Ilmi Ali

Generic Control Station

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

18 May 2013

Metropolia

Author(s) Title Pages Date	Ilmi Ali Generic Control Station 64 pages + 5 appendices 18 May 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Systems
Instructor(s)	Markku Karhu, Head of Degree Programme Angel Chiou, System Engineer

Cassidian, a defense and security company, employs a simulation framework called SIRIUS that allows the development of simulation models and applications and provides a runtime environment for them. SIRIUS is intended for developing as well as testing aircrafts. SIRIUS lacks a control system designed to control the simulation models. The purpose of this project was to provide this by developing a Generic Control Station that would allow managing and controlling simulation models.

Generic Control Station (GCS) provides a graphical user interface, an API, and a service for controlling simulations and observing them. The GUI consists of pages that are designed to control a specific simulation model.

The graphical user interface of GCS was developed on Eclipse Rich Client Platform, using Java. The graphical user interface is built with SWT and JFace windowing toolkits, and the underlying data is designed and generated with EMF Framework. The API and the service were developed as extensions of SIRIUS Framework using C/C++ languages.

As result GCS, a scalable application for controlling simulations, was developed. All the initial requirements set by Cassidian were met in the project.

Keywords	Cassidian, Generic Control Station (GCS), Eclipse RCP, SWT, Jface, API, C, C++, Java, Sirius

Tekijä(t) Otsikko Sivumäärä Aika	Ilmi Ali Yleinen Simuloinnin Ohjausjärjestelmä 64 pages + 5 appendices 18 Toukokuuta 2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Sulautettu tietotekniikka
Ohjaaja(t)	Markku Karhu, Koulutuspäällikkö Angel Chiou, System Engineer

Työn tavoitteena oli kehittää Generic Control Station (GCS- ohjelma), joka sallii Cassidianin käyttämän simulointimallien hallitsemisen ja valvomisen käyttäen graafista käyttöliitymää. Työ tehtiin näytteeksi siitä, että Cassidian voisi kehittää oman ohjausjärjestelmän käyttäen vapaalla saatavilla ohjelmilla.

Työssä kehitettiin järjestelmä, joka sisältää graafisen käyttöliittymän (API) ja palvelun. GCS:n graafinen käyttöliittymä kehitettiin käyttäen Eclipse Rich Client Platformia ja Javaa. Graafinen käyttöliittymä rakennettiin SWT- ja JFace-ikkunointityökaluja käyttäen, ja datamallit on suunniteltiin ja kehitettiin EMF Frameworkillä. API-ja palvelu kehitettiin laajentamalla SIRIUS Frameworkia käyttäen C / C + +- kieltä

Projektin tuloksena oli GCS, skaalattava ohjelma SIRIUS+simuloinnin hallinointiin. Projektin päätteeksi kaikki Cassidianin asettamat vaatimukset saatiin toteutettua.

Avainsanat	Cassidian, Generic Control Station (GCS), Eclipse RCP, SWT, Jface, API, C, C++, Java, Sirius

Metropolia

Contents

Abstract

1 Introduction 1

1.1 Project Background 3
1.2 Structure of the Thesis 4

2 Project Requirements 5

2.1 Functional Requirements 5
2.2 System Requirements 7

3 Technology Used for the Development 10

3.1 Overview 10
3.2 Eclipse Framework 10
3.3 Eclipse Rich Client Platform 11
3.3.1 Plug-ins 12
3.3.2 Extension Points 13
3.3.3 OSGi 13

3.4 Eclipse Workbench 13
3.4.1 Perspective 14
3.4.2 View 14
3.4.3 Editor 14

3.5 SWT and JFace 15

3.6 Eclipse EMF 16
3.6.1 Ecore 16
3.6.2 Code Generation 17
3.6.3 EMF.Edit 17

3.7 Eclipse 4 vs. Eclipse 3.x 19

3.8 Disti GL Studio 19

3.9 Nasa WorldWind 19

4 Backend Development 21

4.1 Overview 21

4.2 Control Station API 22

4.3 Control Station Service 23

4.4 Components 24

4.5 Component Attribute 29

4.6 Communication Protocol 30

4.6.1 Service Procedures 32

4.6.2 Attribute Value Updating 34

4.7 Extensibility 35

5 Graphical User Interface Development 37

5.1 Overview 37

5.2 Module Architecture 37

5.3 EMF Models 39

5.4 User Interface 40

5.5 Network Tree Viewer 41

5.6 Simulation Control 43

5.7 SideBar 44

5.8 Control Pages 46

5.8.1 AbstractView 46

5.8.2 Button Control 47

5.8.3 Scale Control 47

5.8.4 Spinner Control 48

5.8.5 Scene Set-up page 49

5.8.6 Position Page 50

5.8.7 Weather Page 51

5.8.8 Malfunctions Page 52
5.8.9 General Page 53

5.9 Additional Plugin Development 54
5.9.1 Map Page 55
5.9.2 Display Page 56

6 Conclusion 59

6.1 Identified Problems 59

6.2 Future Development 60

References 61

Appendices

Appendix 1. ObjectBase Attributes

Appendix 2. WeatherBase Attributes

Appendix 3. Isession Ecore Model

Appendix 4. Update Value

Appendix 5. General View

List of Figures

Figure 1: GCS Overview .... **Error! Bookmark not defined.**1
Figure 2: SIRIUS General System Overview [17] 9
Figure 3: Eclipse SDK Architecture [14] 12
Figure 4: SWT & Swing Architecture [15] 15
Figure 5: Simple Ecore Model. 16
Figure 6: Jface Tree Viewer 18
Figure 7: Nasa WW Window [16] 20
Figure 8: GCS Architecture...22
Figure 9: Service Execution 23
Figure 10: Component Architecture 25
Figure 11: Communication Establishment 31
Figure 12: Service Procedure Execution 33

Figure 13: GCS GUI Architecture ... 38
Figure 14: GCS GUI Layout ... 42
Figure 15: Network Tree Viewer ... 42
Figure 16: Simulation Control ... 44
Figure 17: SideBar ... 45
Figure 18: Control Creation Process ... 46
Figure 19: Button Control ... 47
Figure 20: Scale Control ... 48
Figure 21: Spinner Control ... 48
Figure 22: Scene Set-up page ... 49
Figure 23: Position page ... 50
Figure 24: Weather page ... 52
Figure 25: Malfunctions page ... 53
Figure 26: General page ... 54
Figure 27: Map page ... 55
Figure 28: Display page ... 57

List of Tables

Table 1: ObjectBase Attributes ... 27
Table 2: WeatherBase Attributes ... 28
Table 3: Service Procedures ... 33

Listings

Listing 1: ComponentBase Initialize ... 25
Listing 2: Create New Attribute ... 35

Abbrevations

API	Application Programming Interface.
C	Programming Language.
C++	Programming Language.
GCS	Generic Control Station. Subject of the thesis.
GUI	Graphical User Interface.
IOS	Instructor Operating Station.
Java	Programming Language.
OSGi	Open Services Gateway initiative
RCP	Rich Client Platform. A platform for building Applications.
RPC	Remote Procedure Call.
SDK	Software Development Kit.
SWT	Standarf Widget Toolkit.
XML	Extensible Markup Language.

1 Introduction

Cassidian, a company operating in the field of security and defense, employs a simulation framework called SIRIUS, for developing models and services, as well as a simulation environment for them to execute. It allows simulation models to execute on different computer systems and communicate through the framework [1].

When developing, prototyping or testing a simulation, there is a need for controlling the simulation and managing scenarios. Although Cassidian has software for this, not one is based on open source technologies.

The aim of the thesis is to describe the development of Generic Control Station (GCS) software and how it is used to control simulations as shown in Figure 1. The thesis is divided into two main parts, describing the graphical user interface, which is in Java, and the application programming interface, including a service, which is in C/C++. About 70% of practical was done in Java and the remaining 30% in C/C++.

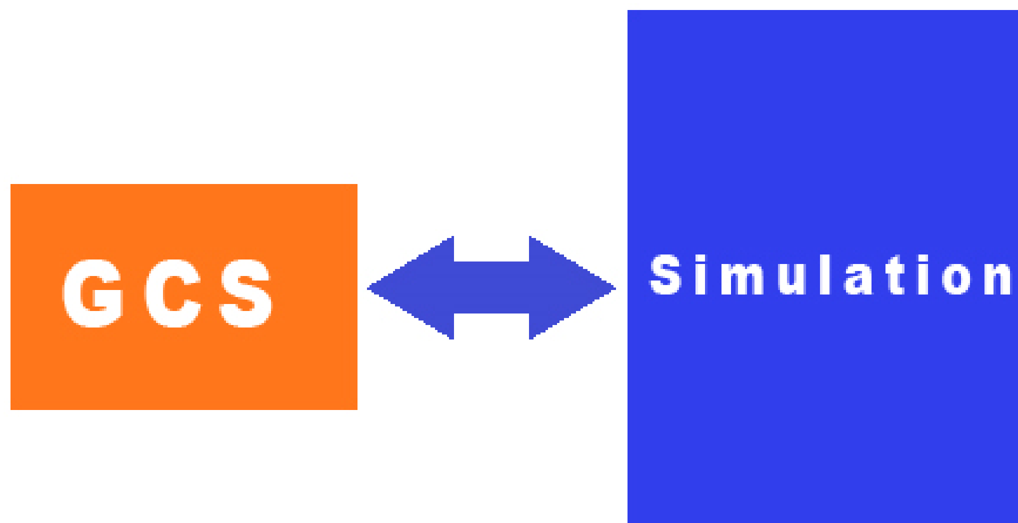


Figure 1: GCS Overview

Duo to an initial requirement by Cassidian the graphical user interface (GUI) is based on Eclipse Rich Client Platform, which gives the software dynamic modularity and allows the software to be extended with plug-ins.

The simulation framework allows developers to create and run models in C/C++, ADA and Fortran. For these models to be compatible with the Generic Control Station, they have to implement an interface, which also handles the communication protocol between the models and the Generic Control Station.

The goal of the project described in thesis was to integrate the simulation models with the Generic Control Station so that the running simulation can be controlled and observed, allowing such things as aircraft reposition, latitude longitude freeze, and weather control.

1.1 Project Background

Cassidian is a division of EADS (European Aeronautic Defense and Space Company), and it was formerly known as EADS Defense and Security. Other divisions of EADS are Airbus, Eurocopter and Astrium. Together with Cassidian these companies make the four pillars of EADS. EADS has about 133,000 employees in total; of this total 28,200 work at Cassidian. Cassidian is furthermore divided into three units: Cassidian Systems, Cassidian Electronics and Cassidian Air Systems. The headquarters are located in Germany Unterschlessheim, Cassidian Air Systems has its main location at Manching [2.]

Cassidian has such products as unmanned air systems, for example Euro Hawk, Barracuda and Talarion, and aircraft for example Eurofighter. The customers of Cassidian Air Systems are mainly military, and responsible to countries such as Germany, France and UK. EADS was formed in 2000. It consists of three companies that merged, these companies were Aérospatiale-Matra,DaimlerChrysler Aerospace AG (DASA), and Construcciones Aeronáuticas SA (CASA). The merger took place as the result of the European governments wanting to integrate their defense contractors into a single company [2.]

A flight simulator is a system that simulates the flight conditions, whether it is internal system (engines, gears, wings) or external environment (clouds, humidity, rain). Flight simulators can range from a simple computer model running on a laptop, to a full replica cockpit. Their type depends on their usage, which is generally training of pilots and crew and design and development of an aircraft [3.]

The advantages of flight simulators, other than being essential for training purposes, include saving time and money. Training a crew or a pilot with it is much more economical than with a real plane; the difference rate between the cost has been reported to being even as high 1:40 [3].

1.2 Structure of the Thesis

The thesis consists of six chapters. Chapter 1 provides an introduction to the project explaining why the project was required and what the project is about.

Chapter 2 goes into detail about the requirements for the project, starting with what functions the GCS is required to have. The chapter also explores the requirements from the system level.

Chapter 3 describes and analyses the technology that the GCS is based on, mainly describing the Eclipse Platform.

Chapter 4 focuses on the development of the backend of the GCS detailing the development of the interface that handles the communication between the GCS and the simulation.

Chapter 5 focuses on the development of the GUI of the GCS, using the Eclipse Platform.

Chapter 6 provides a conclusion. It also gives a description of identified problems, and possible improvements for the future.

2 Project Requirements

2.1 Functional Requirements

An IOS (Instructor Operating Station) is a system that allows a flight instructor to observe a flight simulator, and alter its conditions. As its name suggests, it is mainly used for training a crew by altering conditions, for example, in which they perform a landing task. The instructor can easily manipulate the simulated aircraft and its external environment. The instructor can set up different scenarios such as engine malfunction, stormy weather, icy conditions, and system malfunction.

The control station employs some of the same elements and functions as an IOS. However, it is not an IOS. This is due to its use and limited capabilities compared to the IOS. The main difference being that the control station is not used for instruction purposes but for controlling the running simulation. Nevertheless the control station follows closely the design and architecture of an IOS, in fact like IOS, it is based on the Arinic 610B standard, but does not implement all its functionality.

The ARINC Report 610B [3] describes the design guidelines that will be taken into account for the development of the Generic Control Station (GCS). The four basic function categories are:

- scenario set-up
- simulation control
- optimization
- maintenance Set-Up

The Scenario Set-Up category allows the user to set the initial conditions, as well as change the conditions during the simulation. It consists of the following functions:

- latitude/longitude change
- altitude change
- heading change

- airspeed change
- attitude change
- weight change
- fuel load change
- payload change
- reset to initial conditions
- temperature/pressure change
- wind change

The Simulation Control category gives the user functions such as freeze and unfreeze and gives overall control over the running simulation. It consists of the following functions:

- simulation freeze
- flight freeze
- fuel freeze
- latitude/longitude freeze
- altitude freeze

The Optimization category allows the user to optimize the running simulation to save time or simulation effectiveness. It consists of the following functions:

- speed optimization
- snapshot take
- snapshot recall
- multiple snapshots

The Maintenance Set-Up category can be considered as an additional category. It allows to control scenarios where there are faults on the aircraft. It consists of the following functions:

- fault logging parameter set
- fault memory clear
- fault memory load

From the GCS side the control of these functions was essential, but given the scope and time limit of the thesis, it was important to narrow down the overall functionality of

the GCS on some areas, and expand it on others. Therefore, the functionality of the GCS was restricted to:

- simulation control
- scenario set-up
- malfunctions set-up

The Scenario Set-Up in this case would be the same as in the Arinic Report [3].

The requirements for the simulation control are also described in HLA/RPR FOM 2.0D17[4]. It gives the simulation management the following additional functions:

- create entity/remove entity
- start/resume
- stop/freeze
- acknowledge
- action request/action response
- data query
- set data

The Entity function is out of the scope of this thesis as it is only required if the IOS is also controlling a synthetic environment, but the other functions are vital for the simulation control.

