Escuela Técnica Superior de Ingeniería del Diseño
Universidad Politécnica de Valencia
Grado en Ingeniería Aeroespacial

TRABAJO FIN DE GRADO

MONITORIZACIÓN GRÁFICA DEL TRÁFICO AÉREO MEDIANTE LA PLATAFORMA OPENMAP

Autor: Fredd Alexander Duque Farías
Tutor: Ángel Rodas Jordá

Valencia, Julio 2017

Escuela Técnica Superior de Ingeniería del Diseño
Universidad Politécnica de Valencia
Grado en Ingeniería Aeroespacial

TRABAJO FIN DE GRADO

MONITORIZACIÓN GRÁFICA DEL TRÁFICO AÉREO MEDIANTE LA PLATAFORMA OPENMAP

Autor: Fredd Alexander Duque Farías
Tutor: Ángel Rodas Jordá

Valencia, Julio 2017

Agradecimientos

A mi familia por su incondicional apoyo. Decisión a decisión a lo largo de mi vida han estado ahí, dándome esa pequeña palmada en la espalda que a todos alguna vez nos hace falta. A mi padre, a sus largas charlas y consejos. Al amor de mi madre. A la gracia de mis hermanos que siempre me ha hecho esforzarme por ser mejor. A mis compañeros de clase, a los que debo grandes historias a lo largo de estos cuatro años y sin los cuales hoy no sería el mismo. A mis compañeros de piso por recordarme algo que casi había olvidado. A mi tutor, por su comprensión, adaptabilidad y, sobre todo, trato cercano.

A todos aquellos que han compartido conmigo esta aventura, mi más sincero y sentido agradecimiento por darme la oportunidad de crecer a vuestro lado.

Resumen

El presente trabajo fin de grado desarrollará una aplicación que de soporte a la monitorización de tráfico aéreo, estableciendo un firme marco de programación con carácter didáctico y fácilmente escalable. Se creará juntamente con la plataforma de libre uso OpenMap, cuyas librerías darán lugar a una interfaz gráfica moderna y llena de herramientas. Esta aplicación se desarrollará en Java, lo que permitirá su fácil difusión, modificación y cooperación entre la comunidad aeronáutica de la Universidad Politécnica de Valencia. Pero para todo ello, se realizará previamente un recorrido por las diferentes aproximaciones al diseño de un radar que se han visto a lo largo del grado en Ingeniería Aeroespacial, para finalmente acabar con esta nueva herramienta que se dejará en manos de la universidad para futuros proyectos.

Palabras clave: Java, monitorización del tráfico aéreo, OpenMap

Resum

El present treball fi de grau desenvoluparà una aplicació que de suport a la monitorització de trànsit aeri, establint un ferm marc de programació amb caràcter didàctic i fàcilment escalable. Es crearà juntament amb la plataforma de lliure ús OpenMap, les llibreries del qual donaran lloc a una interfície gràfica moderna i plena de ferramentes. Aquesta aplicació es desenvoluparà en Java que permetrà la seva fàcil difusió, modificació i cooperació entre la comunitat aeronàutica de la Universitat Politècnica de València. Però per a tot això, es realitzarà prèviament un recorregut per les diferents aproximacions al disseny d'un radar que s'han vist al llarg del grau en Enginyeria Aeroespacial, per finalment acabar amb aquesta nova eina que es deixarà en mans de la universitat per a futurs projectes .

Paraules clau: Java, monitorització del trànsit aeri, OpenMap

Abstract

The following final degree project, will develop an application that supports the air traffic monitoring, establishing a firm programming framework with a didactic character and easily scalable. It will be created along with OpenMap, a free-use platform whose libraries will give place to a modern graphical interface and full of tools. This application will be developed in Java, which will allow its easy diffusion, modification and cooperation between the aeronautical community of the Polytechnic University of Valencia. But for all of this, previously we will study different approaches that have been seen to the design of a radar throughout the degree in Aerospace Engineering, to finally ending with a new application that will be ceded to the university for future projects.

Keywords: Java, air traffic monitoring, OpenMap

Índice General

Índice

Índice Figuras

Índice de tablas

Índice de captura código

Índice Anexo

Glosario

- **Add:** del inglés añadir.
- **Applet**: es un componente de una aplicación que se ejecuta en el contexto de otro programa, por ejemplo, en un navegador web.
- **Buffer**: en programación es un espacio de memoria, en el que se almacenan datos de manera temporal, su principal uso es para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia (entrada/salida) de datos irregular o por la velocidad del proceso.
- **Callsign**: es una palabra en inglés compuesta por call (llamada) y sign (signo). Es usada en la radiocomunicación para designar a uno de sus participantes. En nuestro case se refiera a la aerolínea de la nave que lo transmite.
- **CheckBox**: del inglés, casilla de verificación.
- **Datum**: es un conjunto de puntos de referencia en la superficie terrestre con los cuales las medidas de la posición son tomadas y un modelo asociado de la forma de la tierra (elipsoide de referencia) para definir el sistema de coordenadas geográfico.
- **Framework**: del inglés marco de trabajo.
- **Generate:** del inglés generar.
- **Label:** del inglés etiqueta.
- **Placemarck**: Una marca de posición (Placemark) es un recurso (Feature) con un elemento geométrico (Geometry) asociado.
- **Proyección Mercator**: es un tipo de proyección cilíndrica tangente al Ecuador. Como tal, deforma las distancias entre los meridianos (en la tierra son como "gajos" de polo a polo) en líneas paralelas, aumentando su ancho real cada vez más a medida que se acerca a los polos.
- **Proyección ortográfica**: es un sistema de representación gráfica que consiste en representar elementos geométricos o volúmenes en un plano mediante proyección ortogonal. Se obtiene de modo similar a la «sombra» generada por un «foco de luz» procedente de una fuente muy lejana. Su aspecto es el de una fotografía de la Tierra.
- **Remove**: del inglés borrar.
- **Render**: del inglés reproducir o representar
- **ShapeFile**: es un formato sencillo y no topológico que se utiliza para almacenar la ubicación geométrica y la información de atributos de las entidades geográficas. Dichas entidades se pueden representar por medio de puntos, líneas o polígonos (áreas).

- **Squawk**: es un código de identificación de 4 dígitos que se inserta en el transponder de una aeronave.

- **Stream**: en Java medio utilizado para leer datos de una fuente y para escribir datos en un destino. Tanto la fuente como el destino pueden ser archivos, sockets, memoria, cadena de caracteres o procesos.

- **TextField:** del inglés campo de texto.

- **Thread**: del inglés hilo. En Java corresponde a una clase que nos permite realizar diversas tareas al mismo tiempo, es decir, ejecutar varios hilos a la vez.

- **Waypoint:** son coordenadas para ubicar puntos de referencia tridimensionales utilizados en la navegación. La palabra viene compuesta del inglés way (camino) y point (punto).

Siglas

- **ADS-B**: Sistema de Vigilancia Dependiente Automática, por sus siglas en inglés Automatic Dependent Surveillance Broadcast
- **API**: Interfaz de Programación de Aplicaciones, por sus siglas en inglés Application Programming Interface.
- **ATC**: control de tráfico aéreo, por sus siglas en Inglés Air Traffic Control.
- **CSV**: Valores Separados por Comas, por sus siglas en inglés Comma-Separated Values.
- **GNSS**: Sistema Global de Navegación por Satélite, por sus siglas en Inglés Global Navigation Satellite System.
- **IDE**: Entorno de Desarrollo Integrado, por sus siglas en inglés Integrated Development Environment.
- **IFF**: Identificación Amigo o Enemigo, por sus siglas en inglés Identification Friend or Foe.
- **IP:** Protocolo de Internet, por sus siglas en inglés Internet Protocol.
- **PSR**: Radar Primario de Vigilancia, por sus siglás en Primary Surveillance Radar.
- **SACTA**: Sistema de Automatizado de Control del Tráfico Aéreo.
- **SSR**: Radar Secundario de Vigilancia, por sus siglas en inglés Secondary Surveillance Radar.

Notación:

Se ha escrito en cursiva todas aquellas palabras escritas en inglés. Además, de todo lo relacionado con el lenguaje de programación Java, sean clases, métodos, etc. Sin embargo, no se aplicará a nombre propios, siglas o acrónimos de entidades y organismos públicos o privados.

1. Introducción

Como estudiante de ingeniería aeroespacial, conociendo sus competencias y más en particular para mi especialidad de aeronavegación, el control del tráfico aéreo es de vital importancia. Es clara su relevancia, pues el esfuerzo actual de la comunidad aeronáutica se está centrando en crear las herramientas para mejorar la eficiencia del uso del espacio aéreo, esfuerzo que podríamos llamar implantación de la navegación PBN. Sabiendo que uno de los métodos usados es la mayor comunicación y automatización, entendiendo esto como un aumento del intercambio automático de información, lo que nos lleva a este trabajo fin de grado. El desarrollo de la tecnología en el campo de las comunicaciones e información (TIC), es lo que hace esto posible, por ello se ha creído de vital importancia no solo entender cómo se comunica una aeronave sino también profundizar en ello. Con este fin, se presenta un trabajo fin de grado que lleva al estudio de Java, un lenguaje de programación multiplataforma, algo más que necesario para cualquier ingeniero, a entender a un nivel profundo como se envía información entre aeronave-controlador y finalmente, ofrecer una herramienta que ayude al estudio del tráfico aéreo, que sea de uso totalmente libre y anime e impulse al desarrollo y mejora.

1.1. Motivación

Respecto a los motivos personales que han llevado concretamente a la elección de este trabajo, los podemos encontrar en asignaturas como Transporte, Navegación y Circulación Aérea, en la cual se tocó por primera vez el SACTA, un programa desarrollado por la colaboración de varios profesores, usado para la monitorización de vuelos. Dicho programa en lenguaje Java, despertó mi interés sobre la monitorización aérea. Quería entender cómo es posible que se pueda ver en tiempo real las aeronaves y sus parámetros, sea posición, velocidad, altitud, etc. Dicha curiosidad, se incrementó cuando este año en la asignatura de Ingeniería de los Sistemas de la Navegación Aérea, vimos cómo se comunicaban las aeronaves y entendí que aquel programa lo que hacía era leer los datos enviados por los transpondedores de las aeronaves, sistema llamado ADS-B. Pero, leer definiciones en unas diapositivas no es suficiente para comprender realmente su funcionamiento, es necesario meterse en materia.

Nota importante: aunque se llame al programa SACTA este es meramente una herramienta didáctica del grado de Ingeniería Aeroespacial. Nombre que realmente es propiedad de un sistema profesional de Indra, para ser exactos al Sistema Automatizado de Control de Tránsito Aéreo (SACTA)

1.2. Objetivos

Como objetivo general de nuestro trabajo tenemos:

- Desarrollo de una aplicación multiplataforma para la monitorización del tráfico aéreo basada en la plataforma OpenMap.

Como objetivos secundarios derivados del objetivo principal:

- Aprender a programar en un lenguaje multiplataforma, Java.
- Aprender a utilizar la plataforma OpenMap.
- Comprender la comunicación antena aeronave-antena receptor.
- Recolección y documentación de programas, código y todo lo relacionado con radares desarrollado hasta la fecha, en el contexto de las asignaturas del grado ingeniería aerospacial.
- Integración de OpenMap con el Design de Netbeans.

1.3. Estado del arte

En cuanto a la monitorización de vuelos, se podría decir que se encuentran 3 posturas, la profesional, es decir, aquellas herramientas usadas por los propios controladores aéreos, sea por ejemplo el SACTA de Indra. Las de uso público general como se mostrará en este apartado y las DIY (del inglés: hazlo tú mismo). Como usuarios interesados en el tema, nos planteamos estas dos últimas opciones. Ahora, como se verá en este apartado el problema de las de uso general a pesar de tener grandes herramientas no se podrá acceder a ellas a no ser que se pague. Por ello, el interés en tener una herramienta que nos proporcione esas posibilidades de manera gratuita.

Se quiere matizar algo importante, el departamento de aeronavegación del grado de ingeniería aeroespacial, dispone ya de un programa de monitorización, el SACTA. Sin embargo, este trabajo fin de grado pretende aportar una nueva herramienta, que desde un código sencillo y manejable asiente las bases para la creación de no solo una aplicación de monitorización del tráfico aéreo sino también la gestión del mismo, para que futuros alumnos o profesores puedan fácilmente construir y escalar sobre esta nueva base. Aquí veremos la importancia de usar la plataforma OpenMap, pues nos cederá las herramientas necesarias para que la nueva aplicación disponga de una verdadera interfaz gráfica de nivel y gran potencial, capaz de mostrar mapas con varias capas, cambiar entre proyecciones y todo a partir de herramientas predefinidas de código totalmente libre.

Una vez aclaro esto, dejando de lado las herramientas profesionales, introducimos el contexto de la aplicación que se ha creado respecto a las diferentes herramientas que existen en el mercado.

Es importante recalcar que existe una infinidad de páginas web y aplicaciones tanto para pc como para móviles dedicadas a la monitorización de vuelos.

Tenemos ejemplos como:

- **FlightRadar24:** es una de las páginas web más conocidas que muestra en tiempo real el tráfico aéreo mundial, indicando posición, velocidad, tipo de avión, historial de vuelos entre muchas otras herramientas. Sin embargo, una de las mayores pegas de esta web es que la información realmente útil con la que se pueda trabajar o estudiar el tráfico aéreo está solo disponible para usuarios de pago y en diferentes membresías.

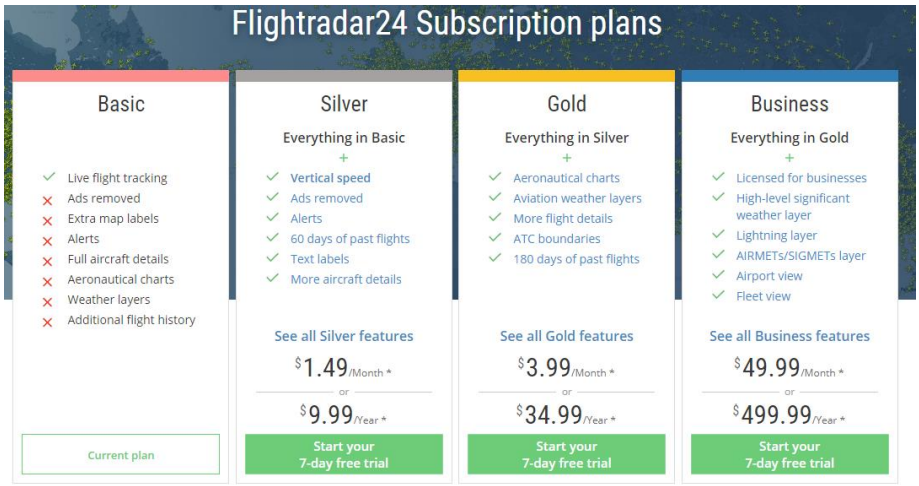

Figura 1: Membresías FlightRadar24 (propiedad de FlightRadar24)

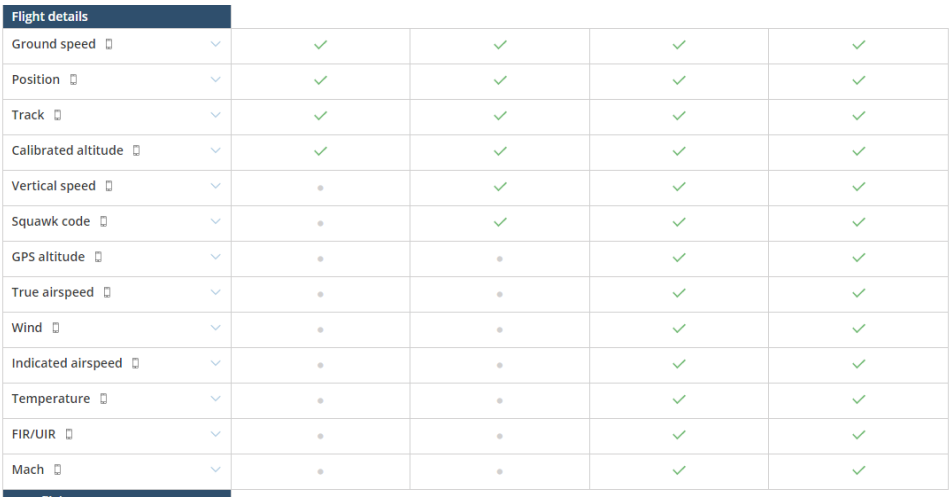

Figura 2: Diferencias entre membresías FlightRadar24 (propiedad de FlightRadar24)

Past flights				
Flight history	7 days	60 days	180 days	365 days
Aircraft history	7 days	60 days	180 days	365 days
Playback	7 days	60 days	180 days	365 days
Aircraft on ground	Last 60 minutes	Last 7 days	Last 30 days	Last 90 days
Download CSV/KML files	•	10 per month	100 per month	1,000 per month

Figura 3:Diferencias entre membresías FlightRadar24 (propiedad de FlightRadar24)

Weather Layers				
Volcanic eruption	✓	✓	✓	✓
Current weather	•	✓	✓	✓
Cloud	•	•	✓	✓
Precipitation	•	•	✓	✓
AIRMETs/SIGMETs	•	•	•	✓
High level significant weather	•	•	•	✓
Lightning	•	•	•	✓
Map Layers				
ATC boundaries	•	•	✓	✓
Oceanic tracks	•	•	✓	✓
Aeronautical Charts	•	•	✓	✓

Figura 4: Diferencias entre membresías FlightRadar24 (propiedad de FlightRadar24)

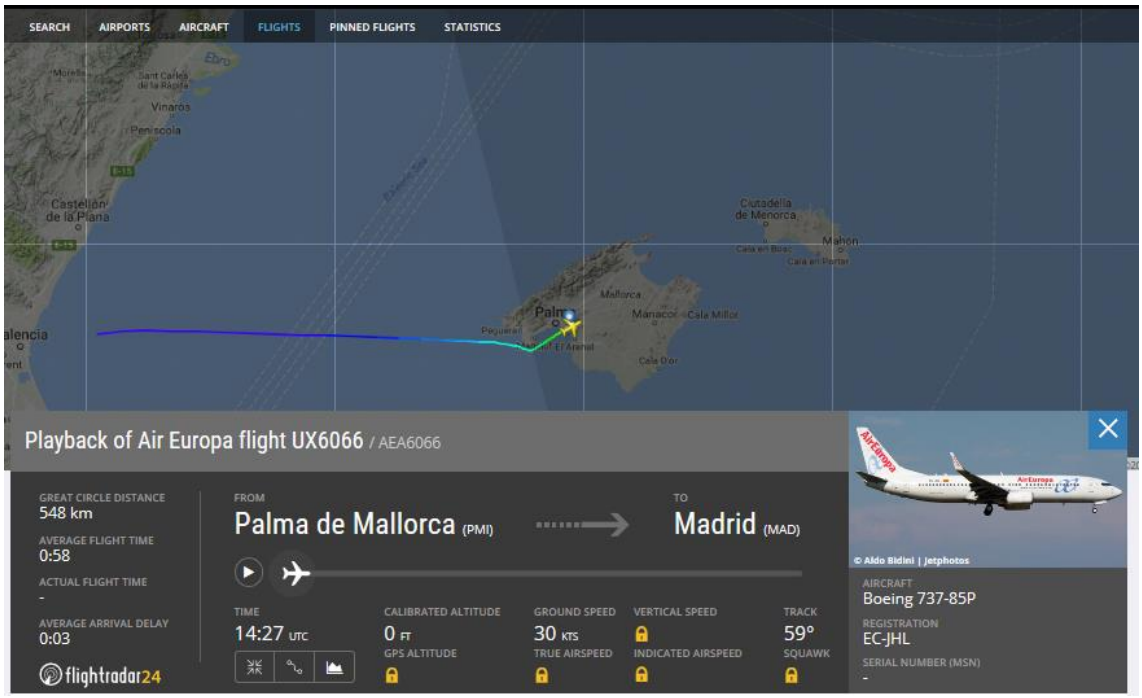

Figura 5: Interfaz FlightRadar24 para el seguimiento de un vuelo (propiedad FlightRadar24)

Como se puede apreciar en las Figuras 2,3 y 4 la información realmente útil para el estudio como *True Airspeed*, *Indicated Airspeed*, sectores ATC o la descarga misma de los datos en formato CSV o KML está solo disponible para miembros. Por ello, pese a ser una herramienta muy potente y completa se ve realmente limitada en su versión gratuita.

- **FlightAware**: según la propia compañía es la más grande del mundo en cuanto a rastreo de aviones a nivel global y provee servicios del mismo a más de 10.000 operadores y empresas de servicios aeronáuticos. Al igual que FlightRadar24, se basa en la lectura de datos ADS-B de proveedores de envergadura, entre ellos ARINC, SITA, Satcom Direct, Garmin, Honeywell GDC y UVdatalink, así como más de 150 estaciones propias.

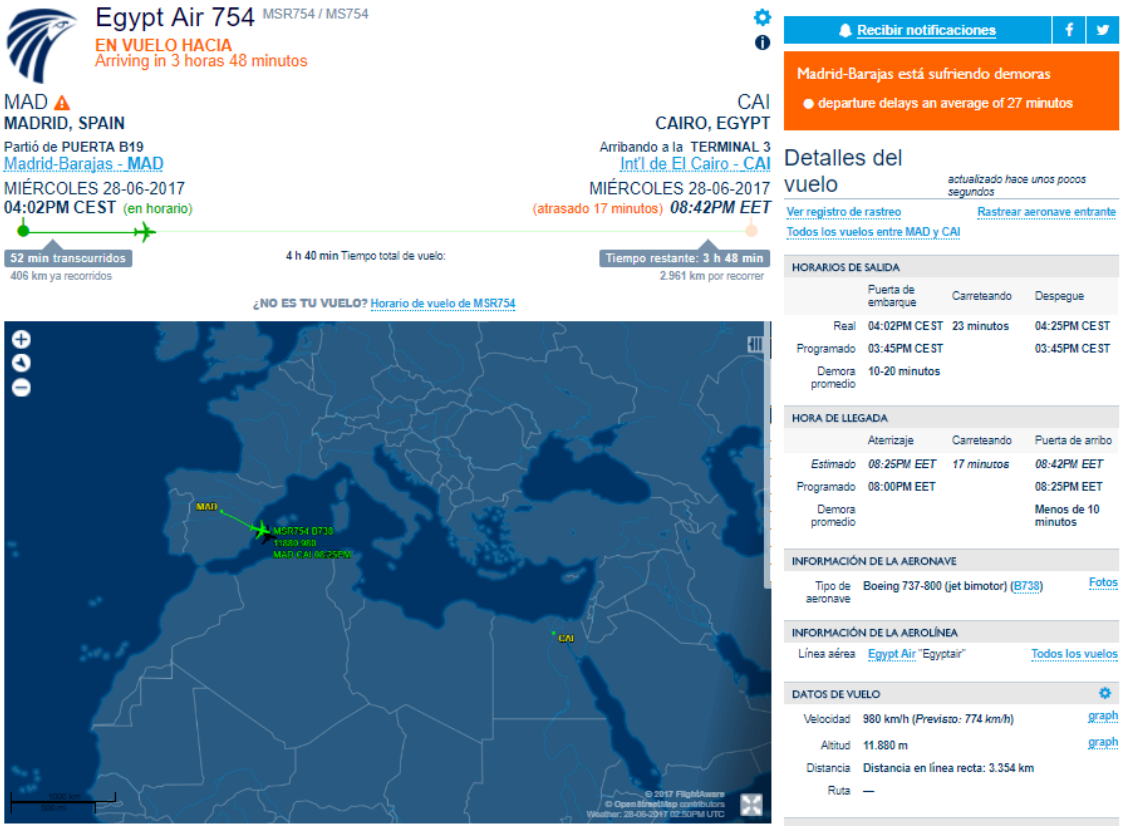

Figura 6: Interfaz FlightAware para el seguimiento de un vuelo (propiedad FlightAware)

Estas dos grandes empresas ofrecen membresía gratuita para aquellos usuarios que proporcionen datos o mejor dicho cobertura a su red. Para el caso de FlightRadar24, ellos mismos te envían un set completo totalmente gratis

para que puedas hacerlo y en el caso de FlightAware si dispones de una antena te ofrecen el software gratuito o incluso tienen tutoriales para que puedas crear tu propia antena con una Rasberry Pi.

- **PlaneFinder**: es una plataforma no tan conocida, pero con grandes aptitudes. Al igual que las dos anteriores ofrece cobertura a nivel global, pero a diferencia de ellos, gran parte de la información que encontrábamos solo mediante membresía aquí es gratuita. Por ejemplo, el hecho de poder descargar histórico en formato KML, aunque hay que tener ciertos conocimientos para ello. Se proporciona la siguiente página web, en la cual se explica el procedimiento para poder llevarlo a cabo.

  https://www.metabunk.org/converting-planefinder-net-tracks-to-google-earth-kml.t8312/

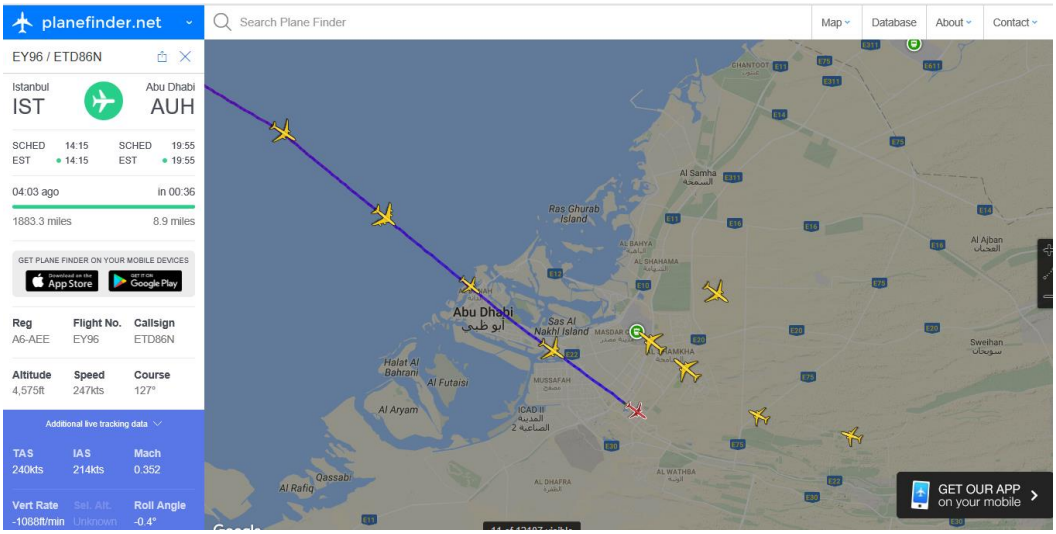

Figura 7:Interfaz PlaneFinder para el seguimiento de un vuelo. (propiedad PlaneFinder).

Además, estas tres grandes compañías disponen de aplicaciones móviles para iOS y Android.

Como se ha visto, hay herramientas muy potentes actualmente para la monitorización del tráfico aéreo. En nuestro caso particular, en el desarrollo del grado se ha visto y tocado la plataforma FlightRadar24 para diversos trabajos sobre todo de la Asignatura de Gestión del Espacio Aéreo.

1.4. Materiales y métodos

Para el desarrollo de este proyecto, ha sido necesario estudiar y conocer los materiales y métodos que aquí presentamos. En primer lugar, como todo programa necesita de un lenguaje de programación, en este caso **Java** y de un entorno, **Netbeans**. De unas librerías especiales para poder crear interfaces complejas desde un código sencillo, **OpenMap**. En segundo lugar, la asignatura de master **Asignatura Sistemas de Gestión de Vuelo por Computador**, aporta recursos para comprender como funciona un sistema de monitorización o incluso crearlo y finalmente, **KML**, por su importancia en cuanto a nivel de uso en archivos de intercambio de información relacionados con posicionamiento, rutas, georreferencias etc. De hecho, como se verá más adelante, incluso se creará una capa de rutas a partir de información almacenada en archivo kml.

Java: es un lenguaje de programación orientado a objetos y hoy en día es uno de los lenguajes más usado en todo el mundo. Se fundamenta en cinco pilares: la misma orientación a objetos, la posibilidad de ejecutar un mismo programa en diversos sistemas operativos, la inclusión por defecto de soporte para trabajo en red, la opción de ejecutar el código en sistemas remotos de manera segura y la facilidad de uso. Por todo esto, se ha elegido este lenguaje, así nuestra aplicación podrá ser fácilmente modificada, usar los objetos ya creados, ejecutarla en cualquier sistema operativo y en general facilidad de evolución. Además, como motivo personal, se considera imprescindible este lenguaje, bajo la creencia de que para cualquier ingeniero debe poder crear sus propias herramientas y usarlas allá donde vaya.

