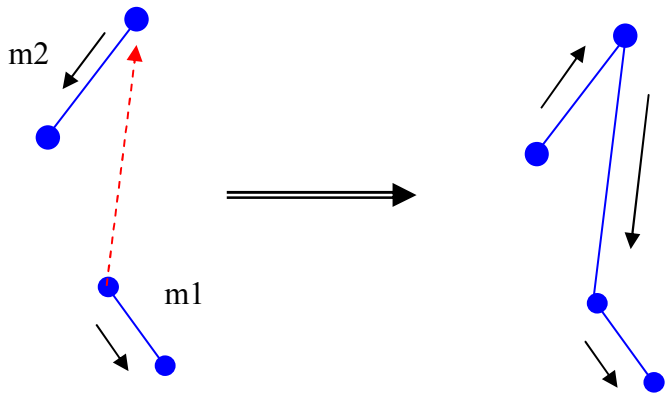
- Añadiendo por el inicio, se selecciona el waypoint final de la minirruta m2: El resultado de la fusión será una minirruta en la que la minirruta m1 se ubicará en la posición final y la minirruta m2 en la inicial. No hará falta modificar el orden de recorrido de los waypoints en este caso.



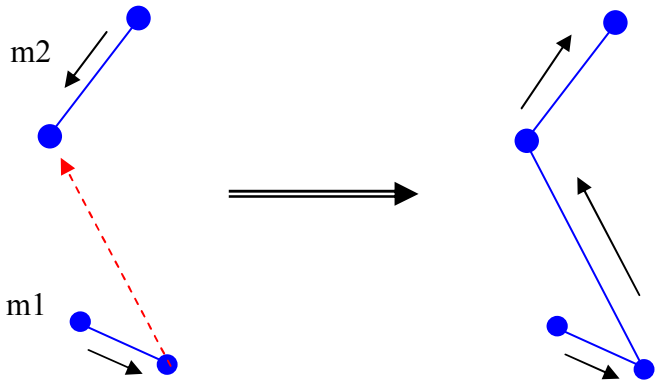
- Añadiendo por el final, se selecciona el waypoint inicial de la minirruta m2: En este caso, será la minirruta m1 la que se posicionará en la posición inicial y la minirruta m2 en la posición final de la nueva minirruta resultante de la fusión.



- Añadiendo por el inicio se selecciona el waypoint inicial de la minirruta m2: Habrá que decidir qué minirruta se considera la inicial y cuál la final. Se ha decidido que, en este caso, y al estar añadiendo un waypoint por el inicio, la minirruta m1 se ubique al final de la minirruta resultado. La minirruta m2 verá invertido su orden, ya que el waypoint inicial se ha unido al waypoint inicial de la minirruta m1, y el waypoint final de la minirruta m2 se ha convertido en inicial.



- Añadiendo por el final se selecciona el waypoint final de la minirruta m2: En este caso, la minirruta m1 será la que se posicione al inicio, mientras que la minirruta m2 se posicionará al final de la nueva minirruta resultante. Habrá que invertir el orden de recorrido de los waypoints de la minirruta m2.



Restricción 4:

No puede haber un waypoint dentro de una zona prohibida.

Control de la restricción:

La clase Ruta será la que controle esta restricción, si bien, se ha implementado de forma parcial. El control de esta restricción en tiempo de edición del plan, debería constar de dos partes:

- Al añadir un waypoint, verificar que no se encuentra contenido en ninguna zona prohibida (implementado).

- Al añadir o modificar una zona prohibida, se debe comprobar que en esta zona prohibida no está contenido ningún waypoint (no implementado).

Otra forma de controlar esta restricción es durante la generación de la ruta. Se puede mostrar un error al usuario durante la generación de la ruta si no se cumple que todos los waypoints están ubicados fuera de cualquier zona prohibida. Aunque se implemente la restricción en tiempo de edición, puede que también sea necesario implementarla en la generación de la ruta, ya que el usuario puede haber cargado un plan de navegación que no haya sido generado con el programa planificador.

Restricción 5:

Tanto las zonas prohibidas como las áreas deben ser polígonos convexos.

Control de la restricción:

Esta restricción está controlada en tiempo de edición en la clase Zona. De esta clase hereda la clase Zona Prohibida y la clase Area. Mientras el usuario establezca nuevas ubicaciones para nuevos vértices para una zona (área o zona prohibida), sólo serán añadidos los vértices que generan una nueva zona convexa. Esto se consigue con

el algoritmo de los polígonos convexos de Graham, visto en el análisis de los algoritmos de planificación.

Restricción 6:

Al finalizar la edición de un plan de navegación, éste debe contener un waypoint inicial y uno final. Esta restricción permitirá al usuario conocer el lugar de partida del AUV y el lugar donde finalizará su misión.

Control de la restricción:

Esto lo controlará la clase Ruta. En la generación de la ruta se comprueba que ésta posee un waypoint fijado como inicial y otro fijado como final. En caso de que esto no ocurra, no podrá generarse la ruta hasta que el usuario lo corrija.

Restricción 7:

Al finalizar la edición de una ruta, cuando tengamos la versión que se emitirá al vehículo para que éste la procese, todos los waypoints de la ruta deben estar contenidos en minirrutas. En el fichero XML del plan de navegación no se permiten waypoints que estén fuera de un elemento ruta. Como la Minirruta es el correspondiente a la ruta en el modelo de datos, es necesario que esto se cumpla antes de generar el fichero final del plan de navegación.

Control de la restricción:

Cuando el usuario ha terminado de definir una ruta, y antes de generar una misión con este plan de navegación, debe "Generar la ruta". Esta será una acción que se encontrará en la vista del plan de navegación, y que provocará que se ejecute un algoritmo que se encargará de unir todos los waypoints y áreas de la ruta mediante Minirrutas, además de validar otras restricciones como la comentada anteriormente en la que la ruta debe tener un waypoint inicial y otro final.

La clase Ruta tendrá una función que se encargará de generar esta ruta. Se utilizarán algoritmos para la evitación de obstáculos. En un futuro también se pueden añadir algoritmos que permitan buscar la ruta más corta en la que recorreríamos todos los waypoints y áreas.

6.6 Controladores

En esta aplicación, tal y como se comentaba anteriormente en los patrones de diseño, el control de las distintas interfaces se delega en distintos controladores, habiendo un controlador que será el que asignará el control a las distintas interfaces según las acciones del usuario.

Existe un controlador principal, denominado Controlador. Será una clase de tipo Singleton por lo que, al igual que lo que ocurre con todos los controladores, solo podrá haber una instancia de esta clase.

Para los cuatro planes que constan de una lista de tareas, es decir, todos los planes excepto el de navegación, se ha implementado una clase denominada ControladorPlan. Esta clase será la encargada de implementar todas las funciones que pudieran ser comunes a estos planes.

En la Ilustración 63 se muestra un esquema general que incluye a las clases padre de los controladores de cada plan.

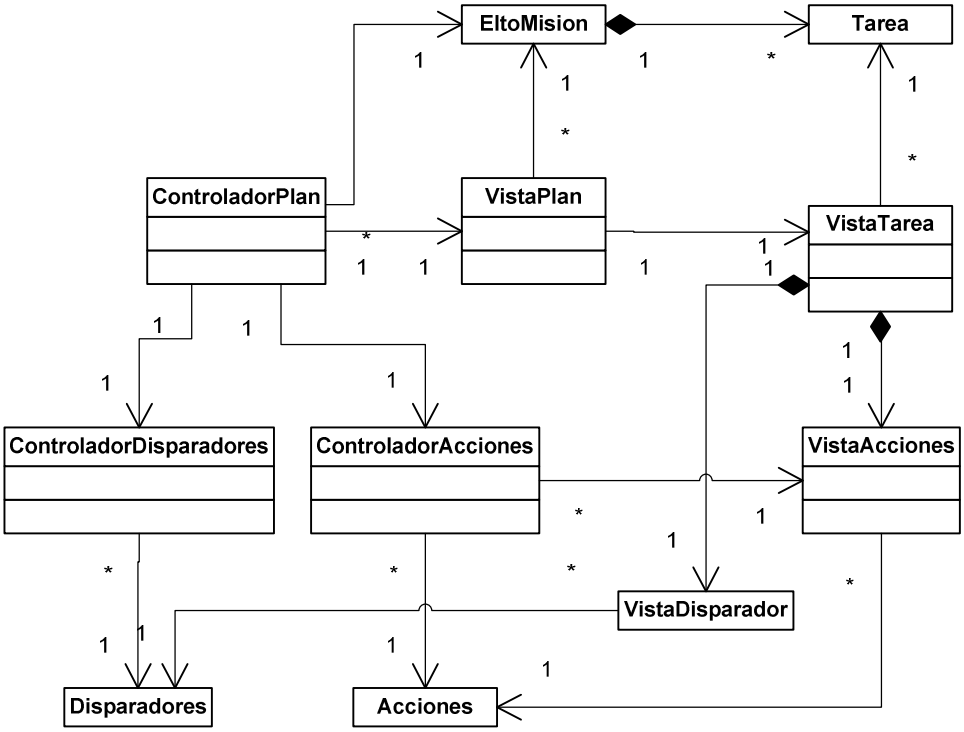


Ilustración 63: Estructura general de los planes

Se ha aprovechado que hay una clase Acciones, de la que heredan los cuatro tipos de acciones, y que hay una clase EltoMision, de la que heredarán los cuatro tipos de planes para implementar unos controladores y unas vistas lo más genéricas posibles. Con esto se consigue que el código sea lo más reutilizable posible. Esta reusabilidad se basa en que estos cuatro tipos de planes tienen una estructura similar, compuestos por una lista de tareas que contiene Disparadores (comunes a todos los planes) y Acciones (que aunque no son comunes a todos los planes, se ha implementado la clase Acciones de la que heredan todos).

Habrá un controlador para cada uno de estos cuatro planes que heredará del ControladorPlan, tal y como se ve en el diagrama de clases de la Ilustración 64.

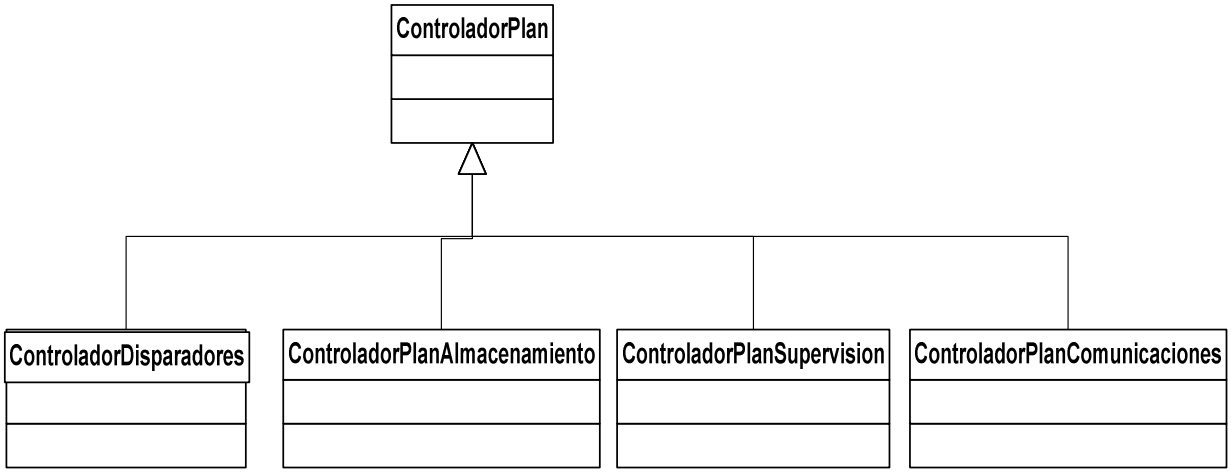


Ilustración 64: Controladores

Cada uno de estos controladores se encargará, sobre todo, de las acciones propias de cada plan, ya que las vistas de las acciones de estos planes son distintas debido a las acciones. Además mostrará al usuario el nombre adecuado del plan con el que está trabajando en ese momento y todo lo que tenga que ver con ese plan que no comparta con los demás.

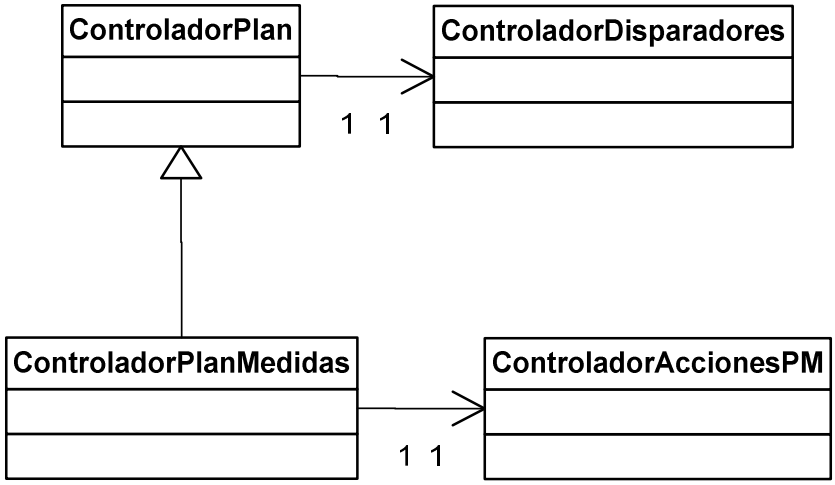


Ilustración 65: Controlador del Plan de Medidas

En este esquema de la Ilustración 65 se puede ver que el ControladorPlanMedidas estará relacionado con el ControladorDisparadores y el ControladorAccionesPM, por lo que, todo lo que tenga que ver con los disparadores lo reutilizarán todos los controladores de los demás planes.

El ControladorPlanNavegacionMapa, sin embargo, será distinto al de estos planes. Su diagrama de clases será algo parecido a lo siguiente:

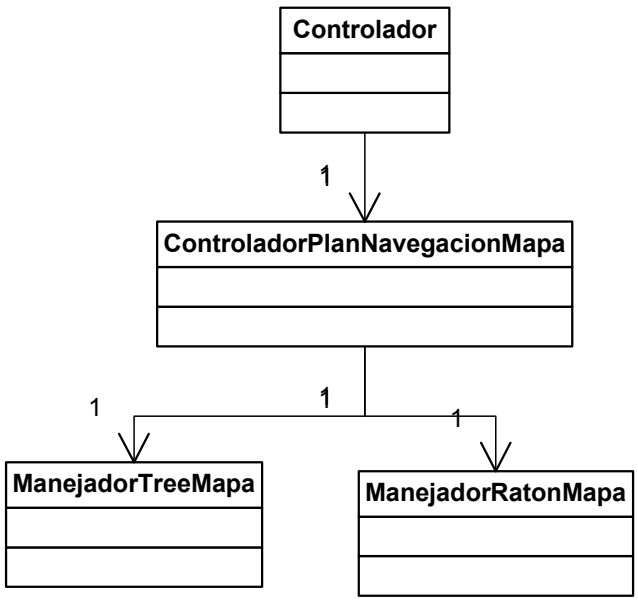


Ilustración 66: Controlador del Plan de Navegacion

Como vemos en el diagrama de la Ilustración 66, el ControladorPlanNavegacionMapa delega en las clases controladoras ManejadorTreeMapa y ManejadorRatonMapa. Estas dos clases van a ser las que manejen los eventos ocurridos en un árbol situado a la izquierda del mapa que mostrará todos los componentes de la ruta, y la clase manejadora de los eventos del mapa respectivamente.

6.7 Vistas

Como viene sucediendo con los modelos de datos y los controladores, las vistas para los cuatro planes similares de la misión van a tener muchos puntos en común. De hecho, la única diferencia de importancia será la ventana donde se especifican las acciones, que será lo único en lo que difieran estos planes.

Habrá una única clase VistaPlan, que será la que mostrará un JTree con la lista de Tareas del plan. Las diferencias entre las distintas vistas de los distintos planes serán resueltas a nivel de controladores.

A continuación se muestra el diagrama de clases correspondiente a las vistas de los planes de medidas:

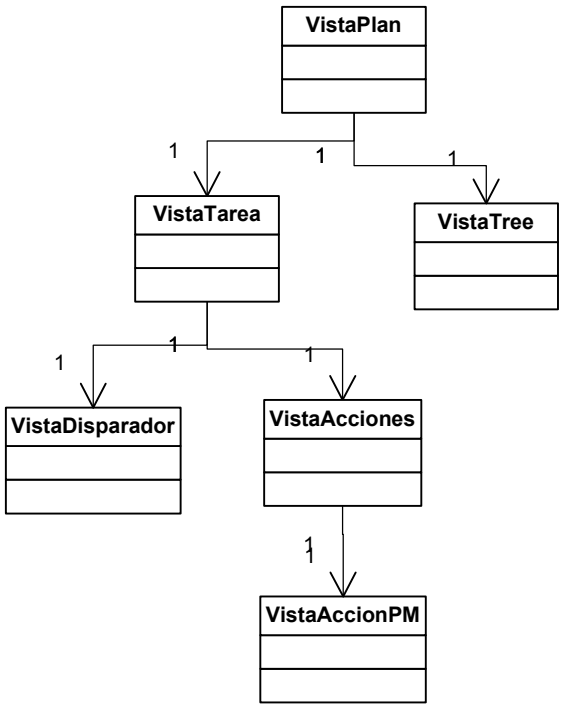


Ilustración 67: Vistas de los planes

Aunque aquí se ve que la clase VistaPlan contiene a la VistaTarea, la verdad es que la clase VistaPlan no tiene conocimiento de esta ventana en concreto. La clase VistaTarea tendrá un componente denominado JDesktopPane en java, el cual es una especie de escritorio que puede incluir varias ventanas, en el que se insertará la VistaTree, que será una vista en forma de árbol que contendrá todas las tareas del plan en cuestión, y donde el controlador de cada plan se encargará de insertar la VistaTarea cuando sea necesario y con la vista de acciones necesaria. El aspecto de este componente es el que se muestra en la Ilustración 68.

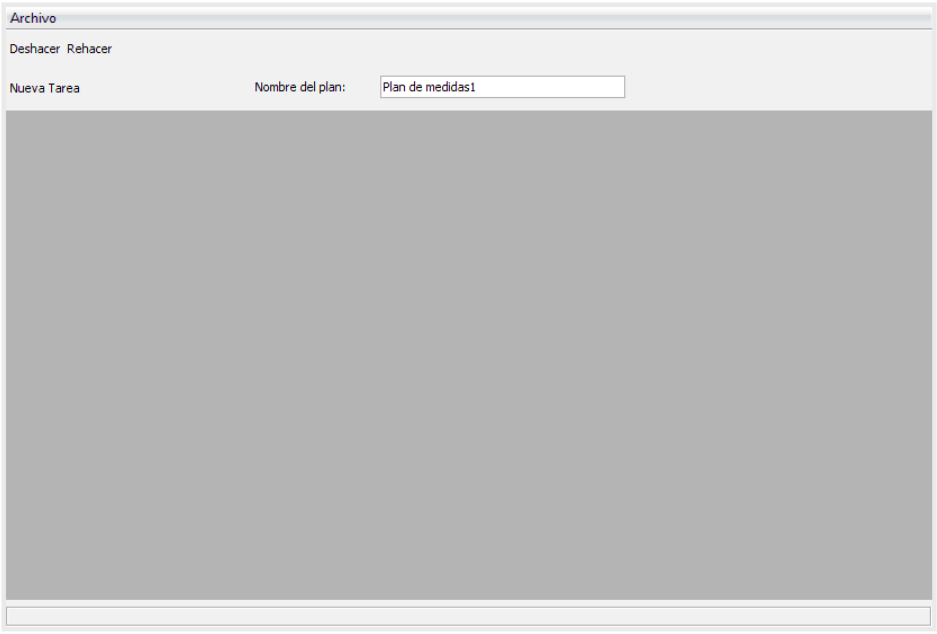


Ilustración 68: Vista del plan

Como se observa en la Ilustración 68 la zona gris oscura de la vista será donde se ubicará el JDesktopPane que contendrá las vistas VistaTree y VistaTarea. En la parte inferior de esta zona negra se encuentra una barra de notificaciones, en la que se mostrarán los errores que se vayan produciendo mientras el usuario edita el plan con el que esté trabajando.

La VistaTarea contiene una VistaDisparador y una VistaAcciones. En este caso, las vistas propias de cada plan no heredarán de la VistaAcciones, sino que la Vista Acciones contendrá, además de comandos para que el usuario pueda insertar, en una lista, distintas acciones, eliminarlas o modificarlas, un espacio donde se podrá insertar un JPanel que contendrá las especificaciones de las acciones para ese plan en concreto.

En la Ilustración 69 se muestra el diseño de la VistaAcciones.

Ilustración 69: VistaAcciones

A la derecha de los botones para añadir, modificar y eliminar se podrá insertar el panel para la vista de las acciones correspondientes (en el ejemplo del diagrama de clases anterior, la VistaAccionPM). En la Ilustración 70 se muestra el ejemplo de la VistaAccionPM que iría incrustado en la VistaAcciones:

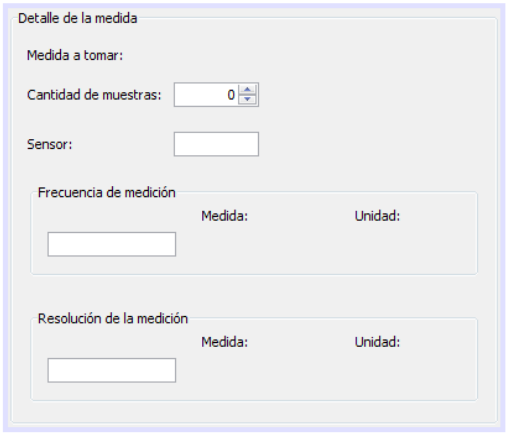


Ilustración 70: Accion Medir

La VistaAcciones estará contenida en una pestaña de la VistaTarea, mientras que la otra pestaña de esta vista contendrá los disparadores. En la Ilustración 71 vemos el diseño de la clase VistaTarea.

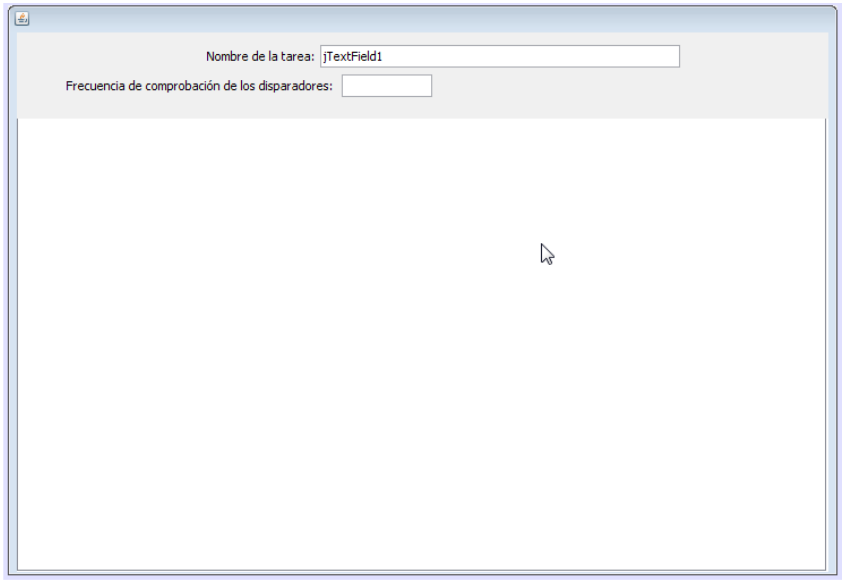


Ilustración 71: Vista Tarea

En la Ilustración 71 en la zona que se ve en blanco se asignará un un panel con distintas pestañas, entre las que se encontrará la pestaña para definir las acciones y la pestaña para los disparadores.

Cuando se introducía los controladores, se esbozó el esquema general donde se especificaba qué controladores iban a controlar qué vistas. A continuación, y una vez visto cómo se subdividen estas vistas, se van a mostrar los diagramas UML en los que se puede ver qué controladores actúan a qué nivel de las vistas.

Se muestra en la Ilustración 72 el ejemplo para el plan de medidas, los otros cuatro planes similares tienen la misma estructura.

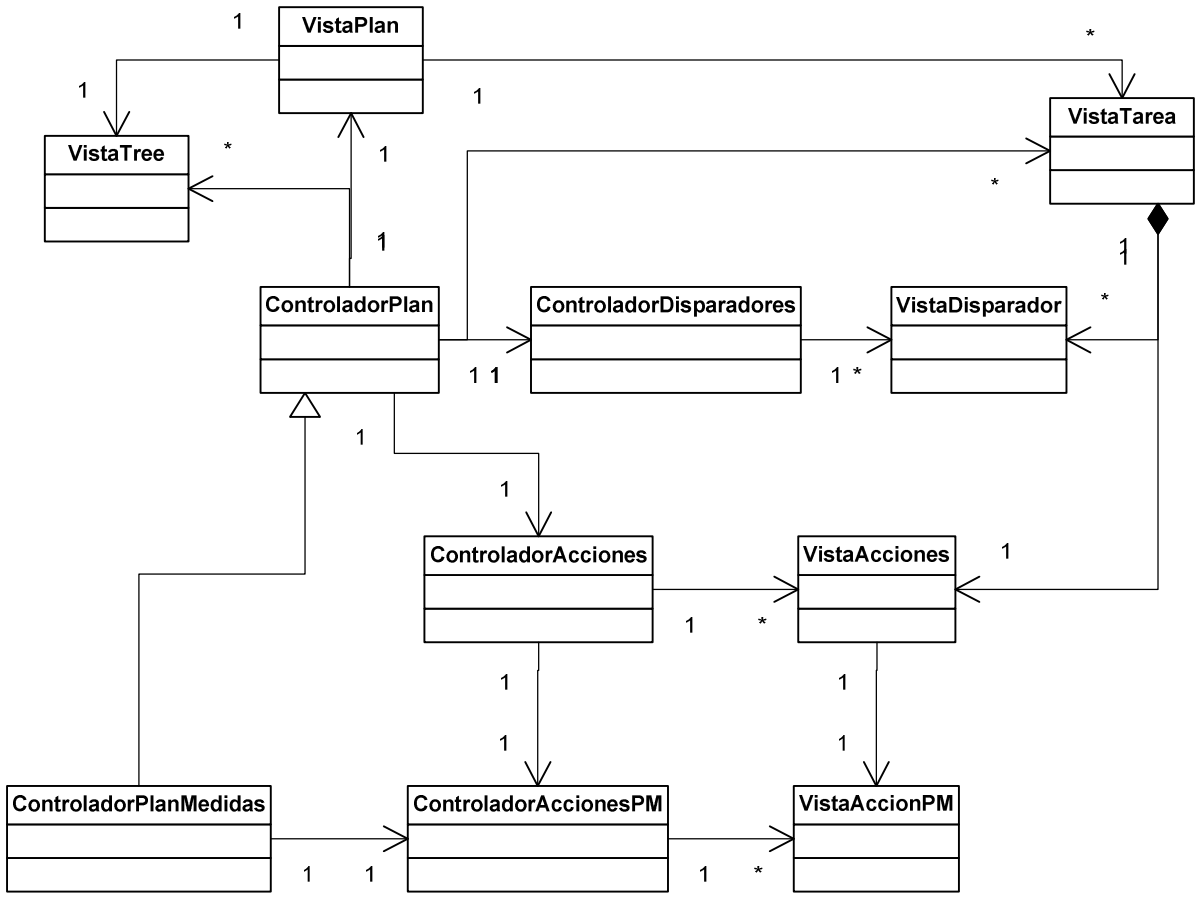


Ilustración 72: Vistas y Controladores para el plan de medidas

En ese diagrama se puede apreciar lo siguiente:

- El ControladorPlan será el que controle tanto la VistaPlan, como la VistaTree además de la VistaTarea.
- Para los disparadores, el ControladorPlan delegará el control en la clase ControladorDisparadores, que capturará los eventos producidos en la vista VistaDisparador.
- El ControladorPlanMedidas se encargará de asignar la VistaAccionesPM a la VistaAcciones y controlará los eventos que se produzcan en esta vista.
- La VistaAcciones está controlada por el ControladorAcciones, que será común a todos los planes.

El diagrama de clases para las vistas del plan de navegación será como se muestra en la Ilustración 73.

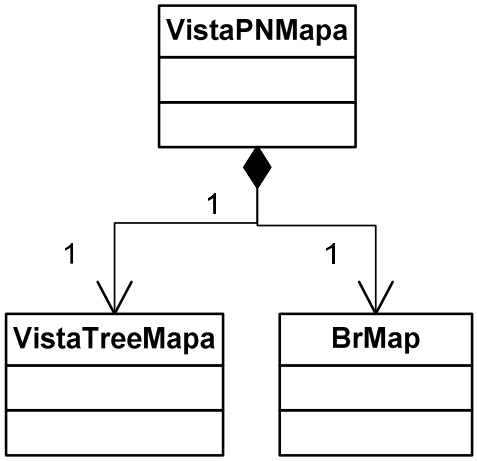


Ilustración 73: VistaPNMapa

La VistaPNMapa estará compuesta de una vista en forma de árbol que mostrará todos los componentes del plan de navegación, y un mapa (la clase BrMap). Cada una de estas vistas estará controlada por una clase controladora distinta.

