Prototype for a Mixed Separation Air Traffic Control User Interface

Ólafur Freyr Hjálmsson
DTU

Abstract

This Masters thesis of engineering discusses issues related to aircraft separation in an air traffic control environment where mixed separation standards are applied. The simulated test environment contains aircraft with and without radar coverage as well as aircraft equipped with ADS-B equipment. The feature complete air traffic control training simulator Amelia was modified to serve as an interactive prototype used for usability testing with appropriate users. Amelia was provided by Tern Systems Inc. along with their invaluable expert advice. Specific additional features were designed to carry out the required tasks, but as it turned out some of these had to be abandoned due to time constraints. The full design is presented here along with what became the actual implementation. Despite these problems the graphical user interface necessary to carry out usability testing was completed and the missing parts were included using mockup screenshots printed to paper.

The Icelandic Civil Aviation Authority participated in the usability testing where seven active duty air traffic controllers volunteered to give their impressions of the modifications. Even though their work environment is different from Amelia on some key points, they agreed that it was sufficiently similar so that they had no significant problems working with Amelia. The work environment is introduced here and the differences are pointed out. Two rounds of testing took place in the month of February 2005, using the modified Amelia prototype, where additional modifications were made based on feedback after the initial round and tested in the second round.

Concluding from the results of the two usability tests the following recommendations are made: Each individual aircraft on the air traffic controller situation display should indicate which separation standards apply to it at the moment. The separation requirements of each aircraft should be easy to show and focus should be on giving precise information regarding separation between exactly two aircraft. The situation display system should be able to highlight the area where a conflict occurs as soon as it may be detected.

Contents

Chapter 1

Introduction

The text you are about to read is the Masters thesis of engineering by Ólafur Freyr Hjálmsson. The project covered by this thesis was conceived in cooperation with Tern Systems Inc. based in Iceland with the participation of air traffic controllers from the Icelandic Civil Aviation Authority (ICAA).

Tern Systems graciously provided office and computing facilities for use during the project work. For that they deserve many thanks. The knowledge of their staff was a tremendous help.

The software developed for this project was based on their air traffic control (ATC) training simulator product Amelia. For the purpose of this project Tern Systems provided access to the source code of Amelia with the aim to extend it to become a prototype for working with a mixed environment of aircraft separation.

Aircraft separation is the process by which air traffic controllers and aircraft pilots cooperate to keep aircraft at a safe distance of each other. When aircraft are out of radar range, they have traditionally been kept separated by an estimation based on known factors such as aircraft speed and heading as well as verbal communication between pilots and air traffic controllers. Efforts aimed at the use of computers to aid in this process have been ongoing steadily as computer technology has developed. This has resulted in various tools available to air traffic controllers, such as unified radar displays, digital management of flight data, situation displays, software based communication consoles and digital communication with aircraft. A sophisticated system of automated aircraft status reports is gradually being taken into service by many airlines and aviation authorities. This system is referred to as Automatic Dependent Surveillance - Broadcast (ADS-B).

Since different separation standards apply to aircraft being managed by the traditional procedural process from the aircraft using ADS-B equipment, the software user interface used by air traffic controllers is currently lacking a distinction between the aircraft. There is also lacking some way for the controller to see how much airspace is needed for the aircraft to be fully separated. This has not been necessary before since the current environment only has one set of rules in effect. Therefore all aircraft needed the same airspace reservation and the controllers did not need any clarification. There is also the possibility that some future development of ADS-B will impose a new set of separation rules, making this need for distinction and airspace clarification even more varied.

The goal of this project is to propose a solution to the problems of lacking distinction and clarification on the current air traffic controller situation display. For the purpose of reaching that goal, a prototype of a full featured situation display simulating the working environment of the controllers is to be developed. That enables user testing to be performed in order to evaluate the effectiveness of the proposed solution, which can then be refined based on the results and retested.

The text is organized in 3 main parts. First there is a short discussion of the basics, followed by design, implementation and testing of the two versions of the prototype.

Beginning with the basics, chapter 2 contains an overview of air traffic control, an introduction to the procedures and concepts followed by an overview of the software tools in use at ICAA as well as a precise definition of some common terms. The project work is discussed in more detail in chapter 3 to provide a more complete overview. The software upon which the prototype is based, Amelia, is also covered there relating to specific details relevant to this project. To round off the basics, a more thorough discussion of aircraft separation with relation to this project can be found in chapter 4. A strategy for solving the task at hand is developed there.

The design for the first prototype is then presented in chapter 5. After that in chapter 6 the actual implementation of this design is laid out as well as how it is integrated into Amelia. The user interface is a critical part thereof, so all changes made there are crucial. User testing of the first prototype is discussed in chapter 7 as well as the conclusions that can be drawn from the results.

Further enhancements to the prototype are presented in chapter 8 where the original design is seen in light of the conclusions that were drawn at the end of the previous chapter. A refined solution, referred to as version 2.0, is then presented in chapter 6 where the design enhancements are implemented. The final user testing is discussed in chapter 7 where the reactions to the refinements from the first user test are especially highlighted.

General discussion of what further work can be addressed can be found in chapter 11, followed finally by some conclusions which can be drawn from the results.

Chapter 2

The ATC Environment

In order to set the stage properly, it is necessary to provide a presentation of the air traffic control environment. The following description is specifically about the Icelandic Civil Aviation Authority, but the basic principles should nonetheless be comparable with most modern ATC centers. Any discrepancies should be minor, but this can be quite hard to predict since the aviation authorities and area control centers (ACC) are numerous and deploy different software which are able to communicate using standardized protocols. Such differences are numerous and unpredictable, so the focus here is kept on the facilities available for study, the ICAA Reykjavík ACC.

The following sections provide a shallow overview of the ATC environment and are presented here to introduce the unfamiliar reader to this world. The contents of this chapter may be glanced over or skipped by those already well familiar with this topic. Experienced professionals would perhaps regard this text as overly simplified, but it exists here only to present the parts of air traffic control that are considered for this project. It should be considered as a simplification in order to maintain focus on the core issues.

The various procedures which are followed for typical air traffic are introduced in section 2.1, followed by a look at the software tools available to the air traffic controllers at the ICAA ACC in section 2.2. Finally some of the common terms are defined in section 2.3 to give an unambiguous understanding of their meaning to the uninitiated reader. If at any time a mysterious concept is found thrown about precariously, it will most likely be explained there or in appendix A where additional terms are defined and listed in alphabetical order.

2.1 Flight Procedures

The formal standards and recommended practices used in international aviation are maintained by the International Civil Aviation Organization (ICAO), the specialized agency of the United Nations whose mandate is to "ensure the safe, efficient and orderly evolution of international civil aviation"[1]. This organization exists through the constitution of ICAO, which is the Convention on International Civil Aviation, drawn up by a conference in Chicago in November and December 1944, and to which each ICAO contracting state is a party. Standards and procedures published by the ICAO are agreed on through committee work and conference meetings involving representatives from the contracting states. Further information is available at [1].

A vital part of these procedures is the demand that an aircraft pilot must submit a proposed flight plan to the aviation authorities governing the airspace to be traveled through. The flight plan is then compared to the known air traffic before getting clearance. The cleared flight plan might have been modified by the air traffic controllers, but the pilot is required to follow his assigned plan within permitted deviance and maintain a specific cruising flight level that is estimated to be free of traffic. The details of the flight plan are then continuously updated by the air traffic controllers as the need arises for the duration of the flight.

The pilot is required to send position reports to the managing air traffic controller based on the aircraft instruments. These are compared to the expected position at the time and any adjustments to flight plans are made in case of delay or expediency. This rigid procedure of sending reports by radio and maintaining flight plans is referred to as the "procedural" technique of managing air traffic. When there is radar contact with the aircraft (at around 250 Nautical Miles) the air traffic controller has a radar screen at his disposal and the position reports can be verified by the radar position. The radar updates are also much more frequent than pilot position reports, so the aircraft is completely managed by radar acquired position updates and the aircraft is said to be managed by radar. When aircraft are managed by radar, their position is known with much greater accuracy and therefore it is possible to manage more traffic using shorter separation minima. In order to distinguish the aircraft from the other air traffic, it is equipped with a transponder. When the radar wave reaches the aircraft, the transponder amplifies and retransmits the signal while encoding a transponder code in it's own transmission. This is in turn detected by the radar station and relayed to the radar operator. This provides two signals from the same location, the radar return wave as well as the transponder transmission. When used together, they provide an identification code to the object picked up on radar and the object is said to be radar identified.

In an effort to attain radar like position precision, where there are no radar stations, the process of sending position reports can be automated by using computers. A technology in widespread use is Controller to Pilot Data Link Communications (CPDLC), which is based on providing textual communication between air traffic controller and pilot. This is used in conjunction with radio communication to increase accuracy in position reporting from the pilot and ATC direction from the controller. Language barriers can some times lead to misunderstandings and CPDLC is mainly meant to minimize these.

The experimental ADS-B technology mentioned earlier is in it's most basic form based on having a computer on board the aircraft, which broadcasts position reports and other information, with a frequency in the order of seconds. The precise position of the aircraft acquired using GPS equipment, resulting in accuracy within a few meters. These position reports can then be received at ground stations with a radio antenna and relayed to the air traffic controller just like a radar position. The maximum range for the transmission is expected to be around 200 Nautical Miles, which is shorter than radar range. Since ADS-B ground stations use less sophisticated equipment than radar stations, it is expected that they will be more numerous and thereby cover a greater area. Other aircraft in the vicinity equipped with ADS-B are also able to receive these broadcasts and become aware of traffic that might conflict with their flight path. This is even planned to become very useful for collision alerts on taxiways and runways at airports. There are different approaches to providing ADS-B functionality, all of which are referred to here as the generic term ADS-B technology.

For crossing the North Atlantic Ocean, where there is limited radar coverage, most of the flights take place during the same general time periods. Flights depart in the morning from Europe heading west and return in the evening for night flights heading back. This causes some congestion and to minimize the effect, flights are directed to follow precalculated tracks. See figure 2.1 for examples. These tracks are the most efficient routes for crossing the

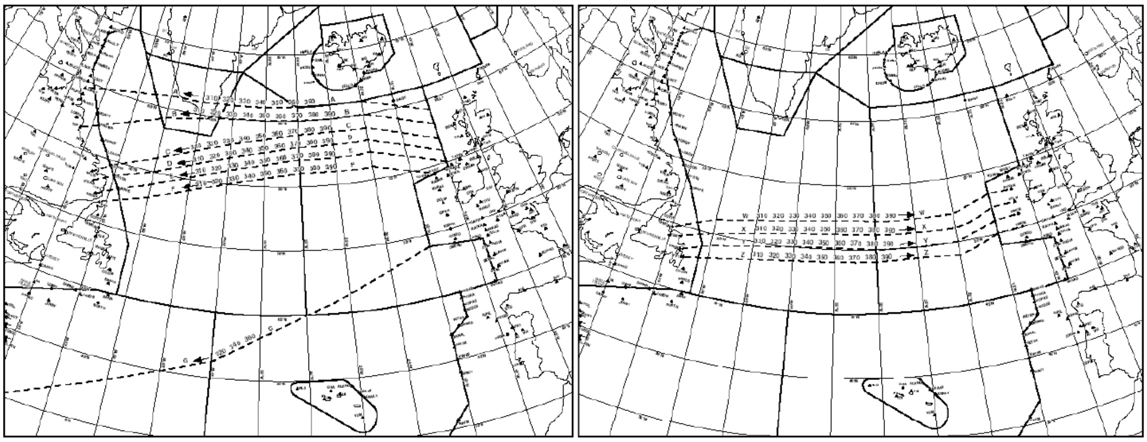


Figure 2.1: *(a) westbound tracks and (b) eastbound tracks.*

Atlantic taking into account weather patterns, earth rotation during the crossing and distances between major cities on the two continents as well as other factors. Further information is available at [2].

The airspace governed by the ICAA is divided into 4 sectors: North, South, East and West. Figure 2.2 shows this division. Radar coverage is practically complete in the South sector over Iceland and the East sector with radar data coming from the Faroe Islands. The North and West sectors over Greenland are without radar coverage so air traffic passing these sectors is managed using the procedural technique. There are some military radar installations in Greenland, but they are not available for civilian use.

When travelling across the North Atlantic the radar coverage is therefore incomplete at the present time and a mixture of procedural and radar techniques are usually employed. At the East boundaries of the ICAA airspace, partial radar coverage is provided by Norway and Scotland, while at the West boundaries there is a gap in radar coverage until Canada is reached.

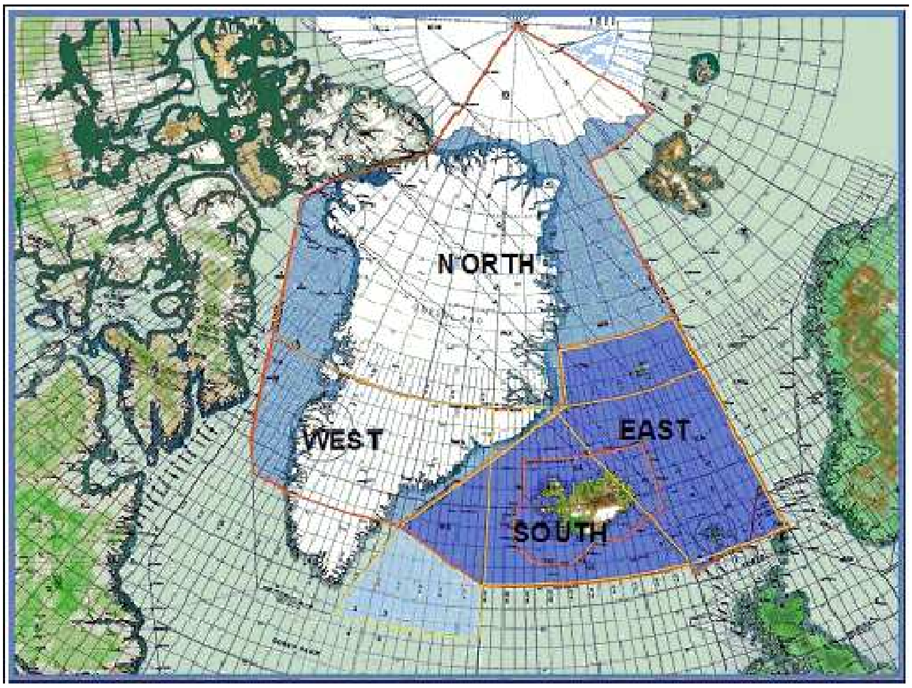


Figure 2.2: *The airspace governed by the Icelandic Civil Aviation Authority.*

2.2 Air Traffic Control

The ATC environment is dedicated to guiding aircraft safely and efficiently through the designated airspace. For this purpose the pilots of these aircraft and the air traffic controllers use a set of tools that facilitate their needs. The most basic of those tools include the aircraft instruments for the pilot and ground based radar as well as updated flight plan information for the controller. Two way verbal radio communication is heavily used to coordinate their actions.

A modern ATC facility like the Reykjavík ACC provides a range of modern computer based tools for the air traffic controller. These include a radar display, consolidating information from multiple radar sources. A centralized storage of flight data allows controllers to instantly view the changes relating to a flight under the command of a different controller and make their plans accordingly. The flight plans are then projected onto a graphical situation display for use in conjunction with the radar display. In addition to these, a datalink connection with the aircraft provides a textual messaging system for quickly and unambiguously communicate standard messages.

Figure 2.3 gives an idea of how conditions are in the ICAA Reykjavík ACC. Note that the person in the foreground is not directing aircraft, but acts as the shift supervisor. The two people in the background are observing and directing actual air traffic on three computer monitors. Usually one screen is dedicated to detailed flight data, one to closeup situational display and the third one to another situational display spanning a greater area to provide overview. This is however up to the individual controller's taste since the environment can be customized to their preference.

Figure 2.3: *Picture taken in the ICAA Reykjavík area control center (from [3]).*

Future enhancements include a fully automated position and status reporting mechanism using the ADS-B system. Automated communication between onboard computing equipment and ground stations gives the controller more information on the state of the aircraft. Moreover, this provides for the possibility of airborne awareness where each aircraft pilot is better able to respond to surrounding traffic.

The following provides descriptions of the specific software tools, either in use or planned for use at ICAA, from the perspective of what purpose they serve. These exact tools are usually commissioned directly from ICAA and are thus not necessarily in use at other aviation authorities' ACCs even though they are commercially available from Tern Systems. Similar systems can be expected to be used however, as all communication protocols and user interface interaction are standardized and well defined.

Radar Display The Radar Data Processing & Presentation System (RDDPS) provides continuously updated radar information combined from all major radar stations around Iceland. Figure 2.4 shows a screenshot of a radar workstation display. The screen maintains a light information load during normal operations. A highly important, but easily overlooked piece of information is the current time as known by the radar system shown in the upper left corner of the display. Directly below is some information regarding the current view, among which are the exact geographical position of the mouse cursor, flight filtering, PPOS (pre position) setting and active radar sources. Each aircraft is shown as a diamond shaped symbol along with a datablock showing callsign, flight level and second portion of the position update time. The flight heading and airspeed is indicated by the line pointing from the aircraft symbol. The information portion states that PPOS is set to 2 minutes, which means that the lines pointing from the aircraft extend as far as the distance travelled by this aircraft in 2 minutes at its current speed. This line is referred to as the aircraft speed vector.

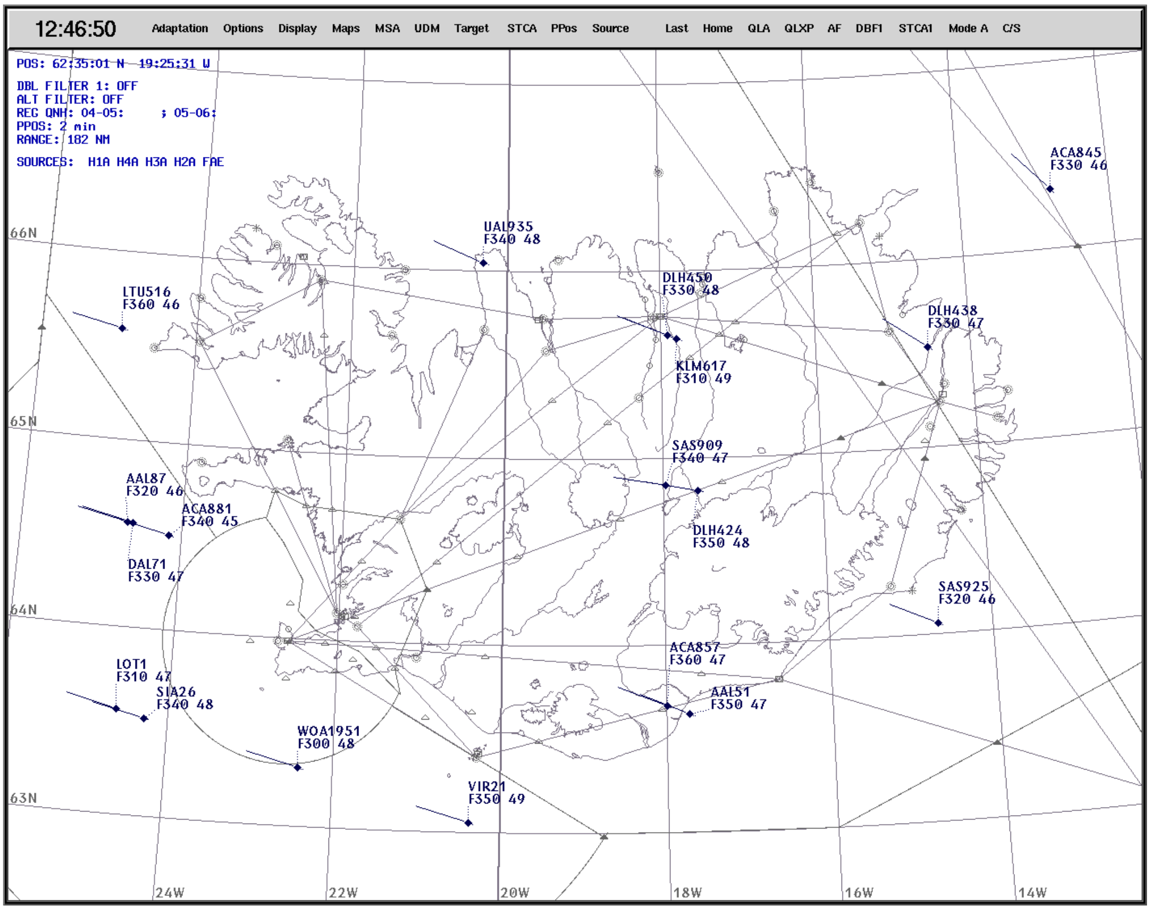


Figure 2.4: *Screenshot of the radar display system in use at the ICAA (from [3]).*

Flight Data The ICAA commissioned the Flight Data Processing System (FDPS) in 1989 and after a suitable testing period as a backup system it was taken into general use on April 12. 2002 [4]. The flight data is shown as an electronic version of the previously used paper strip system of operation. This system provides the controller with a clear overview of the traffic in his sector while using a familiar way to encode the information. Figure 2.5 shows a screenshot of this system. It is quite obvious that FDPS presents a great wealth of information to the controller. Full details of all the facilities are not covered here, but only the main points. First of all, the titlebar at the top of the window indicates that this instance of FDPS is controlling traffic in the East (E) sector. This already excludes all traffic not relevant to this sector, i.e. the aircraft that never enter it. The strips are organized based on flight level, showing F350 at the top and so forth, further excluding the aircraft not being managed by this controller or relevant to his work. When dealing with heavy traffic in the sector, the controllers will divide their effort based on flight level, so one controller only directs traffic between two flight levels. Each strip is divided into columns, each containing specific information. The first column contains most importantly the aircraft callsign (center, large), current heading (top left), aircraft model (top right) and airspeed (bottom right). The second column indicates which sector (East, West, North or South) is the controlling sector for that aircraft. The fourth column contains the flight level, showing any potential change as well. Flight FLI765 is an example of a flight in the process of landing, descending from

Figure 2.5: *Screenshot of the FDPS system in use at ICAA (from [3]).*

flight level F350 to sea level F000.

The remaining columns contain the waypoints on the flight plan, ending with the waypoint where control is passed to the next ACC, which may be the approach tower of an airport. Each waypoint is represented by geographical coordinates and the time estimated for which the aircraft passes the waypoint. The waypoints are also lined up in the columns corresponding with longitude coordinates, facilitating a quick scanning technique of each longitude for potential conflicts. To illustrate by example, flights KLM663 and AFR337 are assigned the same flight level F340 and both pass through 020W followed by 030W (eastbound). They do however not travel by the same latitude nor at the same time, KLM663 passes through 65N 020W at 12:25, followed by 66N 030W at 12:56 and AFR337 passes through 69N 020W at 12:00, followed by 70N 030W at 12:28, a full 240 nautical miles (445km) to the north and around 25 minutes earlier. This example also highlights the need to, not only keep track of altitude, latitude and longitude but also the time factor, the fourth dimension in the aircraft position.

Situation Display The Situation Display System (SDS) processes the data from FDPS and provides a graphical overview for the controller. Figure 2.6 shows a screenshot of this system. The similarities between SDS and the RDDPS are rather obvious. Each

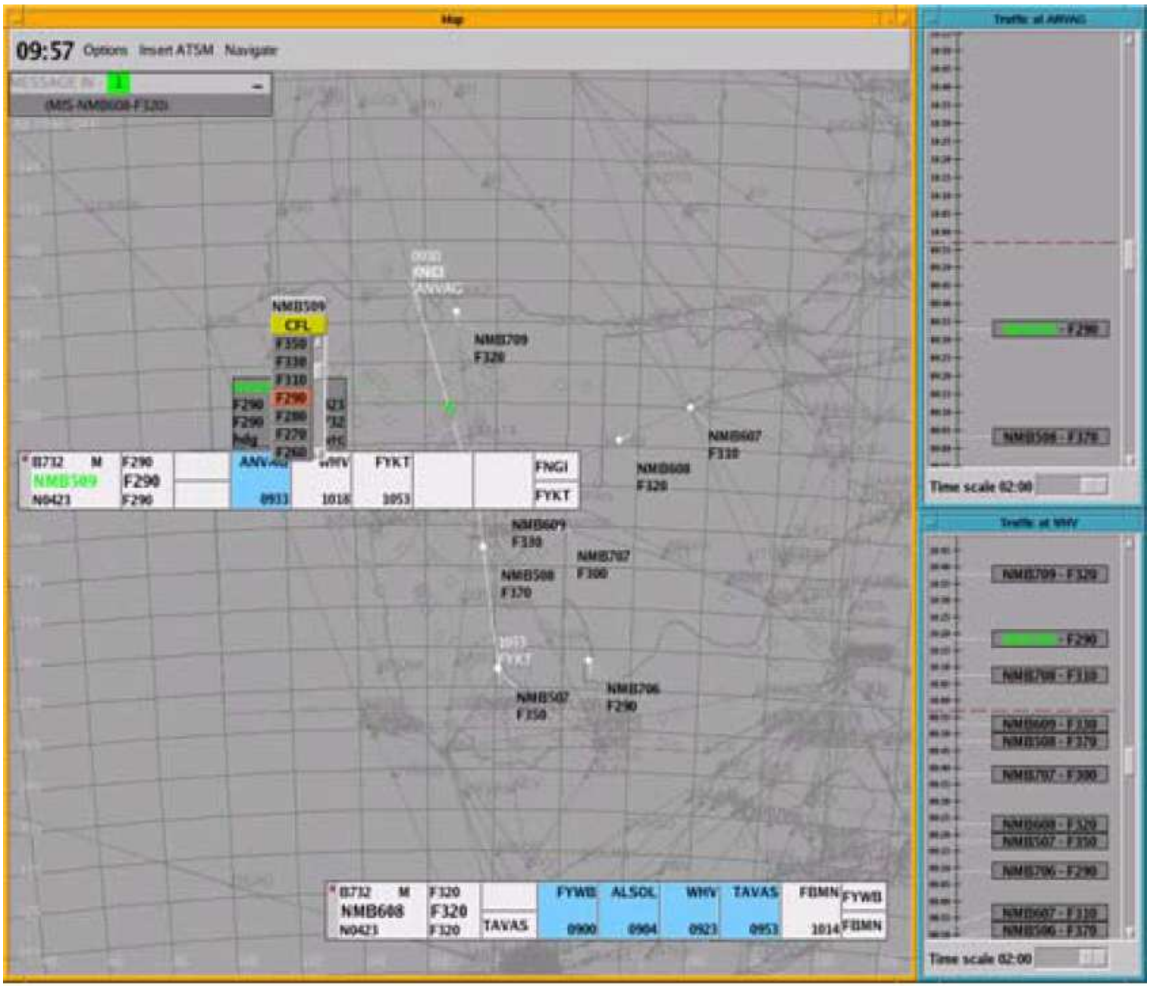


Figure 2.6: *Screenshot of the SDS system in use at ICAA (from [5]).*

aircraft is for instance also represented by a diamond shaped symbol and a speed vector as well as as a datablock just as in RDDPS. The SDS map however has in addition flight data strip available with the same basic information as in FDPS. The windows on the right part of the screenshot show the altitude arrangement of the flights relevant to the air traffic controller. It becomes clear that the SDS is an effort to merge the functionality of the FDPS with the user interface of the RDDPS to create a user interface suitable to present the FDPS data in a graphical manner familiar to the users.

Voice Communication The TwrCom Voice Communication System (VCS) is a multichannel voice communication system that enables simultaneous radio, telephone, internal communications and public address through a touch screen interface. Figure 2.7 shows a screenshot of this system. It is imperative that the controller is able to communicate with all aircraft under his control, as well as his colleagues both across the room or in other ACCs adjacent to his area of control. Even though this system is not directly part of this project, it is mentioned here since voice communication plays such a large part

Figure 2.7: *Screenshot of the TwrCom VCS soon to be taken into use at ICAA (from [5]).*

in air traffic control procedures. The screenshot shows a simple array of buttons, each of which can be used for a predefined direct call or for application specific functionality such as teleconferencing or similar. In addition to these, a numeric pad is always available for dialing a telephone number directly.

Text Communication CPDLC is a system used to provide the controller and pilot with a textual communications method. Figure 2.8 shows a screenshot of the system developed by Tern Systems. The screenshot shows CPDLC contact with three aircraft with

Figure 2.8: *Screenshot of the CPDLC part of the Datalink workstation from Tern Systems (from [5]).*

callsigns: BAW110, BAW112 and BAW120. There are some messages shown, both initiated by pilots as well as air traffic controllers with message time on the far right side. The messages from pilots are shown with a light background while messages from the controller is shown with a blue background. One example is the pilot of BAW110 requesting ascent to flight level F370 and the controller directing him to listen to the radio transmission for further instructions. The pilot then acknowledges that he has understood. It is quite evident from just this screenshot that the communication follows the rules of standard radio communications, using a predefined message format.

Automatic Surveillance ADS-B is an extended communication system used for automatic transmissions between computer systems embedded in aircraft and ATC computer systems on the ground. Figure 2.9 shows a screenshot of the system developed by Tern Systems. The more general term ADS is used in this tool, covering the older ADS-C

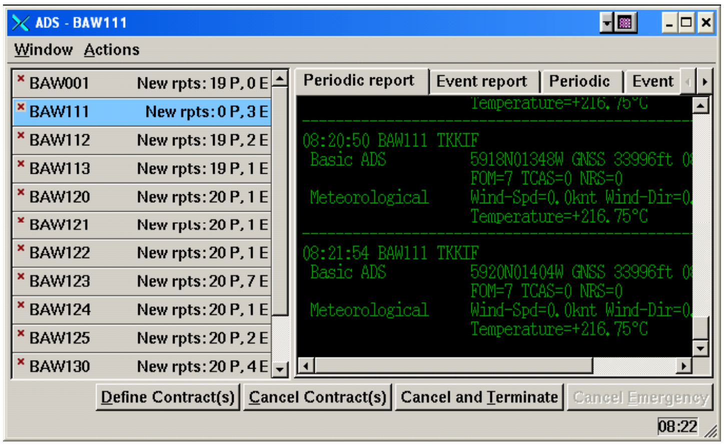


Figure 2.9: *Screenshot of ADS part of the Datalink workstation from Tern Systems (from [5]).*

(C stands for Contracts) as well. The difference between ADS-C and ADS-B lies in the communication protocol and the fact that ADS-C does not broadcast the position report, but instead negotiates a one-to-one contract between ground stations and airborne units. ADS technology is referred to here only as ADS-B since it is the newer form and considered more useful due to it's broadcast nature. The screenshot shows a list of aircraft to the left with all callsigns of the form BAWddd (where d signifies a numerical digit). The aircraft with callsign BAW111 is highlighted so the right part of the window shows it's two latest periodic reports. Each periodic report shows the time and aircraft callsign, basic ADS information as well as other information depending on the ADS-B transmitter on board the aircraft. In this example the two reports were sent within four seconds of each other. The aircraft moved from position 5918N 01348W to 5928N 01404W and maintained the altitude of 33,996 feet, or probably flight level F340. The message also shows the acronym GNSS signifying that these information were acquired by using a Global Navigation Satellite System, such as GPS.

2.3 Definition of Terms

Some of the following terms have been used throughout the above text, but it is always useful to precisely define common terms soon after their introduction. The follwoing definitions were mostly taken from [6] and [7] representing the common use of these terms. Further information on especially ADS-B can be gathered from [8]. Even more useful terms can be found in appendix A where the ordering is alphabetical instead of here where it is ordered subjectively by relevance and increasing compliexity.

Flight Level (FL) A surface of constant atmosphere pressure which is related to a specific pressure datum, 1013.2hPa, and is separated from other such surfaces by specific pressure intervals.
Altitude above sea-level in 100 feet units measured according to a standard atmosphere. Strictly speaking a flight level is an indication of pressure, not of altitude. Since all aircraft use the same method to measure air pressure, this ensures that the aircraft will use the same flight levels in each others close proximity.
Each is stated in three digits that represent hundreds of feet. For example, flight level 250, indicated as F250, represents a barometric altimeter indication of 25,000 feet.

Aircraft Call Sign A group of alphanumeric characters used to identify an aircraft in air-ground communications. For convenience, the term is frequently abbreviated to "call-sign" when used in appropriate contexts.

Indicated Air Speed (IAS) The speed of an aircraft as shown to the pilot by the airspeed indicator on board. IAS will differ from TAS at air densities other than some reference density. This is the speed used in pilot/controller communications under the general term "airspeed."

True Air Speed (TAS) The actual speed of an aircraft relative to the air through which it is flying corrected for temperature and air density. Used primarily in flight planning and en route portion of flight. When used in pilot/controller communications, it is referred to as "true airspeed" and not shortened to "airspeed."

Mach Number The ratio of speed to the speed of sound in the medium. For aircraft, the medium is air for which the speed of sound differs with temperature and to a lesser extent humidity. Mach number is widely used in air traffic control and is for example indicated as M080 for an aircraft travelling with a true airspeed at 80% of the speed of sound.

Heading The direction in which the longitudinal axis of an aircraft is pointed, expressed in clockwise degrees from North. Usually magnetic heading (with respect to the magnetic north pole), sometimes true heading (with respect to the geographic north pole).

Position (POS) Location of a point or an object considered as a point in 3-D space.
In air traffic control, the position of an aircraft is usually given as the geographical coordinates on the surface of the earth directly below it.

Current Flight Position A ground-referencerable representation, in x, y, z of the current position of a flight based on latest sensor data.

Reporting Point A specified geographical location in relation to which the position of an aircraft can be reported.

Nautical Mile (nm, NM) A measure used in navigation. The unit is equal to 1852 m.

Track 1- A target report resulting from the correlation, by a special algorithm (tracking) of a succession of radar reported positions for one aircraft. The report normally contains smoothed position and speed vector information.
2- Time sequence of state vectors of an object estimated by some real time filtering technique using radar plot data as input.

Route The 2D trajectory of an aircraft, expressed in terms of significant points, airways or geographical points.

Air Traffic Service Route A specified route designed for channelling the flow of traffic as necessary for the provision of air traffic services.

Way-point (WP) A specified geographical location used to define an area navigation route or the flight path of an aircraft employing area navigation. Sometimes also called route point or RP.

Flight Plan (FPL) Contains specified information provided to Air Traffic Services Units, relative to an intended flight or portion of a flight of an aircraft.

Current Flight Plan The flight plan, including changes, if any, brought about by subsequent clearances.

Flight Path The set of positions an aircraft has occupied, is occupying or is expected to occupy.

Aircraft Conflict Two aircraft are involved in an aircraft conflict if the distance between any probable position of one aircraft at a certain time, and any probable position of the other aircraft at that same time, will be less than the required separation criteria.

Separation Situation in which two aircraft do not violate the separation criteria.

Separation Criteria The required minimum vertical, lateral, and longitudinal distances between two aircraft in order not to be involved in an aircraft conflict, or a nominal routes overlap.

Separation Minima The minimum required distance between aircraft. Vertically usually 1000 ft below flight level 290, 2000 ft above flight level 290. Horizontally, depending on the radar, 3 NM or more. In the absence of radars horizontal separation is achieved through time-separation (e.g. 15 minutes between passing a certain navigation point).

Vertical Separation Separation between aircraft expressed in terms of flight levels or altitude between tracks.

Horizontal Separation Separation between aircraft expressed in terms of time, distance or angular displacement between tracks.

Lateral Separation Separation between aircraft expressed in terms of distance or angular displacement between tracks.

Longitudinal Separation Separation between aircraft expressed in terms of time or distance along tracks.

Mach Number Technique The term used to describe the technique of clearing turbojet aircraft to maintain specified Mach numbers in order to maintain adequate longitudinal separation between successive aircraft at, or climbing or descending to, the same level.

Procedural Separation The procedure of systematically examine flight plans for conflicts in order to keep aircraft separated.

Non-Radar Separation The separation used when aircraft position information is derived from sources other than radar.

Radar Separation The separation used when aircraft position information is derived from radar sources.

Transponder A combined receiver and transmitter whose function is to transmit signals automatically when triggered by an interrogator.

Radar Identification The process of correlating a particular radar blip or radar position symbol with a specific aircraft.

Radar Monitoring The use of radar for the purpose of providing aircraft with information and advice relative to significant deviations from nominal flight path, including deviations from the terms of their air traffic control clearances.

Radar Contact The situation which exists when the radar blip or radar position symbol of a particular aircraft is seen and identified on a radar display.

Radar Position Symbol (RPS) A generic term for the visual indication in a symbolic form, on a radar display, of an aircraft obtained after digital computer processing of positional data derived from primary radar and/or SSR.

Controller Pilot Data Link Communications (CPDLC) Sometimes referred to as Two-Way Pilot Data Link, is the means of communication between controller and Aircrew, using data link in conjunction with or instead of voice, for ATC.

Automatic Dependent Surveillance-Broadcast (ADS-B) A function on an aircraft or surface vehicle that broadcasts position, altitude, vector and other information for use by other aircraft, vehicles and by ground facilities.

Air Traffic Service (ATS) A generic term meaning: Flight Information Service (FIS), Alerting Service (AL), Air Traffic Advisory Service (ATAS), Air Traffic Control Service (ATCS), Area Control Service (ACS), Approach Control Service or Aerodrome Control Service.

Air Traffic Management (ATM) As defined by the Special Committee on Future Air Navigation Systems (FANS), a system consisting of a ground part and an air part, both of which are needed to ensure the safe and efficient movement of aircraft during all phases of operation. The airborne part of ATM, according to FANS, consists of the functional capability which interacts with the ground part to attain the general objectives of ATM. The ground part of ATM comprises the functions of Air Traffic Services (ATS), Airspace Management (ASM) end Air Traffic Flow Management (ATFM). Air traffic services are the primary components of ATM.