2.2 System Requirements

The Generic Control Station is primarily a control unit; it gives the user the control over the simulation, but does not actually implement these functions. In order for the Generic Control Station to control these functions, there has to be a running simulation, which implements a generic interface. The interface handles the connection between the simulation and the Generic Control Station.

The interface should be generic enough to allow any simulation model to run in it. It should also allow users to create any weather scenarios they wish to simulate, as well as any faults. Dividing the interface into three parts, aircraft, weather and faults, allows modularity, and makes the interface simpler. It also gives the possibility leave any of the three interfaces un-implemented if this should be necessary.

The simulation framework is a separate system that allows extending its capabilities through services. The simulation framework has network capabilities, allowing running the simulation on one station and observing it from another.

An important aspect was to build the system so that the integration would be simple and created with little effort from the developer side. Initially based on the architecture level, the system should have the following characteristics:

- easy integration
- seamless communication
- dynamic control

Having established this, the Generic Control Station should meet the following key system requirements:

- The control station shall have interactive controls. These controls are button, spinner and scale.
- It shall allow the user to control the running models.
- It shall have a complex graphical display, such as a Primary Flight Display, to indicate the altitude, speed, heading, pitch and roll.
- It shall have a map to represent the location of the simulated object.
- It shall integrate into the existing simulation framework.

The communication between the different parts of the system should also be seamless and happen in the background. An essential part is establishing communication between the simulation and the Generic Control Station. The GCS should be situated as a host on the system architecture level as shown in Figure 2. The models use an interface that can communicate directly with the GCS. The workbench already has a com-

munication system implemented that is based on UDP, but it is not fit for the GCS as it is limited to observing signals and changing their values.

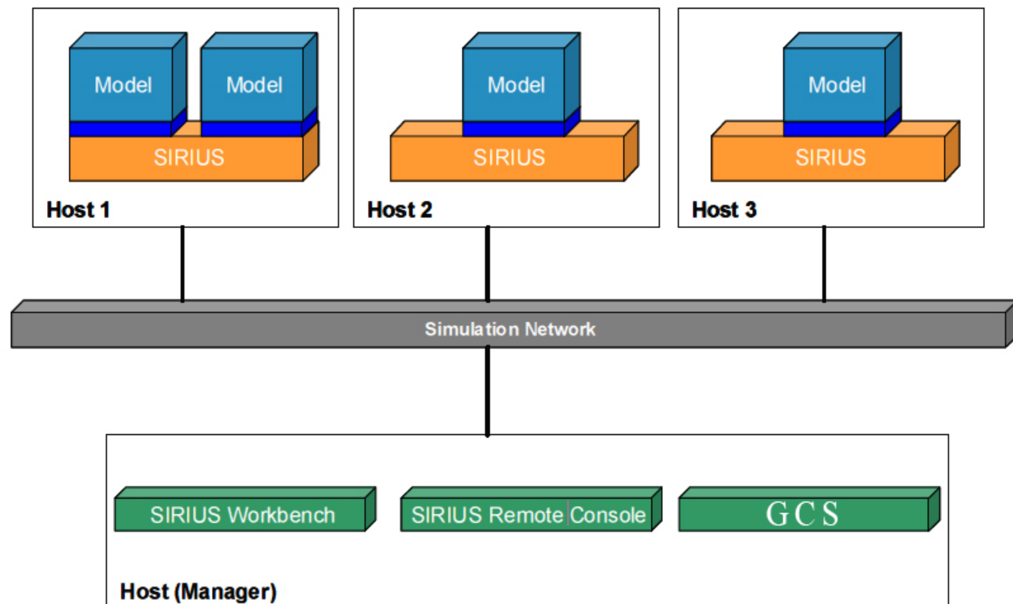


Figure 2: SIRIUS General System Overview [1]

GCS controls should be general enough to allow the control of a simulation without limiting its capabilities. Ideally the control system should be dynamic enough to allow the developer to extend its controls using the API.

An API is a set of standards that describes the way different parts of an application communicate with each other and exchange data. As API is application to application communication, it is widely used these days as a way for developers to use publicly available or other services. For example Microsoft Windows API allows developers to develop applications that run on the Windows environment. An API can be based on by set of abstract rules defined in documentation, and meant for libraries that implement the actual API.

3 Technology Used for the Development

3.1 Overview

In the initial requirements given by Cassidian, the GCS was to be based on Eclipse Framework. Eclipse employs multiple technologies such as EMF, SWT and JFace which are explored in this chapter.

The map and the display were requirements that were not specified by Cassidian, and neither was the technology they should be based on. For this reason, available technologies had to be explored.

For the display there were ultimately two options. The first was to develop a graphical display with OpenGL, the second to use an available solution for display development, such as Disti GL or Vaps XT. The first option would require much more work, as the display would have to be built from grounds up with OpenGL. The second option was to provide an available solution designed to develop displays with as little coding as possible. Both Vaps XT and Disti GL provide code generators; however, only Disti GL provides Java code generators. Since Eclipse Framework is based on Java, the ultimate choice was to develop the display with Disti GL.

Nasa World Wind (Nasa WW)and Google Earth were the two options for the map. The advantage of Google Earth was that it has more imagery data; however, Nasa WW is open source and thus allows for the developer to tailor it to their specific need. In this case Nasa WW was chosen for the map.

3.2 Eclipse Framework

Eclipse is an integrated development environment (IDE) that supports a multitude of different programming languages, such as Java (both Java SE and Java EE), C/C++, Perl and Ruby.

Eclipse is an open source project, meaning that it is free to download and develop on. Eclipse provides a stable platform for developers and students alike, whether it is to design, develop or deploy a commercial quality application. It gives the developers the power to develop applications on multiple platforms, since Eclipse is not platform specific.

3.3 Eclipse Rich Client Platform

One of the main tasks of Eclipse Platform is that plug-in should integrate seamlessly with the workbench as well as with other plug-ins [5]. Eclipse RCP is a plug-in development environment. It is widely used in large companies such as Google and IBM. In its essence it is built atop the Eclipse Platform, giving developers access to the core IDE, for their own development process. The term Rich Client Platform comes from the minimal set of plug-ins used to create a rich client application. Eclipse IDE itself is a rich client application. The benefits of Eclipse RCP is that it allows the developer to create and design sophisticated commercial quality applications with ease. Its strongest suite is its modularity and that it uses plug-ins to contribute. Its modularity allows putting components together to build any client application [7].

Eclipse RCP is constituted of the following applications:

- Equinox
- Core Platform
- Standard Widget Toolkit (SWT)
- JFace
- Eclipse Workbench

All the dynamic components are controlled by the runtime engine, which is part of the Core Platform. The idea of the core platform is building plug-ins to extend the system [3]. As its name suggests the Core Platform along with SWT is at the core of RCP application, as it is the base that everything else is built on as shown in Figure 3.

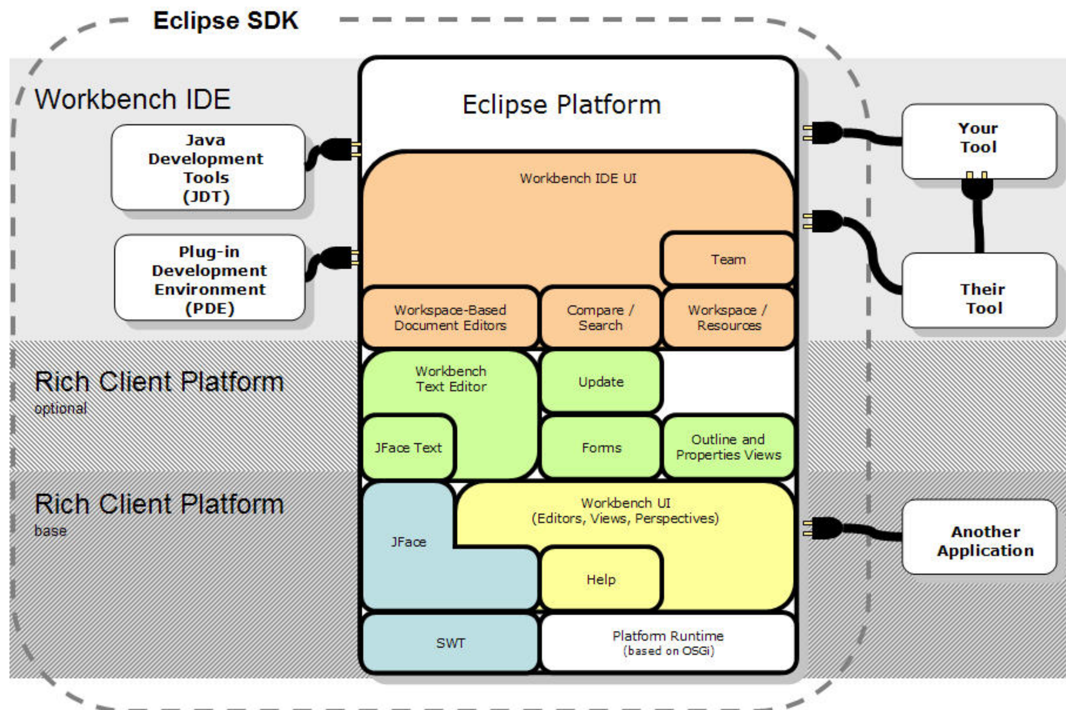


Figure 3: Eclipse SDK Architecture [14]

3.3.1 Plug-ins

A plug-in is essentially an OSGI bundle, because plug-ins are implemented using the OSGI framework [6]. The minimal plug-ins that define and therefore are required to run an RCP application are `org.eclipse.core.runtime` and `org.eclipse.ui`. In addition to these plug-ins, the user can add other self-made or already available plug-ins to define their own rich client application. In fact as mentioned before as far as the modularity of RCP application is concerned, everything in the application is a plug-in, with the only exception of the runtime.

An advantage of the plug-in is the dynamic loading that the OSGI service model provides. Plug-ins do not pay memory or performance penalty until they are activated by the runtime when a function a plug-in provides is requested [7].

3.3.2 Extension Points

RCP applications can declare interfaces that other plug-ins can contribute to. These are called extension points. Extension points are declared in an XML file called plug-ins and they are located in the root folder of the application. The platform maintains a registry of installed plugins and the functions they provide in the MANIFEST.MF file [7].

3.3.3 OSGi

OSGi specification is an answer to a restriction that the stand-alone Java Virtual Machine environment has. It is a dynamic module system that manages other components. Its focus is the development of new software, as well as the integration of existing software into new systems [7]. These components can be reusable, and the power of OSGi framework comes from constructing an application from these small components.

Because of the powerful dynamic component model provided by OSGI, it was chosen as the underlying runtime for Eclipse RCP and the IDE [7]. Equinox is the implementation of the OSGi framework that Eclipse uses. It uses a manifest file to dynamically control the life cycle of a bundle. These bundles are an important concept of OSGI, as a bundle is a dynamic component that can be remotely installed, started, stopped, updated and uninstalled without a reboot [6]. A bundle is self-contained, it defines its dependencies to other modules and services, as well as its external API [8]. All of this is defined in a manifest file.

3.4 Eclipse Workbench

As Eclipse IDE is also an RCP application, its workbench is a great example of what an RCP application consist of. It is built around the following concepts:

- Perspective
- Views
- Editors

In addition to these, there are workspaces and projects, but since they are not relevant to the topic of the thesis they will not be discussed in detail.

3.4.1 Perspective

A perspective is a collection of various views (e.g., Navigator, Outline, and Tasks) and editors [5]. A perspective specifies everything that is drawn on the screen, whether it is a view or an editor. A perspective can be compared to a page in a book, in the sense that only one perspective is visible at any time. In on RCP application perspective itself is declared in the extension point `org.eclipse.ui.perspective`.

3.4.2 View

A view is part of the workbench that is generally used to visually present data, for example by displaying object properties. In an RCP application a view is added via the `org.eclipse.ui.views` extension point in the plugin.xml file. The user has to declare their view either programmatically in the perspective class, or using extensions, in the perspective extension point. Views must implement in the `org.eclipse.ui.IViewPart` interface. Commonly this is done by extending the class ViewPart. Views share a common set of behaviors with editors due to the `org.eclipse.ui.part.WorkbenchPart` superclass which they inherit [5]. However, there are important differences such as if any change is made to the resources or data, it takes immediate effect on the view, whereas in an editor it has to be saved.

3.4.3 Editor

An editor is the part of the workbench that allows the user to edit an object [6]. Unlike views, which can be arranged in multiple ways, editors occupy only one pre-defined area in the perspective. In an RCP application an editor is added via `org.eclipse.ui.editors` extension point in the plugin.xml file. The user has to declare their editor either programmatically in the perspective class, or using extensions in the perspective extension point. Editors are typically resource based, whereas views can show resources from multiple sources or even something totally unrelated to resources [5]. Editors must implement the `org.eclipse.ui.IEditorPart` interface. Commonly this is done by extending the class EditorPart.

3.5 SWT and JFace

Standard Widget Toolkit (SWT) is a widget toolkit developed by IBM. The entire Eclipse UI is based on SWT, as it has better performance over the AWT and Swing (which is based on AWT) widget toolkits.

The main reason for choosing SWT over Swing or AWT for the Generic Control Station is the fact that Eclipse RCP UI is based on SWT. It is possible to use Swing as well as AWT in RCP, but there is less documentation about it, and thus far less support. However Swing has been around longer than SWT and it is in general more widely used. For example if "Java SWT" is typed into the Google search engine, at the time of writing this thesis, 10,800,000 results could be found for SWT, and for Swing twice as many, i.e. 22,000,000 results.

An application using SWT has the look and feel of a native application; this is due to SWT being a wrapper around the native window system as shown in Figure 4. This gives SWT an edge over Swing in terms of efficiency, as it uses native requests, making it faster [8]. As a downside, SWT widgets require manual object de-allocation; this is due to the native nature of SWT, as widgets are not tracked by JVM [6].

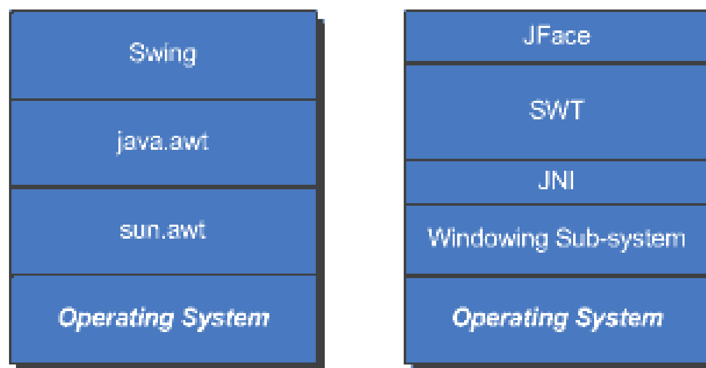


Figure 4: SWT & Swing Architecture [9]

JFace is a toolkit that is based on SWT, making it also window-system-independent. However, JFace does not hide SWT API [10].