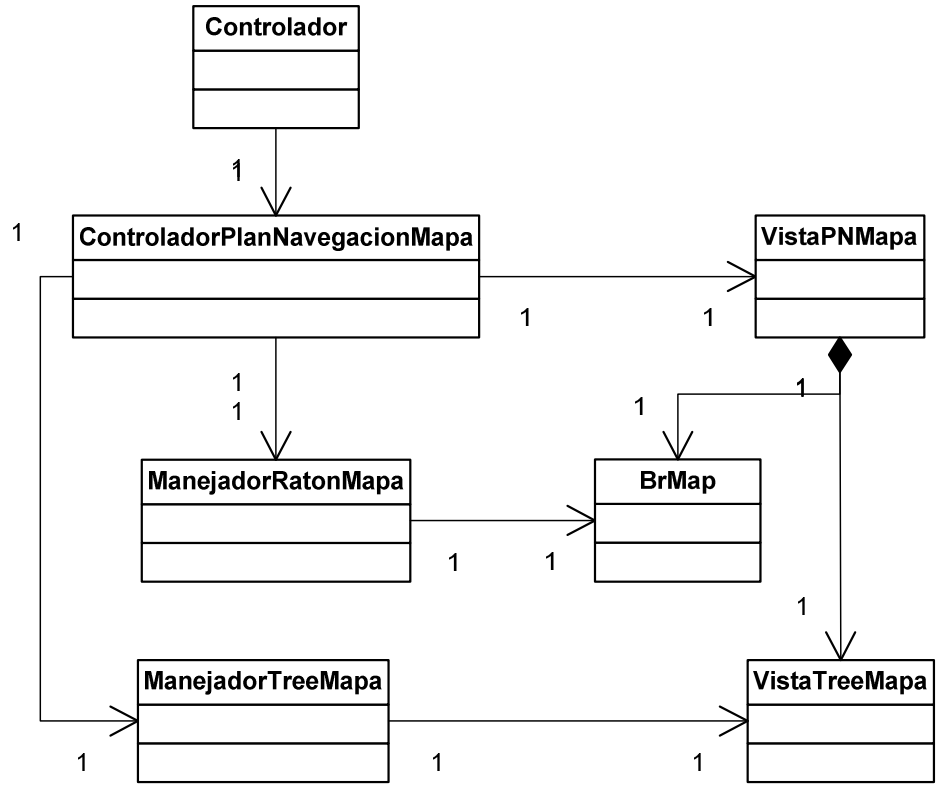


Ilustración 74: Controladores y vistas del plan de navegación

El BrMap será el mapa que se mostrará en la ventana, sobre el que el usuario dibujará la ruta a seguir por el AUV. Ésta estará controlada por el ManejadorRatonMapa, que capturará todos los eventos y los utilizará para ir dibujando la ruta.

El ManejadorTreeMapa controlará los eventos que se produzcan sobre el árbol que muestra la ruta.

En la Ilustración 75 se puede ver el diseño de la VistaPNMapa:

Ilustración 75: VistaPNMapa

Se puede ver el árbol que muestra la estructura de un hipotético plan.

6.8 Almacenamiento/carga de los planes

Los planes que define el usuario en el software planificador deben poder tanto almacenarse como cargarse cuando el usuario lo requiera. En esta sección se va a explicar el diseño de las distintas clases implementadas para este fin.

Aunque en un principio los planes de la misión se van a almacenar en ficheros XML, no se descarta que en un futuro pudieran almacenarse en otras plataformas, como por ejemplo, en tablas dentro de una base de datos. Por lo que se ha intentado que los modelos e datos sean lo más independientes posible de la forma de almacenarlo, con lo que si se modifica la forma de almacenar los planes, el modelo de datos y el funcionamiento del programa en general se conservaría intacto.

Por ello, el almacenamiento o la conversión de los datos que contienen los modelos de datos no están implementados dentro del propio modelo de datos, sino que se han creado clases específicas para este fin que, al igual que lo que ocurría con las vistas, el modelo de datos va a desconocer por completo.

Para almacenar los datos en XML se ha utilizado la librería JDOM 1.1.1. Esta librería integra el Document Object Model (DOM), y la Simple API for XML (SAX), y soporta XPath y XSLT.

Se ha creado una clase denominada GestorRecursos. Esta clase se va a encargar de servir de “fachada” entre las clases que almacenan los planes en el sistema y las demás clases de la aplicación.

6.8.1 Almacenamiento de planes

En la Ilustración 76 vemos un ejemplo con el plan de navegación.

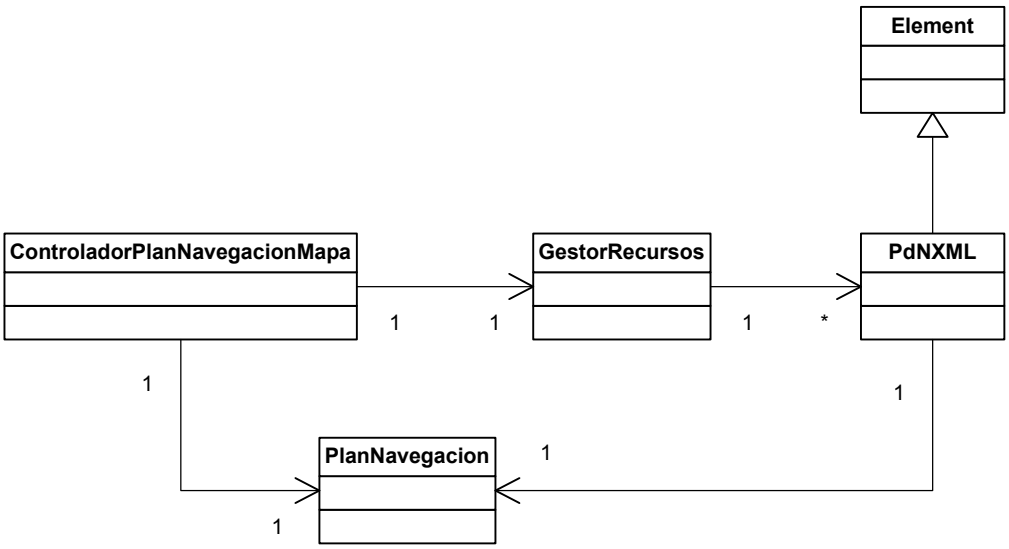


Ilustración 76: Gestor de recursos en el Plan de Navegación

Como vemos en la Ilustración 78, el Controlador del plan de navegación está asociado al gestor de recursos. Cuando necesite almacenar un plan de forma permanente, va a llamar a una función del gestor de recursos que se encargará de almacenar el plan de navegación.

La función que almacena el plan de navegación va a crear un objeto de tipo PdNXML, que será una clase heredera de la clase Element. La clase Element forma parte de la librería JDOM, y va a representar un elemento XML. PdNXML será el elemento raíz del fichero XML que queremos crear. Además, contendrá todas las funciones necesarias para generar el fichero XML a partir del plan de navegación que le ha pasado por parámetros el gestor de recursos.

Este esquema será similar para todos los planes de la misión. Estarán, cada uno de sus controladores, asociados a la clase GestorRecursos y ésta clase asociada a las clases que generarán el XML para cada uno de los planes, que al igual que en el plan de navegación, heredarán de la clase Element.

Para facilitar la implementación y comprensión de la clase PdNXML, esta clase va a estar asociada a otras clases que se encargarán de generar elementos para cada uno de los elementos del plan de navegación. Todas estas clases heredarán además de la clase Element, al igual que la clase PdNXML, ya que cada una de estas clases actuará como un elemento XML para el elemento principal, representado por la clase PdNXML. Se puede ver en el diagrama de clases de la Ilustración 77.

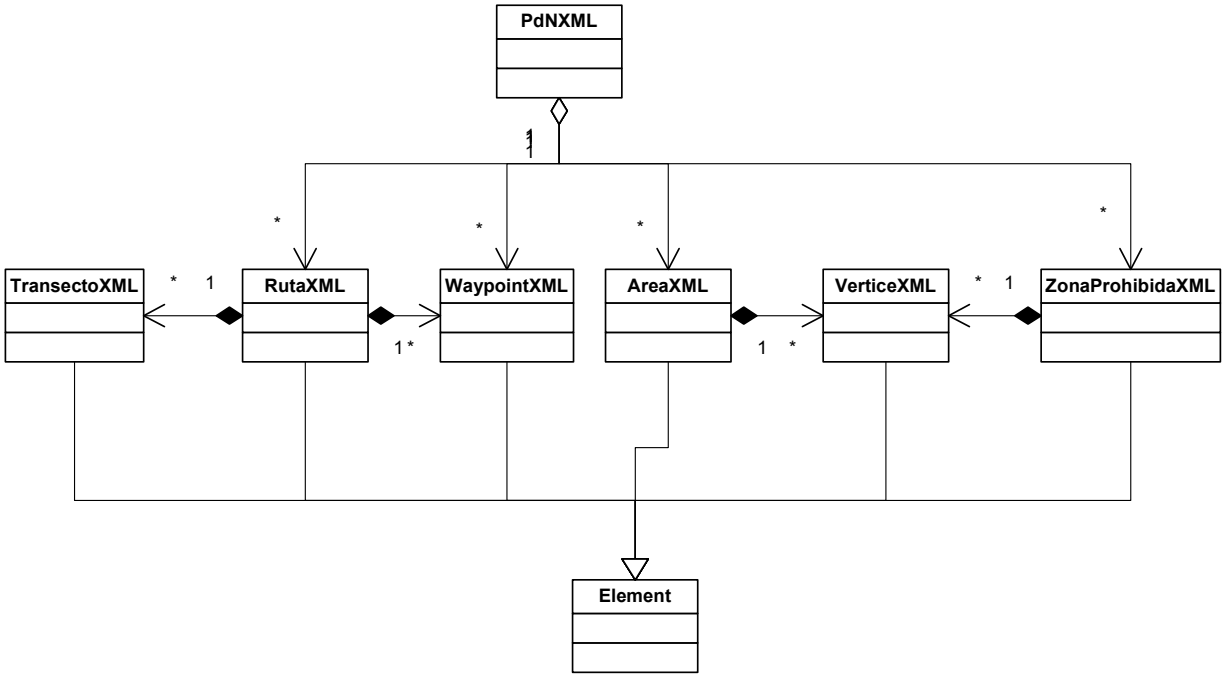


Ilustración 77: PdNXML

Como se puede ver en el diagrama de clases de la Ilustración 78, la clase AreaXML además estará compuesta por una clase RecorridoXML, que representará el elemento Recorrido del área debido a su complejidad. Esta clase incluirá a su vez los disparadores (clase DisparadoresXML), debido a que se usan en varios planes, y estará asociada a la clase ElementoConUnidad, que también se han construido con otra clase por este mismo motivo.

La clase ElementoConUnidadXML no va a heredar de la clase Element. Esto es debido a que no va a representar a un Elemento en sí, sino que va a implementar una serie de rutinas de tipo "*static*" que se encargan de construir el objeto de tipo Element necesario, ya que esta clase puede contener valores de distintos tipos (Vector3D, enteros, flotantes, …) y en función de uno de sus parámetros de entrada.

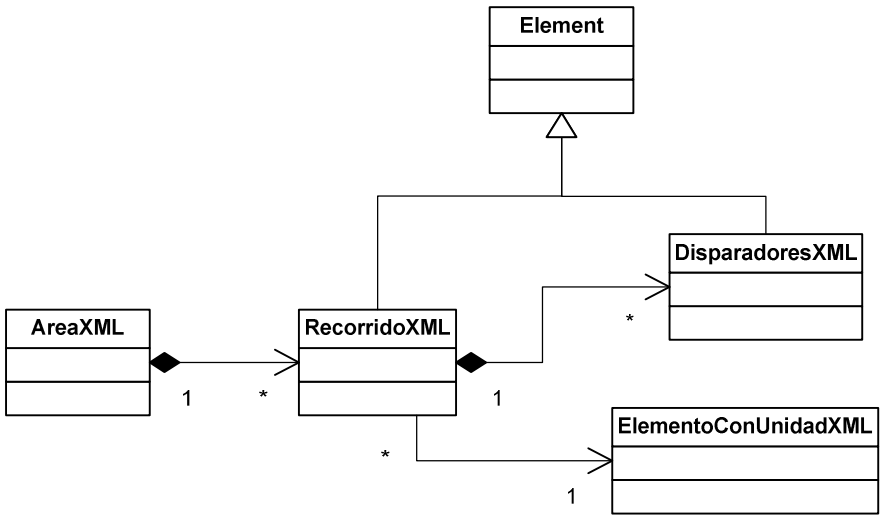


Ilustración 78: AreaXML

Se puede ver en el diagrama del PdNXML, en la Ilustración 77, que el WaypointXML también está contenido directamente en la clase PdNXML, cuando debería estar contenido únicamente en le clase RutaXML. Esta asociación se debe a que el plan de navegación que se va a almacenar puede que no esté completo, es decir, que el usuario lo esté editando aún, por lo que puede que hayan waypoints que no estén asociados a ninguna minirruta, que se traduce directamente a la ruta en el fichero XML del plan de navegación. Como se ha comentado anteriormente, el plan de navegación, una vez finalizado, no debería contener waypoints en este nivel, sino que deberían estar todos dentro de una Ruta.

Los demás planes, en cuanto al gestor de recursos, serían similares al plan de navegación. El controlador accede al gestor de recursos para almacenar o cargar un plan. La diferencia radicará en las clases implementadas para generar los ficheros XML de esos planes. Por ello no se va a volver a repetir el mismo esquema en el que se mostraba el gestor de recursos para todos los planes, sino que nos centraremos en las diferencias entre el plan de navegación y los demás planes.

En la Ilustración 79 se muestra el diagrama de clases del almacenamiento del plan de comunicaciones.

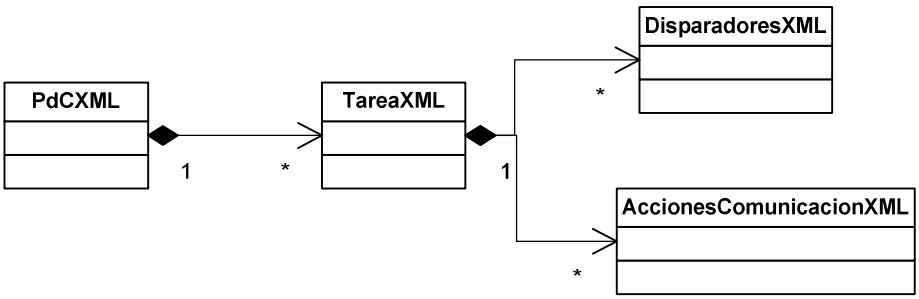


Ilustración 79: PdCXML

Las clases PdCXML, TareaXML y AccionesComunicacionXML también heredarán de la clase Element. La clase DisparadoresXML es la misma que veíamos en el diagrama de clases RecorridoXML.

Los demás planes serán similares a éste. La diferencia, como veíamos en apartados anteriores, se producirá en las acciones, que serán diferentes para cada plan. Por ello, la clase TareaXML discernirá entre cada tipo de acciones que tiene que almacenar en el fichero con una instrucción de tipo "instanceof" de Java. Esta comprobación nos permitirá saber qué plan estamos tratando, ya que comprobaremos las acciones que están almacenadas en el objeto de tipo Tarea que se le pasa a la clase TareaXML en su constructor, y por lo tanto, qué tipo de acción tenemos que comprobar para generar el fichero.

6.8.2 Almacenamiento de la misión

Para almacenar la misión tan sólo ha sido necesaria una clase, la clase MisionXML, debido a la sencillez del fichero XML que se debe generar.

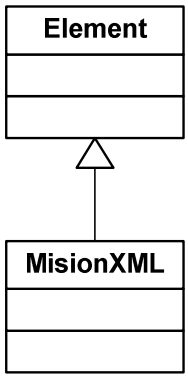


Ilustración 80: MisionXML

6.8.3 Carga de planes

Para la carga de planes, al igual que como ocurría con el almacenamiento de planes, se va a hacer a través del Gestor de Recursos.

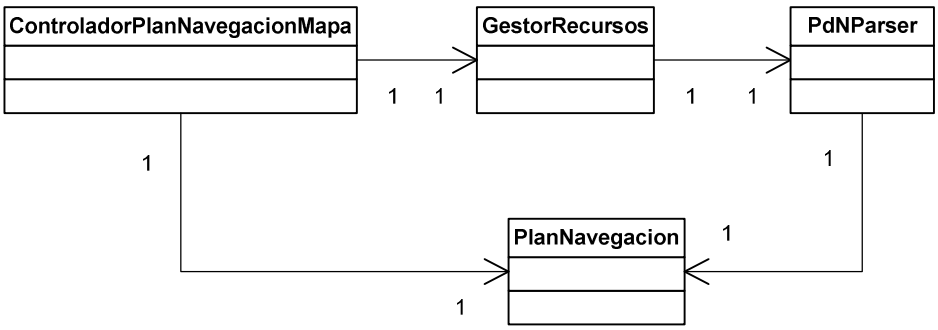


Ilustración 81: PdNParser

El ControladorPlanNavegacionMapa solicitará al gestor de recursos al cargar de un plan de navegación. El usuario habrá especificado ya en ese momento qué plan de navegación desea cargar. El gestor de recursos leerá el fichero XML y empleará la clase PdNParser para parsearlo y crear el modelo de datos solicitado.

En este caso, la clase PdNParser no va a ser una clase heredera de la clase Element, sino que va a ser una única clase que actuará de parser del fichero. No habrán más clases, como ocurría con el PdNXML, el PdNParser se encargará de cargar el fichero al completo. Para ello va a tener una función denominada ParsearDocumento, a la que se le pasará una variable de tipo Element de entrada que contendrá el elemento raíz del fichero XML del plan de navegación leído por el gestor de recursos. Esta función generará el plan de navegación con el elemento leído y lo devolverá como variable de retorno de la función.

Todos los planes van a tener el mismo esquema la cargar de fichero XML. Va a haber una clase PdCParser, una clase PdAParser, una clase PdMParser y una clase PdSParser además de la PdNParser ya vista.

6.8.3.1 Clase Parser

Pero si tenemos una clase Parser para cada plan, ¿Qué ocurre con los elementos comunes que tienen todas las clases, como los disparadores? Para ello se ha creado la clase Parser.

Las clases para las que se va a implementar el parser serán las que sean comunes a todos los planes. En principio van a ser básicamente tres:

- Disparadores
- Tarea
- ElementoConUnidad

La clase Parser no va a tener constructor, puesto que no vamos a crear instancias de esta clase. Todas sus funciones serán de tipo static. Las funciones principales que se encargarán de parsear estos elementos son:

- Parsear Disparadores: Tendrá como entrada el elemento XML que contendrá la descripción de los disparadores que se intentan parsear, y devolverá una instancia de la clase Disparadores.

```
static public Disparadores parsearDisparadores(Element e_disp)
```

- Parsear Tarea: Tendrá como entrada un elemento XML que definirá la tarea a parsear y una instancia de la clase Tarea. Esta función sólo cargará los disparadores y los atributos propios de la clase Tarea, dejando las acciones para que las cargue cada parseador concreto de cada plan. No devolverá nada puesto que las modificaciones las realizará sobre el objeto de tipo Tarea que se le pasa por parámetros.

```
static public void parsear_Tarea(Element e_tarea, Tarea tarea)
```

- Parsear ElementoConUnidad: para parsear los elementos de este tipo se ha implementado varias funciones, una para cada tipo de objeto que se le quiere asignar al atributo "valor" de la clase ElementoConUnidad.

    - Valor de tipo Double: se pasa el número a parsear como una ristra.

```
static public ElementoConUnidad<Double> LeerElementoConUnidadDouble(Element
e_pose, String x)
```

    - Valor de tipo Vector3D: Se pasará una ristra con el número de la coordenada x, otra ristra para la coordenada y o otra ristra para la coordenada z.

```
static public ElementoConUnidad<Vector3D> LeerElementoConUnidadV3D(Element
e_pose, String x, String y, String z)
```

6.8.4 Carga de la misión

La clase MisionParser se va a encargar de parsear la estructura del fichero XML de misiones. Al igual que con el almacenamiento de la misión, tan sólo se necesitará una clase, ya que este fichero tiene una estructura sencilla.

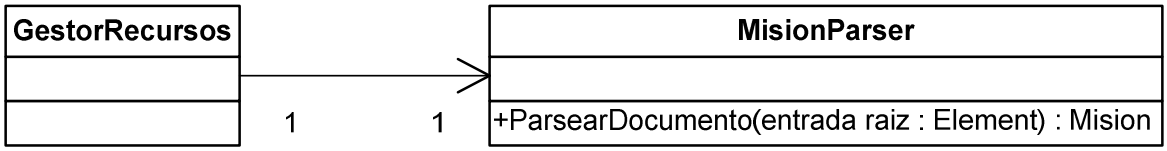


Ilustración 82: MisionParser

Parte IV. Implementación

Capítulo 7. Implementación del sistema

En este capítulo se mostrará el nivel de realización del prototipo al finalizar el proyecto. Se verán algunas de las opciones implementadas en el software y la forma en la que el usuario interactúa con el programa.

7.1 Plan de navegación.

Aunque lo ideal a la hora de implementar una aplicación de este tipo sería disponer de una base de datos que contuviera un conjunto de cartas náuticas con las que el usuario dispusiera de la información suficiente como para generar un plan de navegación, esto no era un objetivo abarcable en este proyecto, por lo que se ha optado por suministrar las imágenes con las que el usuario desee trabajar como base o mostrar un mapa con la ayuda de Google maps sobre la que el usuario planificará la ruta.

Para implementar el plan de navegación, inicialmente se planteó como una interfaz MDI (multiple documents interfaz), en la que el usuario tendría tantas vistas del plan de navegación abiertas como quisiera, y en la que en cada vista el zoom podría variar, además de la imagen de fondo que utilizara. En la Ilustración 83 se muestra un ejemplo de esta implementación:

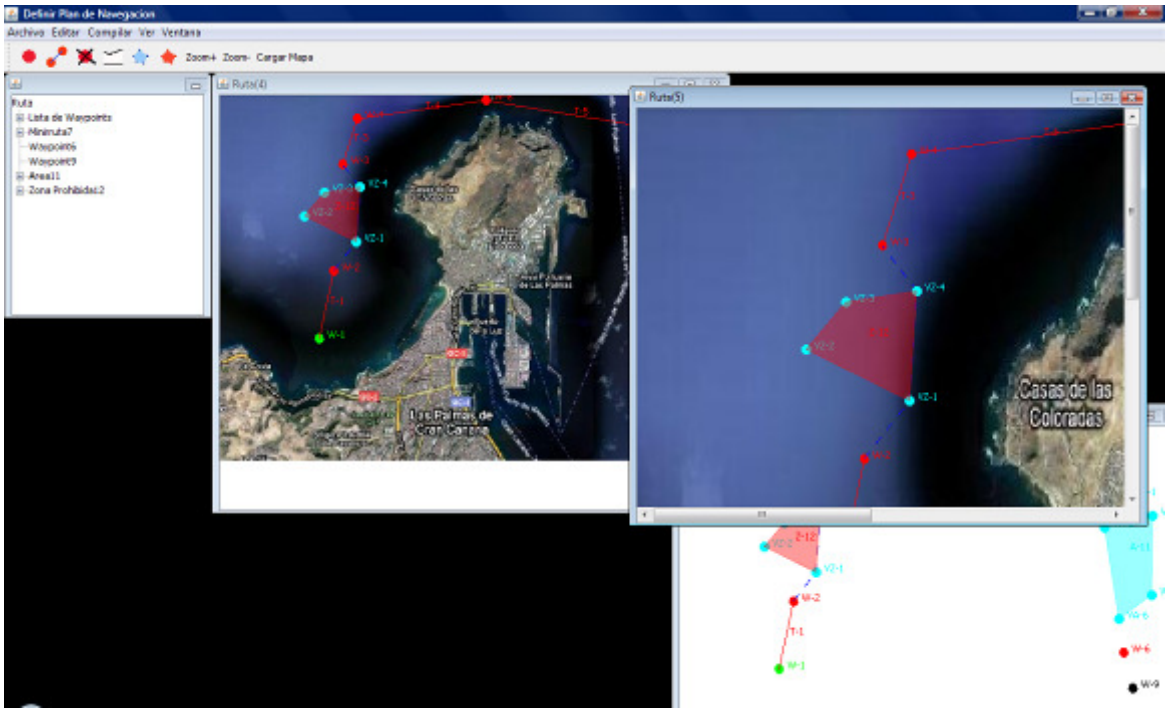


Ilustración 83: Entorno MDI para el PdN

Vemos como en la ventana del centro hay un mapa donde se muestra el islote, y donde el zoom permite mostrar la ruta al completo. En la ventana de la derecha el zoom es mayor, y en la ventana de la esquina inferior derecha ni siquiera hay un mapa cargado.

Con esta implementación podríamos tener varias vistas de la ruta, con distinto zoom, y en la que en cada ventana podríamos tener una imagen distinta. Por ejemplo, en una tener una imagen aérea de la zona mientras que en la otra vista tendríamos un mapa batimétrico, con el que el usuario podría orientarse a la hora de establecer las profundidades de los waypoints.

Al final, se optó por un entorno en el que los paneles estuvieran fijos. En este entorno, tal y como veremos más adelante dispondremos de dos paneles, el árbol a la izquierda y el mapa en un único panel a la derecha del árbol.

7.1.1 Mapas de Google en Java. Librería JDICplus.

Hay determinadas librerías en Internet que permiten incrustar un navegador en un panel (clase JPanel de Swing), como puede ser JDIC **[JDIC]** o la más moderna JDICplus**[JDIC]**. En un principio se planteó incrustar el navegador con el JDIC. A continuación se explicará cómo se utiliza JDIC, para a continuación mostrar la forma en la que se incrusta un navegador con el más moderno JDICplus.

La idea con el JDIC, para integrar el Google Maps, es la de crear una página web propia, en la que incrustar el Google Maps y almacenarla en una ruta local del equipo. Esta página web puede incluir otros ficheros que implementen funciones en lenguaje JavaScript, o estar incluidas en el propio fichero .html. a continuación se debe implementar en código JavaScript la forma de leer la información que tenemos que mostrar en el mapa de Google, y generar “overlays” (dibujos que irán superpuestos en el mapa de Google) con esa información para mostrarlos éstos sobre el mapa.

7.1.1.1 Comunicación con el navegador.

El principal problema de esta aproximación es que hay que buscar una forma de comunicar todos los datos a mostrar en el mapa a la página web que debemos crear, y además también necesitamos la comunicación en el otro sentido, desde la página web hasta la aplicación en Java, para poder capturar los eventos que realice el usuario sobre el mapa. Básicamente, las coordenadas donde está haciendo click el usuario.

La idea es que desde Java se puede generar un fichero XML con la información de la ruta cada vez que ésta sea actualizada por el usuario, ejecutar un Script de la página web que se ha creado y a la que se ha accedido desde el navegador incrustado mediante la librería JDIC en un JPanel (por ejemplo, una función de JavaScript), y que el Script se encargue de leer este fichero XML con la información actualizada, generar los nuevos overlays y mostrarlos.

Ésta sería la comunicación entre el programa en Java y el navegador incrustado en un JPanel. En la Ilustración 84 se puede ver en un diagrama.

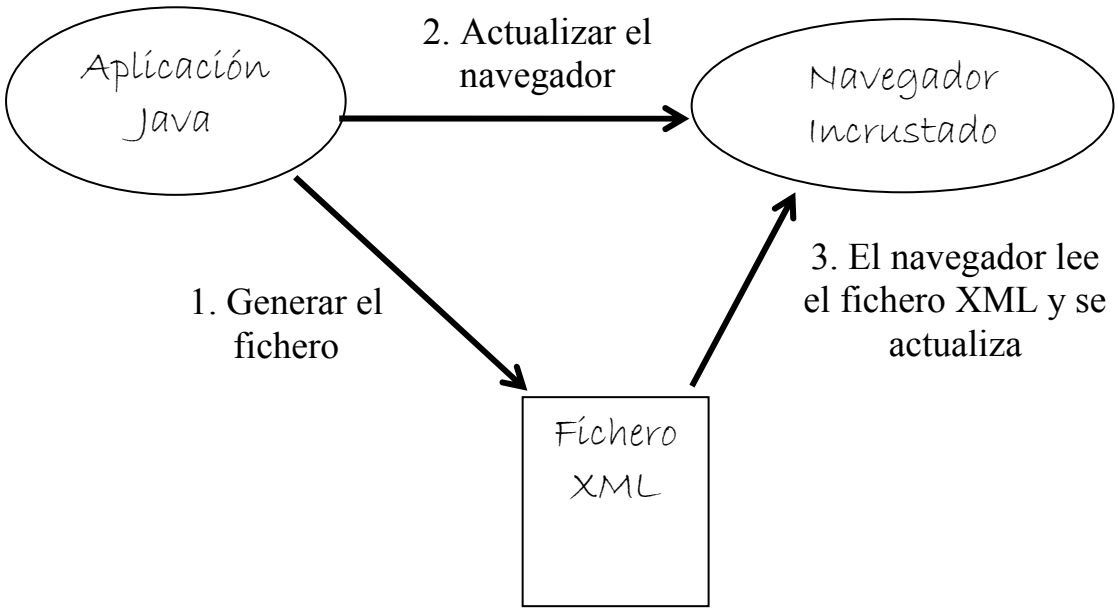


Ilustración 84: Funcionamiento del mapa

En este diagrama, en el punto 2 no se va a actualizar la página web por completo, sino, como se mencionaba antes, se llamará a una función implementada en ésta que se encargará de actualizar los datos necesarios para que el usuario no sufra la demora que supondría una actualización completa de la página web.

Un ejemplo de cómo quedaría este tipo de mapa lo podemos ver en otra aplicación realizada para otro proyecto en el que el objetivo era el seguimiento y control de boyas marinas. Se puede ver este programa en funcionamiento en la Ilustración 85.