Chapter 3

Simulating an ATC Environment of Mixed Aircraft Separation

Having set the stage in the previous chapter, the basics of air traffic control should now be familiar to the reader. It is therefore possible to move on to defining the objectives of this project using the terminology used in air traffic control.

For the purpose of this project Amelia was modified to emulate the relevant part of the ATC environment. Therefore it is necessary to describe how Amelia emulates the ATC environment and how Amelia is constructed.

The following section 3.1 provides an overview of what was involved in this project and what aspects were simplified to make the solution more manageble. Followed by that in section 3.2 is a description of the visible features available in Amelia along with a discussion of the relevant components of the software. Much of the functionality available in Amelia is irrelevant to this project and is therefore not described here.

3.1 The Task At Hand

The general aim of this thesis is to explore how mixed separation can be represented on the situation display for the air traffic controller. To accomplish that a prototype is constructed using the classic software development cycle as known from various sources. A specific development cycle for prototyping is proposed in [9] and a slightly modified version is shown in figure 3.1. The problem domain has now been analyzed in the previous chapter soon to be followed in the next section by a discussion of the main prototyping tool available. It is now time to clarify our objectives after which the initial design is specified, implemented and evaluated by user testing.

The results of the evaluation is then used to revisit the analysis and propose a refinement to the design. This refinement is then implemented and user testing is performed again. This process usually continues until few or no problems are detected after evaluation or the software is retired. In the case of this prototype, two iterations are deemed sufficient to reach a conclusion due to the project time constraints.

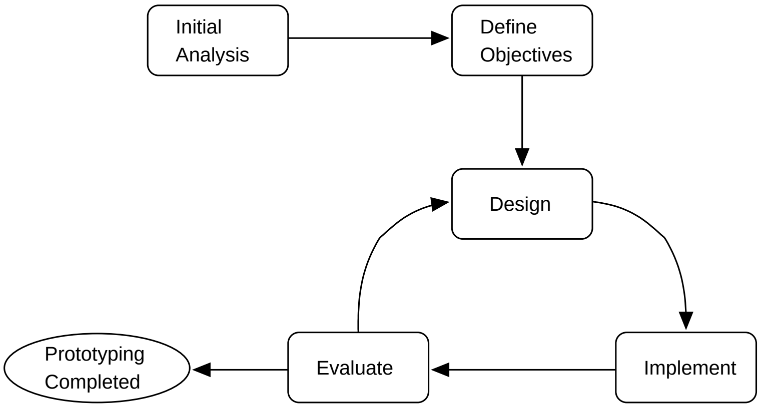


Figure 3.1: *The software prototype development cycle.*

Some simplifications have been made in order to be able to produce results. The goal is not to deliver a fully functional and robust system for air traffic control, but a sufficiently functional prototype of the working environment air traffic controllers are used to. Even though a robust design is presented, the main goal is to have a representative population of users test the prototype and provide feedback. Since the prototype is not subjected to correctness requirements it is acceptable to simplify the handling of aircraft separation.

The lack of distinction and clarification mentioned in the very beginning in chapter 1 are the core of what needs to be explored here. In addition to these concepts, a secondary goal of this project is to look into how conflicts may be presented to the controller. When a detailed model of the aircraft flight path is available along with separation criteria at each stage of the flight, all known trajectories may be examined for intersections in a robust mathematical fashion. That provides the opportunity to discover and report any conflicts. This reporting needs further examination since it is currently only partially available in the controllers' working environment.

Testing these improvements is not trivial. Air traffic controllers are busy people with stringent demands in their work environment. Getting actual professionals to test these improvements requires authorization from supervisors and must be done without taking too much of their time. Fortunately for this project, seven of them volunteered and were able to take the time to deliver precious feedback to the modifications proposed. Three of the air traffic controllers took part in the first round of testing where they collectively as a group performed the tasks and answered the questions of the test procedure. Five air traffic controllers then took part in the second round, including one of the participants from the first round. The second time around the controllers performed the tasks individually.

3.2 Overview of the ATC Simulator Amelia

As a means to create a otherwise feature complete applicaton with additional prototype features the source code to Amelia was provided by Tern Systems. This meant that the air traffic controllers could sit down in front of a running instance of the application and it would be as close to their normal workstation environment as possible. Any development of basic features such as map drawing functionality is thus eliminated as well as aircraft motion and flight plan editing. Although Amelia has no direct support for ADS-B features, it supports the correct way to mark an aircraft as ADS-B equipped. This needs to be exploited correctly by the prototype.

Figure 3.2 shows the complete Amelia application with all subwindows enabled. The sub-

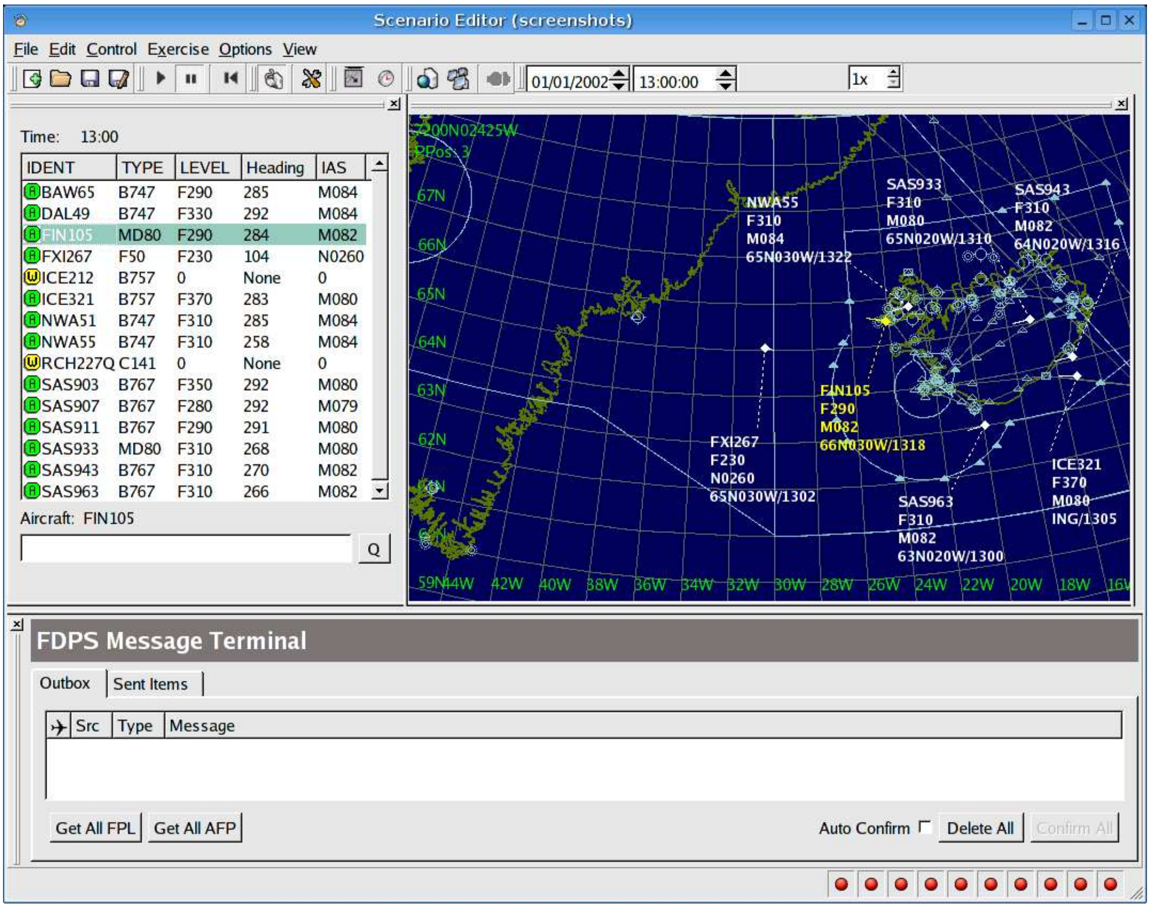


Figure 3.2: *Screenshot of the ATC simulator Amelia.*

windows are: Pseudo-pilot on the left, FDPS Message Terminal at the bottom and the Map on the right. Pseudo-pilot provides complete access to aircraft controls, hence the name Pseudo-pilot. It displays a list of all aircraft which are defined for the simulation scenario showing what type of aircraft it is, flight level, heading and airspeed. Some of the aircraft are waiting to become active, so they are marked with a (W) symbol to the far left. The aircraft that are active in the scenario are marked with a (A) symbol and are visible somewhere on the map. The FDPS Message Terminal subwindow makes it possible to send and receive messages with the FDPS subsystem. Neither Pseudo-pilot nor the FDPS subsystem are used for this project at all and are merely shown here for demonstration purposes.

The Map display shows something familiar from section 2.2. The way each aircraft is marked with a symbol, a directional speed vector and a datablock showing the aircraft's state has previously been discussed in relation to RDDPS and SDS on figures 2.4 and 2.6 respectively. The datablock shows some different information, namely the aircraft callsign, flight level and airspeed as well as next waypoint coordinates and estimated time of arrival at that point.

Besides that the top left corner of the map shows two lines of text. The first one shows the geographical coordinates for where the mouse cursor is currently placed. The other one shows the current PPOS setting, which is shown as 3, meaning that here, just as with RDDPS, the length of the aircraft speed vector shows the distance and heading that the aircraft would travel in 3 minutes.

The selected aircraft is highlighted in yellow, as well as being selected on the Pseudo-pilot list. The currently selected aircraft is FIN105 near the western edge of Iceland. When an aircraft is selected it provides a popup menu for controlling it in the same manner as Pseudo-pilot.

There are a few elements on the program toolbar, situated below the menu bar at the top of the window, that should be mentioned. The three buttons no. 5, 6 and 7, marked with "play", "pause" and "rewind" are for controlling the progress of the exercise scenario. Their function is to resume, pause and restart the simulation respectively. On the far right of the toolbar there are three other elements that look like a calendar, a clock and a multiplication factor. The first two are for setting the time of the exercise scenario while the third one is for setting a clock speed factor. It makes it possible to jump back and forth in time and set the exercise to run faster than real time to quickly see how a scenario develops. The exercise date and time are meaningless in any other way than that they provide an internal passage of time.

Amelia is a fairly sizable software product, complete with features suitable for training air traffic controllers. The program is written in C++ using the QT library for graphical user interface widgets. Not counting user interface code, the 730 source code files weigh in at around 150.000 lines including comments.

The QT library by Trolltech Inc. of Norway is a cross platform application development framework. As well as being a feature rich GUI library, QT has many facilities for networking and string manipulation, some of which are used in Amelia. Being cross platform means that the same code base can be used for compilation for virtually any operating system platform. This includes Linux, Windows, Mac OS X as well as the many Unix variants, such as HPUX, Sun Solaris and the BSD family. Since the QT library is not a vital part of this project it is not described here in any detail, but the interested reader can find more information at [10].

Some programmer utilities from the Boost libraries are also used in Amelia. Those include pointer reference counting for automatic object cleanup, a technique known as smart pointer, as well as type casting utilities. Those are fairly minor enhancements, but important since they help avoid the major pitfalls of the C++ programming language. The purpose of the Boost libraries is exactly to provide a base collection of stable, peer reviewed software components that represent the established practice in the field of C++ programming. This effort was started by members of the C++ Standards Committee after informal discussions during a committee meeting in France. At the time of this writing ten of the Boost libraries are being discussed for possible inclusion in a future C++ Standard, including the smart pointer library used here. The source code is distributed with explicit permission for inclusion in commercial products and is therefore extremely valuable to software companies distributing proprietary software products such as Amelia. More information on the Boost libraries can be found at [11].

Besides the QT and Boost libraries, extensive use is made of the Standard Template Library. STL should be well known to any C++ programmer, but further information can be found at [12].

The source code for Amelia is further organized into local libraries which are then linked to the main executable in the final stage of compilation. Their function can roughly be grouped together into four broad categories. There are the base libraries for aiding in routine programming tasks. Common libraries are shared between the various software products from Tern Systems and are mainly for geographical calculations as well as air traffic functionality. There are simulator libraries for manipulating a simulated air traffic environment as opposed to a live environment where data is fed into the system from outside sources such as radar stations. Finally there are the libraries for the graphical user interface, such as drawing a geographical map or displaying a dialog window. A detailed list of libraries can be found in Appendix, section D.1.

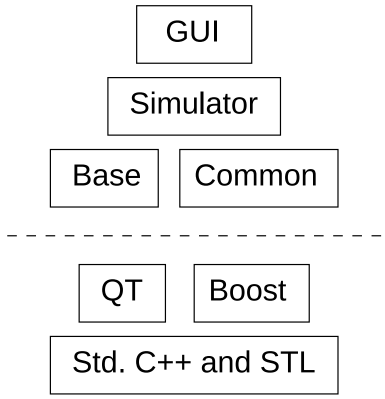


Figure 3.3: *The libraries of Amelia. Each one builds upon the libraries below or beside it (approximately).*

The relationship between the various libraries are schematically sketched in figure 3.3. The figure shows the approximated way in which the libraries gradually build on the feature set of the others below or besides them.

The classes that are related to this project and deserve a closer examination are mostly related to the graphical user interface since the purpose of this project is to perform user testing of improvements to the graphical elements.

The main window of the Amelia application shown in figure 3.4, only displaying the Map portion, is encapsulated in the class `ScenarioEditor`. This is a QT application which among other things loads settings and prepares the graphical map showing the air traffic. The classes responsible for drawing the map are shown in figure 3.5 along with the relationship between them. The `BasicMap` class provides basic map drawing facilities, including taking data files with geographical information and drawing coastlines and other distinguishing features. Extending this functionality with air traffic services is the `AtsMap` class which is, along with `BasicMap`, shared among some other Tern products and does therefore not completely implement an air traffic simulator. This is accomplished with the `SimMap` class which fulfills the ATS role using purely simulated functionality.

The `AtsMap` class contains, among other information, a list of `AircraftHandler` objects, `aircrafts`, that provide the graphical rendering of all the active aircraft in the exercise scenario. This is also a class shared with other Tern products and the simulated functionality is

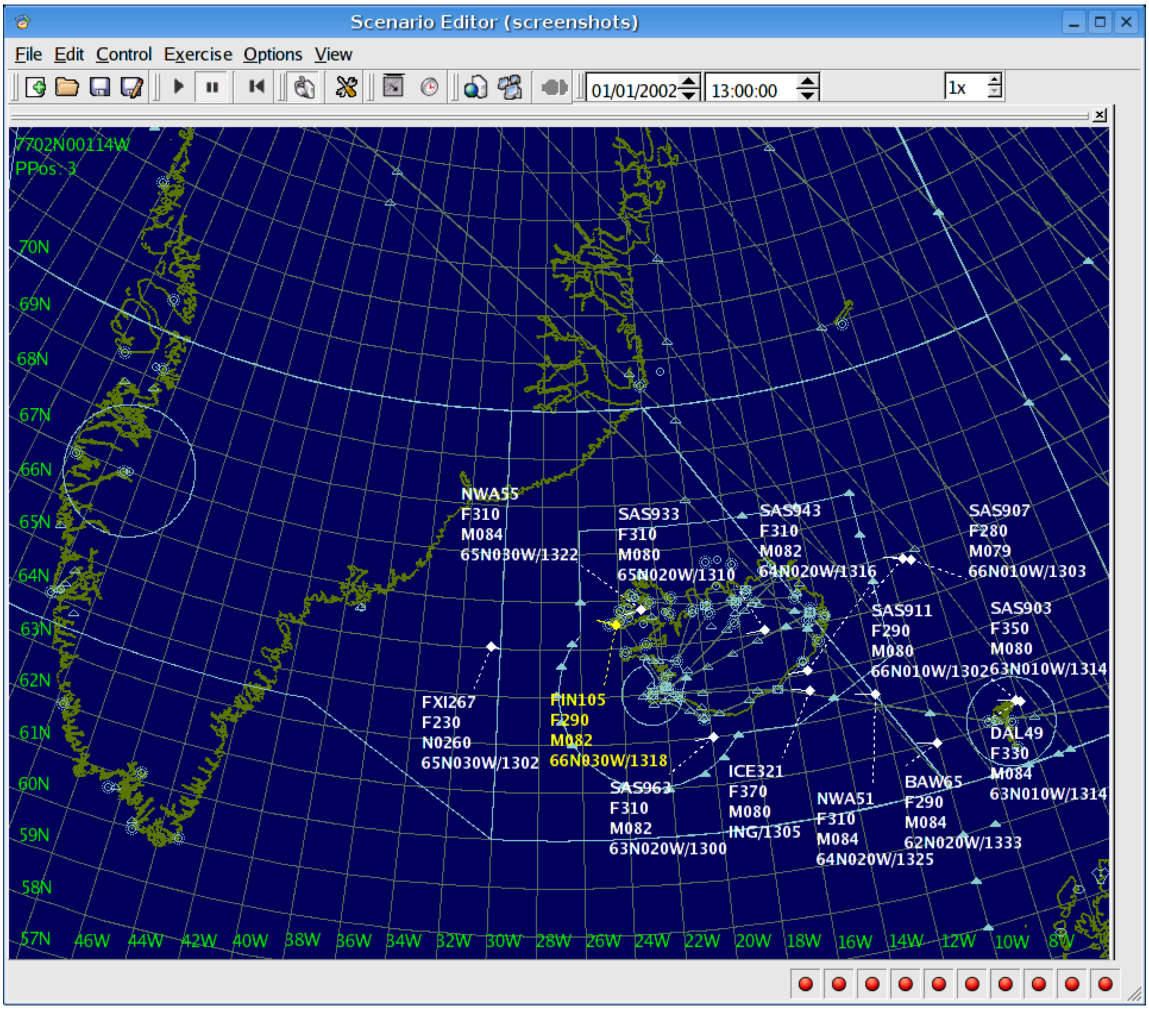


Figure 3.4: *Screenshot of the map portion of Amelia.*

handled by the `SimAcHandler` class (which stands for Simulated Aircraft Handler).

Settings and state are managed centrally in Amelia by a tree structure referred to as Property Map (PM for short). The PM architecture is based on XML to a certain degree. It makes it possible to organize data in a tree structure with `n` child nodes along with a number of properties for each node in the tree. A `Property` node in the tree may be considered a leaf carrying a value. Figure 3.6 shows how this tree structure can be accessed end edited from the Amelia application. The left side shows a tree structure of the Property Map nodes where the highlighted node would have the complete path `"/Scenario/Radars/H1"`. The center shows some of the properties attached to the highlighted node, properties which are recognized by the user interface. The right side shows all properties attached to the node, including those not recognized by the user interface.

The highlighted property has the complete path `"/Scenario/Radars/H1/POSITION"` and the value `"640118N0223921W"`. That translates to the fact that the radar station named "H1" of the active scenario has the position 64 degrees 01 minute and 18 seconds North, 022 degrees 39 minutes and 21 seconds West. This can also be recognized in the center section where the Position group shows the Lat(itude) and the Lon(gitude) in the decimal represen-

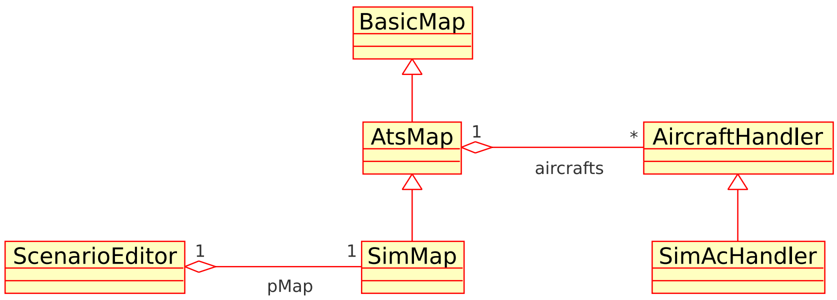


Figure 3.5: *The map related graphics classes of Amelia and their relation.*

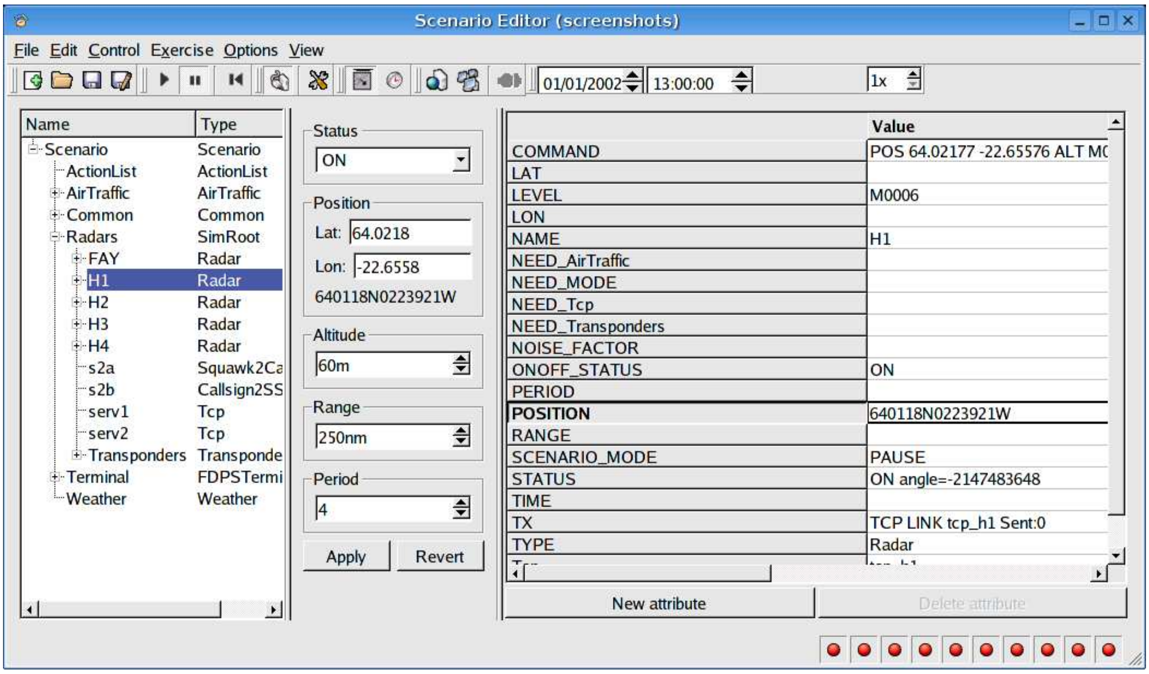


Figure 3.6: *The Property Map as visible in the Amelia application.*

tation as well as the textual representation from the `"POSITION"` property.

The most prominent classes used to implement the PM architecture are shown in figure 3.7, while much of the PM architecture has been omitted in order to keep focus on the parts that are essential to this project.

The main class is `PropertyMap`, which is used to manipulate the `Property` objects stored in the PM tree structure. Such objects are nothing more than name-value pairs which need to be stored together. The `PropertyMap::Ref` type is defined as a `SmartPtr` containing a `PropertyMap` as `pointee`, in essence making it automatically garbage collected. As a programming tool the template class `SmartPtr` implements reference counting and automatic object cleanup to avoid memory leaks. This is a class implemented a little bit differently than the Boost equivalent introduced earlier, due to specific requirements of the Property Map structure. The two essential operators make it possible to refer to `PropertyMap::Ref` the same way as regular pointer types, only accessing the shared `pointee` member. This relationship between the three classes is shown in the figure as merely associations of undeterminate type.

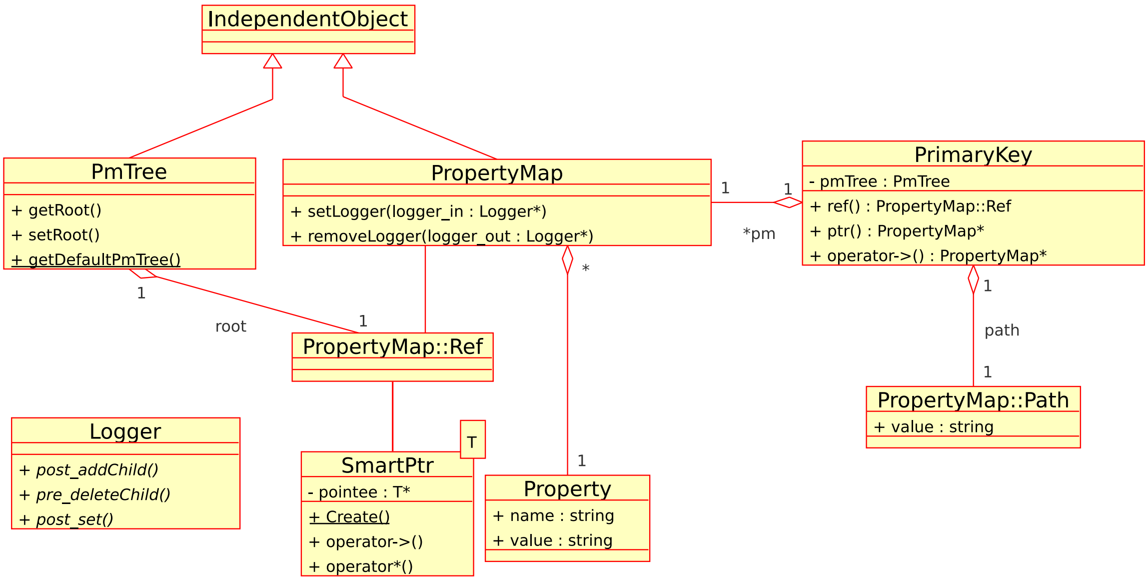


Figure 3.7: *The Property Map classes.*

The `PmTree` class stores a pointer to the root `PropertyMap` node in the tree as a `Property-Map::Ref`. Both `PropertyMap` and `PmTree` inherit from `IndependentObject` making it possible to treat `PmTree` as if it were the `root` `PropertyMap` node of the `PmTree`. In many cases only one `PmTree` is active for the whole application. It then makes sense to define a single default `PmTree` which may be accessed by static class methods such as `getDefaultPmTree()`.

Changes made to the tree are advertised using an event mechanism facilitated by the `Logger` interface. This is a pure abstract class which provides three callback methods used for propagating changes throughout the application. If a class is meant to refresh it's internal state whenever changes are made to the tree, it inherits from the `Logger` class and implements the callback methods. This new object is then registered as a logger with the appropriate `PropertyMap` node in the `PmTree` and receives notification of changes from that moment on.

The `PrimaryKey` class provides a path based entry into the PM tree, instead of a pointer based entry by the `PropertyMap` class. This is merely a memory optimization, but it bears mention because of the interchangeableability between `PrimaryKey` and `PropertyMap` objects. This path class `PropertyMap::Path` is exactly the mechanism which enables the path translation discussed above in relation to the `"POSITION"` property.

Chapter 4

Analysis of Mixed Aircraft Separation

A closer examination of separation standards is now in order. The topic will not be covered here in minute detail, but instead introduced to a sufficient degree. That opens up for a discussion of the requirements for working with air traffic control environments where mixed standards of separation are applied.

Section 4.1 contains a short overview of separation standards as well as a simplified model. Functionality related to mixed separation which is desireble to implement is covered in section 4.2 and an abstract view of the proposed solution is presented in section 4.3.

4.1 Separation Standards

We point to section 2.3 as well as appendix A for the correct definitions of the different separation standards.

Precise separation standards may be found in [2], which is in fact the authoritative standard in the U.S. A small section of this document provides a glimpse of what aircraft standards look like:

```
   8-7-4. LATERAL SEPARATION
In accordance with Chapter 8, Offshore/Oceanic
Procedures, Section 4, Lateral Separation, apply the
following:
 a. 60 NM or 1 degree latitude between:
  1. Supersonic aircraft operating above FL 275.
  2. Aircraft which meet the MNPS and which:
   (a) Operate within MNPS airspace; or
   (b) Are in transit to or from MNPS airspace;
or
   (c) Operate for part of their flight within
   MNPS airspace but are cleared to operate immediately
   above or below such airspace for a portion of their
   flight.
```

```
b. 90 NM or 1 and 1/2 degrees latitude between
aircraft operating:
 1. Within WATRS;
 2. Between the U.S., Canada, and Bermuda;
 3. West of 55° West between the U.S., Canada,
 or Bermuda and points in the Caribbean ICAO
 Region.
c. 120 NM or 2 degrees latitude between aircraft
not covered by subparas a or b above.
```

This is just one section, applying to narrowly limited cases. There are multiple such cases, both for lateral, longitudinal and vertical separation.

For simplification purposes, the concept of a reserved airspace is now defined to each side as well as in the direction of flight in both directions, i.e. equal lateral and longitudinal separation based on distance. Furthermore, only three simplified types of separation will be considered:

Procedural 60 Nautical Miles.

ADS 30 Nautical Miles.

Radar 10 Nautical Miles.

These values are deemed to represent a fairly nominal expectation for separation standards.

The following simple set of rules can then be used for determining which separation applies:

1. If the aircraft is within radar coverage, Radar separation is in effect.
2. Otherwise if the aircraft is equipped with ADS-B technology, ADS separation is in effect.
3. Otherwise Procedural separation is in effect.

4.2 Desired Functionality

The need to manage aircraft for which different rules of separation apply poses challenges to the software tools that the air traffic controllers use in their work. What this project aims to accomplish is to propose a solution to the most pressing deficiencies. The following key areas are identified.

Quick Overview What is needed in the operating environment of air traffic controllers is an addition to the situation display that enables the controllers to quickly and intuitively assess the situation and gain an overview.

Detailed Separation Information When the controller needs further information about separation it should be readily and quickly available and show clearly which separation standards apply.

Display Conflicts If lack of separation should occur, the controller should immediately become aware of the situation and how separation is broken.

4.3 Proposed Changes

Two new concepts need to be introduced as a part of the proposed changes. The Programme for Harmonized Air Traffic Management Research in EUROCONTROL [13] (PHARE) was conducted between 1989 and 1999 by European Organisation for the Safety of Air Navigation (EUROCONTROL) in cooperation with a number of European research establishments assisted by national authorities. It was aimed at developing and testing advanced concepts in air traffic management, including air traffic control. Among the ideas discussed and tested within the framework of PHARE was subdividing the aircraft flight path into phases and regarding it as a three dimensional tube through which the aircraft travels. Each phase represents a stage in the flight where most aircraft travel properties may be considered constant, but at least one may be changing. This means for instance that a change in altitude would signify a new phase, during which only the altitude is changing. As soon as the new altitude is reached a new phase begins where all properties are constant again. This is of course a simplification, but it has proved to be beneficial for modeling the state of each aircraft during their flight.

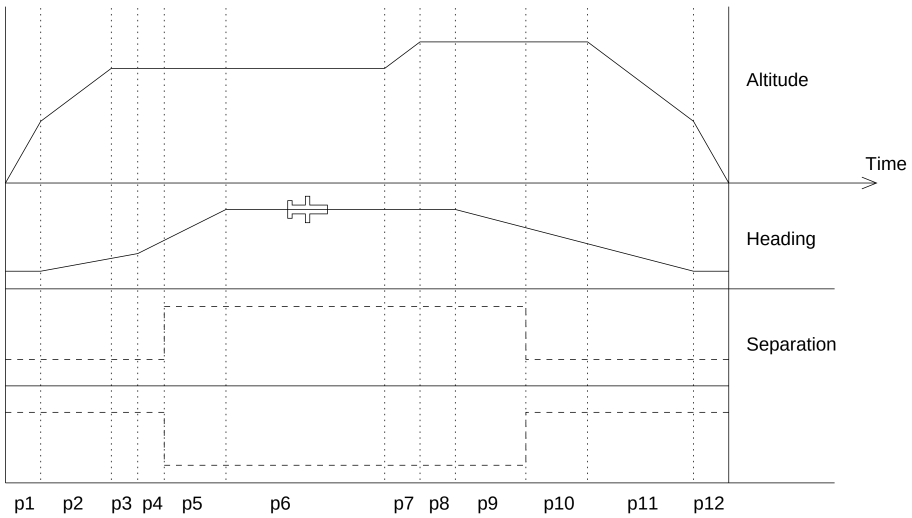


Figure 4.1: *Overview of how a flight is divided into phases.*

This idea is illustrated in figure 4.1 for altitude, heading and separation for a typical flight. Missing from this illustration is e.g. changes related to speed. What is shown is a phase division into 12 phases marked as p1 through p12 with takeoff at the beginning of phase p1 and landing at the end of phase p12. The top portion shows how altitude changes with time during the flight and how that contributes to dividing the flight into phases as well as how unchanged altitude spans many phases. The middle shows how the flight heading changes either independently or coinciding with altitude changes. At the bottom the changes in minimum separation are also illustrated. This shows how the aircraft is within radar range during the initial four and again during the final three phases. During phases p5 - p9 the aircraft is then outside radar coverage and different rules of separation apply with a larger separation distance.

The details concerning each phase of this flight are as follows:

1. The aircraft is taking off and climbing at maximum with a steady heading.
2. The aircraft has not yet reached cruising altitude, and is therefore still climbing after having turned towards a new waypoint given by the air traffic controller.
3. The aircraft has now reached cruising altitude.
4. The aircraft has now reached a waypoint and has turned towards the next one, maintaining steady altitude.
5. The aircraft has now exited radar covered area and is subject to rules of separation which require more distance from other aircraft. Meanwhile, the aircraft is maintaining steady altitude and heading.
6. After having reached a waypoint in the flight plan and changed heading, the aircraft continues at the same altitude towards the next waypoint.
7. It becomes necessary to ascend to a higher cruising altitude for some reason known only by the pilot and the air traffic controller.
8. After ascending, the aircraft continues along it's way.
9. A waypoint has been reached and heading is changed towards the next one.
10. The aircraft has now reached radar coverage for the destination and radar separation may now be used.
11. The aircraft is now nearing the destination airport and has begun descending in preparation for landing.
12. The aircraft is now on final approach and is about to land after having reached the approach waypoint.

The above is of course a simplified view of the progress of a flight, but it does include many of the elements common to any flight. As a model of a typical flight it may be considered a sufficiently accurate depiction of such.

As shown in the above example, one of the properties which might signify a new phase is a change in separation. When entering an area covered by radar, separation may be applied with shorter distance between aircraft, than is necessary outside radar coverage. This may also happen if an aircraft equipped with ADS-B crosses between two airspaces under the control of different aviation authorities where one is prepared to handle ADS-B and the other is not.

This is where the concept of tubes comes in. A tube may be viewed as the three dimensional tunnel through which the aircraft travels, with the outer boundaries determined by the minimum separation necessary. The tube follows the flight path and the flight separation can therefore be efficiently modelled using phases. As time passes and the aircraft travels through the tube, it basically sits inside a bubble of protected airspace which moves along the tube, marking the outer boundaries of it. This makes for a 4 dimensional trajectory which may be tested for intersections with other trajectories.

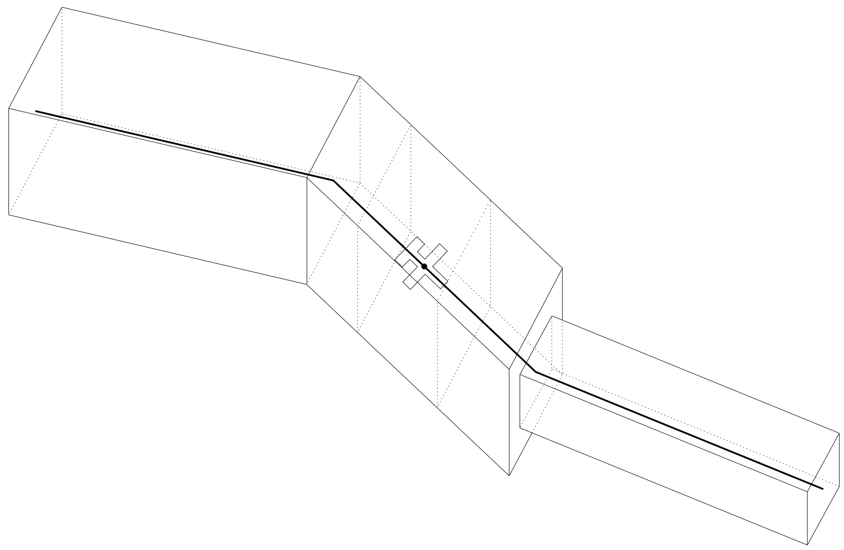


Figure 4.2: *Simplified sketch showing how a tube modeled with flight phases may be visualized in three dimensions.*

This is illustrated in figure 4.2 where the tube for one aircraft is sketched in three dimensions. The aircraft is about to enter a phase where the rules of separation allow a shorter separation distance, and the tube therefore becomes narrower. This might be considered to represent the phases p8 - p10 from the example above. The longitudinal separation minimum is marked on either side of the aircraft in dotted lines. This is the cube shaped bubble surrounding the aircraft at the depicted time.

Based on these concepts of tubes and phases, and after discussions with air traffic controllers and experienced ATC software programmers the following is proposed as a general solution to the needs discussed in the previous section.

Quick Overview For a quick overview of which rules of separation apply to which aircraft, the symbol for the aircraft are to be modified to reflect their situation.

Detailed Separation Information Displaying the required separation of each aircraft can be accomplished by drawing the outlines of the reserved airspace along the projected flight path. Put another way, the horizontal projection of the tube related to the flight will be drawn.

Display Separation Conflicts When two parallel lines cross two other parallel lines, an area roughly the shape of a skewed rectangle is formed. This area should be highlighted very clearly to attract the controllers attention quickly.