The following is a list of some of the UI components of JFace:

- Viewers (ListViewer, ComboViewer, TreeViewer, TableViewer)
- Text, dialog, preference and wizard frameworks
- System resource management (Image, font, color)

Viewers are used to help with the interaction between data models, and widgets used to visually represent the model [6]. Actions are events that are called when a user clicks a toolbar or menu item. In addition to these, one of the strong suits of JFace is the data binding framework. It connects the properties of objects together, so that if change happens to one, the other is synchronized to the change.

3.6 Eclipse EMF

Eclipse Modeling Framework is an integral part of RCP application data sharing. EMF effectively unites Java, XML and UML technologies to enable rapid design [11]. In simple terms it allows code-generation from models and also generation of models with code. It brings together modeling and programming, allowing describing the behavior of the application with a model and generating the code from the description. However, simply put EMF is far more than a code generator. One of these aspects is model change notification, which will be detailed in section 3.6.2.

3.6.1 Ecore

A meta model is a model of a model which is referred to as Ecore in EMF. Ecore is a simplified sub-form of UML, as it is only concerned with one aspect of UML, class modeling [11]. Figure 5 shows a simplified Ecore model.

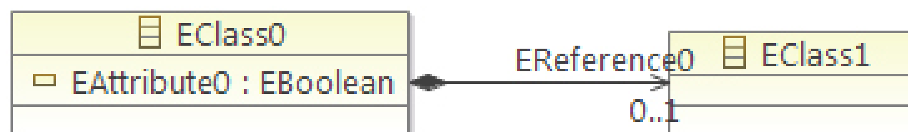


Figure 5: Simple Ecore Model

The elements in Figure are as follows:

- **Eclass** is the representation of a Java interface and its implementation class. It can have attributes associated with it, in this example Eclass0 has one attribute Eattribute0, while Eclass1 has no attributes.
- **Eattribute** is representative of a class attribute. It has a name and a type, in this case Eattribute0 is the name, and the type is boolean.
- **Ereference** is represative of a class association. It has a name and boolean flag to indicate its containment [11]. In this case, Eclass0 is the container of Eclass1.

3.6.2 Code Generation

Code generation, although not the only aspect of EMF, is certainly the most important aspect. The Eclass mentioned above is generated as an interface and as a class that implements the interface. In addition to these, it is also possible to generate Content and Label Providers for the classes; this will be further explained in section 3.6.3. In Figure the generated interface and class from Eclass, Eclass0 are Eclass0 interface and Eclass0Imp class. The reason for this division into an interface and a class is that the developers of EMF believe it to be a pattern that any good model such as API would follow [11].

All the generated interfaces are extended from Eobject, and the generated classes are extended from its implementation EobjectImp. This is the same for EMF as Java's Object base class, from which Java classes are extended. What is important about the Eobject is that it in turn extends from Notifier. This adds an important feature to all generated classes. Every time a change happens in an attribute, it can send a notification about it. The receiver of this notification is called an adapter. An adapter can be another Eobject, or it can be used to update views.

3.6.3 EMF.Edit

The EMF.Edit framework is used to build viewers to represent the underlying EMF model data, or alternatively to build editors to edit the model data. It provides Content and Label Providers, property source support and other classes to allow EMF models to be represented using JFace viewers [12].

An Important part of JFace is viewers, which are an integral part of data representation in EMF.Edit. JFace viewers use the content provider to navigate the content and the

label provider to retrieve the label text and icons for the objects [11]. Figure shows an example Tree Viewer.

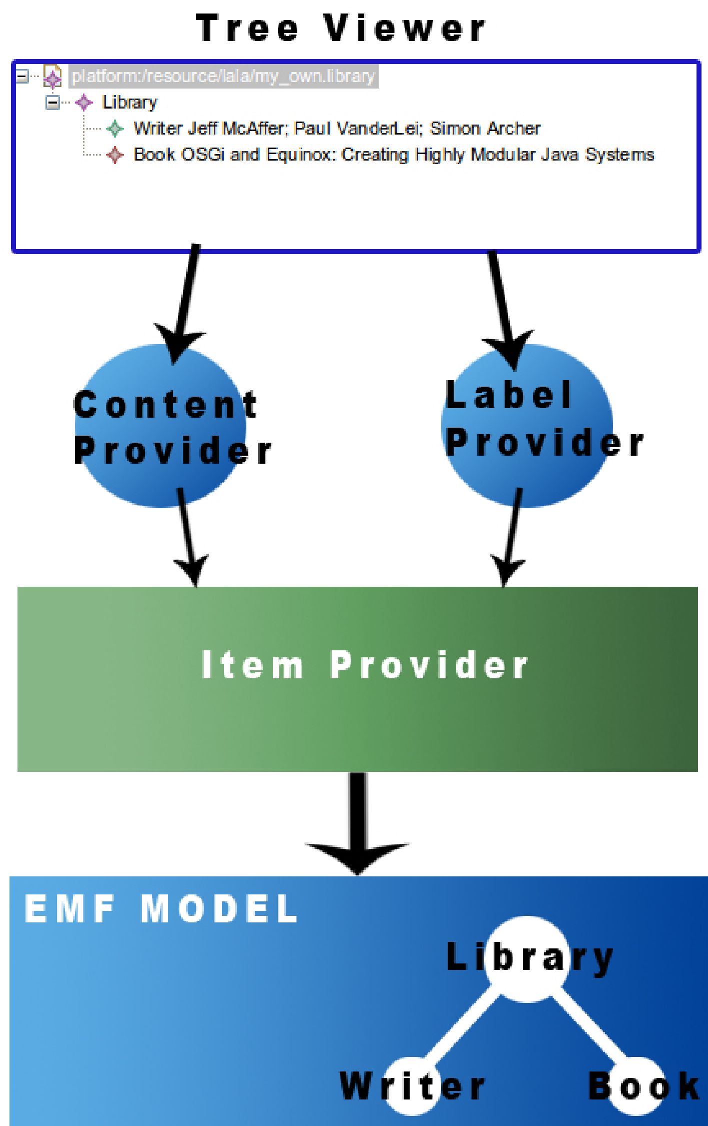


Figure 6: Jface Tree Viewer

The tree viewer uses a content provider and label provider to access the Library models in the item provider.

3.7 Eclipse 4 vs. Eclipse 3.x

Whereas Eclipse 3 uses extension points to contribute to the application, Eclipse 4 abandons this approach. Eclipse 4 is based on models in the approach of Eclipse EMF. Applications have to make their contributions using models. Eclipse 4 makes creating applications easier and faster. However after initially considering using Eclipse 4, Eclipse 3 was chosen in this project for compatibility with existing essential plugins developed by Cassidian.

3.8 Disti GL Studio

Disti GL Studio is commercial software designed to create 2D/3D graphics and interactive controls for variety of applications. It is been used for example to create virtual cockpits to automotive dashboards [13]. It allows the developer to create a display using a graphical interface and then generating code for it, which is easy to integrate. It allows creating graphics with virtually no programming from the developer; however, when implementing the behavior of the controls, programming is required. GL Studio can generate C++ and Java code from the designs that can be easily integrated.

3.9 Nasa WorldWind

The World Wind Java (WWJ) SDK is a virtual globe built on top of Java OpenGL [6]. It resembles Google Earth, but to the extent that WWJ is open source. WWJ uses satellites to provide data via NASA dataset servers and projects them onto a virtual globe viewer featuring applications such as high resolution imagery, surface analytics, terrain profiling, earthquake tracking, and airspace viewing, to name a just few features [14].

The WWJ API is mainly defined by interfaces; the following is a list of the major interfaces:

- WorldWindow
- Globe
- Layer

- Model
- SceneController
- View

WorldWindow is at the core WWJ class hierarchy. It provides a canvas for Swing/AWT on which the scene is drawn [6]. Likewise the SceneController and View are responsible for the view of the model and render and update the scene. In order to be used with SWT, SWT/AWT, a bridge has to be used. Layer, Globe and Model are the visual elements, Layer being the imagery dataset and its information, which uses a map tiling system. The layer is projected on the globe to make a model, using the Cartesian coordinate system to divide the sphere [6]. The globe in Nasa WW is as shown in the Figure 7.

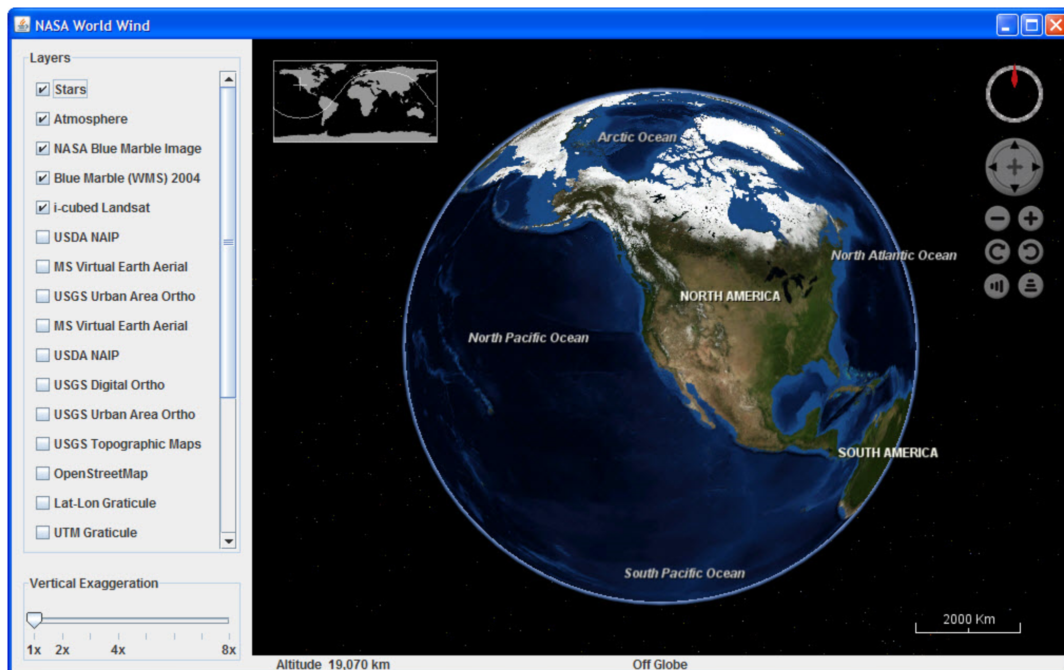


Figure 7: Nasa WW Window [x]

4 Backend Development

4.1 Overview

The development of the Generic Control Station was divided into two stages. The first one is the core development, which this chapter deals with. The main part of the core development can be thought of as the skeleton of the application. This is where most of the logics that run on the background take place.

The simulation framework offers an API that is easily extensible via services. To handle the communication between the models and the GCS, a service is loaded on the runtime that acts as a server that the GCS connects to and thus establishes an effective communication with the models. The system architecture is shown in Figure .

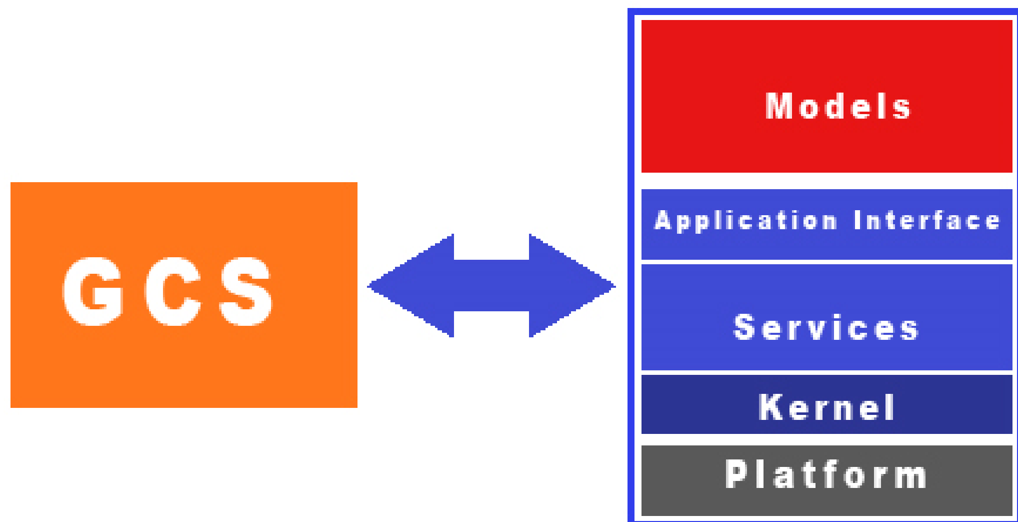


Figure 8: GCS Architecture

As shown in Figure 8, the GCS makes use of the available framework. In this case the service layer and the application interface. The communication between the simulation and the GCS happens through the service. The simulation models access the service through a set of interfaces, which allows the models to integrate into the GCS, as shown in Figure 8.

4.2 Control Station API

In order for a developer to develop a model that is compatible with the GCS, there had to be a description and a set of rules to do this. The developer would have to follow a certain interface, which would allow their model to run on the GCS. Furthermore, this interface should be compatible, and ideally built around the existing interface that the simulation models mainly use, in this case the Airbus Standard AP 2633. For these reasons an API (Application Programming Interface) had to be developed for the GCS.

In this case, the API is a static library which has to be linked to the developed model, as well as dynamic service which is accessed on the runtime. The API, is the pure logic part of the application. It is in C/C++ and acts as intermediate between the simulation framework models and the GCS. It handles all the necessary communication between the models. The basis of the API is built as an extension of the simulation framework that Cassidian employs. It is analogous to the OSGI logic, where the different components can dynamically be loaded on the runtime as services. These services are loaded in the beginning of the runtime and shared between the components.

4.3 Control Station Service

The Control Station Service is developed as an extension of the simulation framework. It is loaded by the runtime at the beginning of the simulation as shown in Figure . For the service to be loaded at the runtime, it has to be first called in a config file.

The life cycle of the service is detailed in the following figure.

