A2: Concrete Architecture Report
FlightGear Simulator

CISC/CMPE 322/326

March 25th, 2024

Group 5: GoneFishing
Arshan Abdi (20298549)
Kenneth Laird (20222078)
Anson Liu (20232192)
Derek Ma (20109427)
Curtis Pike (20174323)

Abstract

In the previous paper, the FlightGear Flight Simulator was analyzed in order to understand its basic components and their interactions, but failed to properly delve into its conceptual architecture. As such, with this paper in order to properly compare the concrete against the conceptual architecture another group's paper was referenced for the sake of time and accuracy. This paper outlines the derivations process, descriptions of the top-level concrete subsystems, the discrepancies between the conceptual and concrete architectures, the non-trivial use cases and the lessons learned. The source code was categorized and analyzed through the program Understand by Scitools, revealing new subsystems and dependencies as well as their respective calls to one another. Analysis on these discrepancies and non-trivial use cases followed to further improve the architecture being referenced as well as the group's understanding of the software.

Introduction and Overview

Acknowledging the shortcomings in our prior conceptual architectural analysis, we incorporated a comprehensive conceptual architecture from another group because of its detailed subcomponents and diagrams. This choice allows for a deeper study into FlightGear;s architecture, also employing the understanding of SciTools to generate a dependency graph that highlights the complex interrelations between the subsystems. Our examination of FlightGear is underpinned by a repository style.

Derivation Process

Due to a poor foundation in our previous conceptual architecture, we needed to expand our analysis by borrowing another group's design. We have chosen to incorporate group 3's conceptual architecture due to their detailed derivation of the subcomponents complete with a description, input, and output for each, as well as their use of clear and well-constructed diagrams.

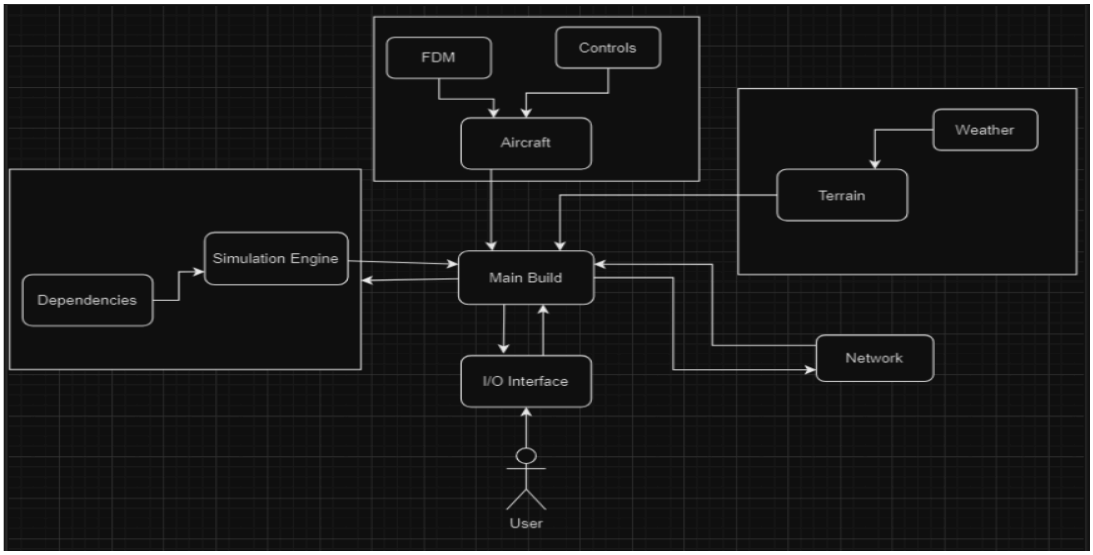

Figure 1.1: New conceptual architecture, from group 3

After acquiring our new conceptual architecture, we used SciTools Understand platform to generate a dependency graph to better understand the concrete architecture. Following the steps of the tutorial, we imported the data and generated an unorganized map of dependencies. In order to create a better understanding, we performed a rough categorization of each source code folder sorting them based on our newly acquired conceptual architecture. Once the folders were generally categorized, we were able to dig deeper into the source code and further examine the interactions to adjust our categories.

Based on the created concrete architecture and reviews from the previous assignment, we concluded that FlightGear used a repository style architecture. This is because the FlightGear main build acts as the data repository where other parts of the simulation continuously send and request data. We used this characterization for the reflection analysis on the differences between the conceptual and concrete architectures.

Top-Level Concrete Subsystems and Interactions

As mentioned previously, another group's conceptual architecture for FlightGear was referenced to develop the concrete architecture. The system is divided into four submodules that work in tandem to create a realistic flight simulation. To summarize, in the conceptual architecture the Simulation Engine, Terrain, Aircraft and Network are all separated submodules that communicate with the Main Build, which interacts with the user through a I/O Interface. These submodules have their own dependencies, but in the concrete architecture the relations and dependencies become more complex, allowing for further investigation. That is to say while in the conceptual diagram, an element may not depend on the user. However, that element may use the input from the user in some way in the concrete implementation of FlightGear.

The Main Build functions as the core of the architecture by acting as a dependency for other systems as well as having dependencies in others. Its responsibilities include logging events and reporting errors on the backend, keeping track of options set by the user, coordinating scripting, among other things. In short, the Main Build provides a basis for the system to create a realistic flight simulation successfully.