These changes are sketched in figure 4.3 to give an approximate idea of how they would appear. The two aircraft, one depicted as a diamond and the other as a circle, are subject to

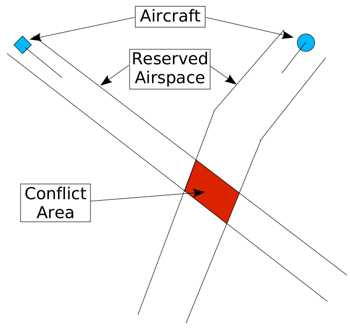


Figure 4.3: *Initial sketch of the proposed changes with explanatory labels.*

different separation standards. The one on the left has the necessary equipment to allow for reduced lateral separation and the outlines of the reserved airspace are therefore narrower. A conflict area is highlighted here in red, the color most naturally associated with danger. It should now be possible to quickly realize which rules apply to which aircraft, given that the air traffic controllers undergo the proper training. Two potentially important tools are also provided to the controllers, one which can show exactly what area is required by each aircraft and another which can show predicted dangers along this area.

As with any software model of reality, there are some shortcomings to this approach. First of all, this only shows the lateral separation, neither longitudinal nor vertical separation can be shown in this way because of the nature of the map. Only the view from above can be shown on the map and even though it does not offer a graphical view of either flight level or longitudinal separation, it can still be used to bring to the attention of the controller conflicts involving these separation standards.

Chapter 5

Prototype Design, Version 1.0

Based on the analysis presented in previous chapters, a design for both the internal representation of data as well as user interface elements can now be introduced. An overview of these areas are introduced in section 5.1, followed by specifically the graphical user interface elements that need changes in section 5.2. Finally a model of the class structure is presented in section 5.3. The fully developed model also includes some classes already present in Amelia showing where the new classes fit in to the overall structure of Amelia.

5.1 Planned Changes

Due to the way the prototype is based on existing code, it is considered important to make an effort to isolate the changes made to Amelia. This is mainly for being better able to evaluate the newly developed portion on its own instead of mixing it with the Amelia source code. This also enforces the notion that any additions to Amelia are experimental and are at a prototyping stage. Since Amelia is already highly object oriented, this will be achieved through developing new classes which inherit from existing Amelia classes. Any new concepts are then to be fitted into the existing Amelia framework.

Areas that need modifications:

Graphical Aircraft Representation It is clear that since the intention is to mark the aircraft with different symbols, their graphical representation needs to be modified.

Access to New Features In order for new features to blend in with the user interface it is necessary to provide the elements needed to activate and deactivate the new features. The goal has always been to produce a *rich* prototype, one that interacts with the user close to the level of a finished product.

New concepts that need implementation were introduced in section 4.3:

Tubes The separation boundaries of each flight may be considered as a tube, a sort of a tunnel reserved for a specific flight. This idea was visualized earlier in figure 4.2.

Phases Dividing a flight path into phases is a central new concept since the existing functionality is to only keep track of flight plan waypoints and project the expected position at a desired point in time. The two approaches are somewhat different since the desired information is not the same. Phases are needed to be able to compare complete flight paths. A sequence of phases is an efficient way to represent a tube. This concept was likewise introduced earlier by the example surrounding figure 4.1.

Tube Processing It is not enough to merely represent the flight path as a tube. The tubes must be compared and examined for conflicts. This should be handled by a separate module since this role does not fit into any of the existing parts. Amelia does not implement any conflict detection where this functionality could be integrated.

Graphical Tube Representation How a tube for a specific flight is drawn should be separated from the internal representation as well as the tube processing. It should be controlled by the module in charge of drawing the aircraft, but the drawing of separation boundaries should be handled by an independent module. The aircraft graphics module is responsible for drawing the aircraft symbol at the specific instant as well as simple lines representing the known flight plan and the projected flight path. The finer points of separation rules should not be the concern of the aircraft module. This is however a matter of software design taste as the opposite position can easily be argued. A rough sketch was presented earlier in figure 4.3.

5.2 Changes to the User Interface

Some of the changes already introduced need to be implemented as part of the graphical user interface. These are:

Graphical Aircraft Representation Great care needs to be put into selecting appropriate symbols for each separation rule. Taking into account the Ground Human Machine Interface portion[14] of PHARE as well as the existing set of symbols in use at ICAA, tree symbols are selected. Figure 5.1 shows the planned symbols. These symbols are

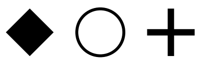


Figure 5.1: *The aircraft symbols for a) Radar coverage, b) ADS-B equipped aircraft, c) traditional "procedural" management.*

selected because of their recognizability, simplicity and visually distinct features. The first one, representing radar coverage, will be referred to as "diamond" since it resembles the symbol for the diamond suit of playing cards. This symbol is currently used for this function in the RDDPS and it would therefore be recognizable by the air traffic controllers. The second one, representing ADS-B equipped aircraft, will be referred to as "circle" for obvious reasons. There is no known precedent for showing this type of aircraft on a situation display so the symbol is selected as one that is sufficiently distinct from the others. The third symbol, representing aircraft managed with traditional procedural separation standards, will be referred to as "cross". This is also a symbol that controllers are familiar with from the SDS the same usage. The relative sizes of the symbols used here may be slightly different from what the controllers expect or are used to from the other systems. For the first prototype, all symbols are the same size as is evident from figure 5.1.

Access to New Features Amelia already provides popup menus for each aircraft as well as the map, so access to the new features should be provided there. The feature requiring access is activation of separation boundary display. Figure 5.2 shows how the popup menus look in their original form. Various functionality of each aircraft can be ac-

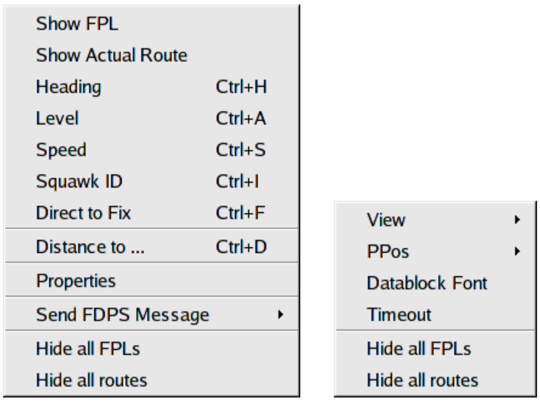


Figure 5.2: *The original two popup menus for a) each aircraft and b) the map itself when no aircraft is selected.*

cessed using their popup menu and some functionality which is applicable to all aircraft at once can be accessed using the map popup menu. Showing and hiding the flight plan (FPL) and the flight path (Actual Route) is already provided via these menus. Adding the possibility to show and hide separation boundaries as the third option directly after any "... route" menu point seems straight forward.

Graphical Tube Representation Since Amelia displays a view from the top, it is obvious that separation boundaries can only be drawn as lines parallel to the flight path. Since the separation boundaries are meant to represent the separation from other aircraft required by the aircraft, it is parallel the actual path instead of the flight plan. The opposite would be the case if the separation boundaries were an indicator of how far from the flight plan the aircraft may deviate.

5.3 Model of Changes

In order to implement the desired changes, the following classes are conceived:

SepAcHandler Inheriting from `SimAcHandler`, this class handles the specific separation related issues of graphical aircraft representation as well as handling a change of symbol for when different rules of separation apply.

SepMap Inheriting from `SimMap`, this class gives access to new features related to showing separation boundaries on the simulator map.

Tube Being a new class, it encapsulates a flight tube.

TubePhase Another new class, this one represents a single phase of a flight path.

SepManager The centralized tube processing takes place in this class, keeping its information updated by subscribing to the Logger event model described in section 3.2.

TubeHandler This class implements the graphical tube representation and is controlled by `SepAcHandler`, but it has knowledge of how `TubePhases` are organized in order to extract the relevant information from a `Tube`.

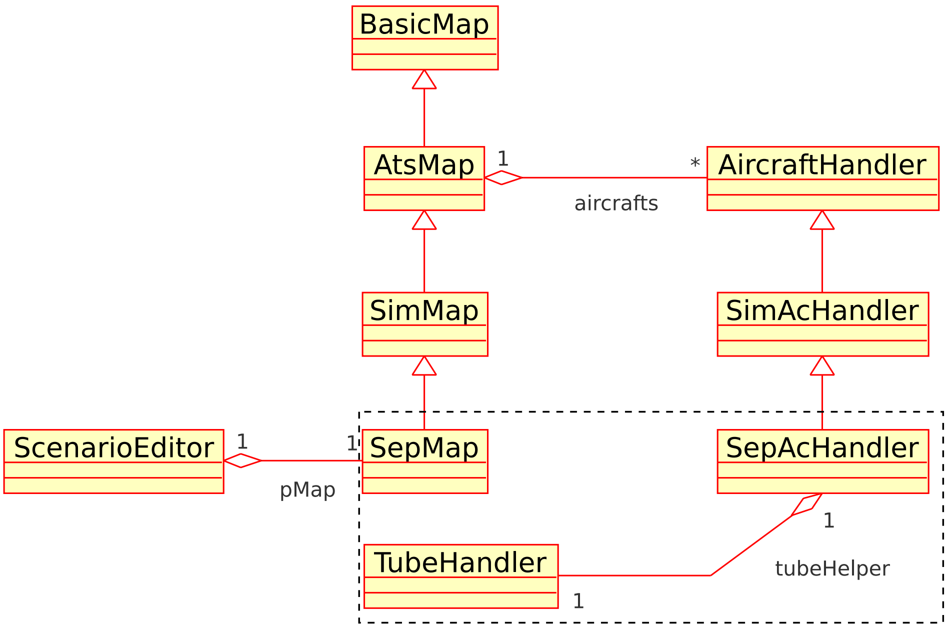


Figure 5.3: *The introducton of new graphical classes to Amelia.*

Figure 5.3 shows how the graphical classes fit into the existing framework of Amelia classes presented earlier in section 3.2. There is a direct relation to figure 3.5 which only shows the classes that already existed in Amelia, whereas the new classes are highlighted by a square drawn with a dashed line.

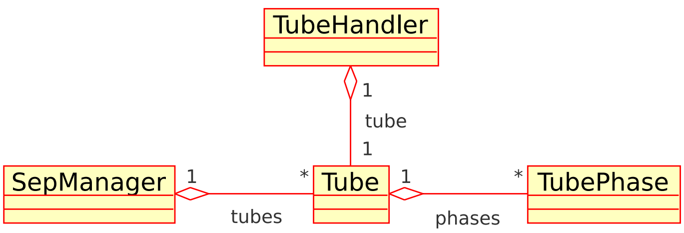


Figure 5.4: *The new tube related classes.*

The new classes that are needed to handle the tubes and the flight path phases are shown separately in figure 5.4. The `TubeHandler` class is shown in both figures since it is in fact intended to be both a graphical class as well as a tube manipulator.

Besides the conceptual design presented above, the following minor problem with the existing design needs to be fixed satisfactorily.

As mentioned briefly in section 3.2 the list of `AircraftHandler` objects, `aircrafts`, is stored in the `AtsMap` class. It is however the responsibility of the `SimMap` object to populate the list with concrete objects of the preferred type and in the case of `SimMap`, that is `SimAcHandler`. In order to enable the modifications, the list needs to be populated with objects of `SepAcHandler` type.

Quick solutions to this dilemma could be to either empty the list and repopulate it in `SepMap` or change the implementation of `SimMap` to not populate the list and do it for the first time in `SepMap`. Both approaches, however, are wasteful and inappropriate.

The solution chosen was to introduce a new group of classes to act as "Creator of `AircraftHandlers`". This is similar to the Abstract Factory pattern defined in [15]. The classes introduced are: Abstract base class `AircraftHandlerFactory`, as well as `SimAcHandlerFactory` and `SepAcHandlerFactory` which provide concrete implementations.

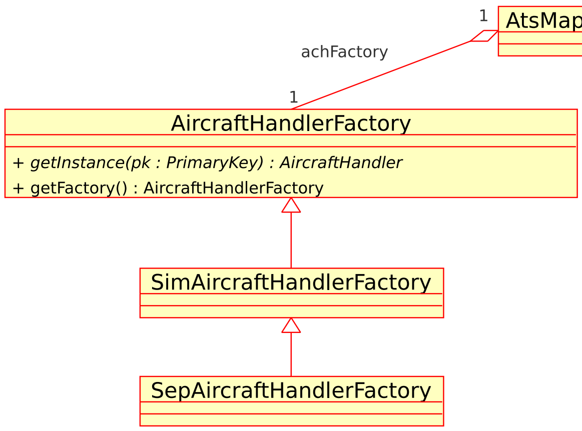


Figure 5.5: *The `AircraftHandlerFactory` classes introduced to the design.*

Figure 5.5 shows how the `AircraftHandlerFactory` classes fit in as an interface for creating `AircraftHandler` objects. The `AtsMap` class is modified to contain a member variable, `achFactory`, of type `AircraftHandlerFactory`. When the `AtsMap` object is created it needs to takes an `AircraftHandlerFactory` as parameter to be used to populate the list of `AircraftHandler` objects. That way it is ensured that while being created, the `AtsMap` object is able to reach a consistent state.

The places where the `SimMap` class adds items to the `aircrafts` list are now modified from

```
aircrafts.push_back( SimAcHandler::create(pk) );
```

to

```
aircrafts.push_back( achFactory->getInstance(pk) );
```

The new version is almost identical, but instead of referencing the `SimAcHandler` class directly, it now uses a generic mechanism which is initialized along with the enclosing `SimMap` object to produce the correct object type.

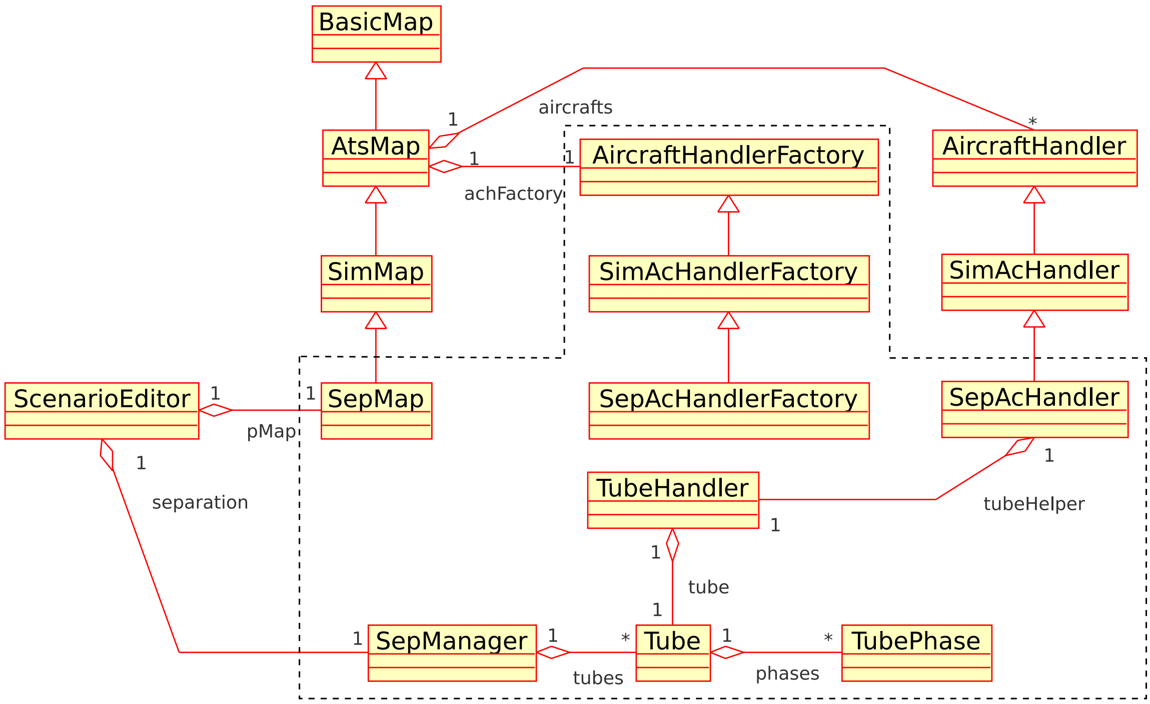


Figure 5.6: *Overview of all classes introduced to the Amelia and how they fit in with existing classes.*

After putting it all together the resulting complete design can be seen in figure 5.6 again with new classes surrounded with dashed lines. This is the result of joining figures 5.3, 5.4 and 5.5 together. This figure shows that there is a dual encapsulation of `Tubes` by `ScenarioEditor`, via both `SepManager` as well as `SepMap`. This is intentionally so, since there is in fact no guarantee that all aircraft are visible on the map at all times, e.g. flights that have not yet taken off or entered the airspace are hidden from view until they become relevant. It is not ensured that `aircrafts` contains flights that are not visible on the map. These flights may only exist in the central `PmTree`. The `SepManager` object which is activated by `ScenarioEditor` needs to know about all flights crossing into the active control area within a specific time frame.

Therefore the following holds true at any time:

```
SepManager::tubes.size() >= AtsMap::aircrafts.size()
```

Chapter 6

Prototype Implementation, Version 1.0

Before presenting the complete implementation of the first prototype, a shortcoming in relation to the design must be pointed out. The Map related classes proved to be too time consuming to complete, so only a partial implementation of the Tube classes was possible. The decision was made to focus on the critical modifications necessary for the graphical user interface in order to be able to complete the user testing. Without a functional graphical user interface, the project would have been less useful.

In the following sections all the implementation details are covered, both the portions that played a vital part in the user interface as well as the unused portions of the Tube related classes. Starting in section 6.1 with screenshots to show what portions of the user interface were changed, followed by an implementation overview in section 6.2.

6.1 Changes Made to User Interface

For demonstration purposes figure 6.1 shows what graphical elements were implemented for the first prototype. Visible elements include one aircraft of each separation standard, procedural, radar and ADS-B. All aircraft are showing their separation boundaries and it is clear that the widths differ by which separation standard applies.

The Popup menus will be changed to what is depicted in figure 6.2 Simply 2 resp. 1 line added to each of the menus.

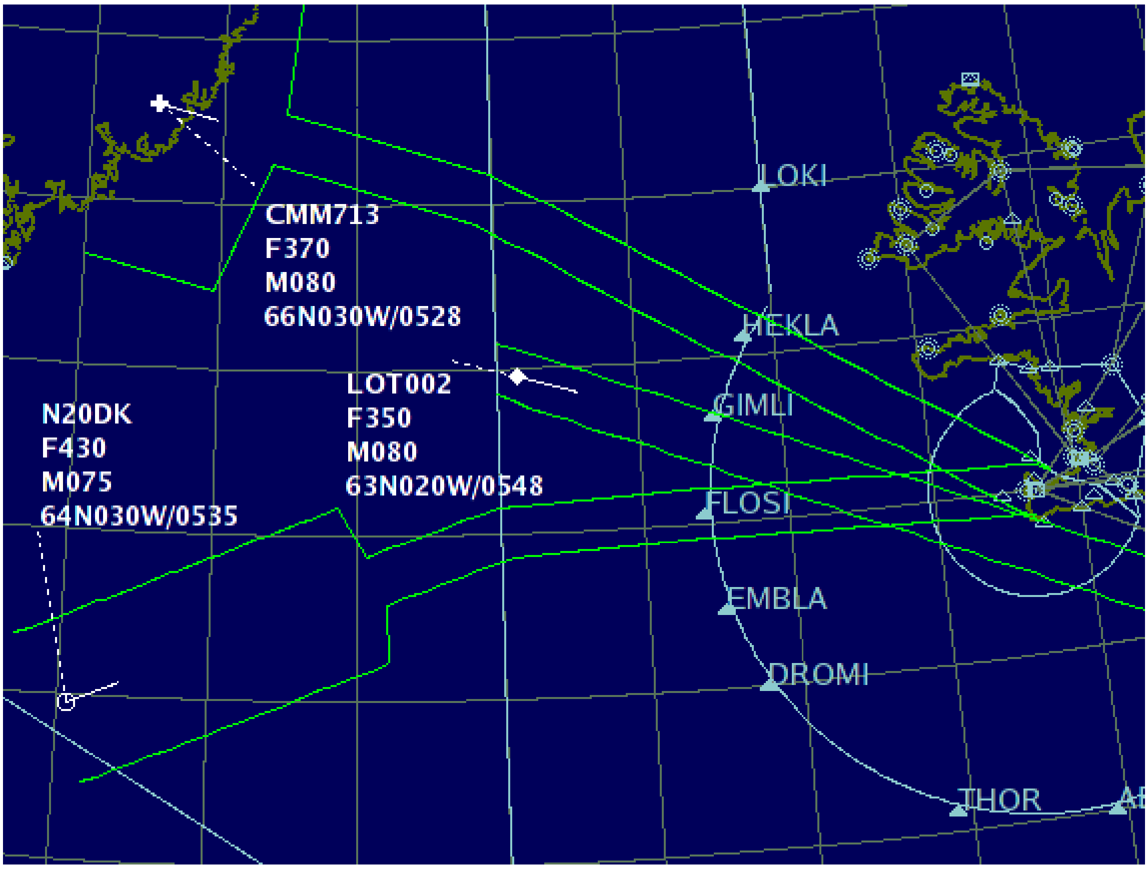


Figure 6.1: *Screenshot of all features specific to the first prototype.*

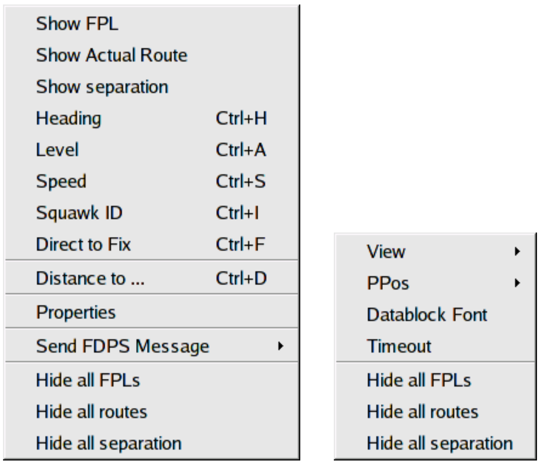


Figure 6.2: *The two popup menus in the first prototype for a) each aircraft and b) the map itself when no aircraft is selected.*

6.2 Implementation Overview

A new namespace is added: sep and the following classes added.

```
SepManager
Tube
TubePhase
```

The following new classes are added to the preexisting namespace ats_map.

```
SepMap
AcHandlerFactory
SimAcHandlerFactory
SepAcHandlerFactory
SepAcHandler
TubeHandler
```

The source code for the first prototype is not included, but appendix C contains detailed class documentation and source code listing for these classes as they appear in the second prototype.

Chapter 7

User Testing, Round One

User testing is the heart of this project. Here it is now, the core of the matter itself.

Starting off in section 7.1 with a discussion about the procedure governing the first user test, we carry on to presenting the questions and tasks put before the users in section 7.2. Results are presented in section 7.3 followed by a discussion in section 7.4 on the conclusions that may be drawn from the results. All the original Icelandic language source material can be found in appendix B.

7.1 Procedure

The method selected for testing the prototype is the "Cooperative Evaluation" discussed in [16], chapter 32. A test coordinator is present while the participant answers questions and solves tasks. The test coordinator writes down the answers and takes notes of his observations of the participant. The coordinator may then offer clarification, either if the participant asks or is deemed by the coordinator to be in need of further information. After the test itself, any clarification that remains for the test coordinator is taken care of.

For this test there were three participants cooperating on solving the questions and tasks at the same time. This provides for the possibility of more discussions and sparring amongst the participants creating an atmosphere where ideas are more likely to surface. All three participants were highly experienced air traffic controllers, both in their profession as well as in the role of user interface testers.

The participants were seated together by a desktop computer with a 19" CRT monitor running Amelia with a preconfigured exercise scenario ready to be run. Some introduction to the topic and short verbal instructions were made, including that this is not a test of them, but of the software. Before starting the test, an introductory text was presented and the participants urged to read it in detail. This text is shown in figure reffig:usertest1-intro-en in it's original form and scale. This text was designed with the goal of providing all the necessary information for a participant to be able to test the prototype. Any discrepancies from their work environment is introduced as well as the new additions so they know what features to focus on.

Each question or task was presented one at a time to the participants, printed separately on a A5 page. Figure 7.2 shows how the first question looks, also in it's original form and scale. While the participants ponder the question or solve the task, they have this piece of paper directly in view. That way they are able to revisit the question or task during the course of their solution of it and are not distracted by other questions or tasks before or after.

Having gone through all the questions and tasks the participants gave some recommendations for improving the test procedure and were soon on their merry way after approximately a one and a half hour process.

Rules of Separation on the Controller Map

The purpose of this project is to examine how best to show different rules of separation on the airtraffic ontroller map. This is done by modifying the ATC simulator Amelia from Tern Systems Inc.

About the simulator
The following operations are available in the simulator:
- Popup menu for the map or a plane
  - The map menu is accessed by clicking the right mouse button on the empty map
  - The plane menu is accessed by clicking the right mouse button on the symbol or the datablock for the plane.
- View the plane flight plan
  - The menu point “Show FPL” on the plane popup menu shows the flight plan for the plane with coordinates by each point on the flight plan.
- To move the map center click the middle mouse button (the mouse wheel)
- Zoom in and out with the mouse wheel.

The simulator differs from the work environment in the following ways, among other:
- It is possible to remove the flight plans of all planes from the map by selecting the menu point “Hide all FPLs” on either the map or the plane menu.
- The datablock shows some other information
  - Line 1 shows call ID
  - Line 2 shows flight level
  - Line 3 shows mach number
  - Line 4 shows the next point on the flight plan “/” arrival tima at that point
- Radar range is simulated, the average range of each radar station is used to determine if a plane is within radar range.

Changes related to this project:

1. The plane symbols change by the following simple rules.
   - If the plane is within RADAR the radarsymbol is shown: ♦
   - Otherwise if the plane has ADS equipment the ADS symbol is shown: ○
   - Otherwise the usual “procedural” symbol is shown: +
2. The boundaries of minimum separation can be shown for each plane separately.
   - These boundaries mark the distance the plane requires to be successfully separated from any other plane.
   - Only lateral separation is shown.
   - The airtraffic controller chooses to show or remove the separation boundaries by right clicking the plane and choosing “Show separation” from the menu that appears.
   - The airtraffic controller can remove all separation boundaries by right clicking the empty map and choosing “Show separation” from the menu that appears.

Figure 7.1: *The introductory text presented to all test participants.*

1. Can you find examples of planes with all three different rules of separation in effect?

Figure 7.2: *The format for each question, actual size is A5 page.*

7.2 Questions and Tasks

The questions asked and tasks presented to the participants were the following, presented here along with a rationale for their purpose.

1. **Can you find examples of planes with all three different rules of separation in effect?**
   This question is aimed at having the participant identify the three symbols used for the aircraft. The initial application window was opened in a way so that it exposed at least one of each kind.

2. **Show the separation boundaries for planes with callsigns ICE652, CGIRE, ACA844 og N96LT.**
   This task is aimed at making the participant familiar with the separation boundaries.

3. **Clear the separation boundaries from question 2.**
   This task is aimed at making the participant more familiar with the separation boundaries, especially the different ways in which they can be hidden.

4. **The paths of the planes with callsigns CMM713 and GRL781 intersect. Is the separation sufficient?**
   This question involves that the participant uses the familiarity with separation boundaries to investigate the answer. Figure 7.3 shows this situation with the separation boundaries of GRL781 enabled as well as the flight plan for both. The situation shows two aircraft on a parallel track, opposing direction, where one is ADS-B equipped and the other is not. Since GRL781 is within the separation boundaries of CMM713, there exists a separation conflict between them even though CMM713 is outside the sepa-reation boundaries of GRL781. It only matters that one is too close to the other.

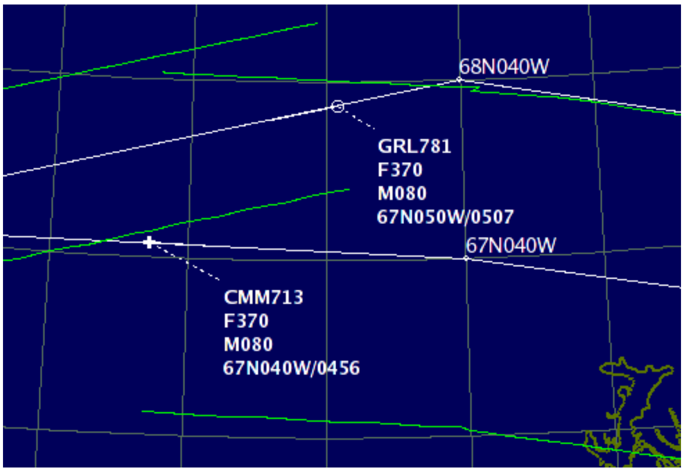


Figure 7.3: *Separation conflict for flights CMM713 and GRL781.*

5. **The paths of the planes with callsigns BAW284 and ACA844 intersect. Is the separation sufficient?**
   This question involves that the participant becomes even more familiar with separation

boundaries to investigate the answer. Figure 7.4 shows this situation with the separation boundaries of both aircraft enabled as well as their flight plans. The situation shows two ADS-B equipped aircraft on a parallel track where the separation boundaries do not actually help in finding the conflict. This example was deliberately chosen to demonstrate the weakness of this approach. The way participants can see that there is a conflict is that both aircraft are estimated to cross over the same coordinates within 8 minutes of each other, a violation of longitudinal separation. Flight GRL608 has no impact since it is traveling at lower flight level so it is greyed out in this picture.

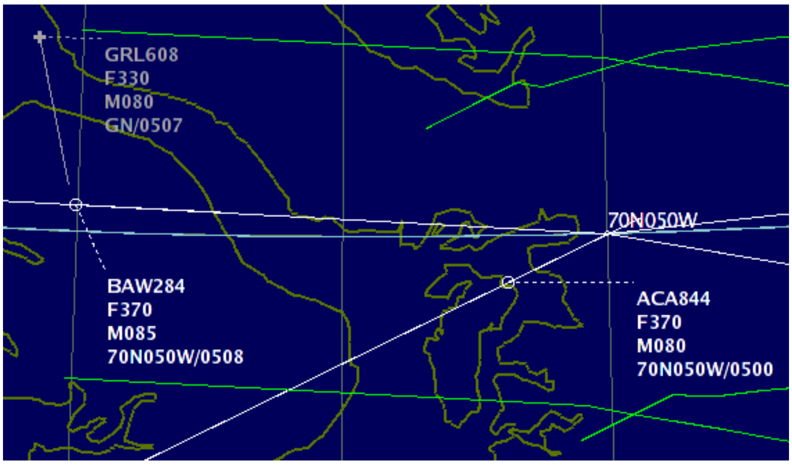


Figure 7.4: *Separation conflict for flights BAW284 and ACA844.*

6. **The plane with callsign ACA896 now requests flight level F330. Find out if the separation remains sufficient.**
   This task involves that the participant becomes even more familiar with separation boundaries to investigate the solution. This involves finding any potential conflicts by hand or by using their preferred method. There are in fact two aircraft which conflict with ACA896 at flight level F330, GRL608 and VIR20, only GRL608 does so sooner. Figure 7.5 shows this situation with the separation boundaries of ACA896 enabled as well as both their flight plans. The situation highlights the usefulness of the improvements when making decisions regarding altitude change.

7. **Is it helpful to see the aircraft symbols change according to the rules of separation that apply?**
   This is the first question asking the participants of their opininon regarding the improvements. At this point some of the aircraft have moved in or out of radar coverage so their separation minimum has changed accordingly.

8. **Are the aircraft symbols sufficiently different to be able to differentiate between the rules of separation that apply?**
   Asking about the clarity of the clarification feature is quite important.

9. **Do you find the operations to show and clear separation boundaries satisfactory?**
   This might give som unexpected results, yet the usage of popup menus has become quite standardized.

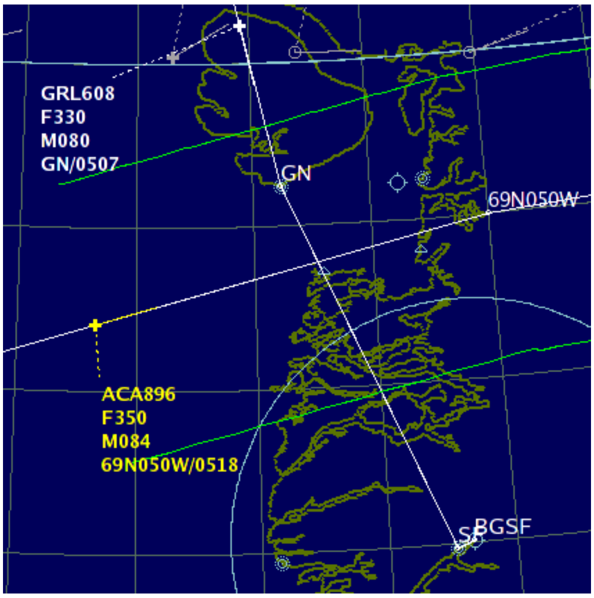


Figure 7.5: *Potential separation conflict for flights ACA896 and GRL608 at flight level F330.*

10. **Do separation boundaries show something that you could not see before?**
    This question touches on the possibility that this improvement may in fact not be an effective improvement.

11. **Should separation boundaries be shown less protruding, e.g. a softer color in the background?**
    The bright green color was chosen because of the sense of urgency it implied as well as the way it made the boundaries stand out.

12. **What do you think of this way to show separation boundaries on the map?**
    This is the first open ended question on the list. Getting thoughts and new ideas is highly desirable.

13. **Are these changes useful in assessing separation?**
    This question tries to dig deeper into the participants opinion of the improvements.

14. **Are these changes sufficient in assessing separation and if not, what is missing?**
    This question tries to dig even deeper and also asks an open ended question for elaboration and new ideas.

15. **Would it be helpful to see an aircraft's estimated time of arrival at each point in the flight plan?**
    This question was not actually part of this project, but was included to ask about a new feature Tern Systems was planning.

16. **What would you change?**
    Finally, the ultimate open ended question.

7.3 Results

The following are excerpts from the notes and observations made by the test coordinator while the participants answered the questions and solved tasks in section 7.2. The original notes in Icelandic are included in appendix section B.1.1.

1. After some discussion, 3 examples are located. Some comments about the question. Mention that the cross and the diamond could be made more different.
2. Succeed in showing the separation boundaries. Remarks made about ADS-B equipped planes within radar range are indistinguishable from other planes within radar range, but which are not ADS-B equipped.
3. The designated computer operator selects each aircraft and hides separation boundaries. Test coordinator points out the "Hide all" functionality, remarks are made about that that is as should be expected. Popup menu is criticized for being too crowded with unfamiliar menu points on account of it being a simulator instead of the work environment.
4. Participants discover the correct answer based on the flight plan, but afterwards take a look at separation boundaries.
5. Problematic to use flight plan for this example. Functionality, from their usual work environment, which would be used for this case is missing. Separation boundaries are used, correct conclusion reached. Remarks are made about the map becoming crowded when 2-3 aircraft show their flight plan and separation boundaries. The designated computer operator selects "Hide all" to resolve that situation.
6. Two aircraft are found and using flight plan and distance measurements, correct conclusion is reached. Again the commen about missing functionality from their work environment.
7. All agree it is helpful, but that there needs to be clear distinction between symbols. Remark that the cross is too much like the diamond it could turn out to be dangerous to trust the symbols. Suggest special attention in case radar coverage is suddenly lost.
8. All participants agree that the circle is fine, but the cross is definitely too similar to the diamond. One suggests that if the cross would be larger then the difference would be sufficient and the others agree. Suggestion from one of the others on a change of color is not supported by the remaining two. Remark that open and filled diamond already has distinct meaning in the RDDPS system.
9. "Yes, absolutely fine".
10. Make the statement that this image was previously only available in the air traffic controller's mind, but "now it can be printed". All agree that the map quickly fills upp with lines. Separation boundaries are useful, but only for 1-2 aircraft at a time.
11. Two distinct views: "Screaming colors encourage you to keep the map tidy", while on the other hand too bright colors demand too much attention. Remark that if separation

boundaries are shown for around 10 aircraft, it becomes impossible to follow what is going on and softer a color would hardly help in that position. Could be useful for watching one aircraft over a longer period, using a softer color. Remark that color schemes are already user defined in the work environment, this should be the same. The color depends on the background color. Finally they agree that green is fine, but a softer color would be better.

12. All agree that this is a quick way to see how separation may change. One points out the aspect of proper training. It turns out that they weren't quite sure what the boundaries are supposed to show, but jointly make an educated, correct, guess. One suggests that conflict points should be painted on the boundaries. Another suggests that examining two specific aircraft together whould be a clever way to use this.

13. Remark that this could be quite useful for monitoring aircraft which only pass through the sector.

14. Partly answered in no. 12. Additionally the matter of radar identify is not addressed in the current version of the prototype. When a plane is heading into radar coverage, radar separation should be clearly shown as an estimation until the aircraft has been positively identified on radar. Remark on the need to take fully into account all rules of separation, but for this purpose it is acceptable. The concept of examining two aircraft specifically is revisited and further endorsed.

15. Not relevant here.

16. Remark that datablock being too large and might benefit from reorganization. Otherwise see 12. and 14.

7.4 Conclusion

Some of the issues raised by the participants are too extensive to realistically be able to address. The following issues were selected for improvement:

Reduce Popup Menu The aircraft popup menu is the primary tool for manipulating the visible aircraft. Having simulator functionality on appear on the menu is distracting for the participants, enough so that most of them mentioned it. Reducing the menu to the bare minimum should be a trivial modification, but improve on the participant experience with the simulator.