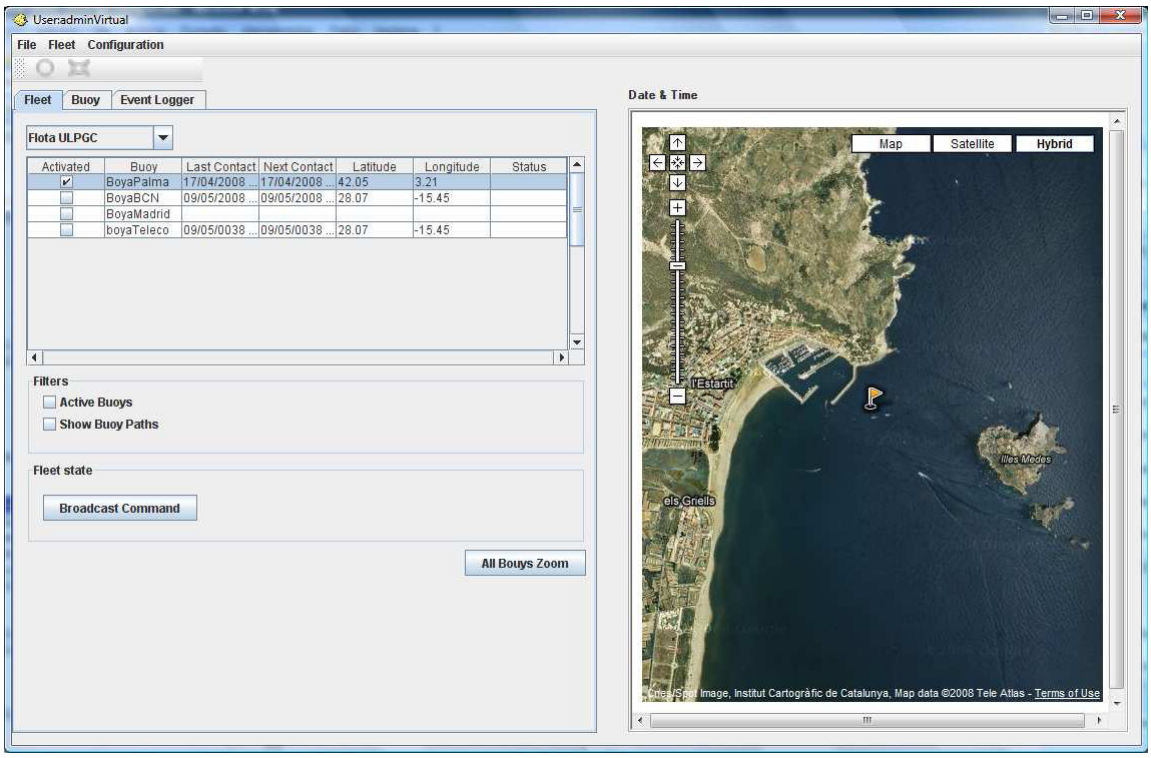


Ilustración 85: Aplicación de seguimiento de Boyas Marinas

A la izquierda hay una tabla en la que el usuario puede seleccionar la boya a mostrar en el mapa, y a la derecha está el mapa con la boya, aproximadamente en el centro del mapa. También podremos mostrar información extra cuando el usuario selecciona la boya en el mapa, tal y como se muestra en la Ilustración 86.

Ilustración 86: Información de una Boya

Una vez llegados a este punto, ya hemos resuelto el problema de la comunicación entre Java y el navegador. Ahora hay que resolver el siguiente problema, la comunicación entre el navegador y Java.

La librería JDIC incluye la posibilidad de capturar determinados eventos del navegador. Esta característica nos podría ayudar en nuestro objetivo. Es posible capturar actualizaciones, por ejemplo, de la barra de estado del navegador.

Desde el navegador, y con funciones implementadas en JavaScript y la API de Google Maps, es posible capturar los eventos de clic del usuario sobre el mapa y

obtener las coordenadas en las que se ha producido el clic. Una vez que tengamos las coordenadas seleccionadas por el usuario, el código implementado en JavaScript actualizará la barra de estado del navegador, la aplicación construida en Java detectará esta actualización y recogerá la ristra que se ha almacenado en ella. En esta ristra, la función correspondiente de JavaScript habrá insertado los datos de las coordenadas en las que se ha producido el clic.

La aplicación Java tendrá que leer esa ristra y parsearla para obtener las coordenadas. Una vez conocidas las coordenadas, actuará de una forma u otra según la acción que esté realizando el usuario sobre la ruta. Se explica en la Ilustración 87.

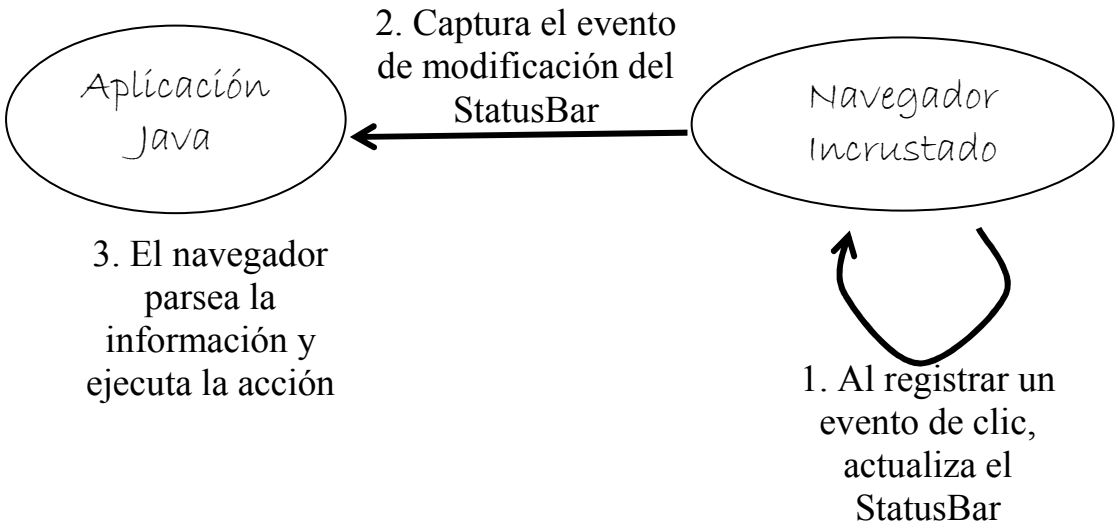


Ilustración 87: Recoger eventos del navegador

La aplicación, después del paso 3 comenzará con el diagrama anterior, modificando el XML y actualizando el navegador.

7.1.1.2 Google Maps con el JDICplus.

El JDIC plus es una versión más moderna y sencilla de utilizar que el JDIC. La mayor ventaja de esta librería es que ya tiene preparada una clase, denominada **BrMap**, que implementa lo que hemos visto antes. Es decir, un JPanel, con Google Maps incrustado y, lo mejor de todo, podemos utilizar otra clase, denominada **BrMapSprite**, que representará los polígonos que se dibujan en el mapa. Tendremos un objeto de este tipo por cada elemento de la ruta que queremos representar.

Esta clase además nos permitirá capturar los eventos de ratón que se produzcan sobre el mapa. Tan sólo tendremos que capturarlos en la aplicación Java y generar BrMapSprites que dibujen los elementos de la ruta en el mapa.

En la Ilustración 88 podemos ver un panel con Google Maps usando JDICplus que ilustra su uso.

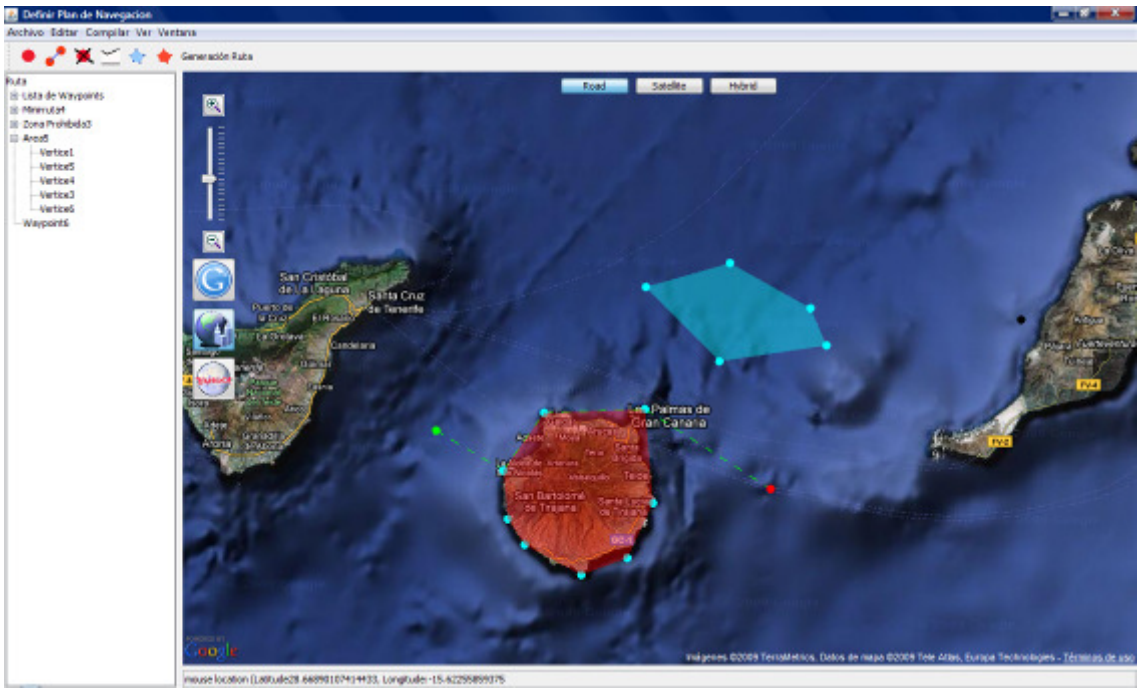


Ilustración 88: Ruta de prueba

Ilustración 89: Ruta de prueba(2)

En la Ilustración 89 vemos la zona prohibida, que incluye a toda Gran Canaria, y la evitación de obstáculos implementada.

7.1.1.3 Inestabilidad de JDIC plus y alternativas.

Añadir la librería JDIC plus al proyecto ha vuelto a la aplicación un poco inestable. Esto hace que se haya pensado en intentar en un futuro usar alguna alternativa para JDICplus, como puede ser la librería, también de libre distribución, OpenMap**[OPENMAP]**.

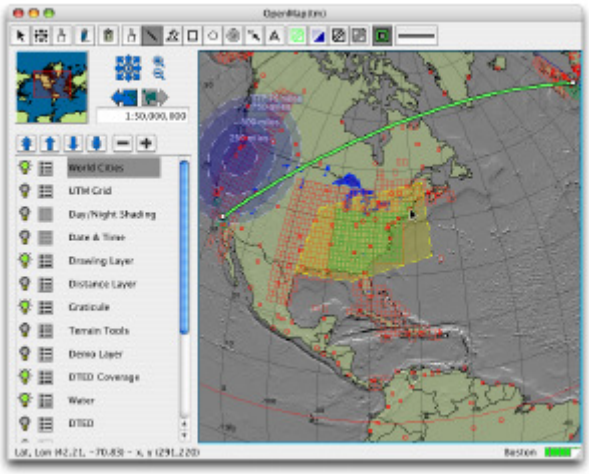


Ilustración 90: Aplicación con la librería OpenMap

7.1.2 Representación de los elementos de la ruta en el mapa

En este apartado se van a explicar cuáles son las distintas representaciones de los elementos de la ruta en el mapa.

- ❖ Waypoint: habrán tres tipos de waypoint: el waypoint inicial, el final y los waypoints intermedios en la ruta.
  - o Waypoint inicial: será un punto de color verde sobre el mapa.

Ilustración 91: Waypoint inicial

  - o Waypoint final: será un punto de color negro sobre el mapa.

Ilustración 92: Waypoint final

  - o Waypoints intermedios: serán los puntos de color rojo.

Ilustración 93: Waypoint intermedio

- ❖ Área: Serán polígonos de color azul transparente. Sus vértices son puntos de color azul claro.

Ilustración 94: Área

- Zonas prohibidas: Serán polígonos de color rojo transparente. Sus vértices, al igual que lo que ocurría con el área, serán de color azul claro.

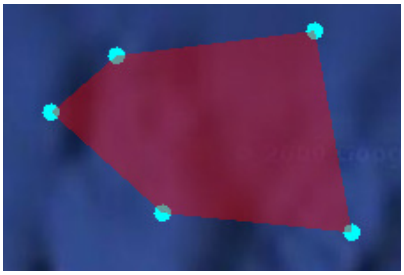


Ilustración 95: Zona prohibida

- Transecto: Es una línea que une dos waypoints. Hay dos tipos:
  - la que contiene una única línea recta

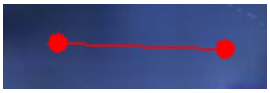


Ilustración 96: Transecto

  - la que representa una trayectoria provisional para la evitación de obstáculos.

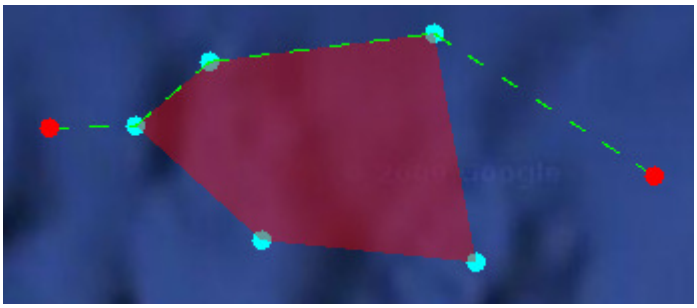


Ilustración 97: Evitación de obstáculos

Estos elementos cambiarán de color cuando estén seleccionados, a un color azul oscuro.

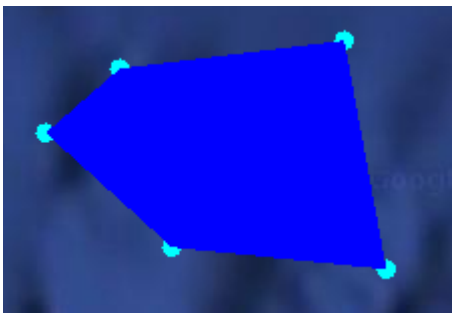


Ilustración 98: Objeto seleccionado

7.1.3 Generando una ruta

Una vez que el usuario haya definido en parte una ruta, tenemos que asegurarnos que ésta cumple con los requisitos de la ruta.

Para generar una ruta, se ha necesitado una lista de algoritmos, analizados previamente en este documento, los cuales se han implementado en distintas clases de la aplicación. Entre ellas, las más importantes son:

- Clase Adaptador: aunque se llama adaptador, porque en un principio sólo iba a ser utilizada para esto, al final se ha terminado implementando en esta clase una serie de algoritmos que nos ayudarán a resolver la generación de la ruta.

- Clase AlgoritmoGraham: como su propio nombre indica, en esta clase se ha implementado el algoritmo de Graham, que se explicará con detalle a continuación.

7.1.3.1 Clase AlgoritmoGraham

En esta clase se van a implementar las funciones necesarias para implementar el Algoritmo de Graham. Este se utiliza para, a partir de una lista de vértices dados, generar un polígono cerrado que cumpla dos condiciones:

- Debe ser un polígono convexo.

- La superficie debe ser la máxima que se puede obtener con esos vértices.

En esta clase se ha intentado hacer una implementación lo más genérica posible, utilizando clases estándar de Java, que pueden existir en cualquier proyecto que involucre a Java como lenguaje de programación.

En esta clase hay implementada una función denominada ejecutar, que será la que se invoque desde otras clases, a la que se pasará una lista de puntos en variables de tipo Point2D de Java. Esta función ejecutará el algoritmo de Graham para esta lista de

vértices y devolverá la lista de vértices que compone el área que ha dado como resultado este algoritmo, ordenada de forma que para une cada pareja de vértices.

A continuación se puede ver la interfaz de la función:

```
public static Vector<Point2D.Double> ejecutar(Vector<Point2D.Double>
                                                        lista_vertices)
```

Esta función será ejecutada cada vez que se cree, amplíe o modifique una zona (área o zona prohibida) en el plan de navegación.

Este algoritmo también ha sido utilizado para comprobar si un punto se encuentra dentro del polígono. Para ello, se simula que se va a modificar el área, añadiendo el punto como vértice. Se llama a esta función y si el área se modifica, significa que el punto se encuentra fuera del polígono. En caso contrario, si la zona se queda tal y como está, significará que el punto está contenido en la zona.

En la Ilustración 99 se muestra un ejemplo.

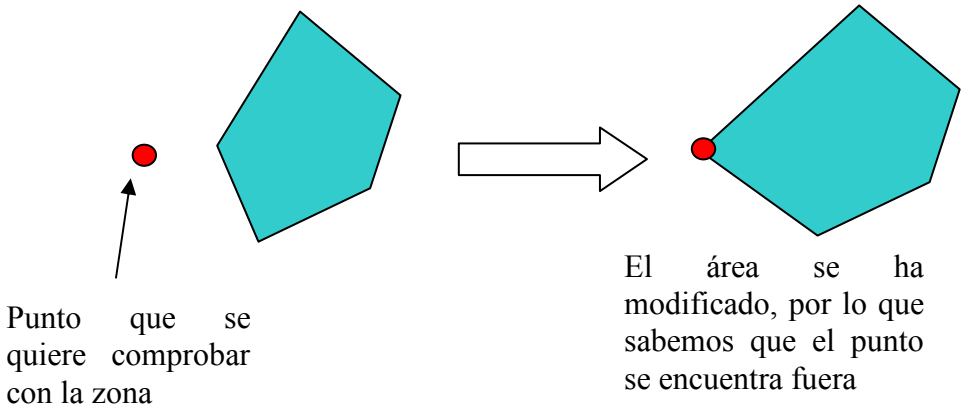


Ilustración 99: Ejemplo de Algoritmo de Graham(1)

En este caso el punto se encuentra fuera de la zona, por lo que la zona inicial y la zona final serían diferentes.

Sin embargo, en caso de que el punto se encuentre dentro de la zona no se modifica el área, como se puede ver en la Ilustración 100.

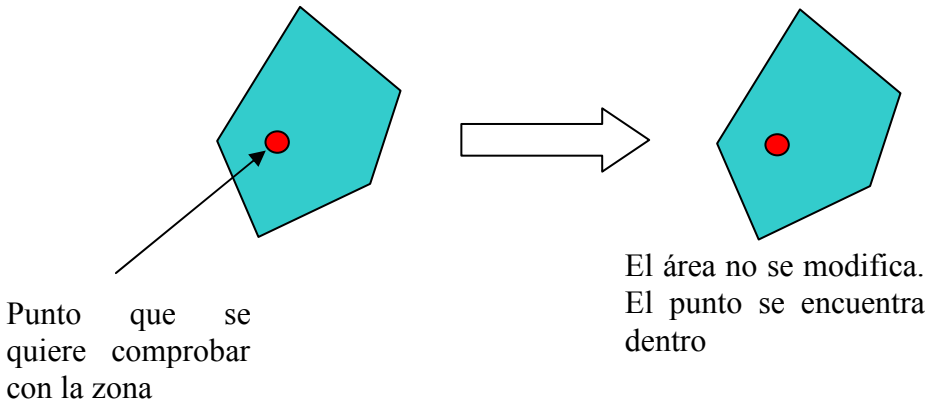


Ilustración 100: ejemplo de Algoritmo de Graham(2)

El polígono inicial y el final serían iguales, por lo que podríamos decir que el punto se encuentra dentro del polígono.

7.1.3.2 Clase Adaptador

Esta clase contendrá una serie de algoritmos que permitirán el cálculo de intersecciones de segmentos, la evitación de obstáculos, etc.

Esta clase funcionara como patrón Adaptador. Las clases del modelo de datos llamarán a funciones de esta clase en lugar de llamar directamente a las funciones que implementan los algoritmos que se quieren aplicar, con el fin de que estos algoritmos implementados fueran lo más genéricos posible, para poder reutilizarlos en el futuro.

El ejemplo claro lo vemos con el algoritmo de Graham. Este algoritmo está implementado, tal y como hemos visto, en la clase AlgoritmoGraham con tipos estándares de Java. El adaptador se encargará de convertir las Zonas (áreas y Zonas prohibidas) en una lista de puntos de tipo Point2D, para después recoger el resultado de la función y volverlos a convertir a vértices.Con esto conseguiremos reutilizar el código para ejecutar el algoritmo de Graham en futuras implementaciones creando una clase Adaptadora para adaptar su modelo de datos a los parámetros de entrada de la función.

Además de esta adaptación, en esta clase se implementan los siguientes algoritmos:

- Intersección: a esta función se le pasan dos variables de tipo Line2D que contendrán un segmento cada una. El resultado de la ejecución de la función será una variable de tipo Point2D, que indicará con un valor que los segmentos se cruzan en ese punto, y con un null que los segmentos no se tocan.

- obstaculosEnLinea: A esta función se le pasa un punto inicial y un punto final, además de una lista de obstáculos (zonas prohibidas). Se encargará de comprobar qué obstáculos de la lista de obstáculos se encuentran entre

el punto inicial y el final. Una vez terminada su ejecución, devolverá un vector con la lisa de obstáculos que se interponen entre los dos puntos.

- algoritmoBUG1: Esta función implementa la variante al algoritmo BUG (ver sección 5.9.3.4, algoritmo BUG1) para evitación de obstáculos. Se le pasaran por parámetros el punto de partida, el punto final, la lista de obstáculos y un vector a rellenar por la función en el que devolverá la lista de transectos para evitar la lista de obstáculos.

7.1.3.3 Evitación de obstáculos

La evitación de obstáculos, con el algoritmo BUG1, se ha implementado de forma que, dinámicamente, el usuario pueda ver cómo se traza el camino que va a utilizar el AUV para evitar el obstáculo.

Cada vez que se genera un nuevo transecto, se buscan todos los obstáculos que puedan intersectar con el segmento establecido entre el waypoint inicial y el final del transecto. En el transecto se almacenará la trayectoria a seguir entre los dos waypoints de forma que vendrá determinada por los vértices de los obstáculos que se encuentren en medio. Esto es así porque estamos empleando el algoritmo BUG1, por lo que lo que almacenamos es la forma en al que el vehículo bordeará los obstáculos.

En la Ilustración 101 se puede ver un ejemplo sencillo de la evitación de obstáculos.

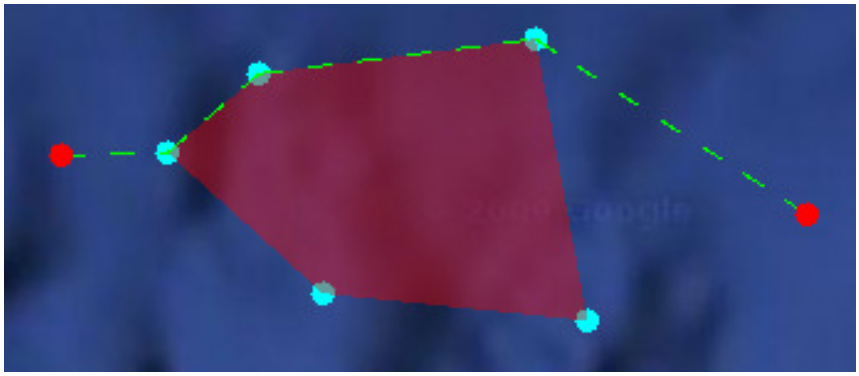


Ilustración 101: Evitación de obstáculos (1)

El transecto se almacenará de forma provisional, con el waypoint inicial del transecto, el waypoint final, y la lista de vértices de la zona prohibida por los que pasa la trayectoria.

Al añadir o quitar vértices a la zona prohibida, se recalcula la trayectoria para la evitación de obstáculos automáticamente. Lo podemos ver en la Ilustración 102.

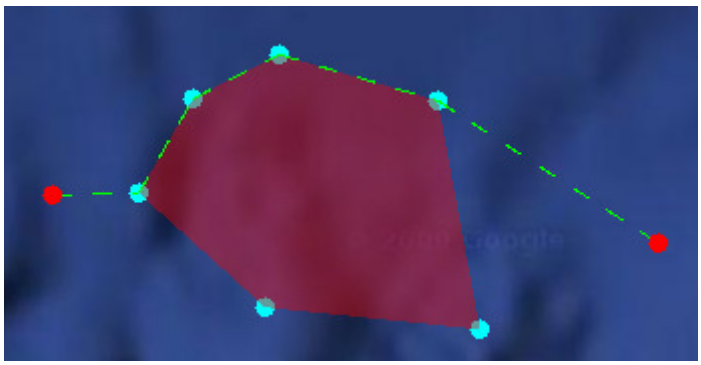


Ilustración 102: Evitación de obstáculos (2)

También se recalcula cuando se añade o modifica cualquier zona prohibida a la ruta, como se puede ver en la Ilustración 103.

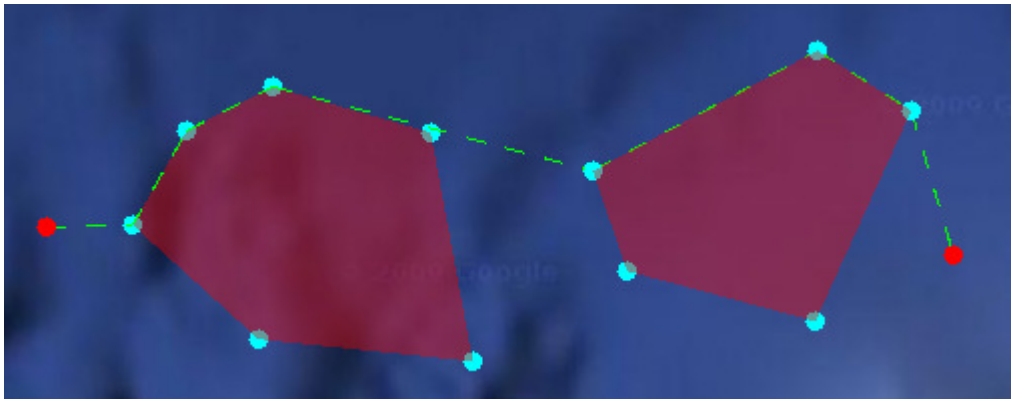


Ilustración 103: Evitación de obstáculos (3)

7.1.3.4 Generación de la ruta

Para la generación de la ruta, lo ideal sería tener una serie de waypoints sueltos en el mapa, una serie de minirrutas que indicaría la obligatoriedad de recorrer unos waypoints en un orden determinado, áreas y zonas prohibidas. Con todos estos elementos en el mapa, un planificador podría determinar cuál sería el camino más adecuado para pasar por todos los waypoints y áreas de la ruta de la forma más rápida y recorriendo la distancia mínima posible.

En la implementación del prototipo se pide al usuario que especifique el orden en el que desea que sean recorridos todos los elementos de la ruta. El orden estará implícito en el árbol(clase JTree de Swing) que se muestra a la izquierda del mapa. El usuario puede adelantar o atrasar la posición de un elemento en la lista.

Para generar la ruta, se sigue el siguiente algoritmo:

1. El primer elemento de la ruta debe ser un waypoint o una minirruta. En caso de ser un waypoint, se crea una minirruta en la que se inserta el waypoint. Con esta minirruta vamos a trabajar a partir de ahora hasta que se encuentre un área o el fin de la ruta.

2. Mientras haya elementos en la ruta, recorrer la lista de elementos en orden. A partir de aquí hay varias posibilidades según de qué tipo sea el siguiente elemento de la ruta:

    a. Si es un Waypoint: Se añadirá el waypoint a la minirruta que se está construyendo.

    b. Si es otra minirruta: Se fusionarán la minirruta anterior y la actual.

    c. Si es un área: Este es el caso más complejo. Se buscará el vértice más cercano al último waypoint. Se creará un waypoint en esta posición y se añadirá a la minirruta. En este punto hay que comprobar qué tipo de elemento es el siguiente en la ruta.

        i. Si es un área: en ese caso, habrá que buscar la pareja de vértices entre las dos áreas más cercanos, puesto que se van a unir por esos vértices. Se selecciona el vértice más cercano a la siguiente área.

        ii. Si es un waypoint: Se selecciona el vértice del área más cercano a ese waypoint.

        iii. Si es una minirruta: Se selecciona el vértice del área más cercano al waypoint inicial de la minirruta.

        iv. Una vez conocemos el vértice más cercano al siguiente elemento de la ruta, generamos un waypoint en ese punto, y una nueva minirruta en la que incluimos el waypoint. A partir de este punto, trabajaremos con esta minirruta.

3. Fin del bucle. Se termina la ejecución cuando se alcanza el último elemento de la ruta.

4. Se recorren todos los elementos de la ruta generada. Por cada transecto de la Minirruta hay que tener en cuenta la evitación de obstáculos. Cada transecto tiene una lista de vértices, de los obstáculos evitados, por los que debe pasar el AUV, ya que cada vez que se genera un nuevo transecto se calcula para éste la evitación de obstáculos. Por ello, para cada transecto hay que generar los waypoints y nuevos transectos necesarios.

Una vez finalizado este algoritmo, tendremos la ruta completa generada.

7.1.3.5 Ejemplos de la generación de la ruta

El usuario puede dibujar una serie de waypoints, áreas y zonas prohibidas sobre el mapa, tal y como se muestra en la Ilustración 104.

