General Aviation Data Framework

Elaine M. Blount*
Unisys Corporation, NASA Langley Research Center, Hampton, VA, 23681

Victoria I. Chung†
Flight Simulation and Software Branch, NASA Langley Research Center, Hampton, VA, 23681

The Flight Research Services Directorate at the NASA Langley Research Center (LaRC) provides development and operations services associated with three general aviation (GA) aircraft used for research experiments. The GA aircraft includes a Cessna 206X Stationair, a Lancair Colombia 300X, and a Cirrus SR22X. Since 2004, the GA Data Framework software was designed and implemented to gather data from a varying set of hardware and software sources as well as enable transfer of the data to other computers or devices. The key requirements for the GA Data Framework software include platform independence, the ability to reuse the framework for different projects without changing the framework code, graphics display capabilities, and the ability to vary the interfaces and their performance. Data received from the various devices is stored in shared memory. This paper concentrates on the object oriented software design patterns within the General Aviation Data Framework, and how they enable the construction of project specific software without changing the base classes. The issues of platform independence and multithreading which enable interfaces to run at different frame rates are also discussed in this paper.

I. Introduction

THE Flight Research Services Directorate at the NASA Langley Research Center (LaRC) provides design, development, implementation, and testing services for simulation and flight aerospace experiments. This support enables researchers to develop and test research ideas to enhance aviation safety, aviation capacity, and the operational needs of the national airspace system[1]. FRSD develops, operates, and maintains three general aviation (GA) research airplanes: Cessna 206H Stationair, Lancair Columbia 300, and Cirrus SR22X in addition to several other types of airplanes to support flight experiments. FRSD designed and built a GA baseline research system for these GA research airplanes at LaRC. The Flight Simulation and Software Branch (FSSB) of the FRSD developed a Generic Aviation Data Framework for the GA baseline research system in order to support experiments performed in all three of these research GA airplanes. The Generic Aviation Data Framework is designed to operate on both Windows and Linux platforms to gather data from hardware devices for use by experimental equipment, data gathering and analysis, and graphics display.

The goal of the GA research system is "to provide a generic research system for the three NASA GA aircraft using as many common features/components as possible to minimize the specific hardware and software necessary for experiments envisioned within the next three to five years."[2] Various objectives within this goal include minimizing the costs and time in reconfiguring aircraft between experiments and the ability to use interchangeable research system components between the aircraft.[3] The original requirements included the hardware, software, and various other components of the planes. Figure 1 depicts the hardware research system components which consist of Air Data Attitude Heading Reference System (ADAHRS), Data Acquisition System, data link system, General Purpose Computers, Global Positioning System, and Universal Access Transceiver (UAT).

Derived requirements were written to detail software design goals that include the ability to vary the interfaces, use the software on both Windows and Linux platforms, use both mouse and bezel button inputs to communicate

* Software Engineer, Unisys Corporation, NASA Langley Research Center/Mail Stop 169, and AIAA Member
† Software Group Lead, FSSB, NASA Langley Research Center/Mail Stop 125B, and AIAA Lifetime Senior Member.

with the system, check the status of interfaces, view the data received by the various interfaces and display graphics.[4] The GA Data Framework was designed to meet the requirements of the GA research system.

