CISC 322/326 Assignment 2: Concrete Architecture of FlightGear

March 24th, 2024

Group 3

Team Name: Project Flight Forge

Name & Student Numbers:

Lucas Coster (20223016)

Divaydeep Singh (20143859)

Antonio Sousa-Dias (20289507)

Alan Teng (20312270)

Donovan Bisson (20230334)

Zhongqi Xu (20200067)

Abstract

In the previous report, we discussed FlightGear's conceptual architecture in terms of high-level architecture, subcomponents, evolution, data flow, concurrency, use cases, and impact on the division of labor. We will build on the previous report, delve into the specific architecture of FlightGear and analyze the dependencies between the subsystems of the software using the Understand tool. Meanwhile, FDM is chosen as the research object in the subsystems. The team read FlightGear's documentation, as well as research papers, and books related to flight simulators, and used the Understand tool to analyze FlightGear's source code. In addition, two use cases are reworked and discussed based on our latest understanding and summarize both the limitations in the report and lessons learned from it.

1. Introduction

Originally released in 1996, FlightGear is a popular free and open-source flight simulator with many features, including real-time weather updates, third-party aircraft, terrain, traffic control systems, AI aircraft, and more. In this report, we delve into Flight Gear's concrete architecture based on the conceptual architecture discussed in Report 1. After summarizing, discussing, and using the Understand tool to analyze each subsystem, our group believes that the conceptual architecture we concluded in the report is not complete and accurate. In addition to the object-oriented style proposed, the FlightGear architecture also shows a repository-style, where multiple subsystems can store and retrieve data, the main code is responsible for organizing and passing controls to all the subsystems. However, in the actual architecture, subsystems also call each other for specific functions and sometimes for third-party software and utilities. Although differences from the conceptual architecture during actual development are to be expected, our team dug deeper into the reasons for these changes. FDM in subsystems was studied to understand more specifically FlightGear's concrete architecture. Based on what we learned while writing this report, we've revisited and modified the two use cases in report 1: launching a simulator & multiplayer session to better reflect FlightGear's concrete architecture and our understanding of it. Finally, we summarize the main findings and reflect on the problems encountered in writing the report and the study, as well as the limitations of the report and the lessons learned.

2. Conceptual Architecture

As mentioned above in the previous report the group focused on the conceptual architecture of FlightGear. After getting feedback and reviewing other groups' conceptual architectures we have devised the final conceptual architecture shown in Figure 1 below.

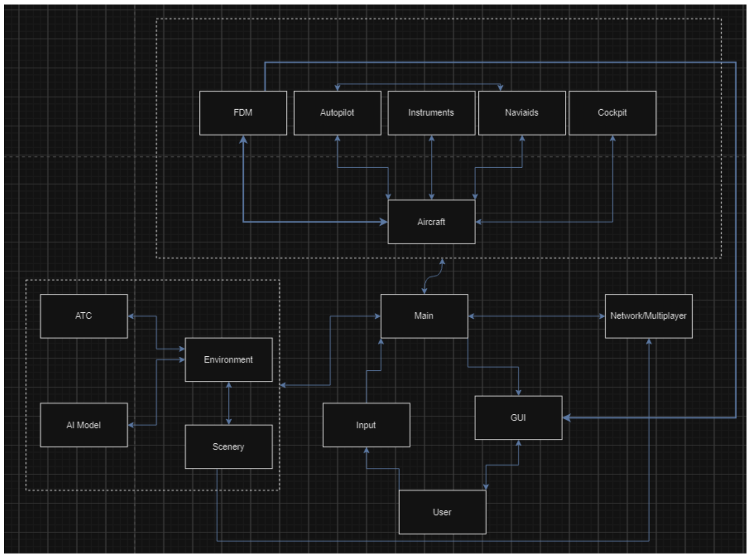

Figure 1: Updated Conceptual Architecture

This architecture will be compared with concrete architecture to investigate unexpected dependencies.

3. Derivation Process

Before beginning to write the report, each team member experimented and researched using a freely donated software —— Understand to ensure that each team member had experience using Understand and understood the concrete architecture of the software. Afterward, our group held a meeting to discuss the proper use of the Understand tool and our findings. The source code was then re-analyzed based on the results of the meeting. A visual of the high-level dependencies is shown in Figure 2.

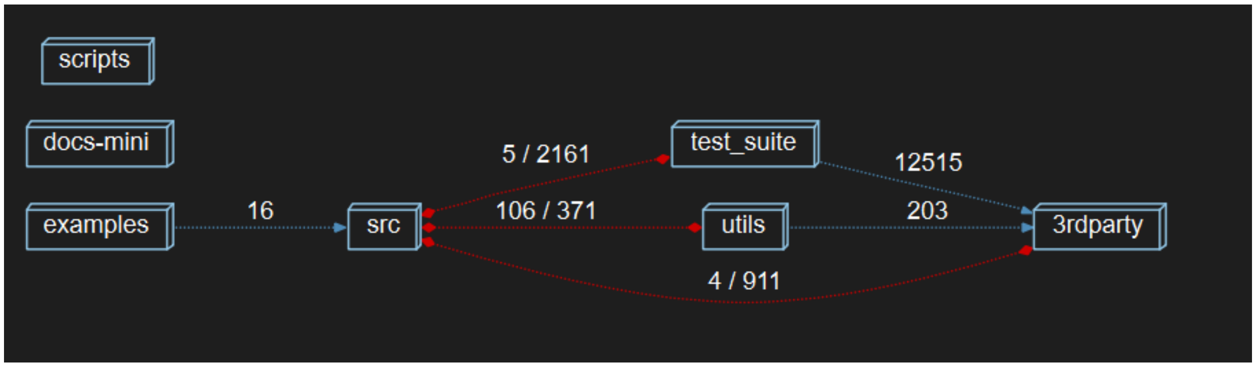

Figure 2: FlightGear Dependency Diagram

We analyzed the nested structure of the source code layer by layer, starting from the top, by looking at the folder names, reading the Readme file, and rechecking that the subsystems we defined in Report 1 were complete and determining their correspondence. The Understand visualization feature was then used to show the communication flows and dependencies between subsystems layer by layer starting from the top level down to individual files. For some files/folders that were vaguely named or named by acronyms, it was necessary to read the Wiki, and Readme files, and analyze the code to discover which subsystem they belonged to. Our group conducted a reflexive analysis of the high-level architecture and the specified subsystem (FDM) by comparing the conceptual architecture with the concrete architecture and found dependencies that were inconsistent with the conceptual architecture. Some depend on other subsystems and some on third-party libraries or software. The chosen subsystem (FDM) is also studied to derive its inner architecture and relationship with other components. Because of the clarity of FlightGear's architecture, our group was able to identify dependencies and arrive at the following concrete architecture for FlightGear.

4. Concrete Architecture

The Concrete Architecture for the FlightGear system as depicted through the Understand tool is shown in Figure 3 FlightGear architecture is a repository-style architecture where multiple subsystems can store and retrieve data. As seen in the image the multiple systems connect back to the main code which organizes operations and coordinates all the subsystems. All subsystems also have dependencies and interactions with each other which will be discussed in depth below. If we break down the top-level concrete architecture into subsystems, we can describe it with the 10 main subsystems listed below. The final subsystem is the main subsystem which acts as a central point for all subsystems, every system has some form of connection to this main and it is responsible for program initialization and logging saved data which is typical for a repository-style system.