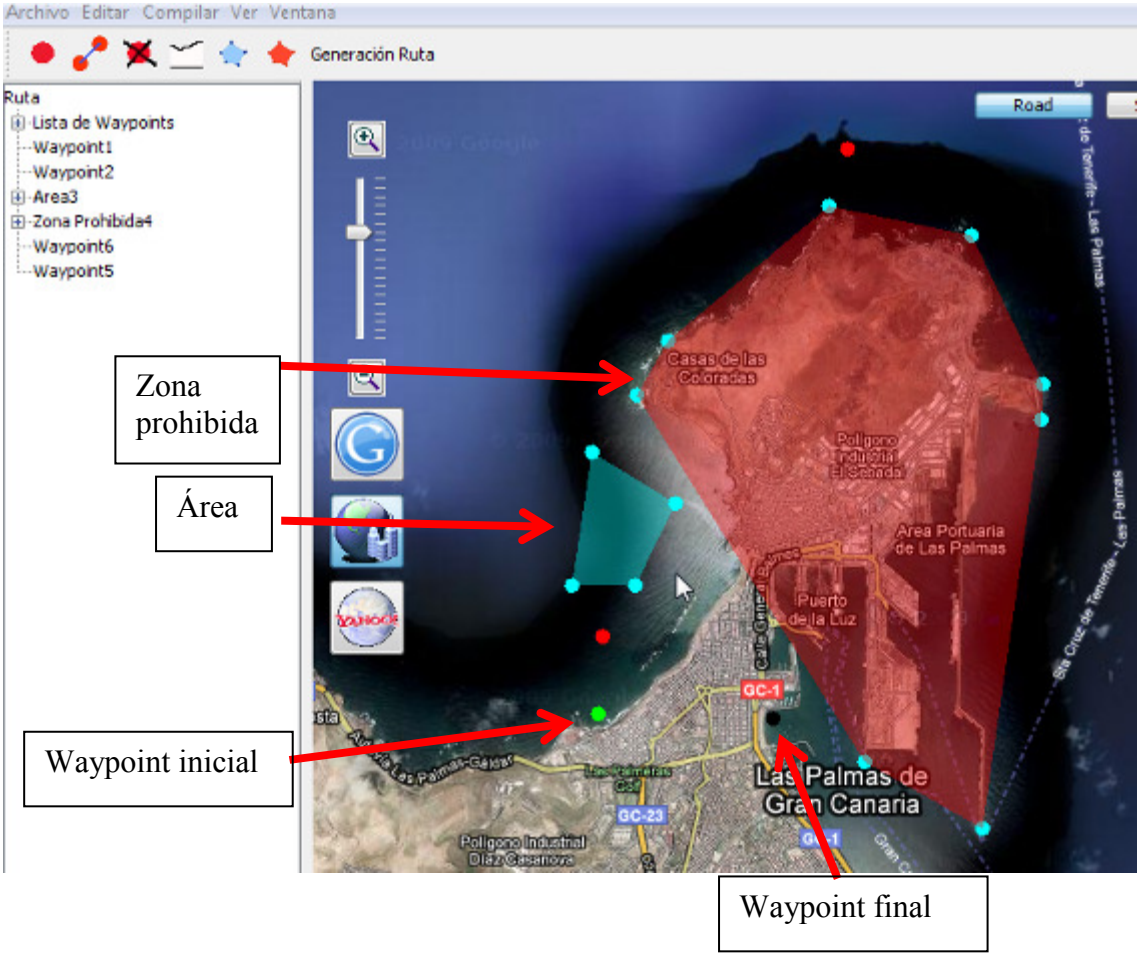


Ilustración 104: La Isleta, Gran Canaria

Se puede ver un waypoint inicial (en verde) en la playa de Las Canteras, un área a explorar (en azul), una zona prohibida a evitar (sobre La Isleta) varios waypoints intermedios para determinar la ruta aproximada que seguirá el vehículo y un waypoint final (en negro) en la playa de Las Alcaravaneras.

Al darle al botón "Generar Ruta", se genera la ruta a seguir, mostrada en la Ilustración 105:

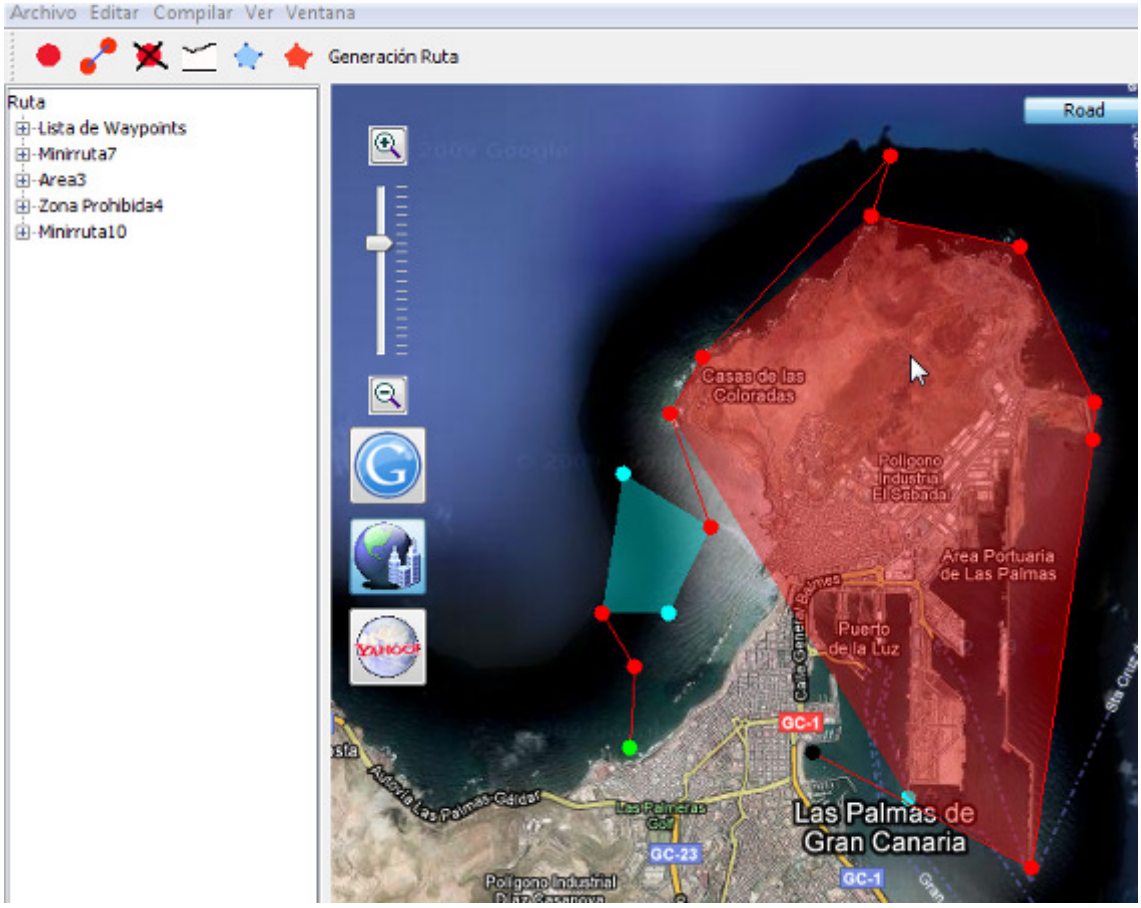


Ilustración 105: Ruta generada (1)

Como se puede ver se han generado una serie de minirrutas, que contienen a los waypoints que se encontraban sin asignar en el mapa.

El árbol de la ruta que se encuentra a la izquierda del mapa, determinará el orden a seguir antes de darle a "Generar Ruta". Se puede ver como en la Ilustración 105, los waypoints se encontraban en el mismo nivel que el área y la zona prohibida, mientras que ahora hay que acceder a la minirruta para ver qué waypoints la contienen (o abrir el nivel de "Lista de Waypoints" para poder ver todos los waypoints que están contenidos en la ruta).

Se han generado además, los waypoints necesarios para unir, por ejemplo, un waypoint a un área. Se puede ver en la Ilustración 106.

Ilustración 106: Ruta generada (2)

También se puede ver el algoritmo de evitación de obstáculos implementado. Como se ha comentado en el capítulo de análisis de algoritmos, el algoritmo implementado es el BUG1 ligeramente modificado.

El área se ha enlazado con el waypoint que iba justo antes en la lista de elementos de la ruta, por el vértice más cercano a éste. También se ha enlazado con el waypoint que le sigue en la lista de componentes de la ruta mediante el vértice que quedaba ubicado en la posición más cercana a éste. Aunque en este caso quedaba una zona prohibida de por medio, por lo que se ha empleado el algoritmo de evitación de obstáculos para evitarla.

En el siguiente ejemplo de la Ilustración 107 se puede ver un ejemplo en el que una minirruta une dos waypoints que están separados por varias zonas prohibidas.

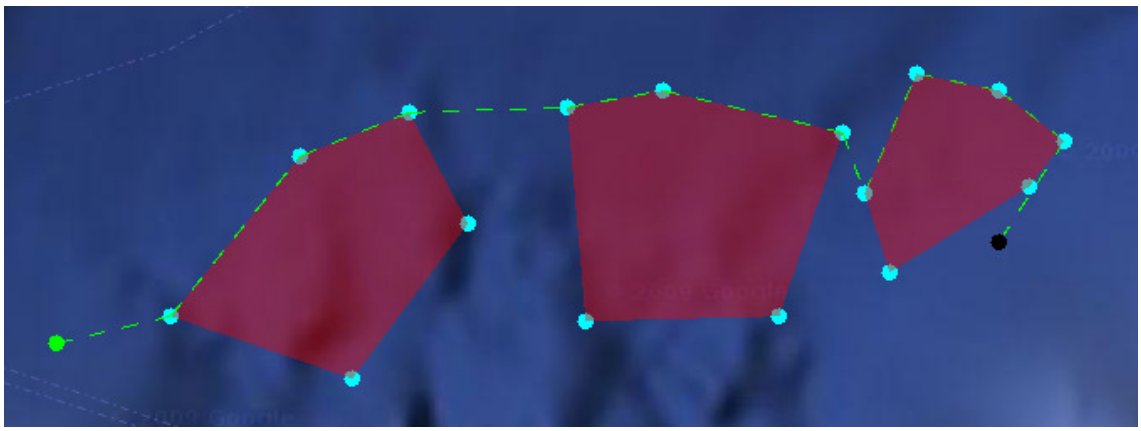


Ilustración 107: Evitación de obstáculos en la ruta

Al generar la ruta se crearán los siguientes waypoints y transectos, tal y como se puede ver en la Ilustración 108.

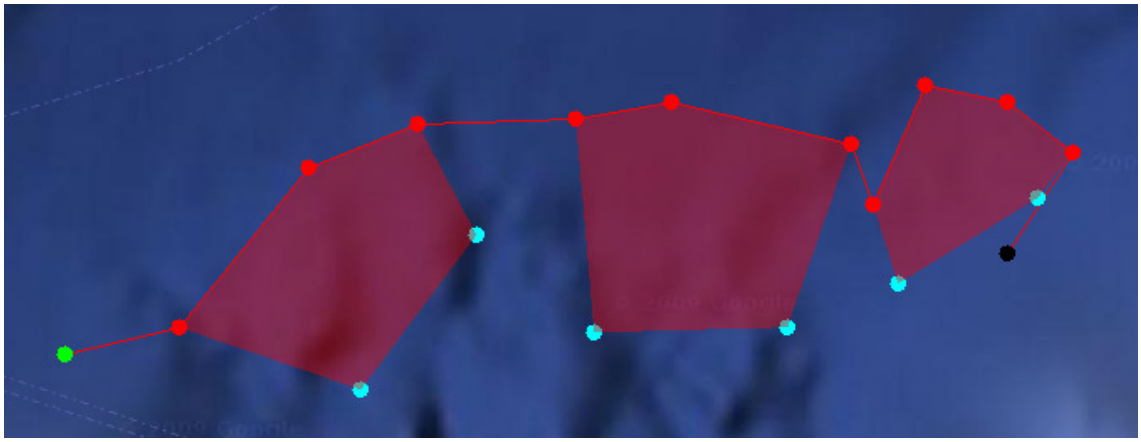


Ilustración 108: Evitación de obstáculos en la ruta generada

7.2 Plan de mediciones, de almacenamiento, de comunicaciones y de supervisión.

Como ya se ha comentado varias veces, la implementación de estos planes es bastante similar entre ellos. Las vistas implementadas tendrán una estructura similar, por ello se explicarán todas ellas en esta sección.

La ventana principal sigue la estructura mostrada en la Ilustración 109.

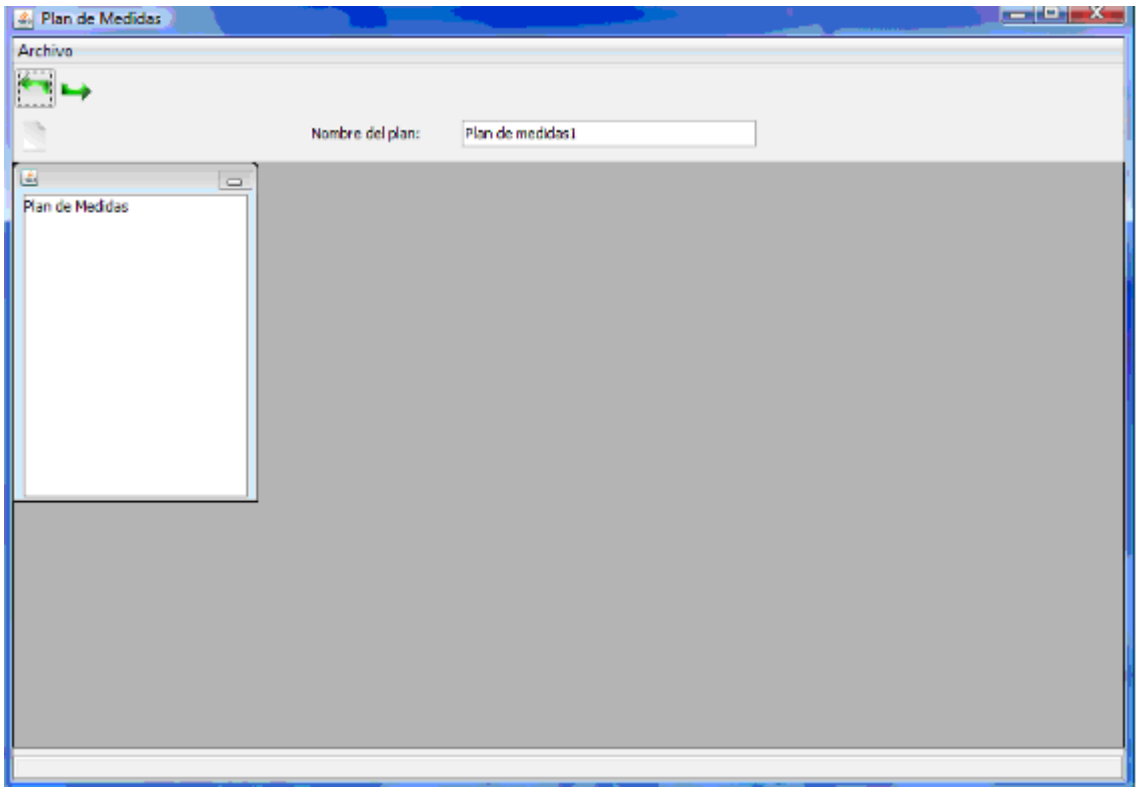


Ilustración 109: Estructura de la ventana de un plan

Vemos otra vez la interfaz de tipo MDI. Tenemos a la izquierda un árbol con la lista de tareas que contendrá el plan con el que estemos trabajando, en este caso el plan de medidas.

Al insertar una nueva tarea se muestra la ventana que se puede ver en la Ilustración 110.

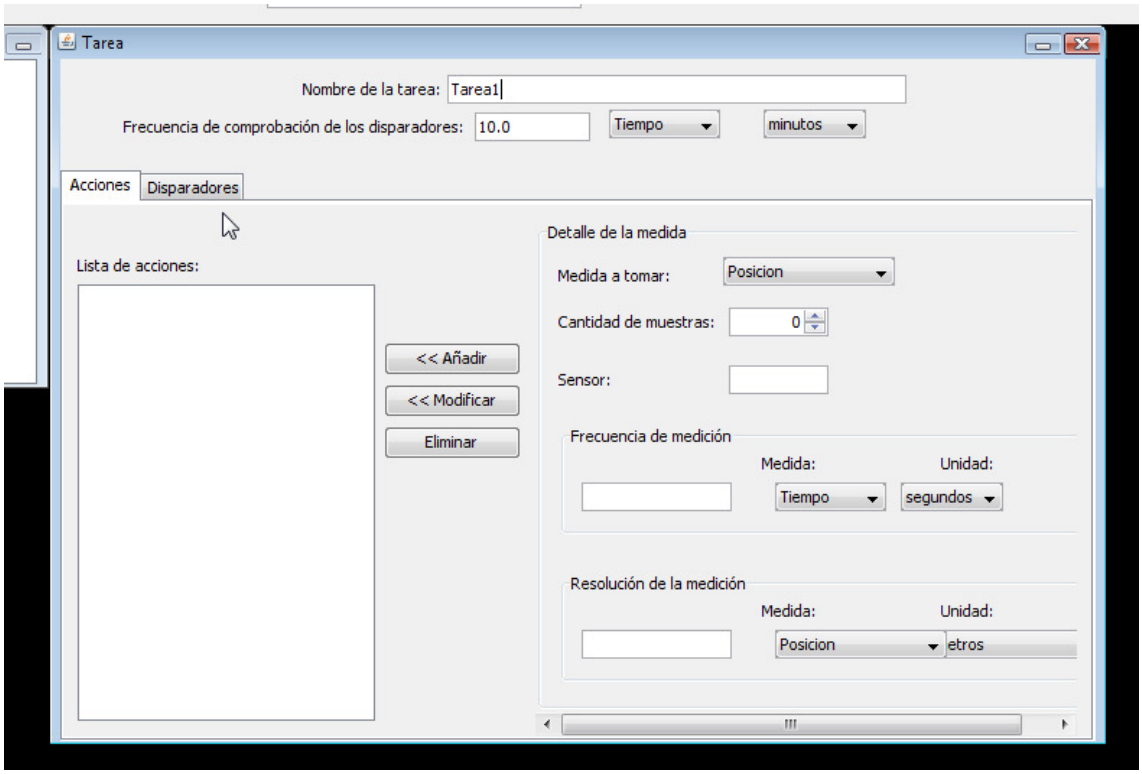


Ilustración 110: Ventana de Tarea

La ventana Tarea, tendrá, para todos los planes, la vista Acciones y la vista Disparadores. La vista Acciones varía según el plan que estemos definiendo, pero la vista Disparadores será la misma para todos los planes.

Lo que sí es constante en la vista de acciones es la lista que se muestra a la izquierda. A esta lista podremos añadir, eliminar o modificar elementos que estén contenidos en ella.

La vista Disparadores está compuesta por dos pestañas, una para las condiciones y otra para los intervalos. Se puede ver en la Ilustración 111.

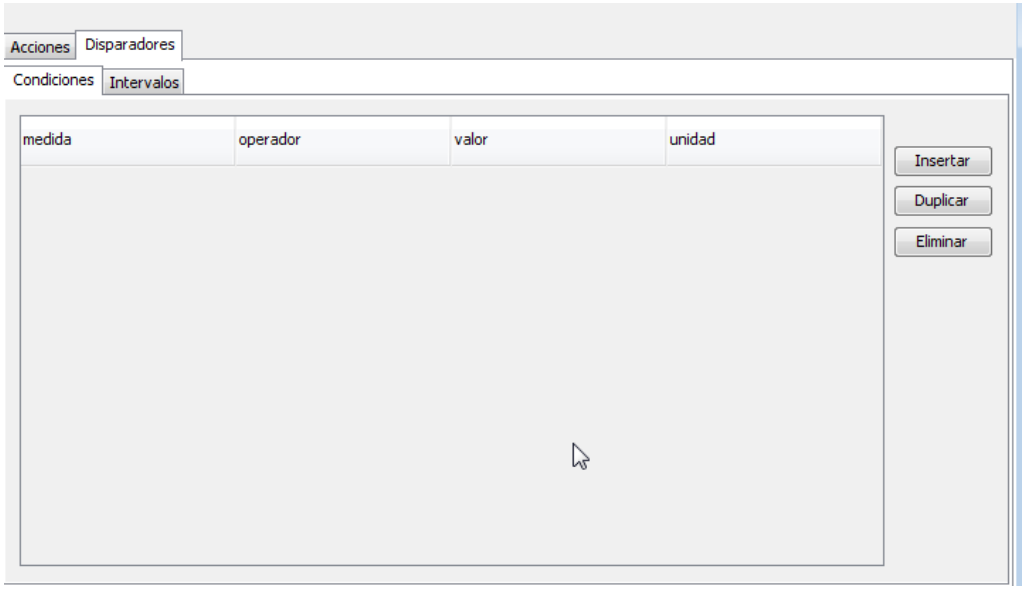


Ilustración 111: Vista Disparadores

La lista de condiciones y la de intervalos se muestran con una tabla editable. La tabla se maneja con los botones situados a la derecha, los botones "Insertar", "Duplicar" o "Eliminar".

En las tablas habrá celdas editables que permitan la introducción de números y otras que contengan listas desplegables, para seleccionar un valor. Por ejemplo, las celdas de medida y unidad. Además, estas listas estarán asociadas entre sí, produciendo que cuando se seleccione una medida, la lista de unidades de esa fila contenga sólo las unidades que se utilizan para medir esa medida.

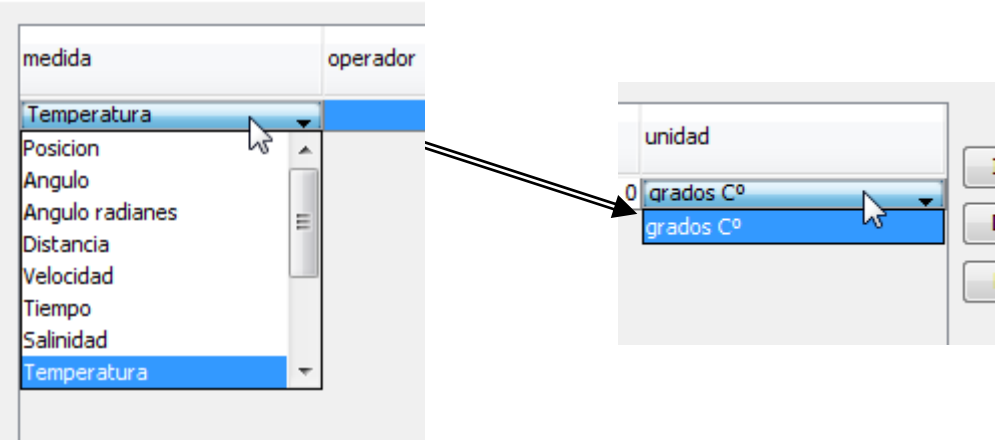


Ilustración 112: Insertando medidas y unidades (1)

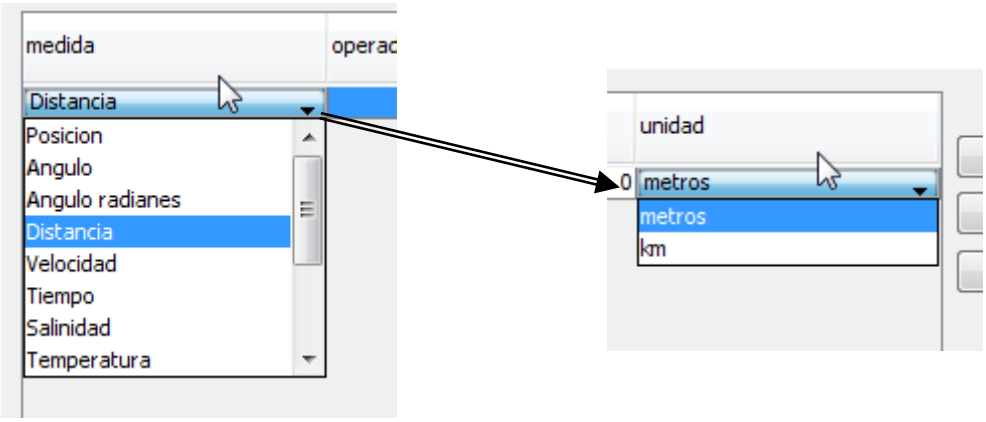


Ilustración 113: Insertando medidas y unidades (2)

La tabla quedaría de la forma que se muestra en la Ilustración 114.

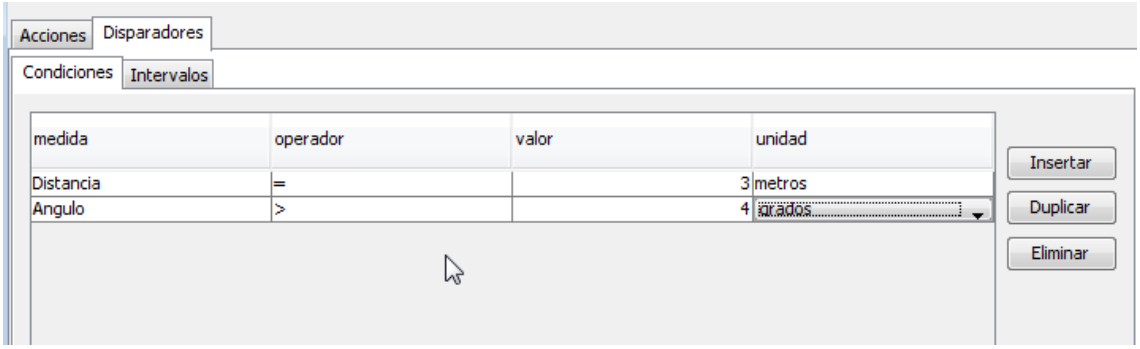


Ilustración 114: Tabla de disparadores

En éstos planes funcionarán los botones Deshacer y Rehacer. Son los que se pueden ver en la Ilustración 115.

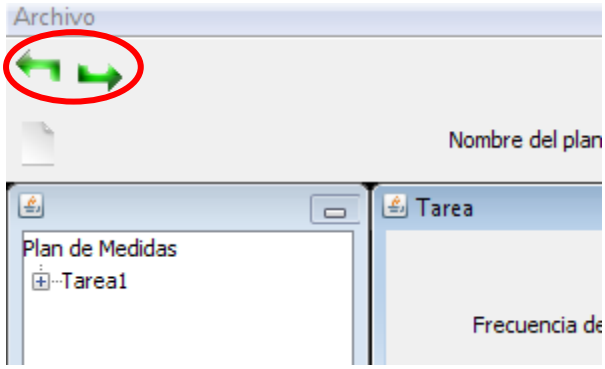


Ilustración 115: Deshacer/Rehacer

Si tenemos varias tareas en el árbol de la izquierda, podremos ir saltando de una a otra seleccionándolas en éste. En la Ilustración 116 se puede ver que se ha seleccionado la Tarea 1, mientras que en la Ilustración 117 se ve la tarea 2.

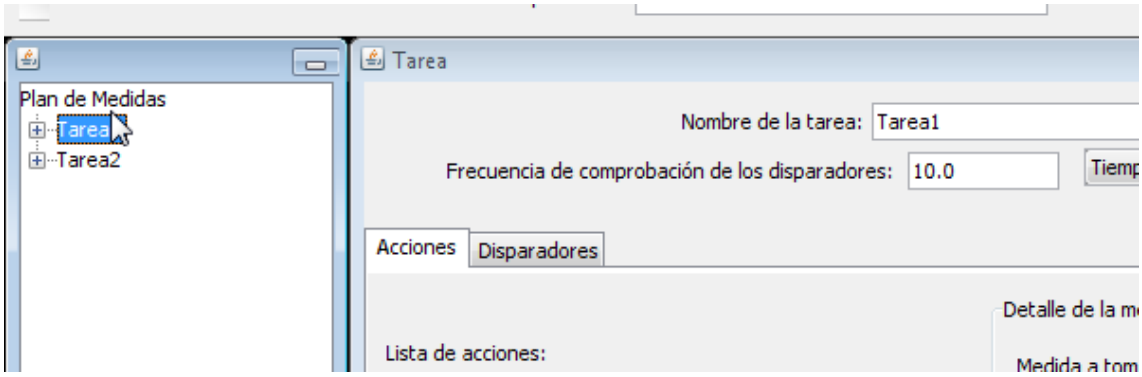


Ilustración 116: Tarea 1 seleccionada

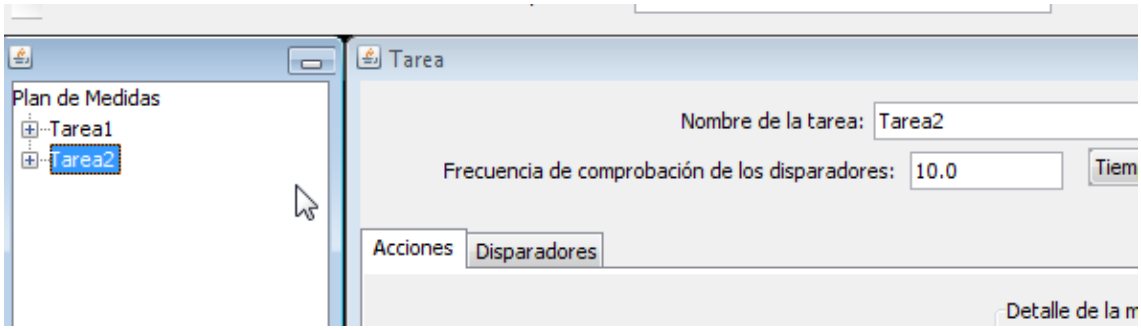


Ilustración 117: Tarea 2 seleccionada

7.3 Validar misión

Esta función es la encargada de generar un fichero XML con la misión. En este punto de la implementación, se validarán diferentes aspectos de los distintos planes y generará el fichero mision.xml, con la misión que ha seleccionado el usuario. En esta vista, con el fin de que la implementación fuera más rápida, se ha mezclado la vista con el controlador.