The ATC, or the air traffic control subsystem interacts with the Main Build, the I/O Interface, the Simulation Engine and has dependencies on the Terrain and Aircraft subsystems. It is responsible for sending air traffic information to the user that it receives and creates through its interactions with the terrain and aircraft systems, which is relayed through the simulation engine and the I/O interface.

The User subsystem interacts with the Main Build and I/O Interface and has dependencies with the Simulation Engine and the Aircraft subsystems to allow the user to properly control and experience the simulation as effectively as possible. It includes all the functionality responsible for handling the user's interaction with the system, and relaying that information to the main build or to wherever the dependencies require.

The Aircraft subsystem interacts with the Main Build, the I/O Interface and the Terrain subsystems. It is responsible for accurately simulating the model of the aircraft chosen and its specifications accurately. It includes the functionality for the cockpit, radio, and other features

related to the aircraft that the user interacts with. The controls for the aircraft are encompassed by this subsystem, which interacts with the User and IO Interface to control the aircraft's behavior.

The Terrain subsystem incorporates the physical geography of the environment and the weather mechanics. It interacts with the Main Build and Simulation Engine subsystems to relay the related information.

The Network subsystem communicates with the Main Build primarily but interacts with many other subsystems like Multiplayer, FDM, Terrain, and IO Interface in order to properly keep track of different interacting data and properly relay them to the network. As such it also has dependencies with Simulation Engine and Aircraft to ensure the accurate data is transmitted.

The Multiplayer subsystem interacts with primarily Network to facilitate online play and communication, but also with the Simulation Engine and the Main Build to keep simulating a realistic experience for the user. It also has dependencies in FDM to ensure further stability for both the user and any other connected players.

The Simulation Engine subsystem has many two-way dependencies. Some include Main Build, Multiplayer, Terrain, and I/O Interface. There are a few one-way dependencies as well. Namely, User and Network. This subsystem is responsible for compiling all of the information across the connected subsystems into one simulation that other systems may use. This is a core part of the system, as it is depended on by many subsystems. As part of our derived concrete architecture it includes the components for AIModle, Airports, Navaids, Systems, Time, and Traffic to accurately simulate flight.

While the User subsystem only handles user input, the I/O Interface subsystem interacts with all the other subsystems. All of these require the interface to properly simulate the user's inputs and output the proper responses. It also has dependencies with the Multiplayer subsystem for the online interaction. This subsystem is responsible for relaying the information about the system to the user through the visuals and sounds with the Graphical User Interface (GUI), cockpit, and any sounds. This subsystem aims to provide the means to give the user an authentic and immersive experience.

The FDM subsystem is responsible for calculating physical forces on the simulated aircraft and as such functions as a vital component interacting with the Main Build, the IO Interface, the Network and the Simulation Engine. The relation with the Simulation Engine is particularly notable as they work heavily in tandem to provide the user with as realistic an experience as possible.