Reorganize Datablock The aircract datablock contains information unfamiliar to the participants or in a different order than they are used to. The Datablock is also too big and gets in the way of viewing aircraft traffic. This should also be a trivial modification which improves the participant experience.

Confirmed Radar ID The participants noted that the standard practise when applying radar separation minima is to only do that if the aircraft has been identified properly. This is not what the prototype does and needs to be corrected.

Distinguish Aircraft Symbols All the participants agreed on the notion that there was too little difference between the cross and the diamond symbols. The suggested solution was to enlarge the cross.

The first two of these are purely cosmetic, but it should be noted that the participants in the first test were more experienced at testing user interfaces than the ones who were known to be willing to participate in the second test. Therefore the decision was made to bring Amelia even closer to the user interface the participants were used to.

The last two on the other hand are more related to the functionality of the improvements and the usability of the user interface. Taking into account confirmed radar identification was not planned until after receiving feedback from the test participants.

Chapter 8

Prototype Design, version 2.0

Following up on the conclusions drawn from the results of the first user testing, the prototype will now be improved and subsequently retested.

Section 8.1 contains a review of the changes that need to be done to refine the prototype. Section 8.2 then describes the improvements to the user interface. Since there was no fundamental change in the implementation model, no section analogous to 5.3 will be presented here.

8.1 Refinement Changes

As identified in section 7.4, the following items need to be refined:

Reduce Popup Menu The unnecessary pilot substitution elements will be removed.

Reorganize Datablock The datablock will be made more compact.

Confirmed Radar ID The Squawk ID function will become radar identify.

Distinguish Aircraft Symbols The cross will be made larger and thicker and the circle larger.

8.2 Changes to the User Interface

Figure 8.1 shows the refined aircraft symbols.

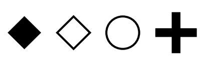


Figure 8.1: *The aircraft symbols for a) Confirmed Radar coverage, b) Estimated Radar coverage, c) ADS-B equipped aircraft, d) traditional procedural management.*

Recall how figure 6.2 showed how the popup menus looked for the first prototype. The aircraft menu was found to be too filled with unnecessary items. The items "Heading", "Level", "Speed" and "Direct to Fix" are all for simulating pilot actions, so they will be removed. To "Squawk ID" is an action where the pilot blinks the transponder for the air traffic controller to be able to identify the aircraft on radar. This functionality will now be expanded to positively identify the aircraft on radar.

Chapter 9

Prototype Implementation, version 2.0

Implementation details for the second prototype are not as extensive as the first time around. This time the implementation was mainly minor tweaking, mainly around the separation boundaries.

Section 9.1 covers the user interface changes, while appendix C contains class documentation as well as source code for review.

9.1 Changes Made to User Interface

For demonstration purposes figure 9.1 shows what graphical elements were implemented for the second prototype, which is taken a few moments after figure 6.1 only with the second prototype and includes a new procedurally separated aircraft after CMM713 came within radar range. Visible elements include one aircraft of each separation standard, procedural, confirmed radar estimated radar and ADS-B. All but one aircraft are showing their separation boundaries and it is clear that the widths differ by which separation standard applies and there is a difference for confirmed and unconfirmed radar coverage.

The Popup menus will be changed to what is depicted in figure 9.2

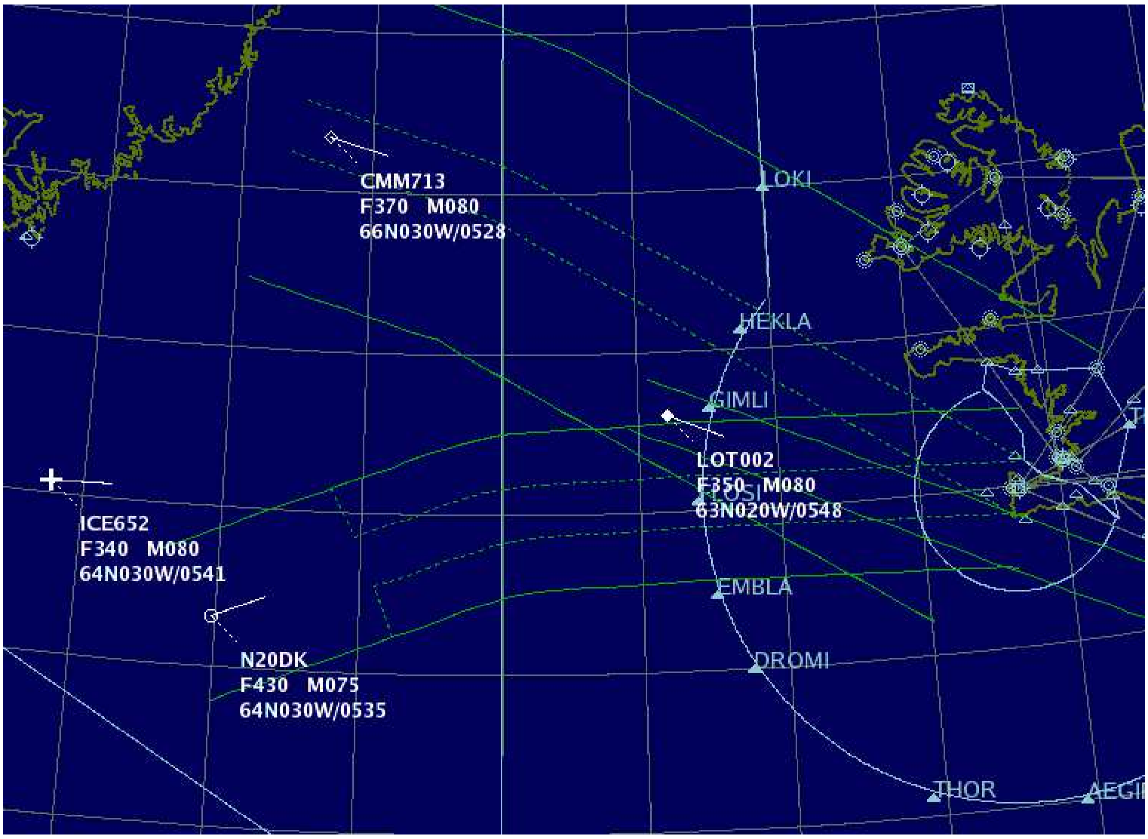


Figure 9.1: *Screenshot of all features specific to the second prototype.*

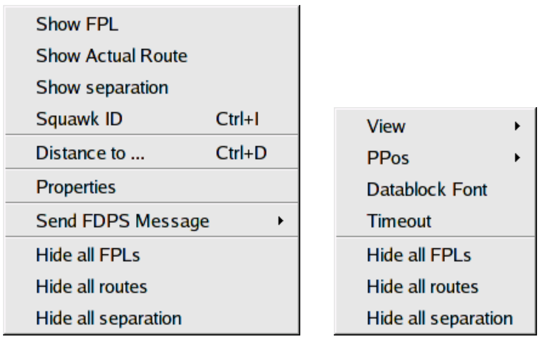


Figure 9.2: *The two popup menus in the second prototype for a) each aircraft and b) the map itself when no aircraft is selected.*

Chapter 10

User Testing, Round Two

Essentially the same user test as before was repeated the second time around. There were new participants who hadn't seen the first prototype and who could also provide interesting individual responses to the questions posed for the group in the first test.

Section 10.1 covers the details of the procedure, with emphasis on the differences from the first user testing. Section 10.2 continues is where the choice of questions is discussed, and compared with the questions from the first user testing. Finally sections 10.3 and 10.4 present the results and conclusions drawn from them. All the original Icelandic language source material can be found in appendix B.

10.1 Procedure

The procedure followed for testing the second prototye was practically the same as for the first prototype, which was discussed in section 7.1. The differences lay in the number of participants for each test run, the questions asked and the computer hardware used as well as the testing of paper mockups of previously untested ideas. Besides that, the participants were treated with cookies and soft drinks to lighten the mood. This time, instead of a group of three persons, five air traffic controllers participated as individuals. One of the participants was also one of the three from the first round of testing, so he only answered the new or changed questions. All the participants had substantial working experience as air traffic controllers, but only moderate to minor experience as user interface testers.

The participant is seated alone in front of a laptop computer with a 15" LCD monitor using an external mouse instead of the built in mouse pad on the laptop.

This introductory text is shown in figure 10.1 in it's original form and scale. It differs slightly from the introduction for the first prototype, since new features were developed for the second one.

The final two questions revolved around a paper mockup prototype. This was produced by manipulating screenshots of Amelia to contain user interface elements not available in the running program. These were the pictures in figures 10.2, 10.3 and figure 10.4. The functionality attempted to portray in these pictures is one of conflict alert between various separation types.

Having gone through all the questions and tasks the participants gave some recommendations for improving the test procedure and were soon on their merry way after approximately a one hour process each.

Rules of Separation on the Controller Map

The purpose of this project is to examine how best to show mixed rules of separation on the airtraffic controller map. This is done by modifying the ATC simulator Amelia from Tern Systems Inc.

About the simulator
The following operations are available in the simulator:

- Popup menu for the map or a plane
  - The map menu is accessed by clicking the right mouse button on the empty map
  - The plane menu is accessed by clicking the right mouse button on the symbol or the datablock for the plane.
- View the plane flight plan
  - The menu point “Show FPL” on the plane popup menu shows the flight plan for the plane with coordinates by each point on the flight plan.
- To move the map center click the middle mouse button (the mouse wheel)
- Zoom in and out with the mouse wheel.

The simulator differs from the work environment in the following ways, among other:

- It is possible to remove the flight plans of all planes from the map by selecting the menu point “Hide all FPLs” on either the map or the plane menu.
- The datablock shows some other information
  - Line 1 shows call ID
  - Line 2 shows flight level and mach number
  - Line 3 shows the next point on the flight plan
- Radar range is simulated, the average range of each radar station is used to determine if a plane is within radar range.
- To simulate confirmation of radar contact the menu point “Squawk ID” is used on the popup menu for the plane that is confirmed.

Changes related to this project:

1. The plane symbols change by the following simple rules.
   - If the plane is within RADAR and confirmed, the radarsymbol is shown: ◆
   - If the plane is within RADAR but not confirmed, the open radarsymbol is shown: ◇
   - Otherwise if the plane has ADS equipment the ADS symbol is shown: ○
   - Otherwise the usual “procedural” symbol is shown: +
2. The boundaries of minimum separation can be shown for each plane separately.
   - These boundaries mark the distance the plane requires to be successfully separated from any other plane.
   - Only lateral separation is shown.
   - If radar contact has not been confirmed for the plane, the separation boundaries continue to be either ADS or procedural. The RADAR boundaries are shown with a dotted line.
   - The airtraffic controller chooses to show or remove the separation boundaries by right clicking the plane and choosing “Show separation” from the menu that appears.
   - The airtraffic controller can remove all separation boundaries by right clicking the empty map and choosing “Show separation” from the menu that appears.
3. If minimum separation is expected to be violated between two planes the following takes place:
   - The points on the flight plan of one where it travels within the separation boundaries of the other are marked with red dots, along with coordinates and time.
   - The points on the separation boundaries of one where its separation is violated are marked with red dots.
   - The part of a planes flight plan where it is estimated to be within the separation boundaries of another plane is drawn with red.
   - Since this functionality is not implemented in the simulator the following mockup pictures illustrate how this will appear to the flight controller.

Figure 10.1: *The introductory text presented to all test participants.*

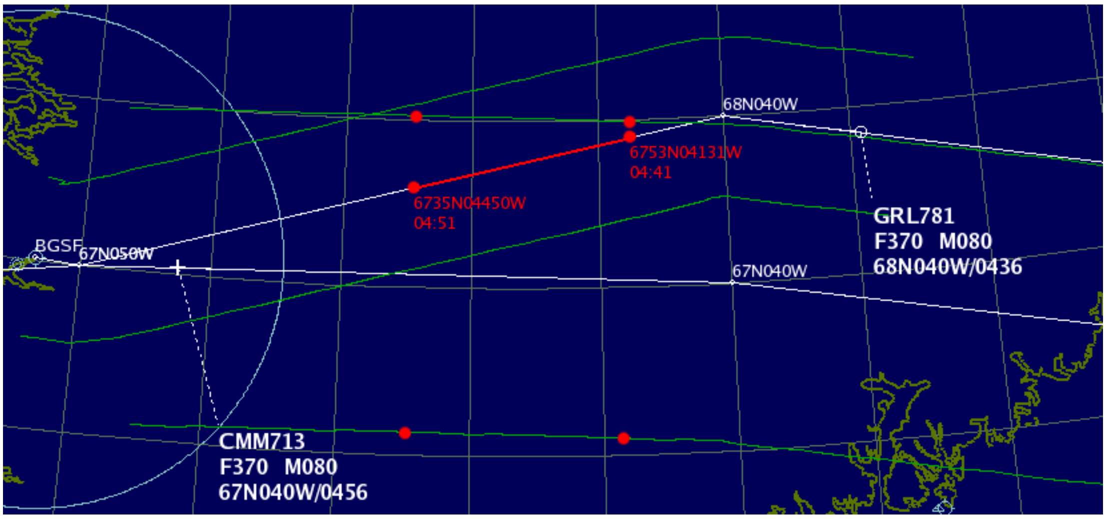


Figure 10.2: *Separation conflict between an ADS-B equipped aircraft and procedural.*

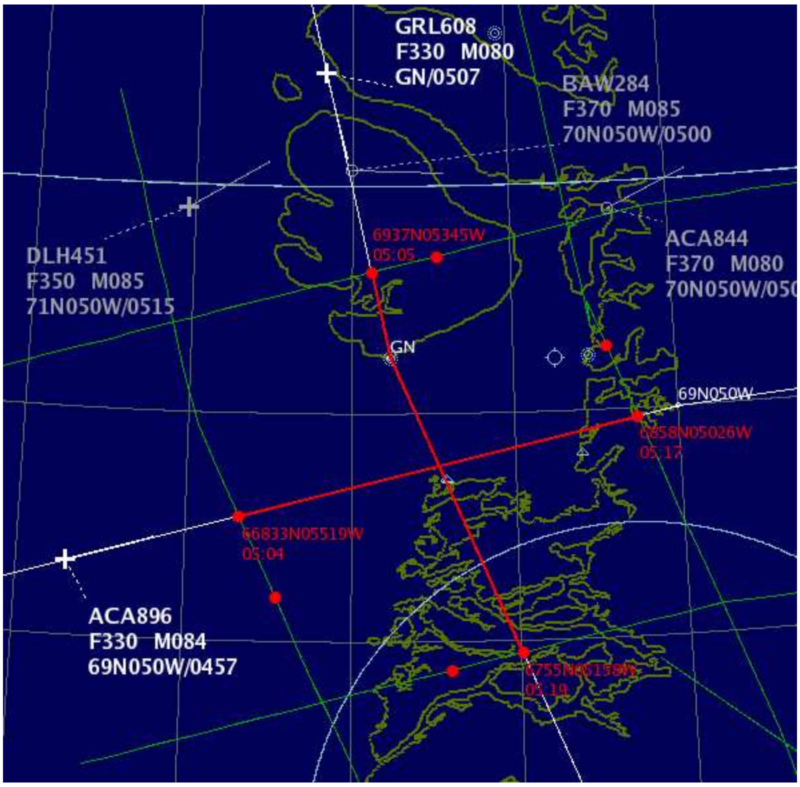


Figure 10.3: *Separation conflict between two procedural .*

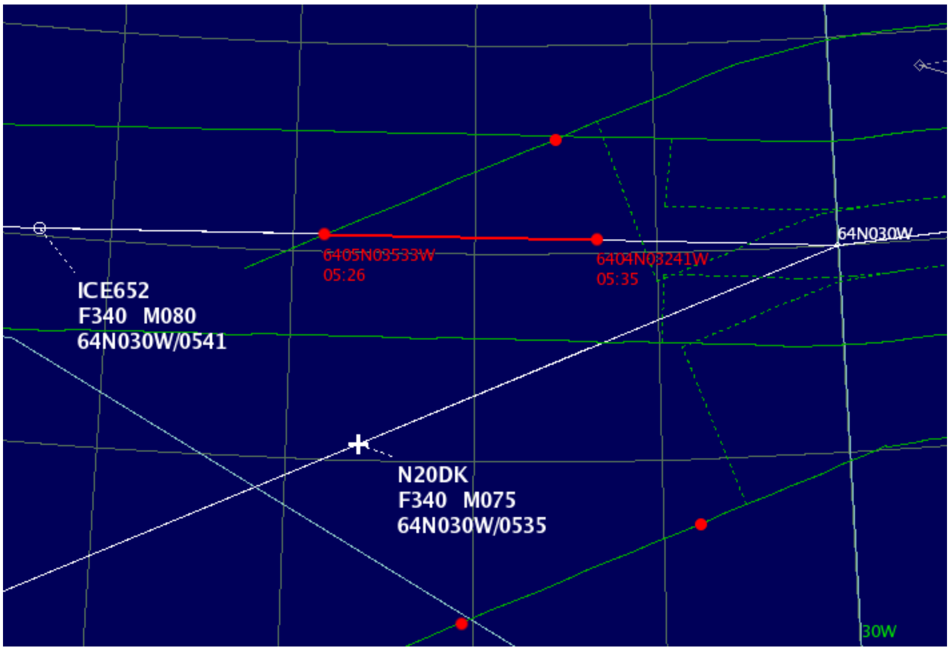


Figure 10.4: *Separation conflict at the edge of radar coverage.*

10.2 Questions and Tasks

The questions asked and tasks presented to the participants were the following. Since most of them are the same as for the first prototype, only the changed and the new ones are presented here along with a rationale for their purpose. See section 7.2 for a comments on the questions not covered here.

1. Can you find examples of planes with all three different rules of separation in effect?
2. Show the separation boundaries for planes with callsigns ICE652, CGIRE, ACA844 og N96LT.
3. Clear the separation boundaries from question 2.
4. The paths of the planes with callsigns CMM713 and GRL781 intersect. Is the separation sufficient?
5. The paths of the planes with callsigns BAW284 and ACA844 intersect. Is the separation sufficient?
6. The plane with callsign ACA896 now requests flight level F330. Find out if the separation remains sufficient.
7. Is it helpful to see the aircraft symbols change according to the rules of separation that apply?
8. Are the aircraft symbols sufficiently different to be able to differentiate between the rules of separation that apply?
9. Do you find the operations to show and clear separation boundaries satisfactory?
10. Do separation boundaries show something that you could not see before?
11. Before: **Should separation boundaries be shown less protruding, e.g. a softer color in the background?**
    After: **Should separation boundaries be shown more or less protruding, e.g. a brighter or softer color in the background, perhaps a different color?**
    This is just a minor rephrase of the same question since the color was indeed made softer as suggested in the first round of testing. See figure 10.5 for how the new color appears.
12. What do you think of this way to show separation boundaries on the map?
13. New: **What do you think about showing separation boundaries together for both confirmed separation and estimated radar separation?** This open ended question aims at the new feature in the second prototype, estimated radar separation. Figure 10.5 shows this feature in action.
14. Are these changes useful in assessing separation?
15. Are these changes sufficient in assessing separation and if not, what is missing?

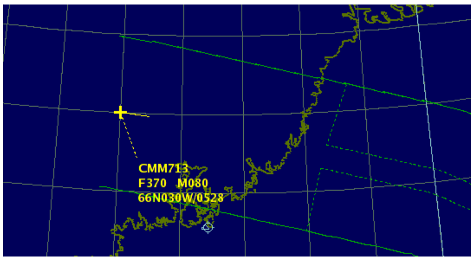


Figure 10.5: *Separation boundaries at the edge of radar coverage.*

16. Would it be helpful to see an aircraft's estimated time of arrival at each point in the flight plan?

17. New: **Would it be helpful to be able to see the area where conflict of separation exists between two planes?**
    This question is aimed at the paper mockups, asking about opinions regarding conflict alerts. For this question and the next, the paper mockups are all put on the table in front of the participant.

18. New: **Is it clear enough with which area it is dealing?**
    This question is a continuation of the previous. It is necessary to ask this, since the paper mockups don't interact and the participant is unable to experience any functionality.

19. What would you change?

10.3 Results

All in all the second round of user testing confirmed the results from the first round. Most if not all air traffic controllers were generally pleased with the modifications, but felt that the map quickly became too cluttered.

The most surprising result was perhaps that all of them felt that the open diamond and the circle were now too similar. This only emphasizes the need to carefully select and test the aircraft symbols.

The one participant who participated in both tests found the modifications since the first prototype to be clear improvements over the previous functionality.

The following are excerpts from the notes and observations made by the test coordinator while the participants answered the questions and solved tasks in section 10.2. Participants will remain anonymous, but when appropriate will be referred to as a, b, c, d and e who also participated in testing the first prototype. The original notes in Icelandic are included in appendix section B.1.2.

17. All participants agree that it would in deed be helpful. Owo suggest that it should be possible to deactivate it or at least confirm that the conflict is noticed by the controller.

One suggests calculating time differences in the flight plan crossing points to better be able to calculate longitudinal separation. Remarks suggest that both flight plans should be drawn in red for the portions where conflicts exist and perhaps highlight the whole area.

18. The general consensus is that it is not clear enough where the conflict lies. Some of the mockups only draw the flight plan for one aircraft in red, whereas most of the participants agree that both should be drawn in red. Remarks again from different participants that perhaps the whole area should be highlighted. Making sure actual rules of separation are used is imperative.

19. While all participants pointed back to their responses to previous questions, one of them remarked that more could be made from the fact that one always compares exactly two aircraft. Another improvement could be to be able to have quick access to isolating specific flight levels and have the rest of the traffic fade into the background for a short while.

10.4 Conclusion

It is clear that the modifications introduced in the second prototype made the modifications more appealing as all of the participants were mostly positive in their views.

The conflict alert mockup was also very positively received by all participants even though the exact functionality of the ideas received some criticism.

Chapter 11

Further work

Building upon this project may be done in many ways, some of which can be mentioned here.

Aircraft Symbols Choosing the correct symbols is a fine art. The symbols used in this study were chosen on the basis of simplicity and recognizability. Other criteria may be used, hopefully resulting in improved usability.

Separation Modeling Separation standards are extremely complex to represent as datastructures usable in software. In this study, the standards were simplified to such extreme that it is unusable in practice. If separation is to become integrated into situation displays, these need to be modeled more closely.

Separation Distinction Using aircraft symbols to distinguish between separation standards is perhaps the wrong way. Examining alternative ways or a combination, would hopefully result in improvements. This may perhaps be dependant upon a detailed model of separation standards.

Conflict Detection Using tubes and phases may result in efficient conflict detection. Unfortunately, this was not a part of this study, but it is an important research topic. Perhaps tubes and phases are the wrong approach, but in order to be exact, conflict detection would depend on a detailed model of separation standards.

Conflict Alert Alerting to conflicts in a clear, yet uninvasive way is an important topic for study. It became clear during user testing that this was something that worried the participants.

Conflict Resolution It is not enaugh to detect and alert to conflicts. The conflict needs to be resolved, and even though controllers are trained to do that by hand, perhaps it could be done more efficiently by enlisting the aid of computers.

Efficiency Study Discovering the efficiency level of the various approaches to directing air traffic could be beneficial. Many of the factors relating to directing mixed separation air traffic are unknown and estimated.

Safety Study It is not at all certain that air traffic controllers are able to cope with directing mixed separation air traffic. Too many rules could be too confusing and the results could be disastrous. Definitely a recommended topic for research.

Chapter 12

Conclusion

The lack of distinction and clarification as introduced in the very beginning in chapter 1 are the core of what was explored here. Historically the only significantly different separation standards have been procedural and radar separation. This has led to such solutions as separating air traffic controller work tasks by sectors with and without radar coverage. This may necessarily be changed in the coming years as use of ADS-B technology increases and ADS-B coverage spreads to areas without radar coverage. Mixed separation will likely become the norm rather than the exception and research into how best to work with that will become even more important than it is now. Dividing the available air traffic controller workforce into control by radar, ADS-B and procedural is not likely to be efficient enough. Common sense dictates that if controllers were able to manage any of these at the same time with a mixed separation situation display it should provide better efficiency.

This study shows that, if provided with the necessary tools, air traffic controllers are able to work with mixed separation situation display. Even though the prototype used for testing the improvements to the situation display was different from the work environment at Reykjavík ACC, the air traffic controllers who participated had hardly any difficulties adjusting. It was the assessment of the test coordinator that the participants were sufficiently comfortable operating the prototype after the initial step of getting familiar with the difference in environments. The efficiency of their work and adherence to safety standards were not part of this study and need to be carefully examined. Although some recommendations are made here, they should obviously not supersede any safety requirements.

Being able to test with both a group as well as individuals gave some interesting results. Issues with the questions and tasks presented to the group participating in round one were not made apparent until some of the individual participants in round two had problems with them. The group effort was able to resolve the issue and work around it, while individuals needed to consult with the test coordinator. This was unfortunate, but not a critical problem since such consultations were not discouraged.

The findings of this study indicate that a situation display working with mixed separation shold provide the following facilities in some form:

Instant Criteria Recognition Being able to recognize which separation criteria applies at the current time is the single most important facility examined in this study. Providing the air traffic controller with some sort of visual aid to trigger this recognition provides the information necessary to quickly assess the current situation. The simple geometric shapes tested in this study gave good results, but it was found that these need to be selected carefully and with appropriate size considerations.

Separation Examination It is not enough to simply be able to assess the situation. Most of the time some closer examination is necessary in order to gain precise insight into the situation at hand. This is always a process involving exactly two aircraft. The separation boundaries tested in this study were found to be inadequate for this purpose. Even though the air traffic controllers found the boundaries useful, it was also found to unnecessary unless when examining conflicts between two aircraft.

Conflict Alert Using the abilities of computer systems to calculate aircraft trajectories and finding conflicts seems like an obviously beneficial use of resources. Showing the air traffic controller where alerts occur is however, a matter of subtle art. The controller must not be distracted by unnecessary alerts resulting from the computer system not being able to "know" all the details surrounding a boundary case. The concepts tested in this study were only done with paper mockup prototypes, but were well received nonetheless.

On the basis of the work described in the above text, these findings are recommended for use in air traffic control environments where mixed separation standards are applied.

Bibliography

[1] International Civil Aviation Organization (ICAO). *The website of the International Civil Aviation Organization*, Last visited 28. February 2005. http://www.icao.org/.

[2] Federal Aviation Authority. *FAA ATC Manual*, Last visited 28. February 2005. http://www.faa.gov/atpubs/ATC/ATC.pdf.

[3] Icelandic Civil Aviation Authority (ICAA). *The website of the Icelandic Civil Aviation Authority*, Last visited 28. February 2005. http://www.caa.is/.

[4] Icelandic Civil Aviation Authority (ICAA). *ICAA Air Traffic Control Information Page*, Last visited 28. February 2005. http://www.caa.is/Forsida/English/Air_Traffic_Control_Division/view.aspx.

[5] Tern Systems Inc. *The website of the Tern Systems Inc.*, Last visited 28. February 2005. http://www.tern.is/.

[6] European Organisation for the Safety of Air Navigation (EUROCONTROL). *EATM glossary of terms*, 9. June 2004. http://www.eurocontrol.int/eatm/gallery/content/public/library/terms.pdf.

[7] Federal Aviation Authority. *FAA Pilot-Controller Glossary*, Last visited 28. February 2005. http://www.faa.gov/atpubs/PCG/PCG.pdf.

[8] Arjun Singh. *Communication Navigation Surveillance / Air Traffic Management (Cns / Atm) Beyond 2012*, 2004. http://www.gisdevelopment.net/technology/gps/pdf/ma04082.pdf, http://www.gisdevelopment.net/technology/gps/ma04082.htm.

[9] Simon Bennett, Steve McRobb, and Ray Farmer. *Object-Oriented Systems Analysis and Design using UML*. McGraw-Hill, 1999. Teaching material from authors previous course work.

[10] Trolltech AS. *The QT C++ Application Development Framework*, Last visited 28. February 2005. http://www.trolltech.com/products/qt/.

[11] Beman Dawes and David Abrahams. *The Website for the Boost C++ Libraries*, Last visited 28. February 2005. http://www.boost.org/.

[12] Inc. Silicon Graphics. *The Standard Template Library*, Last visited 28. February 2005. http://www.sgi.com/tech/stl/.

[13] European Organisation for the Safety of Air Navigation (EUROCONTROL). *Programme for Harmonised Air Traffic Management Research in EUROCONTROL* (phare). http://www.eurocontrol.int/phare/.

[14] European Organisation for the Safety of Air Navigation (EUROCONTROL). *PHARE Ground Human Machine Interface (GHMI) Project: Summary Report*, Aug. 1999. http://www.eurocontrol.int/phare/pdf_files/99-70-02ghmi.pdf.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human-Computer Interaction*. Addison-Wesley, 1994. Suggested reading related to authors previous course work.

[17] European Organisation for the Safety of Air Navigation (EUROCONTROL). *EUROCONTROL Website*. http://www.eurocontrol.int/.

[18] European Organisation for the Safety of Air Navigation (EUROCONTROL). *PHARE Final Report*, Nov. 1999. http://www.eurocontrol.int/phare/pdf_files/99-70-09pharefinal10.pdf.

[19] European Organisation for the Safety of Air Navigation (EUROCONTROL). *Common Modular Simulator Final Report*, May 1999. http://www.eurocontrol.int/phare/pdf_files/99-70-04cms.pdf.

[20] European Organisation for the Safety of Air Navigation (EUROCONTROL). *PHARE Advanced Tools Project Final Report*, Nov. 1999. http://www.eurocontrol.int/phare/pdf_files/98-70-18-v1-fr_v1_1.pdf.

[21] Dofri Jónsson, Bergur Ragnarsson, Júlíus Bergmann Gústafsson, and Ragnar Mar Steinsen. *Amelia User Manual*, August 2004. Unpublished, available at Tern Systems.

[22] Wikipedia Contributors. *Wikipedia, the free encyclopedia*, Last visited 28. February 2005. http://en.wikipedia.org.

Appendix A

Expanded Definition of Terms

The following is a nearly complete list of terms related to this project that need clarification. Most of these are extracted from [6], others are found in various sources in the litterature and represent the common use of these terms.

Aircraft (A/C) 1- Any machine that can derive support in the atmosphere from the reaction of the air other than the reactions of the air against the surface of the earth.
2- The term aircraft refers to fixed wing and rotary wing (helicopters) powered aircraft and balloons.

Aircraft Call Sign A group of alphanumeric characters used to identify an aircraft in air-ground communications. For convenience, the term is frequently abbreviated to "call-sign" when used in appropriate contexts.

Aircraft Conflict Two aircraft are involved in an aircraft conflict if the distance between any probable position of one aircraft at a certain time, and any probable position of the other aircraft at that same time, will be less than the required separation criteria.

Aircraft Identification A group of letters, figures, or a combination thereof which is either identical to, or coded equivalent of, the aircraft call sign to be used in air-ground communications, and which is used to identify the aircraft in ground-ground air traffic services communications.

Air Traffic Control (ATC) A service operated by the appropriate authority to promote the safe, orderly and expeditious flow of air traffic.

Air Traffic Control Service (ATCS) Is a service provided for the purpose of : preventing collisions between aircraft and on the manoeuvring area between aircraft and obstructions and expedition and maintaining an orderly flow of air traffic. (Ref. ICAO Doc 9569 Definitions).

Air Traffic Control Clearance Authorisation for an aircraft to proceed under conditions specified by an air traffic control unit. For convenience, the term is frequently abbreviated to "clearance" when used in appropriate contexts.

Air Traffic Control Sector A defined airspace region for which an associated controller (or controllers) has ATC responsibility.

Area Control Center (ACC) That part of ATC that is concerned with en-route traffic coming from or going to adjacent centers or APP. It is a unit established to provide air traffic control service to controlled flights in control areas under its jurisdiction.

Air Traffic Controller Usually meant to indicate the radar (executive) controller. The person having final responsibility in co-ordinating traffic in a sector. The radar controller (often simply called the controller) is assisted by a planner (advance planning) and assistant.

Air Traffic Management (ATM) As defined by the Special Committee on Future Air Navigation Systems (FANS), a system consisting of a ground part and an air part, both of which are needed to ensure the safe and efficient movement of aircraft during all phases of operation. The airborne part of ATM, according to FANS, consists of the functional capability which interacts with the ground part to attain the general objectives of ATM. The ground part of ATM comprises the functions of Air Traffic Services (ATS), Airspace Management (ASM) end Air Traffic Flow Management (ATFM). Air traffic services are the primary components of ATM.

Air Traffic Service (ATS) A generic term meaning: Flight Information Service (FIS), Alerting Service (AL), Air Traffic Advisory Service (ATAS), Air Traffic Control Service (ATCS), Area Control Service (ACS), Approach Control Service or Aerodrome Control Service.

Air Traffic Service Route A specified route designed for channelling the flow of traffic as necessary for the provision of air traffic services.

Altitude The vertical distance of an object measured from mean sea level.

Approach Control (APP) 1- Unit charged with control of traffic around one or more airports and responsible for the spacing of traffic on final approach. Approach control, that part of ATC that is concerned with departing and arriving aircraft. Arriving aircraft are transferred from ACC via APP to tower control, departing aircraft are transferred from tower control via APP to ACC.
2- The ATC service provided to arriving IFR flights

Automatic Dependent Surveillance (ADS) A function in which aircraft automatically transmit, via a data link, data derived from onboard navigation systems. This data is displayed to the controller on a screen that resembles a radar screen.

Automatic Dependent Surveillance-Broadcast (ADS-B) A function on an aircraft or surface vehicle that broadcasts position, altitude, vector and other information for use by other aircraft, vehicles and by ground facilities.

Automatic Dependent Surveillance-Contract (ADS-C) It functions similar to ADS-B but the data is transmitted based on a contract between a ground system and an aircraft: Demand contract, periodic contract, event contract and emergency contract.

Calibrated Air Speed (CAS) The speed theoretically shown by a simple airspeed indicator that is free from instrument error and position error. CAS represents TAS at all subsonic speeds at sea level pressure. At higher altitudes CAS must be corrected for compressibility error to give EAS.

Controller Pilot Data Link Communications (CPDLC) Sometimes referred to as Two-Way Pilot Data Link, is the means of communication between controller and Aircrew, using data link in conjunction with or instead of voice, for ATC.

Current Flight Plan The flight plan, including changes, if any, brought about by subsequent clearances.

Current Flight Position A ground-referencerable representation, in x, y, z of the current position of a flight based on latest sensor data.

Current System Trajectory Position A projection of the current flight position on to the flight's system trajectory that serves to determine the longitudinal conformance.

Earth-Centred-Earth-Fixed A right-hand Cartesian co-ordinate system with its origin located at the centre of the Earth - the coordinate system used by GPS to describe three-dimensional location. ECEF co-ordinates are centred on the WGS-84 reference ellipsoid, have the "Z" axis aligned with the Earth's spin axis, the "X" axis through the intersection of the Prime Meridian and the Equator and the "Y" axis is rotated 90 degrees East of the "X" axis about the "Z" axis.

Equivalent Air Speed (EAS) The speed which represents the same dynamic pressure at altitude that would be generated by flying at the corresponding TAS at sea level. At sea level EAS is the same as CAS, but at higher altitudes it must be corrected for density to give TAS.

Flight An aircraft, or group of aircraft that are to be considered as a single entity, known to the ATC system.

Flight Level (FL) A surface of constant atmosphere pressure which is related to a specific pressure datum, 1013.2hPa, and is separated from other such surfaces by specific pressure intervals.
Altitude above sea-level in 100 feet units measured according to a standard atmosphere. Strictly speaking a flight level is an indication of pressure, not of altitude. Since all aircraft use the same method to measure air pressure, this ensures that the aircraft will use the same flight levels in each others close proximity.
Each is stated in three digits that represent hundreds of feet. For example, flight level 250, indicated as F250, represents a barometric altimeter indication of 25,000 feet.

Flight Path The set of positions an aircraft has occupied, is occupying or is expected to occupy.

Flight Plan (FPL) Contains specified information provided to Air Traffic Services Units, relative to an intended flight or portion of a flight of an aircraft.

Galileo European contribution to the next generation GNSS intended to provide the European Nations with greater independence by delivering a civil controlled satellite based navigation system. Implementation of Galileo is expected to stimulate growth in the use of GNSS technology in intermodal transport systems thereby improving mobility safety and quality of life whilst also stimulating economic growth in the areas of receiver manufactures and application development. When considering the possibility of a failure of GPS to maintain a service, the implications for commercial customer in the fields such as telecom, location based services and financial community are clearly not acceptable. For these reasons and in consideration of the potential impacts on European business, the European Commission (EC) and the European Space Agency (ESA) has embarked on the definition phase of Galileo

Global Navigation Satellite System (GNSS) Generic term used by ICAO to refer to a worldwide position, velocity, and time determination system, that include one or more satellite constellations, receivers, and system integrity monitoring, augmented as necessary to support the required navigation performance for the actual phase of operation. Examples of GNSS are GPS, GLONASS and Galileo.

Global Orbiting Navigation Satellite System (GLONASS) The Global Orbiting Navigation System comprises a constellation of 21 satellites plus three spares. These satellites operate in a 11 hour and 15 minute orbit at an altitude of 19, 100 km and each satellite transmits on a unique frequency using the same pseudo noise code. The level of accuracy for GLONASS is similar to GPS operating in the C/A mode. The measuring of the GLONASS broadcast information to determine position and range, is similar to that for the GPS. However, the navigation information of GLONASS, is referenced to the SGC-85 geodetic reference system with time being referenced to UTC.