La vista resultante es la que se muestra en la Ilustración 118.

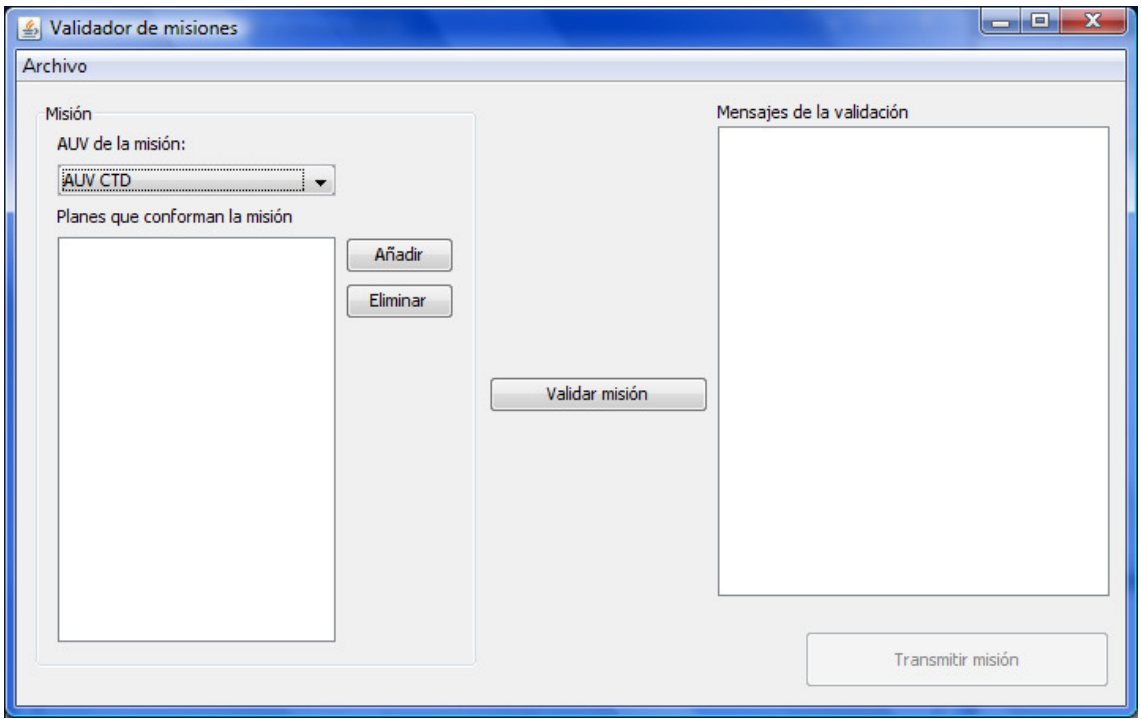


Ilustración 118: Vista Validador

Como se puede observar se dispone de una lista donde figurarán los distintos planes asociados a la misión, una lista desplegable donde se elige el AUV que va a realizar la misión y una ventana donde se mostrarán los distintos mensajes que se han producido durante la validación.

Si la validación de la misión no ha producido ningún error se activará el botón "Transmitir misión".

Las validaciones implementadas son las siguientes:

- Una misión debe tener un plan de cada tipo como mínimo. No puede faltar ninguno de los cinco planes.
- El AUV tiene la instrumentación necesaria para realizar las medidas usadas, tanto en los disparadores de todos los planes como en las acciones de éstos.
- No hay ningún punto de la ruta que el AUV no pueda alcanzar debido a su limitación den la profundidad.
-

Parte V. Resultados.

Capítulo 8. Realización del proyecto.

8.1 Grado de realización del proyecto

Las capacidades implementadas en el prototipo son:

- Planificación de la misión:
  - Plan de navegación: El usuario especifica todos los componentes que puede contener una ruta (waypoints, áreas, zonas prohibidas y minirrutas), especificándolo en un mapa de Google, ayudándonos de su implementación en la geolocalización.

    El usuario es el que decide qué orden va a seguir el submarino al recorrer los componentes de la ruta.

    También se han implementado algunas vistas de propiedades, debido a su importancia, como pueden ser las propiedades de los waypoints, donde se puede establecer la profundidad de éstos, o las propiedades del área, en las que se puede definir las distintas formas en las que va a poder ser recorrida un área.

    Se ha implementado el algoritmo de evitación de obstáculos que se ha explicado en esta memoria, la modificación del algoritmo BUG1. Para ello también ha sido necesario implementar un algoritmo para calcular las intersecciones, para poder buscar los obstáculos que se encuentran entre dos puntos. Además, se ha implementado el algoritmo de Graham para poder generar las áreas y zonas prohibidas como polígonos convexos.

  - Planes de medidas, almacenamiento, comunicación y supervisión: La implementación de estos planes ha sido bastante completa, incluyendo la implementación de la lista de condiciones y la lista de intervalos.
- Validación de la misión: Se ah implementado algunas de las validaciones de la misión. Además, se carga y almacena el fichero XML de la misión.

Se ha podido analizar y diseñar el formato de las comunicaciones entre el monitor de la misión y el submarino. Sin embargo no se ha implementado estas comunicaciones, aunque sí se ha implementado unas librerías XDR para el formato de los datos que se intercambian entre el submarino y el monitor de la misión.

8.2 Pruebas de validación

Para las pruebas de validación se va a generar un plan de cada tipo, almacenarlo, generar una misión y validarla.

8.2.1 Plan de Navegación

Se empezará por el plan de navegación. Para crear el plan de navegación, en la vista principal de la aplicación se puede pulsar sobre el botón "Navegación". Se creará un plan de navegación que contenga waypoints, minirrutas, áreas y zonas prohibidas. En la Ilustración 119 se puede ver la ruta de ejemplo que se quiere generar.

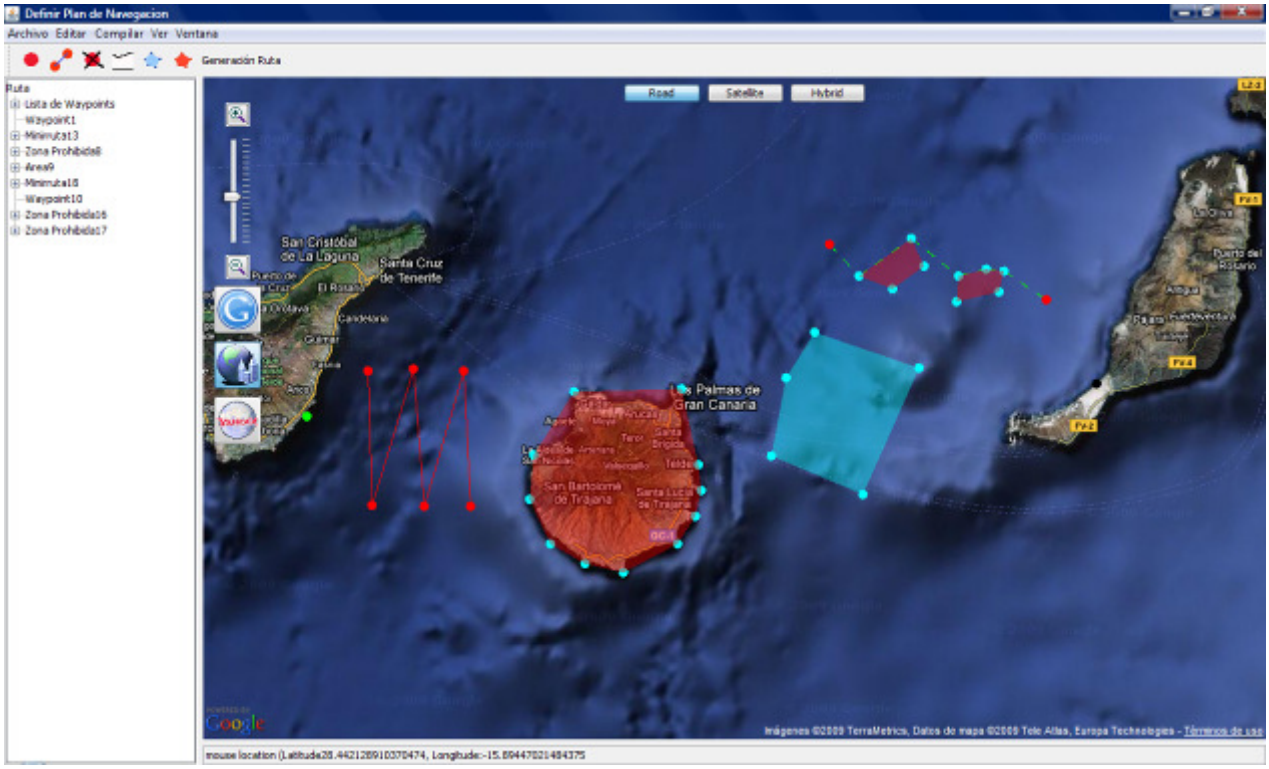


Ilustración 119: Ruta de ejemplo

En la Ilustración 120 se puede ver el mapa con figuras que indican dónde están ubicados los componentes que se mencionan. Contiene dos minirrutas, una de ellas, la m2, con un solo transecto (la que está ubicada entre Gran Canaria y Fuerteventura, al norte del área). En la minirruta m2 se puede ver la evitación de obstáculos.

El orden que sigue va desde el waypoint inicial, situado a la izquierda, en Tenerife, hasta el waypoint final, situado en Fuerteventura.

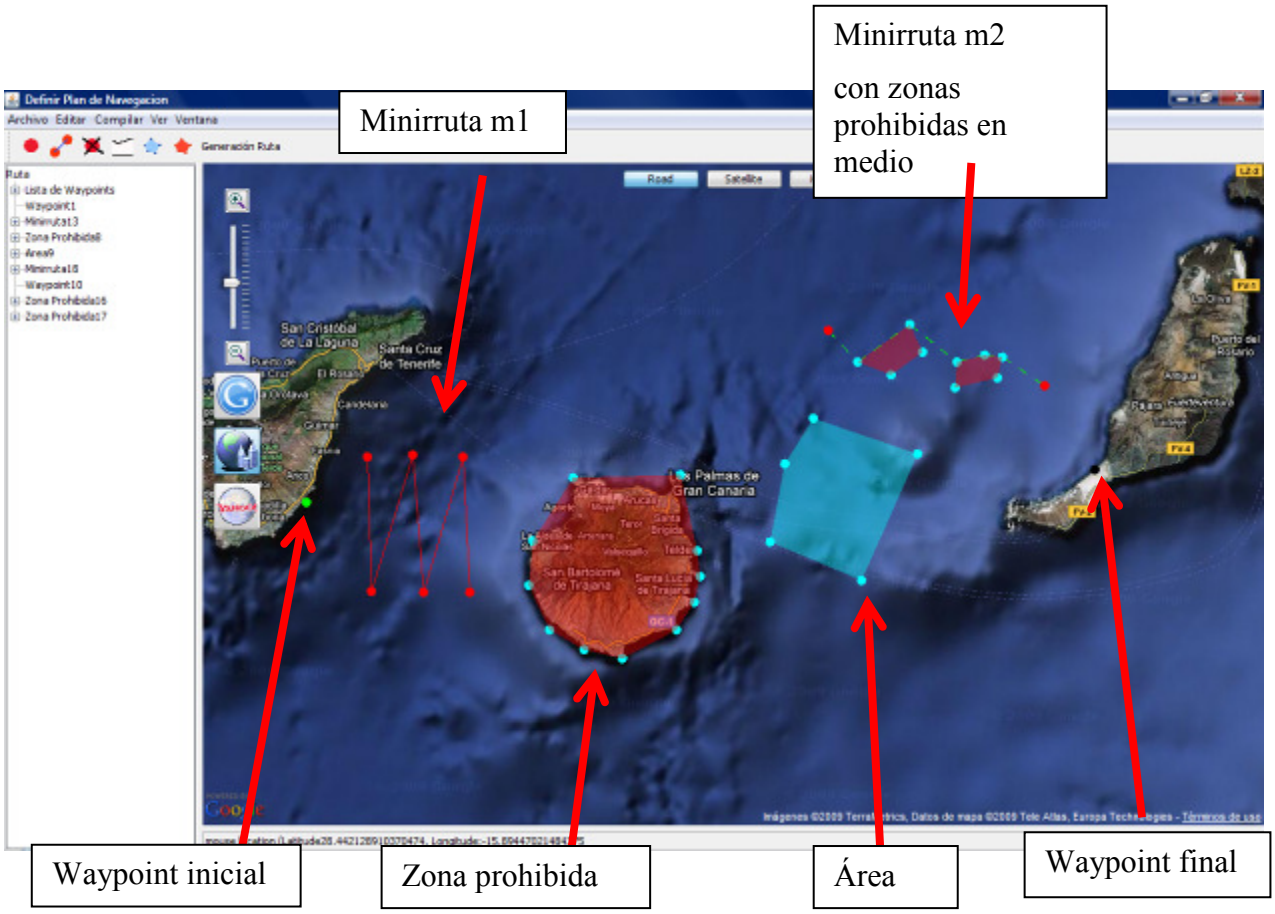


Ilustración 120: Ruta de ejemplo detallada

Para probar las validaciones, en la fase final de la validación de la misión, se establecerán, tal y como se ve en la Ilustración 121, la profundidad de uno de los waypoints a 1000 metros.

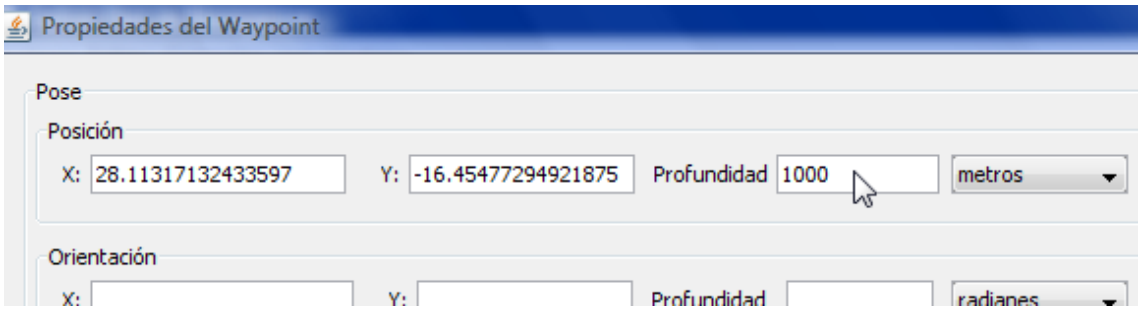


Ilustración 121: Profundidad del Waypoint

También se especificarán varias formas de recorrer el área, según unos disparadores. El submarino deberá permanecer a la deriva durante los primero cinco minutos, para a continuación pasar a un seguimiento de rangos de temperatura. Además, la profundidad máxima que puede alcanzar el submarino en el área es de 500 metros.

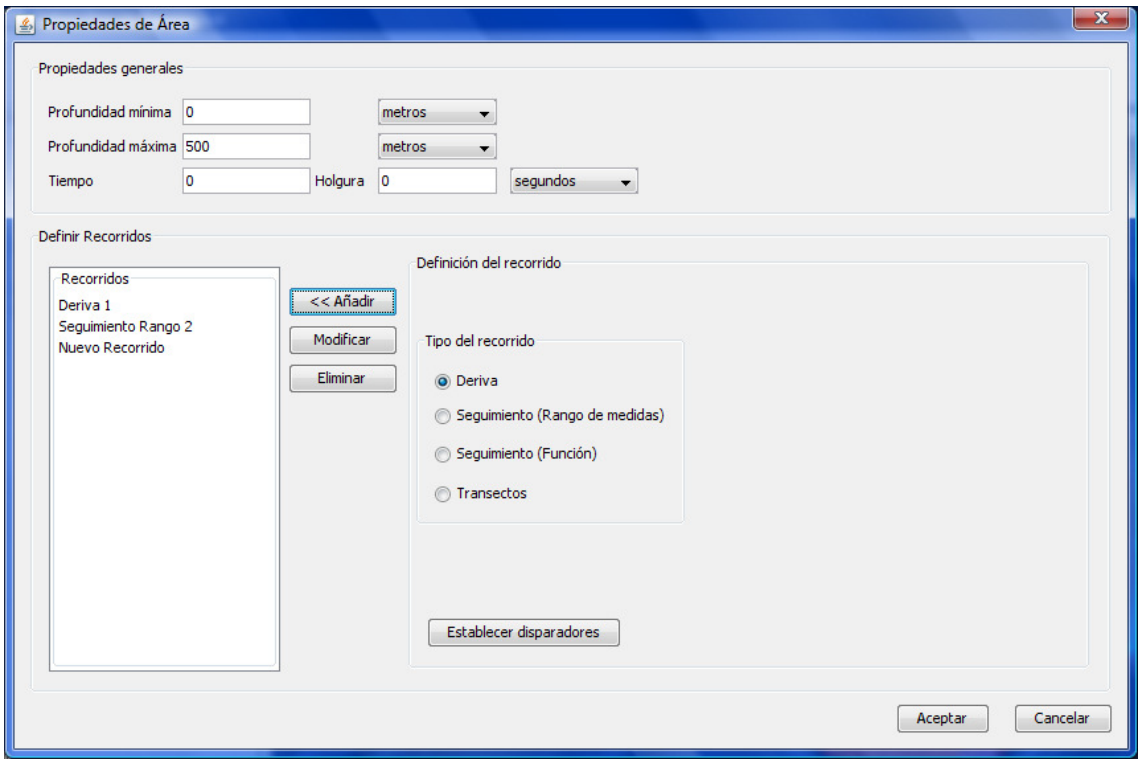


Ilustración 122: Propiedades de área

Se guarda la versión actual del plan de navegación, ya que se va a generar la ruta, y de esta forma se podrá tener una versión a la que volver en caso de que el resultado de la generación de la ruta no sea el más adecuado.

Se guarda el archivo en una ubicación cualquiera.

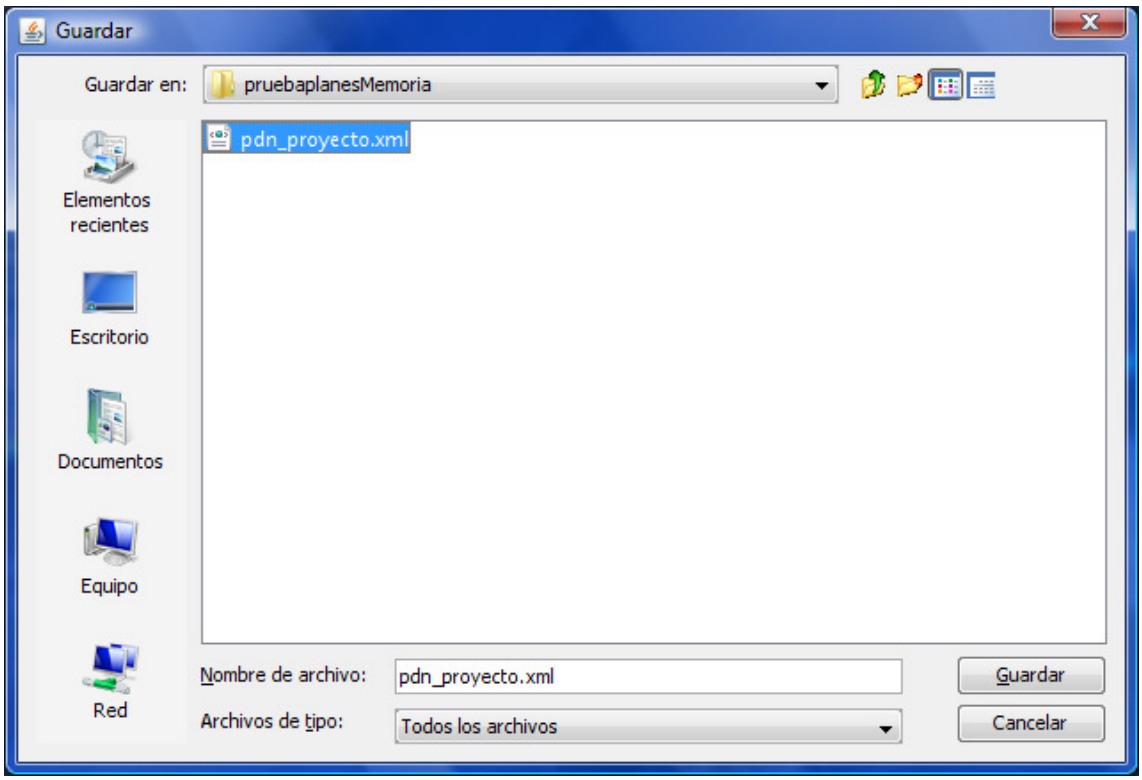


Ilustración 123: Almacenar Plan

¡¡¡IMPORTANTE!!! Ni la ruta del fichero, ni el fichero, pueden contener espacios, debido a que la versión usada del analizador sintáctico Xerces no lo permite.

Se genera la ruta, tal y como se ve en la Ilustración 124.

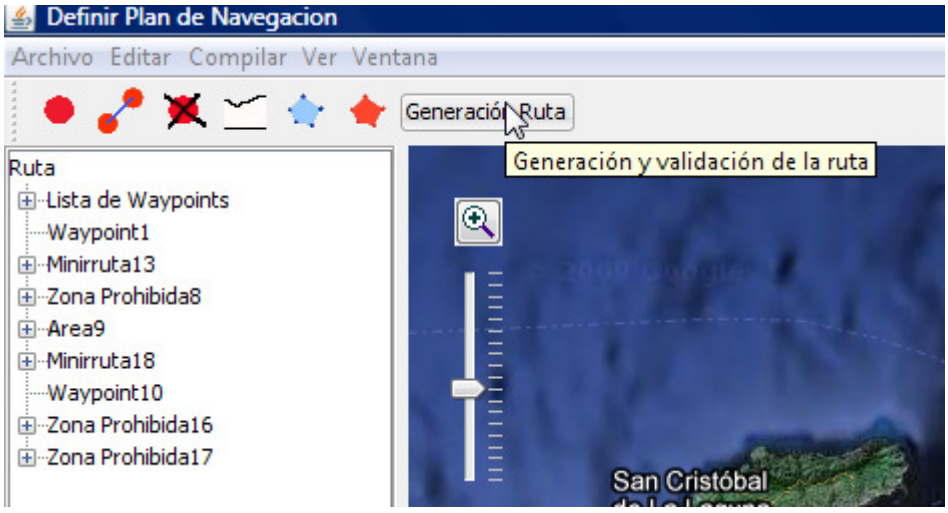


Ilustración 124: Generación de la ruta

El resultado es el que se muestra en la Ilustración 125.

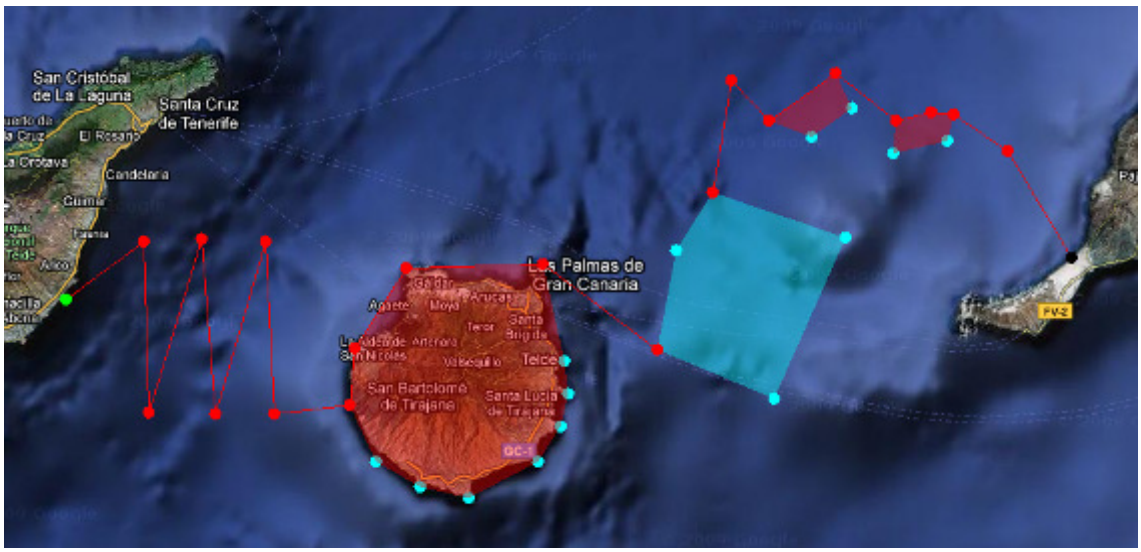


Ilustración 125: Ruta generada

Se almacena el plan de navegación para cargarlo en la misión.

Se puede comprobar lo que se ha guardado, abriendo el fichero XML y comprobando que se hayan almacenado todos los datos especificados.

El waypoint al que le hemos establecido una profundidad de 1000 metros se habrá almacenado de la siguiente forma:

```
<waypoint id="1">
    <pose>
      <posicion x="28.11317132433597" y="-16.45477294921875"
                                          z="1000.0" unidad="metros" />
    </pose>
</waypoint>
```

La unidad especificada (metros) se refiere a la profundidad, ya que la posición se leerá por defecto como latitud-longitud.

El área, con sus dos tipos de recorridos especificados, se ha almacenado de la siguiente forma:

```
<area id="9" nombre="Area 9">
    <vertice id="2" x="28.016226126405623" y="-15.1611328125" z="0.0"
unidad="Grados" />
    <vertice id="7" x="28.20760859532738" y="-15.11993408203125"
z="0.0" unidad="Grados" />
    <vertice id="6" x="28.318888915773826" y="-15.040283203125"
z="0.0" unidad="Grados" />
    <vertice id="5" x="28.231809851211853" y="-14.7491455078125"
z="0.0" unidad="Grados" />
    <vertice id="3" x="27.921620449508442" y="-14.90570068359375"
z="0.0" unidad="Grados" />

    <recorrido id="1">
      <disparadores:disparadores>
        <disparadores:condicion id="1" medida="Tiempo" operador="&lt;"
valor="5.0" unidad="minutos" />
```

```
      </disparadores:disparadores>
      <deriva />
    </recorrido>

    <recorrido id="2">
      <disparadores:disparadores>
        <disparadores:condicion          id="1"          medida="Tiempo"
operador="&gt;=" valor="5.0" unidad="minutos" />
      </disparadores:disparadores>
      <rango     medida="Temperatura"     minimo="10.0"     maximo="15.0"
unidad="grados C°" />
    </recorrido>

  </area>
```

Se pueden ver los tipos de recorridos especificados en el área y sus disparadores.

8.2.2 Plan de Medidas

Se crearán dos tareas para el plan de medidas, y se les asignarán disparadores y acciones para comprobar que se almacena en ficheros XML y se carga correctamente.

La idea es poner, en al menos una acción, la medición de salinidad, ya que uno de los dos submarinos utilizados no tiene soporte para tomar medidas de salinidad, con lo que se produciría un error durante la validación de la misión.

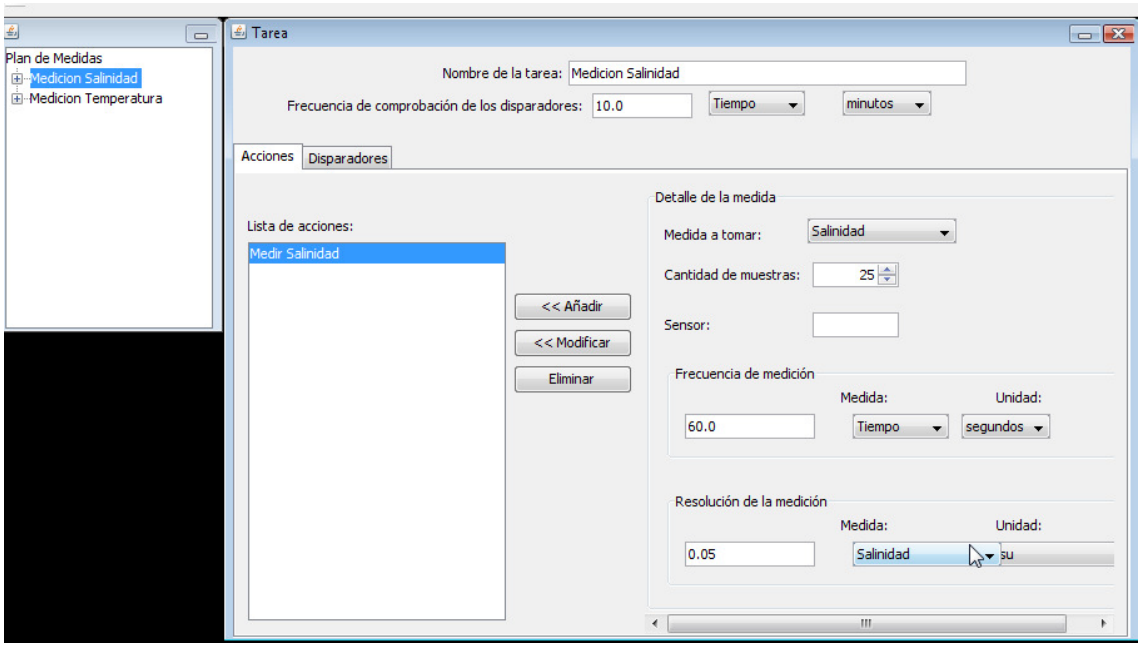


Ilustración 126: Plan de medidas

Se puede probar a guardar el plan de medición, y después cargarlo, para ver si se carga correctamente.