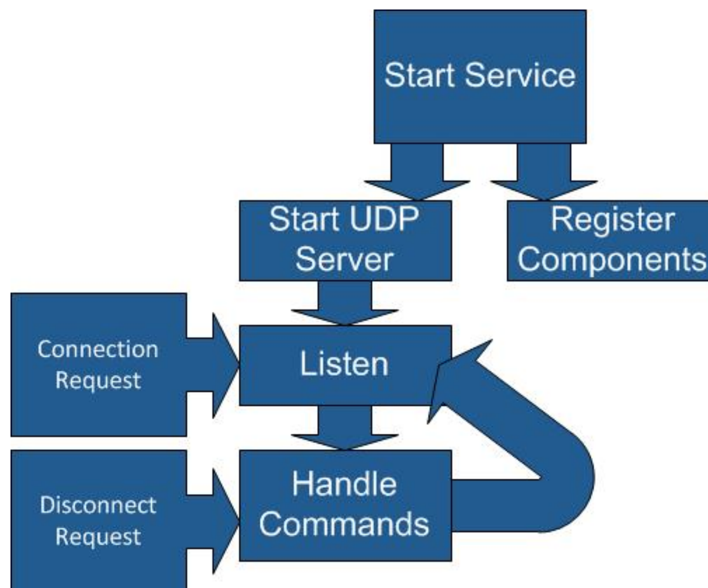


Figure 9: Service Execution

The service is loaded at the beginning of the simulation, after which it creates a separate thread to create the UDP server and listens out for a connection request, while it proceeds in the main thread to register the available components. The reason for using

User Datagram Protocol (UDP) instead of Transmission Control Protocol tcp, was to make use of the advantage when it comes to fast transmission data. After a connection request is received it goes on to establish a connection, after which the communication between the service and the GCS can begin. Finally, after a disconnect request is received it sends the GCS a disconnect response and closes the connection between the two. After this the service returns to listening out for incoming connection, or can be shutdown, which makes it to join the communication thread with the main thread, before the service is effectively terminated. The connection cannot be shut down from the simulation side, as this would cause an error in the simulation.

Its main task is to act as intermediate between the running models and GCS. The Control Station Service handles all the communication between the GCS and the models. It does so by wrapping around the communication protocol. Each component has to be registered to the service in the initialization as shown in Figure . All the registered components can receive messages and commands from the service, which are sent from GCS.

The service is in charge of listening out for all incoming messages and commands from the GCS and delivering them to the intendant recipient and the other away around from the components to the GCS. The communication protocol will be detailed in 4.6.

4.4 Components

As the models are mostly in C, this results in certain limitations. Most importantly the models are unable to use the service. The answer to this problem was the components. The components can be thought of as wrappers for the models that allow them to be integrated into the GCS system. In a sense the API of the system is the components, as they are the only visible (public) part of the interface.

The API consists of components that are a set of C++ interfaces and classes that are necessary for the developer to integrate into their model. The components are the bridge between the service and the simulation model. Their task is to observe the model and send messages to it from the service. Each component is registered to the service at the beginning of model initialization. These components have to be implemented by the developer; they have to extend the interface IModel.

The ComponentBase class is the implementation of the IModel interface; all the components are extended from it, as shown in Figure .

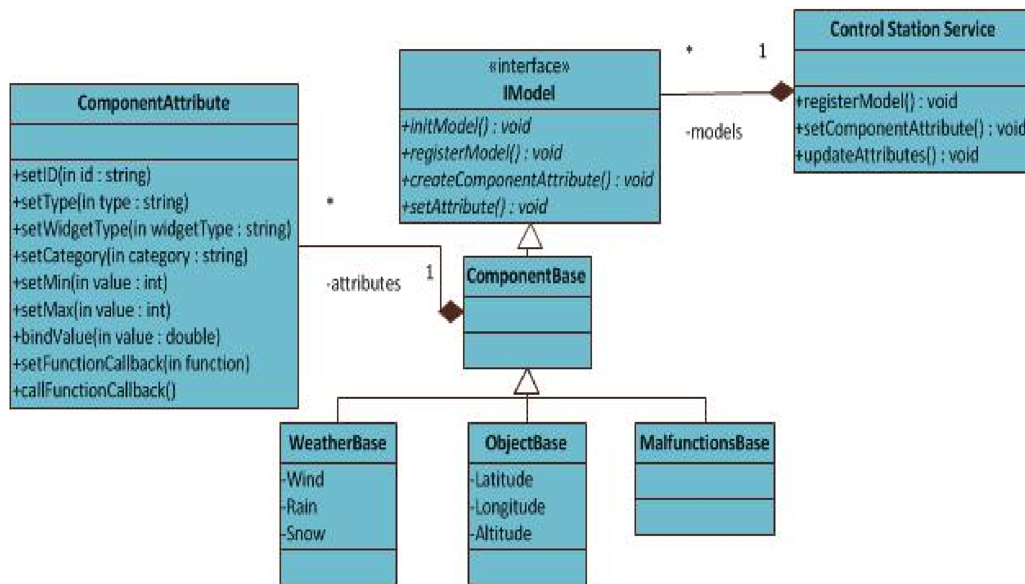


Figure 10: Component Architecture

The Component Base handles all the necessary processes such as registering itself to the Control Station Service, and announcing all its attributes. As an argument the Component Base takes the name of the component and a reference to the runtime, and calls the Control Station Service from it as shown in the listing.

```
void ComponentBase::initModel(String n, ObjectRef<Runtime> runtime)
{

    service = runtime->getObject<ControlStationService>("controlstationservice");
```

Listing 1: ComponentBase Initialize

This returns the service object, provided that the service was loaded at the beginning of the simulation. The service object is used by the component to register itself.

In addition to the Component Base, there are components which extend the Component Base, and add additional features to it. These are ObjectBase, WeatherBase and MalfunctionsBase, which all are for a more specific task. The Component Base itself

does not have any attributes declared, but ObjectBase and WeatherBase do. They employ attributes that can be directly controlled through the GCS. The different bases differ mainly based on their type declaration and the set of attributes they employ.

ObjectBase extends ComponentBase; it is to be used in such models like an aircraft model. Because of its generic nature, it can be used in almost any type of object, for example, car, train and bus.

The attributes for ObjectBase are as follows:

Attribute Id	***Widget Type***	***Category***	***Units***
Latitude	Spinner	Position	Degree
Longitude	Spinner	Position	Degree
Altitude	Spinner	Position	Degree
Heading	Spinner	Position	Degree
Speed	Spinner	Position	Km/h
Total Mass	Spinner	Mass	kg
Empty Mass	Spinner	Mass	kg

Payload Mass	Spinner	Mass	kg
Total Fuel	Spinner	Fuel	kg
Intern Fuel	Spinner	Fuel	kg
Extern Fuel	Spinner	Fuel	kg
Position Freeze	Button	Freeze	-
Altitude Freeze	Button	Freeze	-
Speed Freeze	Button	Freeze	-
Fuel Freeze	Button	Freeze	-

Table 1: ObjectBase Attributes

The WeatherBase, as its name implies is intended for using on weather models. The attributes for WeatherBase are as follows:

Attribute ID	***Widget Type***	***Category***

Rain	Scale	Snow/Rain
Snow	Scale	Snow/Rain
Fog Height	Scale	Ground Fog
Fog Visibility	Scale	Ground Fog
Clouds Visibilty	Scale	Clouds/Visibilty
Clouds Top	Scale	Clouds/Visibilty
Clouds Bottom	Scale	Clouds/Visibilty
Temparature Position	Scale	Temparature
Temparature Air	Scale	Temparature
Temparature Sea	Scale	Temparature

Table 2: WeatherBase Attributes

These attributes are predefined in the ObjectBase and WeatherBase classes, (see appendix 1 and 2).They are initially inactive; it is up to the developer to set them active. The developer can set a callback function that is called whenever the attribute is called.

MalfunctionsBase differs as it has no pre-defined attributes, as it is completely dynamical and it is up to the user to define the attributes for it. This is detailed in section 4.7.

Each of the base classes has pre-defined types. These types have their own corresponding page in the GCS, which goes as follows:

- ObjectBase – Position Page
- WeatherBase – Weather Page
- MalfunctionsBase – Malfunctions Page

Each of these pages is inactive initially, but becomes active if a model using their base class is registered. This process will be detailed furthermore in chapter 5.

4.5 Component Attribute

The class represents the control on the GCS side. It is the link between the running model attributes and GCS controls.

It is composed of the following properties:

- ID
- state
- category
- widget type
- min value
- max value

These properties have a purpose that extends beyond the scope of the API. They define a control which is dynamically created based on the attribute declaration; this will be further explained in section 5.8.1. All the elements are mandatory to be set when creating a new attribute, aside from minimum and maximum values, which are by default set as 0 and 100.

In addition to these properties, the Component Attribute has the following Optional properties:

- binded value
- callback function

These properties do not have to be defined, but are nevertheless important, if not the most important part of the Component Attribute class. The binded value is a double value that is binded to the Component Attribute. The value can then be queried by the GCS using XML RPC, as will be furthermore explained in section 4.6. The callback function is a user defined function that can be set so that every time the corresponding attribute is called the callback function will be triggered.

4.6 Communication Protocol

Communication between the models and the GCS happens through the Control Station Service. The service employs a UDP socket, to which the GCS connects. The UDP connection is mainly used to establish the connection and to close the connection. The decision to use UDP came after the initial try with TCP, as the service acting as a TCP server and running the models at the same time was unsuccessful.

The connection establishment is executed as shown in Figure .

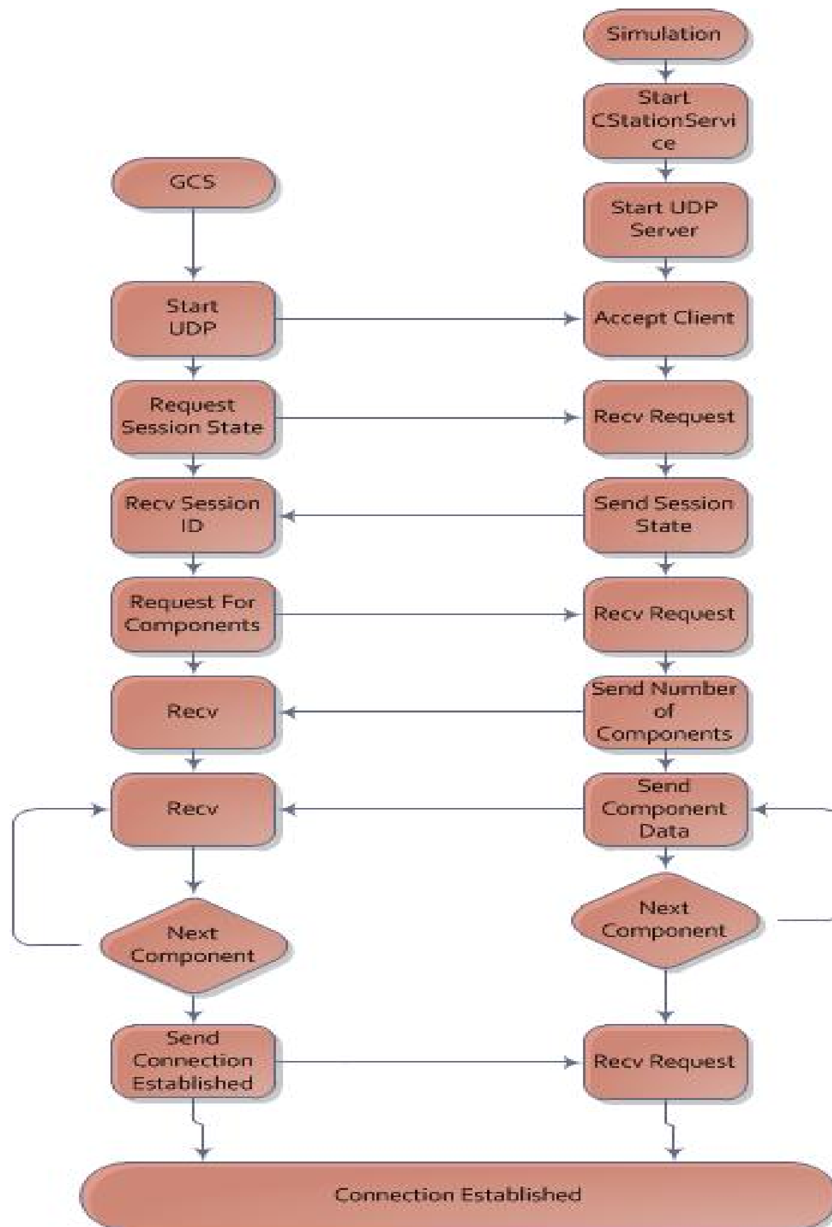


Figure 11: Communication Establishment

The communication begins with the GCS starting a connection to the open UDP port on the server side, which is the service. The server side receives the start connection request and responds to it accordingly until the connection is established, and the communication between the two can begin. The component data the service sends is directly accessed from the registered components.

The data that is sent is as follows:

- component name
- component type
- component state

Furthermore each Components attribute is sent:

- attribute ID
- attribute state
- widget type
- category
- binding state
- min value
- max value

This data is used to construct corresponding models that are direct representations of the components. These models and the connection they have to the components will be further explored in the next chapter.

4.6.1 Service Procedures

When the connection is established, the communication between the GCS and the service can begin. The communication is based on simple commands that the service relays to the components. These commands are generic and each executes a specific function. The command system is based on XML RPC. The simulation framework offers a scriptable interface which allows services to act as XML RPC servers. The GCS can remotely call procedures that are defined in the Control Station Service.

The defined procedures are:

Function	**Parameters**	**Return**

setComponentAttribute	parent, attributeID, value	-
getComponentValue	parent, attributeID	value
disconnect	-	-

Table 3: Service Procedures

These procedures are can be triggered by the GCS after the connection is established. The set procedure is executed as shown in Figure .

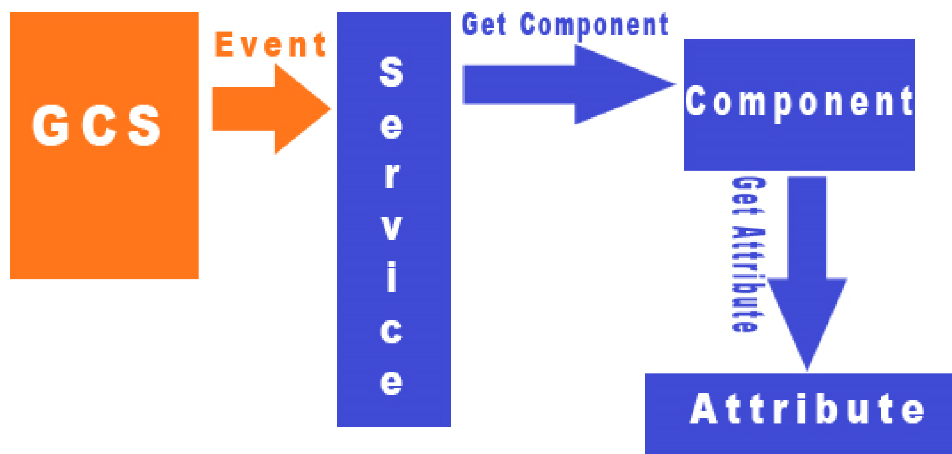


Figure 12: Service Procedure Execution

First an event is triggered, which sends setComponentAtrribute to the service. The service receives the request. First it checks which event is requested, and then it retrieves the parent that was passed as an argument. On finding the parent it retrieves the attribute that was passed as an argument.