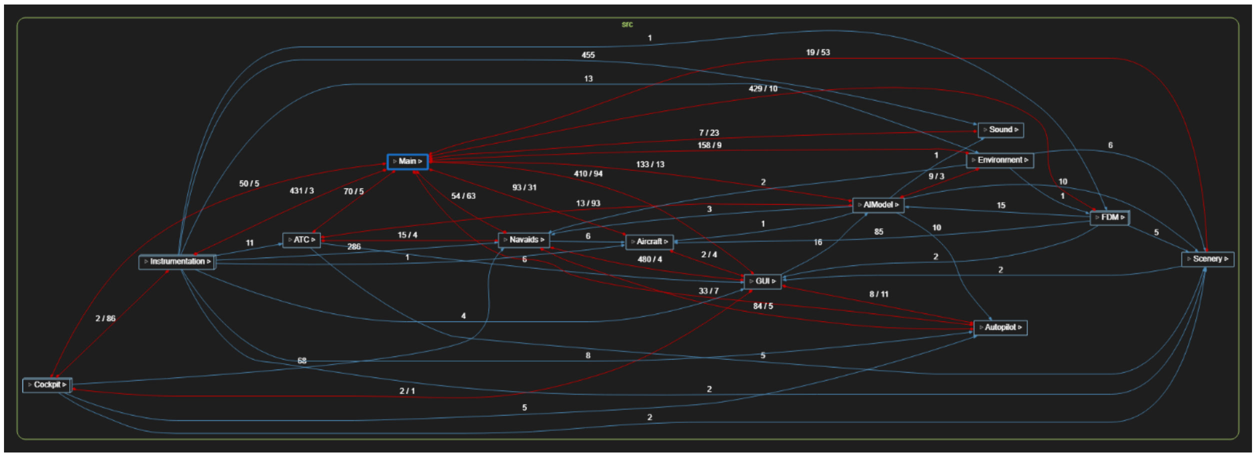

Figure 3: Concrete Architecture from Understand for Subsystems

4.1. Navaids

The Navaids subsystem contains information on routing, flight paths, and airways and is responsible for navigation while in simulation. It depends on the environment and scenery, AI Model, and Cockpit and Instrumentation subsystems. It takes information such as world build, and aircraft controls to define routing specifics and then communications with the GUI, Aircraft, and ATC subsystems to display information to the user and assist with the performance of other subsystems. The navaids main connection is to the GUI where it sends most of its information. This allows the user to acknowledge and update their routing information midflight with inputs into the system. Navaids also connects to the main subsystem where it grabs start-up and global data and is also able to save valuable data it acquires. Figure 9 in Appendix A – Understand Images shows the full Navaid Dependency graph.

4.2. Sound

Sound is the subsystem responsible for all audio in the FlightGear application. It contains files relating to Morse code, voice players, and more. It depends on the Cockpit and instrumentation subsystems and the AI Model subsystem. The only outward connection the system has is too main where it stores all information related to sound. Main is then able to save and apply this information when necessary, during run time. The sound subsystem is slightly smaller than several of the other systems but still plays a vital role in simulation emersion. The full Sound Dependency graph is shown in Figure 11 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for

Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.3. FDM

FDM or Flight Dynamics Model is a mathematical module in charge of controlling physics and their interactions with other objects in the simulation. The FDM subsystem itself contains a multitude of other subsystems which in tandem allow complex calculations to be completed and used by other modules. FDM gathers most of its information from external tools, gaining constants and values from backup utilities and test suites. It also takes information from both the Environment and Scenery and Cockpit and Instrumentation modules which act as variables for certain calculations. FDM then takes its calculated values and sends them off to the GUI, Aircraft, and Scenery modules which will use the calculations in certain visual and simulation-based ways to build the most realistic simulation possible for the user. The full FDM Dependency graph is shown in Figure 12 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.4. Cockpit and Instruments

The Cockpit and Instrumentation is a subsystem responsible for all controls, instrument displays, and visual items in the airplane's cockpit. It contains information on the specific dials and items that should be displayed and how their adjustment will affect other modules. The only thing this module depends on is the GUI, this is because all information coming into the subsystem is based on user input which is handled by the GUI. The module takes in the user's interactions with the cockpit and instruments and sends that information to all the other subsystems, making it one of the most connected. Figure 12 in Appendix A – Understand Images shows the full Navaid Dependency graph.

4.5. Aircraft

The Aircraft subsystem is a smaller module with many files containing details on the aircraft. It organizes and handles all different models of aircraft along with flight performance and history keeping track of statistics and replays. The Aircraft module depends on the FDM, Cockpit and Instrumentation, and the AI Model subsystems as they allow it to understand specifics about the flight, what controls are specific to that aircraft, and how the aircraft should react to certain conditions. The module also has a close relationship with the GUI, a lot of user input and output directly relies on the Aircraft and therefore this connection allows both subsystems to complete their tasks. The full Navaid Dependency graph is shown in Figure 14 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.6. GUI

This module is responsible for all communication from the backend program to the front-end user application. It handles all visual and graphical representations and is a large module containing various files. GUI has dependencies on FDM, Environment and Scenery, and Cockpit and Instrumentation. These modules are what the GUI uses to display user output such as visual scenery, all cockpit controls, and specifics on the flight patterns. The GUI also must handle user input and these relationships can be seen from the dependencies exiting the module. The modules that depend on the GUI are the Aircraft, the Cockpit and Instrumentation, the Autopilot, and Navaid. The large number of dependent modules is because so many systems need information from user input. The full Navaid Dependency graph is shown in Figure 15 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.7. Autopilot

The Autopilot module is responsible for the autopilot control of the aircraft. If the user puts the plane into autopilot mode, the files within this module take control. It includes information on prediction, PLD controllers, and autopilot execution itself. The subsystem depends on the Cockpit and instrumentation module and the AI Model. The AI model controls flight and decisions made by autopilot and the Cockpit passes along information on the specific aircraft. The Autopilot system gathers this information and then sends it to the GUI for user output, the main file for storage and the Navaid module to update routing information. Autopilot has most of its dependencies connected to Navaid which makes sense conceptually. Autopilot needs information relating to aircraft heading and destination to properly pilot the plane and it grabs this information from the Navaid module. The full Navaid Dependency graph is shown in Figure 16 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.8. AI Model

This subsystem oversees all things within the simulation relating to artificial intelligence. Whether that is NPC planes or vehicles, or generated flight plans or dialogue this module connects and controls all non-player inputs and actions. Due to its high importance, it has connections to any modules that interact with AI components, which is most of them. The main dependency of the module is the GUI as it still requires user input to understand computer interaction. It sends its information to all other modules but has the closest connection with the ATC. The full Navaid Dependency graph is shown in Figure 17 in [4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images.

4.9. Air Traffic Control

ATC or Air Traffic Control is what acts as your communication with the ground during simulation. It informs you of weather updates, traffic control, and runway information. It is fully AI-controlled and therefore has a large dependency on the AI model. It also takes in information from the instrument module relating to sending down updates on timing and requesting communication. It sends information out to the Scenery module as well as the GUI using both to display visuals for simulation. Figure 18 in Appendix A – Understand Images shows the full Navaid Dependency graph.

4.10. Scenery and Environment

The final subsystem in our concrete architecture controls all the environmental features that are included in the simulation such as landscape. It has dependencies on nearly every other subsystem, the only two it does not rely on are Sound and Aircraft. After creating all the visuals, it sends information to the GUI and the Navaid systems allowing the user to see the environment created for the navigation to understand land mass and world arrangement. Figure 19 in Appendix A – Understand Images shows the full Navaid Dependency graph.