Si se comprueba el XML se puede ver la medida de la salinidad en la primera tarea:

```
<tarea id="1" nombre="Medicion Salinidad">
```

```
  <disparadores:disparadores>
    <disparadores:intervalo   id="1"   medida="Plan   de   Navegacion"
inicio="1.0" fin="5.0" periodo="1.0" unidad="waypoint" />
  </disparadores:disparadores>
  <acciones>
    <medir id="1" medida="Salinidad" muestras="25">
      <frecuencia valor="60.0" unidad="segundos" />
      <resolucion valor="0.05" unidad="psu" />
    </medir>
  </acciones>
  <periodoInhibicion valor="10.0" unidad="minutos" />
</tarea>
```

Los disparadores tendrán intervalos. Se tratará de medir la salinidad a partir del waypoint 1, hasta el waypoint 5, en cada waypoint.

8.2.3 Plan de Comunicaciones

Vamos a programar una comunicación del submarino cada hora. Se busca que cada hora emita los datos recogidos sobre salinidad.

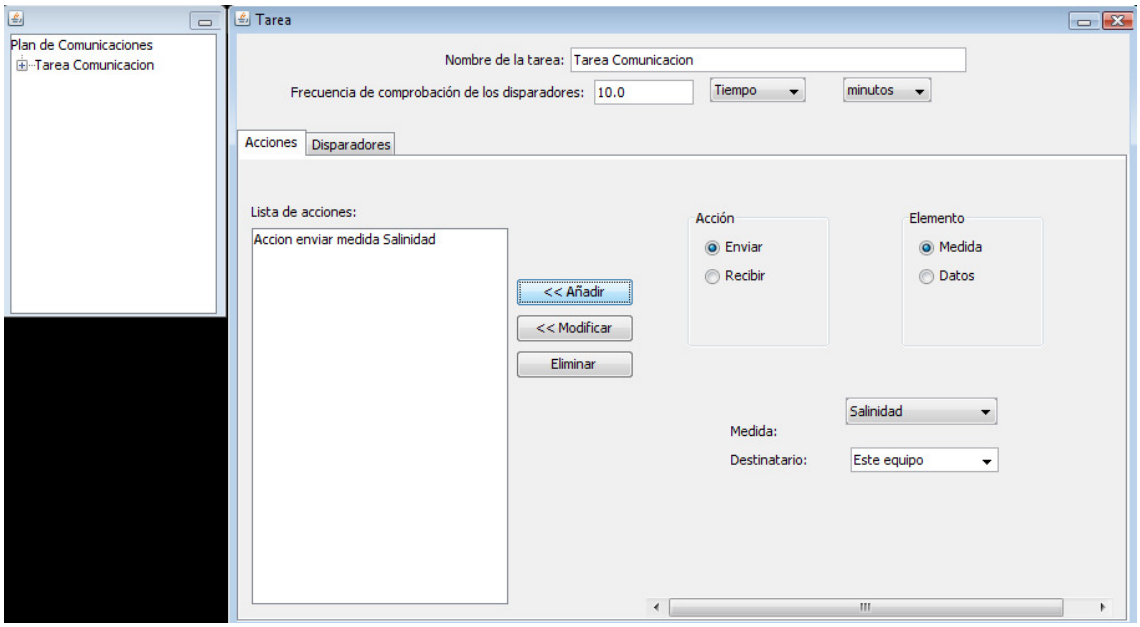


Ilustración 127: Plan de comunicaciones

Al igual que con el plan de medidas, se puede guardar para volver a cargarlo comprobando que se haya guardado correctamente.

La tarea de comunicación especificada se almacenará de la siguiente forma en el fichero XML:

```
<tarea id="1" nombre="Tarea Comunicacion">
  <disparadores:disparadores>
    <disparadores:intervalo id="1" medida="Tiempo" inicio="1.0"
                  fin="99999.0" periodo="1.0" unidad="segundos" />
  </disparadores:disparadores>
  <acciones>
    <enviarMedida id="1" medida="Salinidad" destinatario="Este equipo"
/>
```

```
  </acciones>
  <periodoInhibicion valor="10.0" unidad="minutos" />
</tarea>
```

Se puede ver que tiene como disparadores el intervalo, para comunicarse desde la primera hora de la misión hasta el final de ésta, y durante cada hora. Es decir, cada hora habrá una comunicación para emitir los datos de salinidad hasta que termine la misión.

8.2.4 Plan de Almacenamiento

En el plan de almacenamiento se especificará que se almacenen las medidas de salinidad y temperatura. Por ello se crearán también dos tareas en las que las acciones estén activas cuando se cumplan los disparadores de las tareas especificadas en el plan de medidas.

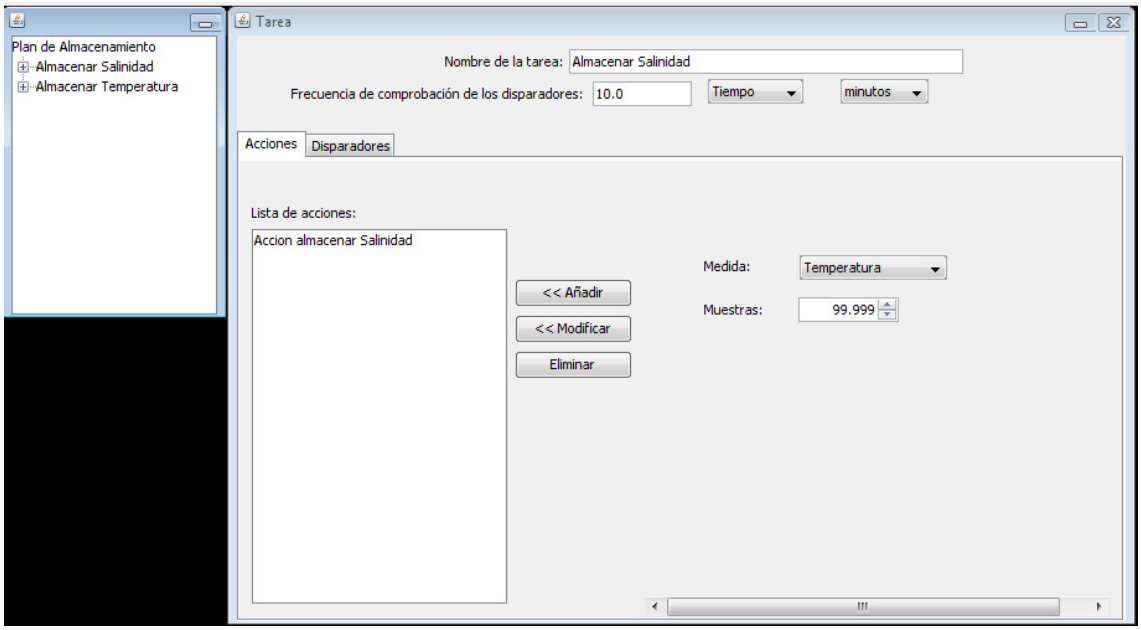


Ilustración 128: Plan de almacenamiento

Se debe comprobar que el fichero XML que se genera cuando se almacena este plan contiene las dos tareas especificadas con sus correspondientes disparadores y acciones. Los disparadores que he definido provocan que en el momento de realizar las mediciones, la tarea correspondiente esté activa para almacenar los datos medidos.

Se pueden ver los XML de las dos tareas a continuación:

```
<tarea id="1" nombre="Almacenar Salinidad">
  <disparadores:disparadores>
    <disparadores:condicion id="1" medida="Plan de Navegacion"
                    operador="&gt;=" valor="1.0" unidad="waypoint" />
    <disparadores:condicion id="2" medida="Plan de Navegacion"
                    operador="&lt;=" valor="5.0" unidad="waypoint" />
  </disparadores:disparadores>
  <acciones>
    <almacenar id="1" medida="Salinidad" muestras="99999" />
  </acciones>
```

```
  <periodoInhibicion valor="10.0" unidad="minutos" />
</tarea>


<tarea id="2" nombre="Almacenar Temperatura">
  <disparadores:disparadores>
    <disparadores:condicion id="1" medida="Plan de Navegacion"
                  operador="&gt;=" valor="6.0" unidad="waypoint" />
    <disparadores:condicion id="2" medida="Plan de Navegacion"
                 operador="&lt;=" valor="10.0" unidad="waypoint" />
  </disparadores:disparadores>
  <acciones>
    <almacenar id="1" medida="Temperatura" muestras="99999" />
  </acciones>
  <periodoInhibicion valor="10.0" unidad="minutos" />
</tarea>
```

8.2.5 Plan de Supervisión

En el plan de supervisión se establecerán comandos que van a hacer abortar una misión a un submarino en el caso de que se encuentre a demasiada profundidad.

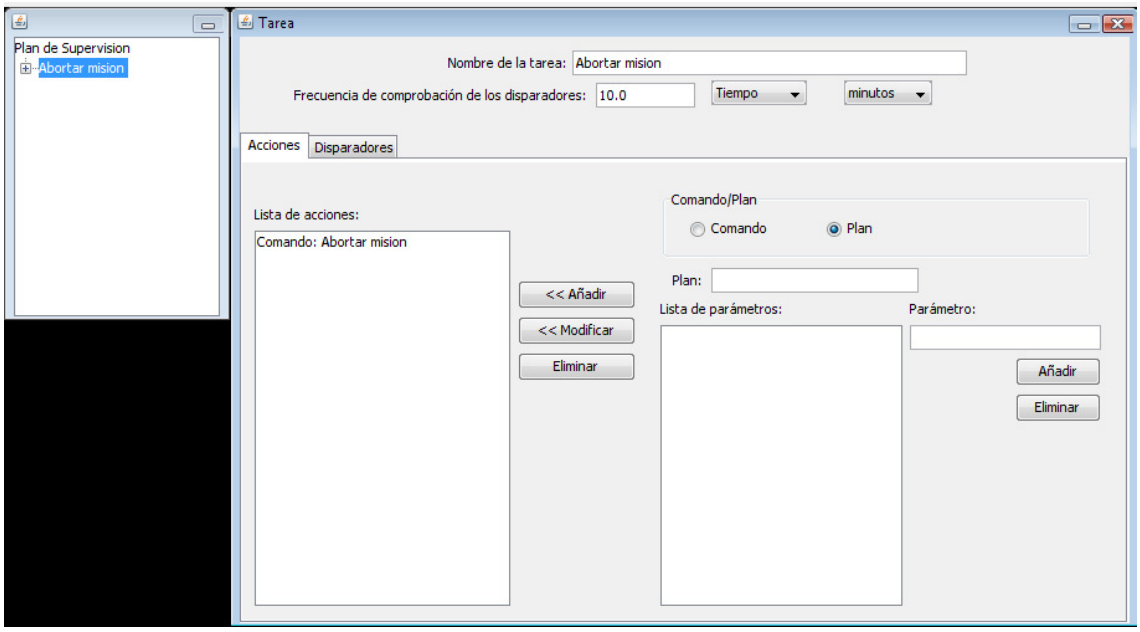


Ilustración 129: Plan de supervisión

Como siempre, se guarda el plan y lo se vuelve a cargar para comprobar si está correcto. Además, se puede ver el fichero XML para comprobar si los datos almacenados son correctos.

```
<planDeSupervision     xmlns:disparadores="http://www.disparadores.com"
id="0" nombre="pds_proyecto.xml">
  <tarea id="1" nombre="Abortar mision">
    <disparadores:disparadores>
      <disparadores:condicion id="1" medida="Profundidad"
                  operador="&gt;" valor="1000.0" unidad="metros" />
    </disparadores:disparadores>
```

```
    <acciones>
      <ejecutarComando id="1" comando="Abortar mision" />
    </acciones>
    <periodoInhibicion valor="10.0" unidad="minutos" />
  </tarea>
</planDeSupervision>
```

8.2.6 Generación de la misión y validación de la misma

Una vez generados todos los planes necesarios para una misión, se puede crear ya la misión en sí.

Lo primero que se hará será cargar los planes que creado para la misión. El resultado es lo que se muestra en la Ilustración 130.

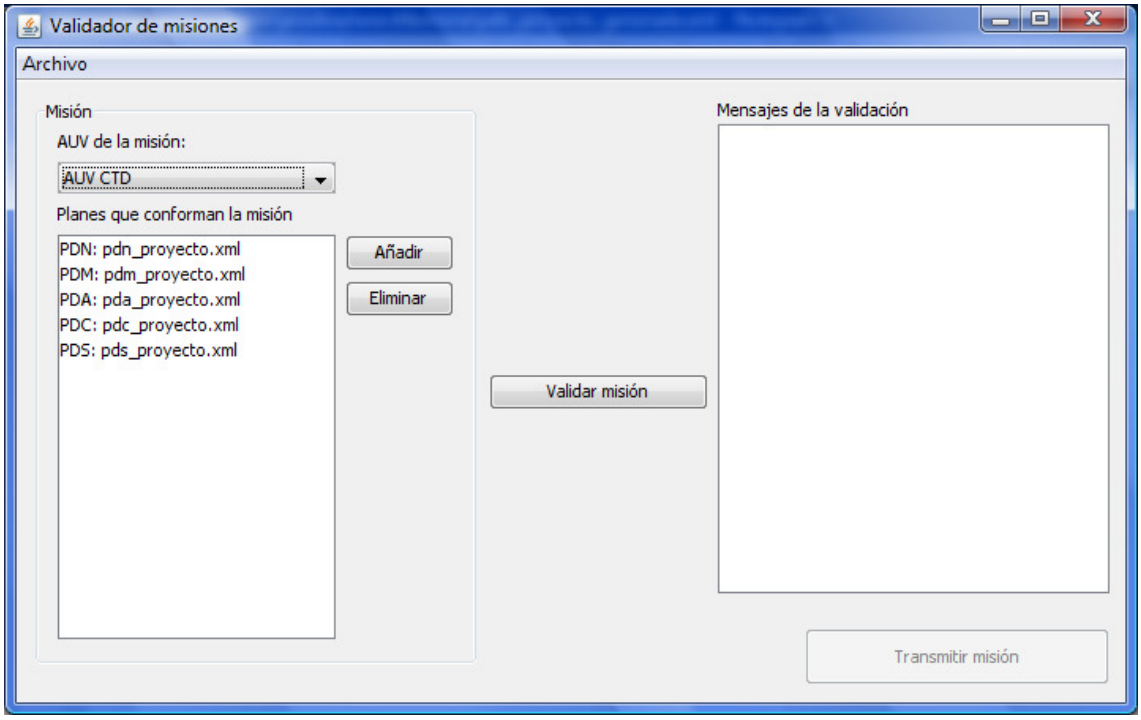


Ilustración 130: Validador de la misión

A continuación se selecciona el submarino que se va a utilizar para la misión, tal y como se muestra en la Ilustración 131.

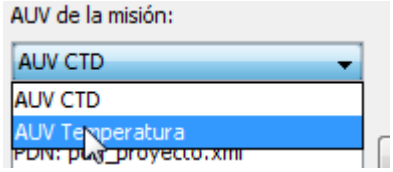


Ilustración 131: Selección de AUV

Se utiliza en la validación el "AUV Temperatura", el cual es el submarino más limitado de los dos, ya que tiene una profundidad máxima de 500 metros y no tiene sensor CTD, por lo que no puede medir salinidad.

Al darle al botón “Validar misión”, obtendremos los siguientes cuatro mensajes de error, como se puede ver en la Ilustración 132.

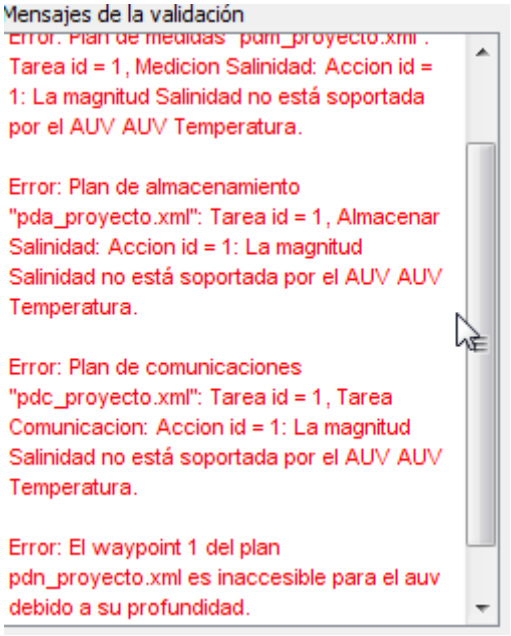


Ilustración 132: Mensajes de error en la validación

Los mensajes informan de cada uno de los proyectos en los que se ha encontrado una medida que no es soportada por el submarino seleccionado.

Además, da un error relacionado con el waypoint ubicado a una profundidad de 1000 metros, ya que el submarino solo puede llegar hasta los 500 metros.

Sin embargo, si se selecciona el submarino “AUV CTD”, tanto la profundidad como la medida de salinidad ya no será un problema. Se validará correctamente la misión y se generará el fichero de misión.

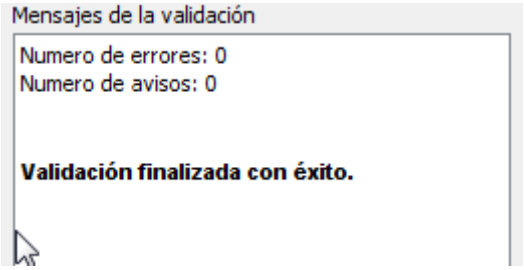


Ilustración 133: Mensajes de la validación

A continuación se puede guardar la misión, comprobar que el fichero XML es el correcto, y volverla a cargar si es necesario. El fichero XML resultante es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<mision id="0" nombre="mision.xml" validada="true" AUV="AUV CTD">

  <pdn
fichero="C:\Users\Eliezer\Documents\pruebaplanesMemoria\pdn_proyecto_g
enerada.xml" />
```

```
  <pdm
fichero="C:\Users\Eliezer\Documents\pruebaplanesMemoria\pdm_proyecto.x
ml" />

  <pdc
fichero="C:\Users\Eliezer\Documents\pruebaplanesMemoria\pdc_proyecto.x
ml" />

  <pda
fichero="C:\Users\Eliezer\Documents\pruebaplanesMemoria\pda_proyecto.x
ml" />

  <pds
fichero="C:\Users\Eliezer\Documents\pruebaplanesMemoria\pds_proyecto.x
ml" />