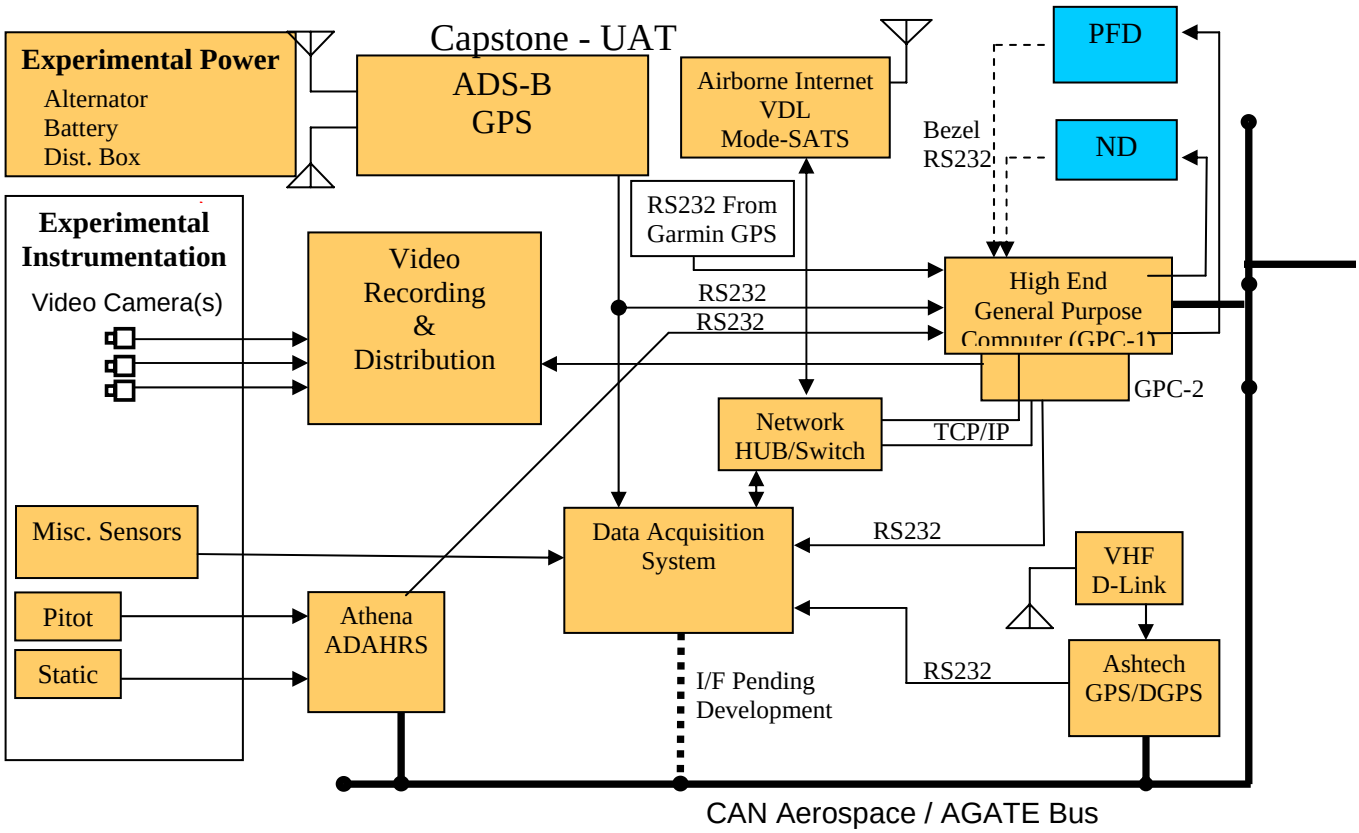

Figure 1. Research System with its Components

II. Overall Design

The GA Framework is written in object oriented design and C++ computer language. Framework code does not change from project to project. Projects consist of different experiments that require specific changes to the code that are not reusable for other experiments. The "framework" code is designed so that "project" code can be derived from key classes to make changes to the system. Figure 2 shows the general architecture of the various classes. The object ga_main instantiates GAMain; GAMain, in turn instantiates Shared Memory Classes for data storage, GAGui for a Graphical User Interface, and the various data interfaces (GADataInterface). Classes reused from the Langley Standard Real-time Simulation in C++ (LaSRS++) framework are implemented for shared memory, graphics, and the SeparateThreadGui[5] class. Classes executing as separate threads include the various interfaces, GAGui and GAMain. All of the Data Interfaces are derived from GADataInterface to ensure that they possess properties specific to GADataInterface. All of the GADataInterfaces are contained in a vector within GAMain after instantiation, and they are acted upon by GAMain in an iterative fashion using the GADataInterface virtual and non-virtual methods.