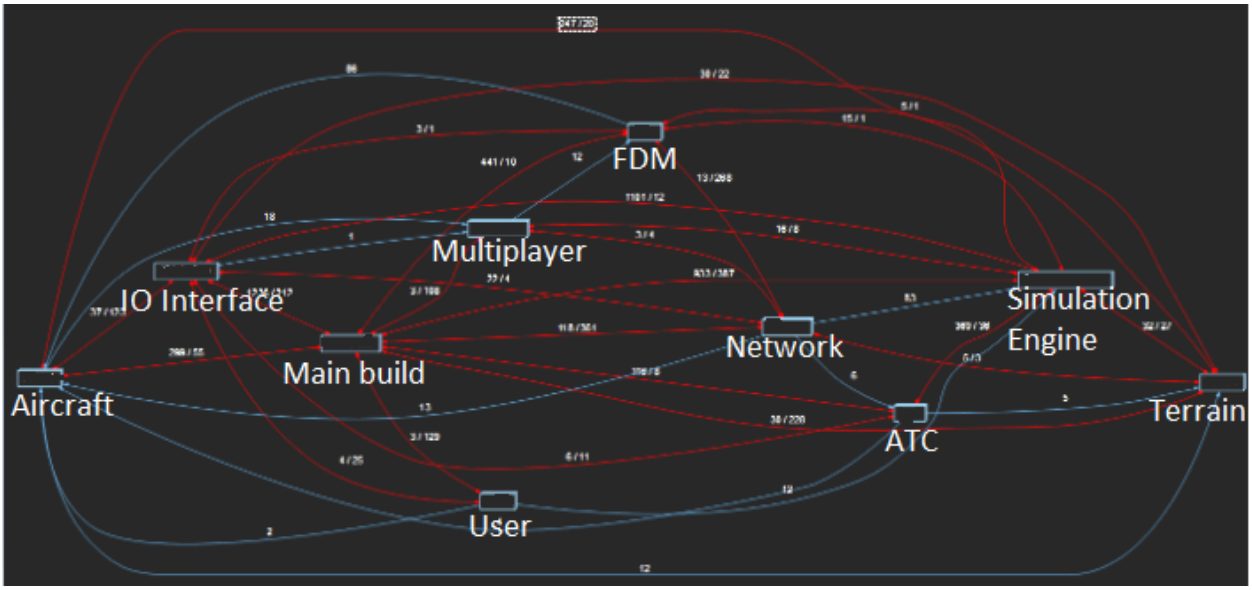

Figure 1.2: Concrete architecture

Differences Between Conceptual and Concrete

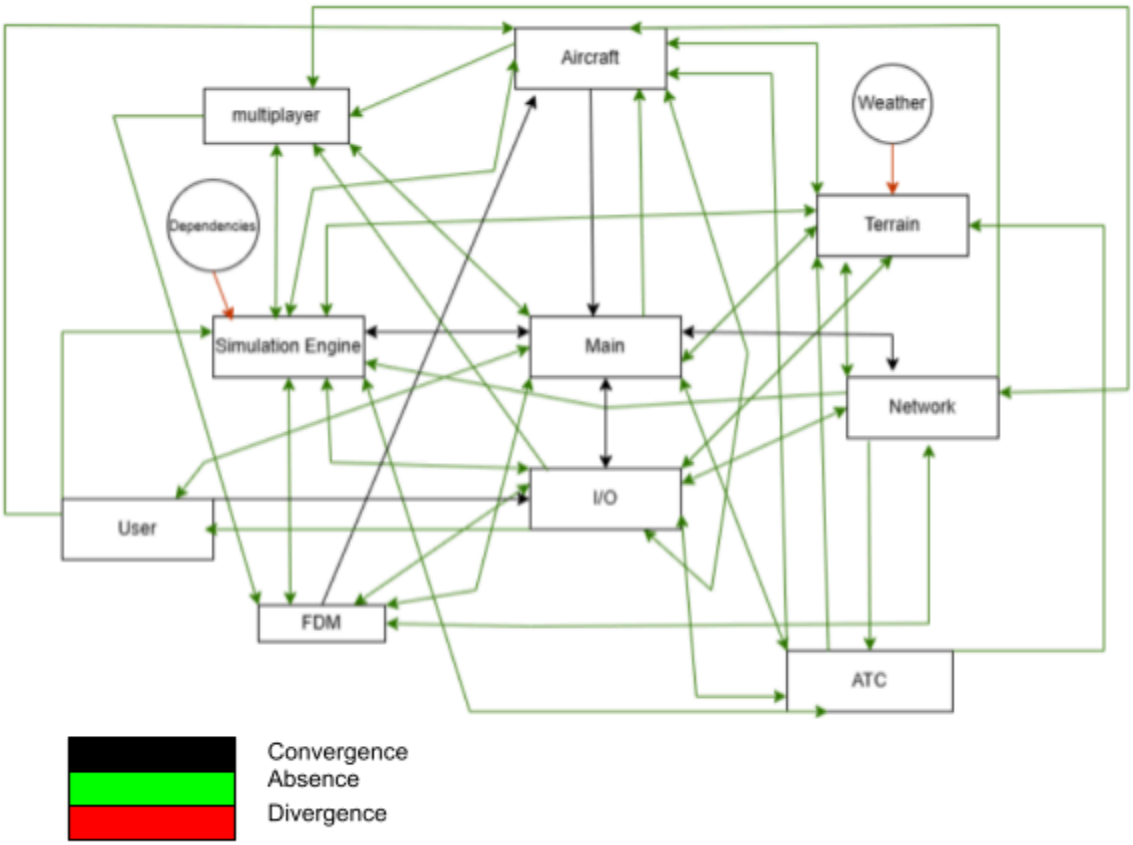

Figure 1.3 Reference architecture for high-level

Divergences

Network → Aircraft

The FlightHistoryUriHandler file in the network subsystem is designed to handle HTTP requests related to the flight history feature within FlightGear. This feature captures and stores the path of an aircraft during its flight, allowing for later retrieval and visualization through web requests. The network depends on the Aircraft subsystem in order to retrieve flight history information.

Network → simulation engine

The network has a dependency with the simulation engine in regards to 3 components within the simulation engine subsystem, the AIModel, Navaids and Airports components.

Network → simulation engine → AIModel

The HLA class within the Network subsystem depends on the AIModel in order to enable interoperability among the simulation classes, and it needs the AIModel data in order to facilitate a multiPlayer environment with AI aircrafts.

Network → simulation engine →Navaids

The fgcomm.cxx class within the Network subsystem (a system for voice communication) depends on Navaids in order to implement a simulation of radio communications with navigational aids facilities, such as VOR stations or NDB beacons, which pilots use for navigation and receiving important information. Also, The system uses navaids to determine the geographical location and proximity of aircraft to specific communication points or facilities. This allows FGCom to simulate range limitations of radio communications, where pilots can only communicate with ground control, ATC, or other aircraft if they are within a realistic distance from a navaid or airport.

Network → simulation engine →Airports

The Network has a dependency on the Airports subsystem for the same reason as why Network has a dependency with Navaids. It needs Airport information in order to facilitate features within the fgcomm class (to enable communication with airports).

Network → I/O

The network has a dependency with the simulation engine in regards to 2 components within the I/O subsystem, the Canvas and Viewer components.

Network → I/O → Canvas

The screenshotUriHandler.cxx class in the network subsystem is used to handle HTTP requests for a screenshot of the simulation. The FlightGear canvas system is a tool that allows for the creation of 2D drawing surfaces which can be overlaid on the 3D world or used in GUI elements. Screenshots may include elements drawn with the canvas system, such as GUI windows, instruments, or any other 2D elements that are part of the FlightGear interface or custom user interfaces. To accurately capture a screenshot that includes these elements, the ScreenshotUriHandler needs to interact with the canvas system to ensure these elements are rendered in the screenshot.