The setComponentAttribute is sent when a user triggers a widget, passing a parameter parent the component that the attribute belongs to, attributeID, the name of the attribute, and the value that is set. The service retrieves the attribute as shown in Figure and then informs the parent component that there is a command waiting for the passed component attribute. The C model that has implemented the component sees then that there is a command waiting and will execute the callback function for the attribute, if defined.

Each component can have a user defined callback function, which will be triggered by the user on the GCS side. Each component has their own control in the user interface of GCS, which is called a widget. There are three types of widgets that a component attribute can have:

- Button
- Scale
- Spinner

How these widgets work will be further discussed in the next chapter.

The getComponentValue returns the value of the requested attribute, but only if it is binded. The binded value is a value that runs in the simulation and can be retrieved when it is requested by passing the name of the attribute and its parent as an argument to the service. The service then retrieves the attribute as seen in the figure, and returns its current value. The getComponentValue is not implemented in GCS currently, it is there for scalability.

The disconnect procedure can be triggered on the GCS side with the disconnect button, more of it in section 5.6. When triggered the connection between GCS and the service is closed as explained in section 4.6.

4.6.2 Attribute Value Updating

The attributes that have a value bound to them need to get updated on the GCS side. One option would be to make a procedure, which the GCS would invoke periodically in order to get the new value. Another way would be to make a notification system that

informs the service whenever the bounded value would change. Eventually the later way was chosen, making it so that the service constantly listens out for notifications from bound values that have changed, and sends the attribute ID and its new value to the GCS (see appendix 4).

4.7 Extensibility

Keeping in mind the philosophy of making the system generic, it was possible for the simulation model developer to make their own attributes and controls. As the main goal of the interface was to allow the control of the simulation, it was important for the interface and the user interface to be close representations of one another. Ideally there were two options, a static and dynamic interface. The static interface would follow closely the user interface, or the user interface would follow closely the interface. The components would implement this static user interface and thus integrate the model into the GCS. The dynamic option would be that the user interface would be created on the runtime through declared interface.

The dynamic interface was the better option in this case, allowing the system to be generic and extensible if needed. As described in section 4.4, each component has a set of pre-defined attributes. It is possible for the developer to make their own set of attributes, which will be registered by the service and used to create controls on the GCS. The following listing shows how to create a new attribute.

```
ComponentAttribute attribute("Latitude",true,"spinner","Position");
attribute.bindDoubleValue(&f);
acomponent->attachComponentAttribute(attribute);
```

Listing 2: Create New Attribute

The first argument is the name of the attribute, the second if it is active (true) or inactive (false), in this case it is active, and if inactive the control is greyed and disabled in the GCS. The third argument specifies the type of control. The different widget types are explained in section 4.4. The last argument is the category that the attribute belongs to; attributes of the same category will be displayed in the same group in the GCS (more about this in the next chapter). After creating the attribute the user can set minimum/maximum values for it, bind to a value which can be observed in GCS as shown

in the listing, or set a callback function which will be called whenever the corresponding control of the attribute is triggered.

Finally the component attribute must be attached to the component; this makes it available on the GCS.

5 Graphical User Interface Development

5.1 Overview

This chapter describes the main parts of the application in detail, and then goes in to detail about their use. The chapter also contains the extension of the application via plug-ins, which are the map plug-in and the display plug-in.

The Generic Control Station is an Eclipse RCP application. This gives it essentially the platform that the Eclipse IDE is built on as building blocks. The GCS allows the control of the models, but does not implement the functions that are to be controlled. Essentially it is a graphical user interface, whereas all the logic happens within the simulation models.

The main task of the GCS is to present the data in proper context and allow control over it. Keeping this in mind, it should have a simple user interface that is user friendly. Furthermore, the philosophy that the Eclipse Platform was based on was that it should be easily extensible. To demonstrate the power of Eclipse's modularity, the map and the display part of the application will be developed as separate plug-ins.

5.2 Module Architecture

Essentially all parts of the GCS are plug-ins, with the exception of the Control Station GUI plug-in being the core where the runtime is. As shown in Figure , the GCS is built on top of Eclipse RCP framework, and divided into five core plug-ins and into two external plug-ins

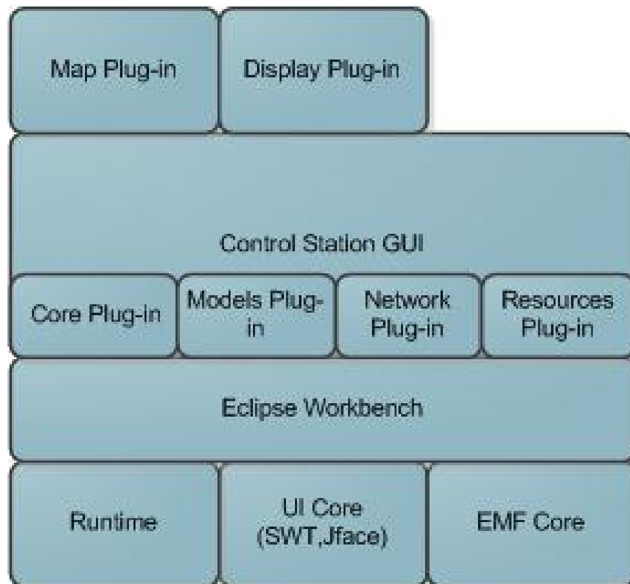


Figure 13: GCS GUI Architecture

The map and the display plug-in are external plug-ins, they are shown in Figure 13 at the top of the GUI rather than at the bottom because the GUI plug-in does not depend on them, and it can run without them, although in such a case the map and display page cannot be accessed.

The five core plug-ins and their purpose are as follows:

- GUI Plug-in is the GUI part of the GCS. It will be detailed in section 5.4.
- Core Plug-in is in charge of creating a new session when GCS connects to a simulation runtime with Control Station Service. A session is a representation of the components running in the simulation, and it allows control over them.
- Models Plug-in is a representation of the components. The model will be further explained in section 5.3.
- Network Plug-in consists of two different plug-ins. One is a plug-in made by Cassidian for their workbench, which was re-integrated into the GCS. The other is developed as part of GCS. These plug-ins will be detailed in section 5.5.
- Resources Plug-in is a simple plug-in that is in charge of loading and providing external resources, such as icons for the application.

5.3 EMF Models

The models of the GCS are representations of the components that run in the simulation framework. They are the underlying data in the GCS which the entire GUI is based on. The models are based on an Ecore diagram, which is generated into interfaces and classes as described in section 3.6.1 (see appendix 3 for the EMF model of ISession).

The models are all contained by session model, which represents the running simulation, having attributes such as state. The session model in turn is a child of the application model.

IControl represents the ComponentBase. Its attributes are:

- Name is the unique identification of the component.
- State is the state of the component.
- Type is the type of the component. The different types are Object, Weather, and Malfunctions.

IAircraftControl represents ObjectBase. It is extended from IControl class. Its attributes are:

- positionFreeze
- altitudeFreeze
- speedFreeze
- fuelFreeze

These attributes will be explained in section 5.8.6. The reason for ObjectBase having its own special class to represent it is that unlike WeatherBase and MalfunctionsBase, ObjectBase is not as generic and it has more functions compared to the two other classes. For instance, ObjectBase is represented in three different pages in the GUI, which are Position page, Display page and Map page, which will be further explored in the subsequent chapters.

IPropertyControl represents the attributes of the components. It also represents the graphical controls of the GUI. Its attributes are:

- PropertyID is the unique identification of the attribute
- WidgetType is the type of widget it is. Different widget options are scale, spinner and button.
- State is the state of the attribute. The state can be true or false, if false control is inactive and grayed.
- Category informs which group the control belongs. Controls with the same category are grouped together in the GUI.
- Value is the value that is shown in the spinner. If bound, the value will be updated accordingly. This will be further detailed in section 5.8.4.
- ExactValue is the true double value of the attribute. If bound the value will be updated accordingly. This will be further detailed in section 5.9.1.
- Min is the minimum value that the control accepts as input.
- Max is the maximum value that the control accepts as input.
- BindAvailable, indicates if a bind is available. If true, the control will receive value updates from the control station service, this will be further detailed in section 5.8.4.

The models are the crucial part of GCS as they are the underlying data that the GUI is based on. The following sections will explore the relationship between the models and the GUI.

5.4 User Interface

The layout of the application is designed for the needs of an IOS. Its design closely follows the design of an IOS system employed by Cassidian, but only to a certain extent, as it is not an IOS.

From the point of view of an IOS, the user requires a large view for the map page, but also the display page should be visible to view the exact altitude for example. The view could be split so the user can observe multiple pages at once, but at the cost of losing the large view. Instead the solution in this case was to add an extra view that the user can open and observe in an extra monitor. The advantage of Eclipse architecture is that each view can be dragged out of the perspective into its own view.

The center of the perspective is populated by the work pages. These are the position page, the weather page, malfunctions page, general page, setup page, map page and display page.

The right side shows the simulation control and the network view as shown in Figure 14; their purpose will be detailed in section 5.5 and 5.6. On the left hand side of the window there is the sidebar which contains page buttons. Each button opens a new page. This will be further explored in section 5.7.

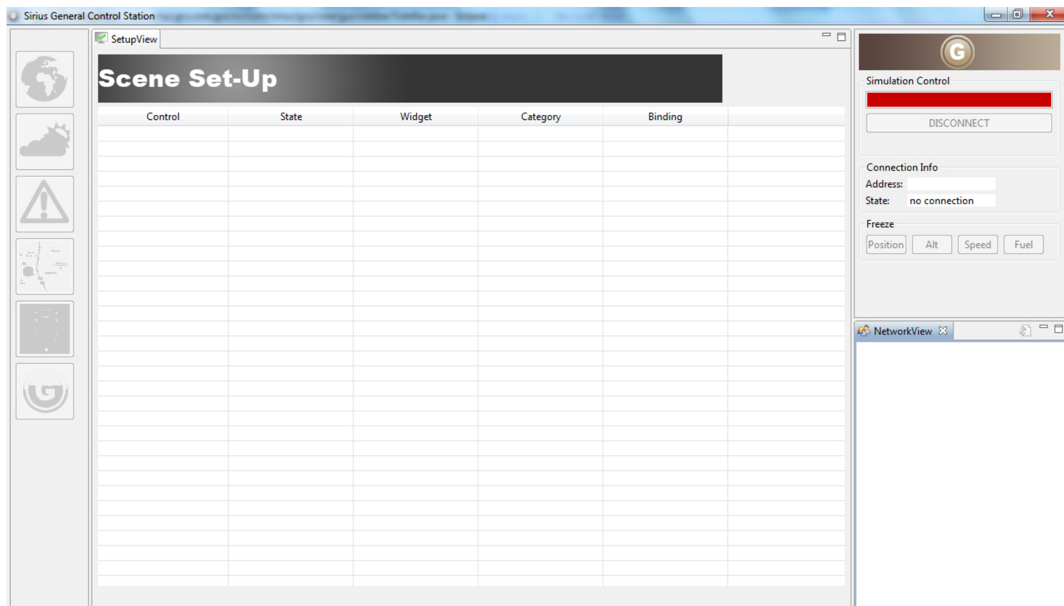


Figure 14: GCS GUI Layout

5.5 Network Tree Viewer

The network tree viewer presents the current simulations running in the network, presenting the network as a master node and the runtime as a child node as shown in Figure . By double clicking the runtime the GCS makes a connection with the runtime, provided that the runtime has GCS service running. On the left corner there is a refresh button for refreshing the view and making a new network scan that shows the current runtimes.

The network tree viewer uses the network plug-in from Cassidian's workbench. The plug-in scans the local area network for running simulations and creates model representations of them. These models are presented in the network tree viewer as shown in Figure .

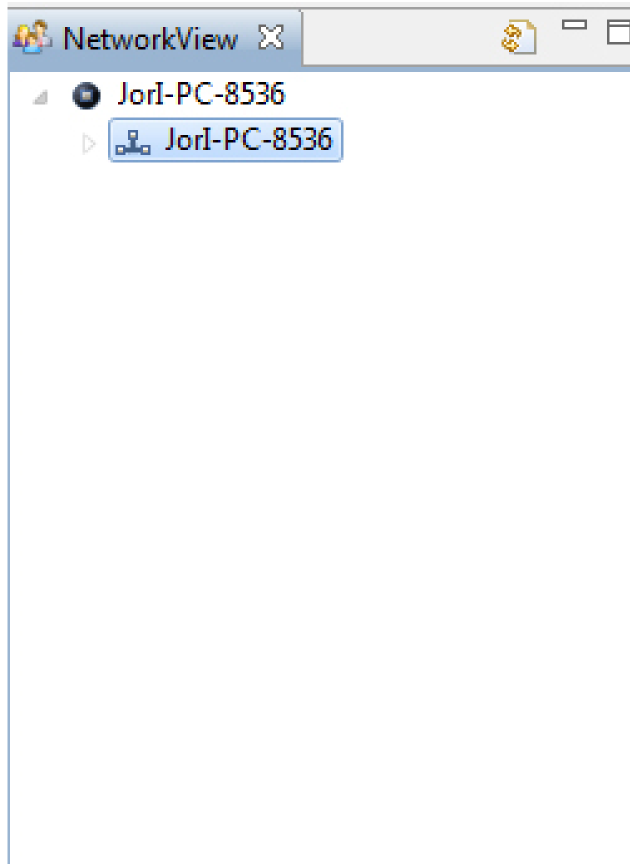


Figure 15: Network Tree Viewer

The connection is established between GCS and the service as described in section 4.6. After receiving the component information from the service, the GCS network plug-in, which is developed as part of the GCS creates model representations of them, giving each component type a model that represents it as described in 5.3.

After the models are created, the corresponding pages can be accessed, each page containing controls created from the attributes of its model. This feature will be further explored in subsequent sections.

Upon establishing connection, the GCS can send setComponent commands when the user triggers a control. This will be further described in section 5.8.1. The GCS can also get updated values for attributes that are bound, as described in section 4.6.2.

5.6 Simulation Control

The simulation control consists of three parts:

- Network State
- Network Info
- Freeze

The simulation shows the user the current state of the connection. The connection bar is initially red as shown in Figure 16, indicating that there is no connection currently. This can be seen in the State field. When an error occurs, the connection bar turns yellow and gives a connection failed message in the State message. Upon successful connection the bar turns green and the network address of the service is shown in the address section, and the state turns to running.