Netbeans: es un entorno de desarrollo integrado (IDE) que permite el uso de un amplio rango de tecnologías de desarrollo tanto para escritorio, como aplicaciones Web, o para dispositivos móviles. Da soporte a lenguajes como Java, PHP, Groovy, C/C++, HTML5… Además, de que puede ser instalado en múltiples sistemas operativos sea Mac OS, Windows o Linux. La principal diferente entre este IDE y otros existentes para Java es su mayor potencia y funcionalidad, así como se verá más adelante, dispone de un asistente para la creación de interfaces gráficas.

OpenMap: es un conjunto de herramientas basado en Java Beans para crear aplicaciones y applets que necesitan de información geográfica. En concreto, OpenMap es un conjunto de componentes Swing que entienden coordenadas geográficas. Estos componentes ayudan a mostrar datos de o sobre mapas, además de permitir manejar y modificar datos entrantes del usuario.

Asignatura Sistemas de Gestión de Vuelo por Computador: el aprendizaje de Java mediante sus diapositivas, ejercicios de programación y recursos varios. Además, esta asignatura guía a sus alumnos por medio de prácticas a la creación de una sencilla

aplicación de monitorización. Dichas prácticas serán ejemplificadas más adelante, siendo una buena manera de introducir conceptos importantes desde ejemplos muy sencillos.

KML: es un lenguaje de marcas (*markup*) derivado de XML. Los lenguajes de marcado son un formato de datos universal para almacenamiento e intercambio de información cuya estructuración se basa en etiquetar (*tag-based*) cada información con el tipo de datos al que pertenece.

Nota: en cuanto al aprendizaje de Java, también ha sido necesaria formación complementaria que se ha llevado a cabo mediante diferentes tutoriales, cuyos enlaces se encuentran en el apartado Java de la bibliografía.

2. Monitorización de vuelos

2.1. Introducción al radar

Radar es un acrónimo procedente del inglés *radio detection and ranging*, detección y medición de distancias por radio, sistema de origen militar anterior a la Segunda Guerra Mundial, originalmente usada para la detección y medida de distancias. Sin embargo, actualmente se pueden obtener datos como la posición, velocidad, aceleración, forma, etc. Por la parte civil, se encuentran sistemas de radionavegación o teledetección y por la parte militar encontramos seguimiento de blancos, IFF, guiado de misiles…

Expliquemos un poco el contexto de nuestro radar y sus semejantes. Los radares se pueden clasificar en dependientes e independientes. Con ello entendemos que necesite o no un sistema exterior (GPS principalmente). Seguidamente, se subclasifica en cooperativo o no cooperativa en función de la necesidad de que la aeronave realice alguna actividad de forma activa. Así, por ejemplo, el radar primario (PSR) es independiente, pues no necesita de ninguna constelación de satélites y no cooperativo al no depender de ninguna acción por parte de la aeronave. Este radar pese al tiempo que ha pasado desde su invención y a sus menores capacidades en comparación con más actuales radares, se sigue manteniendo por su característica única de no depender en absoluto del móvil a detectar, pues su funcionamiento es sencillo, envía una señal de alta potencia que es reflejada por la superficie del fuselaje de la aeronave y con ello, mediante los sistemas y algoritmos pertinentes es procesado y detectado.

Como ejemplo de independiente cooperativo tenemos al radar secundario SSR y su evolución el modo S, la aeronave contesta de forma cooperativa, pero no necesitamos de un sistema GNSS para ubicar la aeronave desde tierra. Finalmente, como dependiente cooperativo, el sistema ADS-B, necesita tanto que la aeronave coopere como que exista y funcione un sistema GNSS para que la aeronave se ubique y lo retransmita.

Comentar que, dentro de los sistemas independientes cooperativos, también tenemos la MLAT (multilateración) donde un conjunto de estaciones receptoras distribuidas por una determinada zona, en general sincronizadas, recibe una señal emitida desde un móvil. Los sistemas de MLAT por sus prestaciones y coste, están llamados a reemplazar al SSR Modo S en 2020.

Figura 8: Tipos de sistemas de vigilancia-radares (Obtenido de ISNA)

Nota: ISNA son las siglas de la asignatura Ingeniería de los Sistemas de Navegación Aérea.

2.2. BaseStation SBS

A la hora de desarrollar un radar, lo primero que tenemos que plantearnos es obviamente como obtendremos los datos, es decir, cómo podremos saber que aviones pasan por encima nuestro. Para ello, la UPV dispone de una antena capaz de recibir los datos ADS-B. Para ser más exactos dispone de una SBS BaseStation.

Usando la aplicación Telnet un usuario se puede conectar a través del puerto 30002 a la unidad SBS y ver la información recibida.

La información se verá:

STA,,5,179,400AE7,10103,2008/11/28,14:58:51.153,2008/11/28,14:58:51.153,RM

MSG,4,5,211,4CA2D6,10057,2008/11/28,14:53:49.986,2008/11/28,14:58:51.153,,,408.3,146.4,,64,,,,,

MSG,8,5,211,4CA2D6,10057,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,,,,,,,,0

MSG,4,5,211,4CA2D6,10057,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,408.3,146.4,,,64,,,,,

MSG,3,5,211,4CA2D6,10057,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,37000,,,51.45735,-1.02826,,,0,0,0,0

MSG,8,5,812,ABBEE3,10095,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,,,,,,,,,,0

MSG,3,5,276,4010E9,10088,2008/11/28,14:53:49.986,2008/11/28,14:58:51.153,,28000,,,53.02551,-2.91389,,,0,0,0,0

MSG,4,5,276,4010E9,10088,2008/11/28,14:53:50.188,2008/11/28,14:58:51.153,,,459.4,20.2,,,64,,,,,

MSG,8,5,276,4010E9,10088,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,,,,,,,,,,0

MSG,3,5,276,4010E9,10088,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,28000,,,53.02677,-2.91310,,,0,0,0,0

MSG,4,5,769,4CA2CB,10061,2008/11/28,14:53:50.188,2008/11/28,14:58:51.153,,,367.7,138.6,,,-2432,,,,,

MSG,8,5,769,4CA2CB,10061,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,,,,,,,,,0

Hay seis tipos de mensajes: MSG, SEL, ID, AIR, STA, CLK. Como se verá más abajo, la mayoría de la información sobre la aeronave se encuentra en el tipo MSG.

ID	Type	Description
SEL	SELECTION CHANGE MESSAGE	Generated when the user changes the selected aircraft in BaseStation.
ID	NEW ID MESSAGE	Generated when an aircraft being tracked sets or changes its callsign.
AIR	NEW AIRCRAFT MESSAGE	Generated when the SBS picks up a signal for an aircraft that it isn't currently tracking.
STA	STATUS CHANGE MESSAGE	Generated when an aircraft's status changes according to the time-out values in the Data Settings menu.
CLK	CLICK MESSAGE	Generated when the user double-clicks (or presses return) on an aircraft (i.e. to bring up the aircraft details window).
MSG	TRANSMISSION MESSAGE	Generated by the aircraft. There are eight different MSG types.

Tabla 1: Tipos de mensajes (Obtenido de woodair.net)

Dentro de la categoría de mensajes de transmisión (MSG) encontramos ocho tipos:

ID	Type		Description
MSG,1	ES Identification and Category	DF17 BDS 0,8	
MSG,2	ES Surface Position Message	DF17 BDS 0,6	Triggered by nose gear squat switch.
MSG,3	ES Airborne Position Message	DF17 BDS 0,5	
MSG,4	ES Airborne Velocity Message	DF17 BDS 0,9	
MSG,5	Surveillance Alt Message	DF4, DF20	Triggered by ground radar. Not CRC secured. MSG,5 will only be output if the aircraft has previously sent a MSG,1, 2, 3, 4 or 8 signal.
MSG,6	Surveillance ID Message	DF5, DF21	Triggered by ground radar. Not CRC secured. MSG,6 will only be output if the aircraft has previously sent a MSG,1, 2, 3, 4 or 8 signal.
MSG,7	Air To Air Message	DF16	Triggered from TCAS. MSG,7 is now included in the SBS socket output.
MSG,8	All Call Reply	DF11	Broadcast but also triggered by ground radar

Tabla 2: Tipos de mensajes de transmisión (MSG) (Obtenido de woodair.net)

Ahora bien, analicémoslo con mayor profundidad. Para cada mensaje tenemos hasta 22 campos separados por comas. Son los siguientes:

Field 1:	Message type	(MSG, STA, ID, AIR, SEL or CLK)
Field 2:	Transmission Type	MSG sub types 1 to 8. Not used by other message types.
Field 3:	Session ID	Database Session record number
Field 4:	AircraftID	Database Aircraft record number
Field 5:	HexIdent	Aircraft Mode S hexadecimal code
Field 6:	FlightID	Database Flight record number
Field 7:	Date message generated	As it says
Field 8:	Time message generated	As it says
Field 9:	Date message logged	As it says
Field 10:	Time message logged	As it says
Field 11:	Callsign	An eight digit flight ID - can be flight number or registration (or even nothing).
Field 12:	Altitude	Mode C altitude. Height relative to 1013.2mb (Flight Level). Not height AMSL..

Field 13:	**GroundSpeed**	Speed over ground (not indicated airspeed)
Field 14:	**Track**	Track of aircraft (not heading). Derived from the velocity E/W and velocity N/S
Field 15:	**Latitude**	North and East positive. South and West negative.
Field 16:	**Longitude**	North and East positive. South and West negative.
Field 17:	**VerticalRate**	64ft resolution
Field 18:	**Squawk**	Assigned Mode A squawk code.
Field 19:	**Alert (Squawk change)**	Flag to indicate squawk has changed.
Field 20:	**Emergency**	Flag to indicate emergency code has been set
Field 21:	**SPI (Ident)**	Flag to indicate transponder Ident has been activated.
Field 22:	**IsOnGround**	Flag to indicate ground squat switch is active

Tabla 3: Todos los campos posibles y su contenido (Obtenido de woodair.net)

Los diez primeros campos contienen información básica y estándar para todos los mensajes y del campo 11 al 22 se corresponde con información específica de la aeronave.

Para hacerlo más visual:

	1	2	3	4	5	6	7	8	9	10		11	12	13	14	15	16	17	18	19	20	21	22
MSG 1	MT	TT	SID	AID	Hex	FID	DMG	TMG	DML	TML		CS											
MSG 2													Alt	GS	Trk	Lat	Lng						Gnd
MSG 3													Alt			Lat	LNG			Alrt	Emer	SPI	Gnd
MSG 4														GS	Trk			VR					
MSG 5													Alt							Alrt		SPI	Gnd
MSG 6													Alt						Sq	Alrt	Emer	SPI	Gnd
MSG 7													Alt										Gnd
MSG 8																							Gnd
SEL												CS											
ID												CS											
AIR																							
STA																							
CLK				-1		-1																	

Tabla 4:Visualización tipo de mensajes-campos (Obtenido de woodair.net)

Representado en verde, los campos donde se envía información, en gris aquellos que tienen información nula, vacía y en negro los que directamente no se usan.

Un par de comentarios relevantes a tener en cuenta:

- La salida del socket emite un -1 para true y un 0 para false. Ninguno de los dos significa que no esté siendo utilizado.

- El *Callsign* (campo 11), tiene 8 caracteres (6 bits en ASCII). Como esta codificado en ASCII, un nulo aparecerá como '@'. Para entenderlo mejor, si el piloto introduce por ejemplo 8 espacio en el *cockpit*, lo que se mostrará en la BaseStation será 8 arrobas ('@@@@@@@@').

Interpolación

Hay cientos campos de información que solo pertenecen a un tipo de MSG, por tanto, no podemos obtener toda la información en un solo mensaje. Por ejemplo, el *Callsign* se encuentra únicamente en el tipo MSG 1, el *VertRate* solo en el MSG 4 y el *Squawk* solo en el MSG 6.

De esta manera, para obtener los 11 campos de información de una aeronave, se requiere de al menos de la recepción de cuatro mensajes, más específicamente el MSG1, MSG 3, MSG 4 y MSG 6. Téngase en cuenta que el MSG 6, solo es enviado si previamente se ha interrogado desde una estación de tierra a la aeronave, por tanto, si ningún radar de tierra está dentro de la cobertura de la aeronave, el MS6 no será enviado. En otras palabras, el *Squawk* solo será visto en la pantalla de un usuario SBS si algún radar en Modo S interroga a una aeronave.

A través de la aplicación *telnet* de Windows podemos fácilmente conectarnos a la antena y visualizar estos paquetes de información escribiendo desde la consola de Windows: telnet servantena.etsid.upv.es 30002

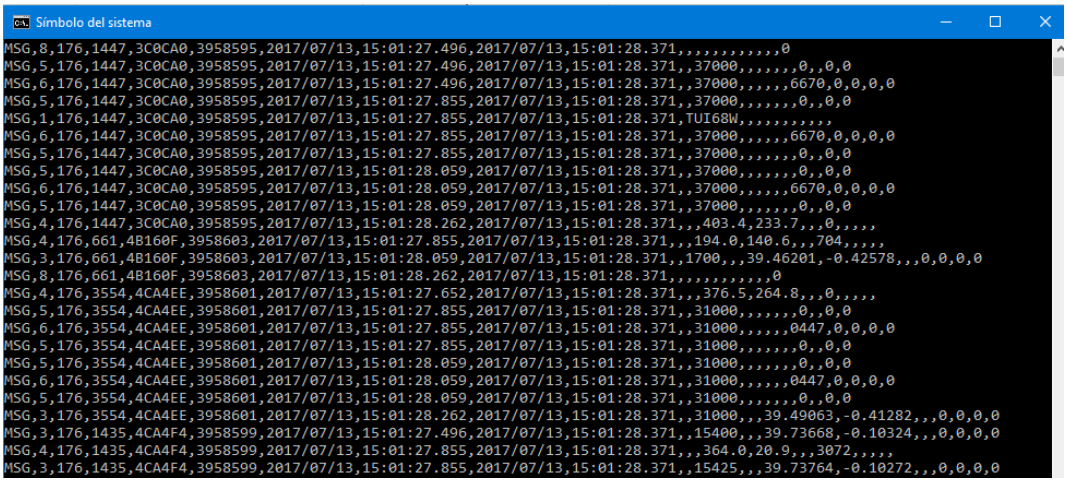

Figura 9: aplicación telnet para conectarse a BaseStation SBS

3. Conceptos fundamentales de Java

Es conveniente repasar una serie de conceptos de Java que se aplicarán en este trabajo. Evidentemente, no se explicará ninguno de ellos en gran profundidad, pues los objetos de Java como los que aquí se expondrán presentan una herencia de gran tamaño. De hecho, un trabajo fin de grado podría dedicarse enteramente al estudio de clases profundas de Java. Sin embargo, con solo unos ápices de sus funciones podremos hacer grandes cosas.

Primeramente, lo más básico. En Java debemos hablar de **clases** y **objetos**. Una clase es una abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener. Un objeto podría entenderse como una instancia de una clase, una entidad que tiene unas propiedades, pero propias de sí mismo y unas operaciones disponibles específicas (métodos). Con lo que se quiere decir que las clases son abstracciones generales y un objeto es un tipo de abstracción particular de una clase. Sea por ejemplo una clase, vehículos que deba tener propiedades y método propios de un coche como un motor, puertas, arrancar, parar etc. Y por objeto tendríamos un BMW, con un motor específico, un número de puertas, y métodos propios de como arranca o para.

Hablemos ahora de un concepto ya introducido, la **herencia**. En Java una de los pilares es la herencia, antes habíamos hablado de una clase llama coches, pero esta podría ser una clase que herede de vehículos de motor y esta a su vez de vehículos. Cuando una clase hereda de otra, adquiere todas las propiedades y métodos de esta, más los propios que se creen en ella. Así, coches al heredar de vehículos de motor podría ya tener métodos definidos para arrancar y parar o propiedades como motor, lo que nos ahorraría trabajo pues simplemente los heredamos y ya podemos disponer de ellos. Con la herencia Java provee de un sinfín de herramientas ya creadas de las cuales podemos heredar sus herramientas para nuestro beneficio.

Otro concepto importante de Java son las **interfaces**. Lo explicaremos con un ejemplo. Pensemos por ejemplo en la clase coches, todos los coches deben tener una serie de características necesarias, sea por ejemplo cambiar ruedas, revisar espejos entre otras acciones que podríamos llamar acciones de carretera cuando definimos una clase y esta implemente una interfaz, esta clase estará obligada a tener todos los métodos y propiedades que se hayan definido en la interfaz, es una manera de tener un control sobre la programación. Imaginemos ahora que vamos a crear una clase llamada motos, esta heredará de vehículo de motor y por tanto tendrá ya sus acciones y propiedades. Sin embargo, habrá ciertos métodos o características que tendría que tener y que no deberíamos olvidar al hablar de una moto, por ello haríamos que moto implemente acciones de carretera de esta manera estará obligada a instanciar y definir todas las características de esta interfaz. Por ello, las interfaces son muy importantes, pues una clase solo puede heredar de otra, pero puede implementar varias interfaces.

Cuando creamos un programa tal vez queramos realizar cálculos con datos que están continuamente entrando e imprimirlos en una gráfica, al mismo tiempo leer un archivo diferente e imprimirlos en otra. Sin embargo, mientras nuestro programa ejecuta la primera acción no podrá ejecutar la segunda, por tanto, no podríamos ver ambas gráficas

al mismo tiempo, pues o se está creando una o se está creando la otra. Para ello Java dispone de ***Threads*** (**hilos**), esto nos permite dividir la capacidad de computación de nuestro ordenador para realizar varias tareas al mismo tiempo. Mencionar también que hay dos clases de *Threads* los independientes y los sincronizados. Los independientes simplemente se ejecutarán y procesaran la información sin tener nada en cuenta, los sincronizados poseen métodos para que los diferentes hilos que tengamos se ejecuten sincronizando sus tareas entre sí, por ejemplo si ambos están realizando cálculos, pero el primer *Thread* ya ha acabado no pasará a su segunda tarea hasta que el segundo *Thread* también termine, o por ejemplo puede que el segundo *Thread* necesite algo de lo calculado en el primero si este acaba antes, no cogería el valor sino esperaría a que el *Thread* 1, le notificara que ya puede tomarlo.

Continuamos con un concepto abstracto, ***Sockets***. Representa un mecanismo para el intercambio de información entre dos programas, que no tienen porqué estar en el mimo ordenador, puede estar en cualquier parte. Un socket queda definido por un par de direcciones IP local y remota, un protocolo de transporte y un par de números de puerto local y remoto. Hay varios tipos de *Socket*, nosotros usaremos ***TCP (Transport Control Protocol)***. Para este tipo se distinguen dos partes, el socket del servidor y el del cliente. Por parte del *socket* del servidor, indicará por qué puerto se harán las escuchas y esperará a la llamada de un cliente para aceptar la conexión. Por la parte del socket del cliente indicará donde se encuentra y por qué puerto quiere conectarse.

Algo muy relacionado son los conceptos de ***Listener*** y **Generator**. Hablamos de *Listener* cuando digamos que el objeto en cuestión está a la espera de recibir algo de alguien, normalmente será del *Generator*, pues este concepto se usa para aquello que crea información, datos o algún evento que será escuchado por un *Listener*. Esto es importante conocerlo por el hecho de que se programa en inglés, tanto el propio lenguaje Java como todo lo que se ha creado está en dicho idioma y es importante entender, ciertas palabras de la que más adelante se irán introduciendo.

Otro concepto a conocer son los ***Events***, del inglés eventos. De nuevo muy relacionado con lo anterior, pues se podría entender como el medio de comunicación entre *Listener* y *Generator*. Supongamos que se produce algún cambio que un *Generator* deba comunicar a un *Listener*, es precisamente a este cambio que debe comunicarse al que se conoce por un evento (*Event*).

GUI por sus siglas en inglés graphical user interface, es decir, interfaz gráfica del usuario. A lo largo de este documento se mostrarán diferentes *GUIs* de las aplicaciones creadas. Siguiendo este camino es importante, hablar de *paintComponent* y *repaint*. Cuando creamos una interfaz gráfica, usamos *paintComponent* para imprimir en pantalla objetos sea por ejemplo aviones, pero tenemos que tener en cuenta que, si dicho avión se está continuamente moviendo, deberemos continuamente volver a pintarla en la ubicación correcta. En Java, la manera en que se juega es la siguiente. Cada vez que se produzca un cambio en este caso, en la posición de la aeronave, nuestro programa llamaría a *repaint* que cuando esté listo llamará a *paintComponent* y volverá a imprimir la aeronave en la nueva posición. Sentado esto, hablemos de *Swing*. Básicamente, es una gran librería de

Java par la creación de *GUIs*, con multitud de componentes que nos permiten crear botones, menús, diálogos, etc.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Figura 10: componente Swing de Java

Otro elemento, importante es ***Graphics***, el cual hace posible dibujar simples geometrías, imprimir texto o posicionar imágenes dentro de los límites de un componente, sea por ejemplo un ***Frame*** o ***Panel***. Se entiende por ambos como contenedores de componentes gráficos, podría hablarse de capas. La diferencia radica en que el *Frame* es la capa más profunda, así un *Frame* puede contener un Panel o varios y no lo opuesto.

Relacionamos un poco los últimos conceptos. *Graphics* aparte de tener métodos para dibujar figuras simples, da una especie de marco, a aquello gráfico que se crea. Información como su color de fondo, localización, dimensiones. Esto es imprescindible para poder imprimirlo en pantalla, es decir, todo aquello que queramos imprimir debe tener un contexto. De hecho, *paintComponent* recibe como atributo un objeto *Graphics*. Finalmente, mencionar que existe ***Graphics2D*** que desciende de *Graphics* y proporciona un mayor control sobre las geometrías, transformaciones, manejo de colores, manejo de texto, etc.

Para terminar, hablar de una estructura de datos, **HashMap**. Es uno de los objetos más utilizados en java, implementa la interfaz *Map*. No es más que un conjunto de *key-value*. En este clase podemos guardar cualquier tipo de objeto, asignándole una *key* para más tarde poder recuperar ese objeto mediante la *key*.

En este apartado número 3 se han visto varios conceptos de Java importantes de entender. No han sido elegidos al azar, puesto que pueden parecer conceptos muy dispersos, pero son todos absolutamente necesarios para la comprensión de las aplicaciones que se desarrollarán más adelantes.

4. La plataforma OpenMap

Ahora hablaremos del gran salto y el motivo de este trabajo fin de grado. La integración de la plataforma OpenMap para nuestro radar. En este apartado se afrontará primeramente una introducción a OpenMap, seguidamente de un resumen donde se presenta una visión rápida y global de cómo funciona esta plataforma y continuará con un desarrollo de los aspectos o funcionalidades más importantes de la misma.

4.1. Introducción a OpenMap

Presentamos la plataforma OpenMap. Dispone de una página web (openmap-java.org) de vital importancia no solo para la obtención de los recursos necesarios como librerías, sino también para tutoriales o guías, soporte de la comunidad mediante foros, su propia API, etc.

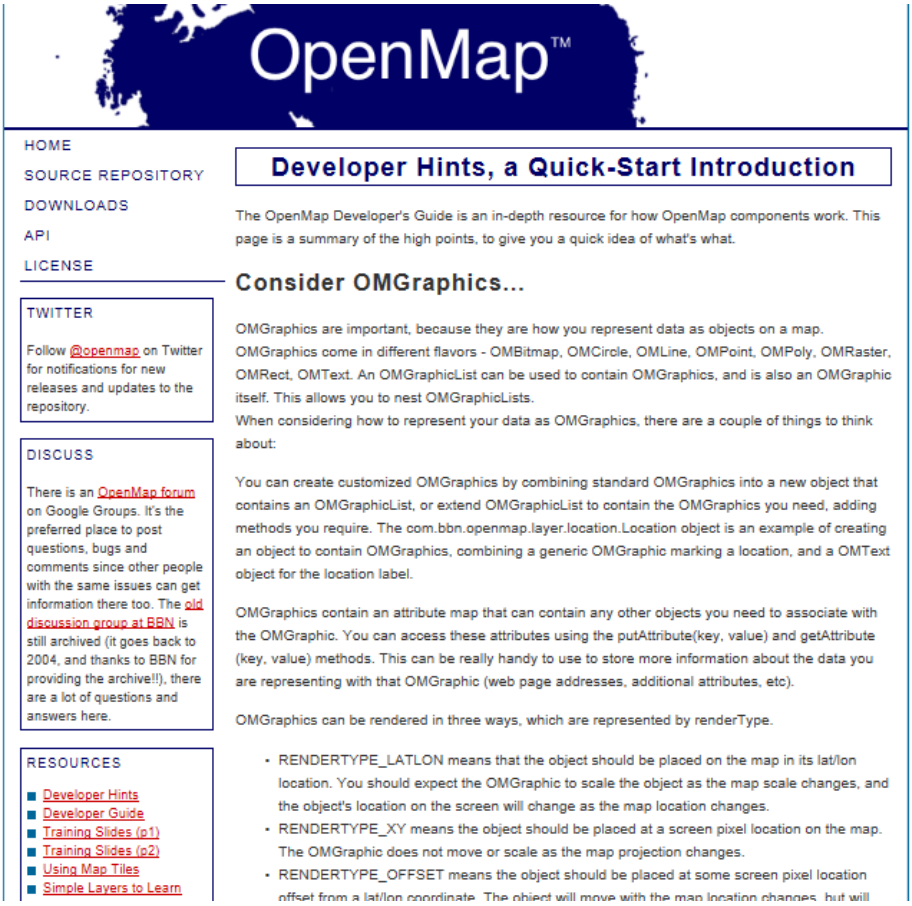

Figura 11:Home de la página web de OpenMap

OpenMap es un conjunto de herramientas destinadas a la visualización geoespacial para el lenguaje Java. Proporcionado como código abierto de BBN Technologies, ayuda a ver y entender información imprimiendo nuevas relaciones o tendencias que, de otra manera, podrían estar escondidas bajo pesados volúmenes de información, o creando nuevos datos a partir de la combinación de la información propia y otras fuentes y mostrar dichas relaciones.

OpenMap es capaz de representar información de mapas de archivos locales y servidores remotos, en una gran variedad de formatos. Además, puede ser ejecutado como aplicación, applet, componente dentro de otra aplicación o dentro de un *servlet* para proveer de mapas a un servidor web.

Como se ha visto, se puede usar en una gran variedad de formas y arquitecturas, pero para este trabajo nos centremos en su uso como aplicación.

4.2. Aplicación OpenMap

OpenMap crea el marco de referencia (*framework*) básico para configurar y ejecutar una aplicación de mapa. Este marco es un contenedor para todos los componentes que añadamos a la aplicación.

Este marco provee de las herramientas para que los componentes se enlacen o conecten entre ellos y así fácilmente formar la base de nuestra aplicación con diferentes interfaces y funcionalidades.

Respecto a las capas en OpenMap pueden usar información o datos de diferentes maneras cuando se trata del entorno de una aplicación. Se pueden crear o añadir características a mapas:

- Por computación.
- Desde la lectura de archivos de datos directamente desde un disco duro local.
- De leer archivos de datos desde una URL.
- De la lectura de archivos de datos contenidos en un archivo jar.
- Uso de información recuperada de una base de datos (JDBC).
- Uso de la información recibida de un servidor (imágenes u objetos de mapa)

4.3. Uso general de las proyecciones

Una de las ideas de ser de OpenMap es que el usuario no tenga que preocuparse por la localización de componentes en un mapa.

Entendemos por proyección en el contexto geoespacial como la translación entre un modelo esférico de la tierra a uno plano (de superficie expresada en dos dimensiones). En función del estudio, una proyección u otra resultará más ventajosa, ya sea, visualmente o por los cálculos a realizar. En el contexto de OpenMap cuando hablemos de proyecciones entenderemos componentes que se encargan de la translación de coordenadas en latitud y longitud a ubicaciones en pantalla.

Las proyecciones de OpenMap solo trabajan de latitudes/longitudes en grados decimales a coordenadas x/y. También a tener en cuenta, que esta plataforma usa como datum el WGS84.

Las proyecciones de OpenMap no solo sirven para pasar coordenadas de un sistema a otro, sino las formas (cuerpos) de los mapas se definen como pares de coordenadas de puntos, conocidos como vectores característicos.