Network → I/O → Viewer

The viewer subsystem in FlightGear is responsible for rendering the 3D world, including aircraft, terrain, sky, weather effects, and all other 3D elements. Capturing a screenshot of the simulation requires accessing the rendered frames from the viewer subsystem to obtain a

snapshot of the current visual output. The viewer provides access to the camera views, rendering settings, and other critical components necessary for generating a screenshot that accurately reflects what the user sees in the simulation.

Network → ATC

The fgcomm class (system for voice communication) depends on the ATC subsystem in order to retrieve vital information about the communication stations in the ATC in order to facilitate communication between the two.

Network → FDM

The native_fdm file in the Network subsystem is designed to interface FlightGear's simulation environment with external systems or applications through a network or data distribution service (DDS), allowing for the exchange of flight dynamics data. Native_fdm serves as a bridge to exchange flight dynamics data between the internal FDM of FlightGear and external systems. The class needs to read from and write to the FDM to accurately convey the current state of the simulated aircraft, including its position, orientation, velocities, and control surface states. When used in an input mode (SG_IO_IN), native_fdm allows FlightGear to receive and apply flight dynamics data from an external source.

Network → MultiPlayer

The hla.cxx enables interpretability of multiple simulation systems, thus hla requires data of the multiplayer object instances to enable this interpretability.

Terrain → Main

The Terrain subsystem is composed of 2 subsystems Environment and Scenery these components explain the new dependencies. The Main subsystem consists of two lower subsystems: the main and the scripting components.

Terrain → Main → Main

The Terrain subsystem relies on the Main component, due to its dependency to have its properties managed by the fg_prop class in the Main subsystem file class. It also depends on the Main to retrieve the globals within the simulation.

Terrain (Scenery) → Main → Scripting

The tilemgr.cxx file within the Scenery component is responsible for managing scenery tiles. The Nasal scripting engine allows for dynamic loading and manipulation of scenery based on scriptable conditions. This means that scenery can be changed or modified in real-time without needing to restart the simulation. By integrating scripting capabilities, FGTileMgr allows for a high degree of customization and extensibility of the scenery.

Terrain (Environment) → Network

The realwx_ctrl file in the Environment component is a class that processes real weather data. Realwx_ctrl class's dependency on the network is essential for its primary function of fetching and processing real-world weather data to enhance the realism of the FlightGear simulator.

Terrain → I/O

The Terrain subsystem has a dependency regarding 2 components within the I/O the GUI component and the Viewer component.

Terrain (Environment) → I/O → Viewer

The environment_mgr class depends on the viewer because it manages aspects of the environment that have a direct impact on the visual simulation.

Terrain (Scenery) → I/O → GUI

The scenery.cxx class in FlightGear is responsible for managing the 3D visual representation of the world, including terrain, models (buildings, trees, etc.), aircraft, clouds, and other scenery objects. The dependency seems to be an indirect one of capturing mouse clicks or interactions with the scenery for purposes like selecting an airport from the map view, interacting with ground objects, or other GUI-driven features. However, these connections are typically managed through higher-level systems that bridge user inputs from the GUI to actions within the scenery system, rather than direct dependencies within the scenery.cxx class itself, though a dependency was found within the file.

Terrain != Weather

The Weather subsystem was removed from the architecture because in the source code there wasn't any indication of their being a subsystem dedicated to weather rather their were subsystems dedicated to Environment and Scenery.

ATC → Terrain (Scenery)

The groundController class is primarily focused on managing ground operations at airports. The dependency on scenery within FGGroundController arises because effective management of ground operations is deeply intertwined with the physical and visual representation of the airport environment provided by the scenery.

ATC → I/O (GUI)

The atcDialog class acts as an intermediary between the user and the simulation's ATC system, leveraging GUI elements to facilitate communication and interaction.

ATC → Simulation engine

ATC → Simulation engine → AIModel

The dependency on AIModel allows ATCManager to effectively manage air traffic control interactions, including those involving both user-controlled and AI-controlled aircraft, within the FlightGear simulation environment.

ATC → Simulation engine → Airports

Air Traffic Control (ATC) operations are closely tied to airport operations. The ground controller may need to coordinate with ATC systems or simulate ATC instructions that are specific to airport layouts and procedures.

ATC → Simulation engine → Navaids

NavDataCache class, which likely handles navigation-related data including navaids.

ATC → Simulation engine → Traffic

The air traffic control relies on the Traffic subsystems to determine its scheduling.

ATC → Main

Includes fg_props and globals.

ATC → Aircraft (Radio)

Radio transmissions for traffic control.

Main → Aircraft

The fg_init class is responsible for initializing the configurations of the flightGear simulation. The aircraft-related information plays a crucial role in configuring the simulator environment according to the selected aircraft, ensuring compatibility.

Main → Terrain

The initialization process of FlightGear relies on environmental and scenery data to set up the simulation environment, configure aircraft parameters, initialize navigation systems, load scenery and visuals and configure weather conditions.

Main → ATC

integrating ATC interaction into the initialization process of FlightGear enhances realism, supports traffic management, ensures safety, promotes immersion, and facilitates training for virtual pilots.