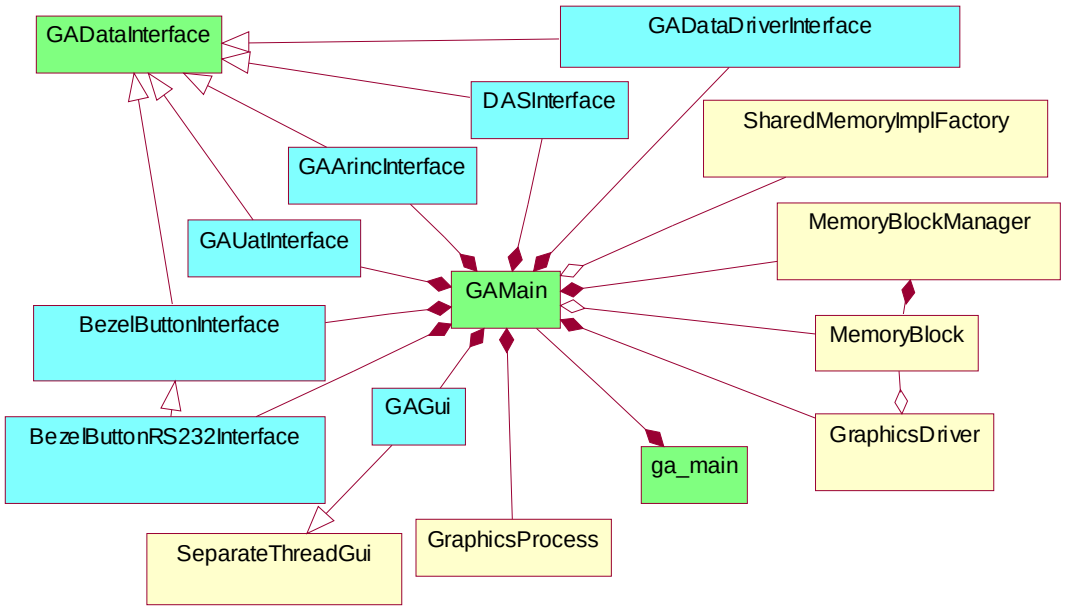

Figure 2. Overall Design of GA Data Framework

Instantiation of the General Aviation Software is straightforward as shown in Figure 3. Shared Memory items are created, Graphics Driver is instantiated, the interface initial conditions file is read and parsed to create the appropriate interfaces. The Graphics Process is executed as a separate process to display graphics, when graphics are requested. In addition to providing the different interfaces required by the system, the initial conditions file also provides the type of the plane to the GA Framework for setting plane specific variables.

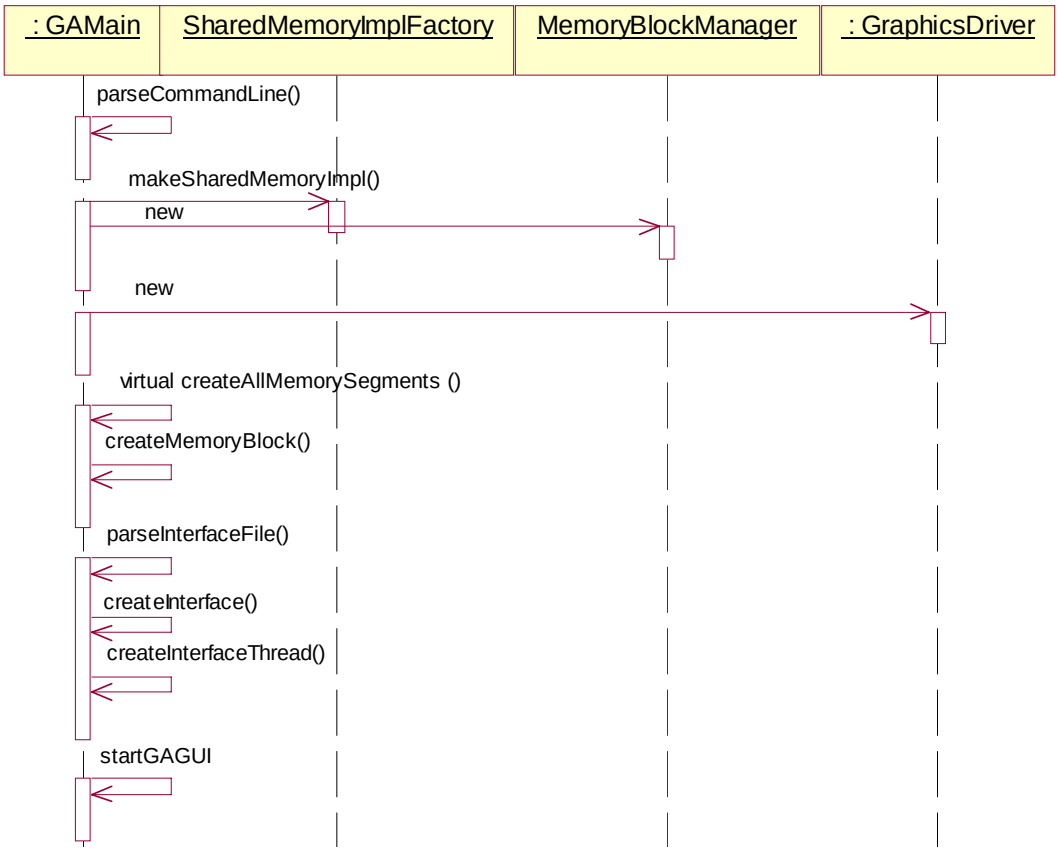

Figure 3. Instantiation of General Aviation Framework

A. General Interface Design and Multi-Threading

All data interfaces are derived from the class GADataInterface regardless of whether the interface receives data from a socket connection, ARINC board, serial connection, or calculates its own data. A data transfer rate of one hertz is specified during construction within the code according to the interface. GADataInterface polls the data transfer rate specified if the data rate is greater than zero. A data rate of zero tells the GADataInterface to perform blocked reads.