Las proyecciones son capaces de determinar cómo los vectores característicos, definidos en términos de longitud y latitud, se representan así mismos. La geometría de estas formas (cuerpos) depende del tipo de línea elegido o especificado. OpenMap maneja tres tipos de líneas:

- *Great Circle lines* (línea ortodrómica): la línea entre dos coordenadas es la distancia más corta entre estos puntos en la superficie de la Tierra.
- *Rhumb lines* (línea loxodrómica): la línea entre dos coordenadas tiene un rumbo constante. Mientras se mueva sobre esta línea de un punto a otro, se mantendrá la misma dirección.
- *Straight* (recto). Las líneas entre coordenadas dibujadas en el espacio de píxeles

4.4. Arquitectura de los componentes

4.4.1. MapBean, MapHandler y MapPanel

MapBean es de vital importancia en OpenMap. Deriva de *java.awt.Container* y contiene *com.bbn.openmap.proj.Projection*, objeto que define la posición geográfica en el mapa representado por el propio *MapBean*. La proyección se define mediante una combinación del tipo de

proyección, la latitud y longitud del centro de la ventana, el factor de escala de la proyección, la altura y anchura de la ventana en pixeles.

En cuanto a las capas (*Layers*), derivan del *java.awt.Component* y son el único tipo de objeto que puede ser añadido a un *MapBean*. De esta manera, entendemos que las capas son componentes dentro de un contenedor *MapBean*. La representación de estas capas dentro de un contenedor *MapBean* se lleva a cabo por medio del mecanismo de interpretación (representación) de Java. Las capas son independientes una de otras, es decir, trabajan independientemente y evitan que *MapBean* tenga que estar pendiente de cada capa.

Recopilando, tenemos en cuenta que las capas también son *ProjectionListeners* por lo tanto, cuando son añadidas a *MapBean*, estarán atentas a cualquier evento que pueda producirse en el mapa (movimientos, zoom o reescalar).

Destacada la importancia de *MapBean*, ahora introducimos *MapHandler* (*com.bbn.openmap.MapHandler*) columna vertebral de OpenMap. Entender el funcionamiento de *MapHandler* es de vital para la personalización de las aplicaciones de OpenMap.

MapHandler es como un contenedor al que se le pueden añadir o quitar objetos de él. Pertenece a Java *BeanContext* (*java.beans.beancontext .BeanContext*). *MapHandler* es usado por aquellos componentes que necesiten manejar otros objetos o servicios.

La interfaz *MapPanel* es un componente que contiene un *MapBean*, *MapHandler* y una serie de menús que puede considerarse como una *JMenuBar* o *JMenu* con sub-menús.

4.5. Imprimiendo información con OMGraphics

OMGraphics son las clases principales usadas para representar objetos en un mapa. Las capas manejan a *OMGraphics* como objetos y se encargan de sus proyecciones, es decir, de donde deberían localizarse en el mapa.

4.5.1. Clases OMGraphics, tipos Render, tipos Line

Existe una gran variedad de *OMGraphics* en función de la forma que se debe representar en el mapa: *OMArc*, *OMCircle*, *OMGrid*, *OMLine*, *OMPoint*, *OMPoly* (incluyendo las subclases *OMDistance* y *OMSpline*),

OMRasterObject (incluyendo las subclases *OMBitmap*, *OMRaster*, *OMScalingRaster* y *OMScalingIcon*) *OMRect* y *OMText*.

Estos objetos se pueden representar de 3 maneras:

- *RenderType_Location*: el objeto se introducirá en sus coordenadas de latitud y longitud. Se debería esperar que el objeto se escale automáticamente cuando la escala del mapa cambie.

- *RenderType_XY*: el objeto se introducirá en una localización dada en función de los pixeles de la pantalla. El objeto no se moverá o escalará si la proyección del mapa cambia.

- *RenderType_Offser*: el objeto se introducirá en una localización dada según pixeles de pantalla en función de un offset de una coordenada en *lat/lon*. El objeto se moverá si el mapa cambia, pero no se rescalará si la escala varia.

Hay tres tipos de líneas asociadas a *OMGraphics* que tienen representación en lat/lon:

- *LineType_Straight*: imprimirá una línea recta entre dos puntos no importa la proyección usada.

- *LineType_GreatCircle*: imprimirá una línea entre dos puntos, la cual guardará la menor distancia geográfica entre ellos.

- *LineType_Rhumb*: imprimirá una línea entre dos puntos de constante rumbo (dirección constante).

4.5.2. Tipos básicos OMGraphics

Gracias a los tipos de *OMGraphics* o a la combinación de ellos, se puede representar prácticamente cualquier tipo de datos sobre el mapa.

OMArc y *OMCircle*: con OMCircle podemos representar un círculo o elipse. *OMArc* representa arcos. Podemos especificar el ángulo y la longitud del arco en grados decimales. *OMCircle* es una subclase de *OMArc*.

OMGrid: Objeto que representa o puede contener información en dos dimensiones en espacios igualmente separados, es decir, en una cuadrícula.

OMLine y *OMArrowHead*: objeto que representa el camino entre dos puntos sobre el mapa. *OMLines* puede usar el objeto *OMArrowHead* y convertirse en flechas. Dichas flechas pueden ponerse al final o al principio de la línea.

OMPoly: representa un camino creado por diferentes puntos. Este objeto puede ser cerrado (*polygon*) o abierto (*polyline*). En caso de ser un polígono se puede pintar su interior.

OMDistance: es una representación de *OMPoly* en *lat/lon* etiquetada con la distancia entre dos puntos y la distancia total acumulada.

OMSpline y *OMDecoratedSpline* implementan un algoritmo para añadir curvas entre los puntos de un polígono. *OMDecoratedSpline* permite también añadir etiquetas sobre la curva.

OMRect: objeto que representa un cuadrado dentro de los límites de unas coordenadas.

4.5.3. Familia OMRasterObject

OMRasterObject es usado para representar imágenes, que se representan de acuerdo al pixel de su parte superior izquierda y se pintan sobre el mapa. Los objetos de este tipo se pueden rotar. De él heredan *OMBitmap*, *OMRaster*, *OMScalingRaster* y *OMScalingIcon*.

OMBitmap son imágenes de dos colores. Puede ser usado para crear imágenes a partir de archivo de texto mediante la clase XBMFile.

OMRaster es la clase principalmente usada para representar imágenes. Un *OMRaster* puede ser creado desde información de pixeles en ARGB con formatos como (JPG, GIF, PNG, TIFF).

OMScalingRaster es una subclase de *OMRaster* que además de permitir especificar la ubicación en el mapa respecto a la parte superior izquierda, también permite especificar la ubicación de la parte inferior derecha de la imagen.

OMScalingIcon es una subclase de *OMScalingRaster*. Usada para representar objetos en el mapa. Centra la imagen sobre una localización y escala la imagen en función de la proyección dada. Se le pueden dar valores de escala máxima, mínima para que decida desde donde y hasta cuando escalar.

OMText: Objeto para introducir texto en el mapa. Permite múltiples líneas de texto, rotarlo, justificación entre otras herramientas.

5. Desarrollo de aplicaciones

Este apartado aborda la evolución de los radares, empezando por el más básico hasta llegar a la aplicación basada en la plataforma libre OpenMap, aplicación llamada TrafficMapMonitoringSync. Se irán introduciendo conceptos y funciones a medida que sean necesarios. Además, a partir de aquí, se considerará que se dominan los conocimientos suficientes de Java para no tener que explicar cada instancia del código, así mismo como OpenMap y el funcionamiento de la antena, pues todo ello ya ha sido previamente desarrollado y no compele a este apartado. También, se aprovecha la oportunidad para explicar que solo se mostrará durante el desarrollo del documento las partes más relevantes del código. Para su visualización ente contacte al redactor de este documento o al mismo tutor, aunque en anexos se incluye las principales partes del código.

5.1. Antecedentes

Para programar una aplicación de la envergadura creada, se necesita primeramente un proceso de aprendizaje, desarrollo de otras aplicaciones más simples para introducirnos en un nuevo lenguaje y al campo de estudio (monitorización aérea). Por ello, se han repasado todas las prácticas de la asignatura de Master Sistemas de Gestión de Vuelo por Computador, material desarrollado por Juan Antonio Vila Carbó así como por el tutor de este trabajo, quienes están a cargo de impartirla. Se toman las siguientes clases comunes de dichas prácticas como base para el desarrollo del presente trabajo.

5.1.1. Clases comunes

Comenzaremos explicando una serie de clases no solo comunes a estas primeras aplicaciones con interfaz gráfica, sino que también heredará con algunas modificaciones nuestra aplicación TrafficMapMonitoringSync.

Clase FlightObject

Comenzaremos con la clase más sencilla. Ha sido heredada en todas las versiones del radar y se encarga de guardar la información de cada aeronave que lea nuestra antena, ya sea una nueva aeronave para la cual creamos todas las variables necesarias para almacenar hasta sus 27 campos considerados o simplemente actualizar su estado. En la captura de código siguiente podemos ver las variables consideradas para cada aeronave.

```
public class FlightObject {

    public String AircraftID;
    public String HexIdent;
    public String FlightID;
    // This fields only updated upon position variation:  MSG==2 or MSG==3
    public String DateMessageGenerated;
    public String TimeMessageGenerated;
    // ---------------------------------------------------------------------
    public String Callsign;
    public String Longitude;
    public String Latitude;
    public String Altitude;
    public String GroundSpeed;
    public String Track;
    public String VerticalRate;
    public String Squawk;
    public String Alert;
    public String Emergency;
    public String SPI;
    public String IsOnGround;

    public double lon;
    public double lat;
    public double alt;
    public double track;
    public double gs;
    public double vrate;

    public long timestamp = (long) 0;

    // additional fields
    public Color color;
    public int xCoord; // normalized graphics coordinates
    public int yCoord;
```

Captura código 1: Variables para cada aeronave

Podemos llamar a esta clase por medio de dos constructores. El primer constructor es *FlightObject()* y el segundo *FlightObject(String _AircraftID, String _HexIdent, String _FlightID, String _DateMessageGenerated, String _TimeMessageGenerated)* que contiene 5 parámetros los cuales corresponden exactamente a los 5 primeros campos o variables de la captura de código 1, el resto de campos se inicializan nulos o con un '0' para evitar errores al grabar los ficheros.

Esta clase presenta once métodos de los cuales 8 métodos son para la actualización de información de la aeronave.

Son los métodos:

- *UpdateCallSign(String value)*
- *updateMSG2(String [] value)*
- *updateMSG3(String [] value)*
- *updateMSG4(String [] value)*

- *updateMSG5(String [] value)*
- *updateMSG6(String [] value)*
- *updateMSG7(String [] value)*
- *updateMSG8(String [] value)*

No es necesario explicar que datos actualiza cada método pues se puede ver claramente en la tabla 4.

El resto de métodos son:

- *toString():* devuelve el *HexIdent* de la aeronave
- *println():* imprime en pantalla toda la información de la aeronave a excepción claro de las variables color, *xCoord* e *yCoord*
- *setGraphicsCoord(int x, int y):* guarda la posición de la aeronave en el sistema del coordenadas del mapa.

Clase AntenaReceiver

En primer lugar, como introducción a la antena y así mismo para mostrar su evolución, se hablará de la clase original *AntenaReceiver* y más delante de la modificada para la *GUI-1*, *AnetenaReceiverListener*. Todas las versiones de radares que se muestran e incluso la versión que se presentará con la plataforma OpenMap, toma como base el siguiente código.

```
AntennaReceiver(String host, int port) throws java.net.UnknownHostException, java.io.IOException {
    phone = new java.net.Socket(host, port);
    br = new java.io.BufferedReader(new java.io.InputStreamReader(phone.getInputStream()));
    pw = new java.io.PrintWriter(phone.getOutputStream());

}
```

Captura código 2: Constructor AntennaReceiver

Como se puede ver en la captura de código 2. Los parámetros que pasamos al constructor son por un lado un host de tipo *string* y por otro el puerto como entero.

Para conectarnos a la antena de la UPV:

- Host: telnet servantena.etsid.upv.es
- Puerto: 30002

Aspectos sencillos pero importantes que debemos ir aclarando sobre *AntenaReceiver* y sus futuras versiones y modificaciones, es que esta clase presenta los métodos *starit()* y *stopit()* que se verán 'heredados' en las futuras versiones de esta clase. El método *starit()* es el encargado de conectarnos a la antena y se mantendrá leyendo mientras la variable *running* sea igual a *true*, variable que igualaremos a *false* mediante el método *stopit()* para terminar la conexión. Esto se ha programado como un *thread* (hilo) como era de esperar pues

mientras nuestro programa lee datos de la antena al mismo tiempo tiene que plotearlos en la pantalla, realizar cálculos gráficos, estar atento a posibles eventos entre muchas otras cosas más y además de manera sincronizada, otra nueva razón por la cual Java ha sido el lenguaje de nuestra elección.

Clase AntennaReceiverListener

Ahora bien, entendido el funcionamiento básico de la lectura de *AntenaReceiver* pasamos a *AntenaReceiverListener*.

Nada más empezar vemos la primera diferencia notable respecto a la clase anterior que únicamente debía leer la antena, esta pretende dar un paso más adelante y encargarse de leerla y almacenar ya esos datos en sus correspondientes variables. Ahora bien, una solución para trabajar con varios hilos, leer y escribir sobre estructuras de almacenamiento es usar un *HaspMap* y sincronizarlo. Sin embargo, cada vez que accede un hilo, bloquea a los otros para que no puedan acceder y esto no es realmente concurrencia, pues un hilo modifica, suelta el recurso y entra el siguiente hilo, es decir trabaja secuencialmente. Solución que se ha encontrado con *ConcurrentHashMap*, ya que, esta bloquea solo una porción del *Map* y no el *Map* completo. Además, *ConcurrentHashMap* es seguro ante hilos, permite infinitas lecturas y hasta 16 escrituras al mismo tiempo.

De esta manera se crearán dos estructuras de datos las cuales se trabajarán mediante *ConcurrentHashMap.* Una será una lista de aquellas aeronaves con información de su posición y otra sin información de posición. Es importante hacer esta distinción, pues será necesaria dicha información para su impresión en el mapa del radar.

```
// List of flights with position information
java.util.concurrent.ConcurrentHashMap<String, FlightObject> traffics;
// List of flights with no position information
java.util.concurrent.ConcurrentHashMap<String, FlightObject> traffics_np;
```

Captura código 3: Uso de ConcurrentHashMap para manejar datos concurrentemente

En cuanto a constructor, tenemos dos. Uno vacío *AntennaReceiverLtner()* y otro *AntennaReceiverLtner(TrafficListener tl, String host, int port*) que presenta el *host* y *port* igual que antes, solo que ahora introduce una variable *tl* de tipo *TrafficListener*.

Aprovechamos para explicar que dentro del paquete GUI-1, tenemos dos interfaces:

TrafficListener: con sus métodos

- *insertFlightObject(FlightObject flight)*

- *removeFlightObject(FlightObject flight)*
- *updateFlightObject(FlightObject flight)*

TrafficGenerator: con sus métodos

- *FlightObject[] getFlightObjects()*
- *byte[] getFlightsByteArray()*
- *ConcurrentHashMap<String, FlightObject> getTraffics()*

Ahora bien, explicaremos brevemente como funciona esta clase, para seguidamente poder centrarnos más en cada aspecto. Como bien sabemos lo que haremos es dividir en dos estructuras de datos o listas los vuelos en función de si tiene o no información de su posición. Ahora, teniendo esto en cuenta, la antena estará continuamente recibiendo información, dicha información será leída línea a línea, descodificada y se actualizará lo que sea necesario. Por ejemplo, llega una nueva lectura, miramos que tipo de mensaje es y si dicho vuelo está o no en alguna de las listas. Con lo que definimos su *state*, el cual toma valor 0 cuando la aeronave es nueva y no está en ninguna de las listas. Valor 1 cuando no está en *traffics* pero sí en *traffics_np*, y finalmente como 2 cuando está en *traffics*. Esto se utiliza para saber qué acciones llevar a cabo, es decir, a qué métodos llamar en función del tipo de mensaje. Por ejemplo, si es *state* 2 y tipo de mensaje MSG 2, se llamaría al método *updateMSG2* de la clase *FlightObject*. Seguidamente, lo único que faltaría hacer es notificar a *TrafficListener*. De nuevo, en función del estado, este lanzaría un método u otro, para el ejemplo *updateFlightObject*.

Clase interna de AntennaReceiverListener, traffic_cleaner

Mediante las listas de *traffics* y métodos pertinentes a *AntenaReceiverListener* y *TrafficGenerator*, se añaden o actualizan las aeronaves y su información. Sin embargo, puede darse el caso de que se pierda la señal de un avión, sea por ejemplo, porque ha salido de nuestro radio de cobertura o simplemente haya aterrizado. Dicha aeronave debe ser extraída y eliminada de la lista *traffics* a la que pertenezca. Para ello se ha creado esta clase y su método *clean_lost_flights()*, los cuales se encargan de eliminar toda avión de la cual no se haya recibido información durante 60 segundos y de la misma manera que la clase principal, si elimina un vuelo debe comunicárselo a *TrafficListener*. Como es de esperar, todo esto se ejecuta en paralelo al resto de actividades de la antena.

5.1.2. Radar básico: GUI1

Asumimos todo lo previamente dicho y comenzamos con la primera aplicación de este trabajo fin de grado.

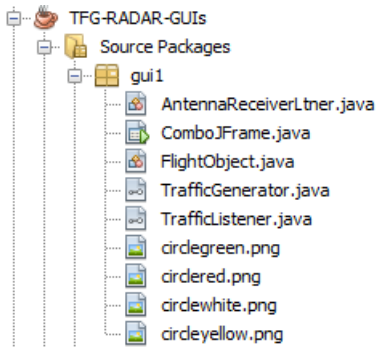

Figura 12: paquete GUI1

El primer radar, es una interfaz sencilla,

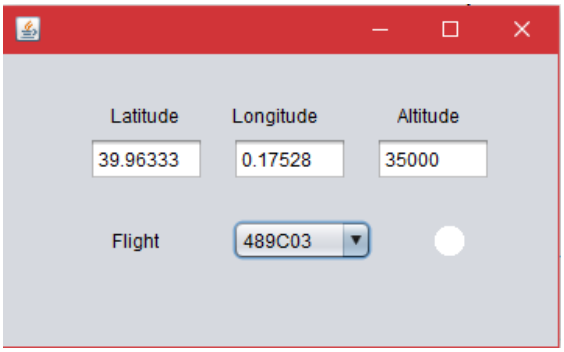

Figura 13: GUI-1

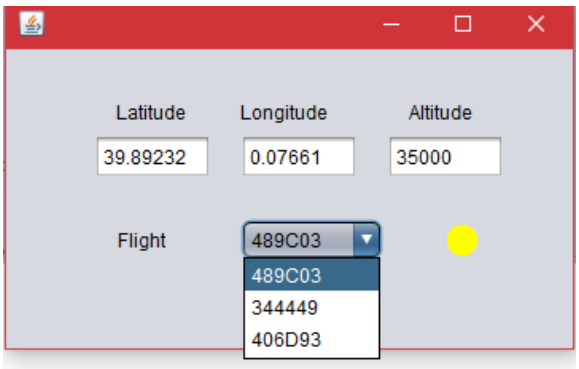

Figura 14: GUI-1 desplegado ComboBox

Como podemos ver consta básicamente de *Labels*, *TextFields* y un *ComboBox*. Aunque, también se ha introducido un círculo que cambia de color en función de si un avión es añadido (color verde), borrado (color rojo), actualizado su estado (color amarillo) o corriendo el programa (color blanco). Aquí, se explica el uso de *TrafficListener*. En *AntennaReceiverLtner* la introducción, actualización o borrado de una aeronave debía ser notificada a *TrafficListener*, pues aquí está la razón, como podemos ver en la captura de código 4, la clase *ComboJFrame* es usada para la interfaz gráfica, es decir, que estos cambios se puedan visualizar en nuestra *GUI-1*.

```
@Override
public void insertFlightObject(FlightObject flight) {
    flightComboBox.addItem(flight);
    System.out.println("added");
    activityLabel.setIcon(fInserted);
    blink();
}

@Override
public void removeFlightObject(FlightObject flight) {

    flightComboBox.removeItem(flight);
    System.out.println("deleted");
    activityLabel.setIcon(fDeleted);
    blink();
}

@Override
public void updateFlightObject(FlightObject flight) {

    FlightObject fo;
    fo = (FlightObject) flightComboBox.getSelectedItem();

    if (flight.equals(fo)) {
        latTextField.setText(flight.Latitude);
        lonTextField.setText(flight.Longitude);
        altTextField.setText(flight.Altitude);
    }

    System.out.println("updated");
    activityLabel.setIcon(fUpdated);
    blink();
}
```

Captura código 4: métodos de TrafficListener

Como método para evitar errores se ha introducido la instrucción *while (ar.isAlive())* dentro del método que maneja el evento para cerrar o terminar la aplicación *formWindowClosing(java.awt.event.WindowEvent evt)*. Cuando se produce el evento, este método cierra la conexión, pero para asegurarnos de que lo ha hecho antes de terminar con el programa usamos la instrucción *isAlive()*, la cual se mantendrá como *true* hasta que la conexión con la antena (*ar*) se haya realmente cerrado.

Un programador, puede afrontar la creación de interfaces de dos maneras en función también del entorno que estemos usando para programar. Por un lado, tenemos la programación clásica de interfaces línea a línea de código, usando los componentes de Java, *AWT*, *Swing*, *Java2D* entre otros. Lo que resulta es una tarea realmente tediosa, pues para solo crear una interfaz sencilla como la *GUI-1* requiere 300 líneas de código o, por otro lado, podemos usar el entorno de programación Java, Netbeans y su asistente el cual facilita enormemente la creación de interfaces gráficas. Pues dispone de un entorno gráfico donde podemos crear todos los componentes de *Swing* simplemente arrastrando o clicando sobre ellos desde su paleta. Además, podemos modificar las propiedades de dichos elementos desde el editor de propiedades y ver qué elementos tenemos creados desde el inspector.

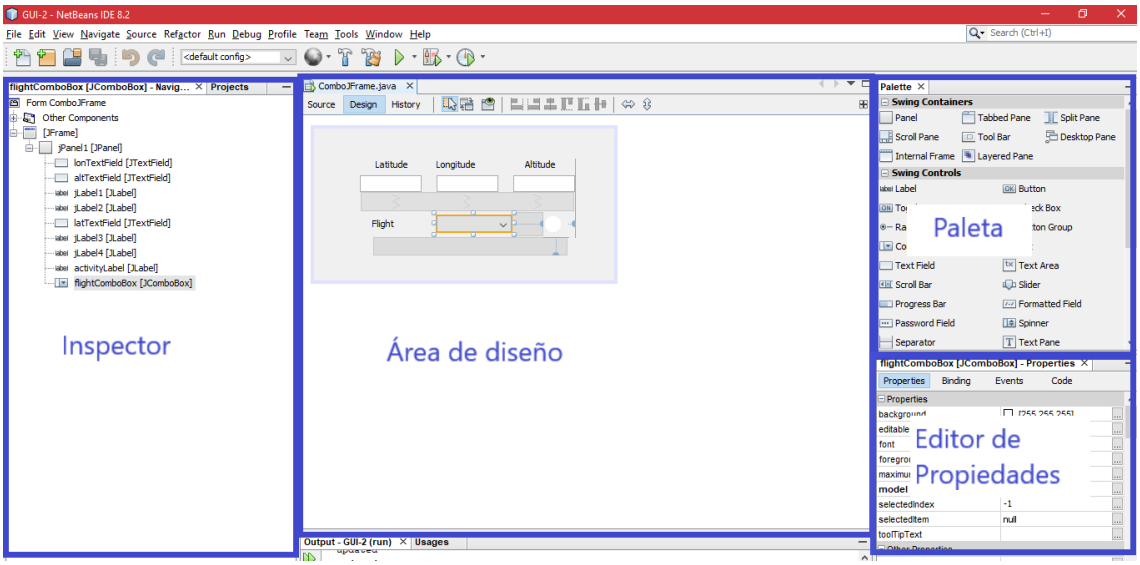

Figura 15: asistente para la creación de GUIs de Netbeans

5.1.3. Radar mapa básico: GUI2

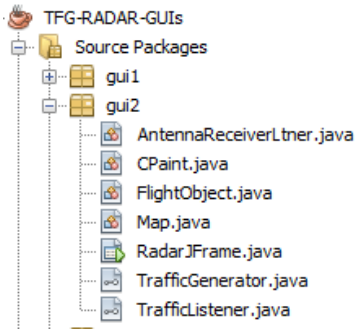

Figura 16:paquete GUI2

La siguiente versión va un paso más adelante y presenta una interfaz gráfica mucho más visual. Como bien se puede apreciar en la Figura 17, se representa un mapa de la Comunidad Valenciana y sobre ella se imprimen los vuelos.

Figura 17:GUI2

Clase Map

Esta clase es realmente sencilla y corta, se utiliza para organizar la información que se leerá del mapa.txt y así posteriormente poder imprimirla en el *JPanel* más fácilmente.

Consta de un constructor *Map(int size)* que lanza la clase interna *Polygon(int size)* que irá guardando dentro de ella los puntos de *Point(double lat, double lon)*, otra clase interna.

Clase CPaint

Es evidente que las aeronaves se moverán en pantalla y tendremos que ir repintándolas sobre el mapa, siendo la función *paintComponent* del objeto donde se pretenda pintar, la encargada de realizar esta tarea. Sin embargo, Netbeans no permite modificar un objeto incrustado directamente desde el *Design* (en este caso se trata del *JPanel* que hemos llamado *RadarJFrame*). La solución es crear una nueva clase de Java a la que llamaremos, *CPaint* y hacerla derivar de *JPanel*. De esta manera conseguiremos modificar componentes incrustados con el *Design* desde una clase ajena sea este caso *CPaint*. Para ello, en el asistente de diseño de

Netbeans (*Design*), haremos clic derecho sobre el objeto en cuestión y del menú desplegable seleccionaremos *Customize Code* (código personalizado) y ahí introduciremos lo siguiente (véase Figura 18):

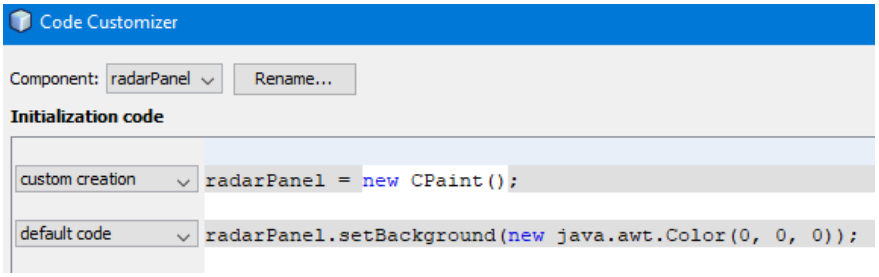

Figura 18: customize code

Una vez arreglado esto, comenzamos con el trabajo *CPaint*. En primer lugar, se establece los límites de nuestro mapa en coordenadas, en otras palabras, las coordenadas de los extremos del mapa de la Comunidad Valenciana que se va a representar. Así como también se calcula el tamaño del mapa.

```
private Double LonWest = Double.valueOf("-1.78"); // CCVV area
private Double LonEast = Double.valueOf("1.22");
private Double LatNorth = Double.valueOf("40.79");
private Double LatSud = Double.valueOf("37.79");

private Double SizeEW = this.LonEast - this.LonWest;
private Double SizeNS = this.LatNorth - this.LatSud;
```

Captura código 5: coordenadas del mapa (límites y tamaño)

El constructor *CPaint()* carga el mapa a partir de un fichero llamado mapa.txt. Este mapa es un gran conjunto de puntos con coordenadas que nos permite dibujar el mapa de la Comunidad Valenciana punto a punto.

Esta clase presenta los siguientes métodos:

- ***setAr(AntennaReceiverLtner ar)*:** simplemente es un método para inicializar la clase *AntennaReceiverListener*.

- ***paintFlightData(Graphics2D g)*:** es el responsable de leer las aeronaves que se obtienen de *AntennaReceiverListener* a través del método *getFlightObjects()* e imprimirlas sobre el mapa. Para ello, aeronave por aeronave lee sus coordenadas las normaliza para saber dónde están respecto a las coordenadas del mapa y si están dentro las imprime. Como se puede ver en la captura de código 6, cada aeronave se representará con un pequeño cuadrado y una línea recta que sale de él hacia su *HexIdent* impreso también en pantalla como muestra la Figura 19.

```
g.fillRect(fo[i].xCoord - 2, fo[i].yCoord - 2, 5, 5);
g.drawLine(fo[i].xCoord, fo[i].yCoord, fo[i].xCoord + 5, fo[i].yCoord - 5);
g.drawString(fo[i].HexIdent, fo[i].xCoord + 5, fo[i].yCoord - 5);
```

Captura código 6: detalles de la impresión de cada aeronave

Figura 19: representación gráfica de una aeronave en la GUI2

- ***loadMap(String file):*** este método es el encargado de leer el fichero 'mapa.txt' y guardarlo en la estructura creada por *Map*, es decir, la clase *Map* es una estructura de datos que usará el método *loadMap* para guardar convenientemente estructurada la información del fichero para que más adelante el método *paintMap* pueda usarlo.