5. Subsystem Review: Flight dynamics model (FDM)

The flight dynamics model is a critical component of FlightGear. At its core, FDM is a set of mathematical equations used to calculate the physical forces acting on a simulated aircraft, such as thrust, lift, and drag [1]. These equations consider the aircraft's geometry, mass properties as well as atmospheric conditions to predict its behavior in response to control inputs and environmental factors. Every simulated aircraft in FlightGear must use one of the supported FDMs. To begin understanding the FDM architecture we created a conceptual diagram in Figure 4 below.

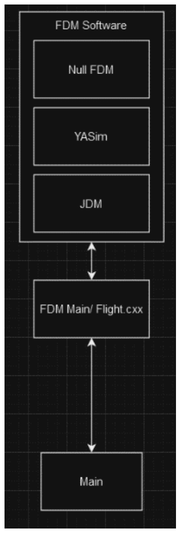

Figure 4: FDM Conceptual Architecture

Currently, FlightGear comes with these integrated FDMs:

1. JSBSim: An advanced, object-oriented FDM that models the aerodynamic and propulsive forces and moments as a function of the flight conditions and aircraft's state [2]. It is highly configurable and supports a wide range of aircraft including gliders and passenger aircraft. JSBSim can be configured to run either as a standalone simulation engine or as a part of FlighGear and offers the highest level of detail and flexibility, making it suitable for research, education, and training applications.
2. YASim: This FDM is designed to be more accessible to users without extensive aerodynamics knowledge. It uses a blade element theory approach to calculate aerodynamic properties based on the aircraft's physical configuration and flight conditions. YASim is particularly useful for simulating aircraft with complex or unconventional geometries and provides a good balance between accessibility and realism making it ideal for hobbyists and designers for testing conceptual aircraft designs.

For this assignment, JSBSim has been selected for its architecture as it is currently the most widely used FDM in the industry and by hobbyists.

JSBSim Flight Dynamics Model

JSBSim is an open-source FDM software library that is written in C++ and uses XML configuration files to model the dynamics of an aerospace vehicle. The library has been incorporated into the FlightGear simulation package and follows an object-oriented architectural approach. FDM is divided into separate components, each responsible for a distinct function or a set of related functionalities. In the context of JSBSim, these components include the aerodynamics module, propulsion module, flight control system module, and the simulation engine itself [3].

JSBSim consists of approximately 70 C++ classes and is distributed into subdirectories based on function. These include math classes, classes that aid input/output and initialization, model classes, and basic classes.

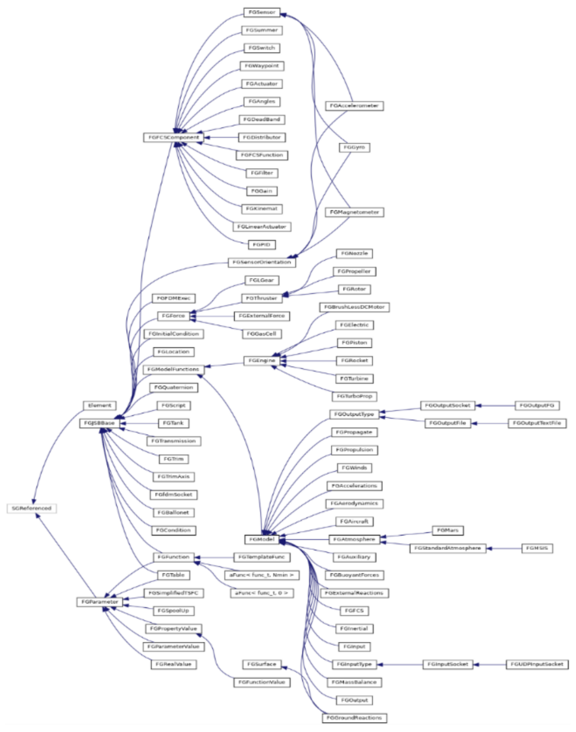

Figure 5: JSBSim Conceptual Class Graph

As shown in the figure, at the core of JSBSim's object-oriented design is a set of base classes from which more specific model classes derive, showcasing key inheritance relationships that facilitate code reuse and modularity.

The integration between JSBSim and FlightGear is facilitated through a series of interfaces and data exchanges that allow the two systems to communicate effectively. These include:

1. Simulation Executive: At the core of the integration is the JSBSim executive class (FGFDMExec), which orchestrates the flight dynamics simulation. This class is instantiated by FlightGear to integrate JSBSim into the FlightGear environment [3].
2. Aircraft configuration files: JSBSim uses XML configuration files to define the physical characteristics of the aircraft being simulated. FlightGear accesses these files to load the specific aircraft model within the simulation.
3. Data Exchange Protocol: JSBSim and FlightGear communicate through a data exchange protocol that transfers simulation state variables (such as position, orientation, velocity, and forces) from JSBSim to FlightGear, and control inputs (like control surface positions and throttle settings) from FlightGear to JSBSim. This bidirectional data flow ensures that the flight dynamics and visual representation remain synchronized.
4. Environmental Data: FlightGear provides JSBSim with environmental data, such as atmospheric conditions, wind, and temperature, which can affect flight dynamics. JSBSim incorporates this data into its calculations to ensure that the simulated aircraft responds realistically to the environment.

6. Reflexion Analysis

6.1. High-Level Analysis

In the last report, the FlightGear conceptual architecture was discussed and diagramed. The structure of this conceptual architecture is seen from the blue lines in the diagram below. Concrete architecture has shown us many unexpected dependencies which we call divergences. Each divergence has a rationale behind the cause and several of them will be examined below. An arrow from A → B shows that A depends *on* B.

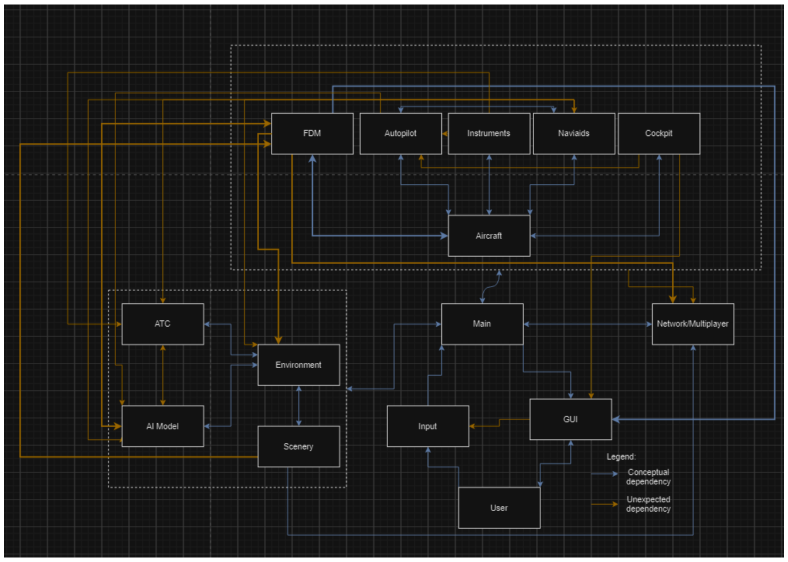

Figure 6: Reflexion Model Image

Cockpit → GUI

Cockpit uses GUI code to render the panels since it has the same functionality. This is done for code reusability. Similarly, the input system requires GUI to link the user to whatever graphical display they are trying to interact with. This is more justifiable, but there can be an argument for merging the two systems since both systems are done to interact with the user. The direct connection can be rationalized from this reusability factor. Rather than rewrite the functions in both locations the code is called from Cockpit to the GUI saving work.