An example of the initial conditions file is as follows:

```
#Plane Types
#---------------
#CESSNA_206X_STATIONAIR PLANE
#LANCAIR_COLUMBIA_300X PLANE
CIRRUS_SR22X PLANE
#
#Interface                      com port (/dev/ttys#) – linux) (com# windows)
#
#TestDriverInterface           NA
#BezelButtonInterace           NA
#
#UATInterface                  /dev/ttyS0
#RS232BezelButtonInterface /dev/ttyS0
#AthenaAdahrsInterface         com1
#SeagullAdahrsInterface        /dev/ttyS0
#
#interface                     com port,  fileout/in       baud   hertz  record size
```

```
#SerialInputReceiveInterface com5       hardware.out 57600
#SerialInputSendInterface     com5       hardware.out 57600 2       83
```

Figure 4 below shows the general contents of a GADataInterface from which all interfaces are derived. GAMain has a static method updateMemory(void* raw_args) used to start threads which execute the GADataInterface virtual method updateSharedMemoryBlocks(). Every interface receives data and updates shared memory using updateSharedMemoryBlocks() and methods called within it. UpdateSharedMemoryBlocks() also keeps track of the time between reads to enable connection checks and to store frame rate statistics. Virtual method GADataInterface::makeDataInvalid() marks data received by the interface and stored in shared memory invalid for use when applicable. Each GADataInterface creates a standard GAAutoRaiseWindow as a stub to view data, but interfaces often create classes derived from GAAutoRaiseWindow to display the specific interface data.

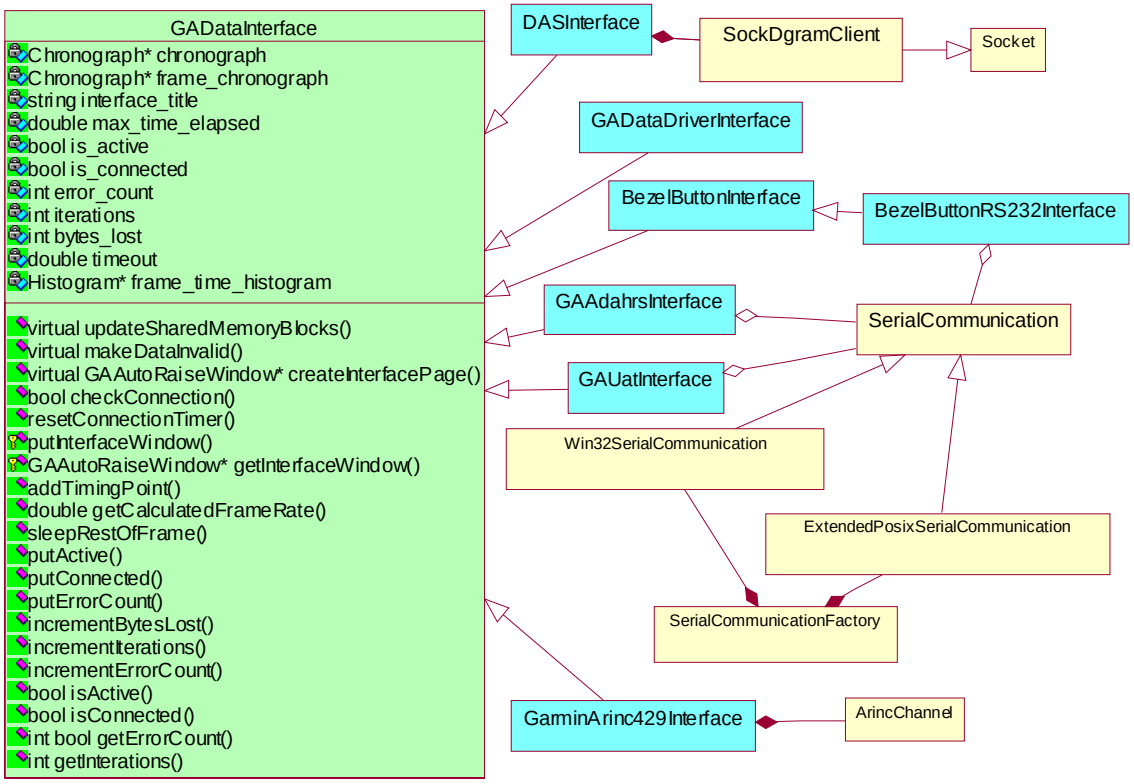

Figure 4. General Interface Design

As shown below in Figure 5, method GAMain::cycle() continuously executes to iteratively check the connections of the interfaces, mark the data invalid if the interface is not connected, send graphics information data via Ethernet to displays if the –ethernet_graphics option is selected, increment the timer and iterations, then sleep the rest of the time frame based on the data transfer rate selected for the GAMain class. Each of the interfaces operates on a separate thread according to desired options, including an individual data transfer rate specified upon interface creation. GAMain manages the interfaces by checking on each interface thread using the method GADataInterface::checkConnection(). GADataInterface::checkConnection() queries the time elapsed since the last update against the maximum time for that interface. If the maximum time has been breached, all of the data updated by the interface is set to invalid: it is too old to be used. During each cycle of update by the interfaces, using GADataInterface::updateSharedMemory() the data is set to valid. Thus, if the hardware is turned off, the timeout would be detected by checkConnection() which would set the data invalid. Once the hardware is turned on, the data

is updated and flagged as valid. During shutdown, GAMain sends messages to all of the interfaces to stop, and destroys all objects created.