Main → FDM

The Flight Dynamics Model (FDM) is a fundamental component of FlightGear's initialization process, providing crucial information and calculations to set up the aircraft accurately, integrate its systems, plan the flight, and account for dynamic environmental factors.

Main → User (Input)

User input's role in the initialization of flight simulation systems like FlightGear, enabling customization, training, research and simulation control.

Main → MultiPlayer

The initialization process in a multiplayer flight simulation relies on the MultiPlayer system to synchronize simulation states, facilitate player interaction, enforce server authority, and ensure scalability.

Aircraft → Terrain

The radar system in the aircraft needs scenery information for initialization and for its ability to navigate and run throughout the flights path.

Aircraft → MultiPlayer

The flight recording system would rely on the MultiPlayer subsystem in order to extract information about all the plans given a multiplayer flight session.

Aircraft → Simulation engine

The aircraft system depends on the simulation engine system, composed of airports, navaids, and time components, to provide essential information and services for navigation, airport operations, time simulation, dynamic environment modeling, and flight planning.

Aircraft → I/O

The aircraft subsystem depends on the Input/Output (I/O) subsystem (specifically the Time, Viewer, Instrumentation and GUI components) for user interaction, instrument display, audio feedback, visual presentation, and data exchange.

I/O → Network

The I/O subsystem in FlightGear depends on the Network system for multiplayer support, integration with external data sources, remote control and monitoring, distributed simulation, and plugin integration.

I/O → ATC

The dependency of the I/O subsystem on the ATC system in FlightGear reflects the importance of seamless communication, information display, and user interaction within a comprehensive flight simulation environment.

I/O → Aircraft (Cockpit)

The mouse input depends on the panel class within the Aircraft subsystem (the Cockpit subsystem) in order to get the panel information, to have a variety of features for the pilots to choose in order to operate the aircraft.

I/O → Terrain

The I/O subsystem may incorporate terrain data to display terrain features on navigational instruments or graphical interfaces. This helps pilots to visually reference terrain landmarks, such as mountains, rivers, or cities, aiding in navigation and route planning.

I/O → Simulation engine

The I/O subsystem depends on the simulation engine to provide real-time data about the simulated environment, such as aircraft position, orientation, weather conditions, and system status. This data is crucial for updating the visual and auditory feedback presented to the user through graphical interfaces, instruments, and sound effects.

I/O → FDM

The I/O subsystem relies on the FDM to obtain accurate information about the aircraft's performance, handling characteristics, and response to user inputs.

I/O → User

The user interacts with the simulation through various input devices such as keyboards, joysticks, and flight control panels. The I/O subsystem receives input from these devices and translates them into commands that affect the simulated aircraft's behavior. This interaction is essential for enabling users to control the aircraft, adjust settings, and navigate the simulated environment effectively.

I/O → MultiPlayer

The I/O subsystem relies on multiplayer networking protocols to exchange data with other users' simulators, such as aircraft positions, movements, and communications. This enables collaborative flying, air traffic control interactions, and shared experiences among users, enhancing the realism and social aspects of the simulation.

MultiPlayer → Network

The network system ensures that data is transmitted efficiently between users, allowing for real-time synchronization of the simulated environment across all connected clients.

MultiPlayer → Main

In the context of multiplayer functionality, the main system coordinates the setup and management of multiplayer sessions, including connection establishment, data transmission, and session termination.

MultiPlayer → Simulation engine

In a multiplayer context, the simulation engine ensures that all participants experience a consistent and synchronized view of the simulated world, regardless of their individual locations or actions.

MultiPlayer → FDM

In multiplayer sessions, each participant's aircraft is represented by its FDM, which calculates the aircraft's behavior based on user inputs and environmental conditions.

Simulation engine → MultiPlayer

The Simulation Engine needs to synchronize the state of the simulated world across multiple instances of FlightGear when multiplayer functionality is active. It receives updates from

other users' simulations regarding their aircraft positions, movements, and interactions with the environment, ensuring consistency and coherence in the shared virtual space.

Simulation engine → Aircraft

The Simulation Engine incorporates the behavior and characteristics of different aircraft models into the simulation environment. It interacts with the Aircraft System to retrieve data on individual aircraft configurations, flight dynamics, and performance parameters, enabling accurate representation and simulation of diverse aircraft types.

Simulation engine → Terrain

The Simulation Engine generates and renders the terrain, including landscapes, landmarks, and topographical features. It relies on the Terrain System to provide detailed terrain data, such as elevation, textures, and object placements, which are essential for realistic scenery rendering and flight simulation.

Simulation engine → ATC

It collaborates with the ATC System to manage AI-controlled aircraft, implement air traffic control procedures, and handle communications between virtual pilots and controllers.

Simulation engine → I/O

The Simulation Engine interacts with input and output devices, including graphical user interfaces, instrumentation panels, sound systems, and external viewers. It relies on the I/O System to receive user inputs, display simulation outputs, provide feedback, and manage various interface components.

Simulation engine → FDM

The Simulation Engine integrates flight dynamics models to simulate the physical behavior of aircraft accurately. It collaborates with the FDM System to calculate aircraft movements, aerodynamic forces, control responses, and other flight-related parameters.

FDM → I/O

Input devices controlled by the I/O system affect the behavior of the aircraft, which in turn influences the calculations performed by the FDM to simulate realistic flight dynamics.

