FlightGear Concrete Architecture

Prof. Amir Ebrahimi
Queen's University
CISC 322
25 March, 2024

Al-Barr Ajiboye (albarr.ajiboye@queensu.ca)
Kristina Arabov (20kah10@queensu.ca)
Jack Caldarone (21jsc5@queensu.ca)
Safowan Mostaque (20sbm4@queensu.ca)
Grace Odunuga (20goo@queensu.ca)
Jenna Wang (20jw99@queensu.ca)

Table of Contents

Abstract……3

Introduction and Overview ……..……3

Architecture……...…4

Derivation Process……...6

Subsystem Descriptions……...6
Aerodynamics……..6
Aircraft Characteristics……7
Flight Simulation……7
Images……..7
Navigation……7
Sounds…… 7
Time…… 7
User Requests……...7
Weather……8

Flight Dynamic Model (FDM) Analysis……..8

Reflexion Analysis……...9

2nd level subsystem Reflexion analysis……10

External Interfaces ……12

Use Cases……...12

Data Dictionary ……14

Naming Conventions ……15

Lessons Learned ……15

Conclusion……16

Works Cited ……..17

Abstract

In the open-source flight simulation arena, FlightGear stands as a notable case of how software architecture can evolve to address the needs for realism, performance, and community involvement. This report delves into FlightGear's architectural journey from its initial Publisher-Subscriber framework concept to a realized Object-Oriented design. This transformation offers insights into the adaptability challenges and methodologies essential for maintaining the simulator's scalability and functionality as it grows in complexity.

A closer look at FlightGear's architecture reveals the critical interplay between various subsystems such as Aerodynamics, Aircraft Characteristics, Flight Simulation, and User Interaction. These components are fundamental to FlightGear's high fidelity in simulating aviation dynamics, facilitated by the Object-Oriented architecture that enhances modularity and reusability. This structure allows for the smooth integration of new features, ensuring the simulator's ongoing evolution.

The analysis further highlights the vibrant open-source community's role in propelling FlightGear's development. Community contributions extend beyond feature expansion to crucial debugging and architecture refinement efforts. The report sheds light on the dynamic between theoretical architectural models and the practical challenges of implementation, offering a perspective on decision-making in complex software development.

Moreover, the shift towards an Object-Oriented architecture emphasizes modern software development's need for flexibility and adaptability. This approach allows for more intuitive subsystem interactions, promoting efficient data and functionality exchanges without a central coordination unit. While this presents unique challenges, it significantly benefits the simulator's performance and user experience.

Ultimately, FlightGear's architectural evolution from concept to implementation exemplifies the impact of open-source collaboration and the importance of adaptable architecture in navigating the expanding scopes of projects and technological progress. This report documents FlightGear's development journey, providing valuable insights into the intersection of architectural theory and practice within large-scale, community-driven software projects. Through this exploration, the report contributes to the broader discussion on best practices in software architecture, highlighting the intricacies of aligning conceptual designs with real-world applications in software engineering.

Introduction and Overview

The development and evolution of FlightGear, an open-source flight simulator, offers a fascinating glimpse into the application of sophisticated software architectural principles within a complex, community-driven software project. Initially conceived to leverage the flexibility and modularity of a Publisher-Subscriber model, subsequent in-depth analysis and practical engagement with the system's internals have necessitated a shift towards an Object-Oriented approach. This evolution underscores the dynamic nature of software development, where theoretical models often undergo significant transformations to accommodate practical realities, performance requirements, and the collaborative inputs of a global development community.

FlightGear's journey from its inception to its current state as a leading flight simulation platform is marked by continuous adaptation and refinement. The initial conceptual architecture was predicated on the Publisher-Subscriber pattern, a choice motivated by the system's need for real-time responsiveness and efficient communication among its disparate components. However, as the project matured, it became evident that the intricacies and interdependencies

within FlightGear's subsystems—ranging from Aerodynamics and Aircraft Characteristics to Flight Simulation and User Interaction—necessitated a more integrated and direct method of interaction, leading to the adoption of an Object-Oriented architecture.

The Object-Oriented architecture offers numerous advantages in the context of FlightGear's development, including enhanced modularity, reusability, and the ability to encapsulate complex behaviors within well-defined class structures. This architectural style facilitates the seamless integration of new features and subsystems, enabling FlightGear to incorporate cutting-edge aviation physics, realistic environmental simulations, and sophisticated flight dynamics models with greater ease. Moreover, the Object-Oriented approach aligns well with the collaborative ethos of open-source development, allowing contributors to work on distinct aspects of the simulator's functionality without necessitating exhaustive familiarity with its entire codebase.

This report delves into the concrete architecture of FlightGear, unraveling the layers of its design and the rationale behind its architectural decisions. Through a meticulous examination of the simulator's subsystems and their interrelations, we aim to illuminate the complexities of building a software system that is not only technically robust but also adaptable to the evolving demands of users, developers, and the aviation community at large. The transition from a Publisher-Subscriber model to an Object-Oriented framework is explored in depth, highlighting the challenges encountered and the solutions adopted to address them.