Global Positioning System (GPS) A space-based positioning, velocity and time system, developed and operated by the U.S. Department of Defense and composed of space, control and user segments. The space segment is composed of 21 satellites (plus three operational spares) in six orbital planes. The control segment consists of five monitor stations, three ground antennas and a master control station. The user segment consists of antennas and receiver-processors that provide positioning, velocity, and precise timing to the user.

Heading The direction in which the longitudinal axis of an aircraft is pointed, expressed in clockwise degrees from North. Usually magnetic heading (with respect to the magnetic north pole), sometimes true heading (with respect to the geographic north pole).

Height The vertical distance of a level, a point, or an object considered as a point, measurement from a specified datum.

Horizontal Separation Separation between aircraft expressed in terms of time, distance or angular displacement between tracks.

Indicated Air Speed (IAS) The speed of an aircraft as shown to the pilot by the airspeed indicator on board. IAS will differ from TAS at air densities other than some reference density. This is the speed used in pilot/controller communications under the general term "airspeed."

Lateral Separation Separation between aircraft expressed in terms of distance or angular displacement between tracks.

Level A generic term relating to the vertical position of an aircraft in flight and meaning variously, height, altitude or flight level.

Longitudinal Separation Separation between aircraft expressed in terms of time or distance along tracks.

Mach Number The ratio of speed to the speed of sound in the medium. For aircraft, the medium is air for which the speed of sound differs with temperature and to a lesser extent humidity. Mach number is widely used in air traffic control and is for example indicated as M080 for an aircraft travelling with a true airspeed at 80% of the speed of sound.

Mach Number Technique The term used to describe the technique of clearing turbojet aircraft to maintain specified Mach numbers in order to maintain adequate longitudinal separation between successive aircraft at, or climbing or descending to, the same level.

Nautical Mile (nm, NM) A measure used in navigation. The unit is equal to 1852 m.

Navigation Control System to determine actions to be taken to follow a planned route. Currently 2D, 3D and 4D navigation, are in use. Using 2D-navigation, a 2D-route (set of geographical co-ordinates) can be flown. 3D-Navigation makes it possible to assure a certain altitude for each geographical position. Some modem aircraft already use 4D-navigation, where it is also possible to guarantee reaching a specific point and altitude at a specified time.

Non-Radar Separation The separation used when aircraft position information is derived from sources other than radar.

Position (POS) Location of a point or an object considered as a point in 3-D space.
In air traffic control, the position of an aircraft is usually given as the geographical coordinates on the surface of the earth directly below it.

Procedural Separation The procedure of systematically examine flight plans for conflicts in order to keep aircraft separated.

Radar Contact The situation which exists when the radar blip or radar position symbol of a particular aircraft is seen and identified on a radar display.

Radar Identification The process of correlating a particular radar blip or radar position symbol with a specific aircraft.

Radar Monitoring The use of radar for the purpose of providing aircraft with information and advice relative to significant deviations from nominal flight path, including deviations from the terms of their air traffic control clearances.

Radar Position Indication (RPI) The visual indication, in non-symbolic and/or symbolic form, on a radar display of the position of an aircraft obtained by primary and/or Secondary Surveillance Radar.

Radar Position Symbol (RPS) A generic term for the visual indication in a symbolic form, on a radar display, of an aircraft obtained after digital computer processing of positional data derived from primary radar and/or SSR.

Radar Separation The separation used when aircraft position information is derived from radar sources.

Reporting Point A specified geographical location in relation to which the position of an aircraft can be reported.

Route The 2D trajectory of an aircraft, expressed in terms of significant points, airways or geographical points.

Separation Situation in which two aircraft do not violate the separation criteria.

Separation Criteria The required minimum vertical, lateral, and longitudinal distances between two aircraft in order not to be involved in an aircraft conflict, or a nominal routes overlap.

Separation Minima The minimum required distance between aircraft. Vertically usually 1000 ft below flight level 290, 2000 ft above flight level 290. Horizontally, depending on the radar, 3 NM or more. In the absence of radars horizontal separation is achieved through time-separation (e.g. 15 minutes between passing a certain navigation point).

Significant Point A specified geographical location used in defining an ATS route or the flight path of an aircraft and for other navigation and ATS purposes.

Standard Geodetic Reference System The co-ordinate system built into the GPS constellation for its navigational output, as the universal co-ordinate system for International Civil Aviation applications. ICAO Annex 15 specifies that with effect from 01 January 1998, the co-ordinates of all points mentioned in AIPS and aeronautical charts be expressed in the WGS-84 System.

Track 1- A target report resulting from the correlation, by a special algorithm (tracking) of a succession of radar reported positions for one aircraft. The report normally contains smoothed position and speed vector information.
2- Time sequence of state vectors of an object estimated by some real time filtering technique using radar plot data as input.

Trajectory 1- A set of reconstructed 4-dimensional points (geographic position + altitude + time) defining the path of an aircraft through space. The points are calculated by using mathematical splice functions.
2- Time sequence of state vectors of an Object simulated or reconstituted from Multi-Radar data by some filtering technique, using all information available including the data after the moment of particular interest.
3- A generic sequence of 4D points that describe the profile of a flight.

Transponder A combined receiver and transmitter whose function is to transmit signals automatically when triggered by an interrogator.

True Air Speed (TAS) The actual speed of an aircraft relative to the air through which it is flying corrected for temperature and air density. Used primarily in flight planning and en route portion of flight. When used in pilot/controller communications, it is referred to as "true airspeed" and not shortened to "airspeed."

Universal Time Co-ordinated (UTC) An atomic time scale is the basis for civil time. It is occasionally adjusted by one-second increments to ensure that the difference between the uniform time scale defined by atomic clocks, does not differ from the Earth's rotation by more than 0.9 seconds.

Vertical Separation Separation between aircraft expressed in terms of flight levels or altitude between tracks.

Way-point (WP) A specified geographical location used to define an area navigation route or the flight path of an aircraft employing area navigation. Sometimes also called route point or RP.

World Geodetic System 1984 WGS 84 is an ellipsoid designed to fit the shape of the entire Earth as well as possible with a single ellipsoid. It is often used as a reference on a worldwide basis, while other ellipsoids are used locally to provide a better fit to the Earth in a local region. GPS uses the centre of the WGS- 84 ellipsoid as the centre of the GPS ECEF reference frame.

Appendix B

Source Material Related to User Testing

This chapter contains the original Icelandic versions of the material used for user testing as well as the resulting notes.

B.1 Questions and Tasks

1. Geturðu fundið dæmi um vélar með allar þrjár mismunandi aðskilnaðarreglur gildandi?

Figure B.1: *The format for each question, actual size is A5 page, original Icelandic version.*

B.1.1 Round One

Questions and results from round one of user testing.

1. Geturðu fundið dæmi um vélar með allar þrjár mismunandi aðskilnaðarreglur gildandi?
   Eftir smá ráðrúm finnast 3 dæmi
   Spurningin virðist hafa verið lélega orðuð
   Þeir virtust ekki hafa lesið innganginn, þekktu ekki táknin
   Þeir minntust á að það mætti vera meiri munur á krossi og tígli.

2. Sýndu aðskilnaðarmörk fyrir vélar með kallmerki ICE652, CGIRE, ACA844 og N96LT.
   Þeir voru ekki alveg vissir um hvernig ætti að sýna aðskilnaðarmörk, hafa greinilega ekki lesið innganginn.
   Þegar ADS útbúin vél er innan radar sést ekki að hún er ADS búin, því þá er bara radar táknið sýnt.
   Þeir voru dálítið í vafa af hverju aðskilnaðarmörkin væru mismunandi breið, áður en þeim var bent á að um væri að ræða ADS búna vél.

3. Hreinsaðu aðskilnaðarmörkin frá spurningu 2.
   Stjórnandinn velur hverja vél fyrir sig og fjarlægir aðskilnaðarmörk fyrir hverja og eina.
   Þegar því er lokið er þeim bent á að hægt er að fjarlægja mörkin fyrir allar í einu og þykir þeim það gott mál, en í raun nokkuð sem við ætti að búast og því ekki neitt sérstakt.
   Orð er haft á því að valmyndin er ofhlaðin, en það er vegna þess að um er að ræða herminn en ekki keyrsluumhverfið sem þeir eru vanir.

4. Ferlar vélanna með kallmerki CMM713 og GRL781 skerast. Er aðskilnaður nægur?
   Þeir sjá strax að aðskilnaður er ekki nægur út frá FPL, en skoða samt með aðskilnaðarmörkum. (Rétt)

5. Ferlar vélanna með kallmerki BAW284 og ACA844 skerast. Er aðskilnaður nægur?
   ACA844 er ekki komin að næsta FPL punkti og því er ekki hægt að sjá komutíma á þarnæsta punkt sem er sameiginlegur punktur.
   Þeir velta fyrir sér komutíma og sakna möguleika frá keyrsluumhverfinu þar sem hægt er að sjá áætlaðan komutíma.
   Stjórnandinn sýnir aðskilnaðarmörk fyrir báðar vélar, en þeir komast ekki að niðurstöðu fyrr en eftir að ACA844 er komin framhjá næsta punkti.
   Þeir komast að þeirri niðurstöðu að vélarnar eru ekki nægilega aðskildar eftir reglum sem gilda í dag, en væru það samkv. ADS reglum. (Rétt)
   Þeir hafa orð á því að kortið verður mjög hlaðið þegar 2-3 vélar eru sýndar með bæði FPL og aðskilnaðarmörkum.
   Stjórnandinn velur "Hide all" fyrir bæði FPL og aðskilnað.

6. Vélin með kallmerki ACA896 biður nú um flughæð F330. Athugaðu hvort aðskilnaður verði áfram nægur.
   Þeir finna conflict við VIR20, sem var óvænt, en rétt.
   Nú vantar aftur tímamælingu, því vélarnar krossa á 3. og 4. punkti á FPL.
   Eftir að haft er orð á að kannski séu fleiri vélar, finna þeir conflict við GRL608 eins og

ætlunin var.
Þeir nota bara FPL og fjarlægðarmælingar til að komast að niðurstöðu.
Niðurstaðan er sú að ekki sé nægur aðskilnaður. (Rétt)

7. Er hjálplegt að sjá tákn flugvélanna breytast eftir þeim aðskilnaðarreglum sem gilda?
   Allir eru þeir sammála um að það þarf að vera skýr munur á táknunum.
   Jafnframt haf þeir orð á eftirfarandi:
   Það er ekki nóg eins og það er núna, krossinn er of líkur tíglinum.
   Það gæti verið hættulegt að treysta á táknin.
   Mætti sýna sérstaklega ef radarsamband tapast.
   Þegar vélin er innan radar ætti táknið að breytast til að vita að það megi nota radar aðskilnað.
   Það þarf að tryggja að radar samband sé rétt.
   Niðurstaðan er sú að það er mjög hjálplegt, jafnvel mætti ganga svo langt að kalla það nauðsynlegt.

8. Eru tákn flugvélanna nógu ólík til að skilja á milli þeirra aðskilnaðarreglna sem gilda?
   Allir eru þeir sammála um að hringurinn skilur sig ágætlega frá hinum, en krossinn er of líkur tíglinum.
   Einn lætur í ljósi þá skoðun að ef krossinn væri stærri yrði munurinn greinanlegur, hinir eru sammála.
   Tillaga frá einum að öðrum lit fellur ekki eins vel inn.
   Þeir benda á að hringur er nú þegar í notkun fyrir aðra virkni.
   Jafnframt benda þeir á að fyllt tákn og lokuð, bæði hringur og tígull, eru notuð til að skilja á milli tracked og non-tracked. (eða álíka)
   Niðurstaðan er sú að samræma þarf notkunina á táknum með keyrsluumhverfinu, en fyrir þessa notkun er hringurinn OK, en breyta þarf krossinum svo hann skilji sig meira frá tíglinum.

9. Finnst þér aðgerðirnar til að sýna og hreinsa aðskilnaðarmörk fullnægjandi?
   "Já, í besta lagi".

10. Sýna aðskilnaðarmörk eitthvað sem þú gast ekki séð áður?
    Þeir segja allir að þessi mörk voru áður bara til í huga hvers flugumferðarstjóra, en "nú er hægt að prenta þessa mynd".
    Þeir eru allir sammála um að skjárinn er fljótur að fyllast af línum.
    Niðurstaðan er sú að það er hjálp í því að sjá mörkin á skjánum, en bara 1-2 vélar í einu.

11. Ættu aðskilnaðarmörk að vera sýnd minna áberandi, t.d. í daufari lit í bakgrunninum?
    Hér voru skiptar skoðanir:
    "Æpandi litur hvetur menn til að hreinsa kortið".
    Mætti vera daufari, því of skær litur dregur athyglina frá öðru.
    Umferðin druknar í línum.
    Haft er á orði að eftir c.a. 10 vélar hættir maður að skilja hvað er að gerast, en daufari línur myndu samt varla hjálpa í þeirri stöðu.
    Einn þeirra bendir á að ef flugumferðarstjóri vill fylgjast með einni vél í lengri tíma væri eflaust betra að nota lit sem fellur betur inn í bakgrunninn.
    Annar bendir á að í keyrsluumhverfinu getur hver valið sína uppáhalds litasamsetningu.

Þá verður valið á lit mjög afstætt og yrði í rauninni að vera undir hverjum flugumferðarstjóra komið.
Niðurstaðan verður sú að liturinn grænn er í lagi, en daufari tónn væri betri.

12. Hvað finnst þér um þessa aðferð til að sýna aðskilnaðarmörk á kortinu?
    Þeir voru allir sammála um að þetta er fljótleg leið til að sýna hvernig aðskilnaður breytist. "Allt í lagi að geta séð þetta".
    Einn benti á að notkun á þessari aðferð er líka spurning um þjálfun. Það verður að læra vel á það hvað mörkin tákna.
    Það kom í ljós að það var ekki alveg ljóst fyrir þeim hvernig mörkin voru hugsuð. Þeir stöldruðu aðeins við og urðu sammála um (réttilega) að mörkin táknuðu í raun frátekið loftrými hverrar vélar, ekki samband vélarinnar við hinar umhverfis.
    Einn þeirra benti á að það mætti gjarnan sýna punktana þar sem aðskilnaður er brotinn. T.d. við að sýna punktana með hnitum og teikna í rauðum lit þann legg flugsins þar sem aðskilnaður er brotinn.
    Einn þeirra fékk þá hugmynd að þægilegt væri að geta sýnt sérstaklega sambandið milli tveggja útvalinna véla. t.d. við að draga á milli vélanna og þá hengist seinni vélin á þá fyrri.
    Annar benti þá á að kerfið ætti jafnvel að gera þetta sjálfkrafa.

13. Eru þessar breytingar gagnlegar við að meta aðskilnað?
    Þeir bentu á að þetta væri helst gagnlegt við gegnumflug.

14. Eru þessar breytingar nægjanlegar til að meta aðskilnað og ef ekki, hvað vantar?
    Sjá 12.
    Annars stóð helst í þeim vandamál í sambandi við radar auðkenni.
    Þegar vél er á leið inn í radar þyrfti að sýna áætlaðan aðskilnað vegna radar í öðrum lit en núverandi aðskilnað og jafnvel teikna hvort tveggja.
    Með öðrum orðum að teikna áætlaðan aðskilnað með öðrum lit, þegar aðskilnaðar fjarlægðin kemur til með að minnka.
    Bent var á að taka þyrfti tillit til nákvæmra aðskilnaðarreglna, en þetta var ásættanlegt fyrir prufu.
    Þeir voru allir sammála um að eins og lýst var í spurningu 12, þá væri mjög áhugavert að get merkt tvær vélar og skoðað sambandið þeirra á milli.

15. Væri hjálplegt að sjá væntanlegan aðkomutíma flugvélar við hvern punkt í flugplaninu?
    Já, það er nú þegar möguleiki í keyrsluumhverfinu.

16. Hverju myndir þú breyta?
    Einn þeirra bendir hér á að aðskilnaðarmörk segja eiginlega meira um áætlað radardrægi en aðskilnað þegar komið vestur fyrir 30°W því þar er eingöngu procedural notað.
    Aths. við datablock:
    Setja mætti hraða upp í sömu línu og hæð.
    Tími að næsta punkti tekur of mikið pláss.
    Jafnvel mætti sýna eingöngu kallmerki, en skipta yfir í nánari upplýsingar þegar smellt er á vélina.
    Almennt ætti það sem tekur pláss að vera valmöguleiki fyrir notandann til að skyggja ekki á það sem skiptir máli.
    Fyrir utan svör við 12 og 14 er ekkert sem þeim datt í hug.

B.1.2 Round Two

Questions and results from round two of user testing.

Participants will remain anonymous, but when appropriate will be referred to as a, b, c, d and e who also participated in testing the first prototype.

1. Geturðu fundið dæmi um vélar með allar þrjár mismunandi aðskilnaðarreglur gildandi?
   a) Fannst spurningin óljós, en fann að lokum rétt dæmi.
   b) Vill reyna að meta hvort allar vélar séu aðskildar. Tekur ekki til sín leiðbeiningar og endar með niðurstöðu: Nei
   c) Já, finnur 3 dæmi vandræðalaust.
   d) Fann auðveldlega 3 dæmi. Hafði orð á því að við fyrstu sjón lítur út fyrir að það séu allt of mörg tákn, en svo við nánari athugun reynist það ekki neitt vandamál.

2. Sýndu aðskilnaðarmörk fyrir vélar með kallmerki ICE652, CGIRE, ACA844 og N96LT.
   a) Ekkert mál.
   b) Já, en bendir á að það er gagnslaust að tala um aðskilnað ADS vélar frá procedural, því ADS sé ekki í notkun.
   c) Ekkert mál.
   d) Fann merkin. Erfitt að sjá hvað er hvað. Ætti eiginlega að skoða bara 2 í einu (þetta er sagt þegar allar vélarnar hafa mörkin sýnd).

3. Hreinsaðu aðskilnaðarmörkin frá spurningu 2.
   a) Fann strax "Hide all"
   b) Ekkert mál.
   c) Hreinsar á hverri fyrir sig og fær eftir það ábendingu um "Hide all".
   d) Notar strax "Hide all"

4. Ferlar vélanna með kallmerki CMM713 og GRL781 skerast. Er aðskilnaður nægur?
   a) Athugar fyrst aðskilnaðarmörk, svo FPL, niðurstaða: Nei (Rétt)
   b) Skoðar í fljótu bragði án hjálpargagna og kemst að þeirri niðurstöðu að þær séu ekki aðskildar. Spyr nánar um hvernig áætlað radardrægi er reiknað. Niðurst: Nei
   c) Notar aðskilnaðarmörk til að sjá að aðskilnaður er ekki nægur. Niðurst: Nei
   d) Skoðar fyrst með aðskilnaðarmörkum, svo FPL. "Þetta er náttúrulega ekki nógu gott". Niðurst: Nei

5. Ferlar vélanna með kallmerki BAW284 og ACA844 skerast. Er aðskilnaður nægur?
   a) Skoðar fyrst FPL, sér að ferlar skerast, athugar aðskilnaðarmörk, niðurstaða: Nei (Rétt)
   b) Sýnist að þær haldist í 60nm fjarlægð. Skoðar bara FPL. Línurnar segja ekki neitt. Samkvæmt reglum sem gilda í dag er niðurstaða: Nei.
   c) Notar aðskilnaðarmörk, en sér ekki tíma fyrir þarnæsta punkt á flugplani. Kemst samt að niðurstöðu: Nei.
   d) Skoðar fyrst með aðskilnaðarmörkum. "ó nei, ó nei". Skoðar samt "Distance to". Niðurst: Nei.

6. Vélin með kallmerki ACA896 biður nú um flughæð F330. Athugaðu hvort aðskilnaður verði áfram nægur.

a) Hreinsar aðskilnaðarmörk, sýnir svo aðskilnaðarmörk fyrir báðar vélar ásamt FPL. Gerir ekki ráð fyrir vélinni að norðan, en eftir ábendingu um að athuga nágrenni skjámyndarinnar, sér þá GRL608. Niðurst: Nei (Rétt)
b) Finnur conflict við VIR20. Skoðar bara FPL, Reiknar 15-20 nm í skurðpunkti. Niðurst: Nei.
c) Sér út frá FPL að GRL608 sé í conflict, eftir að hafa fengið ábendingu um að athuga nágrenni (er búinn að vera svo snöggur með 1-5 að vélin er enn langt undan). Niðurst: Nei.
d) Sér fyrst ekkert athugavert, fær ábendingu um að athuga nágrenni. "Þarna er einn djöfull". Niðurst: Nei.

7. Er hjálplegt að sjá tákn flugvélanna breytast eftir þeim aðskilnaðarreglum sem gilda?
a) Já, mjög.
b) Já, þegar farið er að sýna blandað yfirlit. Gæti verið varhugavert. Möguleiki á að rugla táknum saman.
c) Já og nei. Heimurinn er aðeins flóknari en svo að það sé nóg. Nefnir svæðið þar sem aðsk. brotnar.
d) Já, sérstaklega m.t.t. radar.

8. Eru tákn flugvélanna nógu ólík til að skilja á milli þeirra aðskilnaðarreglna sem gilda?
a) Óljóst hvort hringurinn sé nógu ólíkur opnum tígli. Mjög gott að hafa krossinn svona stóran.
b) Lítill munur á opnum tígli og hring. Verða að vera mjög ólík.
c) Já, en ADS hringurinn mætti vel breytast í stærð eftir tíðni uppfærslna. Þá sést af stærð hringjarins hversu nýleg þessi staðsetning er.
d) Já, en hringur og tígull eru fulllík. "Maður þarf að skoða aftur."
e) Röng notkun á staðf. og ekki staðf. radar. Lítill munur á hring og tígli.

9. Finnst þér aðgerðirnar til að sýna og hreinsa aðskilnaðarmörk fullnægjandi?
a) Já, eðlileg notkun.
b) Já, virka alveg fínt.
c) Já, alveg eins og maður hefur lært það annars staðar.
d) Já, eins og gera má ráð fyrir.

10. Sýna aðskilnaðarmörk eitthvað sem þú gast ekki séð áður?
a) Já, ekki sést áður fyrir hverja vél fyrir sig. Hefur bara séð þessa hugmynd birtast sem statiskar flugleiðir (rútur).
b) Nei, ekkert sem maður er ekki með í kerfinu fyrir. Gæti nýst betur við flóknar leiðir þar sem punktar lenda á miðjum track.
Útskýrir nánar að á norðursvæðinu er algengt að punktar á flugplani séu valdir á milli lengdargráða, en það flækir handvirka útreikninga á aðskilnaði. Þessar breytingar gætu hjálpað í þeim stöðum.
c) Nei, væri sennilega jafn fljótur að sjá stöðuna út frá pappírsræmum.
d) Það er hægt að sjá það sama fyrir sér í huganum, en á þennan hátt eru minni líkur á mistökum. Augljósara.

11. Ættu aðskilnaðarmörk að vera sýnd meira eða minna áberandi, t.d. í skærari eða daufari lit í bakgrunninum, etv. annar litur?
a) Þetta er bara allt í lagi. Má ekki vera of skært, þá er það of mikið áreiti.

b) Nei, held nú ekki. Þetta væri mismunandi eftir þeirri litasamsetningu sem er notuð.
c) "Í raun er þetta allt í lagi svona". Jafnvel má notast við fyllta braut í bakgrunninum, semi-transparent. Hann útskýrir svo nánar að þar sem líklegt væri að aðeins tvær vélar séu skoðaðar í einu yrðu aðskilnaðarmörk skýrari við að sýna þau sem fyllt svæði. Línurnar einar sér falla ofan í hvora aðra og erfitt verður að meta hverjar séu parið sem marki svæðið og hverjar séu mörkin fyrir aðra vél.
d) Græni liturinn fellur of vel í bakgrunninn. Kannski þykkari línu, meira áberandi.
e) Já, þetta er betra en í fyrstu prufu.

12. Hvað finnst þér um þessa aðferð til að sýna aðskilnaðarmörk á kortinu?
a) Líst vel á það. "Góð viðbót að nýta tölvutæknina í þetta".
b) Ekki tími til að skoða þetta í radarsvæðinu, en gæti nýst við procedural umhverfi.
c) Í lagi, en mætti vera skýrari. Skoða mætti Traffic Advisory kerfi (TICAS). Hann teiknar hér aðskilnaðarmörk vélar sem perulaga svæði sem stækki eftir því hversu langt er frá síðustu staðfestri staðsetningu (í procedural). Bendir á að ef kl. er 12:01 getur hún verið einhvers staðar á milli 12:00:30 - 12:01:29 sem er heil mínúta. Á venjulegum hraða er það 8 nm sem fer fljótt að skipta máli þegar mörkin eru 30 nm (ADS).
d) Þetta er bara sniðugt, sérstaklega með ADS. Gæti auðveldað skipulag yfir lengri leiðir eða ef flugleiðirnar hafa ekki sömu viðmiðun (breiddargr. munar).

13. Hvað finnst þér um að sýnd eru saman aðskilnaðarmörk fyrir bæði staðfestan aðskilnað og áætlaðan RADAR aðskilnað?
a) Það er mjög lógískt. Gott að geta séð hvaða möguleikar eru framundan.
b) Enginn tími til að pæla í svona teikningum.
c) Flott að sjá, en maður vill helst bara sjá það sem er í gildi og er þekkt. Procedural vél á að vera slík þangað til hún er auðkennd.
d) Já, þetta er í fínu lagi. Gæti auðveldað skipulag annarra véla. Mætti vel vera valfrjálst hvort ytri línurnar (staðfest) væru notaðar.
e) Já, finnst það alveg ágætt. Reyndar er sá galli á að ef tvær brotalínur eru teiknaðar ofan í hvora aðra sést ekki lengur brotalína heldur heil lína.

14. 14.Eru þessar breytingar gagnlegar við að meta aðskilnað?
a) Já, þetta eru mjög notendavæn hjálpargögn.
b) Það gæti nýst við aðskilnað milli tveggja véla, annars sjá svar við sp. 12.
c) Of mikið af uppýsingum. Væri gagnlegt að einangra eitt plan (eina flughæð).
d) Já, sérstaklega við það sem nefnt var í svari við sp. 12.

15. Eru þessar breytingar nægjanlegar til að meta aðskilnað og ef ekki, hvað vantar?
a) Vantar tíma og lengdaraðskilnað. Tími er jú mikið notaður.
b) "Nægilegt með þeim skilyrðum sem við megum nota þetta." Það væri hægt að nota þetta til að meta aðskilnað framundan, t.d. til að sjá hvenær vélar eru nógu aðskildar til að komast í sömu hæð.
c) Nei, vantar að einangra plön (flughæðir) og að gera greinanlegri aðskilnaðarmörk hverrar vélar.
d) Já, fyrir hliðaraðskilnað. Maður notar hefðbundnar aðferðir fyrir lengdaraðskilnað.

16. Væri hjálplegt að sjá væntanlegan aðkomutíma flugvélar við hvern punkt í flugplani?
a) Já, jafnvel þótt möguleiki sé á tímamælingu með mús.
b) Ekkert svar.

c) Já, en varhugavert að tímarnir geta verið mismunandi: Proedural hefur tímann frá flugmanni, ADS hefur tímann frá tölvu um borð í flugvél, radar hefur tímann frá ratsjárstöð. Sjá svar við sp. 12 með dæmi um sama tíma sem gefur allt að 8 nm fjarlægðarmismun.
d) Já, tvímælalaust.

17. Væri hjálplegt að geta séð svæðið þar sem aðskilnaður er brotinn milli tveggja véla?
a) Já, það er hjálplegt að sýna mörkin.
b) Skilyrði að geta slökkt á því, mætti kannski verða hluti af conflict alert kerfinu.
c) Mætti reikna tímamismun í skurðpunkti flugplananna (þ.e. flugleiðanna). Þetta er hjálplegt, en það mætti vera ferningur, s.s. dekkt svæði sem afmarkar vandræðin.
d) Já, mætti skyggja svæðið þar sem bufferar skarast.
e) Já, en það væri ágætt að geta slökkt á því, þ.e. staðfest að maður hafi fengið skilaboðin. Vantar að geta séð tímamismun í punktinum þar sem vélarnar mætast. Vantar rauð strik á flugleiðir beggja véla. Vafasamt að gera ráð fyrir radar staðfestingu. Tenging milli tveggja véla væri þægilegt.

18. Er nógu skýrt um hvaða svæði er að ræða?
a) Ferlar beggja véla mættu vera rauðir. Báðar vélar eru í raun í conflict.
b) Svolítið villandi, mætti stika út (merkja) svæðið þar sem hvorug vél má halda sömu hæð.
c) Nei, mætti notast meira við punkta þar sem aðskilnaður brotnar (í sambandi við lengdaraðskilnað).
d) Ekki alveg, mætti skyggja. Auka punktar eru óþarfir. Ekki sniðugt að kerfið sé að plana hvenær radar samband næst (eins og sést á spurningu 3).
e) Nei, það þarf að huga betur að því miðað við raunverulegar aðskilnaðarreglur. Meet time opposite direction traffic er t.d. gildandi í einu dæminu.

19. Hverju myndir þú breyta?
a) Sjá svar við sp. 18.
b) Erfitt að sjá, m.v. að ADS er ekki þekkt (ekki búið að taka það kerfi í notkun).
c) Gera mætti meira úr því að verið sé aðallega að bera saman 2 vélar. Sem framlengingu af því, þá að verið sé að bera saman 1 vél við allar aðrar í sömu hæð. Hér kemur inn sú hugmynd að einangra ákveðið flugplan (flughæð) og útiloka allt annað frá íhugun.
d) Ekkert sem ekki er þegar nefnt, en sérstaklega að nota skýrari liti á mörkunum.
e) Sjá svör við spurningum.

B.2 Introductory Text

Round One

Introductory text in Icelandic from round one of user testing can be seen in figure B.2.

Round Two

Introductory text in Icelandic from round two of user testing can be seen in figure B.3.

Aðskilnaðarreglur á korti flugumferðarstjóra

Þetta verkefni gengur út á að athuga hvernig best sé að sýna mismunandi aðskilnaðarreglur á korti flugumferðarstjóra. Þetta er gert með því að breyta flugherminum Amelia frá Flugkerfum hf.

Um herminn
Eftirfarandi aðgerðir eru til staðar í herminum:

- Valmyndir fyrir kort eða flugvél
  - Valmynd korts kemur fram við að smella með hægri hnappi á tómt kortið
  - Valmynd vélar kemur fram við að smella með hægri hnappi á tákn eða datablock vélar.
- Skoða flugplan vélar
  - Valkosturinn “Show FPL” á valmynd vélar sýnir flugplan vélarinnar með hnitum við hvern punkt á flugplaninu.
- Hægt er að miðjustilla kortið með því að smella á miðjuhnapp músarinnar (músahjólið)
- Zoom inn og út með músahjóli.

Hermirinn er frábrugðinn keyrsluumhverfinu m.a. á eftirfarandi hátt:

- Hægt er að fjarlægja af kortinu flugplön allra vélanna með því að velja “Hide all FPLs” á ýmist valmynd korts eða vélar.
- Datablock sýnir aðrar upplýsingar
  - Lína 1 sýnir kallmerki
  - Lína 2 sýnir flughæð
  - Lína 3 sýnir Mach tölu
  - Lína 4 sýnir næsta punkt á flugplani “/” komutíma á punktinn
- Radarsvæði eru hermd, notast er við meðaldrægi radarstöðva til að ákvarða hvort vél sé innan radar.

Breytingar tengdar þessu verkefni:

1. Tákn flugvéla breytist eftir einföldum reglum.
   - Ef flugvélin er innan RADAR er sýnt radartáknið: ◆
   - Annars ef hún hefur ADS búnað er sýnt ADS táknið: ○
   - Annars er sýnt venjulegt “procedural” tákn: +
2. Útmörk lágmarks aðskilnaðar er hægt að sýna fyrir hverja vél fyrir sig.
   - Þessi mörk er sú fjarlægð sem vélin krefst aðskilin frá hverri annarri vél.
   - Aðeins hliðaraðskilnaður er sýndur.
   - Flugumferðarstjóri velur að sýna eða fjarlægja aðskilnaðarmörk með því að hægri smella á flugvél og velja “Show separation” á valmyndinni sem birtist.
   - Flugumferðarstjóri getur fjarlægt öll aðskilnaðarmörk með því að hægri smella á kortið og velja “Hide all separation” í valmyndinni sem birtist.

Figure B.2: *The introductory text presented to all participants in the first user test, original Icelandic version.*

Aðskilnaðarreglur á korti flugumferðarstjóra

Þetta verkefni gengur út á að athuga hvernig best sé að sýna blöndu af mismunandi aðskilnaðarreglum á korti flugumferðarstjóra. Þetta er gert með því að breyta flugherminum Amelia frá Flugkerfum hf.

Um herminn
Eftirfarandi aðgerðir eru til staðar í herminum:
- Valmyndir fyrir kort eða flugvél
  - Valmynd korts kemur fram við að smella með hægri hnappi á tómt kortið
  - Valmynd vélar kemur fram við að smella með hægri hnappi á tákn eða datablock vélar.
- Skoða flugplan vélar
  - Valkosturinn “Show FPL” á valmynd vélar sýnir flugplan vélarinnar með hnitum við hvern punkt á flugplaninu.
- Hægt er að miðjustilla kortið með því að smella á miðjuhnapp músarinnar (músahjólið)
- Zoom inn og út með músahjóli.

Hermirinn er frábrugðinn keyrsluumhverfinu m.a. Á eftirfarandi hátt:
- Hægt er að fjarlægja af kortinu flugplön allra vélanna með því að velja “Hide all FPLs” á ýmist valmynd korts eða vélar.
- Datablock sýnir aðrar upplýsingar
  - Lína 1 sýnir kallmerki
  - Lína 2 sýnir flughæð og Mach tölu
  - Lína 3 sýnir næsta punkt á flugplani
- Radarsvæði eru hermd, notast er við meðaldrægi radarstöðva til að ákvarða hvort vél sé innan radar.
- Til að herma staðfestingu á radarsambandi er notaður valkosturinn “Squawk ID” á valmynd þeirrar vélar sem á að staðfesta.

Breytingar tengdar þessu verkefni:
1. Tákn hverrar flugvélar breytist eftir einföldum reglum.
   - Ef flugvélin er innan RADAR og staðfest, er sýnt radartáknið: ◆
   - Ef hún er innan RADAR en er ekki staðfest, er sýnt opið radartákn: ◇
   - Annars ef hún hefur ADS búnað er sýnt ADS táknið: ○
   - Annars er sýnt venjulegt “procedural” tákn: ✚
2. Útmörk lágmarks aðskilnaðar er hægt að sýna fyrir hverja vél fyrir sig.
   - Þessi mörk er sú fjarlægð sem vélin krefst aðskilin frá hverri annarri vél.
   - Aðeins hliðaraðskilnaður er sýndur.
   - Ef radarsamband hefur ekki verið staðfest fyrir vélina eru aðskilnaðarmörk ennþá ýmist ADS eða procedural mörkin. Þá eru RADAR mörk sýnd með brotalínu.
   - Flugumferðarstjóri velur að sýna eða fjarlægja aðskilnaðarmörk með því að hægri smella á flugvél og velja “Show separation” á valmyndinni sem birtist.
   - Flugumferðarstjóri getur fjarlægt öll aðskilnaðarmörk með því að hægri smella á kortið og velja “Hide all separation” í valmyndinni sem birtist.
3. Ef fyrirsjáanlegt er að aðskilnaður milli tveggja véla er brotinn gerist eftirfarandi:
   - Þeir punktar á flugplani annarrar þar sem hún ferðast innan aðskilnaðarmarka hinnar eru merktir með rauðum deplum auk staðsetningar og tíma.
   - Þeir punktar á aðskilnaðarmörkum annarrar þar sem aðskilnaður hennar er brotinn eru merktir með rauðum deplum.
   - Sá hluti flugplans vélar þar sem áætlað er að hún sé innan aðskilnaðarmarka annarrar vélar er teiknaður í rauðum lit.
   - Þar sem þessi virkni er ekki útfærð í herminum sést á myndum hér að neðan hvernig þetta kemur til með að birtast flugumferðarstjóra.

Figure B.3: *The introductory text presented to all participants in the second user test, original Icelandic version.*

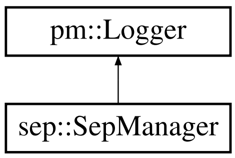
Appendix C

New Classes

C.1 SepManager

C.1.1 sep::SepManager Class Reference

`#include <SepManager.h>`
Inheritance diagram for sep::SepManager::



Public Types

- typedef boost::shared_ptr< **SepManager** > **Ptr**

  Provide a shared pointer type for shorthand notation.

Public Member Functions

- virtual ~**SepManager** ()

  Destructor.

- bool **initialize** ()

  Initialize the separation manager.

- virtual void **post_addChild** (**PM** ∗parent, **PM** ∗child)

  Part of the interface for **pm::Logger**(p. 224).

- virtual void **pre_deleteChild** (**PM** ∗parent, **PM** ∗child)

  Part of the interface for **pm::Logger**(p. 224).

- virtual void **post_set** (**PM** $*$node, const **pm::Property** &property)

  Part of the interface for **pm::Logger**(p. 224).

- **Tube::Ptr getTube** (const **PK** &pk) const

  Get access to the tube for pk.

Static Public Member Functions

- static **Ptr getPtr** (**SepManager** $*$in)

  Factory method for easy creation of shared pointers.

- static **Ptr getInstance** ()

  Factory method for easy access to the only instance of **SepManager**(p. 89).

Protected Types

- typedef **pm::PropertyMap PM**

  Shorthand notation typedef.