FDM → Main

The FDM subsystem may depend on the Main system for access to global settings, system resources, and initialization routines necessary for its proper functioning.

FDM → Network

In multiplayer scenarios, the FDM subsystem may rely on the Network system to exchange data with other instances of FlightGear participating in the multiplayer session. This data exchange enables synchronized simulation of aircraft movements, positions, and interactions across multiple connected clients.

FDM → Simulation engine

The FDM subsystem collaborates closely with the Simulation Engine to simulate the physical behavior of aircraft accurately.

User → Simulation engine

The User system depends on the Simulation Engine to provide real-time updates on the simulation state, including aircraft position, orientation and velocity.

User → Aircraft

It provides user interfaces for configuring and controlling various aspects of aircraft operation, such as selecting aircraft models, adjusting cockpit instrumentation, configuring autopilot systems, and managing flight plans.

User → Main

The User system relies on the Main system for essential infrastructure services and management functions. It may use the Main system for tasks such as initializing user preferences, managing user profiles and settings, handling user input events, and coordinating interactions with other subsystems.

ATC subsystem reflexion analysis

Conceptual Architecture[1]

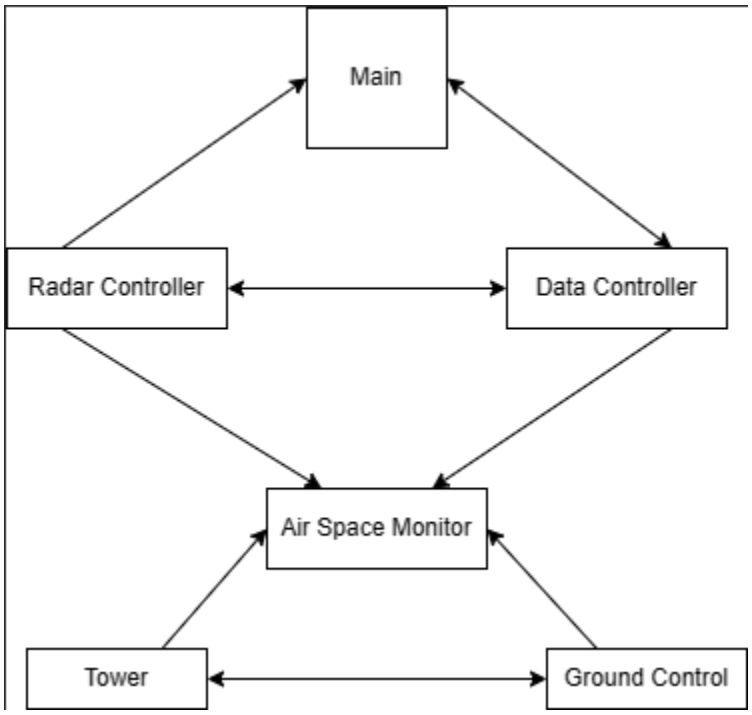

Figure 1.4 Conceptual architecture for ATC

In the conceptual architecture of an Air Traffic Control (ATC) subsystem, the Main component serves as the central hub facilitating communication and data exchange between various specialized controllers. It communicates crucial data and information with the Radar controller and the Data controller. The Radar Controller is responsible for surveillance radar monitoring and separation of air traffic. This controller detects the presence and location of aircraft by analyzing reflections of radio signals from their surfaces. Additionally, it retrieves further information about aircraft identity, altitude, and other relevant data through transponder signals. The Data Controller plays a vital role in collecting, processing, and displaying radar data, ensuring that controllers have accurate and up-to-date information for making informed decisions.

Another integral component is the Ground Controller, which guides aircraft while they are on the ground, managing taxiing, gate assignments, and other ground movements. Meanwhile, the Tower Controller monitors aircraft positions within the terminal control area, overseeing departures and arrivals, and ensuring safe spacing and sequencing of aircraft movements within the airspace surrounding the airport. Together, these components form a cohesive system designed to manage and ensure the safe and efficient flow of air traffic both on the ground and within controlled airspace.