Furthermore, the report acknowledges the pivotal role of FlightGear's global development community in shaping the simulator's architecture. The collaborative contributions of numerous individuals and organizations have been instrumental in driving FlightGear's continuous improvement, underscoring the value of open-source principles in fostering innovation, diversity of thought, and rapid iteration. This community-driven development process, coupled with a flexible and scalable architecture, has enabled FlightGear to not only emulate the complexities of real-world aviation but also to pioneer new avenues in flight simulation technology.

In summary, the evolution of FlightGear's architecture from its conceptual beginnings to its concrete implementation encapsulates the challenges and triumphs of developing a large-scale, open-source software project. This report aims to provide a comprehensive overview of FlightGear's architectural journey, offering insights into the decision-making processes, technical considerations, and collaborative dynamics that have shaped its development. Through this exploration, we seek to contribute to the broader discourse on software architecture, highlighting the lessons learned and best practices gleaned from the FlightGear project.

Architecture

In our conceptual report, we made the claim that FlightGear follows a Publisher-Subscriber model, however now that we have access to the concrete architecture, it has become much clearer to us that a more accurate architectural style for FlightGear would be the Object-Oriented style. The reasons why we made the decision to change this architectural style will be fully explained in a later section, however the main reason is that upon viewing the dependencies of the overall system, it became evident that the Publisher-Subscriber model simply did not fit FlightGear. In Figure 1, it can be seen that many subsystems have dependencies on each other, heavily contradicting the Publisher-Subscriber model, where there should be one subsystem that all the other subsystems are dependent on. These subsystems should have very few other dependencies other than this main one, which is very clearly not what is happening with FlightGear.

As previously mentioned, many of these subsystems have dependencies on, and are interacting with each other in some way. This is a very important aspect of the Object-Oriented architectural style, subsystems are able to share information without needing to communicate through a main subsystem, although FlightGear does have a main subsystem. Additionally, information flow is not controlled, rather the information can move in whatever order it needs to, rather than being stuck to a rigid path like the pipe and filter architectural style.

The architecture of FlightGear heavily benefits from this Object-Oriented style, due to the fact that classes are very good at accounting for objects that have multiple interfaces or applications. This is especially good for FlightGear, due to the large number of class functions, which can be seen by the amount of dependencies between subsystems. This is a common pattern seen with Object-Oriented architecture, components will directly call upon each other, requiring the classes to preserve the data types [1]. This is a huge benefit of this style, because it means that objects are able to hide their data representations from an instance, meaning that this data becomes much more versatile in the amount of functions it can take on. This leads to what is seen in the concrete architecture of FlightGear, lots of independent subsystems acting freely with each other. This can lead to some drawbacks of the Object-Oriented architectural style: the constant swapping of information between different parts can lead to confusion about the identity of a class instance, as well as issues caused by the order in which these instances are being accessed or edited.

This can be a pretty major disadvantage that other styles, such as Repository or Client-Server style do not suffer from, however we believe Object-Oriented style is still a better fit for FlightGear for a few key reasons. Repository style seems unlikely for FlightGear due to the amount of information being communicated between subsystems, making the direct communication of Object-Oriented architecture much more appealing. Additionally, the ability to hide data representations from the subsystem that is being communicated with allows much faster and uninterrupted communication. Client-Server is another strong alternative, however it also has some drawbacks that make Object-Oriented architecture the more reasonable choice for FlightGear. The main reason is that Client-Server architecture requires a stable internet connection at all times, something that is not and should not be a requirement to run FlightGear. FlightGear does have a multiplayer functionality, however it is not the main draw of the program and many users never utilize the multiplayer options. This style would cause many key features of FlightGear to be completely unusable without an internet connection, due to FlightGear's reliance on SQL. FlightGear uses SQL to store lots of information, so if SQL was being accessed through a server, FlightGear would be unable to run.