- typedef **pm::PrimaryKey PK**

  Shorthand notation typedef.

- typedef std::map$<$ **PM::string**, **Tube::Ptr** $>$ **TubeMap**

  The internal **Tube**(p. 97) *storage type.*

Protected Member Functions

- **SepManager** ()

  Construct the Sepmanager with reference to pm::defaultPMTree.

Static Protected Member Functions

- static **PM::Ref getAirTraffic** (**PM::Ref** pmRoot)

  Get AirTraffic node in the Property Map below pmRoot.

Private Attributes

- **TubeMap m_tubes**

  All flight tubes are recorded here and kept up to date via Logger *notifications.*

Static Private Attributes

- static **Ptr m_instance** = SepManager::getPtr(new **SepManager**())
  The only instance of **SepManager**(p. 89) *that should ever be created.*

C.1.1.1 Detailed Description

Manager of separation calculations.
This class is the central hub for calculating separation. It receives updates to the propertymap and recalculates flight tubes correspondingly.

C.1.1.2 Member Typedef Documentation

typedef pm::PrimaryKey sep::SepManager::PK `[protected]`
Shorthand notation typedef.

typedef pm::PropertyMap sep::SepManager::PM `[protected]`
Shorthand notation typedef.

typedef boost::shared_ptr<SepManager> sep::SepManager::Ptr
Provide a shared pointer type for shorthand notation.

typedef std::map<PM::string, Tube::Ptr> sep::SepManager::TubeMap `[protected]`

The internal **Tube**(p. 97) storage type.
We need to map flight callsign to **Tube**(p. 97).

C.1.1.3 Constructor & Destructor Documentation

sep::SepManager::~SepManager () `[virtual]`
Destructor.

sep::SepManager::SepManager () `[protected]`
Construct the Sepmanager with reference to pm::defaultPMTree.
The only constructor is protected to prevent direct creation. Use **getInstance()**(p. 91).

See also:
getInstance()(p. 91)

C.1.1.4 Member Function Documentation

static PM::Ref sep::SepManager::getAirTraffic (PM::Ref *pmRoot*) `[static, protected]`
Get AirTraffic node in the Property Map below *pmRoot*.

static Ptr sep::SepManager::getInstance () `[inline, static]`
Factory method for easy access to the only instance of **SepManager**(p. 89).

static Ptr sep::SepManager::getPtr (SepManager ∗ *in*) `[inline, static]`
Factory method for easy creation of shared pointers.

Tube::Ptr sep::SepManager::getTube (const PK & *pk*) const
Get access to the tube for *pk*.

bool sep::SepManager::initialize ()
Initialize the separation manager.

void sep::SepManager::post_addChild (PM ∗ *parent*, PM ∗ *child*) `[virtual]`
Part of the interface for **pm::Logger**(p. 224).

See also:
  pm::Logger(p. 224)

void sep::SepManager::post_set (PM ∗ *node*, const pm::Property & *property*) `[virtual]`
Part of the interface for **pm::Logger**(p. 224).

See also:
  pm::Logger(p. 224)

void sep::SepManager::pre_deleteChild (PM ∗ *parent*, PM ∗ *child*) `[virtual]`
Part of the interface for **pm::Logger**(p. 224).

See also:
  pm::Logger(p. 224)

C.1.1.5 Member Data Documentation

SepManager::Ptr sep::SepManager::m_instance = SepManager::getPtr(new SepManager()) `[static, private]`
The only instance of **SepManager**(p. 89) that should ever be created.

TubeMap sep::SepManager::m_tubes `[private]`
All flight tubes are recorded here and kept up to date via *Logger* notifications.
The **Tube**(p. 97) relating to flight with name "n" is stored in m_tubes["n"].

C.1.2 Header File

```
/*! \file SepManager.h
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains SepManager class declaration
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/18 14:24:00 $
 --------------------------------------------------------------------------*/
```

```
#ifndef _SEPMANAGER_H_
#define _SEPMANAGER_H_

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

#include "pm/Logger.h"
#include "pm/PropertyMap.h"
#include "pm/Property.h"
#include "pm/PrimaryKey.h"

#include "sep/TubePhase.h"
#include "sep/Tube.h"
//#include "sepMap/TubeHandler.h"

/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/
#include <cstdlib>
#include <list>
#include <map>

/*---------------------------------------------------------------------------
 | Forward class declarations
 ---------------------------------------------------------------------------*/

/*! The sep namespace is for separation calculations. */
namespace sep
{
  /*-------------------------------------------------------------------------
   | Forward class declarations
   -------------------------------------------------------------------------*/
  // Overview of sep:
  class TubePhase;  // is a single phase in a flight tube.
  class Tube;       // is a list of TubePhase elements.
                    // It represents a complete flight based on an aircrafts flight plan
//  class TubeHandler;// is the view component of a flight tube.
                    // It is responsible for drawing separation boundaries with conflict areas.
  //class SepManager; // is the overseeing entity that warns about conflicts and calculates tubes


  /*-------------------------------------------------------------------------
   | Namespace wide stuff
   -------------------------------------------------------------------------*/
  // Simplest to just introduce the whole namespace instead of typedef magic.
//  using namespace sim2;

  // Still we use typedef magic for some poor buggers.
//  typedef pm::PropertyMap::Ref PMRef;
//  typedef sim2::UtcTime UtcTime;

  /*!
   * Manager of separation calculations.
   *
   * This class is the central hub for calculating separation.
   * It receives updates to the propertymap and recalculates flight tubes correspondingly.
   */
  class SepManager : public pm::Logger
```

```
  {
    //! Allow a test class to access the internals.
    //friend class TestSepManager;

  protected:
    //! Shorthand notation typedef
    typedef pm::PropertyMap PM;
    //! Shorthand notation typedef
    typedef pm::PrimaryKey PK;
    //! The internal \a Tube storage type.
    /*! We need to map flight callsign to \a Tube. */
    typedef std::map<PM::string, Tube::Ptr> TubeMap;

  public:
    //! Provide a shared pointer type for shorthand notation.
    typedef boost::shared_ptr<SepManager> Ptr;
    //! Factory method for easy creation of shared pointers.
    static inline Ptr getPtr(SepManager *in) { return Ptr(in); }
    //! Factory method for easy access to the only instance of SepManager.
    static inline Ptr getInstance() { return m_instance; }

    //! Destructor
    virtual ~SepManager();

  /*------------------------------------------------------------------------
   | Public Methods
   ------------------------------------------------------------------------*/
    //! Initialize the separation manager.
    bool initialize();

   //! Part of the interface for \a pm::Logger.
   /*! \sa pm::Logger */
    virtual void post_addChild(PM *parent, PM *child);

   //! Part of the interface for \a pm::Logger.
   /*! \sa pm::Logger */
    virtual void pre_deleteChild(PM *parent, PM *child);

   //! Part of the interface for \a pm::Logger.
   /*! \sa pm::Logger */
    virtual void post_set(PM *node, const pm::Property &property);

    //! Get access to the tube for \a pk.
    Tube::Ptr getTube(const PK& pk) const;

   protected:
    //! Construct the Sepmanager with reference to pm::defaultPMTree.
    /*! The only constructor is protected to prevent direct creation. Use getInstance().
     * \sa getInstance()
     */
    SepManager();

    //! Get AirTraffic node in the Property Map below \a pmRoot.
    static PM::Ref getAirTraffic(PM::Ref pmRoot);

  private:
    //! All flight tubes are recorded here and kept up to date via \a Logger notifications.
    /*! The \a Tube relating to flight with name "n" is stored in m_tubes["n"]. */
    TubeMap m_tubes;

    //! The only instance of SepManager that should ever be created.
    static Ptr m_instance; // Look for initialization in SepManager.cpp
  };
}
#endif /* _SEPMANAGER_H_ */
```

C.1.3 Source File

```
/*! \file SepManager.cpp
    \author Olafur Freyr Hjalmsson
    \ingroup sep
```

```
    \brief Contains SepManager class methods
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/18 14:24:00 $
 ---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
 | Declaration of this class
 ---------------------------------------------------------------------------*/
#include "sep/SepManager.h"

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

#include "pm/Logger.h"
#include "pm/PropertyMap.h"
#include "pm/Property.h"
#include "pm/PrimaryKey.h"
#include "pm/PmTree.h"
#include "pm/PmUtils.h"
#include "world/ClientUtils.h"

#include "sep/TubePhase.h"
#include "sep/Tube.h"
//#include "sepMap/TubeHandler.h"

/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
/*---------------------------------------------------------------------------
 | Constructors / Destructor
 ---------------------------------------------------------------------------*/
  // Keep in mind that this constructor is protected and getInstance() should be used instead.
  SepManager::SepManager()
  {
    LOG_MACRO(LOG_WARNING, "::SepManager() default constructor");
  }

  SepManager::~SepManager()
  {
    LOG_MACRO(LOG_WARNING, "::~SepManager()");
    LOG_MACRO(LOG_WARNING, "SepManager instance had");
    for(TubeMap::iterator tubeIter = m_tubes.begin(); tubeIter != m_tubes.end(); ++tubeIter)
    {
      LOG_MACRO(LOG_WARNING, "Tube for aircraft "<<(*tubeIter).first);
    }

    PM::Ref pmRoot = pm::PmTree::getDefaultRoot();
    if(!pmRoot) {
      LOG_MACRO(LOG_WARNING, "No default PM root.");
    }
```

```
    else {
      PM::Ref pmAirTraffic = getAirTraffic(pmRoot);
      if(!pmAirTraffic) {
        LOG_MACRO(LOG_WARNING, "No Active AirTraffic.");
      }
      else {
        pmAirTraffic->removeLogger( this );
      }
    }
  }
/*---------------------------------------------------------------------------
 | Public function definitions
 --------------------------------------------------------------------------*/
  void SepManager::post_addChild(PM */*parent*/, PM *child)
  {
    LOG_MACRO(LOG_WARNING, "::post_addChild()");
    if(child->type() == "Aircraft") {
      LOG_MACRO(LOG_WARNING, "post_addChild() caught it when the aircraft named "<<child->name()<<" was added to PM tree");
      PM::string name(child->name());
      m_tubes[name] = Tube::create(PK( child ));
    }
  }

  void SepManager::pre_deleteChild(PM */*parent*/, PM *child)
  {
    LOG_MACRO(LOG_WARNING, "::post_deleteChild()");
    if(child->type() == "Aircraft") {
      LOG_MACRO(LOG_WARNING, "post_deleteChild() caught it when the aircraft named "<<child->name()<<" was removed from PM tree");
    }
  }

  void SepManager::post_set(PM *node, const pm::Property &property)
  {
    LOG_MACRO(LOG_WARNING, "::post_set()");
    if(node->type() == "Aircraft") {
      LOG_MACRO(LOG_WARNING, "post_set() caught it when the aircraft named "<<node->name()<<" had the property "<<property.name<<" changed to "<<property.value);
    }
  }

/*--------------------------------------------------------------------------*/
  Tube::Ptr SepManager::getTube(const pm::PrimaryKey &pk) const
  {
    LOG_MACRO(LOG_WARNING, "::getTube()");
    return Tube::create((pm::PrimaryKey&)pk);
  }

  bool SepManager::initialize()
  {
    LOG_MACRO(LOG_WARNING, "::initialize() : hello.");
    PM::Ref pmRoot = pm::PmTree::getDefaultRoot();
    if(!pmRoot) {
      LOG_MACRO(LOG_WARNING, "No default PM root.");
      return false; // No PM root is an unexpected error, so we return false
    }
    PM::Ref pmAirTraffic = getAirTraffic(pmRoot);
    if(!pmAirTraffic) {
      LOG_MACRO(LOG_WARNING, "No Active AirTraffic.");
      return false; // No scenario is an unexpected error, so we return false
    }

    m_tubes.clear();
    PM::RefList acList = pm::findByType(pmAirTraffic, "Aircraft");
    if(acList.empty()) {
      LOG_MACRO(LOG_WARNING, "Empty acList.");
    }
    else {
      LOG_MACRO(LOG_WARNING, "Looping.");
      for(PM::RefList::iterator acIter = acList.begin(); acIter != acList.end(); ++acIter)
      {
```

```
        PM::string name((*acIter)->name());
        LOG_MACRO(LOG_WARNING, "Adding flight "<<name.c_str()<<" to SepManager tube map");

        m_tubes[name] = Tube::create(PK( *acIter ));
        LOG_MACRO(LOG_WARNING, "This ac done.");
      } //for each aircraft
    }
    // Catch all AirTraffic Logger events
    LOG_MACRO(LOG_WARNING, "Setting Logger below AirTraffic.");
    pmAirTraffic->setLogger(this);
    LOG_MACRO(LOG_WARNING, "::initialize() : done.");
    return true;
  }
/*---------------------------------------------------------------------------
  | Static class members
  ---------------------------------------------------------------------------*/

  pm::PropertyMap::Ref SepManager::getAirTraffic(pm::PropertyMap::Ref pmRoot)
  {
    LOG_MACRO(LOG_WARNING, "::getAirTraffic() : hello.");
    if(!pmRoot) {
      LOG_MACRO(LOG_WARNING, "Received invalid PM root as parameter.");
      return pmRoot;
    }
    PM::Ref scenario = world::findActiveScenario(pmRoot);
    if(!scenario) {
      LOG_MACRO(LOG_WARNING, "No Active Scenario.");
      return pmRoot;
    }

    return pm::findOneByType(scenario, "AirTraffic");
  }

  SepManager::Ptr SepManager::m_instance = SepManager::getPtr(new SepManager());

} //namespace
```

C.2 Tube

C.2.1 sep::Tube Class Reference

#include <Tube.h>

Public Types

- typedef std::list< **TubePhase::Ptr** > **PhaseList**

  Provide a list type for shorthand notation.

- typedef boost::shared_ptr< **Tube** > **Ptr**

  Provide a shared pointer type for shorthand notation.

Public Member Functions

- **Tube** ()

  Construct the **Tube**(p. 97).

- **Tube** (**PK** pk)

  Construct a new **Tube**(p. 97) *from an aircraft.*

- $\sim$**Tube** ()

  Destructor.

- **PhaseList getPhases** ()

  Get access to the internal **TubePhase**(p. 103).

- **PK getPrimaryKey** () const

  Get access to the PrimaryKey for the aircraft for which this **Tube**(p. 97) *represents.*

Static Public Member Functions

- static **Ptr getPtr** (**Tube** $*$in)

  Factory method for easy creation of shared pointers.

- static **Ptr create** (**PK** ac)

  Factory method for easy creation of shared Tubes.

Protected Types

- typedef **sep::TubePhase::SepType SepType**

  SepType is defined privately, only for convenience.

- typedef **pm::PrimaryKey PK**

  Shorthand notation typedef.

Protected Member Functions

- void **update** ()

  Update phases. The flight path is traversed and the complete tube is recalculated.

Private Attributes

- **PhaseList m_phases**

  List of flight phases.

- **PK m_ac**

  Reference back to the aircraft in question.

C.2.1.1 Detailed Description

Complete flight tube.
This class represents a complete flight tube. It represents a complete flight based on an aircrafts flight plan and actual travelled path.

C.2.1.2 Member Typedef Documentation

typedef std::list<TubePhase::Ptr> sep::Tube::PhaseList
Provide a list type for shorthand notation.

typedef pm::PrimaryKey sep::Tube::PK `[protected]`
Shorthand notation typedef.

typedef boost::shared_ptr<Tube> sep::Tube::Ptr
Provide a shared pointer type for shorthand notation.

typedef sep::TubePhase::SepType sep::Tube::SepType `[protected]`
SepType is defined privately, only for convenience.

C.2.1.3 Constructor & Destructor Documentation

sep::Tube::Tube ()
Construct the **Tube**(p. 97).

sep::Tube::Tube (PK *pk*)
Construct a new **Tube**(p. 97) from an aircraft.

sep::Tube::~Tube ()
Destructor.

C.2.1.4 Member Function Documentation

static Ptr sep::Tube::create (PK *ac*) `[inline, static]`
Factory method for easy creation of shared Tubes.

Tube::PhaseList sep::Tube::getPhases ()
Get access to the internal **TubePhase**(p. 103).

PK sep::Tube::getPrimaryKey () const `[inline]`
Get access to the PrimaryKey for the aircraft for which *this* **Tube**(p. 97) represents.

static Ptr sep::Tube::getPtr (Tube * *in*) `[inline, static]`
Factory method for easy creation of shared pointers.

void sep::Tube::update () `[protected]`
Update phases. The flight path is traversed and the complete tube is recalculated. This only works if *in* is before or after *this* tube.

C.2.1.5 Member Data Documentation

PK sep::Tube::m_ac `[private]`
Reference back to the aircraft in question.

PhaseList sep::Tube::m_phases `[private]`
List of flight phases.

C.2.2 Header File

```
/*! \file Tube.h
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains Tube class declarations
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/09 13:28:00 $
 ---------------------------------------------------------------------------*/
#ifndef _TUBE_H_
#define _TUBE_H_

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

//#include "logmacro.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
//#include "pm/PropertyMap.h"
//#include "pm/Property.h"
#include "pm/PrimaryKey.h"

#include "sep/TubePhase.h"

/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/
#include <cstdlib>
#include <list>

/*---------------------------------------------------------------------------
 | Forward class declarations
 ---------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
  // Overview of sep:
```

```
  // TubePhase is a single phase in a flight tube.
  // Tube is a list of TubePhase elements and represents a complete flight based on an aircrafts flight plan
  // TubeHandler is the view component of a flight tube.
  //   It is responsible for drawing separation boundaries with conflict areas.
  // SepManager is the overseeing entity that warns about conflicts and calculates tubes

  class TubePhase;

  /*!
   * Complete flight tube.
   *
   * This class represents a complete flight tube.
   * It represents a complete flight based on an aircrafts flight plan and actual travelled path.
   */
  class Tube
  {
    //! Allow a test class to access the internals.
    //friend class TestTube;

  protected:
    //! SepType is defined privately, only for convenience.
    typedef sep::TubePhase::SepType SepType;
    //! Shorthand notation typedef
    typedef pm::PrimaryKey PK;

  public:
    //! Provide a list type for shorthand notation.
    typedef std::list<TubePhase::Ptr> PhaseList;


    //! Provide a shared pointer type for shorthand notation.
    typedef boost::shared_ptr<Tube> Ptr;
    //! Factory method for easy creation of shared pointers.
    static inline Ptr getPtr(Tube *in) { return Ptr(in); }
    //! Factory method for easy creation of shared Tubes.
    static inline Ptr create(PK ac) { return getPtr(new Tube(ac)); }

    //! Construct the Tube.
    Tube();

    //! Construct a new Tube from an aircraft.
    Tube(PK pk);

    //! Destructor
    ~Tube();

  /*-----------------------------------------------------------------------
   | Public Methods
   -----------------------------------------------------------------------*/
    //! Get access to the internal TubePhase
    PhaseList getPhases();

    //! Get access to the PrimaryKey for the aircraft for which \a this \a Tube represents.
    PK getPrimaryKey() const {return m_ac;}

    //! Create a new \a Tube with \a this and \a in merged.
    /*! This only works if \a in is before or after \a this tube. */
//    Tube::Ptr merge(const Tube &in) const;

    //! Create a new tube with this and in merged.
//    Tube::Ptr merge(const Tube::Ptr in) const;

  protected:
    //! Update phases. The flight path is traversed and the complete tube is recalculated.
    void update();

  private:
    PhaseList m_phases;  //!< List of flight phases
    PK m_ac;             //!< Reference back to the aircraft in question.
  };
```

```
}
#endif /* _TUBE_H_ */
```

C.2.3 Source File

```
/*! \file Tube.cpp
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains Tube class methods
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/09 13:28:00 $
 ---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
 | Declaration of this class
 ---------------------------------------------------------------------------*/
#include "sep/Tube.h"

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
//#include "pm/PropertyMap.h"
//#include "pm/Property.h"
#include "pm/PrimaryKey.h"

#include "sep/TubePhase.h"

/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/
#include <cstdlib>
#include <list>

// Look for sep namespace documentation in SepManager.h
namespace sep
{
/*---------------------------------------------------------------------------
 | Constructors / Destructor
 ---------------------------------------------------------------------------*/
  Tube::Tube()
  {
    LOG_MACRO(LOG_WARNING, "::Tube() default constructor");
  }

  Tube::Tube(PK pk) :
    m_ac(pk)
  {
    if(pk.isValid()) {
      LOG_MACRO(LOG_WARNING, "::Tube(PrimaryKey="<<pk.ref()->name().c_str()<<")");
    }
    else {
      LOG_MACRO(LOG_WARNING, "::Tube(PrimaryKey) with invalid PrimaryKey!");
    }
```

```
  }

  Tube::~Tube()
  {
    LOG_MACRO(LOG_WARNING, "::~Tube()");
  }
/*-----------------------------------------------------------------------------
 | Public function definitions
 ----------------------------------------------------------------------------*/
  Tube::PhaseList Tube::getPhases()
  {
    LOG_MACRO(LOG_WARNING, "::getPhases()");
    if(m_phases.empty())
      update();
    return m_phases;
  }

/*
  Tube::Ptr Tube::merge(const Tube &in) const
  {
    LOG_MACRO(LOG_WARNING, "Tube::merge(Tube)");
    return Tube::Ptr();
  }

  Tube::Ptr Tube::merge(const Tube::Ptr in) const
  {
    LOG_MACRO(LOG_WARNING, "Tube::update(Tube::Ptr)");
    return Tube::Ptr(this);
  }
*/
/*-----------------------------------------------------------------------------
 | Protected function definitions
 ----------------------------------------------------------------------------*/
  void Tube::update()
  {
    LOG_MACRO(LOG_WARNING, "::update()");
  }
} //namespace
```

C.3 TubePhase

C.3.1 sep::TubePhase Class Reference

`#include <TubePhase.h>`

Public Types

- typedef boost::shared_ptr< **TubePhase** > **Ptr**

  Provide a shared pointer type for shorthand notation.

- enum **SepType** { **SEP_PROC** = 0, **SEP_RADAR**, **SEP_ADS**, **NR_SEP_TYPES** }

  SepType *is a numeric representation of the distinct types of separation rules.*

Public Member Functions

- **TubePhase** (Position p, Level l, Heading h, Speed s, UtcTime t, **SepType** st=SEP_-PROC)

  Construct a new phase of a **Tube**(p. 97).

- $\sim$**TubePhase** ()

  Destructor.

- void **update** (const **TubePhase** &ref, bool before)

  Update this phase based on ref*,* before *tells us if ref comes before* this*.*

- void **update** (const **TubePhase** &before, const **TubePhase** &after)

  Update this phase based on before *and* after*,* this *is in between.*

- double **getSeparation** ()

  Get the lateral separation distance in Nautical Miles.

- bool **IsValid** () const

  Get internal status of this phase of the tube.

- Position **getPosition** () const

  Get the aircraft position exiting this phase of the tube.

- Level **getLevel** () const

  Get the aircraft flight level exiting this phase of the tube.

- Heading **getHeading** () const

  Get the aircraft heading exiting this phase of the tube.

- Speed **getSpeed** () const

  Get the aircraft speed exiting this phase of the tube.

- UtcTime **getTime** () const

  Get the time when the aircraft exits this phase of the tube.

- **SepType getSepType** () const

  Get the separation type which is valid for the aircraft during this phase of the tube.

- double **getAngle** () const

  Get the relative turn angle the aircraft enters when exiting this phase of the tube.

- double **getClimb** () const

  Get the aircraft climb rate during this phase of the tube.

- double **getAcceleration** () const

  Get the aircraft acceleration rate during this phase of the tube.

- Position **getSepLeft** () const

Get the separation boundary point to the left of **getPosition**()(p. 108).

- Position **getSepRight** () const

  Get the separation boundary point to the right of **getPosition()**(p. 108).

Static Public Member Functions

- static **Ptr getPtr** (**TubePhase** $*$in)

  Factory method for easy creation of shared pointers.

- static **Ptr create** (Position p, Level l, Heading h, Speed s, UtcTime t, **SepType** st=SEP_PROC)

  Factory method for easy creation of shared TubePhases.

- static double **getSeparation** (**SepType** sepType)

  Get the lateral separation distance in Nautical Miles for the specified sepType.

Protected Member Functions

- **TubePhase** ()

  Construct empty **TubePhase**(p. 103).

Private Attributes

- bool **m_valid**

  Does this object contain values representing a valid flight phase?

- bool **m_initialized**

  Have all derived values been processed?

- Position **m_position**

  Aircraft exit position.

- Level **m_level**

  Aircraft exit flight level.

- Heading **m_heading**

  Aircraft heading during this phase.

- Speed **m_speed**

  Aircraft exit speed.

- UtcTime **m_time**

  Time the aircraft exits this phase.

- **SepType m_sepType**

  Type of separation in place during this phase.

- double **m_angle**

  Relative turn angle when exiting this phase (derived).

- double **m_climb**

  Climb rate during this phase (derived).

- double **m_acceleration**

  Acceleration during this phase (derived).

- Position **m_sepPosLeft**

  Separation boundary point to the left of m_position *(derived).*

- Position **m_sepPosRight**

  Separation boundary point to the right of m_position *(derived).*

Static Private Attributes

- static double **m_sepDist** [NR_SEP_TYPES] = {60.0,10.0,30.0}

  Separation distances corresponding with each SepType (in nautical miles).

C.3.1.1 Detailed Description

A single phase of a flight tube.
This class represents a single phase of a flight tube. A phase of a tube represents a specific status of an aircraft. During a phase the aircraft is either: in level flight, about to enter a turn, climbing or accelerating position, level, heading speed and time are all valid at the exit point sepType is the predefined separation type valid during this phase angle is the relative turn angle (in radians) of the aircraft at the exit point (positive is right turn) climb is the rate of climb during the phase. acceleration is the acceleration of the aircraft during the phase.

C.3.1.2 Member Typedef Documentation

typedef boost::shared_ptr<TubePhase> sep::TubePhase::Ptr
Provide a shared pointer type for shorthand notation.

C.3.1.3 Member Enumeration Documentation

enum sep::TubePhase::SepType
SepType is a numeric representation of the distinct types of separation rules.

Enumeration values:

SEP_PROC Procedural separation, 60 nm, i.e. default.

SEP_RADAR RADAR separation, 10 nm.

SEP_ADS ADS separation, 30 nm.

NR_SEP_TYPES Total number of sep types, used when creating arrays and iterating through them.

C.3.1.4 Constructor & Destructor Documentation

sep::TubePhase::TubePhase () `[protected]`
Construct empty **TubePhase**(p. 103).
This is not allowed, so the constructor is marked protected.

sep::TubePhase::TubePhase (Position *p*, Level *l*, Heading *h*, Speed *s*, UtcTime *t*, SepType *st* = `SEP_PROC`**)**
Construct a new phase of a **Tube**(p. 97).

sep::TubePhase::∼TubePhase ()
Destructor.

C.3.1.5 Member Function Documentation

static Ptr sep::TubePhase::create (Position *p*, Level *l*, Heading *h*, Speed *s*, UtcTime *t*, SepType *st* = `SEP_PROC`**)** `[inline, static]`
Factory method for easy creation of shared TubePhases.

double sep::TubePhase::getAcceleration () const `[inline]`
Get the aircraft acceleration rate during this phase of the tube.

double sep::TubePhase::getAngle () const `[inline]`
Get the relative turn angle the aircraft enters when exiting this phase of the tube.
The angle is positive when turning right, negative when turning left.

double sep::TubePhase::getClimb () const `[inline]`
Get the aircraft climb rate during this phase of the tube.

Heading sep::TubePhase::getHeading () const `[inline]`
Get the aircraft heading exiting this phase of the tube.

Level sep::TubePhase::getLevel () const `[inline]`
Get the aircraft flight level exiting this phase of the tube.

Position sep::TubePhase::getPosition () const `[inline]`
Get the aircraft position exiting this phase of the tube.

static Ptr sep::TubePhase::getPtr (TubePhase ∗ *in*) `[inline, static]`
Factory method for easy creation of shared pointers.

static double sep::TubePhase::getSeparation (SepType *sepType*) `[inline, static]`
Get the lateral separation distance in Nautical Miles for the specified *sepType*.

double sep::TubePhase::getSeparation () `[inline]`
Get the lateral separation distance in Nautical Miles.

Position sep::TubePhase::getSepLeft () const `[inline]`
Get the separation boundary point to the left of **getPosition()**(p. 108).

Position sep::TubePhase::getSepRight () const `[inline]`
Get the separation boundary point to the right of **getPosition()**(p. 108).

SepType sep::TubePhase::getSepType () const `[inline]`
Get the separation type which is valid for the aircraft during this phase of the tube.

Speed sep::TubePhase::getSpeed () const `[inline]`
Get the aircraft speed exiting this phase of the tube.

UtcTime sep::TubePhase::getTime () const `[inline]`
Get the time when the aircraft exits this phase of the tube.

bool sep::TubePhase::IsValid () const `[inline]`
Get internal status of this phase of the tube.

void sep::TubePhase::update (const TubePhase & *before*, const TubePhase & *after*)
Update this phase based on *before* and *after*, *this* is in between.

void sep::TubePhase::update (const TubePhase & *ref*, bool *before*)
Update this phase based on *ref*, *before* tells us if ref comes before *this*.

C.3.1.6 Member Data Documentation

double sep::TubePhase::m_acceleration `[private]`
Acceleration during this phase (derived).

double sep::TubePhase::m_angle `[private]`
Relative turn angle when exiting this phase (derived).

double sep::TubePhase::m_climb `[private]`
Climb rate during this phase (derived).

Heading sep::TubePhase::m_heading `[private]`
Aircraft heading during this phase.

bool sep::TubePhase::m_initialized `[private]`
Have all derived values been processed?

Level sep::TubePhase::m_level `[private]`
Aircraft exit flight level.

Position sep::TubePhase::m_position `[private]`
Aircraft exit position.

double sep::TubePhase::m_sepDist = {60.0,10.0,30.0} `[static, private]`
Separation distances corresponding with each SepType (in nautical miles).

Position sep::TubePhase::m_sepPosLeft `[private]`
Separation boundary point to the left of *m_position* (derived).

Position sep::TubePhase::m_sepPosRight `[private]`
Separation boundary point to the right of *m_position* (derived).

SepType sep::TubePhase::m_sepType `[private]`
Type of separation in place during this phase.

Speed sep::TubePhase::m_speed `[private]`
Aircraft exit speed.

UtcTime sep::TubePhase::m_time `[private]`
Time the aircraft exits this phase.

bool sep::TubePhase::m_valid `[private]`
Does this object contain values representing a valid flight phase?

C.3.2 Header File

```
/*! \file TubePhase.h
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains TubePhase class declarations
*/

/*-------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/09 13:28:00 $
```

```
 ---------------------------------------------------------------------------*/
#ifndef _TUBEPHASE_H_
#define _TUBEPHASE_H_

/*----------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
#include "simGeo/Position.h"
#include "simBasicTypes/Level.h"
#include "simGeo/Heading.h"
#include "simBasicTypes/Speed.h"
#include "simBasicTypes/UtcTime.h"
#include "simGeo/ThreeVector.h"
#include "simBasicTypes/Distance.h"
#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
//#include "pm/PropertyMap.h"
//#include "pm/Property.h"
//#include "pm/PrimaryKey.h"

/*----------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/
#include <cstdlib>
#include <list>

/*----------------------------------------------------------------------------
 | Forward class declarations
 ---------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
  // Introduce namespace sim2 to avoid having to use typedef or reference the namespace every time
  using namespace sim2;

  // Overview of sep:
  // TubePhase is a single phase in a flight tube.
  // Tube is a list of TubePhase elements and represents a complete flight based on an aircrafts flight plan
  // TubeHandler is the view component of a flight tube.
  //   It is responsible for drawing separation boundaries with conflict areas.
  // SepManager is the overseeing entity that warns about conflicts and calculates tubes

  /*!
   * A single phase of a flight tube.
   *
   * This class represents a single phase of a flight tube.
   * A phase of a tube represents a specific status of an aircraft.
   * During a phase the aircraft is either:
   *  in level flight,
   *  about to enter a turn,
   *  climbing or
   *  accelerating
   * position, level, heading speed and time are all valid at the exit point
   * sepType is the predefined separation type valid during this phase
   * angle is the relative turn angle (in radians) of the aircraft at the exit point (positive is right turn)
   * climb is the rate of climb during the phase.
   * acceleration is the acceleration of the aircraft during the phase.
   */
  class TubePhase
  {
    //! Allow a test class to access the internals.
    //friend class TestTubePhase;

  public:
```

```
    //! Provide a shared pointer type for shorthand notation.
    typedef boost::shared_ptr<TubePhase> Ptr;

    //! \a SepType is a numeric representation of the distinct types of separation rules.
    typedef enum {
      SEP_PROC = 0, /*!< Procedural separation, 60 nm, i.e.\ default. */
      SEP_RADAR,    /*!< RADAR separation, 10 nm. */
      SEP_ADS,      /*!< ADS separation, 30 nm. */
      NR_SEP_TYPES  /*!< Total number of sep types, used when creating arrays and iterating through them. */
    } SepType;

    //! Factory method for easy creation of shared pointers.
    static inline Ptr getPtr(TubePhase *in) { return Ptr(in); }
    //! Factory method for easy creation of shared TubePhases.
    static inline Ptr create(Position p, Level l, Heading h, Speed s, UtcTime t, SepType st=SEP_PROC)
      { return getPtr(new TubePhase(p,l,h,s,t,st)); }

  protected:
    //! Construct empty TubePhase.
    /*! This is not allowed, so the constructor is marked protected. */
    TubePhase();

  public:
    //! Construct a new phase of a Tube.
    TubePhase(Position p, Level l, Heading h, Speed s, UtcTime t, SepType st=SEP_PROC);

    //! Destructor
    ~TubePhase();

  /*------------------------------------------------------------------------
    | Public Methods
    -----------------------------------------------------------------------*/
    //! Update this phase based on \a ref, \a before tells us if ref comes before \a this.
    void update(const TubePhase &ref, bool before);

    //! Update this phase based on \a before and \a after, \a this is in between.
    void update(const TubePhase &before, const TubePhase &after);

    //! Get the lateral separation distance in Nautical Miles.
    inline double getSeparation() {return getSeparation(m_sepType); }

    //! Get the lateral separation distance in Nautical Miles for the specified \a sepType.
    static inline double getSeparation(SepType sepType){return ((sepType<0 || sepType >= NR_SEP_TYPES)?m_sepDist[SEP_PROC]:m_sepDist[sepType]);}
  /*------------------------------------------------------------------------
    | Getters are inline (no setters, just make a new one dammit...)
    -----------------------------------------------------------------------*/
    //! Get internal status of this phase of the tube.
    inline bool IsValid()         const {return (m_valid && m_initialized);}

    //! Get the aircraft position exiting this phase of the tube.
    inline Position getPosition() const {return m_position;   }

    //! Get the aircraft flight level exiting this phase of the tube.
    inline Level    getLevel()    const {return m_level;      }

    //! Get the aircraft heading exiting this phase of the tube.
    inline Heading  getHeading()  const {return m_heading;    }

    //! Get the aircraft speed exiting this phase of the tube.
    inline Speed    getSpeed()    const {return m_speed;      }

    //! Get the time when the aircraft exits this phase of the tube.
    inline UtcTime  getTime()     const {return m_time;       }

    //! Get the separation type which is valid for the aircraft during this phase of the tube.
    inline SepType  getSepType()  const {return m_sepType;    }

    //! Get the relative turn angle the aircraft enters when exiting this phase of the tube.
```

```
    /*! The angle is positive when turning right, negative when turning left. */
    inline double   getAngle()    const {return m_angle;      }

    //! Get the aircraft climb rate during this phase of the tube.
    inline double   getClimb()    const {return m_climb;      }

    //! Get the aircraft acceleration rate during this phase of the tube.
    inline double   getAcceleration() const {return m_acceleration;}

    //! Get the separation boundary point to the left of \a getPosition().
    inline Position getSepLeft()  const {return m_sepPosLeft; }
    //! Get the separation boundary point to the right of \a getPosition().
    inline Position getSepRight() const {return m_sepPosRight;}

  private:

    bool      m_valid;        //!< Does this object contain values representing a valid flight
 phase?
    bool      m_initialized;  //!< Have all derived values been processed?

    Position m_position;    //!< Aircraft exit position.
    Level    m_level;       //!< Aircraft exit flight level.
    Heading  m_heading;     //!< Aircraft heading during this phase.
    Speed    m_speed;       //!< Aircraft exit speed.
    UtcTime  m_time;        //!< Time the aircraft exits this phase.
    SepType  m_sepType;     //!< Type of separation in place during this phase.

    double   m_angle;       //!< Relative turn angle when exiting this phase (derived).
    double   m_climb;       //!< Climb rate during this phase (derived).
    double   m_acceleration;//!< Acceleration during this phase (derived).

    Position m_sepPosLeft;  //!< Separation boundary point to the left  of \a m_position (
derived).
    Position m_sepPosRight; //!< Separation boundary point to the right of \a m_position (
derived).

    //! Separation distances corresponding with each SepType (in nautical miles)
    static double m_sepDist[NR_SEP_TYPES]; // Look for initialization in TubePhase.cpp
  };
}
#endif /* _TUBEPHASE_H_ */
```

C.3.3 Source File

```
/*! \file TubePhase.cpp
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains TubePhase class methods
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/18 14:24:00 $
 ---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
 | Declaration of this class
 ---------------------------------------------------------------------------*/
#include "sep/TubePhase.h"

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
#include "simGeo/Position.h"
#include "simBasicTypes/Level.h"
#include "simGeo/Heading.h"
```

```
#include "simBasicTypes/Speed.h"
#include "simBasicTypes/UtcTime.h"
#include "simGeo/ThreeVector.h"
#include "simBasicTypes/Distance.h"
#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
//#include "pm/PropertyMap.h"
//#include "pm/Property.h"
//#include "pm/PrimaryKey.h"

/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
/*---------------------------------------------------------------------------
  | Constructors / Destructor
  --------------------------------------------------------------------------*/
  TubePhase::TubePhase():
    m_valid(false), m_initialized(false)
  {
    LOG_MACRO(LOG_WARNING, "::TubePhase() default constructor");
  }

  TubePhase::TubePhase(Position p, Level l, Heading h, Speed s, UtcTime t, SepType st) :
    m_valid(true), m_initialized(false),
    m_position(p), m_level(l), m_heading(h), m_speed(s), m_time(t), m_sepType(st),
    m_angle(0.0), m_climb(0.0), m_acceleration(0.0),
    m_sepPosLeft(p), m_sepPosRight(p)
  {
    LOG_MACRO(LOG_WARNING, "::TubePhase(...) specific constructor");
  }

  TubePhase::~TubePhase()
  {
    LOG_MACRO(LOG_WARNING, "::~TubePhase()");
  }
/*---------------------------------------------------------------------------
  | Public function definitions
  --------------------------------------------------------------------------*/
  void TubePhase::update(const TubePhase & /*ref*/, bool /*before*/)
  {
    LOG_MACRO(LOG_WARNING, "::update(TubePhase,bool)");
  }

  void TubePhase::update(const TubePhase & /*before*/, const TubePhase & /*after*/)
  {
    LOG_MACRO(LOG_WARNING, "::update(TubePhase,TubePhase)");
  }

/*---------------------------------------------------------------------------
  | Static class members
  --------------------------------------------------------------------------*/
  double TubePhase::m_sepDist[] = {60.0,10.0,30.0}; // Initialization of the sep. distances
} //namespace
```

C.4 TubeHandler

C.4.1 sep::TubeHandler Class Reference

#include <TubeHandler.h>

Public Types

- typedef boost::shared_ptr< **TubeHandler** > **Ptr**

  Provide a shared pointer type for shorthand notation.