Concrete Architecture

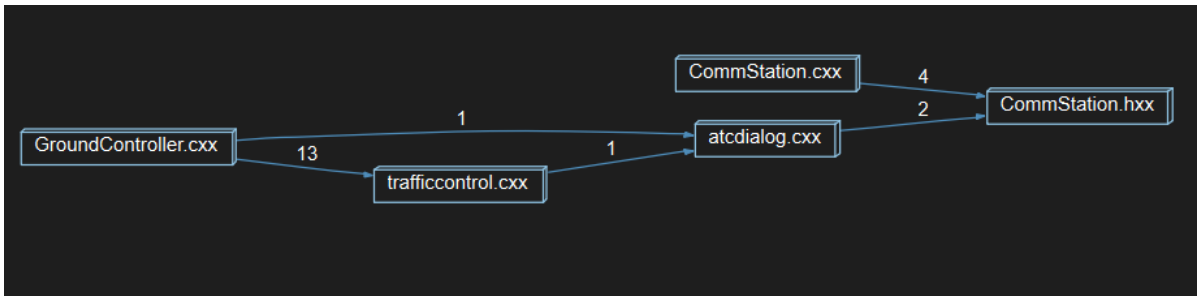

Figure 1.5 Concrete architecture for ATC derived from Understand

Reference Architecture

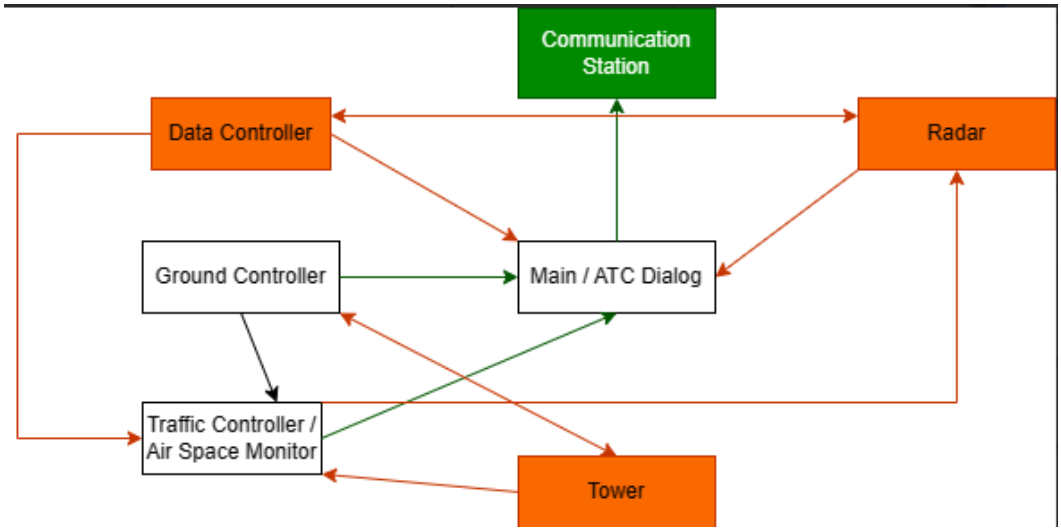

Figure 1.6 Reference architecture for ATC

Reflexion

In the reflection analysis of the reference architecture for the Air Traffic Control (ATC) system, deviations from the concrete architecture were observed. Specifically, the Radar controller, Data controller, and Tower components were not directly implemented as standalone entities. Instead, they were integrated into new components within the system.

The Radar and Data controllers were grouped together into a new component called the communication station. This consolidation likely aimed to streamline communication processes

and enhance data management efficiency by centralizing radar data collection, processing, and distribution tasks.

Similarly, the Tower component was incorporated into the Traffic Controller or Airspace Monitor. By merging these functionalities, the system could potentially achieve better airspace monitoring and traffic management capabilities, as the Traffic Controller would have a broader scope of responsibilities encompassing both en-route traffic and terminal control.

However, certain divergences from the concrete architecture were noted. Specifically, the Ground controller component remained dependent on the Main component for essential functions such as information sharing, decision support, coordination with other controllers, and overall system integration. This dependency ensures seamless integration of ground control operations within the broader ATC system.

Likewise, the Traffic Controller or Airspace Monitor continued to rely on the Main component for centralized management, data aggregation, decision support, communication, coordination, and overall system integration. This dependency reinforces the critical role of the Main component in facilitating efficient and effective air traffic management within the ATC subsystem.

Non-Trivial Use Cases & Sequence Diagrams

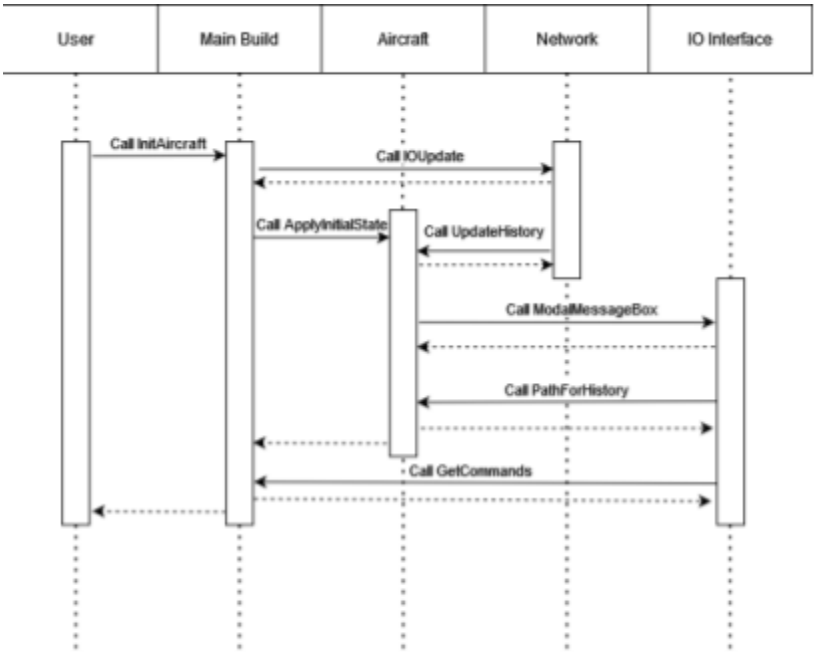

Figure 1.7: Use Case Sequence Diagram: Initializing Aircraft for Simulation

The first use case to be analyzed is the user initializing an aircraft to be simulated. This is one of the first necessary steps in order to properly begin using the software and Understand was once again utilized to decompose the process through inspecting the calls in the dependency graphs. In this use case the user calls to the Main Build to initialize an aircraft, this can be a default aircraft or an existing one in the Flight Simulator directory. The Main build will update the Network with new input or output data, and then call the Aircraft subsystem to initialize the aircraft given or the default. The Network will call the Aircraft subsystem as well to ensure the history of the aircraft being used is logged and then the Aircraft will call the IO Interface to start a Modal Message Box for the user to interact with and set up the interface. The IO Interface will then call to the Aircraft to update its own history to match the other subsystems, as well as call the Main Build to get the commands for the message box. The Aircraft in the

meantime returns the initial state call to the Main as it returns the commands to the Interface before finishing by returning to the User.