Environment → FDM

FDM models aircraft performance based on environmental data such as wind speed, direction, and weather conditions. Similarly, the environment manages changes caused by the aircraft in its immediate vicinity. A direct link between the environment and FDM subsystems can streamline the bidirectional transfer of data and ensure simulation integrity while maintaining realism.

Autopilot → Cockpit

The autopilot module has an unexpected dependency on the cockpit and instrumentation subsystem e. The 1 one unexpected dependency that the autopilot system has is the NavDisplay file which is included in the Cockpit system and is responsible for displaying information related to routing and navigation. This dependency can be rationalized by the relationship autopilot has with the GUI. To display information about the aircraft's flight statistics must understand the navigation display, making this dependency necessary.

ATC → AIModel

The ATC or air traffic control module has a few unexpected dependencies on the AIModel. The dependencies include functions in the AIAircraft section of the ATC and parts of the ATC manager. The dependency is due to the fact the ATC is what handles communication between the user's simulated aircraft and other AI aircraft which are non-player controlled. The unexpected dependency can be rationalized by the fake the ATC is reusing and referencing AI code that had already been used in the main AI model. These connections are explainable and required for the concrete architecture to function correctly.

Scenery → FDM

FDM utilizes data from scenery files to calculate the aircraft's response to external factors including airport terrain, elevation, and runway length. This data is used to provide a realistic simulation environment as the FDM requires a range of data to calculate aircraft range and response to terrain conditions. The direct connection between the two subsystems is required to ensure low latency during data transmission by ensuring the FDM can continuously request data without relying on a different module.

6.2. FDM Subsystem Analysis

The FDM subsystem that was reviewed before also has a few unexpected dependencies when comparing the conceptual architecture as shown before and the full concrete architecture in Figure 7 below.

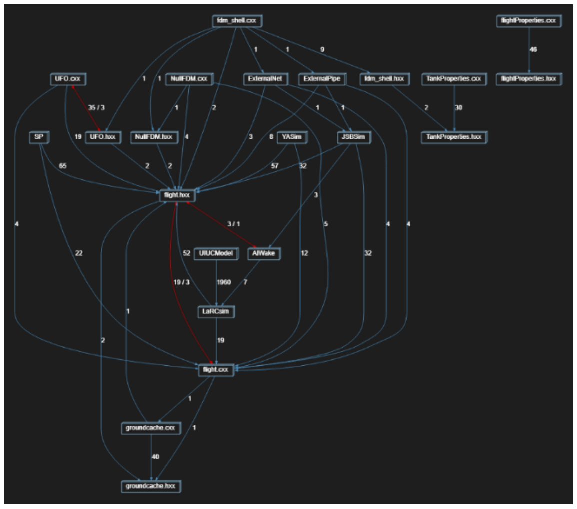

Figure 7: FDM Concrete Architecture Internal Dependency Structure

When we compare this concrete architecture with our proposed conceptual architecture, we can see several discrepancies or unexpected dependencies. We will analyze some of these dependencies and give a rationale for why dependency is needed. An arrow from A → B shows that A depends *on* B.

Flight → AIWake

The first unexpected dependency that the group noticed within the concrete architecture was the Flight file's dependency on the AIWake system. AIWake oversees meshing details within the flight and defines how the aircraft and the wake should mesh. The Flight module has one dependency on the AIWake subsystem, specifically including the AIWakeGroup type from the wakeType file. This dependency can be rationalized by understanding the main flight file requires objects of the AIWakeGroup type and therefore requires the type of definition from AIWake. This dependency reduces the amount of code that is needed and therefore speeds up operation.

Flight → FDM_Shell

Another discrepancy that the group noticed was the one that Flight had on the FDM_Shell. The FDM_Shell subsystem is a collection of cpp and hpp files that relate to flight and tank properties. The shell is what takes this information and connects it to the external pipe. We assumed that this pipe was the only connection that the FDM_shell would have to the rest of the architecture but there was one dependency from the main Flight file. The dependency is a specific interface type FGInterface which is used to capture the primary flight state. The interface defines the class and is required within the main Flight file at multiple points. The direct dependency is a rational decision allowing for the Flight file to have access to this necessary interface.

Flight → GroundCache

The final unexpected dependency that the group investigated was the Flight file's dependency on a GroundCache type. The one dependency was for the FGGroundCache type which handles the relation between the physics of the air and the ground itself. It caches information on ground location, behavioral interactions, and more. The groundCache module is an entirely separate module from the main flight and this is why the dependency was so unexpected. The rationale behind this dependency is that the ground cache object is at one point necessary for the execution of something within the main Flight file and it was easier to call back a reference to this object than create a new one in the header file. This dependency therefore saved on code size but made the dependencies slightly more complicated.

7. Use Cases/Sequence Diagrams

Use Case #1 (Launching a Simulation):

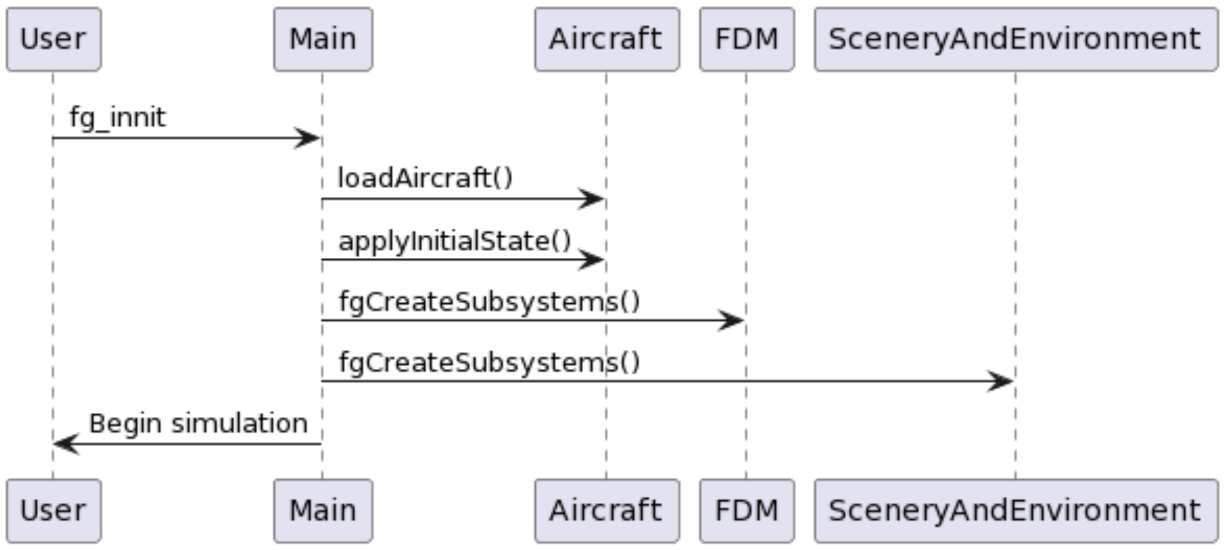

Figure 8: Sequence Diagram for Launching

The use case of launching a simulation begins within the fg_innit method which when triggered by the user in the main subsystem, sets off a series of initialization routines to begin the simulation. This process begins with initializing the aircraft module through the loadAircraft and applyInitialState methods. There is a large method called fgCreateSubsystems which triggers the initialization routine for the remaining subsystems. If a third party wishes to add a subsystem, this is where that subsystem should be initialized.