</mision>
```

Capítulo 9. Conclusiones

Con este proyecto se ha pretendido realizar una aproximación al problema de la planificación y control de misiones para Vehículos Submarinos Autónomos. Para ello, se ha comenzado desde el principio, analizando otras soluciones en el área de la robótica submarina en lo que se refiere a la definición de misiones para vehículos submarinos autónomos (AUVs), para a continuación definir y diseñar un formato de misiones con el que poder definir de forma completa y flexible una misión típica a llevar a cabo por un AUV.

Parte II. Análisis.

En esta parte del documento se ha realizado un estudio detallado de las misiones, analizándose primero las capacidades que pretendíamos que tuviera una misión de un AUV y diseñándose posteriormente una serie de planes que van a componer la misión a realizar por el vehículo. Además, se ha estudiado la definición del AUV y sus componentes en diversos ficheros XML.

Con el objetivo de definir el plan de navegación, se han estudiado una serie de algoritmos de planificación de rutas, evitación de obstáculos, etc que ayudarán al usuario a definir una ruta sobre un mapa.

Además se han estudiado algunos formatos que nos permitirán trabajar con información batimétrica.

Parte III. Diseño del prototipo

En esta parte del documento se diseña el prototipo para la definición y validación de misiones que se va a implementar como resultado de este Proyecto Final de Carrera. El prototipo se va a basar en la planificación de la misión, mediante la definición de las misiones.

Además, se diseñan los formatos de los paquetes de comunicación que se van a intercambiar entre el vehículo submarino autónomo y la aplicación de planificación y control.

Parte IV. Implementación

Se ha implementado un prototipo del software que soportará la definición de cada uno de los cinco planes de la misión, la definición de la misión y la validación de la misma contra la definición de varios AUVs predefinidos en el propio prototipo.

Para la definición del plan de navegación se han empleado el algoritmo de Graham para los polígonos convexos y detectar si un punto está contenido un polígono, algoritmo BUG1 para la evitación de obstáculos y el algoritmo de intersección de segmentos mediante la fórmula de la recta en forma paramétrica.

En la validación de la misión, se han implementado varios test de validación para la definición de la misión. Las validaciones implementadas son:

- Que la misión contenga los cinco tipos de planes.
- Que las medidas que se especifican tanto en los disparadores como en las acciones de las misiones están soportadas por el AUV seleccionado.
- Que la profundidad máxima durante la ruta no supera la profundidad máxima permitida por el AUV.

Capítulo 10. Trabajo Futuro

Realizar un software de planificación y control de misiones para vehículos submarinos autónomos incluyendo no sólo el proceso de definición y validación de misiones, sino también la monitorización de la misión durante su realización, así como su posterior etapa post-misión, de recuperación de información de al misión, etc. sería un objetivo muy ambicioso para circunscribir su realización completa a un Proyecto Final de Carrera. Por ello, al finalizar este proyecto, se pueden definir muchas líneas de de trabajo en las que seguir mejorando el proyecto.

A continuación se listan algunas de las líneas de trabajo que serían abordables:

- **Algoritmos más eficientes a la hora de evitar obstáculos**: El algoritmo BUG es un algoritmo poco eficiente a la hora de evitar obstáculos. Habría que estudiar la implementación de alguno de los algoritmos definidos en el análisis.

- **Algoritmos de optimización de rutas**: Actualmente el usuario define el orden que desea que siga cada uno de los componentes definidos por él en la ruta. El objetivo sería definir algoritmos de planificación y optimización de rutas que ayuden al usuario a encontrar la forma más eficiente de recorrer la ruta.

- **Completar la implementación de más propiedades:** de minirrutas y transectos en la vista del plan de navegación.

- **Interfaz más amigable y sencilla de usar**. La actual interfaz de definición de planes se ha pensado para que sea potente y completa a la hora de definir los planes. se intentó que abarcara todos los casos posibles que se habían definido en los ficheros XML de las misiones. Se podría mejorar la interfaz implementando una capa mayor de abstracción, en la que podamos definir, por ejemplo, mediciones sobre el plan de navegación.

- **Añadir las excepciones como disparadores de las tareas**. Actualmente están definidas en las interfaces los disparadores de tipo Condición e Intervalo. Habría que añadir las excepciones.

- **Implementación de las comunicaciones con el AUV**: Las comunicaciones entre el AUV y el planificador/monitorizador de la misión se han estudiado y podrían implementarse en colaboración con otros proyectos paralelos al actual que también tratan de AUVs. Sería un proyecto de simulación de AUVs y otro proyecto en el que se trata de implementar el propio sistema del control del vehículo.

- **Implementación de la monitorización**: Las comunicaciones serían necesarias a la hora de implementar un controlador de una misión para un AUV.

- **Entornos multiAUV**: En este documento se ha tratado el problema de generar una misión para un único AUV, pero en el futuro se podrían definir misiones multiAVU que involucraran a un conjunto de AUVs explorando una zona del mar y coordinándose entre ellos para llevar a cabo la misión de la forma más eficiente posible.

- **Usar una librería de geolocalización más estable**: La librería *JDIC plus* vuelve muy inestable el programa. Habría que plantearse y estudiar la opción que nos proporciona la alternativa gratuita OpenMap **[OPENMAP].**

Bibliografía

[DEEPC] *Proyecto del submarino DeepC*

www.imar-navigation.de/download/deep_c_auv.pdf

[chelsea] *Control remoto para vehículos submarinos autónomos:*

www.chelsea.co.uk/Technical%20Papers/hsl-iuuvs2000-auv.pdf

[Oporto] *Aplicaciones de monitorización de la universidad de Oporto:*

http://paginas.fe.up.pt/lsts/lsts_www/English/outfall.html

[ise] *Diseño de un AUV:*

http://www.ise.bc.ca/WADEhome.html

[Coste] *Investigación cooperativa en metodologías de programación de misiones para robots submarinos:*

http://sun-valley.stanford.edu/papers/CosteManiereW:96a.pdf

[Komerska] *Diseño 3D para la definición de rutas para un AUV*

http://www.ausi.org/publications/KomerskaWare2003.pdf

[Marius_1] *Diseño de un sistema de gestión de la misión para el AUV Marius.*

http://users.isr.ist.utl.pt/~pjcro/papers/00518615.pdf

[ROMEO] *Control de misiones a través de Internet para ROMEO usando el controlador de misiones CORAL.*

http://users.isr.ist.utl.pt/~pjcro/papers/00800140.pdf

[Marius_2] *Diseño, desarrollo, y testeo en el mar el sistema de control de la misión para el MARIUS:*

http://users.isr.ist.utl.pt/~pjcro/papers/00572781.pdf

[A*] *Algoritmo A*:*

http://www.policyalmanac.org/games/articulo1.htm

[GEOM_COMP] *Geometría computacional:*

webdelprofesor.ula.ve/ciencias/lico/geome_comp/geometria_computacional.pdf

[FLOYD] *Algoritmo de Floyd:*

http://www.pms.ifi.lmu.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization

[RADO03] *"Supervisão e Controlo da Missão de Veículos Autónomos". Rodolfo Alexandre Duarte Oliveira. Universidade Técnica de Lisboa. Instituto Superior Técnico.*

[SRM]*Apuntes de la asignatura Sistemas Robóticos Móviles de la Facultad de Informática, ULPGC. Lectura81.pdf y caminosIII.pdf.*

[IA]*Apuntes de la asignatura Inteligencia Artificial de la Facultad de Informática, ULPGC.*

[MOBOTS] Navegación para robots móviles

http://dmi.uib.es/~aortiz/mobots_navegacion.pdf

[MARIUS] *Diseño, desarrollo, y testeo en el mar el sistema de control de la misión para el MARIUS*

http://users.isr.ist.utl.pt/~pjcro/papers/00572781.pdf

[VMCONTROL] *Vehicle and Mission Control of Single and Multiple Autonomous Marine Robots*

http://welcome.isr.ist.utl.pt/img/pdfs/1505_VMissionControl.pdf

[MIMOSA] Mission management for subsea autonomous vehicles.

http://www.ifremer.fr/fleet//systemes_sm/mimosa/index.html

[W3C] *World Wide Web Consortium*

http://www.w3.org/

[XML-W3C]Lenguaje XML. Referencias en la W3C.

http://www.w3.org/XML/

http://www.w3.org/TR/2008/REC-xml-20081126/

[W3C-XQUERY] XQuery para XML

http://www.w3.org/TR/xquery/

[W3C-XPATH]XPath para XML:

http://www.w3.org/TR/xpath

[W3C-XSD]Esquema XSD.

http://www.w3.org/XML/Schema

[**W3C-DOM**] Document Object Model

http://www.w3.org/DOM/

[SAX] Simple API for XML

http://www.saxproject.org/

[JDOM] Librería JDOM

http://www.jdom.org/

[JAXP] Librería JAXP

https://jaxp.dev.java.net/

[GENERICS] Definición de Generics en Java

http://java.sun.com/docs/books/tutorial/java/generics/index.html

[NETCDF] Página principal para NetCDF.

http://www.unidata.ucar.edu/software/netcdf/

[HDFGROUP] HDF (Hierarchical Data Format)

http://www.hdfgroup.org/

[OPENMAP] Open System Mapping Tecnology

http://openmap.bbn.com/

[**MATLAB**] The Languaje of Technical Computing

http://www.mathworks.com/products/matlab/?BB=1

[JAVA] Java

http://www.java.com/es/

[NETBEANS] Entorno de desarrollo integrado

http://netbeans.org/

[JMATLINK] Librería JMatLink

http://jmatlink.sourceforge.net/

[JLAB] Librería JLab

http://www2.cmp.uea.ac.uk/~it/jlab/index.html

[ENGINE] Librería Engine

http://matlabdb.mathematik.uni-stuttgart.de/download.jsp?MC_ID=9&MP_ID=163

[GOF] Erich Gamma, Richard Helm, Ralph Johnson, John Vislides,; "Design Patterns. Elements of reusable object-oriented software", Adison Wesley, 1995. ISBN 0-201-63361-2

[JDIC] Librería JDIC

https://jdic.dev.java.net/

Apéndice A. Proyectos complementarios

Como ya se ha mencionado anteriormente, este proyecto se concibe en paralelo a otros dos proyectos de la Facultad de Informática con el objetivo de desarrollar una herramienta de exploración submarina completa, en la que dispongamos de un planificador y monitorizador de misiones, un AUV con un sistema de control que "entienda" las misiones definidas en el planificador, y un simulador en el que se puedan probar las misiones en un entorno virtual.

Los proyectos son los siguientes:

- *Sistema Integrado de Control para un Vehículo Submarino Autónomo*. El objetivo de este proyecto es el de diseñar una arquitectura de control para un AUV que permita analizar y explorar aspectos ligados al control, la navegación y la comunicación del vehículo. La aplicación finalmente desarrollada en este proyecto recibe el nombre de ***SickAUV***.

- *Entorno de simulación para el desarrollo de misiones con Vehículos Submarinos Autónomos*. El objetivo de este proyecto es el de diseñar un entorno de simulación, compatible con el sistema de control desarrollado en este proyecto, que permita analizar el desarrollo de una misión por parte de uno o varios AUVs.

- *Boyas oceanográficas*: Además, se ha trabajado en una beca en colaboración con la Facultad de Telecomunicaciones en la que el objetivo principal era el de desarrollar una aplicación que monitorizara el estado de las boyas. Además se implementó el software controlador de la boya que emitía los datos para ser recogidos por el software de control y mostrado en gráficas y en un mapa de Google.

Apéndice B. Manual de usuario.

Cuando se ejecuta la aplicación, la primera ventana que aparece es la que se muestra en la Ilustración 134.

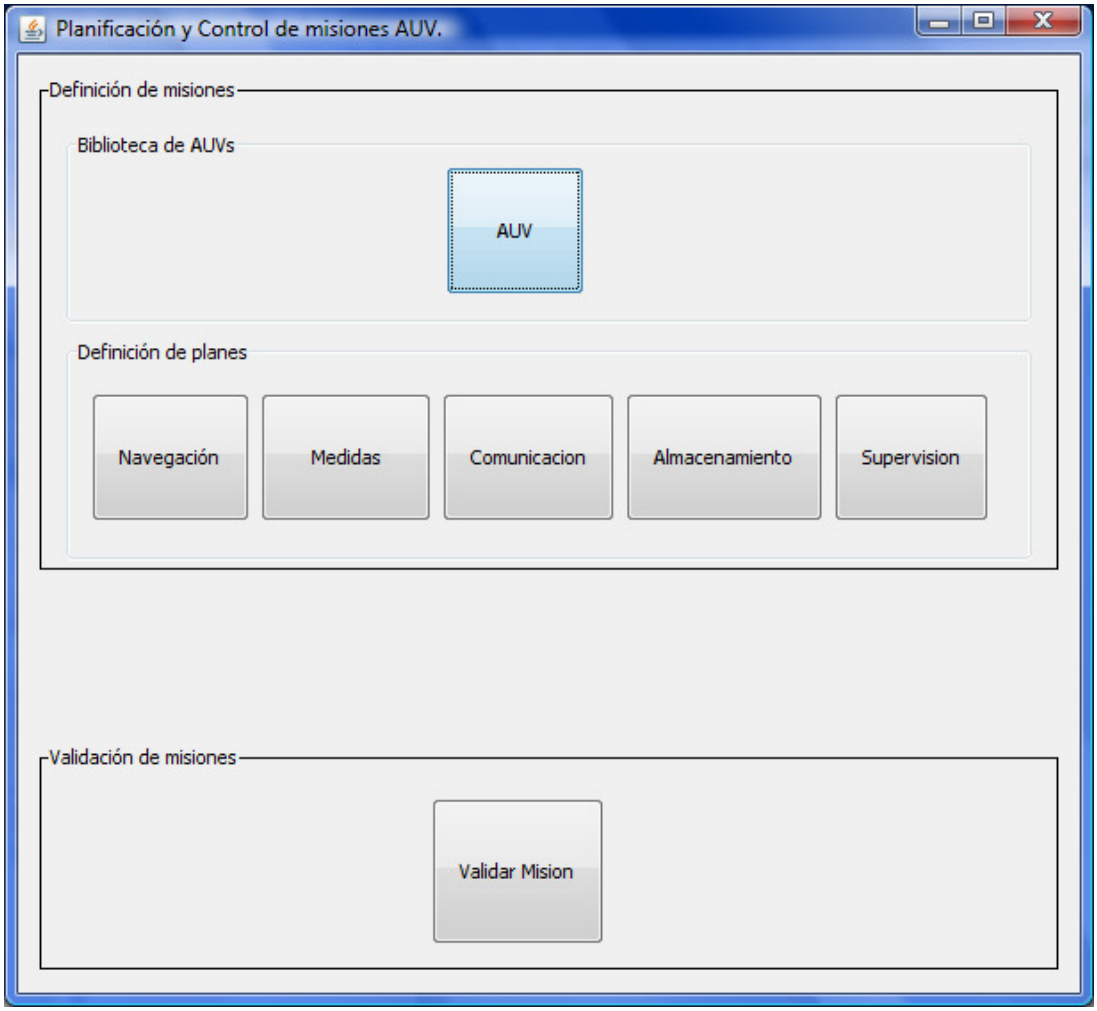


Ilustración 134: Ventana principal

Par ver los AUV que hay definidos en el sistema, se puede pulsar sobre el botón "AUV". Esto abrirá la siguiente ventana:

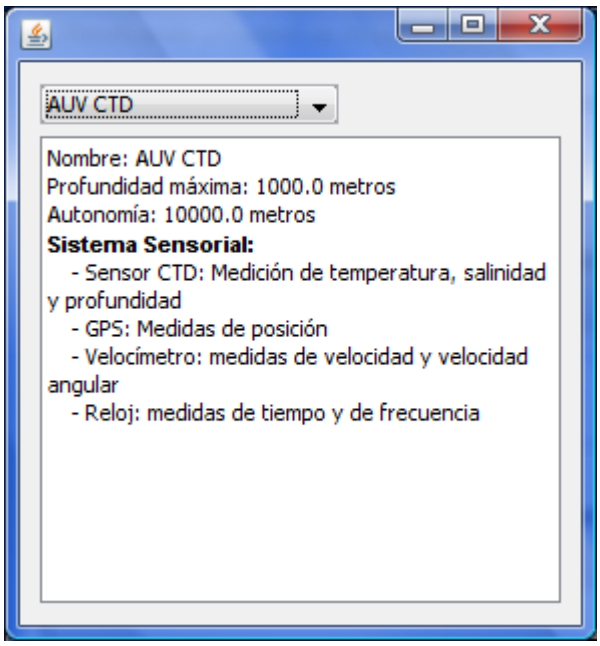


Ilustración 135: Biblioteca de AUVs

Se puede ver la Ilustración 135 una lista desplegable donde se muestra la lista de AUVs definidos, y debajo de éste, la descripción del AUV seleccionado. Si se desea ver otro AUV, solo tenemos que seleccionarlo en la lista desplegable para que se muestre su información.

Esta ventana la podremos tener visible aunque estemos trabajando con otras ventanas de la aplicación. Normalmente, en esta aplicación, podremos tener abiertas varias ventanas de distinto tipo para poder referenciarlas lo más rápido posible.

10.1 Plan de Navegación

Si queremos definir un plan de navegación tenemos que pulsar en el botón "Navegación" de la interfaz principal, como vemos en la Ilustración 136.

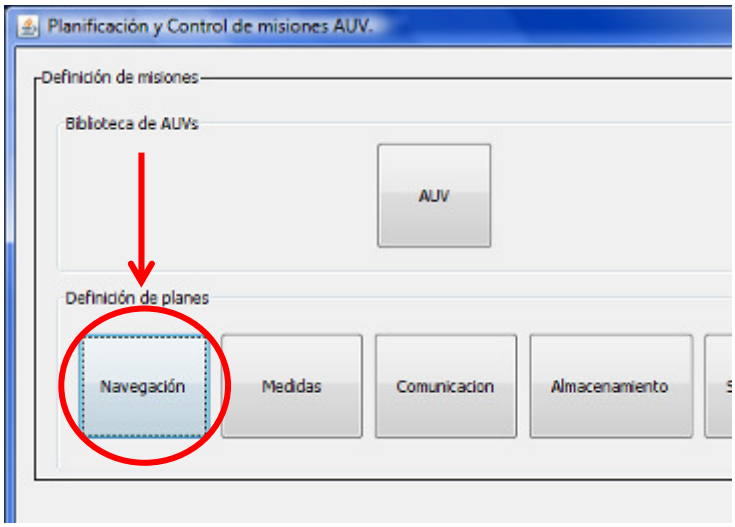


Ilustración 136: Botón Navegación

Al pulsar ese botón, se mostrará la ventana que se muestra en la Ilustración 137.

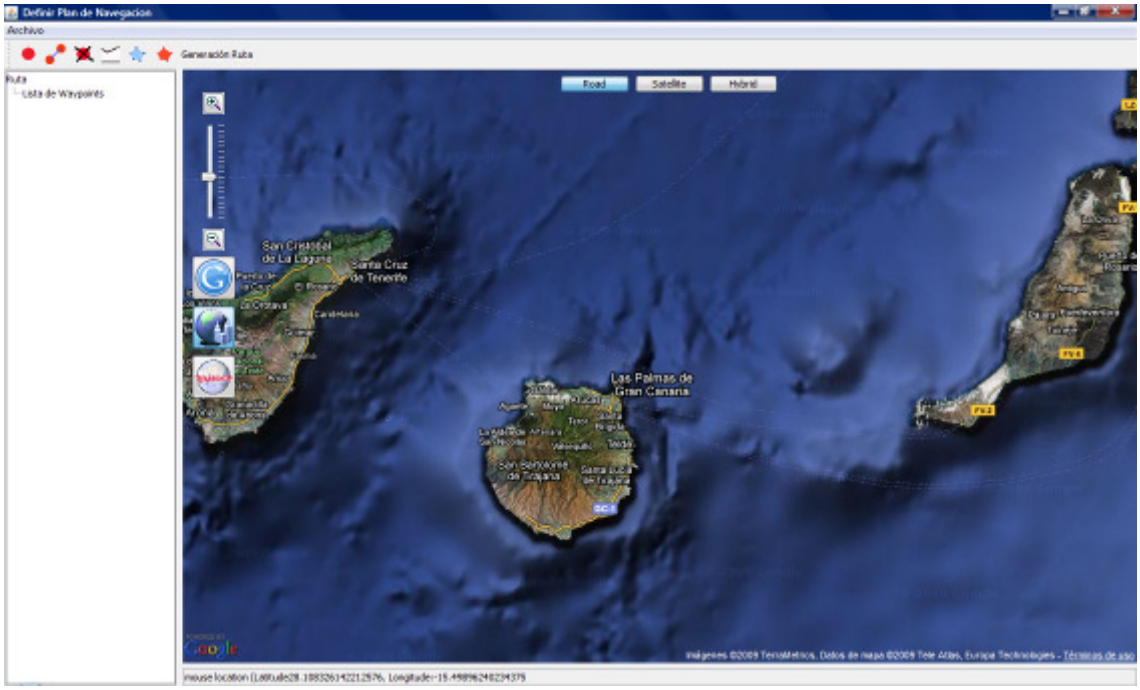


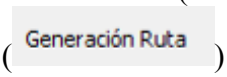
Ilustración 137: Plan de navegación

Hay un árbol ubicado a la izquierda de la ventana, en el que se listarán todos los componentes que agreguemos a la ruta y el orden en el que serán recorridos. A la derecha de este árbol, y ocupando la mayor parte de la ventana, se encuentra en mapa sobre el que definiremos la ubicación de los componentes de la ruta.

En el árbol, se puede ver que el primer nodo se denomina “Lista de Waypoints”. Se ha añadido este nodo, ya que cuando se comiencen a generar las minirrutas, todos los waypoints estarán dentro de éstas. De esta forma, si necesitamos acceder de una forma más rápida y simple al waypoint, solo tenemos que mostrar la lista de waypoints almacenada en esta lista, sin necesidad de acceder a la minirruta que lo contiene.