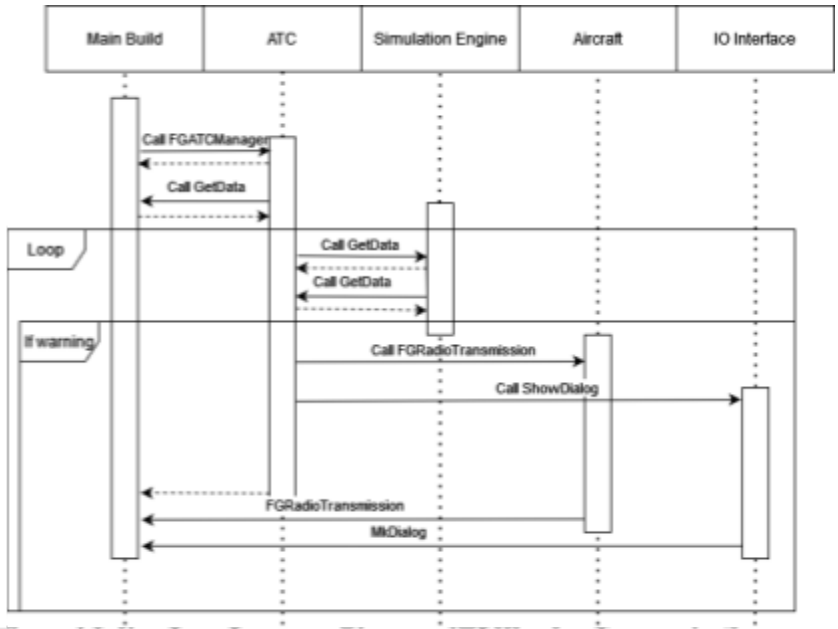

Figure 1.8: Use Case Sequence Diagram: ATC Warning Communication

The second non-trivial use case to be looked at is in case the user must be informed of a traffic warning or notification to change course. To begin with, this use case diagram will assume the user is in flight simulation and the Main Build is running and initialized, thus any communication between the two is assumed. Despite this, we still show the Main Build communicating with the ATC subsystem to create it through the FGATCManager call. The ATC and other subsystems will call a multitude of functions to get necessary air and traffic data from different subsystems that have been condensed into just GetData calls to reduce complexity in the diagram. To start the ATC will call GetData to the Main Build first for current data air and traffic data. It will also communicate with the Simulation Engine subsystem to get data on the airport it originated from, the airport it is landing at, the air traffic data, and data on any simulated models. The Simulation Engine will also call the ATC to get data on the user's aircraft and flight and relay that information to the models. If the ATC subsystem finds from the data acquired that the AI model or the air traffic needs to be communicated to the user to notify them it will send calls to the IO Interface and the Aircraft subsystems accordingly. The Aircraft will simulate a radio to contact the user while the IO Interface will simulate notifications showing on the user's display.

Conclusion

In order to develop better understanding of the FlightGear simulation, we dive deeper into the main system by using SciTools to analyze the architecture of its numerous subsystems. There are many discrepancies between conceptual architectural and concrete architecture. Notably, the analysis of top-level concrete subsystems such as the Simulation Engine, Terrain, Aircraft, and Network modules revealed sophisticated interrelations between the components that collaborate to simulate a highly realistic flight environment. These findings showcase the critical role of concrete architecture analysis in bridging the gap between conceptual planning and actual system behavior, thereby enhancing our understanding of software architecture of the flight simulation.

Lessons Learned and Team Issues

During our FlightGear's concrete architecture, we encountered several unavoidable challenges. Among these was the departure of a team member, which compounded the workload for the remaining members and tightened our deadlines. Despite being reduced to five members, we efficiently allocated tasks among ourselves and successfully met our objectives. Another challenge involved the use of SciTools Understand, which presented a steep learning curve and occasionally suffered from slow response times or errors. Nonetheless, we overcame these hurdles and gained proficiency in the software. One significant insight from this report is learning the SciTools Understand. Through it, we generated dependency graphs for FlightGear, mapping out the intricate web of dependencies within its software and source code. This allowed us to identify which modules rely on others for functionality and discover the purpose of functions based on their code. Moreover, we learned the importance of a well-crafted conceptual design in comparing it to the concrete implementation. Such comparison not only deepens our comprehension of FlightGear's architecture but also highlights the alignments and discrepancies between the conceptual and concrete designs. This process showcases the value of conceptual design in analyzing and improving software architecture, enhancing our ability to distinguish between what was envisioned and what was realized.

References:

1. “Malat WebSpace: Royal Roads University |.” *MALAT WebSpace | Royal Roads University*, malat-webspace.royalroads.ca/. Accessed 26 Mar. 2024.