Use Case #2 (Multiplayer Session):

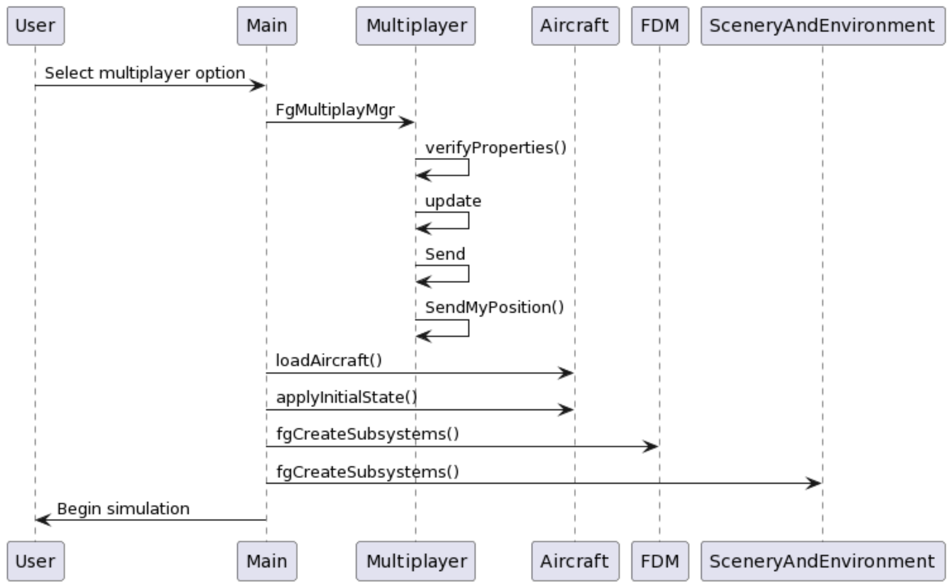

Figure 9: Sequence Diagram for Multiplayer

A multiplayer session begins with the main module calling FgMultiplayerMgr when init is called, which constructs a multiplayer submodule to deal with connection. The method verifyProperties is called, followed by an update, send, and SendMyPosition, which updates the other participants in the session. The initialization carries on as in use case #1, with the additional submodule, multiplayer, handling the synchronization of data.

8. Lessons Learned

Through the process of identifying the concrete architecture for the FlightGear application, the team has learned several valuable lessons that have allowed us to better understand conceptual architecture itself. The first big lesson was when we loaded up the Understand tool the first time and saw just how big and complex the code base was. We learned that there are often large discrepancies between conceptual and concrete architecture. The way code is visualized and the way it is implemented in practice can be very different leading to complex dependencies and difficult to explain code. Another major lesson was how messy open-source code can be. FlightGear is an open-source project which often means a lack of large-scale founding, no centralized coding style, and a decentralized development process. For these reasons, we learned how important it was to fully deep dive into the documentation and understand the meaning behind every connection because very little was made intuitively.

9. Limitation & Conclusion

8.1 Limitation

During the study, our team encountered several problems, most were resolved through our efforts, but this report may still contain omissions.

FlightGear has been in development for almost 30 years, since 1996. There is a large amount of code in the codebase that has been submitted at different times and by different contributors. The names of files or folders are not always clear. Distinguishing where they belong, whether they are part of the current release or deprecated code requires reading the documentation, or the wiki, to get a deeper understanding, but some of the documentation isn't very specific or direct. Therefore, the team had to use multiple subsystems' files to refer to each other to reach a conclusion, which may result in some files' attributes inaccurately.

Understand is closed-source software, although it has been iterated and tested over a very long period, the research team could not be sure of its reliability given the complex structure of FlightGear. Where possible, we would like to repeat the analysis of FlightGear using multiple software and draw more reliable conclusions based on the results.

8.2 Conclusion

In this report, our team has researched the FlightGear concrete architecture through the FlightGear Wiki, source code, and using the Understand tool. We have found that FlightGear is a repository style, which is different from the assumptions made in the previous report. We studied the subsystems in the concrete architecture and analyzed the dependencies of each subsystem through Reflexion analysis, and among the subsystems, we chose FDM for an in-depth study and analyzed it using the same methodology. During the study, we found the difference between the concrete architecture and conceptual architecture of the software, and all the above are fully discussed in the report.

References

[1] "Flight Dynamics Model - FlightGear wiki." Accessed: Feb. 16, 2024. [Online]. Available: https://wiki.flightgear.org/Flight_Dynamics_Model

[2] "JSBSim - FlightGear wiki." Accessed: Mar. 22, 2024. [Online]. Available: https://wiki.flightgear.org/JSBSim

[3] "JSBSimReferenceManual.pdf." Accessed: Mar. 22, 2024. [Online]. Available: https://jsbsim.sourceforge.net/JSBSimReferenceManual.pdf

[4] Papadakis, S., & Meletiou, G. (2016). "Development of a FlightGear-Based Simulator for Unmanned Aerial Vehicle Control and Mission Planning." In 2016 24th Mediterranean Conference on Control and Automation (MED) (pp. 1333-1338). IEEE.

[5] Penn, B., & Vroom, J. (2013). "FlightGear Flight Simulator: Free Open-Source Multiplatform Flight Simulation." In AIAA Modeling and Simulation Technologies Conference (p. 4315). American Institute of Aeronautics and Astronautics.

[6] Mungall, C., & Ross, J. C. (2004). "FlightGear: An Open-Source Flight Simulator Supporting Research." In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (p. 6079). American Institute of Aeronautics and Astronautics.

[7] Barba, L. A., & Givi, P. (2012). "Use of the FlightGear flight simulator as a teaching tool in aeroacoustics." In 40th AIAA Aerospace Sciences Meeting & Exhibit (p. 1014). American Institute of Aeronautics and Astronautics.

[8] Koenig, M., & Baldwin, T. (2010). "Advanced flight simulation with FlightGear." In IEEE Aerospace Conference (pp. 1-11). IEEE.