During the design phase, there was concern about order of processes and how to prevent potential issues regarding process priority and competition for resources. Thus far, there have been no problems with processes competing for resources. Shared memory blocks are created to store data from specific interfaces. Processes do not update shared memory unless that memory block is created for them. All processes can read data from the shared memory blocks, but they only update their own memory blocks. Due to this design rule, GAMain does not currently use mutexes or locks to access the data in shared memory**.**

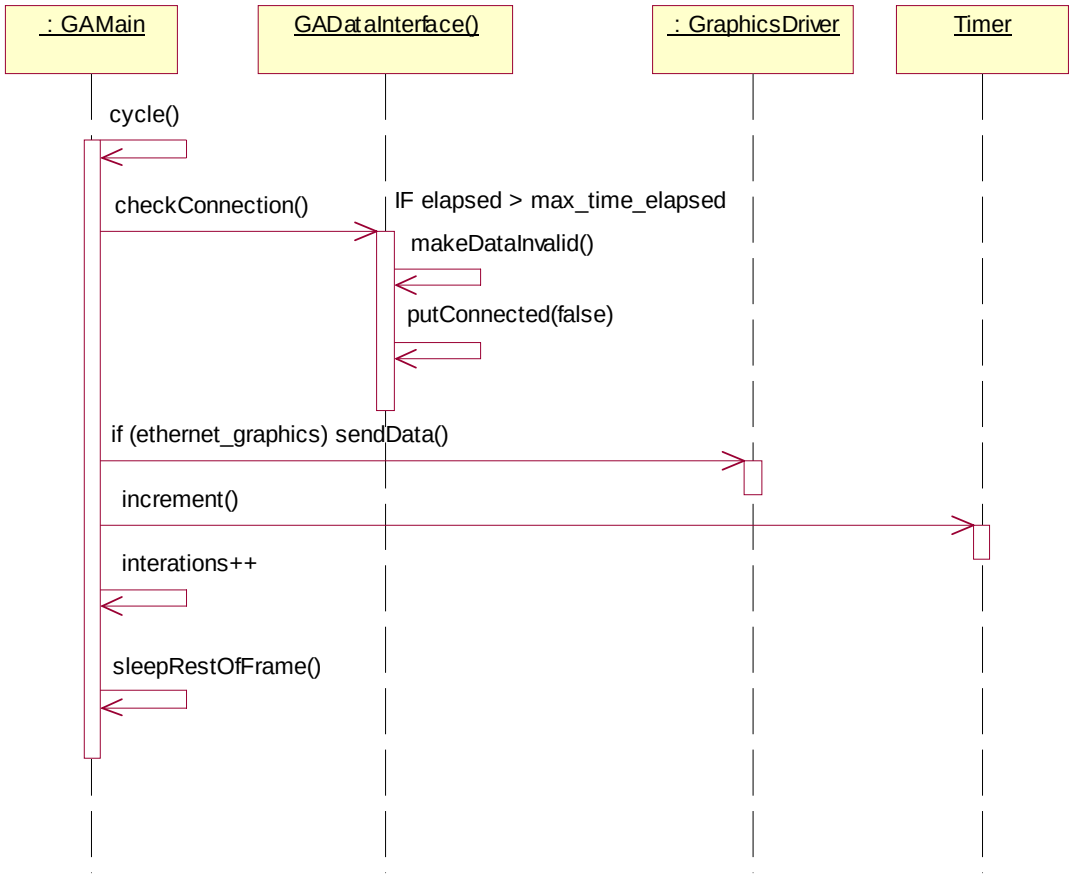

Figure 5. GAMain::cycle() Update for General Aviation Frame

B. GUI Design

The GUI screens shown in Figure 6 are designed to receive data through the class Bezel Buttons or a mouse and are created using GTK**.** GAGui is created using method GAMain::startGui() and is derived from the LaSRS++ class SeparateThreadGui shown in Figure 2. All of the GA GUI's inherit from GAAutoRaiseWindow (from LaSRS++) except for GABezelButtonWindow and GADataInterfaceGuiSection. The GAAutoRaiseWindow (derived from LaSRS++ GuiAutoRaiseWindow class), contains attributes and methods that enable the GUI's to keep track of the bezel button selected, the buttons displayed on the GUI, and various other features required by all of the GA GUI's. The classes filled in blue: DataSourceWindow, DiwplayControlWindow, GAMainWindow, and ViewMemoryWindow are all derived from GAAutoraiseWindow. The grey boxes containing GAManager, GAMainWindowBuilder, and GAViewMemoryBuilder are used to create project specific derived GUI classes when desired.