- ***paintMap(Graphics2D g):*** finalmente, este método imprime los datos del mapa.txt estructurados y guardados en *Map*, más exactamente en map.*data*. Punto por punto se normalizará, imprimirá y unirá para formar el mapa. Para esto se ha usado la clase *Path2d* propia de Java para construir la forma deseada y para unir los puntos sencillamente usando *lineTo*.

  Nota: obviamente para imprimir todo esto es necesario el método *paintComponent* y dentro de él, sentenciar aquello que queremos imprimir. En este caso ha sido *paintFlightData* y *paintMap*, siendo *g* un parámetro de tipo *Graphics2D*

Clase RadarJFrame

La mayoría del código se ha desarrollo fuera de *RadarJFrame*, lo que ha dejado esta clase limpia, sencilla y casi sin código. Implementa *TrafficListener* que como la anterior *GUI*, lo usa para ser notificado de las modificaciones en los vuelos que deberá repintar, mediante el método *updateFlights()* y la instrucción *radarPanel.repaint().*

El constructor *RadarJFrame*() se encarga de inicializar la lectura de la antena (*AntennaReceiverListener*) y la impresión del mapa y las aeronaves (*CPaint*).

Finalmente, decir que, aunque no se haya comentado las clases *AntennaReceiverLtner*, *FlightObject* y las interfaces *TrafficListener* y *TrafficGenerator* se han heredado sin modificación alguna de la *GUI1* y lo mismo será para *GUI3*.

5.1.4. Radar mapa BitMap: GUI3

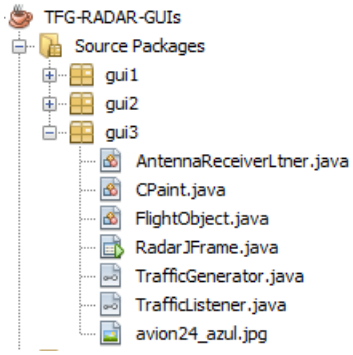

Figura 20: paquete GUI3

Presentamos como *GUI3* un radar con una imagen como fondo, sobre la cual pintamos las aeronaves. Como se muestra en la Figura 21, las aeronaves han sido representadas con un pequeño icono azul, que además se orientan en función del *track* de las mismas.

Figura 21: GUI3

Clase CPaint

De nuevo volvemos a crear esta clase para modificar componentes incrustados desde el *Desing* de Netbeans. Tendremos que seguir los mismos pasos que en la *GUI2*, repecto a Customize Code. Por otro lado, está ha sido mayormente modificada. En la *GUI2 CPaint* extendía de *JPanel*, pero en este caso extiende de *JLabel*. Aunque esto pertenece a la parte del código de *RadarJFrame*, se aprovecha la oportunidad de que hemos introducido el concepto de *JLabel* para explicar cómo se ha creado la imagen de fondo. Dicha operación es realmente sencilla. Simplemente se crea un *JLabel* dentro de nuestro *JFrame* y le añadimos un icono. En este caso nuestra imagen que hará de icono es 'radar.png' y el nombre de nuestro *JLabel* es *radarPanel*.

```
ImageIcon iim = new ImageIcon("radar.png");
radarPanel.setIcon(iim);
radarPanel.setSize(iim.getIconWidth(), iim.getIconWidth());
```

Captura código 7: JLabel usado como fondo de pantalla para imprimir la imagen del mapa

Volviendo a la clase *CPaint* se inicia exactamente igual que la *GUI2*, es decir, con las mismas líneas de código que la captura de código 5. El constructor que antes cargaba la 'imagen.txt', ahora carga la imagen que se usará para representar las aeronaves. Para ello, usamos *imageIO* que reconoce la imagen en formato PNG (también GIF, JPEG, BMP y WBMP) y lo decodifica en un *BufferedImage* que puede ser utilizado directamente por Java 2D.

Esta clase presenta los siguientes métodos:

- ***setAr(AntennaReceiverLtner ar):*** heredado exactamente igual de la *GUI2*.

- ***paintFlightData(Graphics2D g)****:* al igual que en la *GUI2*, se encarga de imprimir todas las aeronaves que se encuentre en la lista de *traffics* de *AnntenaReceiverListener*, recordamos que son aquellas que tienen información sobre su posición. Solo que en vez de usar cuadrados y usaremos un icono con forma de aeronave y lo posicionaremos en función del *track* que está siguiendo. Con este fin, haremos uso de *AffineTransform* que nos permite movernos entre sistemas de coordenadas 2D y da herramientas para hacer giros, cambio de escala, etc. Como vemos en la captura de código 8, el orden para realizar el giro es el inverso, el ángulo

de giro lo obtenemos de la variable *track* de *FlightObject*, el giro se realiza respecto el centro de la imagen y finalmente, mostramos su *HexIdent*.

```
AffineTransform at = new AffineTransform();
// inverse order:
// 3. translate it to the center of the component
at.translate(fo[i].xCoord, fo[i].yCoord);
// 2. do the actual rotation
at.rotate(Math.PI*fo[i].track/180.0);
// 1. translate the object so that you rotate it around the center
at.translate(-aircraft.getWidth() / 2, -aircraft.getHeight() / 2);
g.drawImage(aircraft, at, null);
g.drawString(fo[i].HexIdent, fo[i].xCoord + 5, fo[i].yCoord - 5);
```

Captura código 8: Rotación del icono de la aeronave según su track

Nota: se ha hecho de esta manera para visualizar más claramente el proceso. Sin embargo, existe una instrucción para realizarlo directamente *getRotateInstance(double theta, double anchorx, double anchory).*

Clase RadarJFrame

Como se ha mostrado anteriormente la imagen de fondo se ha introducido por medio de código, es decir, línea a línea. Sin embargo, con el asistente de Netbeans podría haberse realizado de forma automática. Sencillamente habríamos elegido un *JLabel* de la paleta del asistente, lo habríamos extendido manualmente hasta el tamaño deseado, clicado en el apartado icono de sus propiedades y haber seleccionado la imagen pertinente. El código se habría creado automáticamente.

A parte de lo anteriormente explicado, no hay nada remarcable en esta clase pues es prácticamente idéntica a la de la *GUI2*. Únicamente, decir que, para mantener el mapa y sus proporciones, así como donde deberían estar los vuelos respecto a él, se ha prohibido la modificación del tamaño de la ventana. De esta manera nos aseguramos que siempre se mantengan las proporciones, algo realmente importante puesto que estamos visualizando vuelos sobre un mapa. La instrucción responsable de esto es meramente una línea de código en el constructor de *RadarJFrame*, *setResizable(false)*.

5.2. OpenMap: TrafficMapMonitoringSync

Una vez introducidas las aplicaciones radar anteriores, se presenta *TrafficMapMonitoringSync*, la cual incluye la plataforma OpenMap como medio para representar aeronaves sobre un mapa y poder manejar diferentes herramientas, ya sean, capas, proyecciones, entre otras. Por supuesto, se consideran los conceptos de OpenMap desarrollados en el punto 3 de este documento como asimilados para poder afrontar la siguiente explicación. En caso de necesitar apoyo extra, se incluye en el anexo un resumen más elaborado sobre OpenMap.

5.2.1. Punto de partida y estructura

OpenMap es una librería con mucha potencialidad. Sin embargo, las ayudas son bastante escasas, su API aparte de definiciones no presentan grandes ayudas. Ejemplos son difíciles de encontrar y la comunidad de internautas y aficionados no se ha movido mucho. Por tanto, a pesar de tener grandes herramientas escasean los proyectos, ejemplos o cualquier otra ayuda. Esto lleva a que sea un terreno inexplorado, lo cual complica bastante la programación. Para poder entender esta librería, se ha tenido que investigar y respecto a recursos que pueda ofrecer la propia OpenMap, dispone de dos presentaciones en pdf y un documento de unas setenta páginas, pero de nuevo la falta de ejemplos prácticos abunda en todos ellos.

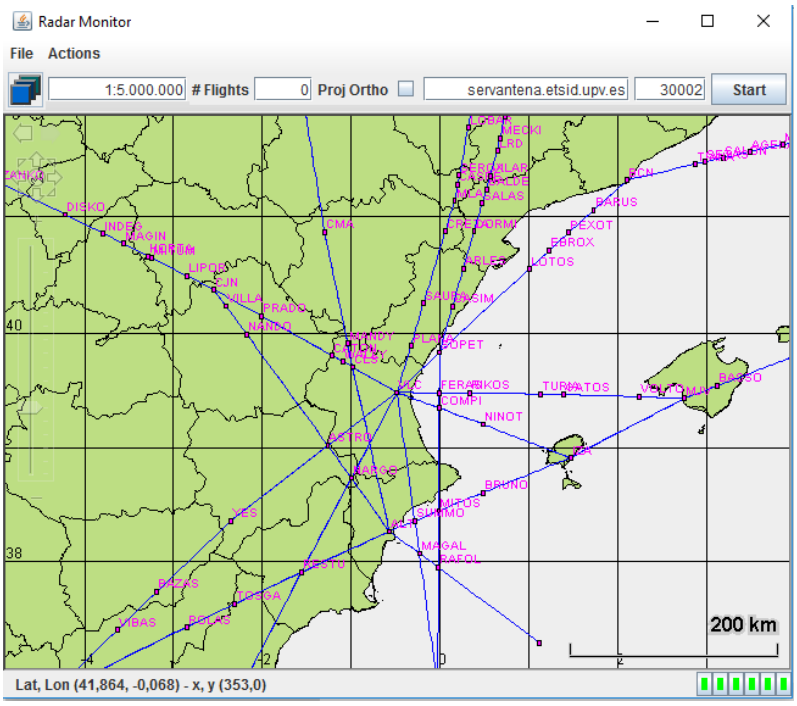

Figura 22: aplicación TrafficMapMonitoringSync

Una vez conocida la base de la cual se partió en OpenMap, pasaremos a explicar cómo presentaremos esta nueva herramienta. Se seguirá la estructura del mismo proyecto en Java, por tanto, se dividirá en seis partes que se corresponden con los seis paquetes del mismo proyecto, aunque no se presentarán en el mismo orden. Pero antes, presentaremos un diagrama de su funcionamiento general.

Cabe explicar que, respecto al mismo nombre de la aplicación, del inglés *TrafficMapMonitoringSync*, hace referencia a que es una aplicación de monitorización (*Monitoring*) del tráfico aéreo (*Traffic*) en una interfaz gráfica, más exactamente sobre un mapa (*Map*). El *Sync* hace referencia a la forma de trabajar o en otras palabras como se comunican los nuevos datos de la antena a nuestra aplicación en general, que en vez de tener un bucle continuamente esté leyendo la antena para ver si hay nuevos vuelos, lo que hacemos es uso de los conceptos de *Listener* y *Generator*, así cuando la antena (*Generator*) tenga algo que comunicar se lo enviará a la clase en cuestión que estará a la escucha (*Listener*). Es una manera más eficiente de trabajar, pues no tenemos un *thread* continuamente leyendo la antena y ahorramos recursos de computación.

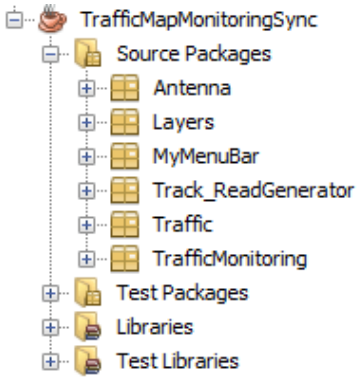

Figura 23:Proyecto TrafficMapMonitorinSync

Nota1: para añadir la librería de OpenMap a nuestro entorno de programación, solo debemos descargarnos la versión que se desee de su misma página oficial (openmap-java.org) donde encontraremos un apartado llamado *Downloads*, del inglés descargas. Una vez descargado, en el apartado de librerías de nuestro proyecto solo deberemos añadir, el archivo *jar* descargado. En el mismo apartado de *Downloads* encontraremos para cada versión de la librería otro archivo con el mismo nombre, pero añadiendo API, son las ayudas que ofrece el entorno de programación sobre errores, posibles instrucciones o sugerencias sobre OpenMap. Se añade de la misma manera.

Nota2: Debemos aclarar que debido a que el desarrollo de este programa surge de diversas y largas colaboraciones hemos creído que lo más apropiado es firmarlos bajo ‘fms’, es decir, Flight Manager System.

5.2.2. Diagrama TrafficMapMonitoringSync

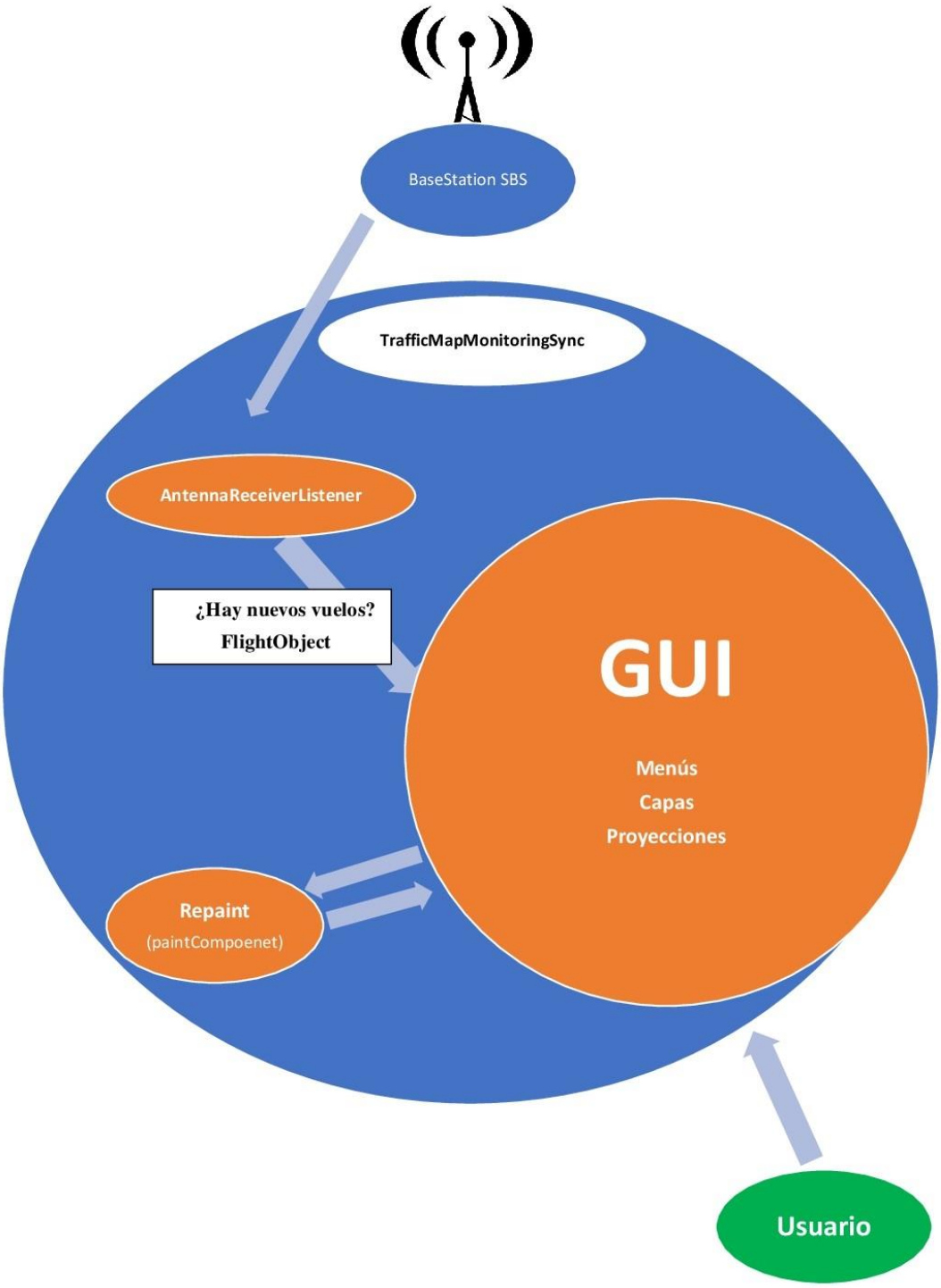

Figura 24: Diagrama aplicación TrafficMapMonitoringSync

5.2.3. Paquete TrafficMonitoring

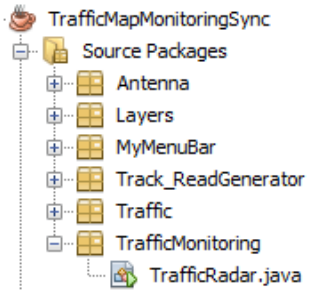

Figura 25: paquete TrafficMonitoring

A primera vista se encuentran tres clases. Empezaremos explicando la clase principal *TrafficRadar* (aquella que contiene el *main*).

Clase TrafficRadar

TrafficRadar está formado por su constructor, la clase interna *MyMapMouseListener*, el método *close_tracker()* y el método *main*.

Nada más comenzar, en el constructor *TrafficRadar* como bien debe hacerse en OpenMap siempre se definen una serie de componentes mínimos necesarios para crear un mapa, se ejemplifica en la captura de código 9 y 10.

```
MapPanel mapPanel = new OverlayMapPanel();

// Get the default MapHandler the BasicMapPanel created.
mapHandler = mapPanel.getMapHandler();

// Get the default MapBean that the BasicMapPanel created.
mapBean = mapPanel.getMapBean();

mapBean.setCenter(new LatLonPoint.Double(39.490556, -0.480278)); // Centered in Valencia
// mapBean.setCenter(40.467927f,  -3.569205f); //Centered in Madrid

mapBean.setScale(5000000f); // to scale in Spain
// mapBean.setScale(2000000f); // to scale in Valencia
```

Captura código 9: declaraciones básicas OpenMap

Como ya sabemos, *MapPanel* (*com.bbn.openmap.gui.MapPanel*) es un componente que contiene a *MapBean*, encargado del manejo de proyecciones y capas, a *MapHandler*, contenedor al que se le pueden añadir o quitar objetos (*OMGraphics*) y a una serie de menús que pueden considerarse como *JMenuBar* o *JMenu*. Por tanto, de lo primero que deberemos hacer siempre, es declarar un *MapPanel* y por consiguiente nuestro *MapBean* y *MapHandler*. Podemos ver, que se ha decido que por defecto cuando se abra la aplicación esta se escale a España y ce centre en la Comunidad Valenciana.

```
mapHandler.add(new LayerHandler());

// Add MouseDelegator, which handles mouse modes (managing mouse
// events)
mapHandler.add(new MouseDelegator());

// Add OMMouseMode, which handles how the map reacts to mouse
// movements
mapHandler.add(new OMMouseMode());
```

Captura código 10: declaraciones básicas y necesarias OpenMap

También es importante y necesario declarar el *LayerHandler*, pues sin él no podríamos para añadir, quitar o cambiar el orden de las capas o enviar eventos a los *Listeners*. *MouseDelegator* y *OMMouseMose* también son imprescindibles, dado que el primero se encarga del manejo de los eventos que pueda producir el ratón y el segundo se encarga de cómo reaccionar a dichos eventos.

Seguimos con la introducción de componentes para manejar la *GUI* de OpenMap, se ilutra el código con imágenes para una comprensíon directa de cada sentencia.

```
// Add a ToolPanel for widgets on the north side of the map.
ToolPanel tpl = new ToolPanel();
mapHandler.add(tpl);

// Add a GUI to control the layers. A
// button to launch it will get added to the ToolPanel.
mapHandler.add(new LayersPanel());
mapHandler.add(new ScaleTextPanel());
miPanel = new MyTextPanel(this);
mapHandler.add(miPanel);
```

Captura código 11: componentes de la GUI de OpenMap

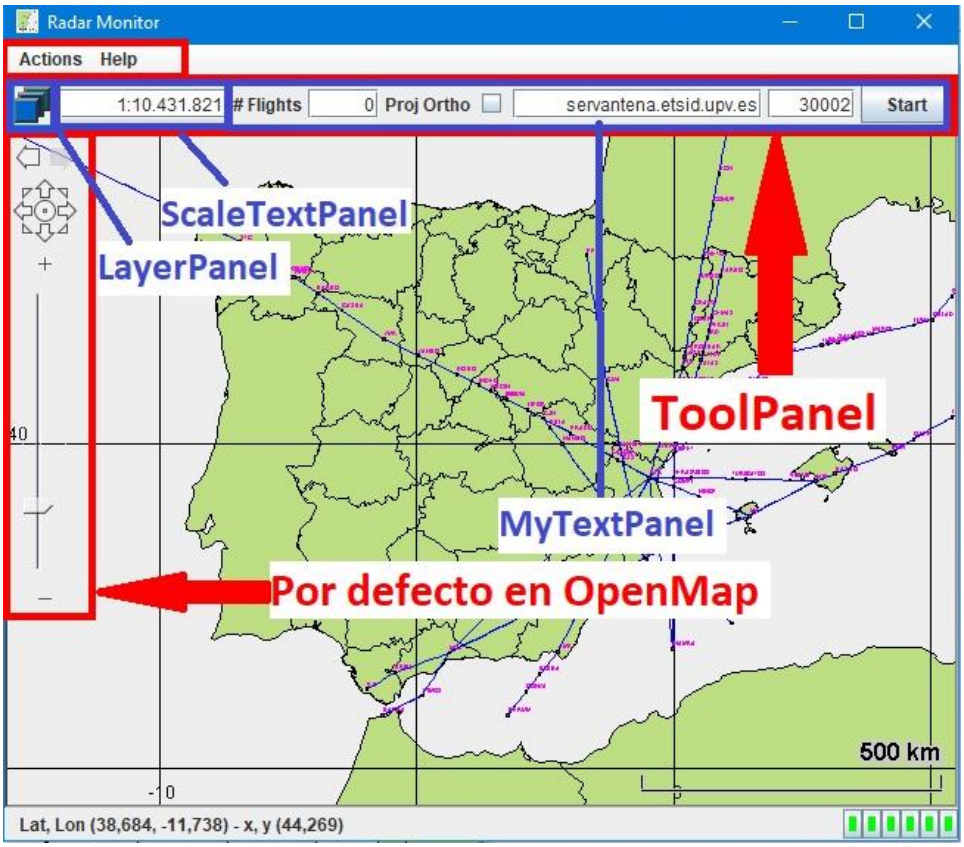

Figura 26: componentes de la GUI de OpenMap

OpenMap por defecto, introduce los *widgets* posicionados en la parte izquierda sobre el mapa, es decir, no es necesario programarlo. Un *slider* para controlar la escala, así como los ocho botones para moverse por el mapa y la fechas de adelante y atrás entre los últimos movimientos realizados.

Para la creación de la barra de menús usamos la clase *JMenuBar*. Java dispone de una serie de menús predefinidos. Sin embargo, *Actions* y *Help* son de creación propia. Se instancian en la captura de código 12 y se visualizan en la Figura 27. Su código se explicará en el paquete OpenMap donde han sido programados.

```
// Add a Menu Bar
 JMenuBar menubar = new JMenuBar();

// Add own menus to MenuBar
 MyMenuActions item = new MyMenuActions(this);
 MyMenuHelp item2 = new MyMenuHelp(this);

 menubar.add(item);
 menubar.add(item2);
 mapHandler.add(menubar);
```

Captura código 12: componentes de la GUI de OpenMap

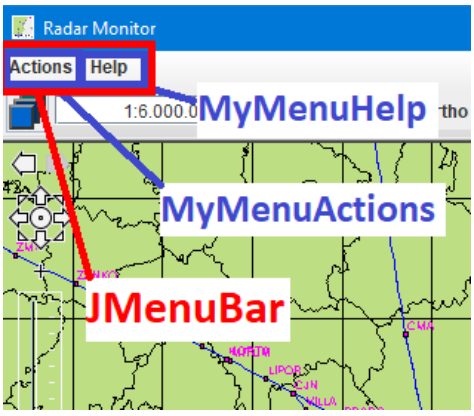

Figura 27:componentes de la GUI de OpenMap

Personalizaciones interesantes de las que se han dotado a nuestra aplicación son, por un lado, el nombre que imprimirá sobre el marco superior de la ventana de Windows (*Radar Monitor*) y por supuesto, el diseño de un icono de la aplicación. La captura de código 13 nos muestra las instrucciones necesarias para ello.

```
OpenMapFrame frame = new OpenMapFrame("Radar Monitor");

// Size the frame appropriately
frame.setSize(768, 768);
mapHandler.add(frame);

//Application Icon
ImageIcon img = new ImageIcon("iconoaplicacionfinal.png");
frame.setIconImage(img.getImage());
```

Captura código 13: personalización de la aplicación

Nota: *setSize* simplemente da un tamaño por defecto, es decir, el tamaño con el que se abrirá la ventana al ejecutar la aplicación.

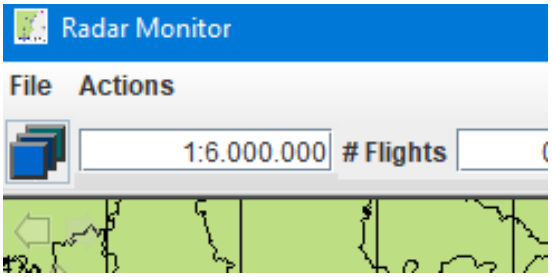

Figura 28: esquina superior izquierda de la ventana de Windows de nuestra aplicación

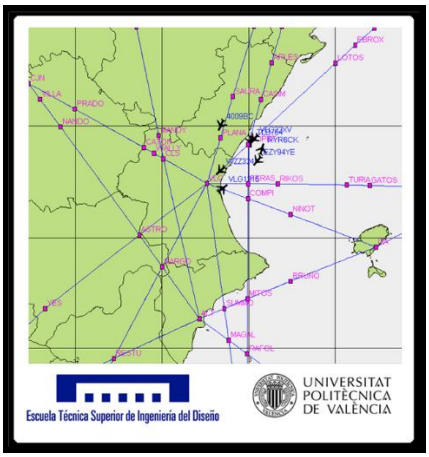

Figura 29: Icono diseñado para TrafficMapMonitoringSync

Hay ciertas capas que se crean y definen directamente dentro de esta misma clase. OpenMap toma la política *'Last on top'*, es decir que la última capa añadida es la que se imprime encima. Para una mejor comprensión y fácil asimilación iremos explicando las capas en el orden, según el cual han sido añadidas, es decir, la primera a explicar será la que se encuentre abajo del todo y la última que será la que se posicione por encima del resto.

Siguiendo con lo anteriormente dicho, la capa base de nuestra aplicación se llama *PoliticalSolid* es un *mapamundi*, del latín, mapa mundial que muestra las fronteras entre países.

Figura 30: capa base PoliticalSolid

La captura de código 14, muestra que a esta capa se ha creado a partir de un archivo tipo *shape* llamado cntry08.shp, al cual se le han dado una serie de propiedades y como toda capa se ha añadido al *mapHandler* para que tenga acceso a cualquier componente. Como ya se ha explicado, las capas son *MapMouseListener*, es decir, reciben eventos a partir de las acciones que el ratón lleva a cabo sobre ellas, ya sea, desplazarse, clicar, arrastras entre otros.

```
//////////////////WORLD POLITICAL SOLID LAYER//////////////////
ShapeLayer shapeLayer = new BufferedShapeLayer() {
    @Override
    public synchronized MapMouseListener getMapMouseListener() {
        return myMapMouseListener;
    }
};
Properties shapeLayerProps1 = new Properties();
shapeLayerProps1.put("prettyName", "Worl Political Solid");
shapeLayerProps1.put("lineColor", "000000");
shapeLayerProps1.put("fillColor", "BDDE83");
shapeLayerProps1.put("shapeFile", "cntry08.shp");
shapeLayer.setProperties(shapeLayerProps1);
shapeLayer.setVisible(true);

// Last on top.
mapHandler.add(shapeLayer);
```

Captura código 14: capa World Political Solid

Las propiedades definidas en el código para esta capa son fácilmente modificables, desde la propia interfaz gráfica que ofrece nuestra aplicación para la modificación de las capas.

La siguiente capa, esculpe las provincias de España. Capa llamada *Povinces of Spain*. Esta capa es exactamente igual a la anterior, por tanto, no es necesario ni instanciar el código ni explicarlo. El archivo del cual ha sido extraída la imagen es ESP_adm2.shp.