Appendix A – Understand Images

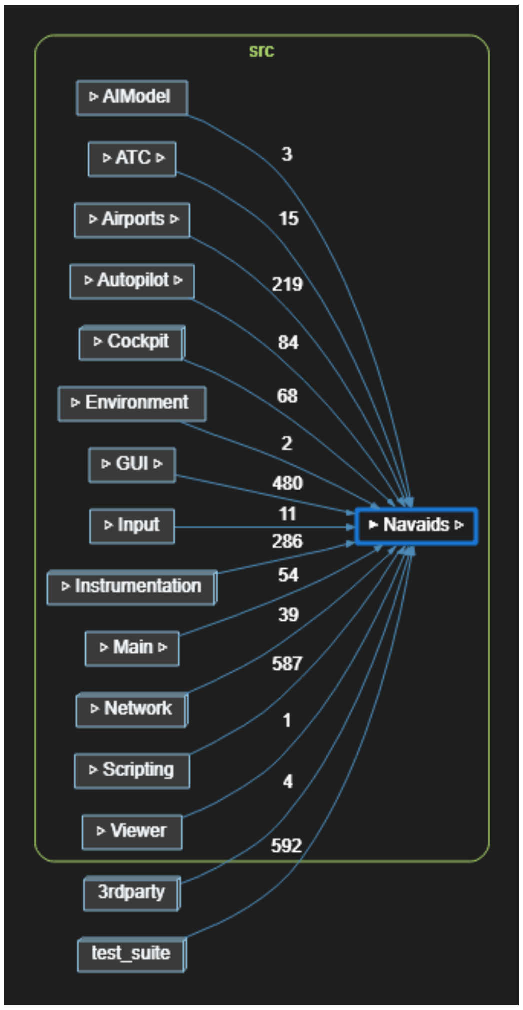

Figure 10: Navaids Dependencies

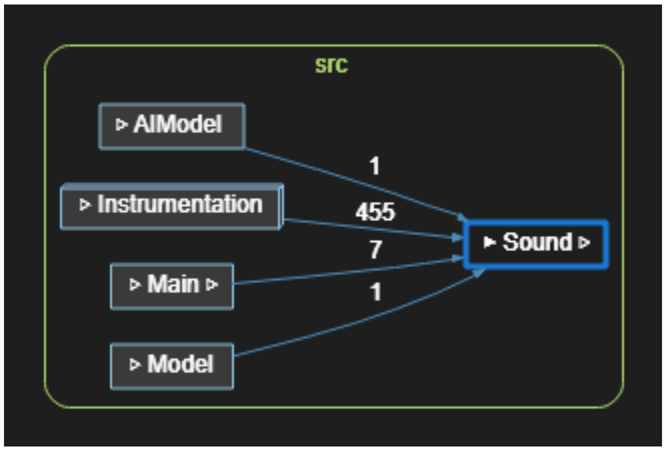

Figure 11: Sound Dependencies

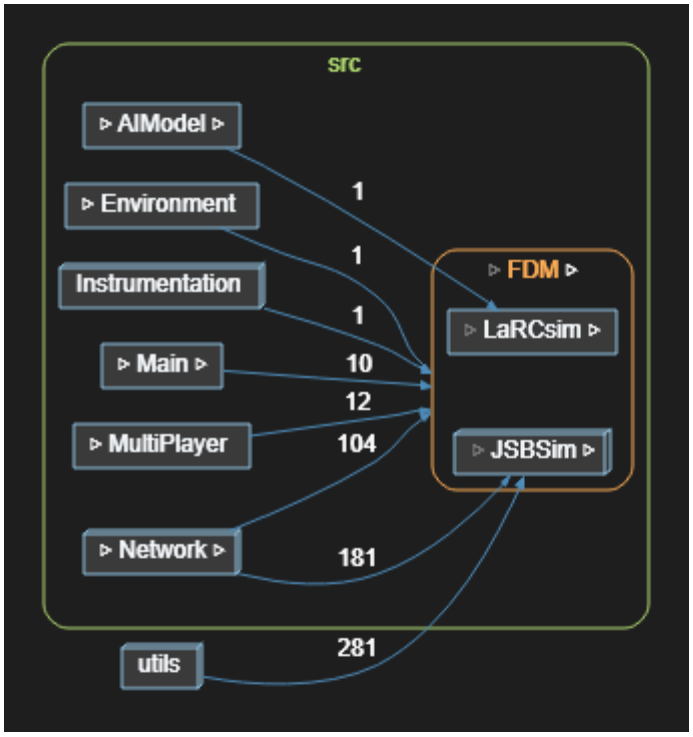

Figure 12: FDM Dependencies

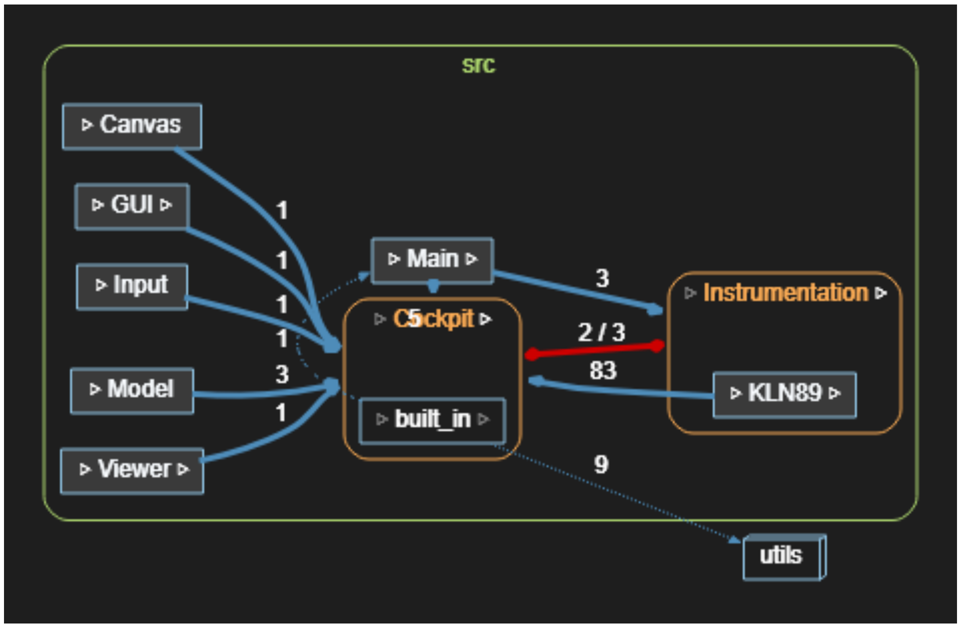

Figure 13: Cockpit and Instrumentation Dependencies

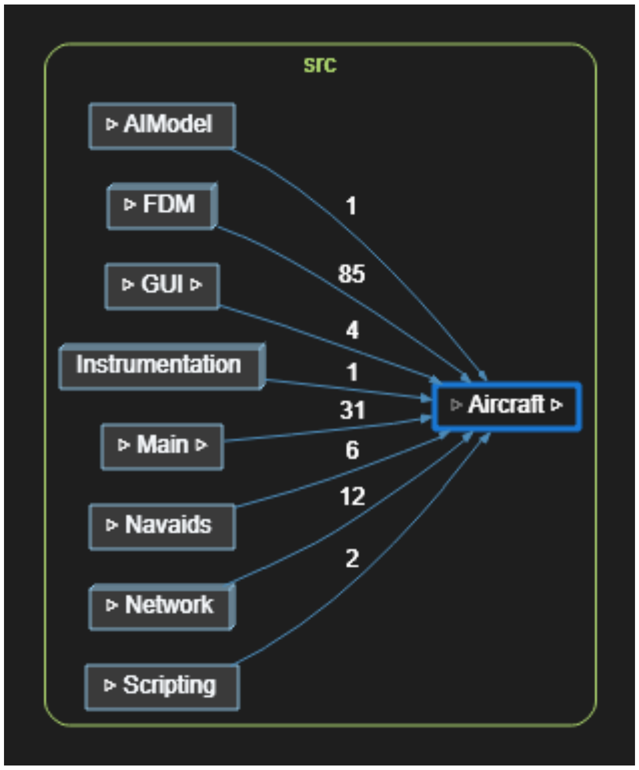