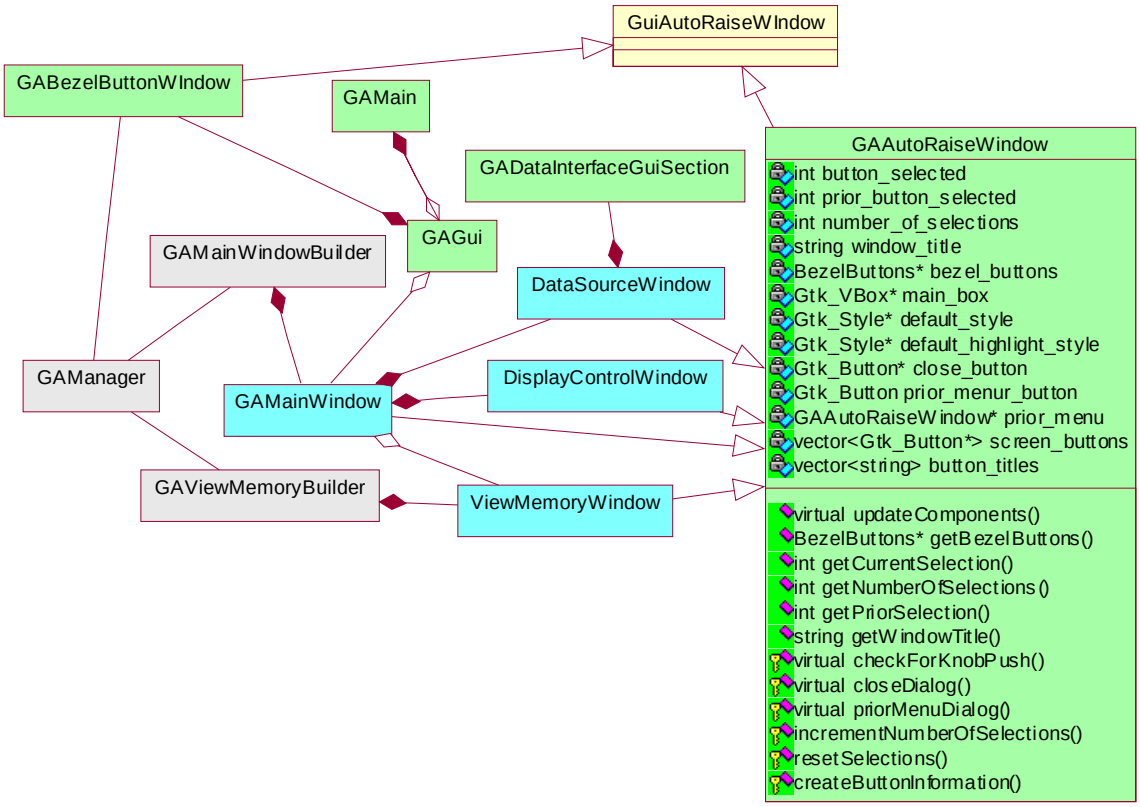

Figure 6. GUI Design

III. Project Code Derivation

Projects often require specific tasks of software necessitating the use of code that could function in an undesirable manner for other projects. The General Aviation Framework reuses some of the general patterns used by the LaSRS++ framework vehicles. Figure 7 shows GARegistration and GAManager along with three builders (GABuilder, GAMainWindowBuilder, GAViewMemoryBuilder) that were created to satisfy the need for project related code. The design shown is based on LaSRS++ method of creating project code for its framework. GAManager is a singleton[6] which is a design pattern according to objected oriented design methodology. GARegistration is a statically created class. Within the constructor for GARegistration, the standard builders for the GAFramework are created, and GAManager is instanced and stores the builders. The builders instantiate classes that are likely to change for projects. In this case, GABuilder creates GAMain, which creates and interacts with interfaces, GAViewMemoryBuilder creates a GUI for viewing memory blocks in shared memory, GAMainWindowBuilder creates the main GUI that may have different options added to perform project work. There is an equivalent class for GARegistration and the builders for each project. GAManager stores builders for all of the projects that have been created. The class G23Main is project software, and the classes used to create the G23 project software are G23Registration, G23ViewMemoryBuilder, G23MainWindowBuilder, and G23Builder. G23Registraion, like GARegistration is static and creates the builders and places them in GAManager. When the GA framework is started, a command line option specifies the project, which calls the correct builders from GAManager.