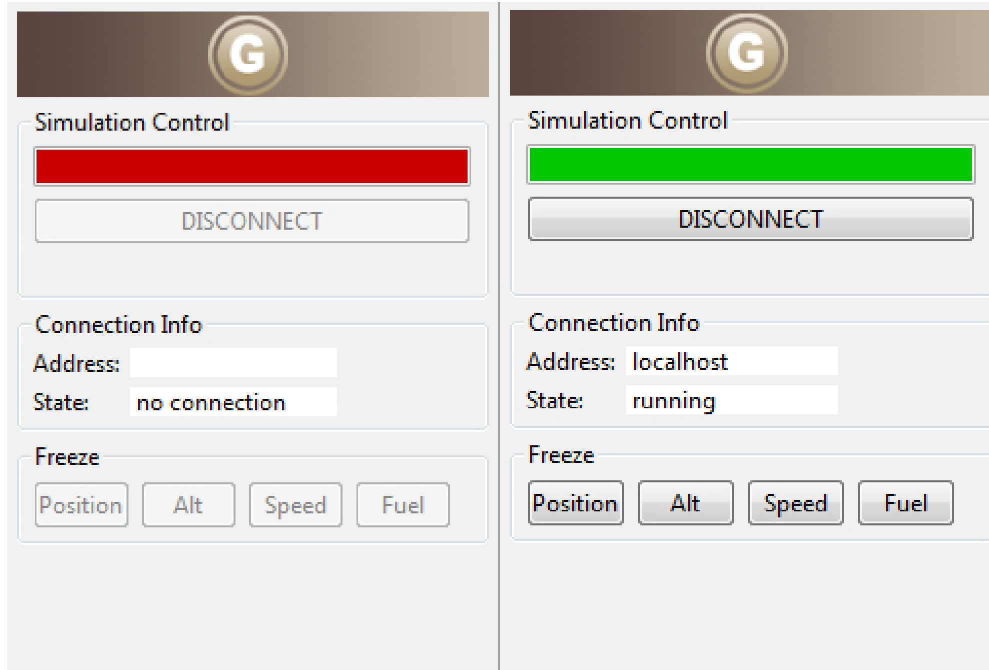


Figure 16: Simulation Control

With the disconnect button the user can disconnect from the current running session, sending a close request and effectively severing the connection between GCS and the running service. However, this does not close the simulation or the service, only the connection between the service and the GCS. The disconnect button also closes all the pages and returns the application to its initial state. The disconnect button is greyed initially, but becomes active after successful connection.

The freeze buttons are closely related to the Position Page, which will be explored in section 5.8.6. They become active only if ObjectBase component is implemented. They send setComponent calls to the service, with parameters on/off. It is up to the developer to implement the callback methods that these controls trigger as explained in section 4.5.

5.7 SideBar

The SideBar contains a set of SWT button widgets. Each widget opens a new page when clicked. Each button has a unique icon that represents its purpose. These but-

tons are all initially inactive, unless the ComponentBase they depend on is implemented.

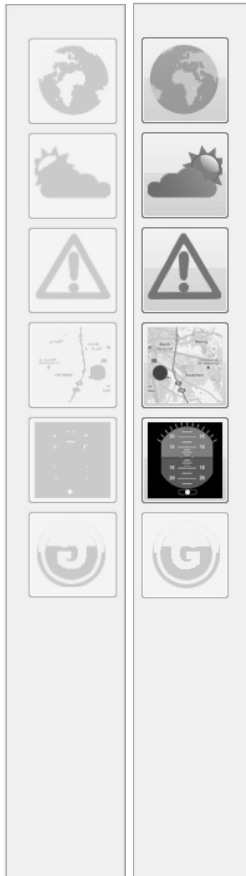


Figure 17: SideBar

The SideBar buttons and the components they are based on are as follows:

- Position page button, depending on ObjectBase.
- Weather page button, depending on WeatherBase.
- Malfunctions page button, depending on MalfunctionsBase.
- General page button depending on ComponentBase, with a type defined by the user.
- Map page button, depending on ObjectBase.
- Display page button, depending on ObjectBase.

If the component they depend on is implemented, the button becomes active when connected to a running simulation session.

5.8 Control Pages

5.8.1 AbstractView

AbstractView is an abstract view class that the Position, Weather, Malfunction and General control pages inherit from. Its main task is to supply the control classes with controls. It does this by having a set of methods that are create based on the IPropertyControl Ecore model a corresponding control. These controls are SWT widgets, which have data binding with the underlying IPropertyControl. Each control when triggered sends a setComponent call to the Control Station Service.

The graphical control creation process in GCS begins with the creation of the IControl models, as shown in Figure 18.

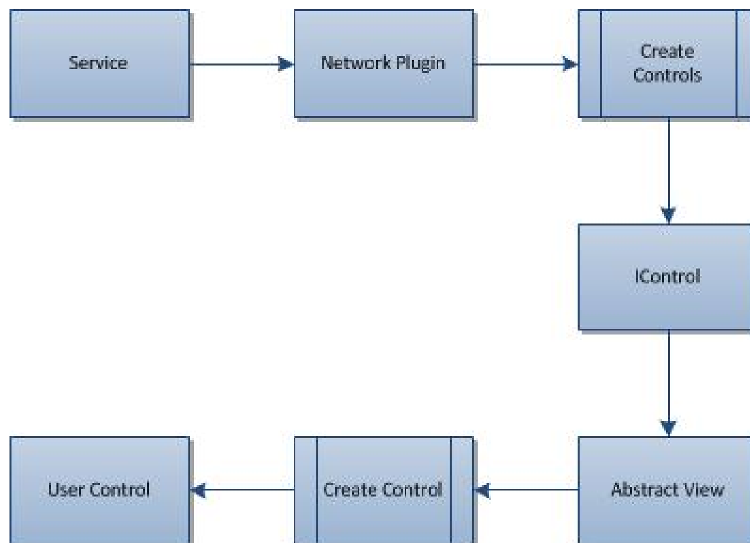


Figure 18: Control Creation Process

IControl as described in section 5.3 is a container class containing IPropertyControls. The IControl is given as a parameter to the createCategory Controls method, which parses through the IPropertyControls that the IControl contains. Each IPropertyControl

is represented with a widget that it has declared for, and each widget displays the ID of the IPropertyControl it belongs to.

The graphical controls are the following:

- Button
- Scale
- Spinner

5.8.2 Button Control

The button is a SWT button widget, with toggle style, meaning that the button has two modes, un-pressed and pressed, as shown in Figure 19. When the underlying attribute is not active, the button control is greyed. Upon user interaction, the button sends a setComponent call to the service. The button sends an on/off call as its value to the service.

Figure 19: Button Control

5.8.3 Scale Control

The scale is a set of two SWT widgets, a spinner and a scale. The two widgets are bound. When one changes, the other reacts to it to show the same value. The reason for having a spinner with the scale is to show the current value. The scale controller requires minimum and maximum values, which it acquires from the IPropertyControl it represents. The scale control works only with integer values, being limited compared to the spinner control. Another limitation that the scale control has compared to spinner is that it doesn't allow value binding. This means that if the component attribute is bound with value on the component side, it will not be shown in the scale controller. The reason for this is that the scale controller is strictly meant for controlling not observing values. When the underlying attribute is not active, the scale control is greyed as shown in Figure 20.

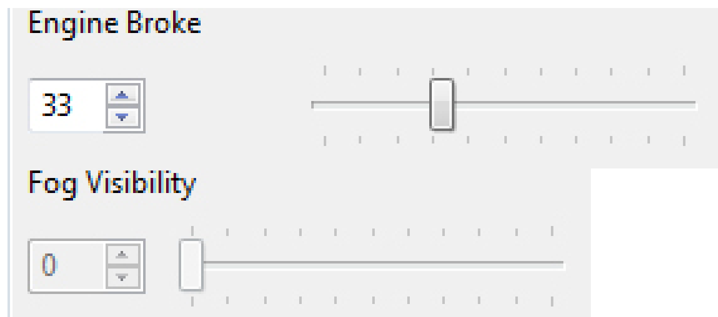


Figure 20: Scale Control

Upon user interaction the scale control sends a setComponent call to the service. The value that is sent is the current scale control selection value.

5.8.4 Spinner Control

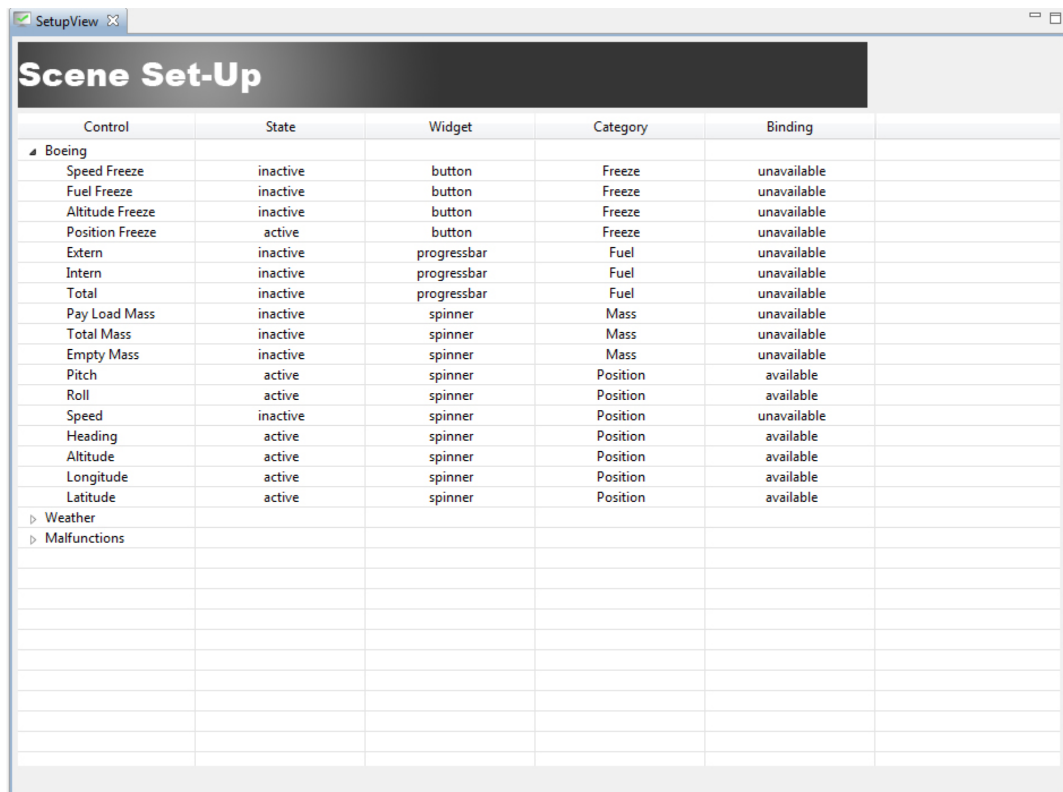
The spinner control is a SWT spinner widget. The spinner controller requires minimum and maximum values, which it acquires from the property control it represents. The spinner control allows bindings with attribute values, so that if the developer has bound the underlying attribute component with a value, it will be updated in the spinner. When the underlying attribute is not active, the spinner control is greyed as shown in Figure 21.

Figure 21: Spinner Control

Upon user interaction the spinner control sends a setComponent call to the service. The value that is sent is the current scale control selection value.

5.8.5 Scene Set-up page

The Scene Setup is a JFace tree viewer, which displays the contents of the session ecore model. The Scene Setup has no functions. It is purely for debugging. Initially the tree viewer is empty, but after a successful connection is established it displays the IControl Ecore models as shown in Figure 22. Each IControl model is displayed as a node with the name of the model. The node can be expanded to display the IProperty-Controls it contains.



Control	State	Widget	Category	Binding
Boeing				
Speed Freeze	inactive	button	Freeze	unavailable
Fuel Freeze	inactive	button	Freeze	unavailable
Altitude Freeze	inactive	button	Freeze	unavailable
Position Freeze	active	button	Freeze	unavailable
Extern	inactive	progressbar	Fuel	unavailable
Intern	inactive	progressbar	Fuel	unavailable
Total	inactive	progressbar	Fuel	unavailable
Pay Load Mass	inactive	spinner	Mass	unavailable
Total Mass	inactive	spinner	Mass	unavailable
Empty Mass	inactive	spinner	Mass	unavailable
Pitch	active	spinner	Position	available
Roll	active	spinner	Position	available
Speed	inactive	spinner	Position	unavailable
Heading	active	spinner	Position	available
Altitude	active	spinner	Position	available
Longitude	active	spinner	Position	available
Latitude	active	spinner	Position	available
Weather				
Malfunctions				

Figure 22: Scene Set-up page

It displays the following properties of IPropertyControl:

- Property ID
- State
- Widget
- Binding

5.8.6 Position page

The position page extends the AbstractView class. The position page is in charge of displaying and allowing the control of the underlying attributes from ObjectBase component. On this page repositioning of an object can be controlled, as well as freezing its position; however as stated earlier, it is up to the developer to implement the triggered event. For example, the developer has implemented the freeze position button so that when pressed it freezes the latitude and longitude of the model. Additionally, the developer can implement a function that will also change the latitude in the running model when the latitude is changed in the GCS.

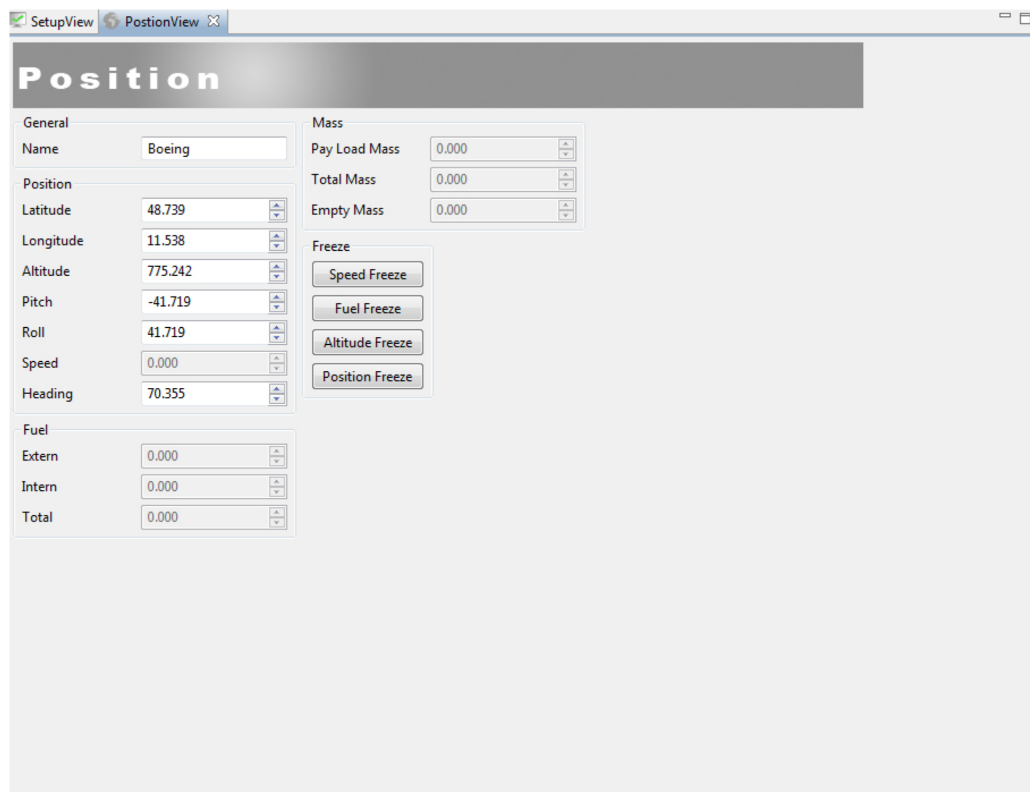


Figure 23: Position page

Attributes that are set active are not greyed and can take input. User inputs trigger an event in each control, which sends a setComponent call, as described in section 4.6.1. Attributes that are not set active are greyed.