Figure 14: Aircraft Dependencies

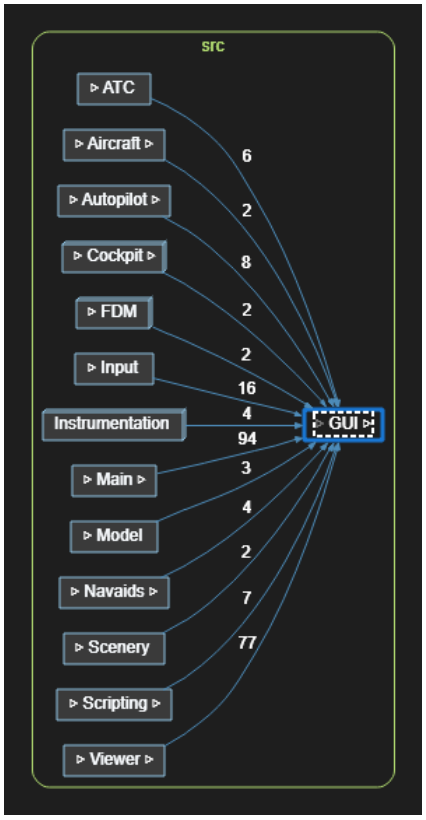

Figure 15: GUI Dependencies

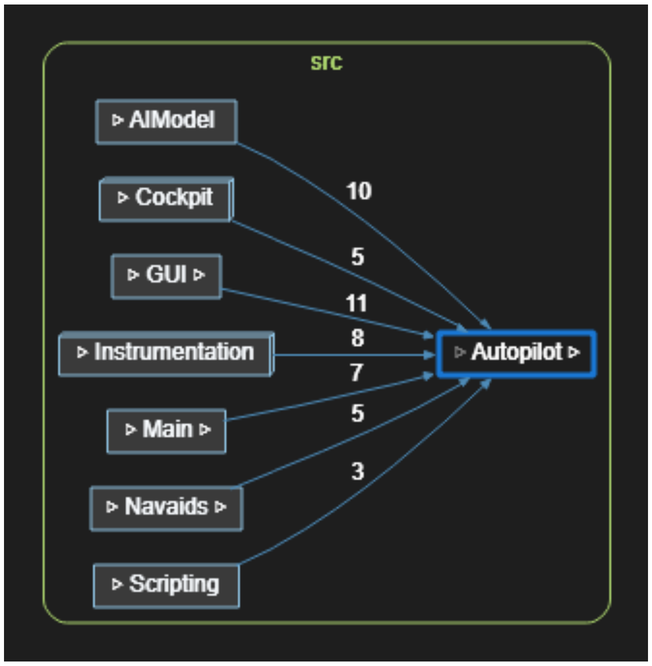

Figure 16: Autopilot Dependencies

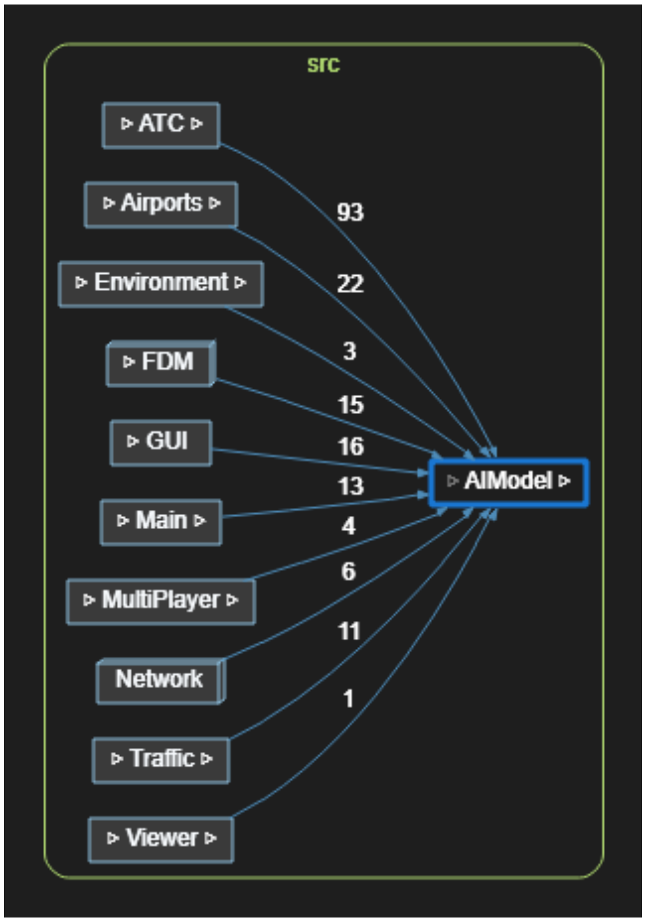

Figure 17: AIModel Dependencies

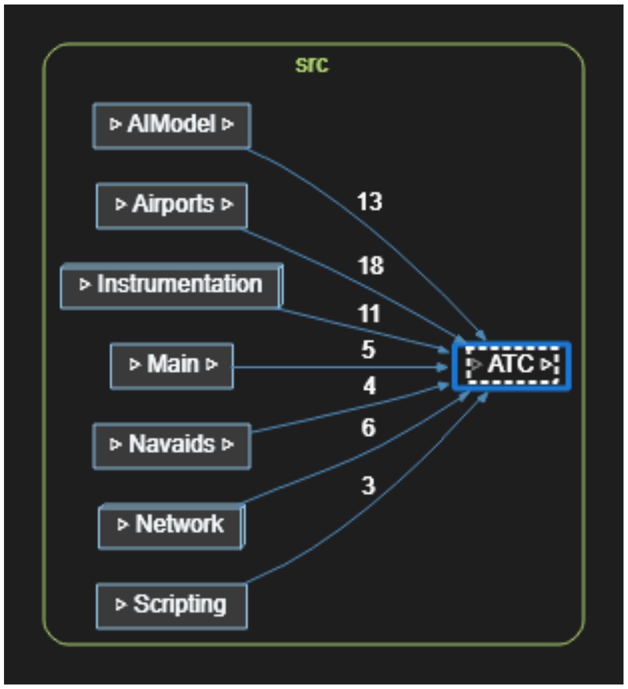

Figure 18: ATC Dependencies

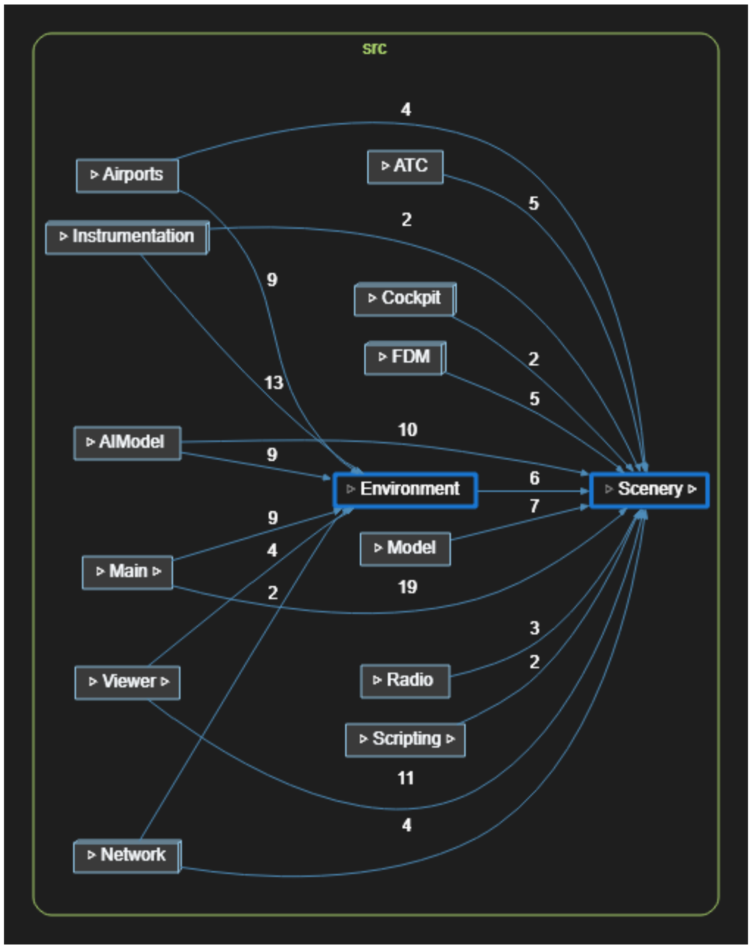