Figura 31: capa Provinces of Spain

Y como última capa creada y definida en esta misma clase tenemos *GraticuleLayer*, capa que simplemente se ha instanciado, pues por defecto viene definida en OpenMap.

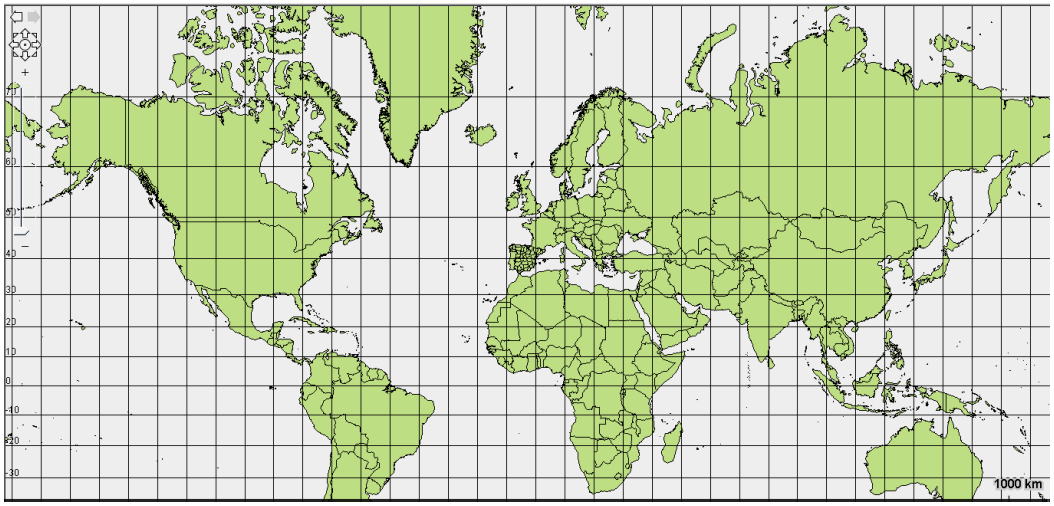

Figura 32: capa Graticule

Estas tres capas nos dan opciones muy variadas para modificar su apariencia, sea el color de fondo o de las líneas, así mismo como el patrón de las mismas o el grosor. Esto no lo tenemos que programar pues OpenMap lo incluye automáticamente para aquellas capas que han sido creadas desde un *shapeFile*. En el caso de *GraticuleLayer* por ser una capa por defecto de OpenMap como se puede ver en la captura, también dispone de una serie de herramientas.

Figura 33: manejador de capas, donde se pueden ver la opción herramientas

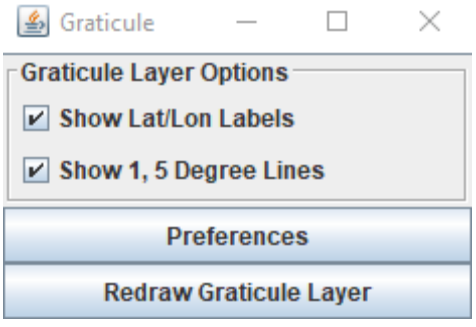

Figura 34: propiedades capa Graticule

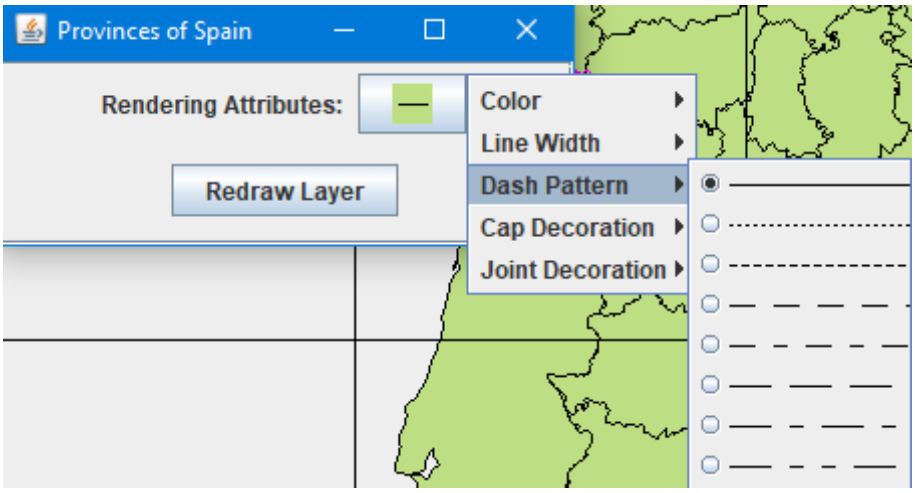

Figura 35: propiedades capa Provinces of Spain

El resto de capas se han creado en el paquete Layers, y por tanto se explicarán más adelante. En la captura de código vemos a continuación podemos ver su declaración.

```
routeLayer = new RouteLayer(this);
routeLayer.setName("Routes Layer");
mapHandler.add(routeLayer);

flightLayer = new FlightLayer(this);
flightLayer.setName("Ariplanes Layer");
mapHandler.add(flightLayer);

trackLayer = new TrackLayer(this);
trackLayer.setName("Tracker Layer");
mapHandler.add(trackLayer);
```

Captura código 15: capas creadas en el paquete OpenMap

Clase Interna MyMapMouseListener

Como sabemos parte importante de OpenMap son los eventos producidos por el ratón, como se reacciona a ellos, como se envía esta información a las capas, etc. En la captura de código 14, podemos ver claramente como la *ShapeLayer World Political Solid*, llama al método *getMapMouseListener()* para ser consciente de lo que hace el ratón.

Con todo lo dicho, esta clase extiende de *MapMouseListener* la interfaz de OpenMap encargada de los eventos del ratón. Personalizaremos el método *mouseClicked(MouseEvent e*) con la finalidad de poder clicar encima de una aeronave y con ello hacer que esta empiece a mostrar el camino que está siguiendo. Desde el mismo momento en el que se ha clicado sobre ella, se podrá de color azul e irá dibujando una línea roja de su *track*.

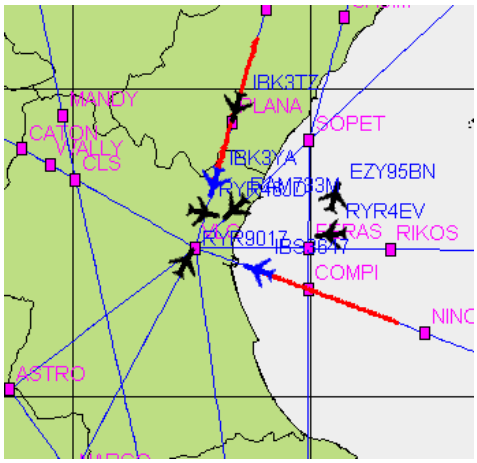

Figura 36: tracker aeronaves

El método *mouseClicked*, tomará de los eventos del ratón mediante *getX()* y *getY()* la ubicación exacta donde se ha clicado en pixeles. También se puede obtener la ubicación mediante *getLatLon()*, pero en este caso trabajar en pixeles es más sencillo, pues una vez obtenida la posición donde hemos clicado en el mapa en pixeles, lo que haremos es recoger la clase *flightObject* y comprobar aeronave por aeronave su posición XY, con un margen de tolerancia en pixeles. Es decir, que buscaremos si existe alguna aeronave que se encuentre en la posición que el ratón ha clicado con un error de ± 12 pixeles.

Si se ha dado el caso que haya alguna aeronave dentro de ese rango, comprobaremos primero si dicha aeronave ya ha sido previamente seleccionada. Si lo ha sido, la eliminaremos de la lista y sino la añadimos.

5.2.4. Paquete Antenna

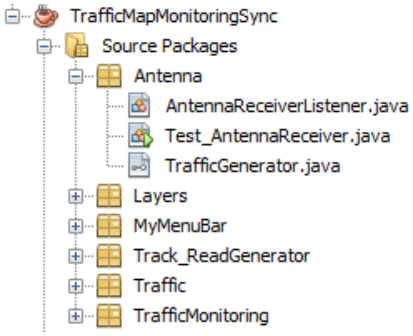

Figura 37:paquete Net

Este paquete se encarga de la conexión con la BaseStation SBS y proveer de la información para que el paquete *traffic* la descodifique y ordene en una estructura manejable. La clase *AntennaReiciverListener* y la interfaz *TrafficGenerator* son prácticamente las mismas heredadas de los anteriores

radares, solo han sufrido mínimas modificaciones de nomenclatura y en el caso de *AntennaReiverListener* ahora los constructores reciben diferentes atributos, aunque realizan exactamente la misma tarea y en cuanto atributo en relevancia solo decir que donde antes teníamos a *TrafficListener* ahora tenemos *FlightLayer*. No es necesario explicar ni ilustrar el código.

5.2.5. Paquete Track_ReadGenerator

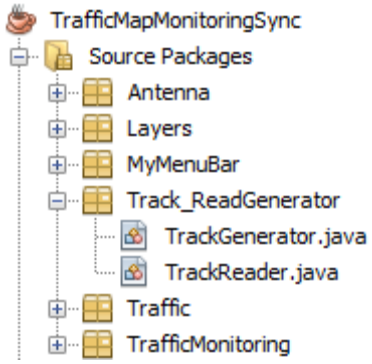

Figura 38:paquete Track_ReadGenerator

Dos clases, *TrackGenerator* que creará un archivo *txt* donde guardará la información que recibirá de la antena y *TrackReader* que leerá dicho archivo para reproducirlo en nuestro mapa como si de tráfico en tiempo real se tratase. Decir que en cuanto a la reproducción podemos elegir la velocidad con la que esta se realice respecto a la real, es decir, el doble, el triple etc. Y en lo que respecta al grabado podemos decidir el tiempo de grabado, así como interrumpir la grabación en cualquier momento.

Como es evidente mientras reproduzcamos o grabemos, queremos poder usar el programa, por ello ambas clases extienden de *Thread* y por tanto tienen un método run.

Clase TrackGenerator

Al constructor de esta clase tendrá por atributos el archivo donde deberá guardar la información, el tiempo de grabado y por supuesto, un objeto del tipo *TrafficRadar (TrackGenerator(String file, int maxt, TrafficRadar father))*.

Respecto a los métodos tenemos *toptracker()* que igualará la variable booleana *is_running* a *false* para que se detenga la grabación. Y como último método *run()*, el cual irá guardando información mientras *is_running* sea *true* y *maxt* mayor que 0.

Explicaremos en código más fácilmente mostrando el resultado del mimo (mirar Figura 39). Se imprimirá la fecha al comienzo del *txt*. Luego se entrará en el bucle del método run y seguirá en él según las condiciones mencionadas anteriormente. Se tomará la hora y se imprimirá seguida del número de vuelos que está recibiendo la antena (2 vuelo en el ejemplo de la Figura 39). En las líneas de abajo para cada una de las aeronaves detectados se escribirá su *AircraftID, FlightID, HexIdent, Callsign, Squawk*, Longitud, Latitud, Altitud, *GroundSpeed, Track, VerticalRate, Alertas* o *Emergencias* entre otros (mirar captura de código 16).

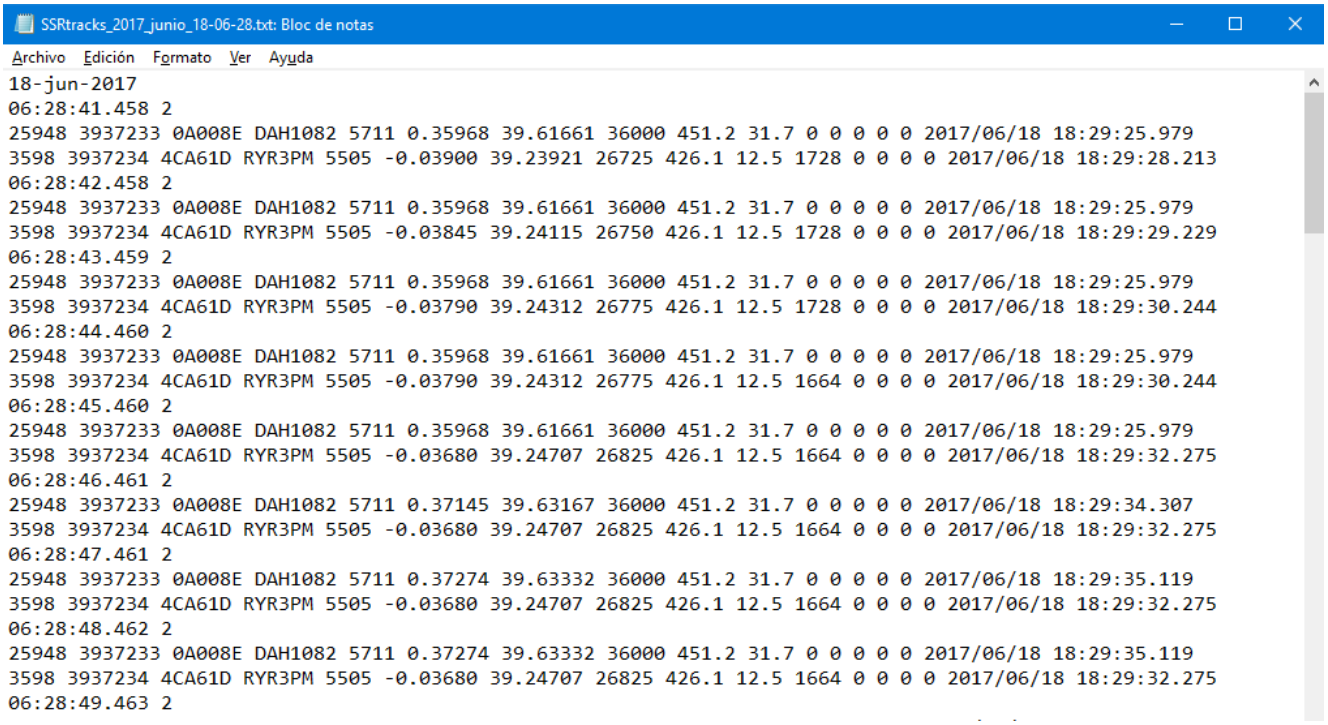

Figura 39:txt generado por TrackGenerator

```
pw.println(aux.AircraftID+" "+aux.FlightID+" "+aux.HexIdent+""
        + " "+cs+" "+sq+" "+aux.Longitude+" "+aux.Latitude+" "
        + ""+aux.Altitude+" "+aux.GroundSpeed+" "+aux.Track+" "
        + ""+aux.VerticalRate+" "+aux.Alert+" "+aux.Emergency+""
        + " "+aux.SPI+" "+aux.IsOnGround+" "+aux.DateMessageGenerated+" "
        + ""+aux.TimeMessageGenerated);
```

Captura código 16: formación impresa de cada aeronave en el documento txt

Para imprimir texto en un archivo usaremos *java.io.PrintWriter(file)* y en concreto el método *println*, com se puede ver en la captura de código 16.

Clase TrackReader

Esta clase será llamada por un evento de la interfaz gráfica de nuestra aplicación. El constructor se presenta de la siguiente manera: *TrackReader(String file, int accel, TrackLayer layer)*. Como vemos, se le debe ceder el archivo a leer, la velocidad con la que se desee reproducir y la capa donde se imprimirá.

Los métodos *starit()* y *stopit()*, poseen una nombre muy descriptivo, tal vez, mencionar que *starit()* también iniciará *mrclean*. Aprovechamos para explicar la clase interna llamada *traffic_cleaner*, la cual se encargará de borrar aquellas aeronaves de las cuales no se haya vuelvo a recibir ninguna señal en 60 segundos. Con ese fin, esta clase posee el método *clean_lost_flights()* que básicamente usará *currentTimeMillis()* para ver cuando la diferencia de tiempo entre la última captura de datos y el tiempo actual es mayor de 60 segundos para borrar la aeronave. Borrara la aeronave del *HashMap* donde está almacenada con el método *remove*.

Entrando en la materia de la lectura, usaremos *java.io.BufferedReader(new java.io.FileReader(file))* para crear un *stream buffered* del archivo y seguidamente usaremos *readLine()*, para leer línea a línea el archivo. La lectura no será complicada, pues sabemos bien como es la estructura de este archivo y cómo está organizado, solo tendremos que ir igualando posiciones a la información correspondiente.

5.2.6. Paquete Layers

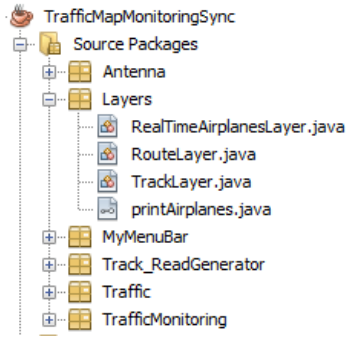

Figura 40: paquete OpenMap

En este paquete retomaremos la explicación de las capas, siguiendo el mismo orden de *'Last on top'*. Tenemos 3 capas y una interfaz.

Interfaz printAirplanes

Cualquier capa que quiera imprimir una aeronave o un conjunto de ellas debe implementar esta interfaz para tener cierto control sobre lo mínimo que se debería imprimirse de ellas. Las capas *RealTimeAirplanesLayer* y *RouteLayer* implementan esta interfaz. Los métodos de la misma son:

```
public interface printAirplanes {

    public OMText createLabel(double lat, double lon, String text, Paint color,int justification);
    public OMRaster createImage(double lat1, double lon1, double track,boolean logged);
    public void paintAirplaneImage(MapBean bean, FlightObject fo);
    public  void paintAirplaneText(MapBean bean, FlightObject fo, Color color);
    public void paintAirplanes(MapBean bean, FlightObject fo[], Color colorTexto, Color colorLog);
    public void toClear();
    public void paint(java.awt.Graphics g);
    public void rePaint(MapBean bean);
}
```

Captura código 17: interfaz printAirplanes

De esta manera esteremos obligas a imprimir siempre como mínimo una etiqueta con identificación de la aeronave y un pequeño icono para representarla. Se explicará cada método detenidamente cuando sea conveniente.

Clase RouteLayer

Esta clase se encargará de construir una capa donde se impriman las aerovías con sus respectivos *waypoint* que pasan por la Comunidad Valenciana a partir de un archivo *kml* donde se encuentra la información para pintarlas.

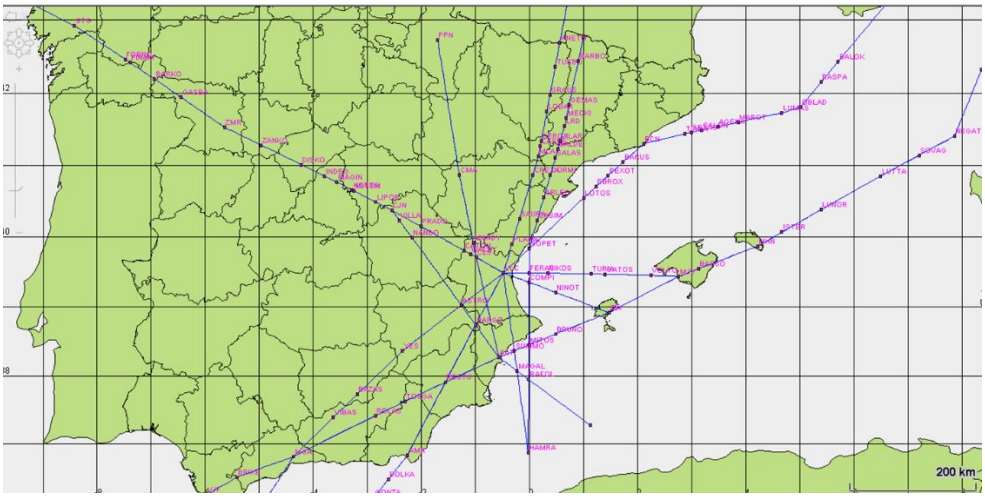

Figura 41: Route Layer

Para ello, haremos uso de una gran variedad de objetos *OMGraphics*, los cuales aprovecharemos para crear métodos que impriman los *waypoints* y las aerovías. Los explicaremos en el mismo orden que se van creando en el código.

Comenzamos con el método *createLine(double lat1, double lon1, double lat2, double lon2, color color)*, el cual devuelve un objeto del tipo *OMLines*, que une dos coordenadas expresadas en *lat/lon*. La línea que une ambos puntos es del tipo *LineType_GreatCircle*. La razón por la cual se ha escogido este tipo, es porque una aeronave debe intentar siempre recorrer la menor distancia posible entre dos *waypoints* y geográficamente esta será una línea de tipo ortodrómica.

El siguiente método es *createPoint(double lat1, double lon1, Color color)*, el cual devolverá un objeto del tipo *OMRect*, es decir, este método se encargará de representar los *waypoints* mediante pequeños cuadrados. Para representar estos cuadraros se ha decido usar líneas del tipo *LineType_Straight*, para que la forma del cuadrado no sea modificada si cambiamos de proyección.

createLabel(double lat, double lon, String text, Paint color) se encargará de devolver un objeto del tipo *OMText*, encargado de imprimir al lado de cada *waypoint* su nombre.

createGraphics(OMGraphicList graphics) no devuelve nada, solo se encarga de lanzar el método *parseFeature(OMGraphicList graphics, String file)* para cada uno de los archivos *kml* que tenemos para imprimir nuestra rutas.

Para leer estos archivos tenemos varios métodos:

- *parseFeature(OMGraphicList graphics, String file):* busca dentro de la estructura del documento placemark.
- *parseGeometry(Geometry geometry):* extrae del documento la geometría.
- *parseDescription(OMGraphicList graphics, String description):* extrae del documento etiquetas de los waypoints.
- *extractData(String description, String label):* para extraer del documento la etiquetas de los waypoints se usa esta función para encontrar la información deseada.

Estos métodos son realmente importantes, imaginemos que tenemos un archivo *kml* de GoogleEarth con *placemarks*, simplemente se lo pasamos como parámetro a nuestro programa y podríamos verlo impreso en nuestro mapa.

Las líneas, puntos y texto, han sido generados por, los métodos explicados, pero para que sean visibles debemos añadir estos objetos a nuestro OMGraphicList, los métodos encargados son:

- *paintLine(double lat1, double lon1, double lat2, double lon2, Color color)*
- *paintPoint(double lat1, double lon1, Color color)*
- *paintText(double lat, double lon, String text, Color color)*

Terminando con esta clase tenemos el método *paint()*, que como se había explicado en Java es el que se encarga de imprimir en pantalla y para el caso particular todos los objetos contenidos en *OMGraphicList*.

Nota: el *paint* aquí nombrado no es exactamente el propio de Java, se explicará en la siguiente capa, pues resulta un ejemplo más interesante, debido a que también entra en juego *repaint*.

Clase RealTimeAirplanesLayer

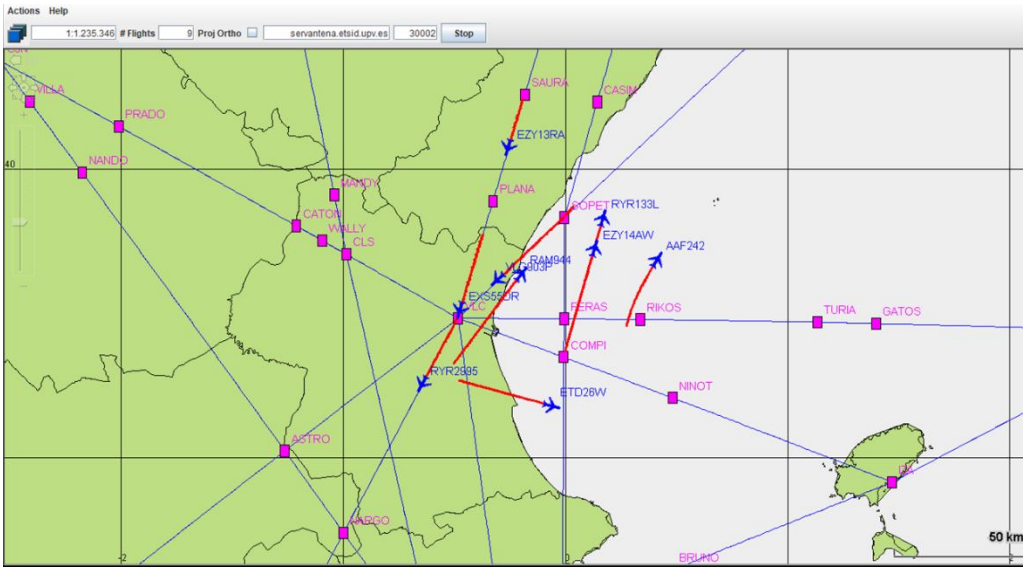

Figura 42: RealTimeAirplanes Layer

Esta capa es la encargada de pintar las aeronaves que la antena recibe. Todo ello se hace en tiempo real y de ahí el nombre de esta clase. Esta capa lanza la clase *AntennaReceiverListener* para conectarse a la antena y recibir los vuelos a pintar. Posee métodos importantes como *Start(String host, int port)*, *Stop()* y *Running()*, que permiten justamente conocer el estado de la antena, así como conectarse o desconectarse de ella. Métodos a los que se llamará desde la *GUI*, cuando en el *ToolPanel* se clique sobre *Start* o *Stop*.

Esta capa pinta por un lado las aeronaves y si clicamos sobre alguna de ella, pintará una línea roja que irá mostrando el camino que está siguiendo, es decir, su track. Para ello, crearemos los métodos pertinentes a partir de deferentes clases de *OMGraphic*, métodos a los que estamos obligado por la interfaz *printAirplanes (mirar captura de código 17).*

Comencemos con el método *createLabel(double lat, double lon, String text, Paint color, int justification)*, el cual se encargará de imprimir en una etiqueta, por tanto devuelve un objeto del tipo *OMText*.

El método *createPolyline(double[] latLon, Color color)* devolverá un objeto de tipo *OMPoly*, que en esencia es un línea poligonal creada a partir de varios puntos.

El método *createImage(double lat1, double lon1, double track, boolean logged)* devolverá un *OMRaster*. Método usado para imprimir una imagen PNG en pantalla, para ser más exactos el icono de la aeronave. Negro o Azul en función de si hemos clicado o no sobre ella para ver su *Track*. Un pequeño detalle que se ha tenido en cuenta pero que es muy importante es la orientación del icono, es decir, que posicionará la aeronave apuntado en la dirección en la que se desplaza. Esto lo podemos hacer gracias a que disponemos de la antena su *track* y simplemente usando la siguiente instrucción orientamos el ícono, *bitmap.setRotationAngle(Math.PI * track / 180)*.

Ahora bien, los métodos explicados únicamente creaban los objetos, son los siguientes quienes los imprimen en el mapa:

- *paintAirplaneText(MapBean bean, FlightObject fo, Color color)*: obtendrá el *Callsign* o en su ausencia el *HexIdent* de *FlightObject* y llamando a *createLabel* lo imprimirá al lado de las coordenadas de la aeronave.

- *paintAirplaneImage(MapBean bean, FlightObject fo)*: obtiene las coordenadas correctas así como el *track* para llamar a *createImage* e imprimir la aeronave.

- *paintLog(FlightObject fo, Color color)*: imprimirá el camino seguido por la aeronave llamando a *createPolyline*. Esto lo hace gracias a que en *FlightObjetct* existe una lista en la que se van guardando todos los puntos o coordenadas por donde pasar la aeronave, *createPolyline* lee esto puntos y los va uniendo a medida que se van creando nuevos, formando así una línea poligonal del camino que ha seguido la aeronave.

- *paintAirplanes(MapBean bean, FlightObject fo[], Color colorTexto, Color colorLog)*: simplemente llama a los métodos anteriores para imprimir lo explicado pero para todas las aeronaves.

Los métodos *paintAirplaneText*, *paintAirplaneImage* y *paintLog* llaman al final de los mismos al método *add* de *OMGraphicList*, de lo contrario no serían visibles en pantalla.

Como se había explicado en los conceptos básicos de Java, no es solo imprimir un objeto una vez, sino que se debe reimprimir cada vez que se produzca una modificación. Esto lo hacemos por medio de *toRepaint()*. Entender que *toRepaint* lo que hará en primer lugar es generar los nuevos objetos, es necesario hacer esto antes de pintarlos, por lo tanto, *toRepaint* llama primero a *generate* y después llamará a *repaint*. Ahora, *repaint* ha sido sobrescrito para OpenMap, con lo se quiere decir que no llamar a paint, sino a render método similar a *paint*, pero propio de OpenMap. Esto

debemos hacerlo así, pues es el modo en que trabaja esta plataforma. Resumiendo, el *repaint* y *paint* de Java pasan a ser *generate* y render en OpenMap.

Para terminar con esta capa hablar de *ChangeNotification(AntennaReceiverListener antenna)*, es de vital importancia, pues cuando la antena reciba nuevos datos, usará este método para reimprimir la capa.