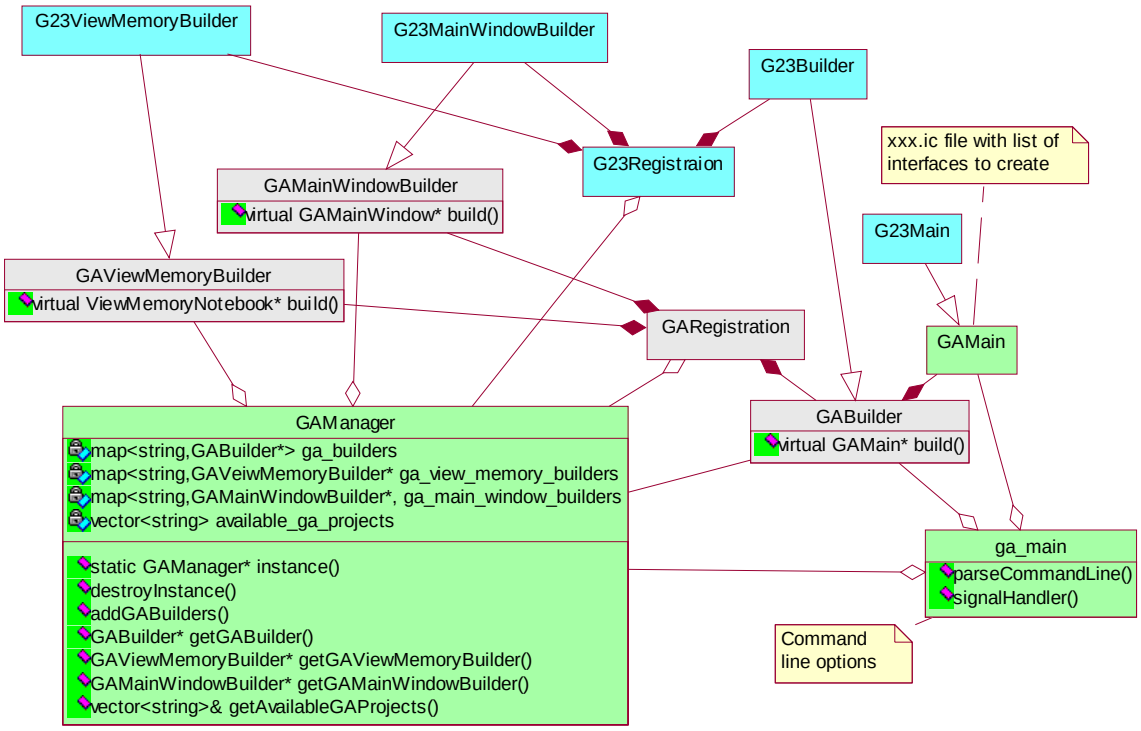

Figure 7. Project Code Deriviation

IV. Graphics Capabilities

The GA Framework has three different command line options for graphics: -nographics, -graphics, and –ethernet_graphics. When –nographics is used, graphics driver is still created, as it creates several needed shared memory blocks that are also commonly used by graphics classes to contain display information. When –graphics is used, the class ProcessHandler is created using the singleton factory ProcessHandlerFactory. ProcessHandler creates separate process and stores information needed to manage the processes. The graphics software is all written for LaSRS++ and reused within the GA Framework. The –ethernet_graphics option causes GraphicsDriver to open a socket connection to a specified host and send the graphics information via the socket. This enables the display of the graphics at a separate location form the computer running the GA Framework.

V. Platform Independence

Whenever possible, current LaSRS++ platform independent software was reused for GA Framework such as the software written for multi-threads[7] and shared memory[8]. Functions that interact with the operating system and are platform dependent are embedded in classes specific to the platform function and use a factory type of creation for serial connections, loading shared objects/dynamic linked libraries, and starting and checking on processes. These classes are written specifically for General Aviation, but with reuse in mind for other applications. All three have the same general pattern shown in Figure 8: a singleton pattern is used to create a factory[7], and the factory contains a method to create the platform specific implementation. When the GA Framework needs a Process Handler, the GA Framework instances the singleton ProcessHandlerFactory, and then calls makeProcessHandler() to get the correctly implemented ProcessHandler to create and manage a process. C++ #if defined statements which query standard LaSRS++ platform compile variables are used within ProcessHandlerFactory::makeProcessHandler() to ensure the correct ProcessHandler is instantiated.