Figure 19: Scenery and Environment Dependencies

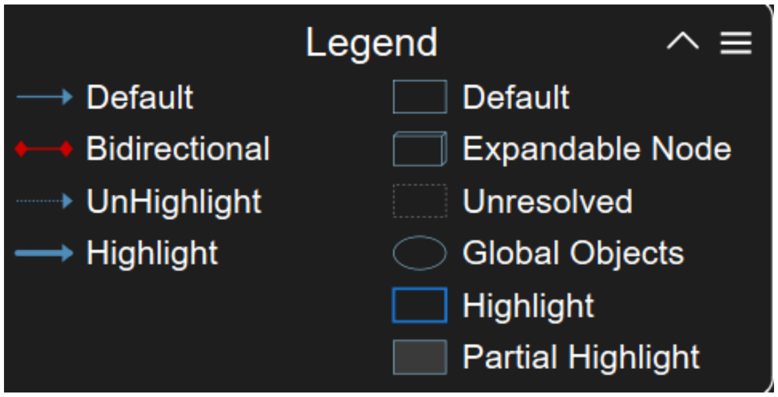

Figure 20: Understand Tool Dependency Legend

Appendix B – FDM Architecture

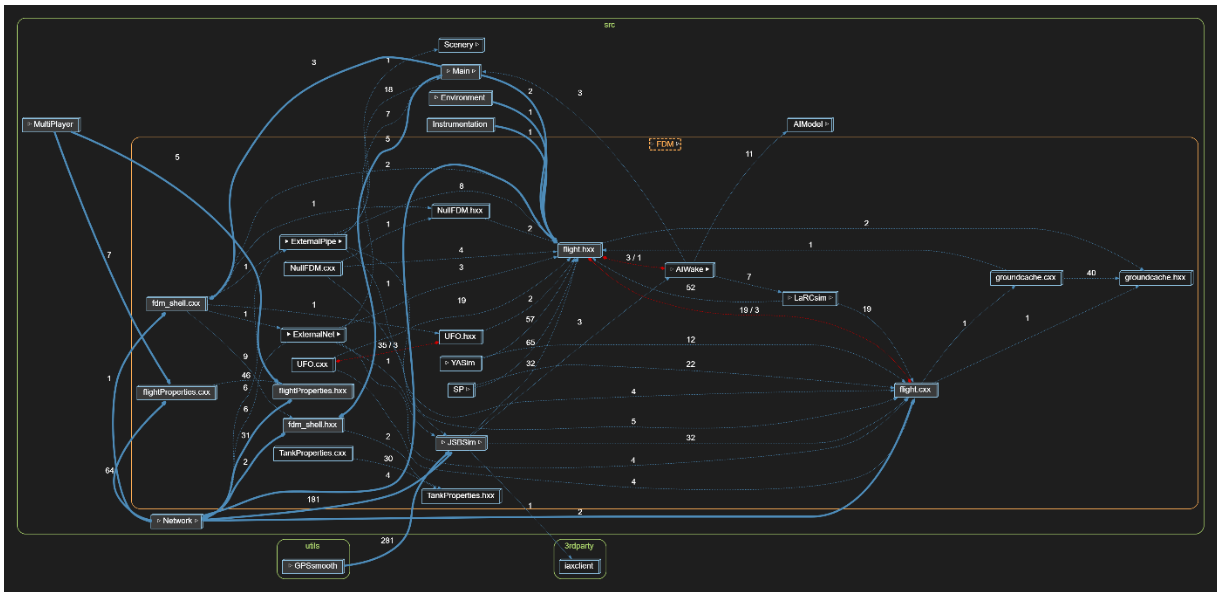

Figure 21: FDM Concrete Architecture External Dependency Structure

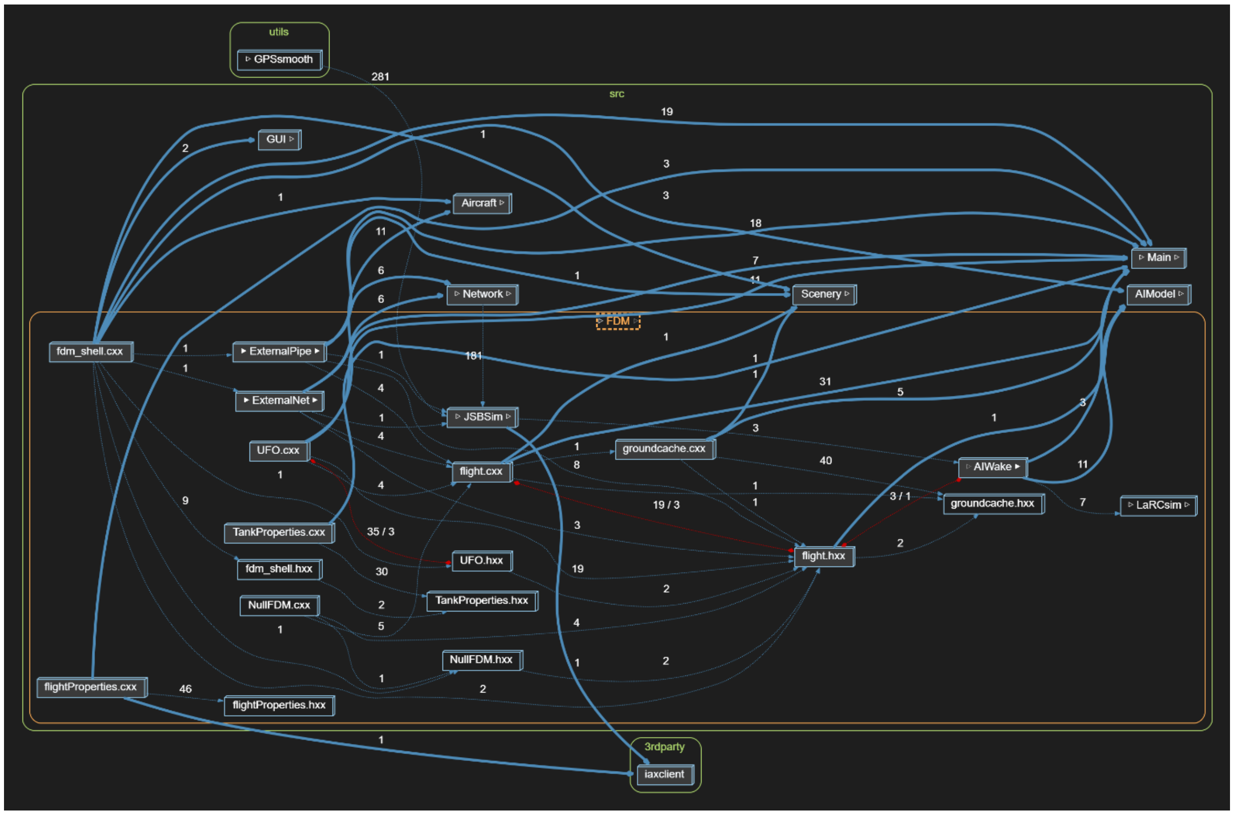

Figure 22: FDM Concrete Architecture External Dependent Subsystems