The position page is accessed from the position button on the side bar. Its layout and controls are dynamically generated from the attributes defined in ObjectBase.

The pre-defined attributes of the position page are as shown in the Figure 23. The developer can add their own attributes to the position page, as described in section 4.7, and they will appear on the page. The developer can create their own new category. Controls with the same category are grouped together. The developer can also add new attributes to the already existing categories.

The freeze buttons are unique in the same way as they are bound with the freeze controls of the simulation control. When a freeze button is triggered on the page, the corresponding button on the simulation control will also change state, and vice versa.

5.8.7 Weather page

The weather page extends the AbstractView class. The weather page is in charge of displaying and allowing the control of the underlying attributes from the WeatherBase component. For example, the developer can implement a function that, will also change in the running model when the snow is changed in the GCS. Attributes that are set active are not greyed and can take input. User inputs trigger an event in each control which sends a setComponent call, as described in section 4.6.1. Attributes that are not set active are greyed.

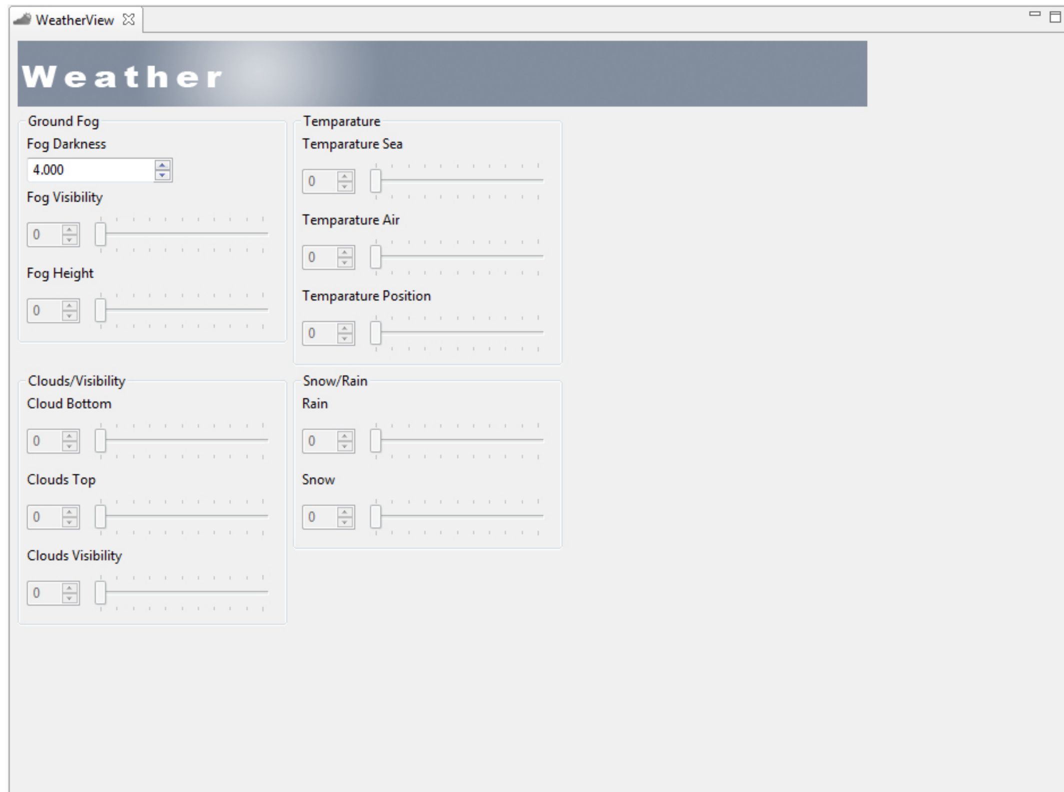


Figure 24: Weather page

The weather page is accessed from the weather button on the side bar. Its layout is and controls are dynamically generated based on the attributes defined in WeatherBase.

The pre-defined attributes of the weather page are as shown in Figure 24. The developer can add their own attributes to the weather page, as described in section 4.7, and the addition will appear on the page. The developer can create their own new category, in which controls will be grouped together, or add new attributes to the already existing categories, which will be added to the same group.

5.8.8 Malfunctions page

The malfunctions page extends the AbstractView class. The malfunctions page is in charge of displaying and allowing control of the underlying attributes from the MalfunctionsBase component. Attributes that are set active are not greyed and can take input.

User inputs triggers an event in each control, which sends a setComponent call, as described in section 4.6.1. Attributes that are not set active are greyed.

Figure 25: Malfunctions page

The malfunctions page is accessed from the malfunctions button on the side bar. Its layout and controls are dynamically generated from the attributes defined in MalfunctionsBase.

MalfunctionsBase has no pre-defined attributes, and as such, the page appears without controls, unless the developer defines their own controls.

5.8.9 General page

The general page extends the AbstractView class; it is a completely dynamical class with no predefined SWT widgets, (see appendix 5). The general page is in charge of displaying and allowing control of the underlying attributes from ComponentBase with a user defined type. Attributes that are set active are not greyed and can take input. User

inputs trigger an event in each control, which sends the setComponent call, as described in section 4.6.1. Attributes that are not set active are greyed.

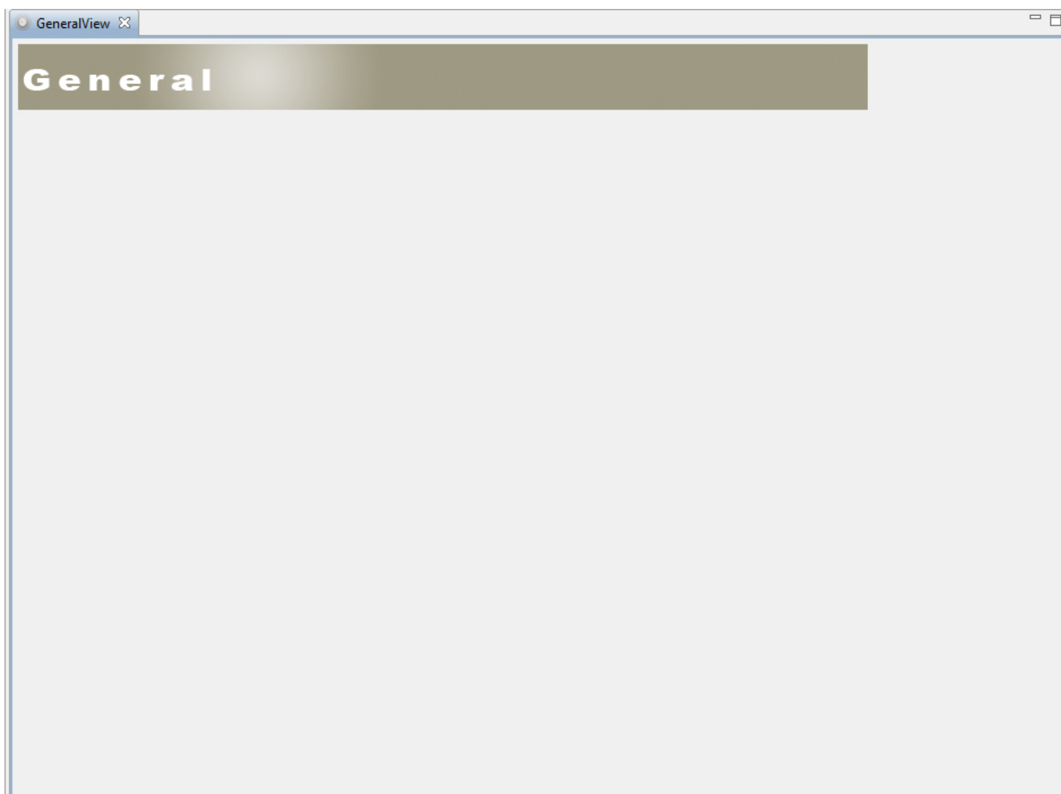


Figure 26: General page

The general page is accessed from the general button on the side bar. Its layout and controls are dynamically generated from the attributes defined in ComponentBase.

ComponentBase has no pre-defined attributes, and as such, the page appears without controls, unless the developer defines their own controls.

5.9 Additional Plugin Development

Keeping with the philosophy of Eclipse RCP modularity, the map and display page were developed as separate plug-ins. Another reason for developing them as separate plug-ins were their dependencies. Both plug-ins depend on JOGL library, and both

have individual external dependencies as well: map page has WWJ dependency and the display page has GL Studio library dependency.

5.9.1 Map page

The map page is opened by pressing the map page button on the sidebar. It displays Nasa WW globe with layers applied. Upon opening the page, first the imagery is loaded from the memory cache, which in turn loads the imagery from a committed server service over the internet.

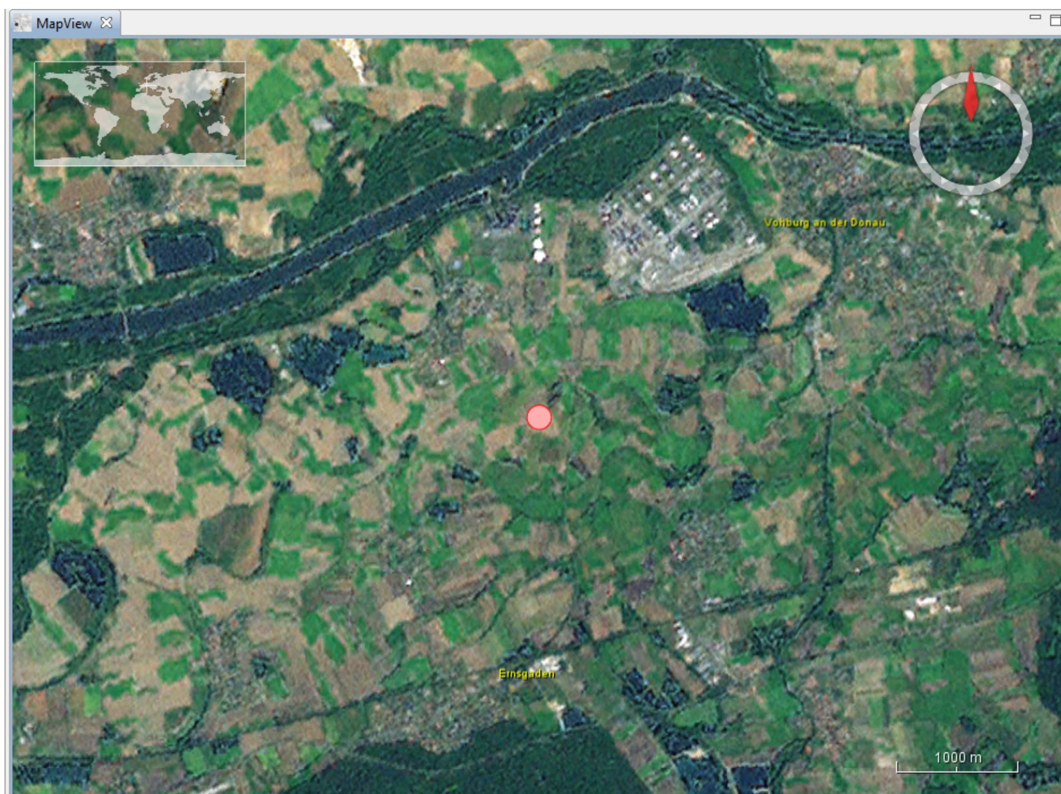


Figure 27: Map page

On the upper right corner, there is a compass, which points to north, and on the left corner a there is a small map of the world, showing the current position. The position of the ObjectBase component is shown as a red dot, using latitude, longitude and altitude attributes from the component. In order for the map page to display the position, the latitude, longitude and altitude attributes of ObjectBase have to be bound with values. The position is updated as the bound attribute changes on the component.

As the position page is showing the position of the ObjectBase component, all the changes made to the latitude, longitude and altitude will take effect also on the page. For example if the user uses the position page to reposition or freeze, the position will also take effect on the map page.

5.9.2 Display page

The display page is opened by pressing the display page button at the sidebar. The display page shows a set of displays that are present in cockpits. These displays are called PFDs (primary flight displays). They consist of a speed indicator on the left, an attitude indicator in the center, an altitude indicator on the right and a heading indicator at the bottom. Each display page shows the values of ObjectBase attributes, with a speed indicator bound to a speed attribute, an altitude indicator bound to an altitude attribute, a heading indicator bound to a heading attribute and an attitude indicator bound to roll and pitch attributes.