En la parte superior de la pantalla aparece una serie de botones que servirán para definir los componentes de la ruta, tal y como se ve en la Ilustración 138.

Ilustración 138: Barra de herramientas

El primer botón se utiliza para insertar waypoints en el mapa, el segundo () para insertar transectos, el tercero () para eliminar componentes de la ruta (cualquiera), el cuarto () no tiene utilidad a día de hoy, aunque fue pensado en su día para establecer profundidades de una forma cómodo para el usuario, el quinto inserta áreas (), el sexto zonas prohibidas (), y por último el séptimo botón (

) genera una ruta a partir de lo introducido por el usuario.

10.1.1 Waypoints

Al pulsar sobre el botón de insertar waypoints, éste se mantendrá seleccionado hasta que se vuelva a pulsar. Mientras este botón esté seleccionado, cada vez que se pulse sobre el mapa se insertará un nuevo waypoint.

Los waypoints pueden ser de 3 colores distintos:

Ilustración 139: Waypoints

- **Verde**: Será el waypoint inicial, el punto de partida de la misión.
- **Rojo**: Serán los waypoints normales, los intermedios entre el waypoint inicial y el final.
- **Negro**: Será el waypoint final. El punto en el que la misión debería finalizar.

Otro color con el que se puede mostrar un waypoint es el azul oscuro, al igual que todos los elementos que estén seleccionados.

Cuando se crea un waypoint lo estamos ubicando en una latitud-longitud del mapa, pero también se pueden definir otras propiedades del waypoint como son la profundidad o la velocidad con la que el submarino debe pasar por ese punto.

Para ello se puede ir a la ventana de propiedades del waypoint. Se puede acceder a ella seleccionando el waypoint en el árbol de la izquierda y pulsando con el botón derecho del ratón, con lo que se abrirá un menú de contexto, en el que habrá que pulsar sobre el botón propiedades, com ose puede ver en al Ilustración 140.

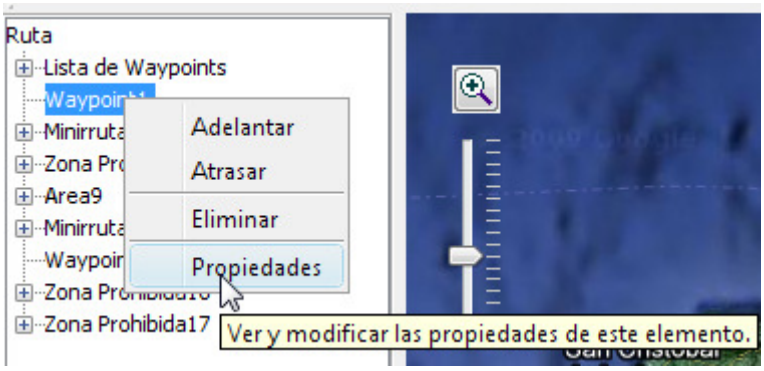


Ilustración 140: Menú contextual del Waypoint

En la ventana que aparece se pueden establecer todas esas opciones que permiten definir el waypoint. Se puede ver esta ventana en la Ilustración 141.

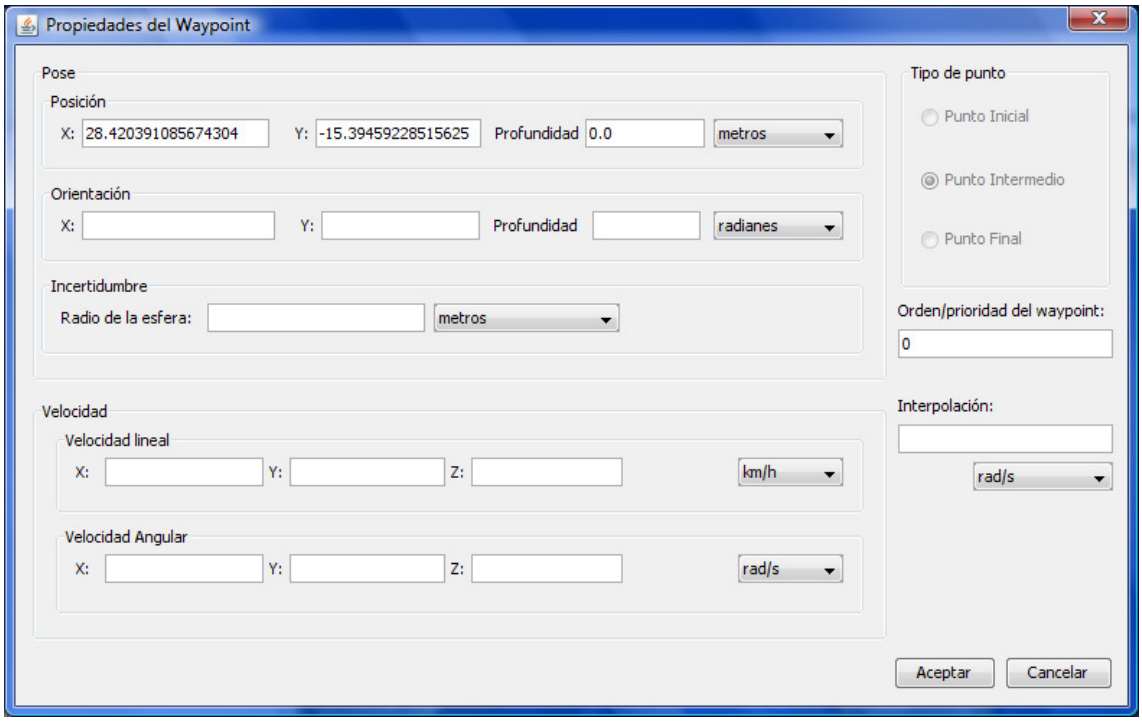


Ilustración 141: Propiedades Waypoint

Al pinchar sobre el botón "Aceptar" se guardan estas opciones.

10.1.2 Transectos y Minirrutas

Para crear un transecto, y con ello una minirruta, se pulsa el botón  . Este botón estará pulsado mientras no se vuelva a hacer clic en él, al igual que lo que ocurría con el botón para insertar waypoints.

El requisito indispensable para insertar un transecto es que haya al menos un waypoint en la ruta. Solo se necesita uno ya que, mientras se edita la ruta, una minirruta puede contener un solo waypoint y después ir ampliándose.

Una vez se haya pulsado ese botón, se puede ir seleccionando los waypoints que se desean unir hasta que se complete la minirruta que se desea crear.

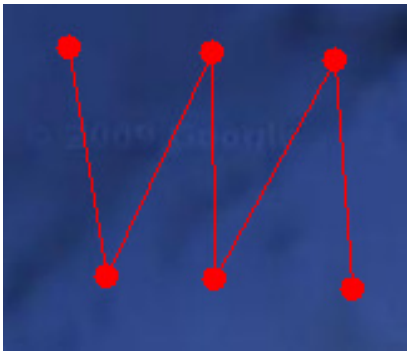


Ilustración 142: Minirruta

En el árbol de la izquierda habrán desaparecido los waypoints del primer nivel para insertarse dentro de las minirrutas.

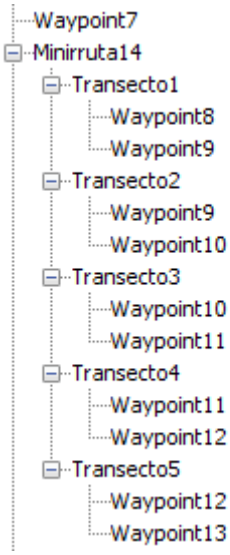


Ilustración 143: Árbol de la ruta

Aún no están implementadas las propiedades de las minirrutas ni de los transectos, por lo que no tiene ninguna función el botón "Propiedades" ubicado en el menú contextual que aparece al seleccionar un transecto o una minirruta y pulsar el botón derecho del ratón en este árbol.

Si una vez insertada una minirruta en el plan de navegación, se necesita insertar un nuevo waypoint al principio o final de ésta, se puede seleccionar la opción "Insertar wp inicio" o "Insertar wp final", tal y como se puede ver en la Ilustración 144.

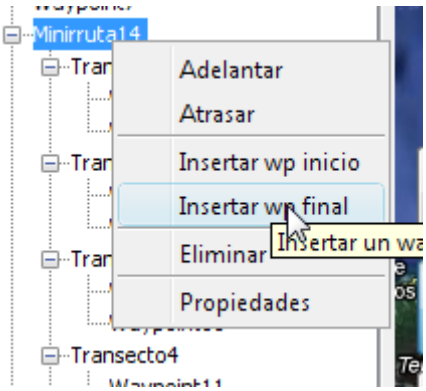


Ilustración 144: Menú contextual de la Minirruta

Se pueden insertar waypoints (por el inicio o por el final) hasta que se deseleccione el botón  .

10.1.3 Áreas

Las áreas se insertan pulsando el botón  . Mientras este botón esté seleccionado se puede seguir insertando vértices del área. Cuando se haya terminado de insertar el área, se volverá a pulsar este botón para dejar de añadir vértices al mismo.

Ilustración 145: Área

A las propiedades del área se accede de forma similar que los anteriores componentes, seleccionándolo en al árbol y pulsando el botón "Propiedades" en el menú de contexto.

Aparecerá la siguiente ventana que se muestra en la Ilustración 146.

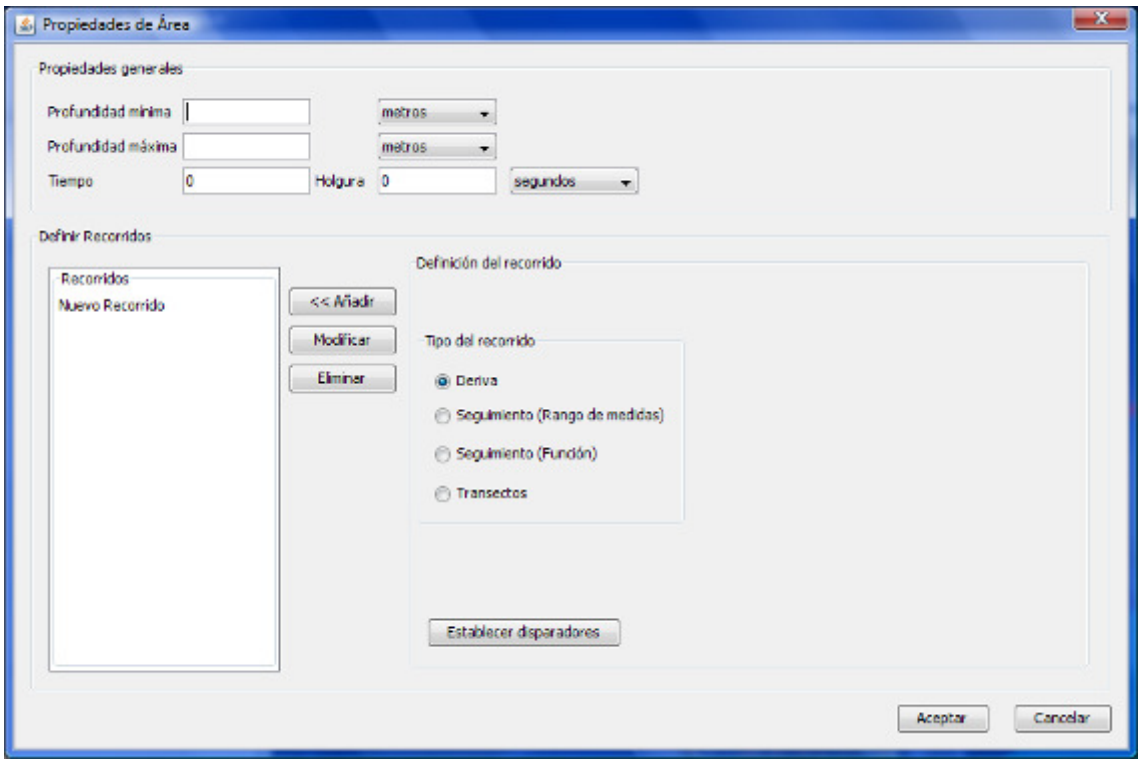


Ilustración 146: Propiedades del área

En la parte superior de la ventana se puede definir algunas propiedades, como la profundidad máxima y mínima y el tiempo que tardará como máximo en recorrer el AUV el área.

En la parte inferior se pueden definir una lista de recorridos, que serán las distintas formas que tendrá el submarino de recorrer el área, según los disparadores establecidos.

Para definir un recorrido, primero se tiene que especificar que tipo de recorrido se desea establecer en el panel que se muestra en la Ilustración 147.

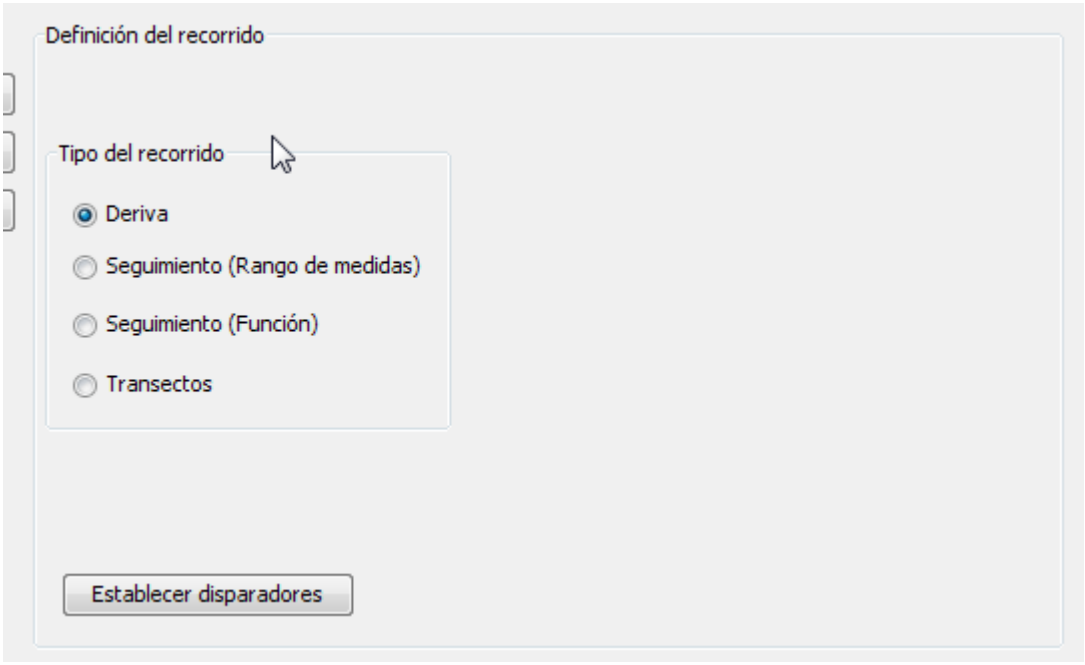


Ilustración 147: Recorrido del área

Según el tipo de recorrido que se escoja se tendrá unas opciones u otras a la hora de definirlo. Como se ha visto, para el tipo Deriva no hace falta definir nada más, ya que será simplemente el submarino dejándose llevar por las corrientes En los demás tipos se tendrán las siguientes opciones:

- Tipo Seguimiento (Rango de medidas):

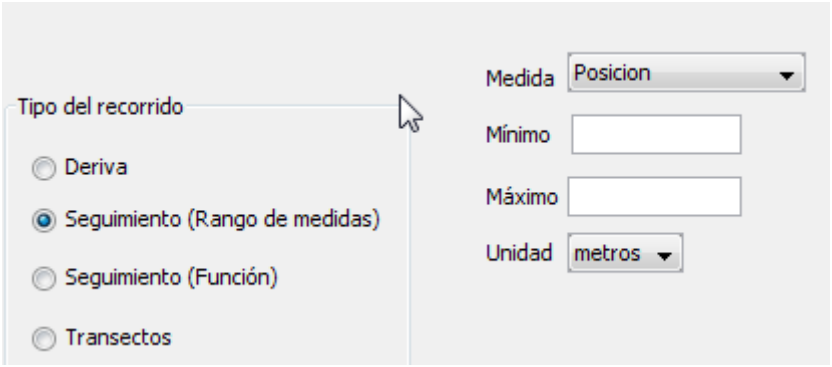


Ilustración 148: Recorrido Seguimiento

- Tipo Seguimiento (Función)

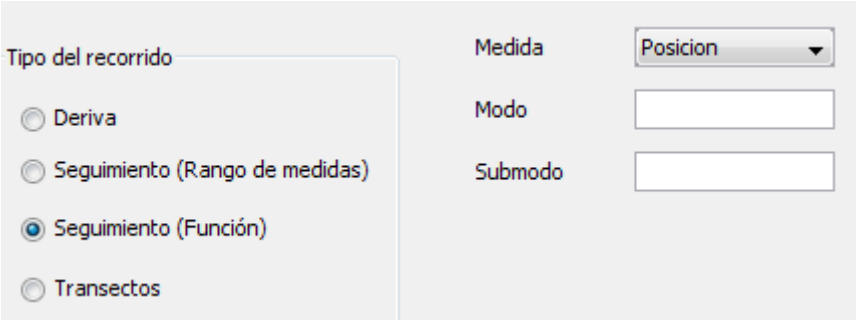


Ilustración 149: Recorrido Seguimiento Función

- Tipo Transectos:

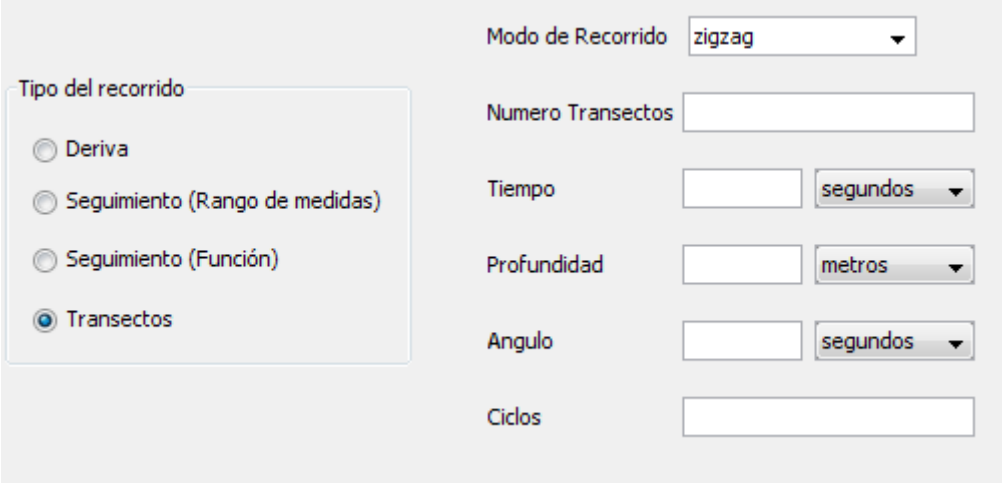


Ilustración 150: Recorrido Transectos

Una vez definido el tipo de recorrido, habrá que establecer unos disparadores que indicarán cuándo se activará este tipo de recorrido. Para ello se pulsa el botón "Establecer Disparadores". Aparecerá la ventana que se muestra en la Ilustración 151.

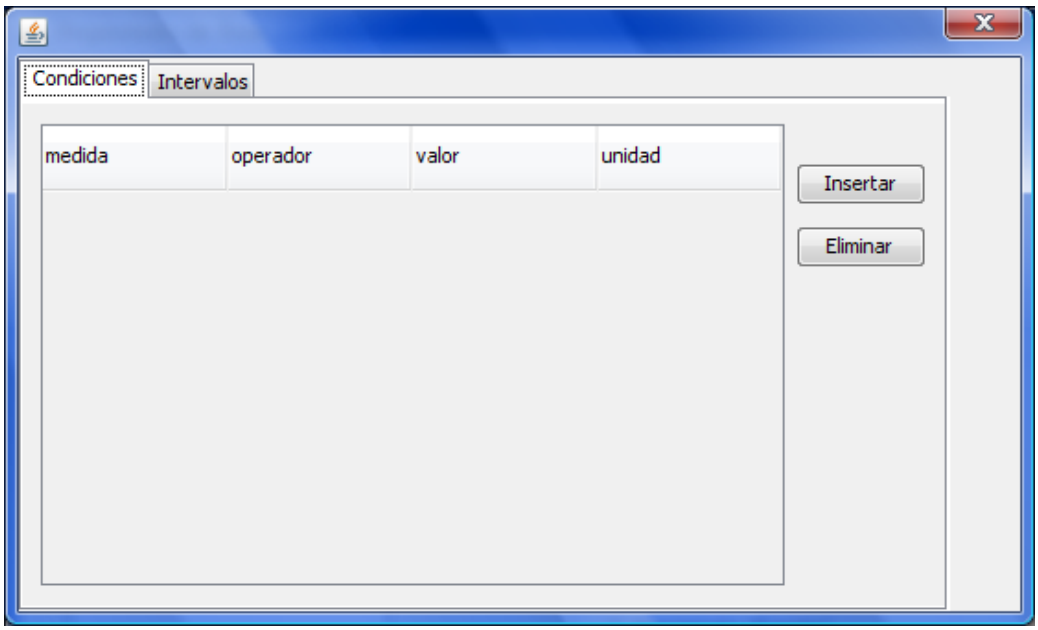


Ilustración 151: Disparadores

En ella se podrá seleccionar la pestaña para añadir condiciones o la pestaña para añadir intervalos. Si se le da al botón "Insertar" se insertará una nueva línea en la tabla y se podrá definir una nueva condición o el intervalo, según lo que se haya seleccionado. Se tienen que rellenar todos los campos de la fila para que se de por buena esa condición o intervalo.

Una vez establecido el modo de recorrido y los disparadores, se pulsa el botón "Añadir" para añadir ese tipo de recorrido a la lista de recorridos del área. También se tendrá la opción de modificar o eliminar recorridos de la lista.

Cuando lo se tenga todo definido se le da al botón "Aceptar" para salir de la ventana.

Se tiene la opción de añadir más vértices del área utilizando el menú contextual que se abre al darle con el botón derecho del ratón sobre el área en el árbol de la izquierda. Para eliminarlos, se puede seleccionar el vértice e y darle al botón "Eliminar".

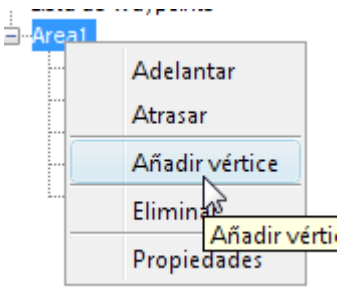


Ilustración 152: Menú contextual Área

10.1.4 Zonas Prohibidas

Una zona prohibida se insertará de forma similar a un área. Se irán añadiendo vértices para ir construyendo un polígono convexo. El botón para insertar zonas prohibidas () se mantendrá pulsado mientras se siga insertando vértices para esa zona prohibida. Para terminar de insertar la zona prohibida, se volverá a pulsar sobre el botón.

10.1.5 Eliminando componentes

Para eliminar un componente de una ruta se tendrán dos opciones. Una es seleccionar el componente en el propio mapa. Se sabrá que está seleccionado puesto que cambiará de color a uno azul oscuro, tal y como se puede ver en Ilustración 153.

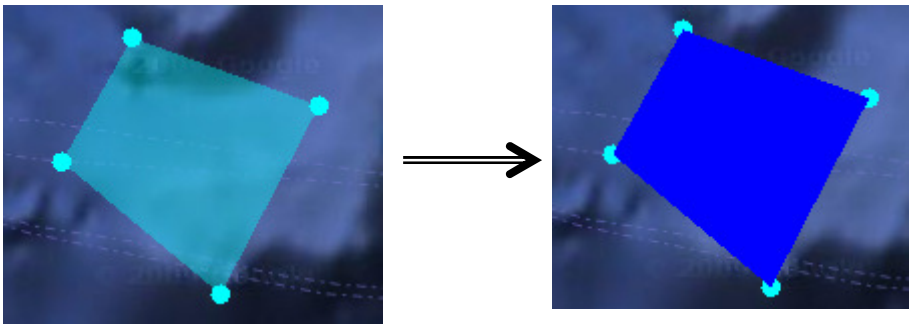


Ilustración 153: Área seleccionada

Para eliminarlo se puede pulsar el botón  . También se puede seleccionar un vértice de un área o una zona prohibida para eliminar el vértice.

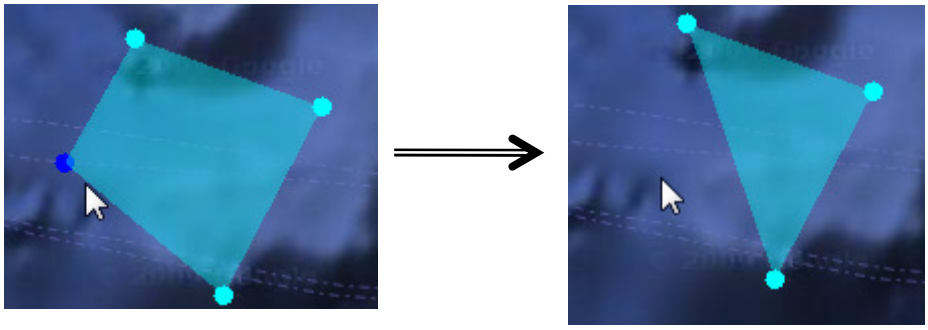


Ilustración 154: Vértice eliminado

La otra forma de eliminar un componente es seleccionándolo en el árbol y dándole a eliminar en el menú contextual que aparece al darle al botón derecho del ratón.

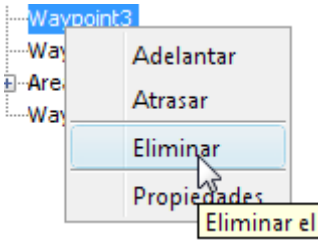


Ilustración 155: Menú contextual del Waypoint

10.1.6 Establecer orden de recorrido

El árbol situado a la izquierda de la ventana, indicará el orden en el que se recorrerán los componentes de la ruta. Se puede modificar el orden con el menú contextual que aparece al darle con el botón derecho sobre cualquier componente de la ruta y pulsar en "Adelantar", para que ese componente se mueva una posición hacia arriba, o darle a "Atrasar" para que ese componente se mueva una posición hacia abajo.

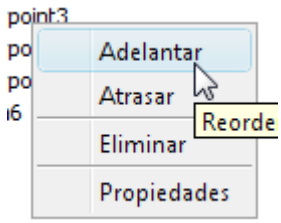


Ilustración 156: Estableciendo el orden

10.1.7 Generación de una ruta

Una vez se tenga el boceto de la ruta, se puede generar una ruta completa. Para generar una ruta se usa el botón "Generar Ruta".

Al darle a este botón la aplicación unirá todos los componentes de la ruta mediante minirrutas para que el submarino pueda recorrerla por completo.

10.1.8 Guardar/Cargar un plan de navegación

Para guardar un plan de navegación se puede seleccionar el menú "Archivo" → "Guardar Plan Navegación". Se selecciona una ubicación cualquiera y el nombre del fichero y se le da a guardar.

Para cargar un plan de navegación se va al menú "Archivo" → "Cargar Plan Navegación". Como antes, se selecciona el fichero a cargar y al cargar el plan debería actualizarse la interfaz.

¡¡¡IMPORTANTE!!!:

El fichero debe almacenarse en una ruta que no contenga espacios, debido a que la versión del parseador XERCES usada no permite espacios en el pathname del fichero.

10.2 Plan de medidas, de almacenamiento, de comunicaciones y de supervisión

Cualquiera de estos cuatro planes se definirá de una forma similar. Solo hay que seleccionar el plan que se desea definir en la vista principal de la aplicación, y especificar cada una de sus tareas.

Si se quiere crear un plan de medidas, por ejemplo, se debe pulsar el botón "Medidas" en la ventana principal de la aplicación.

El plan de medidas consta e una serie de tareas. Esta lista de tareas se verá en el árbol situado a la izquierda de la ventana. Se puede ver en la Ilustración 157.

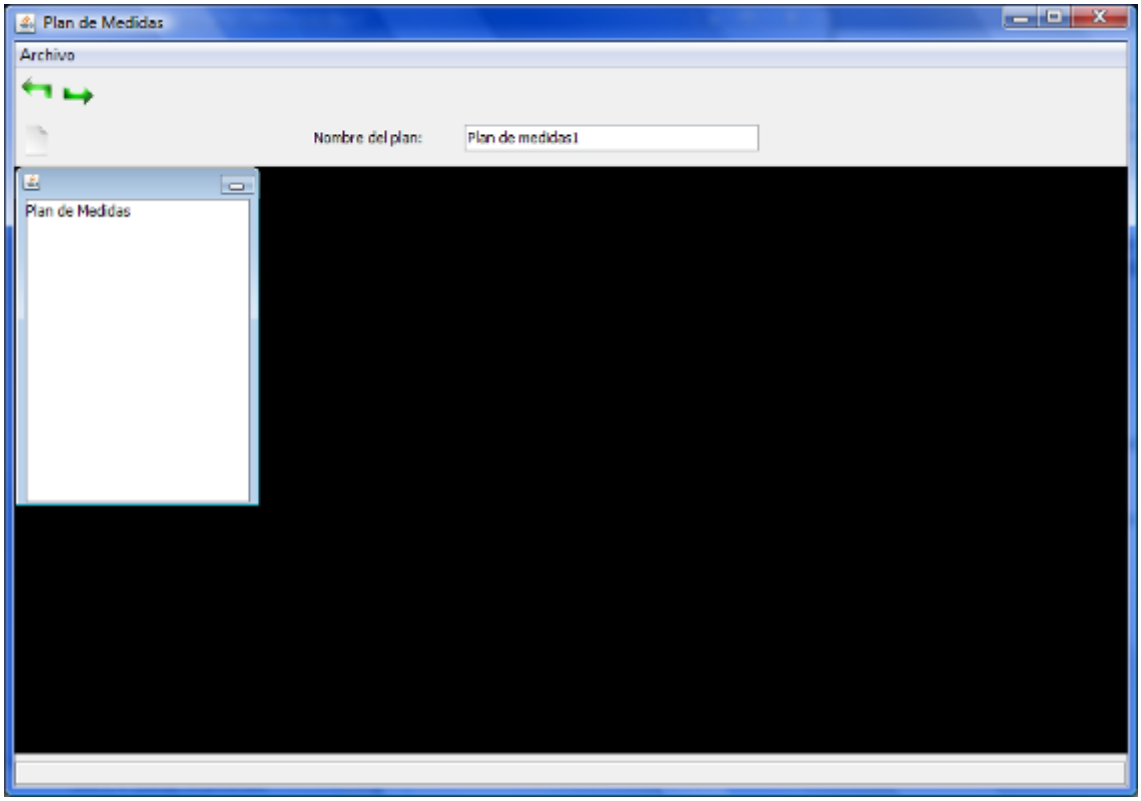


Ilustración 157: Plan de medidas

Si se quiere añadir una tarea hay que darle al botón de "Nueva Tarea", tal y como se muestra en la Ilustración 158.

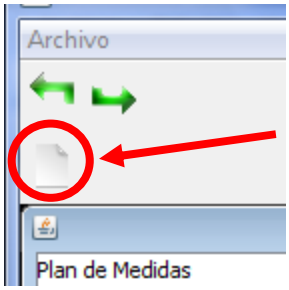


Ilustración 158: Nueva Tarea

Se abrirá una ventana que pedirá el nombre de la nueva tarea(ver Ilustración 159).

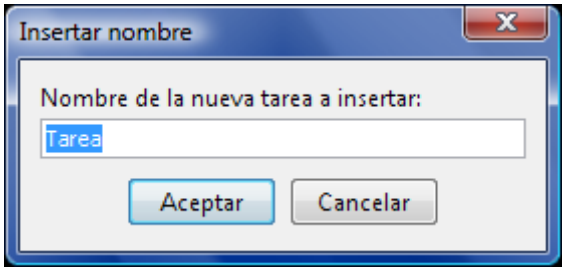


Ilustración 159: Nombre tarea

Al seleccionar un nombre y darle al botón "Aceptar" se añadirá a la lista de tareas que se encuentra en el árbol de la izquierda, tal y como vemos en la Ilustración 160.

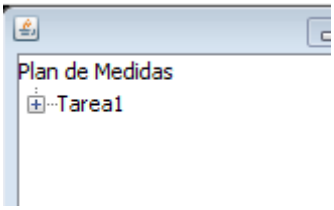


Ilustración 160: Árbol de tareas

Y se abrirá una ventana donde se podrá definir la tarea. La ventana se muestra en la Ilustración 161.

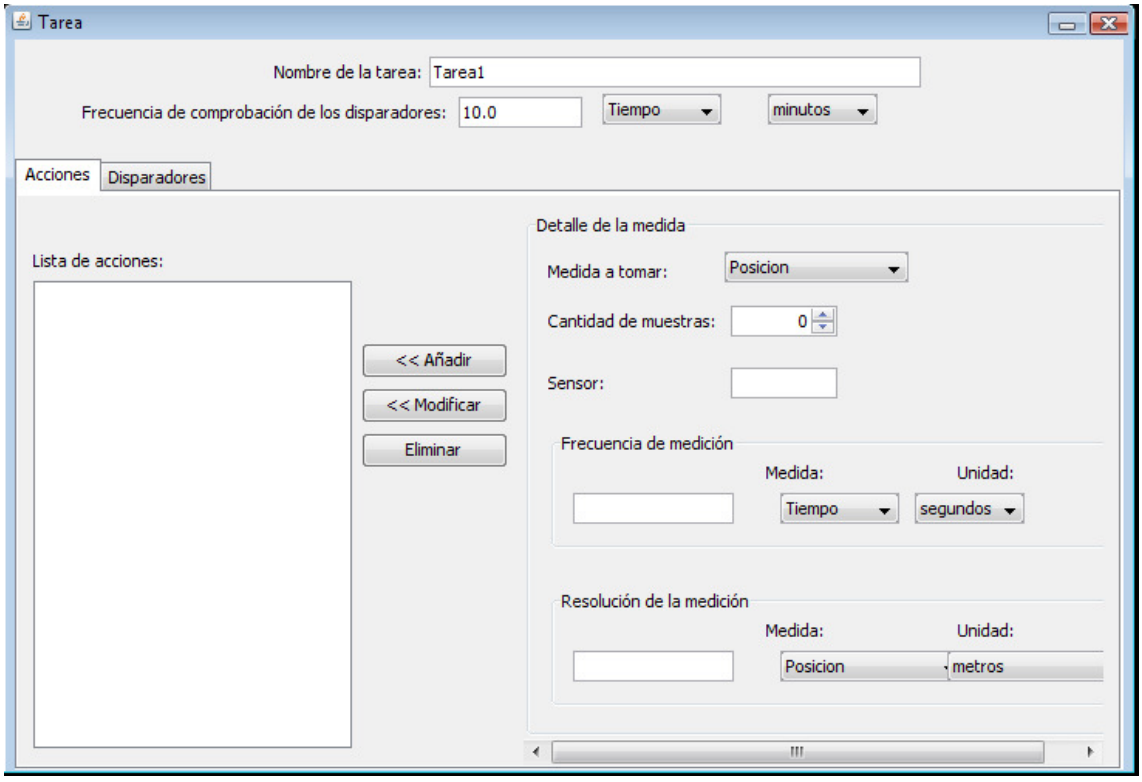


Ilustración 161: Ventana Tarea

Una tarea consta de varias acciones y disparadores. En la pestaña de acciones se puede añadir una lista de acciones, modificar alguna de ellas o eliminarlas. Esta vista será distinta para cada plan.

La pestaña de disparadores, donde se definirán las condiciones que deben cumplirse para que se ejecute la tarea será similar a todos los planes, y como ya se ha visto en el plan de navegación.

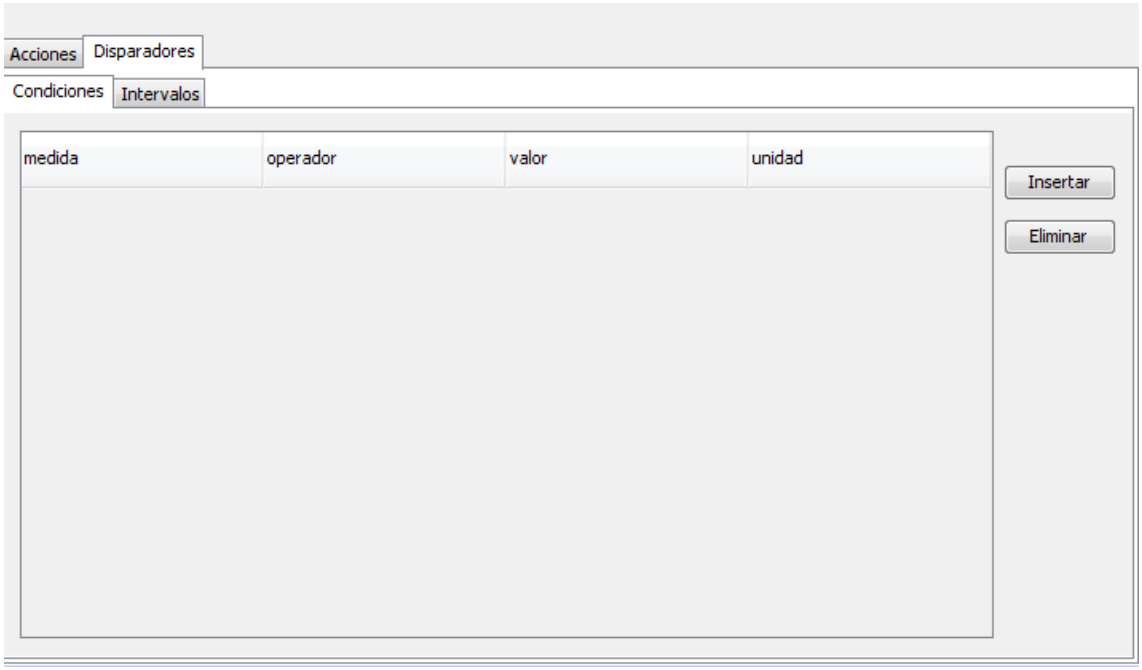


Ilustración 162: Disparadores

Se tiene, en este tipo de planes, los comandos deshacer y rehacer, que se ejecutan pulsando los botones  .

Estos planes también se pueden guardar y cargar de ficheros XML, de la misma forma que se cargaban y almacenaban planes en el plan de navegación.

10.3 Biblioteca de AUVs

Pero para poder hacer algunas pruebas, se han definido en el código dos AUVs cuyas características se pueden ver si se pulsa el botón AUV de la ventana principal.

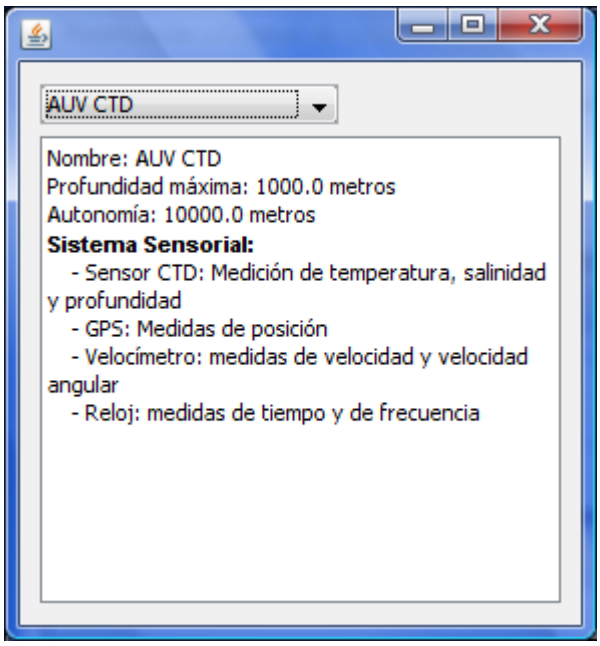


Ilustración 163: Biblioteca de AUVs

10.4 Validación de misiones

Para crear y validar una misión se pulsa sobre el botón "Validar Misión" en la vista principal de la aplicación. Aparecerá la ventana mostrada en la Ilustración 165

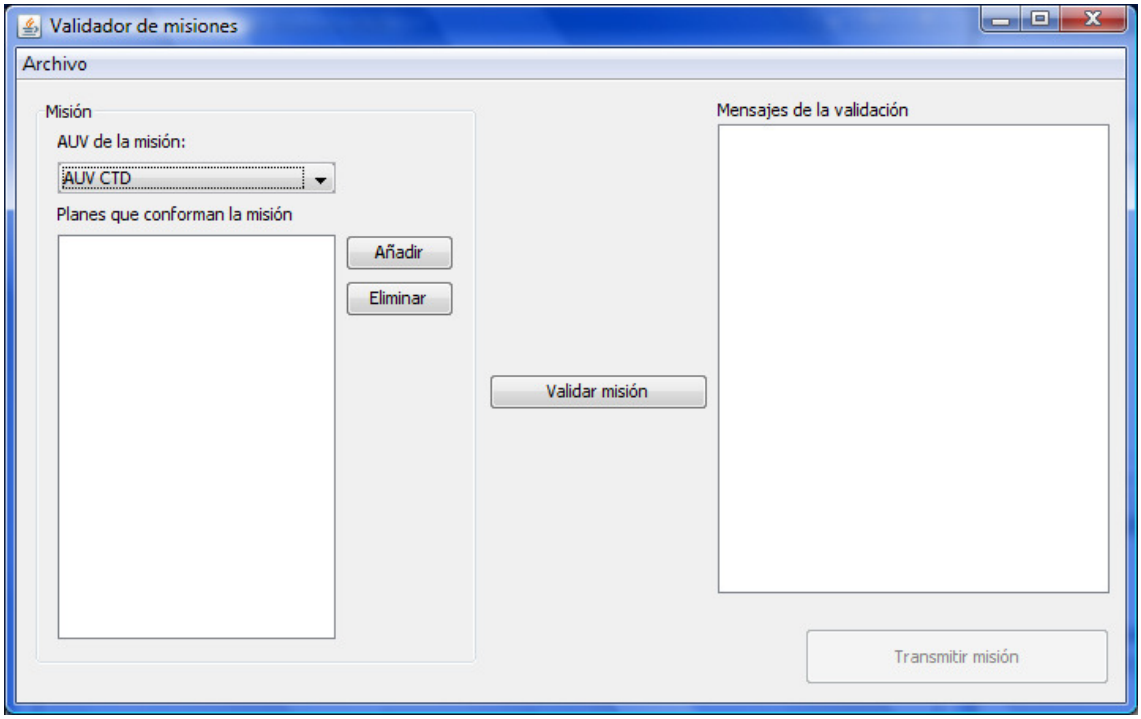


Ilustración 164: Validador de misión.

Ilustración 165: Validador de misiones

Como se puede ver, en la parte izquierda de la ventana se da la opción de seleccionar un AUV. En este momento hay dos AUV definidos en el código.

En la lista que se encuentra debajo de la selección del AUV, se pueden añadir los planes que van a conformar la misión dándole al botón "Añadir". Se pueden seleccionar varios ficheros. La aplicación detectará automáticamente, cuando cargue los planes, de qué tipo de plan se trata.

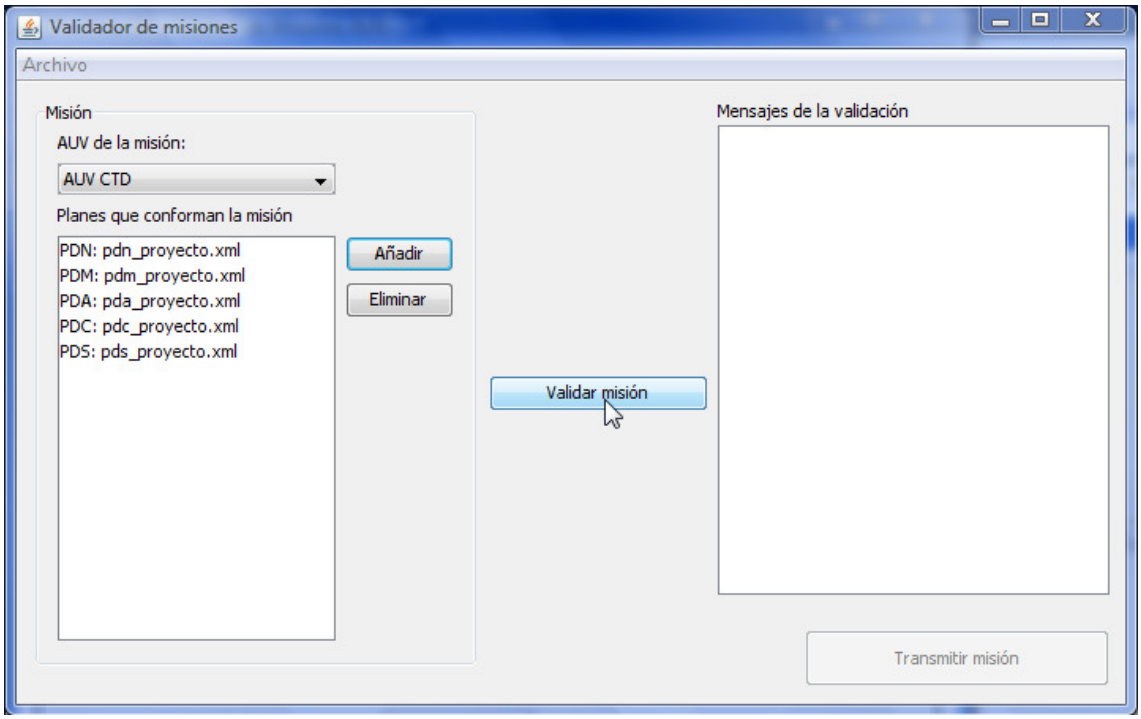


Ilustración 166: Validador de la misión(2)

Cuando se hayan seleccionado todos los planes y el AUV que se pretende utilizar, tal y como se muestra en la Ilustración 166, se puede darle al botón "Validar misión". Se mostrarán los errores en el panel de la derecha. En caso de que no se hayan producido errores, se activará el botón "Transmitir misión".

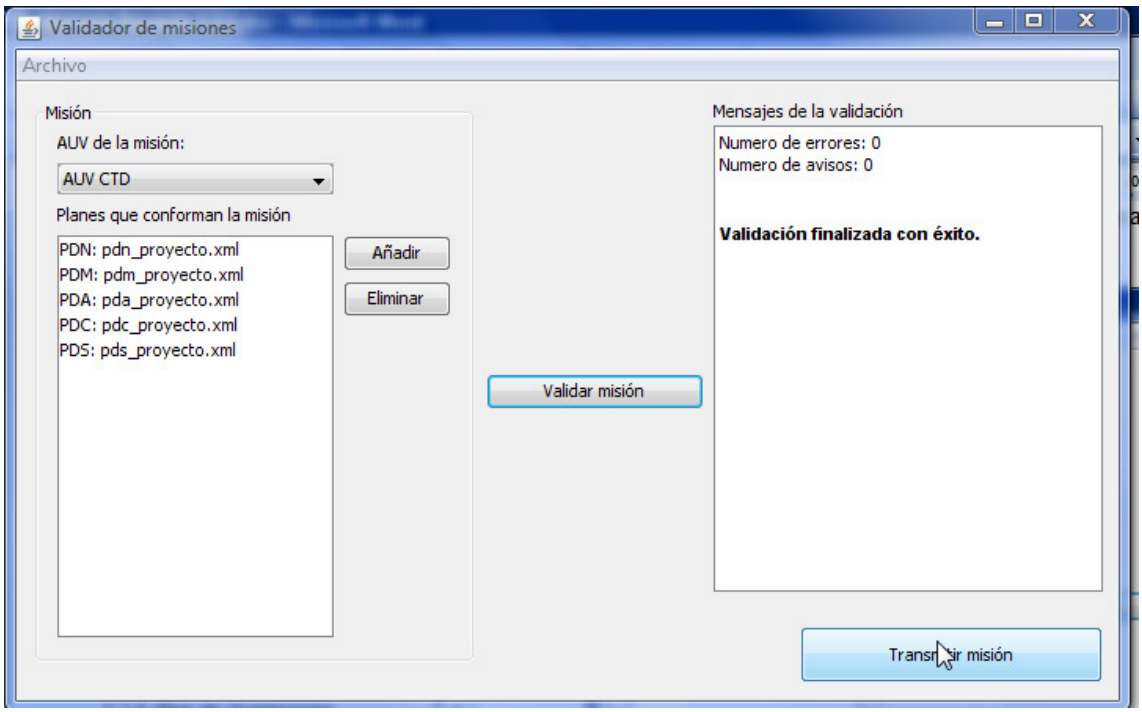


Ilustración 167: Validador de la misión (3)