Public Member Functions

- **TubeHandler** ()

  Construct empty **TubePhase**(p. 103).

- virtual ~**TubeHandler** ()

  Destructor.

- virtual void **drawBoundaries** (ats_map::GPainter ∗painter, **pm::PropertyMap::Ref** pmRoot)

  Draw separation boundaries for the encapsulated **Tube**(p. 97).

Static Public Member Functions

- static **Ptr getPtr** (**TubeHandler** ∗in)

  Factory method for easy creation of shared pointers.

- static **Ptr create** ()

  Factory method for easy creation of shared TubePhases.

Private Attributes

- boost::shared_ptr< **Tube** > **m_tube**

  The **Tube**(p. 97) *handled by* this.

C.4.1.1 Detailed Description

Provides drawing of separation boundaries and conflict zones.

C.4.1.2 Member Typedef Documentation

typedef boost::shared_ptr<TubeHandler> sep::TubeHandler::Ptr
Provide a shared pointer type for shorthand notation.

C.4.1.3 Constructor & Destructor Documentation

sep::TubeHandler::TubeHandler ()
Construct empty **TubePhase**(p. 103).

sep::TubeHandler::∼TubeHandler () `[virtual]`
Destructor.

C.4.1.4 Member Function Documentation

static Ptr sep::TubeHandler::create () `[inline, static]`
Factory method for easy creation of shared TubePhases.

void sep::TubeHandler::drawBoundaries (ats_map::GPainter ∗ *painter*, pm::PropertyMap::Ref *pmRoot*) `[virtual]`
Draw separation boundaries for the encapsulated **Tube**(p. 97).

static Ptr sep::TubeHandler::getPtr (TubeHandler ∗ *in*) `[inline, static]`
Factory method for easy creation of shared pointers.

C.4.1.5 Member Data Documentation

boost::shared_ptr<Tube> sep::TubeHandler::m_tube `[private]`
The **Tube**(p. 97) handled by *this*.

C.4.2 Header File

```
/*! \file TubeHandler.h
    \author Olafur Freyr Hjalmsson
    \ingroup simMap
    \brief Contains TubeHandler class declaration
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/18 14:24:00 $
 ---------------------------------------------------------------------------*/
#ifndef _TUBE_HANDLER_H_
#define _TUBE_HANDLER_H_

/*---------------------------------------------------------------------------
 | Include local header files
 ---------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
#include "basicMap/GPainter.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
#include "pm/PropertyMap.h"
#include "pm/Property.h"
#include "pm/PrimaryKey.h"

//#include "sep/TubePhase.h"
#include "sep/Tube.h"

```

```
/*---------------------------------------------------------------------------
 | Include standard header files
 ---------------------------------------------------------------------------*/
#include <cstdlib>
//#include <cmath>
#include <list>

/*---------------------------------------------------------------------------
 | Forward class declarations
 ---------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
  class Tube;
  class TubePhase;

  /*!
   * Provides drawing of separation boundaries and conflict zones.
   */
  class TubeHandler
  {
    //! Allow a test class to access the internals.
    //friend class TestTubePhase;
  public:
    //! Provide a shared pointer type for shorthand notation.
    typedef boost::shared_ptr<TubeHandler> Ptr;
    //! Factory method for easy creation of shared pointers.
    static inline Ptr getPtr(TubeHandler *in) { return Ptr(in); }
    //! Factory method for easy creation of shared TubePhases.
    static inline Ptr create() { return getPtr(new TubeHandler()); }

    //! Construct empty TubePhase.
    TubeHandler();

    //! Destructor
    virtual ~TubeHandler();

  /*-------------------------------------------------------------------------
   | Public Methods
   -------------------------------------------------------------------------*/
    //! Draw separation boundaries for the encapsulated \a Tube
    virtual void drawBoundaries(ats_map::GPainter *painter, pm::PropertyMap::Ref pmRoot);

//  protected:

  private:
    boost::shared_ptr<Tube> m_tube; //!< The \a Tube handled by \a this.
  };
} // namespace

#endif // _TUBE_HANDLER_H_
```

C.4.3 Source File

```
/*! \file TubeHandler.cpp
    \author Olafur Freyr Hjalmsson
    \ingroup sep
    \brief Contains TubeHandler class methods
*/

/*---------------------------------------------------------------------------
 | Copyright (c) 2005, Tern Systems, Inc.  All Rights Reserved.
 |
 | $Revision: 1.0 $
 | $Date: 2005/02/18 14:24:00 $
 ---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
 | Declaration of this class
 ---------------------------------------------------------------------------*/
```

```
#include "sepMap/TubeHandler.h"

/*----------------------------------------------------------------------------
 | Include local header files
 ----------------------------------------------------------------------------*/
#include "boost/shared_ptr.hpp"

#include "logmacro.h"
#include "basicMap/GPainter.h"
//#include "simGeo/Position.h"
//#include "simBasicTypes/Level.h"
//#include "simGeo/Heading.h"
//#include "simBasicTypes/Speed.h"
//#include "simBasicTypes/UtcTime.h"
//#include "simGeo/ThreeVector.h"
//#include "simBasicTypes/Distance.h"
//#include "simBasicTypes/Duration.h"

//#include "pm/Logger.h"
#include "pm/PropertyMap.h"
#include "pm/Property.h"
#include "pm/PrimaryKey.h"

//#include "sep/TubePhase.h"
#include "sep/Tube.h"

/*----------------------------------------------------------------------------
 | Include standard header files
 ----------------------------------------------------------------------------*/

// Look for sep namespace documentation in SepManager.h
namespace sep
{
/*----------------------------------------------------------------------------
 | Constructors / Destructor
 ----------------------------------------------------------------------------*/
  // Keep in mind that this constructor is protected and getInstance() should be used instead.
  TubeHandler::TubeHandler()
  {
    LOG_MACRO(LOG_WARNING, "::TubeHandler() default constructor");
  }

  TubeHandler::~TubeHandler()
  {
    LOG_MACRO(LOG_WARNING, "::~TubeHandler()");
  }
/*----------------------------------------------------------------------------
 | Public function definitions
 ----------------------------------------------------------------------------*/
  void TubeHandler::drawBoundaries(ats_map::GPainter */*painter*/, pm::PropertyMap::Ref /*pmRoot*/)
  {
    LOG_MACRO(LOG_WARNING, "::drawBoundaries()");
  }
} //namespace
```

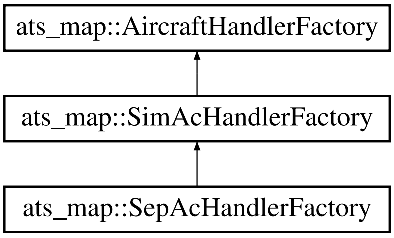
C.5 AircraftHandlerFactory

C.5.1 ats_map::AircraftHandlerFactory Class Reference

Provides a means for creating **AircraftHandler**(p. 175) objects without specific knowledge about the actual object type.

`#include <AircraftHandlerFactory.h>`

Inheritance diagram for ats_map::AircraftHandlerFactory::



Public Types

- typedef boost::shared_ptr< **AircraftHandlerFactory** > **Ptr**

  Shared pointer type for **AircraftHandler**(p. 175) *factories.*

Public Member Functions

- virtual **AircraftHandler::SmartPtr getInstance** (**AircraftHandler::PrimaryKey** pk)=0

  Get a new instance of **AircraftHandler**(p. 175) *wrapped in smart ptr.*

Static Public Member Functions

- static **Ptr getFactory** ()

  Get a factory that will create the correct **AircraftHandler**(p. 175) *type.*

C.5.1.1 Detailed Description

Provides a means for creating **AircraftHandler**(p. 175) objects without specific knowledge about the actual object type.
Abstract class defining the interface for concrete classes that inherit from this one.

C.5.1.2 Member Typedef Documentation

typedef boost::shared_ptr<AircraftHandlerFactory> ats_map::AircraftHandlerFactory::Ptr
Shared pointer type for **AircraftHandler**(p. 175) factories.

C.5.1.3 Member Function Documentation

static Ptr ats_map::AircraftHandlerFactory::getFactory () `[inline, static]`
Get a factory that will create the correct **AircraftHandler**(p. 175) type.
Should not be used, creates null pointer since this is an abstract class.
Reimplemented in **ats_map::SepAcHandlerFactory** (p. 122), and **ats_map::SimAcHandlerFactory** (p. 120).

virtual AircraftHandler::SmartPtr ats_map::AircraftHandlerFactory::getInstance (AircraftHandler::PrimaryKey *pk*) `[pure virtual]`
Get a new instance of **AircraftHandler**(p. 175) wrapped in smart ptr.
Abstract method defining the interface for concrete classes that inherit from this one.
Implemented in **ats_map::SepAcHandlerFactory** (p. 122), and **ats_map::SimAcHandlerFactory** (p. 120).

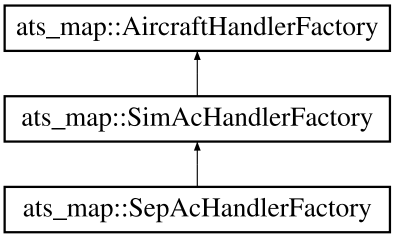
C.5.2 Header File

```
/*! \file AircraftHandlerFactory.h
    \author Olafur Freyr Hjalmsson
    \ingroup atsMap
    \brief Contains AircraftHandlerFactory declaration
*/

#ifndef _AIRCRAFTHANDLER_FACTORY_
#define _AIRCRAFTHANDLER_FACTORY_

#include "pm/PrimaryKey.h"
#include "boost/shared_ptr.hpp"

#include "atsMap/AircraftHandler.h"

namespace ats_map
{
  // Forward declarations
  //class AircraftHandler;
  //class AircraftHandler::SmartPtr;
  //class AircraftHandler::PrimaryKey;

  //! Provides a means for creating AircraftHandler objects without specific knowledge about the actual object type.
  /*! Abstract class defining the interface for concrete classes that inherit from this one. */
  class AircraftHandlerFactory
  {
  public:
    //! Shared pointer type for AircraftHandler factories
    typedef boost::shared_ptr<AircraftHandlerFactory> Ptr;

    //! Get a new instance of AircraftHandler wrapped in smart ptr.
    /*! Abstract method defining the interface for concrete classes that inherit from this one. */
    virtual AircraftHandler::SmartPtr getInstance(AircraftHandler::PrimaryKey pk) = 0;

    //! Get a factory that will create the correct AircraftHandler type.
    /*! Should not be used, creates null pointer since this is an abstract class. */
    static Ptr getFactory() {return Ptr();};
  };
}

#endif

```

C.6 SimAcHandlerFactory

C.6.1 ats_map::SimAcHandlerFactory Class Reference

Provides a means for creating **SimAcHandler**(p. 185) objects without specific knowledge about the actual object type.
`#include <SimAcHandlerFactory.h>`
Inheritance diagram for ats_map::SimAcHandlerFactory::



Public Member Functions

- virtual **AircraftHandler::SmartPtr getInstance** (**AircraftHandler::PrimaryKey** pk)

  Get a new instance of **SimAcHandler**(p. 185) *wrapped in smart pointer.*

Static Public Member Functions

- static **Ptr getFactory** ()

  Get a factory that will create the correct **AircraftHandler**(p. 175) *type (***SimAc-Handler**(p. 185) *in this case).*

C.6.1.1 Detailed Description

Provides a means for creating **SimAcHandler**(p. 185) objects without specific knowledge about the actual object type.

C.6.1.2 Member Function Documentation

static Ptr ats_map::SimAcHandlerFactory::getFactory () `[inline, static]`
Get a factory that will create the correct **AircraftHandler**(p. 175) type (**SimAc-Handler**(p. 185) in this case).
Reimplemented from **ats_map::AircraftHandlerFactory** (p. 118).
Reimplemented in **ats_map::SepAcHandlerFactory** (p. 122).

virtual AircraftHandler::SmartPtr ats_map::SimAcHandlerFactory::getInstance (AircraftHandler::PrimaryKey *pk*) `[inline, virtual]`
Get a new instance of **SimAcHandler**(p. 185) wrapped in smart pointer.
Implements **ats_map::AircraftHandlerFactory** (p. 119).
Reimplemented in **ats_map::SepAcHandlerFactory** (p. 122).

C.6.2 Header File

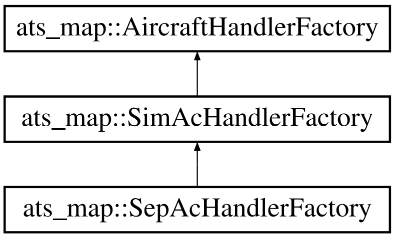
```
/*! \file SimAcHandlerFactory.h
    \author Olafur Freyr Hjalmsson
    \ingroup atsMap
    \brief Contains SimAcHandlerFactory declararion
*/

#ifndef SIM_AC_HANDLER_FACTORY_H
#define SIM_AC_HANDLER_FACTORY_H
```

```

#include "atsMap/AircraftHandlerFactory.h"

namespace ats_map
{
  //! Provides a means for creating SimAcHandler objects without specific knowledge about the actual object type.
  class SimAcHandlerFactory : public AircraftHandlerFactory
  {
  public:
    //! Get a new instance of SimAcHandler wrapped in smart pointer.
    virtual AircraftHandler::SmartPtr getInstance(AircraftHandler::PrimaryKey pk) {return SimAcHandler::create(pk);}

    //! Get a factory that will create the correct AircraftHandler type (SimAcHandler in this case).
    static Ptr getFactory() {return Ptr(new SimAcHandlerFactory());}
  };
}
#endif // SIM_AC_HANDLER_FACTORY_H
```

C.7 SepAcHandlerFactory

C.7.1 ats_map::SepAcHandlerFactory Class Reference

Provides a means for creating **SepAcHandler**(p. 128) objects without specific knowledge about the actual object type.
`#include <SepAcHandlerFactory.h>`
Inheritance diagram for ats_map::SepAcHandlerFactory::



Public Member Functions

- virtual **AircraftHandler::SmartPtr getInstance** (**AircraftHandler::PrimaryKey** pk)

  Get a new instance of **SepAcHandler**(p. 128) *wrapped in smart pointer.*

Static Public Member Functions

- static **Ptr getFactory** ()

  Get a factory that will create the correct **AircraftHandler**(p. 175) *type (***SepAc-Handler**(p. 128) *in this case).*

C.7.1.1 Detailed Description

Provides a means for creating **SepAcHandler**(p. 128) objects without specific knowledge about the actual object type.

C.7.1.2 Member Function Documentation

static Ptr ats_map::SepAcHandlerFactory::getFactory () `[inline, static]`
Get a factory that will create the correct **AircraftHandler**(p. 175) type (**SepAcHandler**(p. 128) in this case).
Reimplemented from **ats_map::SimAcHandlerFactory** (p. 120).

virtual AircraftHandler::SmartPtr ats_map::SepAcHandlerFactory::getInstance (AircraftHandler::PrimaryKey *pk*) `[inline, virtual]`
Get a new instance of **SepAcHandler**(p. 128) wrapped in smart pointer.
Reimplemented from **ats_map::SimAcHandlerFactory** (p. 120).

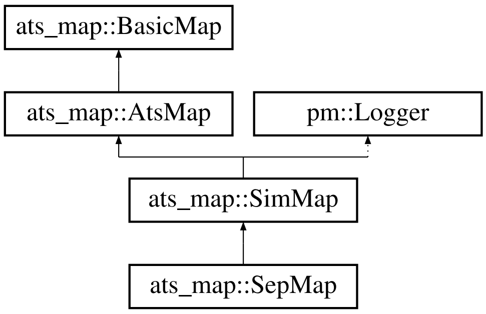
C.7.2 Header File

```
/*! \file SepAcHandlerFactory.h
    \author Olafur Freyr Hjalmsson
    \ingroup atsMap
    \brief Contains SepAcHandlerFactory declararion
*/

#ifndef SEP_AC_HANDLER_FACTORY_H
#define SEP_AC_HANDLER_FACTORY_H

#include "simMap/SimAcHandlerFactory.h"

namespace ats_map
{
  //! Provides a means for creating SepAcHandler objects without specific knowledge about the actual object type.
  class SepAcHandlerFactory : public SimAcHandlerFactory
  {
  public:
    //! Get a new instance of SepAcHandler wrapped in smart pointer.
    virtual AircraftHandler::SmartPtr getInstance(AircraftHandler::PrimaryKey pk) {return SepAcHandler::create(pk);}

    //! Get a factory that will create the correct AircraftHandler type (SepAcHandler in this case).
    static Ptr getFactory() {return Ptr(new SepAcHandlerFactory());}
  };
}
#endif // SEP_AC_HANDLER_FACTORY_H
```

C.8 SepMap

C.8.1 ats_map::SepMap Class Reference

`#include <SepMap.h>`
Inheritance diagram for ats_map::SepMap::



Public Types

- typedef **PM::Obj Obj**

Public Slots

- void **slotUpdateSeparation** ()

  Provides functionality to update the separation of the AircraftHandlers.

- void **slotSepPathsNeedRecalculation** ()

  Ensures that all paths are recalculated.

- void **slotClearAllSeparation** ()

  Clear all separation boundary display.

- void **showSeparation** (int id)

  Toggle "Show separation" point for currently selected aircraft.

Public Member Functions

- **SepMap** (**Obj** pmTree, QWidget ∗parent=0, const char ∗name=0, WFlags f=0, **FactoryPtr** achf=SepAcHandlerFactory::getFactory())

  Construct the **SepMap**(p. 122) *with services, parent object and a name.*

- ∼**SepMap** ()

  Destroy.

- virtual void **post_set** (**pm::PropertyMap** ∗parent, const **pm::Property** &child)

  Overload Logger::post_set() *to perform* **SepMap**(p. 122) *specific stuff.*

Protected Member Functions

- virtual void **createPopup** (**Ref** acRef)

  Create new aircraft popup menu for aircraft acRef.

C.8.1.1 Detailed Description

Provides simulator map with aircraft separation support for amelia. This class inherits from **SimMap**(p. 170) which provides both **AtsMap**(p. 162) and Logger.

C.8.1.2 Member Typedef Documentation

typedef PM::Obj ats_map::SepMap::Obj
Reimplemented from **ats_map::SimMap** (p. 173).

C.8.1.3 Constructor & Destructor Documentation

ats_map::SepMap::SepMap (Obj *pmTree*, QWidget ∗ *parent* = `0`, const char ∗ *name* = `0`, WFlags *f* = `0`, FactoryPtr *achf* = `SepAcHandlerFactory::getFactory()`)
Construct the **SepMap**(p. 122) with services, parent object and a name.

ats_map::SepMap::∼SepMap ()
Destroy.

C.8.1.4 Member Function Documentation

void ats_map::SepMap::createPopup (Ref *acRef*) `[protected, virtual]`
Create new aircraft popup menu for aircraft *acRef*.
Reimplemented from **ats_map::SimMap** (p. 173).

void ats_map::SepMap::post_set (pm::PropertyMap ∗ *parent*, const pm::Property & *child*) `[virtual]`
Overload *Logger::post_set()* to perform **SepMap**(p. 122) specific stuff.
Reimplemented from **ats_map::SimMap** (p. 174).

void ats_map::SepMap::showSeparation (int *id*) `[slot]`
Toggle "Show separation" point for currently selected aircraft.

void ats_map::SepMap::slotClearAllSeparation () `[slot]`
Clear all separation boundary display.

void ats_map::SepMap::slotSepPathsNeedRecalculation () `[slot]`
Ensures that all paths are recalculated.

void ats_map::SepMap::slotUpdateSeparation () `[slot]`
Provides functionality to update the separation of the AircraftHandlers.
These are maintained in the local vector **AtsMap::_aircrafts**(p. 170). It is connected to the signal preUpdateAtsMap, and therefore, when the **AtsMap**(p. 162) is updated all the aircraft-Handlers are up to date.

C.8.2 Header File

```
/*! \file SepMap.h
    \author Olafur Freyr Hjalmsson
    \ingroup atsMap
    \brief Contains SepMap class declaration
*/

/*---------------------------------------------------------------------------
| Copyright (c) 2003, Tern Systems, Inc.  All Rights Reserved.
|
| $Revision: 1.0 $
| $Date: 2004/12/13 11:46:00 $
---------------------------------------------------------------------------*/
#ifndef _SEPMAP_H
#define _SEPMAP_H

/*---------------------------------------------------------------------------
| Include local header files
---------------------------------------------------------------------------*/
#include "SepAcHandler.h"
#include "sepMap/SepAcHandlerFactory.h"
#include "simMap/SimDistReporting.h"
#include "simMap/SimMap.h"
#include "pm/PropertyMap.h"
#include "pm/Logger.h"
#include "pm/PmTree.h"
#include "simGeo/Position.h"
#include "logmacro.h"
#include "commonWidgets/AircraftPopup.h"

#include "sep/SepManager.h"
#include "sep/TubePhase.h"
#include "sep/Tube.h"
#include "sepMap/TubeHandler.h"

/*---------------------------------------------------------------------------
| Include standard header files
---------------------------------------------------------------------------*/
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include <qpixmap.h>
#include <qpopupmenu.h>
#include <qevent.h>
#include <qdom.h>
#include <cstdlib>
#include <map>

namespace ats_map
{
  /*! Provides simulator map with aircraft separation support for amelia.
   *  This class inherits from SimMap which provides both AtsMap and Logger.
   */
  class SepMap: public SimMap
  {
    Q_OBJECT
    // Allow a test class to access the internals.
    //friend class TestSepMap;

  public:
    typedef PM::Obj Obj;

    //! Construct the SepMap with services, parent object and a name.
    SepMap(Obj pmTree, QWidget *parent = 0, const char *name = 0, WFlags f=0, FactoryPtr
achf = SepAcHandlerFactory::getFactory() );

    //! Destroy.
    ~SepMap();

    //! Overload \a Logger::post_set() to perform SepMap specific stuff.
    virtual void post_set(pm::PropertyMap *parent, const pm::Property &child);

```

```
  public slots:
    //! Provides functionality to update the separation of the AircraftHandlers.
    /*! These are maintained in the local vector AtsMap::_aircrafts. It is connected to the
     * signal preUpdateAtsMap, and therefore, when the AtsMap is updated all the
     * aircraftHandlers are up to date.
     */
    void slotUpdateSeparation();

    //! Ensures that all paths are recalculated
    void slotSepPathsNeedRecalculation();

    //! Clear all separation boundary display
    void slotClearAllSeparation();

    //! Toggle "Show separation" point for currently selected aircraft.
    void showSeparation(int id);
  protected:
    //popup
    //! Create new aircraft popup menu for aircraft \a acRef.
    virtual void createPopup(Ref acRef);
  };
}
#endif /* _SEPMAP_H */
```

C.8.3 Source File

```
/*! \file SepMap.cpp
 */

/*----------------------------------------------------------------------------
| Copyright (c) 2003, Tern Systems, Inc.  All Rights Reserved.
|
| $Revision: 1.0 $
| $Date: 2004/11/08 10:29:39 $
----------------------------------------------------------------------------*/

/*----------------------------------------------------------------------------
| Declaration of this class
----------------------------------------------------------------------------*/
#include "sepMap/SepMap.h"

/*----------------------------------------------------------------------------
| Include local header files
----------------------------------------------------------------------------*/
#include "sep/SepManager.h"
#include "sep/TubePhase.h"
#include "sep/Tube.h"
#include "sepMap/TubeHandler.h"

#include "world/ClientUtils.h"
#include "world/Aircraft.h"
#include "world/SimDatabase.h"
#include "world/ClientUtils.h"
#include "simBasicTypes/Speed.h"
#include "simBasicTypes/Distance.h"
#include "simGeo/Heading.h"
#include "pm/PrimaryKey.h"
#include "pm/PmUtils.h"
#include "commonWidgets/HeadingDialog.h"
#include "commonWidgets/LevelDialog.h"
#include "commonWidgets/DistanceMeasuring.h"
#include "commonWidgets/Direct2FixDialog.h"
#include "commonWidgets/SpeedDialog.h"
#include "commonWidgets/TransponderDialog.h"
#include "commonWidgets/Timeslider.h"
#include "commonWidgets/AircraftInfo.h"
#include "commonWidgets/HandoverDialog.h"

/*----------------------------------------------------------------------------
| Include standard header files
----------------------------------------------------------------------------*/
```

```
#include <qbitmap.h>
#include <qcursor.h>
#include <qfile.h>
#include <qregexp.h>
#include <qdatetime.h>
#include <qtimer.h>

namespace ats_map
{
  SepMap::SepMap(Obj pmTree, QWidget *parent, const char *name, WFlags f, FactoryPtr achf ):
      SimMap(pmTree, parent, name, f, achf)
  {
    if(_achFactory == FactoryPtr()) {
      LOG_MACRO(LOG_WARNING, "Creating SepAcHandlerFactory, since _achFactory is empty.");
      _achFactory = SepAcHandlerFactory::getFactory();
    }

    setCaption("SepMap");

    //Menu initialization
    _menuPopup->insertItem("Hide all separation", this, SLOT(slotClearAllSeparation()));

    connect(this, SIGNAL(preUpdateBasicMap()), this, SLOT(slotSepPathsNeedRecalculation()));
    connect(this, SIGNAL(midUpdateAtsMap()), this, SLOT(slotUpdateSeparation()));
    // Get SepManager ready for action:
    //sep::SepManager::getInstance()->initialize();

/* TODO: Need adjustment, taken from SimMap constructor:

    connect(this, SIGNAL(preUpdateBasicMap()), this, SLOT(slotPathsNeedRecalculation()));
    connect(this, SIGNAL(preUpdateAtsMap()), this, SLOT(slotUpdateAircrafts()));
    connect(this, SIGNAL(postUpdateAtsMap()), this, SLOT(slotRepaint()));
*/

  }
  SepMap::~SepMap()
  {}

  /*----------------------------------------------------------------------------
    | Protected function definitions
    ----------------------------------------------------------------------------*/
  void SepMap::slotUpdateSeparation()
  {
    LOG_MACRO(LOG_DEBUG, "About to start updating aircraft separation representation");

    //first draw path and plan so the aircrafts will be painted on top of them
    for(Aircrafts::iterator it = _aircrafts.begin(); it != _aircrafts.end(); ++it){
      ((SepAcHandler&)(**it)).drawFlightSep(_painter, getPMRoot());
    }
    LOG_MACRO(LOG_DEBUG, "Finished updating aircraft separation representation");
  }

  void SepMap::slotSepPathsNeedRecalculation()
  {
    LOG_MACRO(LOG_DEBUG, "Recalculating aircraft separation paths");
    for(Aircrafts::iterator it = _aircrafts.begin(); it != _aircrafts.end(); ++it)
      ((SepAcHandler&)(**it)).setSepNeedsCalculation(true);
  }

  void SepMap::slotClearAllSeparation()
  {
    LOG_MACRO(LOG_WARNING, "Clearing all separation boundaries ");
    for(Aircrafts::iterator it = _aircrafts.begin(); it != _aircrafts.end(); ++it)
      ((SepAcHandler&)(**it)).drawSep(false);
    slotUpdateAtsMap();
  }

  void SepMap::showSeparation(int id)
  {
    Ref acRef = world::findSelectedAircraft(getPMRoot());
    if (_acMenu->isItemChecked(id)) {
      acRef->set("DRAW_SEPARATION","0");
```

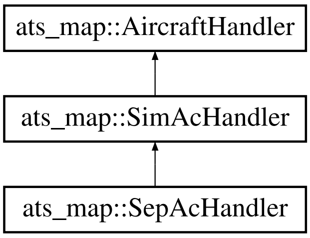
```
      LOG_MACRO(LOG_DEBUG, "Disabling separation boundaries for flight "<<acRef->name());
    } else {
      acRef->set("DRAW_SEPARATION","1");
      LOG_MACRO(LOG_DEBUG, "Enabling separation boundaries for flight "<<acRef->name());
    }
  }

  void SepMap::post_set(pm::PropertyMap *node, const pm::Property &property)
  {
    this->SimMap::post_set(node, property);
    if(node->type()=="Aircraft")
    {
      LOG_MACRO(LOG_DEBUG, "SepMap handling of Node "<<node->type()<<"::"<<node->name()<<" set "<<property.name<<"="<<property.value);
      if( property.name == "COMMAND" || property.name == "FPL" || property.name == "SQUAWK_TIME")
      {
        Aircrafts::iterator it = std::find_if(_aircrafts.begin(), _aircrafts.end(), DerefCompObj(node->name()));
        assert(it!=_aircrafts.end());
        ((SepAcHandler&)(**it)).setSepNeedsCalculation(true);
      }
      else if( property.name == "DRAW_SEPARATION")
      {
        QTimer::singleShot(1, this, SLOT(slotUpdateAtsMap()));
        createPopup(pm::PropertyMap::getRef(node));
      }
    }
  }

  /*---------------------------------------------------------------------------
    | Private function definitions
    ---------------------------------------------------------------------------*/
  void SepMap::createPopup(Ref acRef)
  {
    this->SimMap::createPopup(acRef);
    // Add to menu position 3
    int menuId = _acMenu->insertItem("Show separation", -1, 2);
    _acMenu->connectItem(menuId, this, SLOT(showSeparation(int)));
    _acMenu->setItemParameter(menuId,menuId);
    if (acRef->get("DRAW_SEPARATION")=="1") {
      //  showFPLaction->setOn(true);
      _acMenu->setItemChecked(menuId,true);
    } else {
      //      showFPLaction->setOn(false);
      _acMenu->setItemChecked(menuId,false);
    }
    // Remove pseudopilot options "Level", "Heading", "Speed" and "Direct to fix"
    for(int i = _acMenu->count(); i >= 0; i--) {
      QString text = _acMenu->text(_acMenu->idAt(i));
      if(!text.isEmpty()) {
        if(text.contains("Level") || text.contains("Heading")
         ||text.contains("Speed") || text.contains("Direct") ) {
          _acMenu->removeItemAt(i);
        }
      }
    }
    // Add to the bottom
    _acMenu->insertItem("Hide all separation", this, SLOT(slotClearAllSeparation()));
  }
} //namespace
```

C.9 SepAcHandler

C.9.1 ats_map::SepAcHandler Class Reference

`#include <SepAcHandler.h>`

Inheritance diagram for ats_map::SepAcHandler::



Public Types

- enum **SepType** {
  SEP_PROC = 0, **SEP_ADS**, **SEP_RADAR_NID**, **SEP_RADAR_ID**,
  NR_SEP_TYPES }

  SepType *is a numeric representation of the distinct types of separation rules.*

Public Member Functions

- **SepAcHandler** (**PrimaryKey** pk)

  Construct Separation aware Aircraft Handler based on pk.

- virtual void **drawSep** (bool draw)

  Indicate to this aircraft that it should draw *separation boundaries.*

- virtual bool **drawSep** () const

  Find out if this aircraft should draw separation boundaries.

- virtual void **drawAircraft** (GPainter ∗painter)

  Draw aircraft symbol on painter.

- virtual void **drawFlightSep** (GPainter ∗painter, **Ref** pmRoot)

  Draw aircraft separation boundaries on painter.

- void **calculateSep** (GPainter ∗painter, **Ref** pmRoot)

  Calculate separation boundaries.

- void **setSepNeedsCalculation** (bool s)

  Indicate to this aircraft that it should invalidate current separation boundary coordinates.

- bool **sepNeedsCalculation** () const

  Find out if this aircraft needs to calculate separation boundary coordinates.

- **SepType getSepType** ()

  Get this aircrafts current separation type.

- **SepType getSepType** (sim2::Position pos, **SepType** fallbackSep=NR_SEP_TYPES)

  Get this aircrafts separation type if it were at position pos *with separation type* fallback.

- **SepType getSepTypeDefault** ()

  Get this aircrafts default separation type.

- bool **withinRadar** ()

  Find out if this aircraft is currenly within radar range.

- bool **radarIdentified** ()

  Find out if this aircraft has been radar identified.

Static Public Member Functions

- static **SmartPtr create** (**PrimaryKey** pk)

  Create smart pointer wrapped Separation aware Aircraft Handler based on pk.

- static QPoint **pos2xy** (GPainter ∗painter, sim2::Position pos)

  Convert to appropriate coordinates on painter.

- static sim2::Position **xy2pos** (GPainter ∗painter, QPoint point)

  Convert on painter *to geological position.*

- static int **dist2Pixel** (GPainter ∗painter, sim2::Distance dist)

  Convert dist *to appropriate number of pixels on* painter.

- static **SepType getSepType** (**Ref** ac, sim2::Position pos, **SepType** fallbackSep=NR_-SEP_TYPES)

  Get the separation type of ac *if it were at position* pos *with separation type* fallback.

- static **SepType getSepTypeDefault** (**Ref** ac)

  Get the default separation type of ac.

- static bool **withinRadar** (sim2::Position pos)

  Find out if pos *is within range of any radar.*

- static bool **radarIdentified** (**Ref** ac)

  Find out if ac *has been radar identified.*

- static sim2::Heading **calcHeading** (sim2::Position from, sim2::Position to)

  Calculate the heading from from *to* to.

- static double **calcAngle** (sim2::Position prev, sim2::Position cur, sim2::Position next)

  Calculate the angle at cur *between* prev *and* next.

- static double **calcAngleXY** (const QPoint &p1, const QPoint &p2)

  Calculate the angle between the vector in the cartesian plane from p1 *to* p2 *and the x-axis.*

- static double **calcAngleDiff** (const double &angle1, const double &angle2)

  Calculate the difference in angles between angle1 *and* angle2.

Protected Attributes

- bool **m_sepNeedsCalculation**

  Flag to indicate that separation boundaries need (re)calculation.

- std::vector< **SepPoint** > **m_separation**

  List of separation boundary points, synchronized with _path.

- std::vector< **SepPoint** > **m_separationEst**

  List of estimated separation boundary points.

- **SepType m_sepType**

  This aircraft current separation type.

- double **m_sepValues** [NR_SEP_TYPES]

  Separation values corresponding with each SepType.

- int **m_radarRefresh**

  How many cycles until next refresh.

- bool **m_withinRadar**

  Result from last radar range measurement.

- bool **m_radarIdent**

  Has this aircraft been radar identified.

- **PixmapPtr m_acPixmaps** [NR_SEP_TYPES]

  Pixmaps for use as aircraft symbol for each separation type.

Static Protected Attributes

- static const int **RADAR_REFRESH** = 5

  Update every 5 refresh cycles (each 1 sec.).

Classes

- struct **SepPoint**

  Stores single points along the projected flight path containing the left and right side separation boundary points along with the cutover angle of the flight path.

C.9.1.1 Detailed Description

Provides **SepMap**(p. 122) specific drawing functionalities The class extends **SimAcHandler**(p. 185) with **SepMap**(p. 122) specific drawing functionality, namely the ability to display correcty symbols and draw separation boundaries.

C.9.1.2 Member Enumeration Documentation

enum ats_map::SepAcHandler::SepType
SepType is a numeric representation of the distinct types of separation rules.

Enumeration values:

SEP_PROC Procedural separation, 60 nm, i.e. default.

SEP_ADS ADS separation, 30 nm.

SEP_RADAR_NID Not identified RADAR separation, undefined (0 nm).

SEP_RADAR_ID Identified RADAR separation, 10 nm.

NR_SEP_TYPES Total number of sep types, used when creating arrays and iterating through them.

C.9.1.3 Constructor & Destructor Documentation

ats_map::SepAcHandler::SepAcHandler (PrimaryKey *pk*)
Construct Separation aware Aircraft Handler based on *pk*.