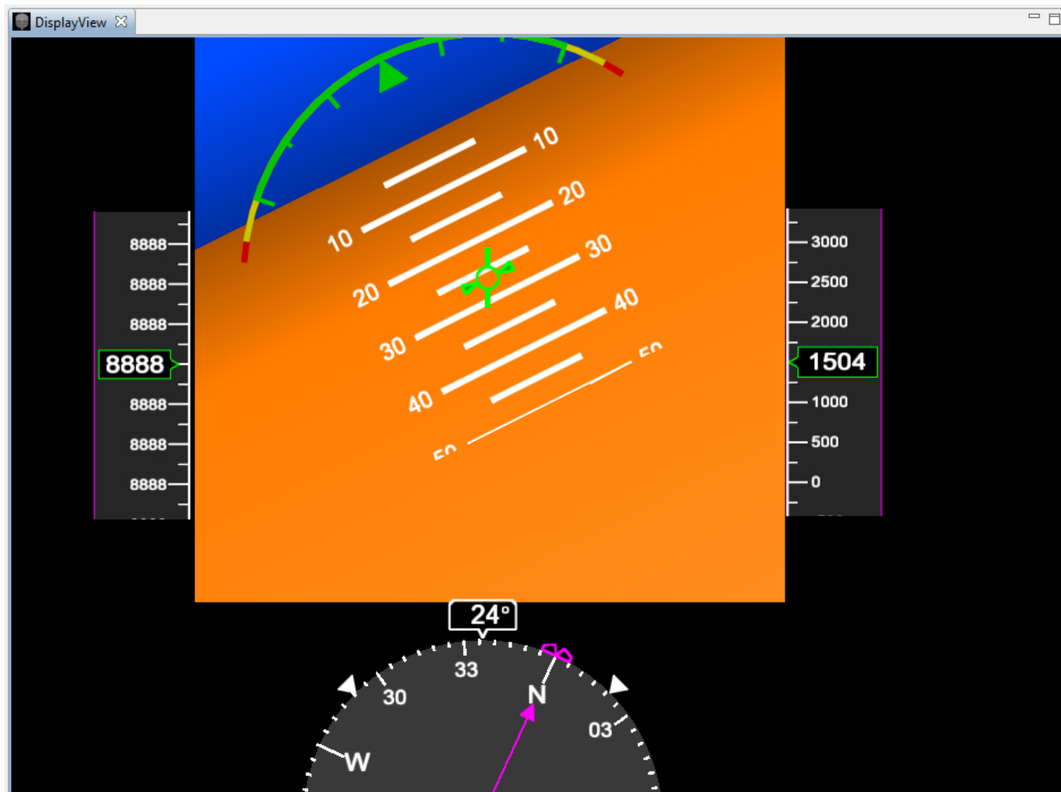


Figure 28: Display page

The display page was created using GL Studio and then integrated into Eclipse RCP as a plugin. The displays can be categorized into three different types: tape display, round display and attitude display.

Altitude and speed displays are tape displays which show the values as meters? aligned vertically. The current value is shown in the label in the middle of the tape display. As the value decreases, the meter travels down, and as the value increases, the meter goes up. When the tape display has no values and is in its initial state, it is displayed as Figure 28 shows.

The round display, which in this case is the heading display, shows the values aligned along the inner circle. The arrow shows the current heading, such as the label on top, showing the numeric value of the current heading.

The attitude display is more complex than the other displays. Primarily, it shows the roll and the pitch. The roll is indicated on the artificial horizon which rotates to indicate the

amount of roll. The above roll indicator and the little arrows around the middle circle display the little arrow as green when the roll value is between -45 and 45. The arrows show yellow when the value between -55 and 55, and as red above that. The pitch is indicated on the ladder which climbs up and down according to the pitch values. The pitch ladder shows values between -90 and 90.

6 Conclusion

The aim of the project was to provide means of control for simulations as well as to provide a standardized interface to implement control. The GCS was built on top of existing technologies and frameworks, specifically platform independent frameworks. Emphasis on using available technologies such as Eclipse RCP and the SIRIUS framework made the development significantly easier, as they provided low level infrastructure.

The GCS architecture provides an easy and scalable platform for controlling and monitoring simulations across the network. It allows the GUI to be extended by the user with custom controls according to the specific API.

Due to the remote control capabilities, it is possible to complement the architecture with additional scripts and applications. The use of platform independent technologies <-- ei pilkkua allows the GCS to be run on multiple architectures and operating systems.

6.1 Identified problems

When developing the GCS there were a couple of problems that where unresolved. These problems could not be fixed for the following reasons: time constraints or design decisions, or simply the cause of the problem was unknown.

One of the main problems was the UDP unreliability. When establishing the connection, or receiving updates from the service, there is a chance that the service sends the wrong packet, which causes the service to stop sending packets. This could be fixed with a reliability mechanism which checks that the packets are received in the right order, and if, not takes precautions to prevent any errors.

The map was another problem, but a minor one, as it can easily be fixed by closing the map view and opening it again. On some occasions, the map camera rotates for no

apparent reason; this sometimes stops the map camera from tracking the object in the map. The reason for this problem was not identified.

6.2 Future development

The GCS architecture was designed to be scalable from the beginning. The system of the frameworks is based on allowing it to be scalable, in form of plug-ins and services. Even the interface allows the user to extend the GUI with extra controls.

An obvious future development would be to allow multiple simulation models to use the same interface. For example, multiple simulation models could use the ObjectBase interface, and be controlled through the Position Page.

One of the future developments that came up at the end of the project was to make the GUI extension with controls to happen with an XML file rather than in the models with C++. The user could declare their controls in an external XML file, which would be read by the service and thus added to the GCS GUI.

Another aspect that could be considered for future development was the map page. The map page could have buttons, which would change the satellite imagery to map imagery, such as in Google Maps.

These are just a couple of examples, but as mentioned above the possibility to extend the GCS was an integral part of the project, as such virtually limitless.

References

1. Sirius simulation framework user Manual. Cassidian; February 2013.

2. EADS at a glance.
URL: http://www.eads.com/eads/int/en/our-company/EADS-at-a-glance.html. Acessed 13 April 2013.

3. Arinic Report 610B Guidance for use of avionics equipment and software in simulators. Aeronautical Radio Inc; 2001.

4.Guidance, rationale, and interoperability manual for the real-time platform reference federation object model (RPR FOM). The Boeing Company; October 2003.

5. Eric Clayberg, Dan Rubel. Eclipse building commercial quality plug-ins. Addison-Wesley; 2008.

6. Vladimir Silva . Practical Eclipse rich client platform projects. Apress; 2009.

7. Lars Vogella. OSGI modularity tutorial. November 2012.
URL: http://www.vogella.com/articles/OSGi/article.html. Acessed 14 January 2013

8. Ozgur Akan. Why I chose SWT against Swing. November 2004.
URL: http://weblogs.java.net/blog/aiqa/archive/2004/11/why_i_choose_sw.html. Acessed 25 January 2013.

9. SWING & SWT.
URL:http://4.bp.blogspot.com/_OkKWVZHU4pc/S357HAyb8gI/AAAAAAAAAIc/2K8LHAJt0N0/s1600-h/figure_03.gif. Acessed 05 January 2013.

10. Lars Vogella. Eclipse Jface overview. January 2012.
URL: http://www.vogella.com/articles/EclipseJFace/article.html. Acessed 27 January 2013.

11. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose. Eclipse modeling framework. Addison-Wesley;2003.

12. The Eclipse foundation. Eclipse modeling framework project. URL: http://www.eclipse.org/modeling/emf/?project=emf#emf. Acessed 10 March 2013.

13. About GL Studio.
URL: http://www.disti.com/Products/glstudio/. Acessed 10 March 2013

14. Software of the year award: World Wind Java. December 2009.
URL: http://www.nasa.gov/offices/oce/appel/ask-academy/issues/volume2/AA_2-12_SF_soya.html. Acessed 29 January 2013.

15. Nasa World Wind.
http://www.nasa.gov/images/content/432280main_aa_2-12_wwj1_big.jpg. Acessed 07 January 2013.

Appendix 1: ObjectBase Attributes

ObjectBase.cpp

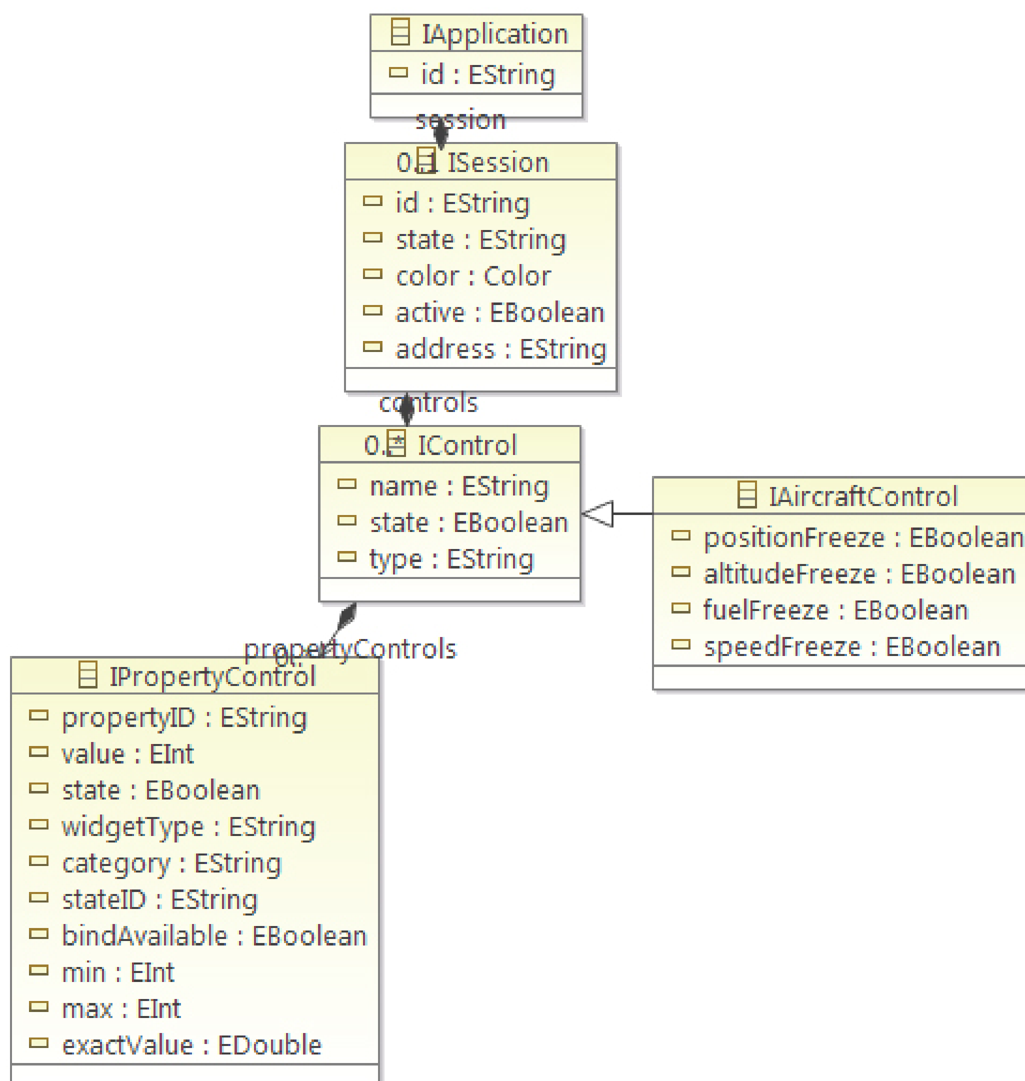
```
void ObjectBase::initComponent(String n, ObjectRef<Runtime> runtime)
{
        setType( "Component" );
        initModel(n,runtime);

        createComponentAttribute( "Total",false, "spinner","Fuel", 0, 10000);
        createComponentAttribute( "Intern",false, "spinner","Fuel", 0, 10000);
        createComponentAttribute( "Extern",false, "spinner","Fuel", 0, 10000);

        createComponentAttribute( "Position Freeze",false, "button" ,"Freeze", 0, 0);
        createComponentAttribute( "Altitude Freeze",false, "button" ,"Freeze", 0, 0);
        createComponentAttribute( "Fuel Freeze",false, "button" ,"Freeze", 0, 0);
        createComponentAttribute( "Speed Freeze",false, "button" ,"Freeze", 0, 0);



        createComponentAttribute( "Heading",false,"spinner","Position", -360, 360);
        createComponentAttribute( "Speed",false,"spinner","Position", 0, 3000);
        createComponentAttribute( "Roll",false,"spinner","Position", -360, 360);
        createComponentAttribute( "Pitch",false,"spinner","Position",-360, 360);
        createComponentAttribute( "Altitude",false,"spinner","Position", 0, 10000);
        createComponentAttribute( "Longitude",false,"spinner","Position", -180, 180);
        createComponentAttribute( "Latitude",false,"spinner","Position", -180, 180);



        createComponentAttribute( "Empty Mass" ,false,"spinner","Mass" , 0, 100000);
        createComponentAttribute( "Total Mass" ,false,"spinner","Mass" , 0, 100000);
        createComponentAttribute( "Pay Load Mass" ,false,"spinner","Mass" , 0, 100000);


}
```

Appendix 2: WeatherBase Attributes

WeatherBase.cpp

```
void WeatherBase::initComponent(String n, ObjectRef<Runtime> runtime)
{
	setType( "Weather");
	initModel(n,runtime);

	createComponentAttribute( "Snow",false, "scale","Snow/Rain");
	createComponentAttribute( "Rain",false, "scale","Snow/Rain");

	createComponentAttribute( "Fog Height",false,"scale","Ground Fog");
	createComponentAttribute( "Fog Visibility",false, "scale","Ground Fog");

	createComponentAttribute( "Clouds Visibility",false,"scale","Clouds/Visibility");
	createComponentAttribute( "Clouds Top" ,false, "scale","Clouds/Visibility");
	createComponentAttribute( "Cloud Bottom",false, "scale","Clouds/Visibility");

	createComponentAttribute( "Temparature Position" ,false,"scale","Temparature");
	createComponentAttribute( "Temparature Air",false, "scale","Temparature");
	createComponentAttribute( "Temparature Sea",false, "scale","Temparature");



}
```

Appendix 3: ISession Ecore Model



Appendix 4: Update Value

ComponentBase.cpp

```
void ComponentBase::update()
{
	for(int i = 0; i < compnentattributecount; i++)
	{
		ComponentAttribute* attribute = &componentAttributes[i];
		if(attribute->binded)
		{
			if(attribute->valueChanged())
			{
				service->sendToClient(getName());
				service->sendToClient(attribute->getID());
				service->sendToClient(Variant(attribute->getDoubleValue()));
			}
		}
	}
}
```

Appendix 5: General View

GeneralView.java

```
public class GeneralView extends AbstractView {
	public static final String ID= "com.sirius.gcs.core.gui.generalview" ;

	@Resource
	INetworkService networkService ;

	@Override
	public void createPartControl(Composite parent) {
	initView(parent, networkService );

	Image G_BAR = Activator. getImage (IconKeys. GENERAL_BAR );

	Label lblNewLabel = new Label(parent, SWT. NONE);
	lblNewLabel.setLayoutData( new GridData(SWT. LEFT, SWT. CENTER, false, false, 2,
	lblNewLabel.setImage(G_BAR);

	int controls =
	IApplication. instance .getSession().getControls().get(3).getPropertyControls().si
	for(int i = 0; i < controls; i++) {

createCategoryControls(IApplication. instance .getSession().getControls().get(3).getPro
		}


	}

	@Override
	public void setFocus() {
		// TODO Auto-generated method stub

	}


}
```