Clase TrackLayer

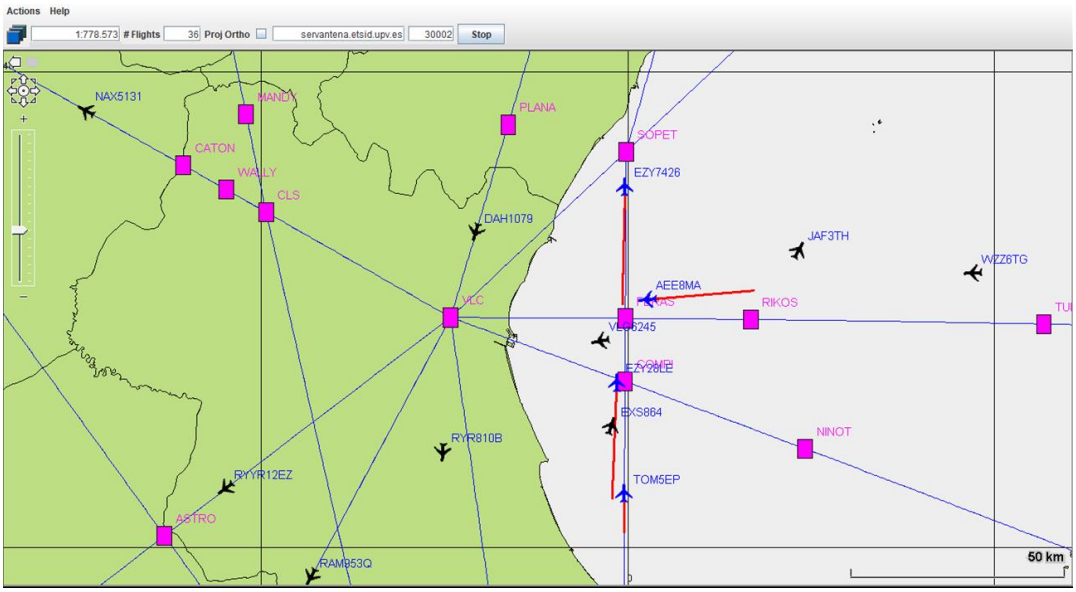

Figura 43: Track Layer

Esta es la última capa de nuestra aplicación y, por tanto, la que se encuentra encima. Su trabajo es representar a partir de un archivo *txt*, vuelos grabados, por tanto, para esta capa es de vital importancia la clase *TrackReader* del paquete Track_*ReadGenerator*.

Como esta capa tendrá en cuanto a la parte gráfica un comportamiento idéntico a la capa *RealTimeAirplanes* dado que implemente la interfaz *printAirplanes* y define cada uno de los métodos de la misma manera, por lo tanto, con intención de no duplicar información para ver estos métodos y su explicación, diríjase al punto anterior.

Lo último que podríamos destacar es *ChangeNotification(TrackReader player)* que como podemos observar ahora tiene como atributo el objeto *player* de *TrackReader* y en *RealTimeAirplanes* tienes a la *AntennaListener*, pero su función es la misma, repintar por completo la capa cuando se produzca una notificación, que esta vez será dada por *TrackReader*.

5.2.7. Paquete Traffic

Este paquete solo contiene la clase *FlightObject*, que para esta nueva aplicación incluye dos métodos nuevos propios de la clase y una clase interna *Point()* con dos constructores y dos métodos.

Respecto a los dos nuevos métodos:

- *updateAll(FlightObject fo):* encargado de actualizar absolutamente todos los campos de *FlightObject*.
- *updatePos(Double lon, Double lat)*: actualiza la posición si el método 'diff', explicado a continuación es diferente a 0.

La clase interna *Point()* ha sido creada para almacenar la posición en sus valores reales, normalizados y calcular la diferencia entre el valor almacenado y el nuevo introducido. Para ello, dispone del método *diff(Double lon, Double lat)* o *diff(Point p)*

5.2.8. Paquete MyMenuBar

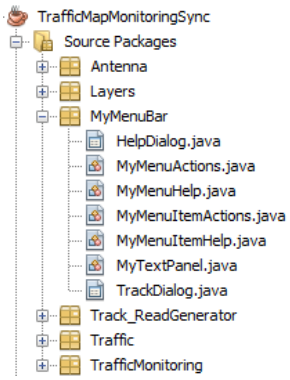

Figura 44: paquete MyMenu

Como se había visto en la Figura 27, se había creado un *JMenuBar* con los menús *Actions* y *Help* y justo debajo de la barra de menús, como se mostró en la Figura 26 un *ToolPanel* llamado *MyTextPanel*. Son en este paquete donde se crean y definen.

Comenzamos con el *ToolPanel* que está enteramente definido en la clase *MyTextPanel* y continuaremos con los menús *Actions* y *Help* que cada uno se forma de un conjunto de tres archivos.

Clase MyTextPanel

Figura 45: MyTextPanel

En primer lugar, explicar una serie de declaraciones simples para poder visualizar varios componentes en nuestra GUI.

```
protected JTextField nFlights = null;
protected javax.swing.JLabel textLabel = null;
protected javax.swing.JCheckBox projection = null;
protected javax.swing.JLabel checkLabel = null;
protected JTextField host = null;
protected JTextField port = null;
protected javax.swing.JButton startStop = null;
protected TrafficRadar father;
```

Captura código 18: componentes para MyTextBar

Con la captura de código 18 vemos que, para imprimir texto, *checkboox*, *labels*, *textfiels* o botones, son los propios componentes de java a los que se llaman, no a ninguno propio de OpenMap. Así mismo, los métodos para alinear modificar estos componentes son enteramente código de Java. Se puede ver en la captura de código 19.

```
String infoText = "# Flights";
String infoLabel = "Proj Ortho";
nFlights = new JTextField("0", 4);
nFlights.setToolTipText(infoText);
nFlights.setMargin(new Insets(0, 0, 0, 0));
nFlights.setHorizontalAlignment(JTextField.RIGHT);
textLabel = new javax.swing.JLabel();
textLabel.setText(infoText);
checkLabel = new javax.swing.JLabel();
checkLabel.setText(infoLabel);
projection = new javax.swing.JCheckBox();
```

Captura código 19: propiedades de los componentes de MyTextPanel

No compele poner captura de código de todas las inicializaciones. Decir que por defecto introduciremos mediante *JTextField*, el host como "servantena.etsid.upv.es", el puerto como "30002" y el número de vuelos como "0".

Muy importante decir, que tanto *JButton* como *JCheckBox* poseen *ActionListener*, para detectar cuando han sido clicados. Respecto al *JCheckBox* al clicar sobre el pasaremos de una proyección Mercator a una Ortográfica. Usar estas proyecciones es realmente sencillo, pues es todo competencia de OpenMap, como se puede ver en la captura de código número 20. Nosotros solo llamamos a estos objetos y ellos no solo se encargan de los cálculos de cambio de proyección sino también de la

parte gráfica. Se aprovecha esta oportunidad para remarcar una ventaja fundamental de OpenMap y es justamente esta, no hay necesidad de hacer cálculos pesados o algoritmos para poder pasar de una proyección a otra, no es necesario perder tiempo pasando de un sistema de coordenadas a otro o incluso de 'x' sistema de coordenadas a pixeles para imprimirlo en pantalla, pues OpenMap se ha creado con ese fin. Nosotros como programadores como bien se ilustra en este ejemplo solo tenemos que llamar a la instrucción que corresponda y ella hará el resto.

```
projection = new javax.swing.JCheckBox();
projection.addActionListener(new ActionListener(){
public void actionPerformed(java.awt.event.ActionEvent evt) {
    System.out.println(projection.isSelected());
    OrthographicLoader ol = new OrthographicLoader();
    ProjectionFactory pf = new ProjectionFactory();
    pf.addProjectionLoader(ol);
    if (projection.isSelected()){
    father.mapBean.setProjection(pf.makeProjection("com.bbn.openmap."
            + "proj.Orthographic", father.mapBean.getProjection()));
     }
    else{
    father.mapBean.setProjection(pf.makeProjection("com.bbn.openmap."
            + "proj.Mercator", father.mapBean.getProjection()));
      }
   }
});
```

Captura código 20: cambiar de proyección al clicar sobre el CheckBox

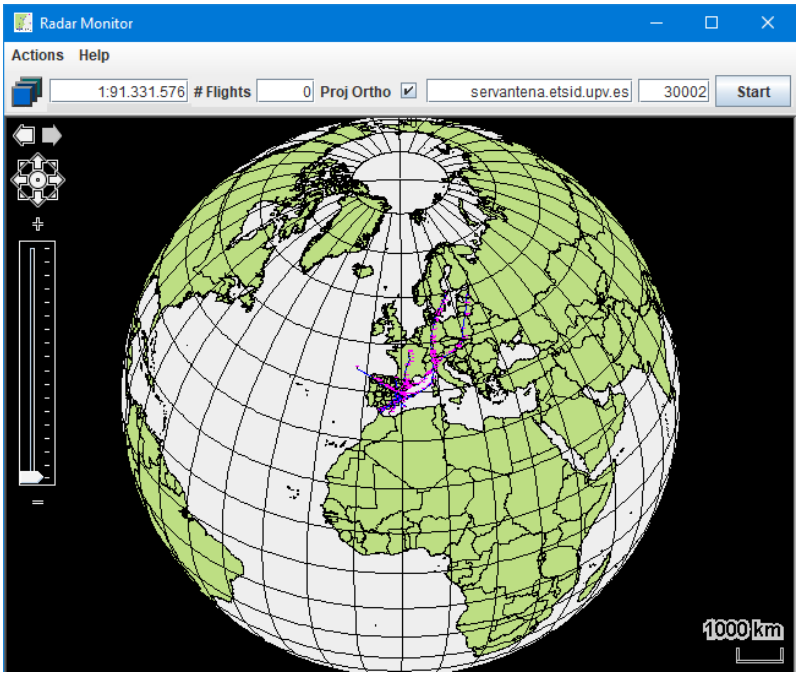

Figura 46: proyección ortográfica de OpenMap

Mencionar que siguiendo la misma filosofía de la captura de código 20 el *JButton* reacciona cuando se da el evento y se conecta a la antena o desconecta de la antena en función del estado previo. Por último, el *TextField* de número de vuelos, mostrará el número de vuelos que está imprimiendo nuestro programa en el mapa. Ese *TextField* es escrito por el método *setnFlights(String text)* que recibirá su *string text* de las capa *TrackLayer* o *RealTimeAirplanesLayer* cuando se ejecute *ChangeNotification*. En otras palabras, es *ChangeNotification* quien introduce el número de vuelos en ese *TextField*. Si las dos capas están mostrando vuelos, el *TextField* se intercalará entre uno y otro para poder mostrar ambos valores.

Clase MyMenuActions, MyMenuItemAction y TrackDialog

En la barra de menú encontramos como primera opción *Actions* que esta está compuesta por *MyMenuActions*, clase encargada de crear el propio menú de *Actions* y al pichar en ella llamará a la clase *MyMenuItemActions*, la cual simplemente crea el ítem *Record/Play Tracks* dentro del menú *Actions*. Al pinchar en él lanza *TrackDialog*, una ventana independiente la cual se encargará de reproducir o grabar vuelos.

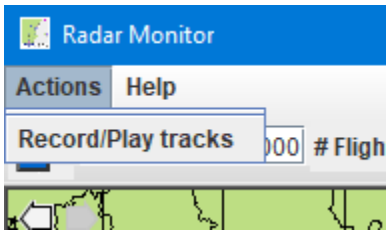

Figura 47: MyMenuActions y MyMenuItemActions

Figura 48: TrackDialog

Para la creación de estos menús, no es necesario ilustrar código. Solo mencionar que para la clase *MyMenuActions* esta debe extender de *AbstractOpenMapMenu* y usando simplemente el método *setText* podemos ponerle en nombre deseado.

Para *MyMenuItemAction* debemos extender de *JMenuItem* y tener en cuenta que se le debe añadir *actionPerformed(ActionEvent e)* para detectar cuando se clica encima para lanzar el *TrackDialog*. El resto de código de estas dos clases es trivial.

En lo que respecta a *TrackDialog*, la interfaz en sencilla, nada que no se haya explicado con anterioridad y respecto a los métodos sí compele explicarlo brevemente.

En primer lugar, se encuentran dos métodos para grabar los vuelos que la antena está recibiendo, *open_tracker()* y *close_tracker()*. Respecto a *open_tracker()* básicamente es una función que abre una nueva ventana para elijamos donde queremos guardar el archivo que se a crear (Figura 49) y llama a la clase *TrackGenerator* la cual grabará los vuelos. Como vemos en la Figura 50 también podemos elegir el tiempo a grabar en segundos, sin máximo de tiempo y con la posibilidad de detener la grabación en cualquier momento pulsando Stop (Figura 50). Stop activará un evento de ratón que llamará a *close_tracker()*, encargado de detener a *TrackGenerator*.

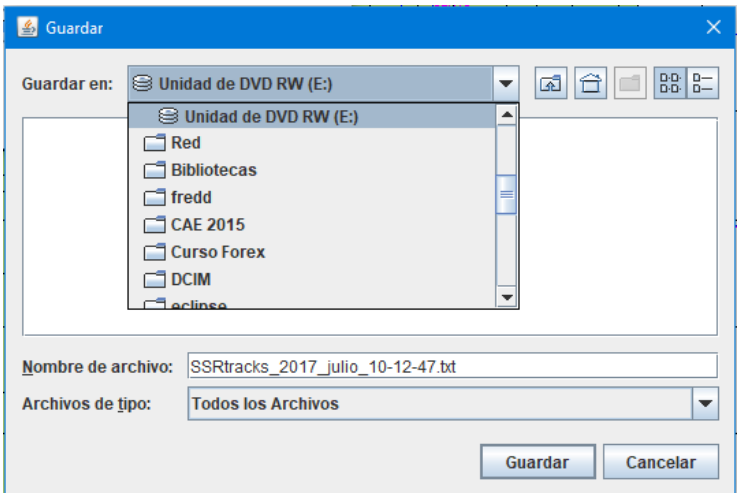

Figura 49: open_tracker

Para el nombre del archivo usaremos el método *java.util.Date()* para obtener la fecha y la hora, se lo pasaremos a *SimpleDateFormat("yyyy_MMMMM_dd-hh-mm")* de tal manera que concatenándolo con el *string SSRTracks_* tendremos por defecto un archivo con la fecha y la hora en la que se grabó. Mirar nombre de archivo (Figura 49).

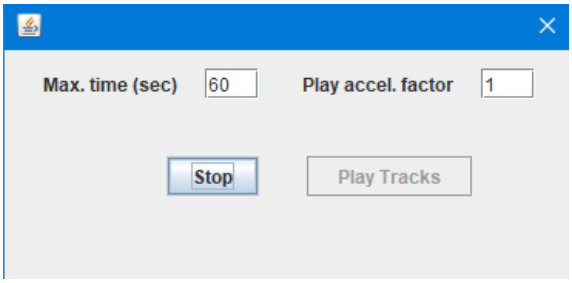

Figura 50: Stop para detener la grabación

En segundo lugar, tenemos dos métodos más encargados de la reproducción de dichos archivos grabados por *TrackGenerator*, son *open_player()* y *close_player()*. De la misma manera que los dos métodos anteriormente explicados, *open_player()* y *close_player()*, se encargan esencialmente de lo mismo. Ahora bien, *open_player()* llamará a *TrackReader* en vez de, *TrackGenerator* el cual leerá estos archivo para su reproducción y *close_player()* permitirá detener la reproducción en cualquier momento. No se mostrarán capturas de pantalla del cuadro de dialogo para elegir el archivo a reproducir o del botón Stop para detener la reproducción pues se evita sobrecargar este documento, además de ser prácticamente idénticos a las Figuras 49 y 50.

Nota: el cuadro de dialogo de la Figura 49, es propio de Java, basta con llamar a *JFileChooser()*.

Clase MyMenuHelp, MyMenuItemHelp y HelpDialog

Para ofrecer un guion, una pequeña ayuda sobre OpenMap, se ha decido incluir en la barra de menú la opción *Help*. Esta está compuesta por MyMenuHelp, clase encargada de crear el ménu de *Help* en la barra de Menú, y al pichar en ella llamará a la clase *MyMenuItemHelp*, la cual simplemente crea el ítem *Help/Ayuda* dentro del menú *Help*. Al pinchar en él lanza *HelpDialog*, una ventana independiente con ayuda sobre OpenMap como muestra la Figura número 51 y 52.

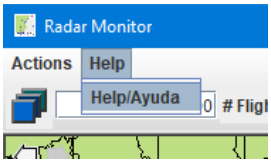

Figura 51: MyMenuHelp y MyMenuItemHelp

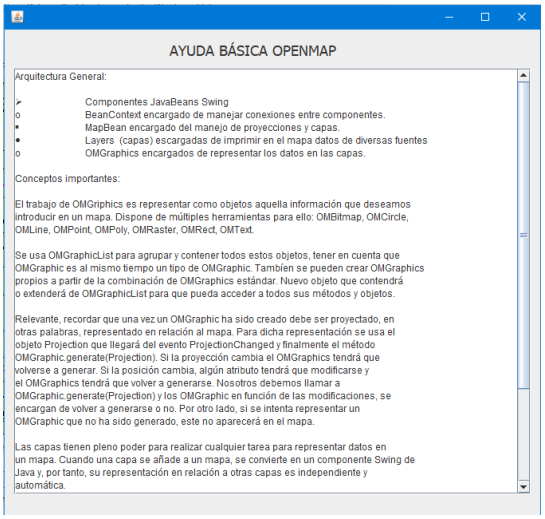

Figura 52: HelpDialog con ayuda sobre OpenMap

5.3. OpenMap + Netbeans

Respecto a la plataforma OpenMap, este trabajo se ha centrado en aprender a programar desde su 'entorno', es decir, usar las herramientas de las que esta dispone para la creación de una GUI. Sin embargo, OpenMap no está limitada a la creación únicamente usando sus herramientas. De hecho, ese es uno de los motivos por los cuales se eligió Netbeans para la creación de este proyecto, pues como ya se ha explicado Netbeans dispone de una muy capaz herramienta para la creación de interfaces gráfica de manera sencilla y solvente. En pocas palabras, lo que se pretende expresar es la posibilidad de crear una aplicación gráfica desde el asistente de Netbeans que incorpore OpenMap. De esta manera podríamos tener todas las ventajas y facilidades que ambas herramientas incorporan.

Como ejemplo de esta posibilidad, se ha creado una pequeña aplicación para ilústralo llamada *DemoNetbeansOpenMap*. Es una aplicación sencilla, pero cumple con su finalidad, de la manera que podemos ver en la captura de pantalla siguiente.

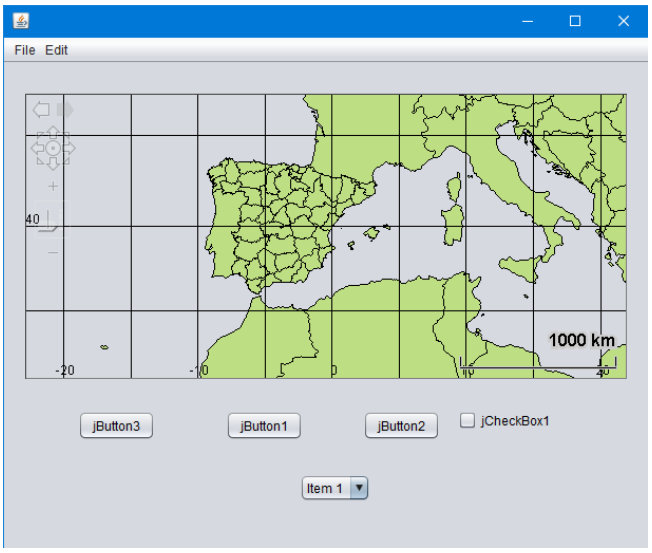

Figura 53: integración de OpenMap con Netbeans

Es reconocible el mapa y widgets en la parte superior izquierda propios de OpenMap. Sin embargo, ahora OpenMap se ha utilizado como un *jPanel* dentro de un *JFrame* creado por el asistende de Netbeans. Los *jButton*, *jCheckbox* y *jComboBox* han sido creado por el asistente. Podríamos decir que es una mezcla entre las primeras interfaces creadas con el asistente de Netbeans y la última creada con OpenMap.

Para la realización de esta última aplicación se ha reutilizado código de *TrafficMonitorinSync*, no compele explicar el código creado, ni introducir capturas del mismo, pues es solo un ejemplo que pretendía demostrar las afirmaciones hechas.

6. Conclusión

A la especialidad de aeronavegación del grado de ingeniería aeroespacial, le compele el estudio y desarrollo de todas aquellas actividades relacionadas con la conducción eficiente de una aeronave a su lugar de destino, asegurando la integridad de los tripulantes, pasajeros, y de los que están en tierra. Es por tanto de gran importancia como ingeniero de este campo conocer que métodos son los mayormente usados hoy en día, así como futuras tendencias. Uno no deja de aprender nunca y menos en un campo estrechamente relacionado con la alta tecnología. Por ello, junto a mi creencia de que todo ingeniero debe ser capaz de crear sus propias herramientas de trabajo, llámese en este caso crear sus propios programas, me ha llevado a este trabajo fin de grado, que obliga al redactor a saber dominar competencias nuevas, en este caso un nuevo lenguaje de programación, así como a haber estudiado, comprendido y programado un radar basado en el sistema de aeronavegación conocido como SBS. Además, ha aportado una gran herramienta llamada TrafficMapMonitoringSync, un nuevo radar basado en los anteriores diseños, pero implementando una interfaz verdaderamente gráfica, es decir, con mapas, diferentes capas y herramientas varias. Esto se ha conseguido gracias a la investigación e implementación en nuestro trabajo de la plataforma OpenMap, herramienta aún desconocida, pero con mucho potencial, pues sobre todo en nuestro campo OpenMap hace que nos olvidemos de pasar de coordenadas en latitudes y longitudes a pixeles de pantalla, esta es su gran ventaja, además claro de las diferentes proyecciones que incluye tipos de líneas entre otras herramientas que resultan realmente propicias para nuestra especialidad.

La idea que se ha perseguido con esta aplicación, es crear una herramienta de estudio y ayuda al departamento de aeronavegación de nuestra escuela. De esta manera, se deja en sus manos un proyecto con gran potencial de desarrollo. Me explico, la misma aplicación se ha creado con esa filosofía, inclusive el propio lenguaje java sigue este camino, el trabajo cooperativo, en el cual se pueden crear grandes proyectos a partir del trabajo de todos, contribuyendo como comunidad. Java permite esto, gracias a su orientación a objetos y la facilidad de reutilizar código y seguir creando sobre las bases que se nos ha dado y justo aquí entra OpenMap, que por un lado es de vital relevancia su aspecto de libre uso y difusión. Por otro, que OpenMap no solo se limita a crear aplicaciones Java para escritorio, sino que va mucho más allá. OpenMap puede ser usado también Applet, Image Server o componente.

El trabajo realizado ha sido arduo divido en dos grandes fases, uno de aprendizaje tanto por el lenguaje Java como por la plataforma OpenMap y, por otro lado, de programación. Además, se ha organizado y restructurado, el material del que se disponía. Se ha limpiado, comentado e incluido nuevas funciones, ya sean desde menús, ayudas, proyecciones o

capas. Se ha demostrado con un sencillo ejemplo que se puede usar OpenMap como componente. Se incluye en anexos una resumen más extenso y profundo sobre funcionalidades que se han tocado y otras que no, para posibles interesado en el tema.

Ya se ha visto la usabilidad de esta herramienta para nuestro campo, la facilidad con las que nos permite incluir nuevas capas a partir de ficheros tipo shape, las proyecciones recalculadas automáticamente y en eso lo que precisamente se buscaba una nueva herramienta que sea sencilla, clara y escalable, que asiente firmes bases para algo mucho más grande que pueda ser desarrollado ya sea por alumno o profesores.

7. Ampliaciones

Como se ha explicado, este trabajo crea un marco para futuros proyectos. Y es precisamente el carácter abierto y versátil de OpenMap junto a Java el que ofrece grandes posibilidades. Las ideas y posibilidades solo quedan limitadas por la imaginación.

Por ejemplo, respecto a la navegación aérea se podrían introducir aspectos importantes como una capa de sectorización del espacio aéreo a partir de datos de Eurocontrol, e introducir y procesar planes de vuelo con las rutas previstas antes de ser voladas. Esto permitirá ver las discrepancias entre la planificación estratégica a largo plazo y la planificación táctica. Y esto podríamos unificarlo junto a trabajos que se han estado haciendo al mismo tiempo que el desarrollo de este trabajo fin de grado como en el caso de las comunicaciones que se han tenido varios avances. Pues respecto a la antena, ahora se está usando el formato BEAST que nos permite procesar más tipo de mensajes, posicionar más aeronaves y soportar más tipos de antenas. Así, se ha empezado ya a integrar datos de varias antenas, concretamente ya tenemos disponible el tráfico de la zona de Barcelona. Se espera tener más colaboradores en breve.

Eso por el lado de aplicación de escritorio, pero no nos olvidemos de las aplicaciones móviles. De hecho, todo esto se podría hacer para Android, que usa el mismo lenguaje de programación, Java. Las posibilidades son infinitas. Desde una aplicación de monitorización a una de gestión del tráfico aéreo o incluso podría llegar a crearse un propio ACAS-XP, pues tendríamos un sistema operativo Android, con una aplicación capaz de posicionar el resto de vuelos respecto al mismo, pues como bien sabemos todo móvil dispone de GPS. Puede que sea algo realmente complejo y no se pueda dar uso comercial, pero para estudio y desarrollo, es más que viable un proyecto de esta envergadura.

Desde pequeñas funciones que se pueden introducir, como cálculos de distancias entre vuelos e imprimirlos en el mapa, a llevar esto a ser un verdadero sistema ACAS-XP, gracias a esta aplicación de monitorización, se asienta una base para futuros desarrollos de la comunidad aeronáutica de la Escuela Técnica Superior de Ingeniería del Diseño.

8. Bibliografía

Java

http://definicion.de/java/
https://docs.oracle.com/javase/7/docs/api/

Tutoriales

https://www.youtube.com/watch?v=coK4jM5wvko&list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk
https://www.youtube.com/watch?v=Z0F7sJaOQtw&list=PL602060AB32FC864B

OpenMap:

http://openmap-java.org/

SBS

http://woodair.net/sbs/Article/Barebones.htm

Codificación SBS

http://woodair.net/sbs/Article/Barebones42_Socket_Data.htm

FlightRadar24

https://www.flightradar24.com/how-it-works

FlightAware:

https://es.flightaware.com/about/

PlaneFinder:

https://planefinder.net/

Estado del arte

http://www.infovuelos.info/
https://es.flightaware.com/

Recursos dela asignatura de master asignatura Sistemas de Gestión de Vuelo por Computador

Recursos dela asignatura de grado asignatura Ingeniería de los Sistemas de Navegación Aérea.

1. Guía OpenMap

1.1. Resumen OpenMap

Arquitectura General:

- Componentes JavaBeans Swing
  - BeanContext encargado de manejar conexiones entre componentes.
    - MapBean encargado del manejo de proyecciones y capas.
      - Layers (capas) escargadas de imprimir en el mapa datos de diversas fuentes
        - OMGraphics encargados de representar los datos en las capas.

Conceptos importantes:

El trabajo de OMGriphics es representar como objetos aquella información que deseamos introducir en un mapa. Dispone de múltiples herramientas para ello: OMBitmap, OMCircle, OMLine, OMPoint, OMPoly, OMRaster, OMRect, OMText.

Se usa OMGraphicList para agrupar y contener todos estos objetos, tener en cuenta que OMGraphic es al mismo tiempo un tipo de OMGraphic.

También se pueden crear OMGraphics propios a partir de la combinación de OMGraphics estándar. Nuevo objeto que contendrá o extenderá de OMGraphicList para que pueda acceder a todos sus métodos y objetos.

Relevante recordar que una vez un OMGraphic ha sido creado debe ser proyectado, en otras palabras, representado en relación al mapa. Para dicha representación se usa el objeto Projection que llegará del evento ProjectionChanged y finalmente el método OMGraphic.generate(Projection). Si la proyección o la posición cambia el OMGraphics tendrá que volverse a generar. Nosotros debemos llamar a OMGraphic.generate(Projection) y los OMGraphic en función de las modificaciones, se encargan de volver a generarse o no. Por otro lado, si se intenta representar un OMGraphic que no ha sido generado, este no aparecerá en el mapa.