C.9.1.4 Member Function Documentation

double ats_map::SepAcHandler::calcAngle (sim2::Position *prev*, sim2::Position *cur*, sim2::Position *next*) `[static]`
Calculate the angle at *cur* between *prev* and *next*.

double ats_map::SepAcHandler::calcAngleDiff (const double & *angle1*, const double & *angle2*) `[static]`
Calculate the difference in angles between *angle1* and *angle2*.

double ats_map::SepAcHandler::calcAngleXY (const QPoint & *p1*, const QPoint & *p2*) `[static]`
Calculate the angle between the vector in the cartesian plane from *p1* to *p2* and the x-axis.

sim2::Heading ats_map::SepAcHandler::calcHeading (sim2::Position *from*, sim2::Position *to*) `[inline, static]`
Calculate the heading from *from* to *to*.

void ats_map::SepAcHandler::calculateSep (GPainter ∗ *painter*, Ref *pmRoot*)
Calculate separation boundaries.
Path is a prerequisite and the information is used to calculate separation. Separation boundaries are different for confirmed and estimated when within radar.

static SmartPtr ats_map::SepAcHandler::create (PrimaryKey *pk*) `[inline, static]`

Create smart pointer wrapped Separation aware Aircraft Handler based on *pk*.
Reimplemented from **ats_map::SimAcHandler** (p. 189).

int ats_map::SepAcHandler::dist2Pixel (GPainter ∗ *painter*, sim2::Distance *dist*) `[inline, static]`
Convert *dist* to appropriate number of pixels on *painter*.

void ats_map::SepAcHandler::drawAircraft (GPainter ∗ *painter*) `[virtual]`
Draw aircraft symbol on *painter*.
Reimplemented from **ats_map::SimAcHandler** (p. 189).

void ats_map::SepAcHandler::drawFlightSep (GPainter ∗ *painter*, Ref *pmRoot*) `[virtual]`
Draw aircraft separation boundaries on *painter*.

virtual bool ats_map::SepAcHandler::drawSep () const `[inline, virtual]`
Find out if this aircraft should draw separation boundaries.

virtual void ats_map::SepAcHandler::drawSep (bool *draw*) `[inline, virtual]`
Indicate to this aircraft that it should *draw* separation boundaries.

SepAcHandler::SepType ats_map::SepAcHandler::getSepType (Ref *ac*, sim2::Position *pos*, SepType *fallbackSep* = `NR_SEP_TYPES`**)** `[inline, static]`
Get the separation type of *ac* if it were at position *pos* with separation type *fallback*.

SepAcHandler::SepType ats_map::SepAcHandler::getSepType (sim2::Position *pos*, SepType *fallbackSep* = `NR_SEP_TYPES`**)** `[inline]`
Get this aircrafts separation type if it were at position *pos* with separation type *fallback*.

SepAcHandler::SepType ats_map::SepAcHandler::getSepType () `[inline]`
Get this aircrafts current separation type.

SepAcHandler::SepType ats_map::SepAcHandler::getSepTypeDefault (Ref *ac*) `[inline, static]`
Get the default separation type of *ac*.

SepAcHandler::SepType ats_map::SepAcHandler::getSepTypeDefault () `[inline]`
Get this aircrafts default separation type.

QPoint ats_map::SepAcHandler::pos2xy (GPainter * *painter*, sim2::Position *pos*) `[inline, static]`
Convert to appropriate coordinates on *painter*.

bool ats_map::SepAcHandler::radarIdentified (Ref *ac*) `[inline, static]`
Find out if *ac* has been radar identified.

bool ats_map::SepAcHandler::radarIdentified () `[inline]`
Find out if this aircraft has been radar identified.

bool ats_map::SepAcHandler::sepNeedsCalculation () const `[inline]`
Find out if this aircraft needs to calculate separation boundary coordinates.

void ats_map::SepAcHandler::setSepNeedsCalculation (bool *s*) `[inline]`
Indicate to this aircraft that it should invalidate current separation boundary coordinates.

bool ats_map::SepAcHandler::withinRadar (sim2::Position *pos*) `[static]`
Find out if *pos* is within range of any radar.

bool ats_map::SepAcHandler::withinRadar ()
Find out if this aircraft is currenly within radar range.

sim2::Position ats_map::SepAcHandler::xy2pos (GPainter * *painter*, QPoint *point*) `[inline, static]`
Convert on *painter* to geological position.

C.9.1.5 Member Data Documentation

PixmapPtr ats_map::SepAcHandler::m_acPixmaps[NR_SEP_TYPES] `[protected]`

Pixmaps for use as aircraft symbol for each separation type.

bool ats_map::SepAcHandler::m_radarIdent `[protected]`
Has this aircraft been radar identified.

int ats_map::SepAcHandler::m_radarRefresh `[protected]`
How many cycles until next refresh.

std::vector<SepPoint> ats_map::SepAcHandler::m_separation `[protected]`
List of separation boundary points, synchronized with *_path*.

std::vector<SepPoint> ats_map::SepAcHandler::m_separationEst `[protected]`
List of estimated separation boundary points.

bool ats_map::SepAcHandler::m_sepNeedsCalculation `[protected]`
Flag to indicate that separation boundaries need (re)calculation.

SepType ats_map::SepAcHandler::m_sepType `[protected]`
This aircraft current separation type.

double ats_map::SepAcHandler::m_sepValues[NR_SEP_TYPES] `[protected]`
Separation values corresponding with each SepType.

bool ats_map::SepAcHandler::m_withinRadar `[protected]`
Result from last radar range measurement.

const int ats_map::SepAcHandler::RADAR_REFRESH = 5 `[static, protected]`
Update every 5 refresh cycles (each 1 sec.).

C.9.2 ats_map::SepAcHandler::SepPoint Struct Reference

Stores single points along the projected flight path containing the left and right side separation boundary points along with the cutover angle of the flight path.
`#include <SepAcHandler.h>`

Public Member Functions

- **SepPoint** ()

  Create empty separation boundary point.

- **SepPoint** (QPoint l, QPoint r, double a=0.0, **SepType** s=SEP_PROC, sim2::UtcTime t=sim2::UtcTime())

  Create separation boundary point with supplied or default values.

Public Attributes

- QPoint **left**

  Left side separation boundary point.

- QPoint **right**

  Right side separation boundary point.

- double **angle**

Angle of turn, positive when turning right, negative when turning left.

- **SepType sep**

  Separation type.

- sim2::UtcTime **time**

  Time index of this point.

C.9.2.1 Detailed Description

Stores single points along the projected flight path containing the left and right side separation boundary points along with the cutover angle of the flight path.

C.9.2.2 Constructor & Destructor Documentation

ats_map::SepAcHandler::SepPoint::SepPoint () `[inline]`
Create empty separation boundary point.

ats_map::SepAcHandler::SepPoint::SepPoint (QPoint *l*, QPoint *r*, double *a* = `0.0`**, SepType *s* =** `SEP_PROC`**, sim2::UtcTime *t* =** `sim2::UtcTime()`**)** `[inline]`
Create separation boundary point with supplied or default values.

C.9.2.3 Member Data Documentation

double ats_map::SepAcHandler::SepPoint::angle
Angle of turn, positive when turning right, negative when turning left.

QPoint ats_map::SepAcHandler::SepPoint::left
Left side separation boundary point.

QPoint ats_map::SepAcHandler::SepPoint::right
Right side separation boundary point.

SepType ats_map::SepAcHandler::SepPoint::sep
Separation type.

sim2::UtcTime ats_map::SepAcHandler::SepPoint::time
Time index of this point.

C.9.3 Header File

```
/*! \file SepAcHandler.h
    \author Olafur Freyr Hjalmsson
    \ingroup atsMap
    \brief Contains SepAcHandler declararion
*/

#ifndef SEP_AC_HANDLER_H
```

```
#define SEP_AC_HANDLER_H

#include "simMap/SimAcHandler.h"
#include "simBasicTypes/UtcTime.h"
#include "simGeo/Heading.h"
#include "simGeo/Position.h"
#include <vector>
#include <cmath>

namespace ats_map
{
  /*! Provides SepMap specific drawing functionalities
   * The class extends SimAcHandler with SepMap specific drawing functionality,
   * namely the ability to display correcty symbols and draw separation boundaries.
   */
  class SepAcHandler : public SimAcHandler
  {
  public:

    //! \a SepType is a numeric representation of the distinct types of separation rules.
    typedef enum {
      SEP_PROC = 0,  /*!< Procedural separation, 60 nm, i.e.\ default. */
      SEP_ADS,       /*!< ADS separation, 30 nm. */
      SEP_RADAR_NID, /*!< Not identified RADAR separation, undefined (0 nm). */
      SEP_RADAR_ID,  /*!< Identified RADAR separation, 10 nm. */
      NR_SEP_TYPES   /*!< Total number of sep types, used when creating arrays and iterating through them. */
    } SepType;

    //! Construct Separation aware Aircraft Handler based on \a pk.
    SepAcHandler(PrimaryKey pk);
    //! Create smart pointer wrapped Separation aware Aircraft Handler based on \a pk.
    static SmartPtr create(PrimaryKey pk){return SmartPtrCreate(new SepAcHandler(pk));}

    //! Indicate to this aircraft that it should \a draw separation boundaries.
    virtual void drawSep(bool draw){getRef()->set("DRAW_SEPARATION", ((draw) ?"1" : "0") );}
    //! Find out if this aircraft should draw separation boundaries.
    virtual bool drawSep() const {return getRef()->get("DRAW_SEPARATION")=="1";}
    //! Draw aircraft symbol on \a painter
    virtual void drawAircraft(GPainter *painter);
    //! Draw aircraft separation boundaries on \a painter
    virtual void drawFlightSep(GPainter *painter, Ref pmRoot);

    //! Calculate separation boundaries
    /*! Path is a prerequisite and the information is used to calculate separation.
     *  Separation boundaries are different for confirmed and estimated when within radar.
     */
    void calculateSep(GPainter *painter, Ref pmRoot);
    //! Indicate to this aircraft that it should invalidate current separation boundary coordinates.
    inline void setSepNeedsCalculation(bool s){m_sepNeedsCalculation=s;}
    //! Find out if this aircraft needs to calculate separation boundary coordinates.
    inline bool sepNeedsCalculation() const {return m_sepNeedsCalculation;}
    //! Get this aircrafts current separation type
    inline SepType getSepType();
    //! Get this aircrafts separation type if it were at position \a pos with separation type \a fallback
    inline SepType getSepType(sim2::Position pos, SepType fallbackSep=NR_SEP_TYPES);
    //! Get this aircrafts default separation type
    inline SepType getSepTypeDefault();
    //! Find out if this aircraft is currenly within radar range
    bool withinRadar();
    //! Find out if this aircraft has been radar identified
    inline bool radarIdentified();

    /* ------------- Static helper functions ------------- */
    //! Convert \pos to appropriate coordinates on \a painter
    static inline QPoint  pos2xy(GPainter *painter, sim2::Position pos);
    //! Convert \point on \a painter to geological position
    static inline sim2::Position xy2pos(GPainter *painter, QPoint point);
    //! Convert \a dist to appropriate number of pixels on \a painter
    static inline int     dist2Pixel(GPainter *painter, sim2::Distance dist);
```

```
    //! Get the separation type of \a ac if it were at position \a pos with separation type \a fallback
    static inline SepType getSepType(Ref ac, sim2::Position pos, SepType fallbackSep= NR_SEP_TYPES);
    //! Get the default separation type of \a ac
    static inline SepType getSepTypeDefault(Ref ac);
    //! Find out if \a pos is within range of any radar
    static        bool    withinRadar(sim2::Position pos);
    //! Find out if \a ac has been radar identified
    static inline bool    radarIdentified(Ref ac);
    //! Calculate the heading from \a from to \a to.
    static inline sim2::Heading calcHeading(sim2::Position from, sim2::Position to);
    //! Calculate the angle at \a cur between \a prev and \a next.
    static        double  calcAngle(sim2::Position prev, sim2::Position cur, sim2::Position next);
    //! Calculate the angle between the vector in the cartesian plane from \a p1 to \a p2 and the x-axis.
    static        double  calcAngleXY(const QPoint &p1, const QPoint &p2);
    //! Calculate the difference in angles between \a angle1 and \a angle2.
    static        double  calcAngleDiff(const double &angle1, const double &angle2);

  protected:
    bool m_sepNeedsCalculation;            //!< Flag to indicate that separation boundaries need (re)calculation.
    /*!
     * \brief Stores single points along the projected flight path containing the left and right side
     *        separation boundary points along with the cutover angle of the flight path.
     */
    struct SepPoint{
      //! Create empty separation boundary point
      SepPoint() {}
      //! Create separation boundary point with supplied or default values
      SepPoint(QPoint l, QPoint r, double a=0.0, SepType s=SEP_PROC, sim2::UtcTime t=sim2::UtcTime()):
        left(l),right(r),angle(a),sep(s),time(t) {}
      QPoint left;        //!< Left side separation boundary point
      QPoint right;       //!< Right side separation boundary point
      double angle;       //!< Angle of turn, positive when turning right, negative when turning left.
      SepType sep;        //!< Separation type
      sim2::UtcTime time; //!< Time index of this point
    };
    std::vector<SepPoint> m_separation;    //!< List of separation boundary points, synchronized with \a _path
    std::vector<SepPoint> m_separationEst; //!< List of estimated separation boundary points

    SepType m_sepType;                     //!< This aircraft current separation type
    double m_sepValues[NR_SEP_TYPES];      //!< Separation values corresponding with each SepType

    // Update radar range info
    int  m_radarRefresh;                   //!< How many cycles until next refresh
    static const int RADAR_REFRESH = 5;    //!< Update every 5 refresh cycles (each 1 sec.)
    bool m_withinRadar;                    //!< Result from last radar range measurement
    bool m_radarIdent;                     //!< Has this aircraft been radar identified

    PixmapPtr m_acPixmaps[NR_SEP_TYPES];   //!< Pixmaps for use as aircraft symbol for each separation type
  };
}
#endif // SEP_AC_HANDLER_H
```

C.9.4 Source File

```
/*! \file SepAcHandler.cpp
 */

/*---------------------------------------------------------------------------
| Copyright (c) 2003, Tern Systems, Inc.  All Rights Reserved.
|
```

```
| $Revision: 1.0 $
| $Date: 2004/11/08 10:29:39 $
---------------------------------------------------------------------------*/

#include "sepMap/SepAcHandler.h"

#include "logmacro.h"
#include "world/ClientUtils.h"
#include "world/Aircraft.h"
#include "world/Scenario.h"
#include "simBasicTypes/Speed.h"
#include "simBasicTypes/Distance.h"
#include "simGeo/Heading.h"
#include "simGeo/Position.h"
#include "pm/PmTree.h"
#include "pm/PmUtils.h"
#include "LM_Const.h"

#include "Icons/ProceduralAircraft"
#include "Icons/RadarNoIdentAircraft"
#include "Icons/RadarIdentAircraft"
#include "Icons/ADSAircraft"

#include <qdatetime.h>
#include <cmath>

namespace ats_map
{
  SepAcHandler::SepAcHandler(PrimaryKey pk):
    SimAcHandler(pk),
    m_sepNeedsCalculation(true),
    m_radarRefresh(0),
    m_radarIdent(false)
  {
    LOG_MACRO(LOG_DEBUG, "Creating SepAircraftHandler for flight " << getPrimaryKey()->name());

    m_sepValues[SEP_PROC]      = 60.0; // 60 nautical mile separation minimum
    m_sepValues[SEP_ADS]       = 30.0; // 30 nm separation minimum
    m_sepValues[SEP_RADAR_NID] = 0.0;  // Initialize to some value
    m_sepValues[SEP_RADAR_ID]  = 10.0; // 10 nm separation minimum

    m_radarIdent = radarIdentified();  // Although m_radarIdent is updated, we do it again here for clarity.
    m_sepType = getSepType();          // Also updates m_withinRadar
    m_sepValues[SEP_RADAR_NID] = m_sepValues[getSepTypeDefault()]; // When aircraft is not identified, use default SepType (PROC or ADS).

    m_acPixmaps[SEP_PROC]      = bitmap2pixmap(ProceduralAircraft_width, ProceduralAircraft_height,   (const uchar*)ProceduralAircraft_bits);
    m_acPixmaps[SEP_RADAR_NID] = bitmap2pixmap(RadarNoIdentAircraft_width, RadarNoIdentAircraft_height, (const uchar*)RadarNoIdentAircraft_bits);
    m_acPixmaps[SEP_RADAR_ID]  = bitmap2pixmap(RadarIdentAircraft_width, RadarIdentAircraft_height,   (const uchar*)RadarIdentAircraft_bits);
    m_acPixmaps[SEP_ADS]       = bitmap2pixmap(ADSAircraft_width, ADSAircraft_height,          (const uchar*)ADSAircraft_bits);
    _airf = m_acPixmaps[SEP_PROC];
  }

  void SepAcHandler::drawAircraft(GPainter *painter)
  {
    if( !isVisible() ) {
      LOG_MACRO(LOG_DEBUG, "Not drawing SepAircraftHandler for invisible flight " << getPrimaryKey()->name());
      return;
    }
    LOG_MACRO(LOG_DEBUG, "Drawing SepAircraftHandler for flight " << getPrimaryKey()->name());
    m_sepType = getSepType();
    _airf = m_acPixmaps[m_sepType];

    this->SimAcHandler::drawAircraft(painter);
```

```
    LOG_MACRO(LOG_DEBUG, "Finished drawing SepAircraftHandler for flight " << getPrimaryKey()->name());
  }


  void SepAcHandler::drawFlightSep(GPainter *painter, Ref pmRoot)
  {
    if( drawSep() ) {
      LOG_MACRO(LOG_DEBUG, getRef()->name()<<"::DRAW_SEPARATION="<<getRef()->get("DRAW_SEPARATION"));

      if( sepNeedsCalculation() )
        calculateSep(painter, pmRoot);

      LOG_MACRO(LOG_DEBUG, "Drawing separation path using "<<m_separation.size()<<" points");
      PM::Ref scenario = world::findActiveScenario(pmRoot);
      world::Scenario* activeScenario = PM::narrow<world::Scenario>(scenario);
      sim2::UtcTime oneMinuteAgo = activeScenario->getCurrentTime()-sim2::Duration(sim2::Duration::OneMinute);

      // Mostly green color, but not quite.
      QColor penColor(0x00,0xb0,0x00);
      // First paint the known course
      painter->setPen(QPen(penColor, 1));
      //TODO: Check if all the points are going to be drawn onto canvas
      if(!m_separation.empty() ) {
        for(std::vector<SepPoint>::iterator it = m_separation.begin(); it+1 != m_separation.end(); ++it){
          if((it+1)->time >= oneMinuteAgo) {
            // Temporary drawing of the tube
            painter->drawXYLine(it->left.x() , it->left.y() , (it+1)->left.x() , (it+1)->left.y());
            painter->drawXYLine(it->right.x(), it->right.y(), (it+1)->right.x(), (it+1)->right.y());
          }
        }
      }

      LOG_MACRO(LOG_DEBUG, "Drawing estimated separation path using "<<m_separationEst.size()<<" points");
      // Then paint an estimated course
      painter->setPen(QPen(penColor, 1, Qt::DotLine));
      //TODO: Check if all the points are going to be drawn onto canvas
      if(!m_separationEst.empty() ) {
        for(std::vector<SepPoint>::iterator it = m_separationEst.begin(); it+1 != m_separationEst.end(); ++it){
          if((it+1)->time >= oneMinuteAgo) {
            // Temporary drawing of the tube
            painter->drawXYLine(it->left.x() , it->left.y() , (it+1)->left.x() , (it+1)->left.y());
            painter->drawXYLine(it->right.x(), it->right.y(), (it+1)->right.x(), (it+1)->right.y());
          }
        }
      }
      LOG_MACRO(LOG_DEBUG, "Done drawing separation boundaries for " << getRef()->name());
    }
  }

  // Calculate Path as a prerequisite and use the information to calculate separation,
  // keep time, use trig. functions
  // Draw confirmed and expected separation boundaries within radar
  //TODO: Check if all the points are going to be drawn onto canvas
  void SepAcHandler::calculateSep(GPainter *painter, Ref pmRoot)
  {
    LOG_MACRO(LOG_WARNING, "Welcome to calculateSep7()");
    if( pathNeedsCalculation() )
      calculatePath(painter, pmRoot);

    m_separation.clear();
    m_separationEst.clear();

```

```
    sim2::UtcTime dep(getPrimaryKey()->get("FPL_DEP"));
    sim2::UtcTime arr(getPrimaryKey()->get("FPL_ARR"));
    sim2::UtcTime curTime(dep);

    // We only continue if the AC has been simulated.
    if( !dep.IsValid() || !arr.IsValid() ) {
      LOG_MACRO(LOG_DEBUG, "Departure or arrival time is invalid for aircraft "<< getPrimaryKey()->name());
      return;
    }

    // Store the previous, current and next positions to compute the angle at current position.
    QPoint prev, cur, next;
    QPoint left, right;
    SepPoint firstSepPoint, curSepPoint;
    SepType sepTypeDefault = getSepTypeDefault();
    SepType sepType = sepTypeDefault;
    bool lastEst = false; // Indicates whether last point was estimated
    double angle = 0.0;
    int sepPixels[NR_SEP_TYPES];
    for (int st=0; st<NR_SEP_TYPES;st++) {
      if(m_sepValues[st] > 0.0)
        sepPixels[st] = dist2Pixel(painter, sim2::Distance(sim2::Distance::NAUTICAL_MILES, m_sepValues[st]));
      else
        sepPixels[st] = 0;
      LOG_MACRO(LOG_DEBUG, "Determined "<<m_sepValues[st]<<" nm as "<<sepPixels[st]<<" pixels");
    }

    double x,y;
    unsigned int count = 1;
    // Iterate through flight path.
    for(std::vector<QPoint>::iterator it = _path.begin(); it != _path.end(); ++it, count++)
    {
      cur = *it;
      sepType = getSepType(xy2pos(painter, *it), sepTypeDefault);
      if(count >1) {
        angle=calcAngleXY(prev, cur);
        //INV: angle is the angle of the vector from prev to cur, in the range [-M_PI..M_PI[
        //angle=fmod(atan2(y2-y1,x2-x1),M_2PI);
        //angle = fmod(atan2(y1-y2,log(tan(x2/2+M_PI_4)/tan(x1/2+M_PI_4))),M_2PI);

        if(count == 2) {
          // Finish processing the first point
          //INV: angle is the angle of the vector from the first point to the second one
          //INV: angle is in the range [-M_PI..M_PI[
          firstSepPoint.angle = angle;
          x = prev.x()+sepPixels[firstSepPoint.sep]*cos(angle+M_PI_2);
          y = prev.y()+sepPixels[firstSepPoint.sep]*sin(angle+M_PI_2);
          firstSepPoint.left  = QPoint((int)x,(int)y);
          x = prev.x()+sepPixels[firstSepPoint.sep]*cos(angle-M_PI_2);
          y = prev.y()+sepPixels[firstSepPoint.sep]*sin(angle-M_PI_2);
          firstSepPoint.right  = QPoint((int)x,(int)y);

          m_separation.push_back(firstSepPoint);

          if(firstSepPoint.sep == SEP_RADAR_NID) {
            // Draw estimated separation as well
            x = prev.x()+sepPixels[SEP_RADAR_ID]*cos(angle+M_PI_2);
            y = prev.y()+sepPixels[SEP_RADAR_ID]*sin(angle+M_PI_2);
            left  = QPoint((int)x,(int)y);
            x = prev.x()+sepPixels[SEP_RADAR_ID]*cos(angle-M_PI_2);
            y = prev.y()+sepPixels[SEP_RADAR_ID]*sin(angle-M_PI_2);
            right  = QPoint((int)x,(int)y);
            m_separationEst.push_back(SepPoint(left, right, angle, SEP_RADAR_ID, firstSepPoint.time));
            lastEst = true;
          }
        }
        if(it+1 == _path.end()) {
```

```
      // At last point, calculate left and right specially.
      //INV: angle is the angle of the vector from the second to last point to the last one
      LOG_MACRO(LOG_DEBUG, "Calculated angle of last point ("<<prev.x()<<","<<prev.y()<<")->("<<cur.x()<<","<<cur.y()<<")="<<angle*RAD2DEG);

      // rotate angle pi/2 left and right,
      // leftAngle = angle+M_PI_2
      // rightAngle = angle-M_PI_2
      x = cur.x()+sepPixels[sepType]*cos(angle+M_PI_2);
      y = cur.y()+sepPixels[sepType]*sin(angle+M_PI_2);
      left  = QPoint((int)x,(int)y);
      x = cur.x()+sepPixels[sepType]*cos(angle-M_PI_2);
      y = cur.y()+sepPixels[sepType]*sin(angle-M_PI_2);
      right  = QPoint((int)x,(int)y);

      LOG_MACRO(LOG_DEBUG, "Calculated left=("<<left.x()<<","<<left.y()<<"), right=("<<right.x()<<","<<right.y()<<")");
      curSepPoint = SepPoint(left, right, angle, sepType, curTime);

      m_separation.push_back(curSepPoint);

      if(curSepPoint.sep == SEP_RADAR_NID) {
       // Draw estimated separation as well
       x = cur.x()+sepPixels[SEP_RADAR_ID]*cos(angle+M_PI_2);
       y = cur.y()+sepPixels[SEP_RADAR_ID]*sin(angle+M_PI_2);
       left  = QPoint((int)x,(int)y);
       x = cur.x()+sepPixels[SEP_RADAR_ID]*cos(angle-M_PI_2);
       y = cur.y()+sepPixels[SEP_RADAR_ID]*sin(angle-M_PI_2);
       right  = QPoint((int)x,(int)y);
       m_separationEst.push_back(SepPoint(left, right, angle, SEP_RADAR_ID, curTime));
       lastEst = true;
      }
     }
     else {
      // At intermediate point, calculate left and right.
      next = *(it+1);
      double angle1=angle;
      double angle2=calcAngleXY(cur, next);
      //INV: angle1 is the angle of the vector from prev to cur,  in the range [-M_PI..M_PI[
      //INV: angle2 is the angle of the vector from cur  to next, in the range [-M_PI..M_PI[

      angle = calcAngleDiff(angle1, angle2);

      //INV: angle is the angle between the vectors (prev,cur) and (cur,next)
      LOG_MACRO(LOG_DEBUG, "Calculated angle of intermediate point ("<<prev.x()<<","<<prev.y()<<")->("<<cur.x()<<","<<cur.y()<<")->("<<next.x()<<","<<next.y()<<")="<<angle*RAD2DEG);

      // rotate angle1 pi/2+angle/2 left and pi/2-angle/2 right,
      // leftAngle = angle1+M_PI_2+angle/2
      // rightAngle = angle1-M_PI_2-angle/2
      x = cur.x()+sepPixels[sepType]*cos(angle1+M_PI_2+angle/2);
      y = cur.y()+sepPixels[sepType]*sin(angle1+M_PI_2+angle/2);
      left  = QPoint((int)x,(int)y);
      x = cur.x()+sepPixels[sepType]*cos(angle1-M_PI_2-angle/2);
      y = cur.y()+sepPixels[sepType]*sin(angle1-M_PI_2-angle/2);
      right  = QPoint((int)x,(int)y);

      LOG_MACRO(LOG_DEBUG, "Calculated left=("<<left.x()<<","<<left.y()<<"), right=("<<right.x()<<","<<right.y()<<")");
      curSepPoint = SepPoint(left, right, angle, sepType, curTime);

      m_separation.push_back(curSepPoint);

      if(curSepPoint.sep == SEP_RADAR_NID) {
       // Draw estimated separation as well
       if(!lastEst) {
        // Last point was not estimated, so we draw curSepPoint as the first estimated point.
        m_separationEst.push_back(curSepPoint);
```

```
            }
            x = cur.x()+sepPixels[SEP_RADAR_ID]*cos(angle1+M_PI_2+angle/2);
            y = cur.y()+sepPixels[SEP_RADAR_ID]*sin(angle1+M_PI_2+angle/2);
            left  = QPoint((int)x,(int)y);
            x = cur.x()+sepPixels[SEP_RADAR_ID]*cos(angle1-M_PI_2+angle/2);
            y = cur.y()+sepPixels[SEP_RADAR_ID]*sin(angle1-M_PI_2+angle/2);
            right  = QPoint((int)x,(int)y);
            m_separationEst.push_back(SepPoint(left, right, angle, SEP_RADAR_ID, curTime));
            lastEst = true;
          }
          else {
            if(lastEst) {
              // Last point was estimated, so we draw curSepPoint as the last estimated point.
              m_separationEst.push_back(curSepPoint);
            }
          }
        }
      }
      else {
        // Initially store empty left, right and angle, but sepType and curTime are valid.
        left = right = QPoint(0,0);
        firstSepPoint = SepPoint(left, right, angle, sepType, curTime);
      }

      // Prepare for next iteration
      prev = cur;
      curTime=curTime+sim2::Duration(sim2::Duration::OneMinute);
    }

    LOG_MACRO(LOG_DEBUG, "Done calculating "<<count<<" separation points for flight "<<getPrimaryKey()->name());
    setSepNeedsCalculation(false);
  }

  SepAcHandler::SepType SepAcHandler::getSepType()
  {
    if(withinRadar())
      return ((radarIdentified()) ? SEP_RADAR_ID : SEP_RADAR_NID);

    // Fall back to default.
    return getSepTypeDefault();
  }

  SepAcHandler::SepType SepAcHandler::getSepType(sim2::Position pos, SepType fallbackSep)
  {
    return getSepType(getRef(), pos, fallbackSep);
  }

  SepAcHandler::SepType SepAcHandler::getSepTypeDefault()
  {
    return getSepTypeDefault(getRef());
  }

  bool SepAcHandler::withinRadar()
  {
    if(m_radarRefresh-- <=0) {
      m_radarRefresh = RADAR_REFRESH;
      world::Aircraft* ac = PM::narrow<world::Aircraft>(getPrimaryKey().ptr());
      m_withinRadar = withinRadar(ac->position());
    }
    return m_withinRadar;
  }

  bool SepAcHandler::radarIdentified()
  {
    if(!m_radarIdent)
      m_radarIdent = radarIdentified(getRef());
    return m_radarIdent;
  }

/* ------------- Static helper functions ------------- */
```

```
  // Convert a Position to x,y coordinates on the painter canvas.
  QPoint SepAcHandler::pos2xy(GPainter *painter, sim2::Position pos)
  {
    if(pos.IsNull())
      return QPoint(0,0);
    LatLon_dt p = pos.GetLatLon();
    painter->rad2deg(p);
    QPoint point = painter->ll2xy(p.lon, p.lat);
    LOG_MACRO(LOG_DEBUG, "Calculated pos2xy("<<pos.AsString()<<")=("<<point.x()<<","<<point.y()<<")");
    return point;
  }

  // Convert x,y coordinates on the painter canvas to Position.
  sim2::Position SepAcHandler::xy2pos(GPainter *painter, QPoint point)
  {
    LatLon_dt p = painter->xy2ll(point.x(),point.y());
    painter->deg2rad(p);
    sim2::Position pos = sim2::Position(p);
    LOG_MACRO(LOG_DEBUG, "Calculated xy2pos("<<point.x()<<","<<point.y()<<")=("<<pos.AsString()<<")");
    return pos;
  }

  // Convert a Distance to pixel distance on the painter canvas.
  int SepAcHandler::dist2Pixel(GPainter *painter, sim2::Distance dist)
  {
    static sim2::Heading east("090");
    static sim2::Position begin("640118N0223921W"); // H1 radar pos.
    QPoint beginXY = pos2xy(painter, begin);
    sim2::Position end(begin,east,dist);
    if(end.IsNull())
      return 0;
    QPoint endXY = pos2xy(painter, end);
    int pixels = endXY.x() - beginXY.x();
    LOG_MACRO(LOG_DEBUG, "Calculated dist2Pixel("<<dist.AsString()<<")="<<pixels);
    return pixels;
  }

  SepAcHandler::SepType SepAcHandler::getSepType(Ref ac, sim2::Position pos, SepType fallbackSep)
  {
    if(withinRadar(pos))
      return ((radarIdentified(ac)) ? SEP_RADAR_ID : SEP_RADAR_NID);

    if(fallbackSep >= 0 && fallbackSep < NR_SEP_TYPES)
      return fallbackSep;

    return getSepTypeDefault(ac);
  }

  SepAcHandler::SepType SepAcHandler::getSepTypeDefault(Ref ac)
  {
    if(ac->get("SURVEILLANCE_EQUIPMENT").find("D",0) != std::string::npos)
      return SEP_ADS;
    return SEP_PROC;
  }

  bool SepAcHandler::withinRadar(sim2::Position pos)
  {
    LOG_MACRO(LOG_DEBUG, "In withinRadar("<<pos.AsString()<<")" );
    if(pos.IsNull())
      return false;

    bool inRange = false;
    RefList radars = pm::findByType( pm::PmTree::getDefaultRoot(), "Radar" );
    for(RefList::iterator radar = radars.begin(); !inRange && radar != radars.end(); ++radar)
    {
      if((*radar)->get("ONOFF_STATUS") == "ON") {
        sim2::Position rpos((*radar)->get("POSITION"));
        sim2::Distance range(sim2::Distance::NAUTICAL_MILES, atof((*radar)->get("RANGE").c_str()));
```

```

        double arcLength = pos.AsThreeVector().arcLengthTo(rpos.AsThreeVector());
        sim2::Distance dist(sim2::Distance::RADIANS, arcLength);
        //sim2::Distance dist = (pos-rp);
        inRange = (dist < range);
        LOG_MACRO(LOG_DEBUG, "Checking radar range: pos="<<pos.AsString()<<" is "<<((inRange)?"":"not")<<" within radar "<<(*radar)->get("NAME")<<" at rp="<<rpos.AsString()<<", distance="<<dist.AsString());
      }
    }

    LOG_MACRO(LOG_DEBUG, ((inRange)?"At least one":"No")<<" radar had position "<<pos.AsString()<<" within range" );
    return inRange;
  }

  bool SepAcHandler::radarIdentified(Ref ac)
  {
    return (ac->get("SQUAWK_TIME") != "");
  }

  // Calculate the heading between from and to.
  sim2::Heading SepAcHandler::calcHeading(sim2::Position from, sim2::Position to)
  {
    LOG_MACRO(LOG_DEBUG, "In calcHeading("<<from.AsString()<<","<<to.AsString()<<")" );
    // TODO: Verify correctness
    if (from.IsNull() || to.IsNull())
      return sim2::Heading();

    double h = from.AsThreeVector().headingTo(to.AsThreeVector());
    return sim2::Heading(sim2::Heading::TRUE_HEADING, h);
  }

  // Calculate the relative angle of turn at intermediate point cur
  // If cur is either at first or last point, return the heading angle relative to the north pole.
  // Returns angle in degrees between [-180 .. 180[
  // angle is positive for right turn, negative for left turn.
  double SepAcHandler::calcAngle(sim2::Position prev, sim2::Position cur, sim2::Position next)
  {
    LOG_MACRO(LOG_DEBUG, "In calcAngle("<<prev.AsString()<<","<<cur.AsString()<<","<<next.AsString()<<")" );
    // 3 cases: at first point, at intermediate point or at end point.
    double angle = 0.0;
    if (cur.IsNull())
      return angle;
    if (prev.IsNull() || next.IsNull()) {
      // INV: We are at the first point or the last point
      if (!next.IsNull()) {
        // INV: We are at the first point
        angle = calcHeading(cur,next).Degrees180_180();
      }
      else if (!prev.IsNull()) {
        // INV: We are at the last point and next is null
        angle = calcHeading(prev,cur).Degrees180_180();
      }
    }
    else {
      // INV: We are at an intermediate point, so now we have 3 positions
      // that we can use to calculate angle of turn to the position cur
      // TODO: Verify correctness
      sim2::Heading h1 = calcHeading(prev,cur);
      sim2::Heading h2 = calcHeading(cur,next);
      angle = h2.Degrees180_180() - h1.Degrees180_180();
      LOG_MACRO(LOG_DEBUG, "Result: calcAngle("<<prev.AsString()<<","<<cur.AsString()<<","<<next.AsString()<<")="<<angle<<"="<<h2.AsString()<<"-"<<h1.AsString());
    }
    return angle;
  }

  // Calculate the angle of the vector from p1 to p2 in the xy canvas plane
```

```
  // Returns angle in radians between [-M_PI .. M_PI[
  // x axis is to east, y axis to south.
  // angle is positive for south, negative for north. x is to east.
  double SepAcHandler::calcAngleXY(const QPoint &p1, const QPoint &p2)
  {
    return atan2(p2.y()-p1.y(),p2.x()-p1.x());
  }

  // Calculate the difference in angles (angle between two vectors)
  // Returns angle in radians between [-M_PI .. M_PI[
  // angle difference is negative to the left, positive to the right (clockwise)
  double SepAcHandler::calcAngleDiff(const double &angle1, const double &angle2)
  {
    if(angle1 < -M_PI_2) {
//    s1 = -1.0;
      if(angle2 < -M_PI_2)
//      s2 = -1.0;
        return fmod(-angle2+angle1,M_PI);
      else
        return fmod( angle2+angle1,M_PI);
    }
    else {
      if(angle2 < -M_PI_2)
//      s2 = -1.0;
        return fmod(-angle2-angle1,M_PI);
      else
        return fmod( angle2-angle1,M_PI);
    }
  }
}
```

Appendix D

Amelia Internals

This chapter is published separately from the main document since it contains portions of source code and API documentation copyright of Tern Systems.