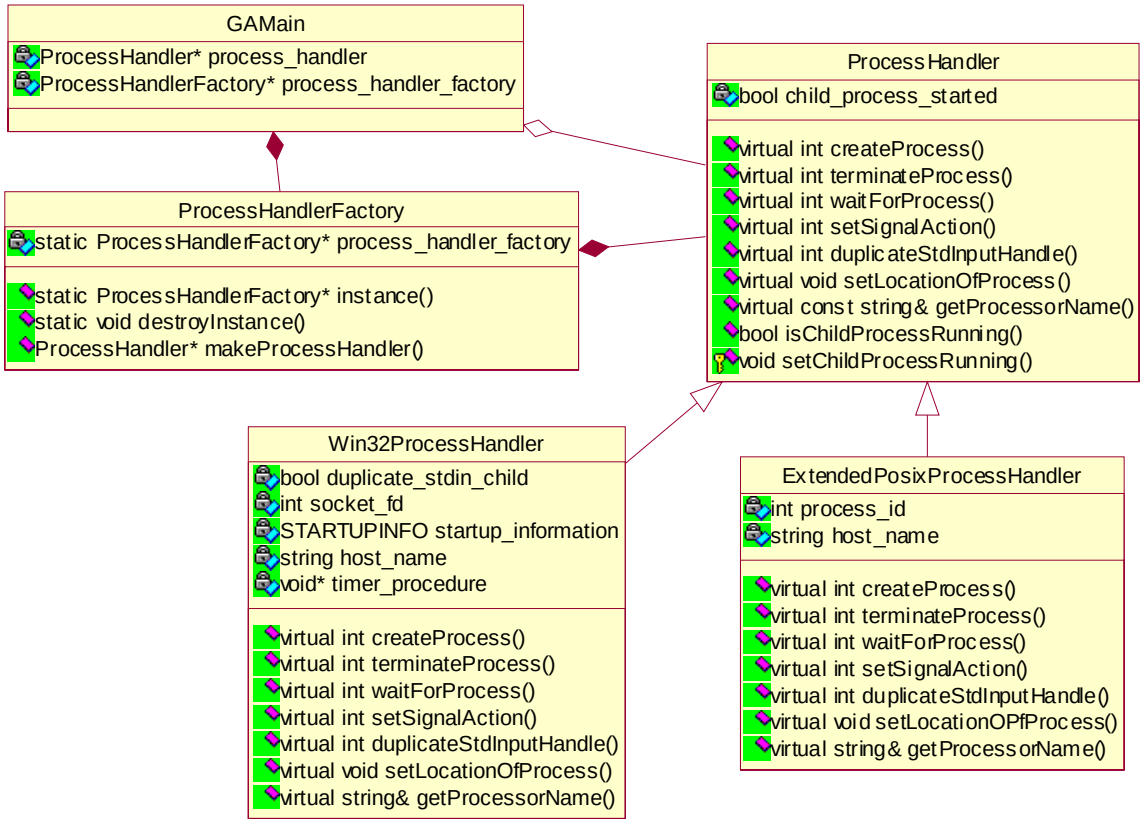

Figure 8. Platform Independent Classes

VI. Conclusion

The General Aviation Framework was successfully used for the Synthetic Vision Systems – General Aviation Equivalent Safety Experiment (SVS-GA-ESE) Project. Changes that were needed specifically for the SVS-GA-ESE Project, were implemented using builders to create the project code as described in Project Code Derivation section. The framework successfully reused many of the LaSRS++ components and patterns and generated new classes that can also be reused by the LaSRS++ framework. The goal and objective of having generic GA research software framework to support maximum hardware systems; minimizing the costs and time in reconfiguring aircraft between experiments; and the ability to use interchangeable components between the aircraft were being met successfully. The General Aviation Framework was designed with for long-term benefits of maximum re-usability, portability, testability, and maintainability through methodical, organized, and thoughtful planning, design, development, verification, and validation throughout the software lifecycle.

Acknowledgments

The authors would like to extend the appreciation and gratitude to Jim Barnes, Project Lead of the SVS-GA-ESE project, for his implementation of the GA Framework within the SVS-GA-ESE project; and Regina Tober, Wei Anderson, and Jerry Karwac for their contributions towards the General Aviation Framework and the SVS-GA-ESE project using the GA Data Framework.

The authors would like to also thank all fellow Flight Simulation and Sosftware Branch (FSSB) software developers as they continually maintain and evolve the LaSRS++ framework used by the simulation and flight projects.

References

[1]"Flight Research Services Directorate" [online website] URL:http://simulators.larc.nasa.gov/ [cited May 31, 2006].

[2]Fisher, Bruce D. and Knox, Charles E., "Baseline General Aviation Aircraft Research System Requirements Document," Version 2.2, September 12. 2003, page. 5, Internal NASA document

[3]Fisher, Bruce D. and Knox, Charles E., "Baseline General Aviation Aircraft Research System Requirements Document," Version 2.2, September 12. 2003, page. 5, Internal NASA document

[4]Blount, Elaine M., "Derived Requirements for Interfaces," Version 1.1, November 2, 2004, Internal FSSB document

[5]Geyer, David W, "The Use of Multiple Threads in An Object-Oriented Real-Time Simulation," AIAA-99-4338

[6]Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John, "Design Patterns,"

[7]Sugden, Paul C., Rau, Melissa A., "Platform-Independence And Scheduling in a Multi-Threaded Real-Time Simulation," AIAA-2001-4244

[8]Geyer, David W., Madden, Michael M, Glaab, Patricia C, Cunningham, Kevin, Kenney, P. Sean, and Leslie, Richard A, "Managing Shared Memory Spaces in an Object-Oriented Real-Time Simulation", AIAA-98-4532,.