Las capas tienen pleno poder para realizar cualquier tarea para representar datos en un mapa. Cuando una capa se añade a un mapa, se convierte en un componente Swing de Java y, por tanto, su representación en relación a otras capas es independiente y automática.

Cuando añadimos una capa al MapBean, por ende, esta pasa a ser un ProjectionListener del mismo MapBean. Por lo cual, al producirse una modificación en el mapa nuestra capa recibirá el evento ProjectionChanged y ya es competencia de ella decidir qué debe hacer al respecto y llamar a repaint() cuando esté lista.

Para el manejo de capas, OMGraphicHandlerLayer realiza un gran trabajo. Toda capa que creemos debería extender de esta clase y simplemente sobrescribir el método prepare() para crear y devolver un OMGraphic que queramos imprimir en nuestra capa.

1.2. Aplicación OpenMap

OpenMap dispone de un componente de aplicación (com.bbn.openmap.app.OpenMap) que crea el marco de referencia (framework) básico para configurar y ejecutar una aplicación de mapa. Este marco es un contenedor para todos los componentes que añadamos a la aplicación.

Este marco provee de las herramientas para que los componentes se enlacen o conecten entre ellos y así fácilmente formar la base de nuestra aplicación con diferentes interfaces y funcionalidades.

Una aplicación de OpenMap se configura mediante un archivo openmap.properties. Los contenidos de este archivo especifican que componentes han sido creados y añadidos a nuestro marco de referencia, incluyendo las capas. Nuevas aplicaciones pueden ser configuradas solamente modificando este archivo desde un editor de texto, sin necesidad de recompilarlo. Los componentes pueden ser modificados de la misma manera, accediendo a este archivo.

Respecto a las capas en OpenMap pueden usar información o datos de diferentes maneras cuando se trata del entorno de una aplicación. Se pueden crear o añadir características a mapas:

- Por computación.
- Desde la lectura de archivos de datos directamente desde un disco duro local.
- De leer archivos de datos desde una URL.
- De la lectura de archivos de datos contenidos en un archivo jar.
- Uso de información recuperada de una base de datos (JDBC).
- Uso de la información recibida de un servidor (imágenes u objetos de mapa)

1.3. Uso general de las proyecciones

Una de las ideas de ser de OpenMap es que el usuario no tenga que preocuparse la localización de componentes en un mapa.

Entendemos por proyección en el contexto geoespacial como la translación entre un modelo esférico de la tierra a uno plano (de superficie expresada en dos dimensiones). En función del estudio, una proyección u otra resultará más ventajosa, ya sea, visualmente o por los cálculos a realizar. En el contexto de OpenMap cuando hablemos de proyecciones entenderemos componentes que se encargan de la translación de coordenadas en latitud y longitud a ubicaciones en pantalla.

Las proyecciones de OpenMap solo trabajan de latitudes/longitudes en grados decimales a coordenadas x/y. También a tener en cuenta, que esta plataforma usa como datum el WGS84.

Las proyecciones de OpenMap no solo sirven para pasar coordenadas de un sistema a otro, sino las formas(cuerpos) de los mapas se definen como pares de coordenadas de puntos, conocidos como vectores característicos.

Las proyecciones son capaces de determinar cómo los vectores característicos, definidos en términos de longitud y latitud, se representan así mismos. La geometría de estas formas (cuerpos) depende del tipo de línea elegido o especificado. OpenMap maneja tres tipos de líneas:

- Great Circle lines (línea ortodrómica): la línea entre dos coordenadas es la distancia más corta entre estos puntos en la superficie de la Tierra.

- Rhumb lines (línea loxodrómica): la línea entre dos coordenadas tiene un rumbo constante. Mientras se mueva sobre esta línea de un punto a otro, se mantendrá la misma dirección.

- Straight (recto). Las líneas entre coordenadas dibujadas en el espacio de píxeles

OpenMap también puede representar datos raster (imagenes) en un mapa, pero se debe entender que las imágenes también tienen sus propias proyecciones. Para que una imagen de base raster se utilice con precisión en OpenMap, los parámetros de proyección del raster deben coincidir con los parámetros de proyección de OpenMap.

Nota: las proyecciones se basan en modelos terrestres esféricos. No tienen en cuenta el aplanamiento elipsoidal (excentricidad).

1.4. Manejo de eventos

Los componentes de OpenMap se comunican entre sí usando el modelo de eventos de Java. Se considera el modelo conocido.

Hay diferentes tipos de eventos almacenados en el paquete com.bbn.openmap.event. Como ejemplo, el NavigatePanel (com.bbn.openmap.gui.NavigatePanel) que cuando sus botones son presionados genera:

- com.bbn.openmap.event.PanEvents: si se presiona las fechas de NavigatePanel llama a PanEvents contiene información que especifica la dirección y distancia.

- com.bbn.openmap.event.CenterEvents: si se presiona el botón central o del medio, se llama a CenterEvents que contiene la información de la localización (longitud y latitud) .

MapBean (com.bbn.openmap.MapBean) (véase su definición en glosario), que es una de los componentes de ventana principales de OpenMap, implementa las interfaces PanListener y CenterListener.

1.5. Arquitectura de los componentes

1.5.1. MapBean, MapHandler y MapPanel

MapBean es de vital importancia en OpenMap. Deriva de java.awt.Container y contiene com.bbn.openmap.proj.Projection, objeto que define la posición geográfica en el mapa representado por el propio MapBean. La proyección se define mediante una combinación del tipo de proyección, la latitud y longitud del centro de la ventana, el factor de escala de la proyección y la altura y anchura de la ventana en pixeles.

En cuanto a las capas (Layers), derivan del java.awt.Component y son el único tipo de objeto que puede ser añadido a un MapBean. De esta manera, entendemos

que las capas son componentes dentro de un contenedor MapBean. La representación de estas capas dentro de un contenedor MapBean se lleva a cabo por medio del mecanismo de interpretación (representación) de Java. Las capas son independientes una de otras, es decir, trabajan independientemente y evitan que MapBean tenga que estar pendiente de cada capa.

Recopilando, tenemos en cuenta que las capas también son ProjectionListeners por lo tanto, cuando son añadidas a MapBean, estarán atentas a cualquier evento que pueda producirse en el mapa (moviemientos, zoom o reescalar).

Destacada la importancia de MapBean, ahora introducimos MapHandler (com.bbn.openmap.MapHandler) columna vertebral de OpenMap. Entender el funcionamiento de MapHandler es de vital para la personalización de las aplicaciones de OpenMap.

MapHandler es como un contenedor al que se le pueden añadir o quitar objetos de él. Pertenece a Java BeanContext (java.beans.beancontext.BeanContext). MapHandler es usado por aquellos componentes que necesiten manejar otros objetos o servicios. Si es el caso, dicho componente debe ser notificado a BeanContext. Para ello, solo se debe hacer que sea BeanContextChild. Si el componente es de creación propia basta con simplemente añadir el BeanContextChild manualmente al MapHandler.

Nota: como MapHandlerChild, el componente puede ser añadido a OpenMap sin necesidad de recompilar el código.

La interfaz MapPanel (com.bbn.openmap.gui.MapPanel) es un compoente que contiene un MapBean, MapHandler y una serie de menús que puede considerarse como una JMenuBar o JMenu con sub-menus.

BasicMapPanel (com.bbn.openmap.gui.BasicMapPanel) es una implementación de la interfaz MapPanel, es usada por defecto en OpenMap. Es una extensión de JPanel con un BorderLayout.

1.5.2. LayerHandler

Para el manejo de capas (Layers) OpenMap dispone del objeto LayerHandler. De las capas disponible veremos aquellas que tengan el atributo Layer.isVisible() activado. Además, LayerHandler dispone de diversos métodos para añadir, quitar o cambiar el orden de las capas y enviar eventos a los Listeners.

1.5.3. MouseEvents

Cuando se realiza alguna acción de ratón sobre un objeto visible en la ventana de la aplicación se produce un MouseEvent (evento de ratón). MouseEvents se encarga de describir que tipo de acción se ha realizado, ya sea, mover el ratón, clicar, soltar, arrastar, entrar o salir de la Ventana y además de dónde se ha sucedido dicho evento.

A estos eventos se puede reaccionar de diferentes maneras. Usaremos MapMouseEvents para describir el comportamiento a estos eventos. MouseDelegator es el responsable de controlar que MapMouseMode es MouseListener y MouseMotionListener, es decir, maneja una lista de MouseModes y sabe cuál está active en cualqueir momento.

1.5.4. Controladors GUI, ToolPanel y Menus

El toolkit de OpenMap contiene variedad de widget que pueden ser usados para controlar el mapa y sus capas. La mayoría se encuentran en com.bbn.openmap.gui.package.

Entre varios destacamos ToolPanel, que es un JToolBar usado para emplazar icono y controles simples sobre el mapa. OMToolComponent es una clase muy interesante, pues nos permite que cualquier clase derivada de ella esté lista para poder mostrarse en el ToolPanel. Algunos OMToolComponents incluyen:

- NavigatePanel: conjunto de flechas que nos permiten movernos en 8 direcciones en el mapa.
- ZoomPanel: par de botones para acercar o alejar la vista.
- OverviewMapHandler: es una pequeña ventana en la cual se puede visualizar donde estamos en el mapa, desde una perspectiva más amplia.

1.6. Imprimiendo información con OMGraphics

OMGraphics son las clases principales usadas para representar objetos en un mapa. Las capas manejan a OMGraphics como objetos y se encargan de sus proyecciones, es decir, de donde deberían localizarse en el mapa.

1.6.1. Clases OMGraphics, tipos Render, tipos Line

Existe una gran variedad de OMGraphics en función de la forma que se debe representar en el mapa: OMArc, OMCircle, OMGrid, OMLine, OMPoint, OMPoly (incluyendo las subclases OMDistance y OMSpline), OMRasterObject (incluyendo las subclases OMBitmap, OMRaster, OMScalingRaster y OMScalingIcon) OMRect y OMText.

Estos objetos se pueden representar de 3 maneras:

- RenderType_Location: el objeto se introducirá en sus coordenadas de latitud y longitud. Se debería esperar que el objeto se escale automáticamente cuando la escala del mapa cambie.

- RenderType_XY: el objeto se introducirá en una localización dada en función de los pixeles de la pantalla. El objeto no se moverá o escalaré si la proyección del mapa cambia.

- RenderType_Offser: el objeto se introducirá en una localización dada según pixeles de pantalla en función de un offset de una coordenada en lat/lon. El objeto se moverá si el mapa cambia, pero no se rescalará si la escala varia.

Hay tres tipos de líneas asociadas a OMGraphics que tienen representación en lat/lon:

- LineType_Straight: imprimirá una línea recta entre dos puntos no importa la proyección usada.

- LineType_GreatCircle: imprimirá una línea entre dos puntos, la cual guardará la menor distancia geográfica entre ellos.

- LineType_Rhumb: imprimirá una línea entre dos puntos de constante rumbo (dirección constante).

1.6.2. Tipos básicos OMGraphics

Gracias a los tipos de OMGraphics o a la combinación de ellos, se puede representar prácticamente cualquier tipo de datos sobre el mapa.

1.6.3. OMArc y OMCircle

Con OMCircle podemos representar un círculo o elipse. OMArc representa arcos. Podemos especificar el ángulo y la longitud del arco en grados decimales. OMCircle es una subclase de OMArc.

1.6.4. OMGrid

Objeto que representa o puede contener información en dos dimensiones en espacios igualmente separados, es decir, en una cuadrícula.

1.6.5. OMLine y OMArrowHead

Objeto que representa el camino entre dos puntos sobre el mapa. OMLines puede usar el objeto OMArrowHead y convertirse en fechas. Dichas fechas puede ponerse al final o al principio de la línea.

1.6.6. OMPoly, OMDistance, OMSpline y OMDecoratedSpline

El objeto OMPoly representa un camino creado por diferentes puntos. Este objeto puede ser cerrado (polygon) o abierto (polyline). En caso de ser un poligono se puede pintar su interior.

OMDistance es una representacion de OMPoly en lat/lon etiquetada con la distancia entre dos puntos y la distancia total acumulada.

OMSpline y OMDecoratedSpline implementan un algoritmo para añadir curvas entre los puntos de un polígono. OMDecoratedSpline permite también añadir etiquetas sobre la curva.

1.6.7. OMRect

Objeto que representa un cuadrado dentro de los límites de unas coordenadas.

1.6.8. Familia OMRasterObject

OMRasterObject es una superclase de OMGraphics, padre de OMBitmap, OMRaster, OMScalingRaster y OMScalingIcon. Este objeto es usado para representar imágenes, que se representan de acuerdo al pixel de su parte superior izquierda y se pintan sobre el mapa. Los objetos de este tipo se pueden rotar.

OMBitmap son imágenes de dos colores. Puede ser usado para crear imágenes a partir de archivo de texto mediante la clase XBMFile.

OMRaster es la clase principalmente usada para representar imágenes. Un OMRaster puede ser creado desde información de pixeles en ARGB con formatos como (JPG, GIF, PNG, TIFF).

OMScalingRaster es una subclase de OMRaster que además de permitir especificar la ubicación en el mapa respecto a la parte superior izquierda, también pertime especificar la ubitación de la parte inferior derecha de la imagen.

OMScalingIcon es una subclase de OMScalingRaster. Usada para representar objetos en el mapa. Centra la imagen sobre una localización y escala la imagen en función de la proyección dada. Se le pueden dar valores de escala máxima, mínima para que decida desde donde y hasta cuando escalar.

1.6.9. OMText

Objeto para introducir texto en el mapa. Permite múltiples líneas de texto, rotarlo, justificación entre otras herramientas.

1.6.10. OMGeometryList y OMAreaList

Parecido a OMGraphicList que contiene objetos OMGeometry. OMAreaList es una clase derivada de OMGeometryList que acepta diferntes tipos de OMGeometries o OMGraphics, los une y represetna como una solo área.

1.6.11. OMGraphics y atributos Rendering

OMGraphics dispone de diferentes atributos que nos permiten controlar como ser representan (rendering) en el mapa:

- Line Paint: usada para controlar individualmente el borde o esquina de un OMGraphic.
- Select Paint: usado para seleccionar (select()) o deseleccionar (deselect()), el Line Paint.
- Fill Paint: para pintar en interior de un OMGraphic.
- Texture Paint: permite añadir una textura a un OMGraphic
- Gradiante Paint: permite añadir un gradiente de color o colores.
- Matting y Matting Paint (pintura mate): permite añadir un color a los bordes, cuando temenos un fondo muy saturo.
- Stroke: controla como se dibuja los bordes o esquinas, especificando el grosor de las líneas, patrones dash (pequeñas líneas discontinuas), bordes acentuado u otros diseños personalizados.

1.6.12. Manejo de OMGraphics

Como ya se ha hablado, OMGraphicList siendo un propio OMGraphic sirve de contenedor para los mismos, pero además permite controlar el orden en el cual se representan estos objetos, encuentra que OMGraphic está más cerca de una ubicación o que ubicación está más cerca d un OMGraphic y es capaz de mover estos objetos dentro de su lista.

1.6.13. Usando MouseEvent con OMGraphics

MouseEvent son usado simplemente para saber que está haciendo el ratón sobre OMGraphics. OMGraphisList dispone de una serie de métodos que nos permite pregunta a MouseEvent sobre sus coordenas:

- OMGraphic findClosest(int x, int y)
- OMGraphic findClosest(int x, int y, int limit): este segundo método solo devuelve un OMGraphic en caso de que el pixel se encuentre dentre de un límite de distancial.
- int findIndexOfClosest(int x, int y)
- int findIndexOfClosest(int x, int y, int limit)
- OMGraphic selectClosest(int x, int y)
- public OMGraphic selectClosest(int x, int y, int limit)
- OMGraphic getOMGraphicThanContains(int x, int y): devuelve OMGraphic que contenga la coordenada especificada.

1.6.14. Combinación/personalización OMGraphics

A partir de la combinación de OMGraphic dados por OpenMap se pueden crear objetos propios. Se añadirá al nuevo objeto la OMGraphicList para disponer de múltiples OMGraphic y métodos.

Respecto a la creación de un OMGraphic personalizado se ha de tener en cuenta 2 métodos:

- public boolean generate(Projection proj): para asegurarse de que todos los objetos de la lista se han proyectado.
- public void render (java.awt.Graphics g): para asegurarse de todos los objetos se han dibujado.

1.7. Imprimiendo información en el mapa

Las capas son generalmente las responsables de imprimir los datos sobre el mapa. Debido al hecho de que son el único componente que puede añadirse al MapBean. Cualquier componente que quiera imprimirse en el mapa, debe hacerlo por medio de una capa.

1.7.1. Capas Básicas (Basic Layers)

Las capas tienen potestad para hacer prácticamente cualquier cosa en orden de representar datos sobre el mapa. Cuando haya diversas capas sobre el mapa, el componente AWT de Java se encargará de cada capa se pinte en el orden correcto.

Las capas son PropertyConsumers, lo que quiere decir, que pueden ser configuradas con las propiedades de Java.

Las capas también pueden usar a MapHandler para encontrar los componentes que necesiten. Sin embargo, las capas deben añadirse manualmente al MapHandler. Para ello, debemos establecer como true la propiedad de la capa addToBeanContex, mediante el método setAddToBeanContext(boolean).

1.7.2. Cambio en proyecciones y representación

Cuando una capa se añade al MapBean, esta es automáticamente añadida como ProjectionListener en el mismo MapBean. Lo que quiere decir que cuando se produzca una modificación en el mapa, la capa será notificada mediante un evento ProjectionChanged y ya será competencia de la misma decir que acción tomar en función del cambio y entonces llamar a repaint().

1.7.3. Interfaces para capas de usuario (Paletas)

OpenMap provee de controles gráficos (paletas) para controlar las capas (com.bbn.openmap.gui.LayersPanel) . Lo obtenemos instanciando Layer.getGUI(), método que devuelve un java.awt.Component, por tonto, componente que podemos personalizar.

1.7.4. OMGraphicHandlerLayer

OMGraphicHandlerLayer es una clase de capa básica que contiene la mayoría de funcionalidades que se necesitan para una capa. La mayoría de capas de OpenMap extienden de este objeto, lo que les permite interactuar y compartir con OMGraphics.

Capas de OpenMap vienen definidas para crear y manejar OMGraphics directamente. Sea por ejemplo, GraticulateLayer la cual crea un OMGraphic internamente, ShapeLayer lee de datos de un archivo, entre otras (véa API para más ejemplos).

1.7.4.1. MapMouseInterpreters

La interfaz MapMouseInterpreter extiende de la interfaz MapMouseListener y representa un MapMouseListener que tiene acceso a OMGraphics. Su comportamiento es sencillo, cuando se producen MouseEvents estos son recibidos por MapMouseListener a través de métodos estándar (mousePressed, mouseRealised, mouseClicked, mouseMoved, mouseDragged, mouseEntered y mouseExit) y luego llama a MapMouseInterpreter para saber cómo actuar.

Encontramos métodos como:

- public boolean leftClick(OMGraphic omg, MouseEvent me): el OMGraphic ha sido clicado con el botón izquierdo del ratón.
- public boolean leftClickOff(OMGraphic omg, MouseEvent me): el usuario ha clicado en algo diferente al OMGraphic que previamente había clicado con el botón izquierdo.
- public boolean rightClick(OMGraphic omg, MouseEvent me): el OMGraphic ha sido clicado con el botón derecho del ratón.
- public boolean rightClickOff(OMGraphic omg, MouseEvent me): el usuario ha clicado en algo diferente al OMGraphic que previamente había clicado con el botón derecho.
- public boolean mouseOver(OMGraphic omg, MouseEvent me): el ratón se ha movido sobre el OMGraphic.
- public boolean mouseNotOver(OMGraphic omg): el ratón ya no se encuentra sobre el OMGraphic, el cual fue previamente notificado que sí se encontraba sobre él.

MapMouseInterpreter también contiene métodos para eventos que se han producido fuere del área de los OMGraphics:

- public boolean leftClick(MouseEvent me): el botón izquierdo del ratón ha sido clicado sobre una ubicación no cubierta por ningún OMGraphic del que MapMouseInterpreter sea consciente.

- public boolean rightClick(MouseEvent me): : el botón derecho del ratón ha sido clicado sobre una ubicación no cubierta por ningún OMGraphic del que MapMouseInterpreter sea consciente.

- public boolean mouseOver(MouseEvent me): el ratón se encuentra sobre una ubicación que no está cubierta por ningún OMGraphic del cual e MapMouseInterpreter sea consciente.

Todos estos métodos se pueden sobrescribir para crear respuestas dinámicas. StandardMapMouseInterpreter es una implementación del MapMouseIntepreter usando GestureResponsePolicies.

1.7.4.2. GestureResponsePolicies

La interfaz GestureResponsePolicies (GRP) describe consultas, notificaciones y peticiones que un MapMouseInterpreter utiliza para reaccionar a sus eventos de ratón. Existen métodos de consulta a los cuales el Interpreter pregunta si las respuestas son apropiadas para un OMGraphic:

- public boolean isHighlightable(OMGraphic omg): método de consulta donde el MapMouseListener pregunta si la GUI debería cambiar para informar al usuario que algo pasará si el OMGraphic es clicado.

- public boolean isSelectable(OMGraphic omg): método de consulta donde el MapMouseListener pregutna si el OMGraphic se puede desplazar o modificar.

Existen métodos de notificación que permiten al GRP saber si el usuario ha realizado algún gesto respecto a un OMGraphic:

- public void select(OMGraphic omg): envía una notificación al GRP cuando un OMGraphic ha sido seleccionado.

- public void deselect(OMGraphic omg): envía una notificación al GRP cuando un OMGraphic ha sido deseleccionado.

- public void highlight(OMGraphic omg): envía una notificación al GRP cuando un OMGraphic debería ser resaltado para hacer saber al usuario que la actual posición del ratón le está señalando.

- public void unhighlight(OMGraphic omg): envía una notificación al GRP cuando un OMGraphic debería volver a su estado por defecto.

Tambíen, se encuentran métodos que permiten al GRP proveer de información sobre los OMGraphic en la GUI:

- public String getToolTipTextFor(OMGraphic omg): petición de un tooltip (pequeño mensaje de texto)s obre el OMGraphic.

- public String getInfoTextFor(OMGraphic omg): petición de más información para imprimir en pantalla sobre el OMGraphic.

- public List getItemsForOMGraphicMenu(OMGraphic omg): petición de un popup menu impreso en pantalla para un OMGraphic

- public List getItemsForMapMenu(): petición de opciones para un lugar que no esté sobre un OMGraphic, como un menú que aparece cuando se clica sobre el mapa.

Es competencia del MapMouseInterpreter cuando los susodichos métodos deban ser llamados y cómo deban actuar a tales eventos del ratón (MouseEvents.)

1.7.4.3. OMGraphicHandlerLayer, GestureResponsePolicy

El OMGraphicHandlerLayer implementará la interfaz GestureResponsePolicy. Capas que extiendan de esta interfaz podrán sobrescribir sus métodos para crear respuestas más afinas a la finalidad de la aplicación buscada. Por defecto, OMGraphicHandlerlayer considera todos los OMGraphic como highlightable (destacables/remarcable) pero no selectable(selecionable).

1.7.5. PlugIns

Se definen los PlugIns en el paquete com.bbn.openmap.plugin. Estos objetos acceden a fuentes de datos para crear y gestionar OMGraphics. Existe una capa específicamente creada para imprimir gráficos en el mapa a partir de PlugIns, PlugInLayaer.

La razón de ser de los PlugIns, es acceder a datos y crear OMGraphics todo en un mismo objeto. PlugIns también dispone de método para GUIs y por tanto claro está de implementar MapMouseListener.

1.7.6. Modificación de OMGraphics mediante DrawingTool

OMDrawingTool (herramientas de dibujo) es un paquete (com.bbn.openmap.tools.drawing.OMDrawingToo) que permite al usuario la modificación de la localización, forma o apariencia de varios OMGraphics. Este objeto se puede añadir al MapHandler para que cualquier componente puede acceder a él.

En lo que respecta a la modificación de OMGraphics, el OMDrawingTool envuelve estos objetos mediante com.bbn.openmap.omGraphics.EditableOMGraphic. Existen varios tipos de EditableOMGraphics, cada uno con conocimiento de cómo manipular ciertos tipos de OMGraphics. EditableOMGraphics es el responsable de superponer puntos sobre la superficie de un OMGraphic y modificar dicho objeto cuando el usuario pinche y manipule la ubicación de los citados puntos.

El reconocimiento de qué EditableOMGraphics, debe usarse para cada OMGraphic se realiza automáticamente mediante el uso de EditToolLoaders. Así, cuando se añade EditToolLoaders al MapHandler, OMDrawingTool puede encontrarlo y dinámicamente cambia que tipos de OMGraphics puede modicar.

1.8. Imprimiendo datos de formatos soportados

OpenMap incluye varios paquetes capaces de imprimir en pantalla gran variedad de imágenes.

1.8.1. Datos de ubicación vía archivos CSV o base de datos

Mediante LocationLayer, OpenMap permite la lectura y representación de datos de ubicación para crear OMGraphics. Estos objetos contienen un marcador que representa el objeto en sí y un OMText que imprime el nombre del objeto.

LocationLayer implementa la interfaz LocationHandler que se encarga de la abstracción de los datos. Para el estudio de este trabajo fin de grado, nos interesaría hablar de CSVLocationHandler. Puede leer archivos CSV, valores del cual están separados por comas. Existen otras interfaces como DBLocation, pero escapa del objetivo de este trabajo.

1.8.2. Datos ESRI Shape

Se presentan varias opciones para imprimir datos de archivos en formato ESRI Shape. Archivos Shape (de forma) contienen vectores geométricos (puntos, líneas, polígonos) y son usualmente separados en tres archivos. El geométrico con extensión .shp, el archivo índice con extensión .shx y el archivo de atributos de los datos con extensión dbf. Los tres comparten el mismo nombre.

Importante, respecto al archivo shp, que contiene la geometría de los objetos que van a ser representados en el mapa. OpenMap solo puede imprimir aquellos archivos de tipo shp que tengan su geometría definida mediante longitudes y latitudes en grados decimales. De otra manera, no serán visibles en el mapa.

2. Código TrafficMonitoringSync

En este apartado del anexo se incluyen las clases que se han considerado más relevantes del proyecto. Además, claro que no se han incluido código que no se considere esencial sea librerías o declaraciones de las miams, comentarios adicionales etc. Con el fin de no extender demasiado el documento.

2.1. Paquete Antenna

2.1.1. AntennaReciverListener

```
public class AntennaReceiverListener extends java.lang.Thread implements
TrafficGenerator {

        java.net.Socket so;
        java.io.BufferedReader br;
        java.io.PrintWriter pw = null;
        // List of flights with position information
        java.util.concurrent.ConcurrentHashMap<String, Traffic.FlightObject> traffics;
        // List of flights with no position information
        java.util.concurrent.ConcurrentHashMap<String, Traffic.FlightObject> traffics_np;
        traffic_cleaner mrclean;
        boolean running = false;
        private boolean change = false;
        RealTimeAirplanesLayer layer=null;

        public AntennaReceiverListener(String host, int port) throws
java.net.UnknownHostException, java.io.IOException {
            so = new java.net.Socket(host, port);
            br = new java.io.BufferedReader(new
java.io.InputStreamReader(so.getInputStream()));
            pw = new java.io.PrintWriter(so.getOutputStream());

            this.traffics = new java.util.concurrent.ConcurrentHashMap<>();
            this.traffics_np = new java.util.concurrent.ConcurrentHashMap<>();
            mrclean= new traffic_cleaner();

        }

        public AntennaReceiverListener(RealTimeAirplanesLayer layer, String host, int
port) throws java.net.UnknownHostException, java.io.IOException {
            so = new java.net.Socket(host, port);
            br = new java.io.BufferedReader(new
java.io.InputStreamReader(so.getInputStream()));
            pw = new java.io.PrintWriter(so.getOutputStream());

            this.traffics = new java.util.concurrent.ConcurrentHashMap<>();
            this.traffics_np = new java.util.concurrent.ConcurrentHashMap<>();
            this.layer = layer;
            mrclean= new traffic_cleaner();

        }
        public AntennaReceiverListener() {
            throw new UnsupportedOperationException("Not supported yet."); //To change body
of generated methods, choose Tools | Templates.
        }

        public void startit() {
            this.running = true;
            this.start();
            mrclean.start();
        }
```

```
        public void stopit() {
            if (this.running) {
                this.running = false;
                this.traffics.clear();
                this.traffics_np.clear();
            }
        }


        // change==true indicates  new traffic
        public void setChange(boolean valor){
                    change = valor;
        }
         // to synchronize with listeners (polling method)

        public boolean getChange(){
                    return(change);
            }

        @Override
        @SuppressWarnings("empty-statement")
        public synchronized void run() {
            try {
                String frame;
                // Flight creation state
                int state; // 0: not in traffic list && not in  traffic_np list
                           // 1: not in traffic list && in traffic_np list
                           // 2: in traffic list
                Traffic.FlightObject flight;

                while (this.running) {
                   //Loop for read lines, decode and update what it needs.
                    frame = br.readLine();
                    if (frame==null) continue;

                    frame = frame.trim();
                    String[] value = frame.split(",");
                    try {
                        if ((value[0].compareTo("MSG") == 0 || value[0].compareTo("ID") ==
0) && (value[4] != null)) {
                            flight = this.traffics.get(value[4]);
                            if (flight == null) {
                                // The flight is not in traffic list
                                state = 1;
                                flight = this.traffics_np.get(value[4]);
                                if (flight == null) {
                                    // The flight is new: not in traffic list nor in
traffic_np list
                                    state=0;
```

```
                                flight = new Traffic.FlightObject(value[3], value[4],
value[5], value[6], value[7]);
                            }
                        } else {
                            // The flight is already in traffic list
                            state = 2;
                        }

                        if (value[0].compareTo("ID") == 0 && value.length >= 11) {
                            flight.UpdateCallsign(value[10]);
                        } else if (value[0].compareTo("MSG") == 0 ) {
                            switch (value[1]) {
                                case "1":
                                    if (value.length >=11)
                                        flight.UpdateCallsign(value[10]);
                                    break;
                                case "2":
                                    flight.updateMSG2(value);
                                    break;
                                case "3":
                                    flight.updateMSG3(value);
                                    break;
                                case "4":
                                    flight.updateMSG4(value);
                                    break;
                                case "5":
                                    flight.updateMSG5(value);
                                    break;
                                case "6":
                                    flight.updateMSG6(value);
                                    break;
                                case "7":
                                    flight.updateMSG7(value);
                                    break;
                                case "8":
                                    flight.updateMSG8(value);
                                    break;
                                default:
                                    System.err.println("AntennaReceiver:      invalid
message type -> " + frame);
                            }
                        } else {
                            System.err.println("AntennaReceiver: spurious message from
antenna -> " + frame);
                        }

                        flight.timestamp = System.currentTimeMillis();

                        if (!"".equals(flight.Latitude)) { // if positioned
                            if (state==0) { //positioned & not previously seen
                                this.traffics.put(flight.HexIdent, flight);
```

```
                            } else if (state==1) { //positioned &  previously seen
                                this.traffics.put(flight.HexIdent, flight);
                                this.traffics_np.remove(flight.HexIdent);
                            }

                            if (this.layer != null)
                                this.layer.ChangeNotification(this);


                        } else // not positioned & not previously seen
                            this.traffics_np.put(flight.HexIdent, flight);
                    }
                } catch (Exception e) {

                    System.err.println("AntennaReceiver:   corrupted   message   from
antenna -> " + frame);

                }
            }
            pw.print("0");
            pw.flush();
            br.close();
            pw.close();
            so.close();
            while (mrclean.isAlive());
            System.out.println("AntennaReceiver: finished.");
        } catch (Exception e) {
            System.err.println("AntennaReceiver: receive error.");
            e.printStackTrace();
        }
    }


    public class traffic_cleaner extends java.lang.Thread {

        @Override
        public void run() {
            int count=0;
            while (running) {
                try {
                    java.lang.Thread.sleep(60);
                } catch (java.lang.InterruptedException e) {
                }
                if (count==1000) {
                    clean_lost_flights();
                    count=0;
                } else
                    count++;
            }
            System.out.println("traffic_cleaner: finished.");
```

```
        }

        private synchronized void clean_lost_flights() {
            java.util.Enumeration<String> enumeration = traffics.keys();
            Traffic.FlightObject i;
            String key;

            long tnow = System.currentTimeMillis();
            while (enumeration.hasMoreElements()) {
                key = enumeration.nextElement();
                i = traffics.get(key);
                if ((tnow - i.timestamp) > 60000) {
                    traffics.remove(key);
                    setChange(true); // Notify change to listeners
                }
            }

        }
    }
    public Traffic.FlightObject[] getFlightObjects(){
    return (traffics.values().toArray(new Traffic.FlightObject[0]));
    }

}
```

2.2. Paquete Layers

2.2.1. RealTimeAirplanesLayer

```
public class RealTimeAirplanesLayer extends Layer implements printAirplanes {

    private OMGraphicList omgraphics;
    public AntennaReceiverListener antenna = null;
    public TrafficRadar father;
    private String host;
    private int port;
    private boolean isRunning = false;
```

```
        public RealTimeAirplanesLayer(TrafficRadar father) {
            omgraphics = new OMGraphicList();
            this.father = father;
        }

        public void Start(String host, int port) {
            this.host = host;
            this.port = port;
            try {
                      this.antenna   =   new   AntennaReceiverListener(this,   this.host,
this.port);
            } catch (IOException ex) {
                System.err.println("Test_AntennaReceiver: could not connect to host");
                Logger.getLogger(Test_AntennaReceiver.class.getName()).log(Level.SEVERE,
null, ex);
            }
            antenna.startit();
            isRunning = true;
        }
        public void Stop() {

            this.antenna.stopit();
            while (this.antenna.isAlive());
            isRunning = false;
        }
        public boolean Running() {
            return (isRunning);
        }

        @Override
        public OMText createLabel(double lat, double lon, String text, Paint color,
                int justification) {
            Font default_font = new Font("TimesRoman", Font.PLAIN, 10);
            FontSizer font_sizer = new FontSizer(default_font, 5000000f, 1, 1, 15);
            OMText label = new OMText(lat, lon, text, default_font, justification);
            label.setFontSizer(font_sizer);
            label.setLinePaint(color);
            label.setSelectPaint(Color.white);
            return label;
        }

        public OMPoly createPolyline(double[] latLon, Color color) {
            OMPoly     poly     =     new     OMPoly(latLon,     OMGraphic.DECIMAL_DEGREES,
OMGraphic.LINETYPE_GREATCIRCLE, -1);
            BasicStroke stroke = new BasicStroke(3);
            poly.setStroke(stroke);
            poly.setLinePaint(color);
            return poly;
        }
```

```
        @Override
        public OMRaster createImage(double lat1, double lon1, double track, boolean logged)
{

            ImageIcon icon;

            if (logged) {
                icon = new ImageIcon("avion24_azul.png");
            } else {
                icon = new ImageIcon("avion24_negro.png");
            }

            OMRaster bitmap = new OMRaster(lat1, lon1, icon);
            bitmap.setRotationAngle(Math.PI * track / 180);
            return bitmap;
        }


        public void paintLog(FlightObject fo, Color color) {

            Traffic.FlightObject.Point p = null;
            int max = fo.log.size();

            double[] llPoints = new double[2 * max];

            int i2 = 0;
            java.util.Iterator<Traffic.FlightObject.Point> j = fo.log.iterator();
            for (int i = 0; i < max; i++) {
                p = j.next();
                llPoints[i2] = p.latitud;
                llPoints[i2 + 1] = p.longitud;
                i2 = i2 + 2;
            }
            omgraphics.add(createPolyline(llPoints, color));
        }

        @Override
        public void paintAirplaneImage(MapBean bean, FlightObject fo) {

            double coordx = bean.getProjection().forward(fo.lat, fo.lon).getX();
            double coordy = bean.getProjection().forward(fo.lat, fo.lon).getY();

            double track = fo.track - 90.0;

            double lat = bean.getProjection().inverse(coordx - 12, coordy - 12).getY();
            double lon = bean.getProjection().inverse(coordx - 12, coordy - 12).getX();

            omgraphics.add(createImage(lat, lon, track, fo.logged));
        }




        @Override
        public void paintAirplaneText(MapBean bean, FlightObject fo, Color color) {


            double coordx = bean.getProjection().forward(fo.lat, fo.lon).getX();
            double coordy = bean.getProjection().forward(fo.lat, fo.lon).getY();


            String texto;
            if (!"".equals(fo.Callsign)) {
                texto = fo.Callsign;
            } else {
                texto = fo.HexIdent;
            }
            double lat = bean.getProjection().inverse(coordx + 12, coordy - 12).getY();
            double lon = bean.getProjection().inverse(coordx + 12, coordy - 12).getX();
```

```
            omgraphics.add(createLabel(lat, lon, texto, color, OMText.JUSTIFY_LEFT));
        }

        public void toClear() {
            omgraphics.clear();
        }
        public void rePaint(MapBean bean) {
            omgraphics.generate(bean.getProjection());
            repaint();
        }

        @Override
        public void paintAirplanes(MapBean bean, FlightObject fo[], Color colorTexto, Color
colorLog) {

            for (int k = 0; k < fo.length; k++) {

                paintAirplaneImage(bean, fo[k]);
                paintAirplaneText(bean, fo[k], colorTexto);
                if (fo[k].logged) {
                    paintLog(fo[k], colorLog);
                }
            }
        }

        public void ChangeNotification(AntennaReceiverListener antenna) {
            Traffic.FlightObject[] fo;
            fo = antenna.getFlightObjects();
            father.myPanel.setnFlights(String.valueOf(fo.length));
            this.toClear();
            this.paintAirplanes(father.mapBean, fo, Color.blue, Color.red);
            this.rePaint(father.mapBean);

        }


        @Override
        public void paint(java.awt.Graphics g) {
            omgraphics.render(g);
        }

       @Override
        public void projectionChanged(ProjectionEvent e) {
            omgraphics.project(e.getProjection(), true);
            repaint();
        }
    }
```

2.2.2. RouteLayer

```
public class RouteLayer extends Layer {

    TrafficRadar father;
    private OMGraphicList omgraphics;

    public RouteLayer(TrafficRadar father) {

          this.father = father;
          omgraphics = new OMGraphicList();
          createGraphics(omgraphics);
    }


    public OMLine createLine(double lat1, double lon1, double lat2, double lon2,
                             Color color) {
        OMLine line = new OMLine(lat1, lon1, lat2, lon2,
OMGraphic.LINETYPE_GREATCIRCLE);
        line.setLinePaint(color);
        return line;
    }


     public OMRect createPoint(double lat1, double lon1, Color color) {
        OMRect rect = new OMRect(lat1-0.02d, lon1-0.02d, lat1+0.02d,
lon1+0.02d,OMGraphic.LINETYPE_STRAIGHT);
        rect.setFillPaint(color);
        return rect;
     }


    public OMText createLabel(double lat, double lon, String text,  Paint color,
                              int justification) {
        Font default_font = new Font("TimesRoman", Font.PLAIN, 10);
        FontSizer font_sizer = new FontSizer(default_font,5000000f, 1, 1, 15);
        OMText label = new OMText(lat, lon, text, default_font, justification);
        label.setFontSizer(font_sizer);
        label.setLinePaint(color);
        label.setSelectPaint(Color.white);
        return label;

    }

     public void createGraphics(OMGraphicList graphics) {
        omgraphics.clear();
        parseFeature(graphics,"UP34.kml");
        parseFeature(graphics,"UM134.kml");
        parseFeature(graphics,"UL150.kml");
        parseFeature(graphics,"UM176.kml");
        parseFeature(graphics,"UN609.kml");
        parseFeature(graphics,"UN733.kml");
        parseFeature(graphics,"UN851.kml");
```

```
            parseFeature(graphics,"UN860.kml");
            parseFeature(graphics,"UM985.kml");

        }


        private void parseGeometry(Geometry geometry) {
       if(geometry != null) {
            if(geometry instanceof Polygon) {
                Polygon polygon = (Polygon) geometry;
                Boundary outerBoundaryIs = polygon.getOuterBoundaryIs();
                if(outerBoundaryIs != null) {
                    LinearRing linearRing = outerBoundaryIs.getLinearRing();
                    if(linearRing != null) {
                        List<Coordinate> coordinates = linearRing.getCoordinates();
                        if(coordinates != null) {
                            for(Coordinate coordinate : coordinates) {
                                parseCoordinate(coordinate);
                            }
                        }
                    }
                }
            }
            if(geometry instanceof LineString) {
                LineString linestring = (LineString) geometry;
                List<Coordinate> coordinates = linestring.getCoordinates();
                if(coordinates != null) {
                     for(Coordinate coordinate : coordinates) {
                         parseCoordinate(coordinate);
                     }
                }
            }

       }
}


        private void parseFeature(OMGraphicList graphics, String file) {
        final Kml kml = Kml.unmarshal(new File(file));
        Feature feature = kml.getFeature();

        if(feature != null) {
            if(feature instanceof Document) {
                Document document = (Document) feature;
                List<Feature> featureList = document.getFeature();
                for(Feature documentFeature : featureList) {
                    if(documentFeature instanceof Folder) {
                        Folder folder = (Folder) documentFeature;
                        List<Feature> featureList2 = folder.getFeature();
                        for(Feature documentFeature2 : featureList2) {
```

```
                        if(documentFeature2 instanceof Placemark) {
                            Placemark placemark = (Placemark) documentFeature2;
                            Geometry geometry = placemark.getGeometry();
                            parseGeometry(geometry);
                            String description = placemark.getDescription();
                            parseDescription(graphics, description);
                        }
                    }
                }
                if(documentFeature instanceof Placemark) {
                    Placemark placemark = (Placemark) documentFeature;
                    Geometry geometry = placemark.getGeometry();
                    parseGeometry(geometry);

                }
            }
        }
     }
    }

    private void parseCoordinate(Coordinate coordinate) {
        if(coordinate != null) {
            paintPoint(coordinate.getLatitude(),              coordinate.getLongitude(),
Color.magenta);
        }
    }

    private void parseDescription(OMGraphicList graphics, String description) {
        if(description != null) {

            String wpt1Ident = extractData(description, "WPT1_IDENT");
            String w1_wgsdlat = extractData(description, "W1_WGSDLAT");
            String w1_wgsdlon = extractData(description, "W1_WGSDLON");

            double lat1 = Double.valueOf(w1_wgsdlat);
            double lon1 = Double.valueOf(w1_wgsdlon);

            String wpt2Ident = extractData(description, "WPT2_IDENT");
            String w2_wgsdlat = extractData(description, "W2_WGSDLAT");
            String w2_wgsdlon = extractData(description, "W2_WGSDLON");

            double lat2 = Double.valueOf(w2_wgsdlat);
            double lon2 = Double.valueOf(w2_wgsdlon);
            paintLine(lat1, lon1, lat2, lon2, Color.blue);

            double coordx = father.mapBean.getProjection().forward(lat1, lon1).getX();
            double coordy = father.mapBean.getProjection().forward(lat1, lon1).getY();

            double longitud = father.mapBean.getProjection().inverse(coordx + 2, coordy -
2).getX();
```

```
            double latitud = father.mapBean.getProjection().inverse(coordx + 2, coordy -
2).getY();

            paintText(latitud, longitud, wpt1Ident, Color.magenta);

        }
    }
        private  String extractData(String description, String label){
                int pos = description.indexOf(label);
                String      cadena      =      description.substring(pos+label.length()+3,
pos+label.length()+18);
                pos = cadena.indexOf("<br>");
                return (cadena.substring(0, cadena.indexOf("<br>")));

        }

        public void paintLine(double lat1, double lon1, double lat2, double lon2, Color
color){
            omgraphics.add(createLine(lat1, lon1, lat2, lon2, color));

        }
            public void paintPoint(double lat1, double lon1, Color color){
            omgraphics.add(createPoint(lat1, lon1, color));

        }

        public  void paintText(double lat, double lon, String text, Color color) {
            omgraphics.add(createLabel(lat, lon, text, color, OMText.JUSTIFY_LEFT));
        }
        @Override
        public void paint(java.awt.Graphics g) {
            omgraphics.render(g);
        }
         @Override
       public void projectionChanged(ProjectionEvent e) {
            omgraphics.project(e.getProjection(), true);
            repaint();
        }
    }
```

2.2.3. TrackLayer

```
   public class TrackLayer extends Layer implements printAirplanes {

        public TrafficRadar father;
        private OMGraphicList omgraphics;
        public    java.util.concurrent.ConcurrentHashMap<String,    Traffic.FlightObject>
airplanes = new java.util.concurrent.ConcurrentHashMap<String,Traffic.FlightObject>();
        public     java.util.ArrayList<Traffic.FlightObject>     aux_llista     =     new
java.util.ArrayList<Traffic.FlightObject>();
```

```
        protected boolean playing = false;


        public TrackLayer(TrafficRadar padre) {
            omgraphics = new OMGraphicList();
            this.father = padre;
        }


        @Override
        public OMText createLabel(double lat, double lon, String text,  Paint color,
                                  int justification) {
            Font default_font = new Font("TimesRoman", Font.PLAIN, 10);
            FontSizer font_sizer = new FontSizer(default_font,5000000f, 1, 1, 15);

            OMText label = new OMText(lat, lon, text, default_font, justification);
            label.setFontSizer(font_sizer);
            label.setLinePaint(color);
            label.setSelectPaint(Color.white);
            return label;
        }

        @Override
         public  OMRaster  createImage(double  lat1,  double  lon1,  double  track,boolean
logged){

           ImageIcon icon;

           icon = new ImageIcon("avion24_negro.png");
           OMRaster bitmap = new OMRaster(lat1, lon1, icon);
            bitmap.setRotationAngle(Math.PI*track/180);
           return bitmap;
         }


        @Override
         public void paintAirplaneImage(MapBean bean, FlightObject fo){

            double coordx = bean.getProjection().forward(fo.lat, fo.lon).getX();
            double coordy = bean.getProjection().forward(fo.lat, fo.lon).getY();

            double track = fo.track - 90.0;

            double lat = bean.getProjection().inverse(coordx - 12, coordy - 12).getY();
            double lon = bean.getProjection().inverse(coordx - 12, coordy - 12).getX();

            omgraphics.add(createImage(lat, lon, track, fo.logged));
        }
```

```
        @Override
        public  void paintAirplaneText(MapBean bean, FlightObject fo, Color color) {

            double coordx = bean.getProjection().forward(fo.lat, fo.lon).getX();
            double coordy = bean.getProjection().forward(fo.lat, fo.lon).getY();

            String texto;

            if (!"".equals(fo.Callsign))
                 texto = fo.Callsign;
            else
                 texto = fo.HexIdent;

            double lat = bean.getProjection().inverse(coordx + 12, coordy - 12).getY();
            double lon = bean.getProjection().inverse(coordx + 12, coordy - 12).getX();

            omgraphics.add(createLabel(lat, lon, texto, color, OMText.JUSTIFY_LEFT));
        }

        @Override
        public void paintAirplanes(MapBean bean, FlightObject fo[], Color colorTexto, Color
colorLog){

            for (int k=0; k<fo.length; k++){

                 paintAirplaneImage(bean, fo[k]);
                 paintAirplaneText(bean, fo[k], colorTexto);
            }
        }

        @Override
        public void toClear() {
            omgraphics.clear();
        }

        @Override
        public void rePaint(MapBean bean){
            omgraphics.generate(bean.getProjection());
            repaint();
        }



        public void cleanAirplane(String key){
                     this.airplanes.remove(key);
              }

       public void ChangeNotification(TrackReader player){
                    Traffic.FlightObject[] fo;
                    fo = player.getFlightObjects();
```

```
                    father.myPanel.setnFlights(String.valueOf(fo.length));
                    this.toClear();
                    this.paintAirplanes(father.mapBean, fo, Color.blue, Color.red);
                    this.rePaint(father.mapBean);
                    for (int i = 0; i < fo.length; i++){
                        Traffic.FlightObject aux = this.airplanes.get(fo[i].HexIdent);
                        if(aux==null){
                                this.airplanes.put(fo[i].HexIdent, fo[i]);

                        }
                        else{
                                aux.updateAll(fo[i]);
                        }
                    }
        }

        public void close_player(TrackReader player){
               if(player != null){
                            player.stopit();
                     }
         }

        @Override
        public void paint(java.awt.Graphics g) {
            omgraphics.render(g);
        }

        public void projectionChanged(ProjectionEvent e) {
            omgraphics.project(e.getProjection(), true);
            repaint();
        }
    }
```

2.2.4. PrintAirplanes

```
    public interface printAirplanes {


        public OMText createLabel(double lat, double lon, String text, Paint color,int
justification);
        public OMRaster createImage(double lat1, double lon1, double track,boolean logged);
        public void paintAirplaneImage(MapBean bean, FlightObject fo);
        public  void paintAirplaneText(MapBean bean, FlightObject fo, Color color);
        public void paintAirplanes(MapBean bean, FlightObject fo[], Color colorTexto, Color
colorLog);
        public void toClear();
        public void paint(java.awt.Graphics g);
        public void rePaint(MapBean bean);
    }
```

2.3. Paquete TrafficMonitoring

2.3.1. TrafficRadar

```
public class TrafficRadar {

    //MapMouseListener:
    final MyMapMouseListener myMapMouseListener = new MyMapMouseListener();
    MapHandler mapHandler;

    public RealTimeAirplanesLayer flightLayer;
    public RouteLayer routeLayer;
    public TrackLayer trackLayer;

    public MapBean mapBean;
    public MyTextPanel myPanel;

    private java.util.concurrent.ConcurrentHashMap<String, Traffic.FlightObject> logs;

    Traffic.FlightObject MouseIsOnIt;

    /* TrackGenerator and TrackReader */
    protected TrackGenerator tracker = null;
    protected TrackReader player = null;

    /* State */
    protected boolean connected_radar = false;
    protected boolean recording = false;


    public TrafficRadar() throws MalformedURLException {

        try {


            this.logs = new java.util.concurrent.ConcurrentHashMap<>();
            MouseIsOnIt = null;

            MapPanel mapPanel = new OverlayMapPanel();

            // Get the default MapHandler the BasicMapPanel created.
            mapHandler = mapPanel.getMapHandler();

            // Get the default MapBean that the BasicMapPanel created.
            mapBean = mapPanel.getMapBean();

            mapBean.setCenter(new   LatLonPoint.Double(39.490556,   -0.480278));   //
Centered in Valencia
```

```
// mapBean.setCenter(40.467927f,  -3.569205f); //Centered in Madrid

mapBean.setScale(6000000f); // to scale in Spain
// mapBean.setScale(2000000f); // to scale in Valencia


//////////////////////Important to do//////////////////////

mapHandler.add(new LayerHandler());

// Add MouseDelegator, which handles mouse modes (managing mouse
// events)
mapHandler.add(new MouseDelegator());

// Add OMMouseMode, which handles how the map reacts to mouse
// movements
mapHandler.add(new OMMouseMode());

// Add a ToolPanel for widgets on the north side of the map.
ToolPanel tpl = new ToolPanel();
mapHandler.add(tpl);

// Add a GUI to control the layers. A
// button to launch it will get added to the ToolPanel.
mapHandler.add(new LayersPanel());
mapHandler.add(new ScaleTextPanel());
myPanel = new MyTextPanel(this);
mapHandler.add(myPanel);

// Add a Menu Bar
JMenuBar menubar = new JMenuBar();

// Add own menus to MenuBar
MyMenuActions item = new MyMenuActions(this);
MyMenuHelp item2 = new MyMenuHelp(this);

menubar.add(item);
menubar.add(item2);
mapHandler.add(menubar);

 ////////////////////// Initialization//////////////////////
OpenMapFrame frame = new OpenMapFrame("Radar Monitor");

// Size the frame appropriately
frame.setSize(768, 768);
mapHandler.add(frame);

//Application Icon
ImageIcon img = new ImageIcon("iconoaplicacionfinal.png");
frame.setIconImage(img.getImage());
```

```
//////////////////WORLD POLITICAL SOLID LAYER/////////////////
ShapeLayer shapeLayer = new BufferedShapeLayer() {
    @Override
    public synchronized MapMouseListener getMapMouseListener() {
        return myMapMouseListener;
    }
};
Properties shapeLayerProps1 = new Properties();
shapeLayerProps1.put("prettyName", "Worl Political Solid");
shapeLayerProps1.put("lineColor", "000000");
shapeLayerProps1.put("fillColor", "BDDE83");
shapeLayerProps1.put("shapeFile", "cntry08.shp");
shapeLayer.setProperties(shapeLayerProps1);
shapeLayer.setVisible(true);

// Last on top.
mapHandler.add(shapeLayer);


//////////////////////PROVINCES OF SPAIN //////////////////////
ShapeLayer shapeLayer2 = new BufferedShapeLayer() {

    @Override

    public synchronized MapMouseListener getMapMouseListener() {
        return myMapMouseListener;
    }
};
Properties shapeLayerProps2 = new Properties();
shapeLayerProps2.put("prettyName", "Provinces of Spain");
shapeLayerProps2.put("lineColor", "000000");
shapeLayerProps2.put("fillColor", "BDDE83");
shapeLayerProps2.put("shapeFile", "ESP_adm2.shp");
shapeLayer2.setProperties(shapeLayerProps2);
shapeLayer2.setVisible(true);

mapHandler.add(shapeLayer2);


////////////////////// REMAINING LAYERS ////////////////////////

mapHandler.add(new GraticuleLayer());

routeLayer = new RouteLayer(this);
routeLayer.setName("Routes Layer");
mapHandler.add(routeLayer);

flightLayer = new RealTimeAirplanesLayer(this);
flightLayer.setName("Real Time Airplanes Layer");
```

```
            mapHandler.add(flightLayer);

            trackLayer = new TrackLayer(this);
            trackLayer.setName("Track Layer");
            mapHandler.add(trackLayer);

              frame.addWindowListener(new WindowAdapter() {
                @Override
                public void windowClosing(WindowEvent e) {
                    if (flightLayer.Running()) {
                        flightLayer.Stop();
                    }
                    System.exit(0);
                }
            });
        } catch (MultipleSoloMapComponentException msmce) {
            // The MapHandler is only allowed to have one of certain
            // items. These items implement the SoloMapComponent
            // interface. The MapHandler can have a policy that
            // determines what to do when duplicate instances of the
            // same type of object are added - replace or ignore.

            // In this example, this will never happen, since we are
            // controlling that one MapBean, LayerHandler,
            // MouseDelegator, etc is being added to the MapHandler.
        }
    }

    public class MyMapMouseListener implements MapMouseListener {

        @Override
        public String[] getMouseModeServiceList() {
            return new String[]{SelectMouseMode.modeID};

        }

        @Override
        public boolean mouseClicked(MouseEvent e) {

            MapMouseEvent mme = (MapMouseEvent) e;

            Traffic.FlightObject[] fo;
            AntennaReceiverListener ar = flightLayer.antenna;
            fo = ar.getFlightObjects();

            int ratonX = mme.getX();
            int ratonY = mme.getY();
            int i;
            for (i = 0; i < fo.length; i++) { // find coordinates in vector
```

```
                    double     coordx     =     mapBean.getProjection().forward(fo[i].lat,
fo[i].lon).getX();
                    double     coordy     =     mapBean.getProjection().forward(fo[i].lat,
fo[i].lon).getY();

                    if ((coordx < ratonX + 12) && (coordx > ratonX - 12) && (coordy <
ratonY + 12) && (coordy > ratonY - 12)) {
                        break;
                    }
                }

                if (i < fo.length) { // found

                    Traffic.FlightObject foAux;
                    foAux = logs.get(fo[i].HexIdent);

                    if (foAux == null) { // new track: put in list

                        fo[i].logged = true;
                        logs.put(fo[i].HexIdent, fo[i]);
                        System.out.println("inserting track " + fo[i].HexIdent);

                    } else { // flight in track: remove

                        fo[i].logged = false;
                        logs.remove(fo[i].HexIdent);
                        System.out.println("deleting track " + fo[i].HexIdent);
                    }
                }
                return true;

            }

            @Override
            public boolean mousePressed(MouseEvent e) {
                return true;
            }

            @Override
            public boolean mouseReleased(MouseEvent e) {
                return true;
            }

            @Override
            public void mouseEntered(MouseEvent e) {
            }

            @Override
            public void mouseExited(MouseEvent e) {
            }
```

```
            @Override
            public boolean mouseDragged(MouseEvent e) {
                return true;
            }

            @Override
            public boolean mouseMoved(MouseEvent e) {
                return true;
            }

            @Override
            public void mouseMoved() {
            }

        }

        public void close_tracker() {
            if (this.tracker != null) {
                this.tracker.stoptracker();
                this.recording = false;

            }
        }

        public static void main(String[] args) {
             // Schedule a job for the event-dispatching thread:
            // creating and showing this application's GUI.
            javax.swing.SwingUtilities.invokeLater(() -> {
                try {
                    new TrafficRadar();
                } catch (MalformedURLException ex) {
                    Logger.getLogger(TrafficRadar.class.getName()).log(Level.SEVERE,
null, ex);
                }
            });
        }
    }
```