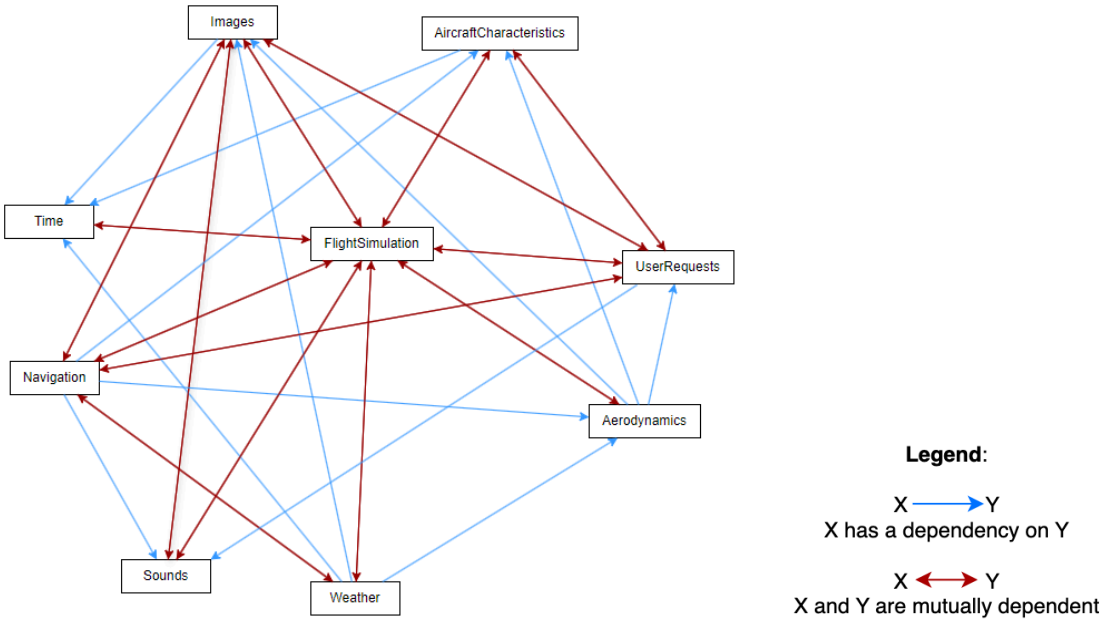

Figure 1. Diagram of the concrete architecture with categorized subsystems.

Derivation Process:

Our group derived that FlightGear must utilize an Object-Oriented style primarily through the use of Understand. Following the provided tutorial, our group initialized specific categories of subsystems (e.g. Navigation, UserRequests, etc.) for simplicity and then added relevant components from FlightGear's source code to those categories. Multiple attempts were made to apply the Publisher-Subscriber style to FlightGear's components, however, the new architecture never followed our proposed conceptual style. Looking back on our proposed conceptual architecture, we believe we were still reasonable in assuming it was Publisher-Subscriber due to the emphasis on real-time changes [1] and the fact that it aligns with the design of standard engineering flight simulators [2]. However, FlightGear does not utilize that architectural style due to the direct interactions between components and the fact that there is not a sole broadcasting system within the source code. By considering the arrangement of the source code and the given reasons why we didn't think it utilized the Client-Server or Repository styles, we concluded that there is no other feasible architectural style other than Object-Oriented.

Subsystem Descriptions:

After examining the top-level architecture of FlightGear using Understand, it is clear that the architecture is incredibly intricate and uses a large number of moving parts that interact with one another. While referencing the subsystems of our proposed conceptual architecture, we observed that the AircraftMovement subsystem we introduced was difficult to categorize due to its components being applicable to UserRequests and Navigation. Other directories, such as utils, test_suite, and 3rd_party were not included as our concrete architecture focuses on the bulk of the functionality of FlightGear, which is handled by the src directory.

Aerodynamics – This subsystem is comprised of the FDM component of the source code. It is primarily responsible for determining variables related to physics, atmosphere, thrust, wind, movement, and flight properties [3]. Aerodynamics depends on input from UserRequests,

Images, and AircraftCharacteristics, it affects the Navigation and Weather subsystems, and has a mutual interaction with FlightSimulation (i.e. the overall simulation).

Aircraft Characteristics – This subsystem is concerned with specific aspects of the aircraft being used, such as its initial state, flight history, support code, and any recording functionalities of its flights [3]. In regards to the diagram, it is comprised of the Aircraft and Systems components. It is mainly depended on by a multitude of subsystems (i.e. Navigation, FlightSimulation, Aerodynamics, and UserRequests) due to its impact on how the aircraft currently in use responds to the user's requests and behaves in the simulation.

Flight Simulation – This subsystem oversees the overall flight simulation software. The most significant elements of this system are Main, Viewer, and Scripting, which manage the software's initialization, global state variables, command-line routines, multi-platform support, rendering, and camera management [3]. FlightSimulation also contains components manage add-ons to the software (i.e. Add-ons), AI objects within the simulation (i.e. AIModel), and any multiplayer functionality (i.e. MultiPlayer and Network) [4], as well as those that are not depended on by other subsystems (i.e. Include and EmbeddedRoutines). This subsystem interacts with all the other subsystems to keep the simulation current.

Images – This is responsible for dynamically managing displays, textures, camera behaviour, 3D model orientations, and calculating scenery tiles [3]. As such, this system consists of the Canvas, Cockpit, Model, and Scenery components. The Images subsystem has numerous mutual interactions/dependencies with systems such as Navigation, Sounds, FlightSimulation, and UserRequests, and direct dependencies from Weather and Aerodynamics.

Navigation – The Navigation subsystem comprises a large number of source code components, specifically Airports, Instrumentation, Navaids, Traffic, and Autopilot. It is responsible for predicting future values in the simulation, keeping track of current variables related to the flight, managing traffic, and determining flight paths [3]. It is mutually dependent with Images, UserRequests, Weather, and the overall flight simulation, and requires input from Aerodynamics, Sounds, and AircraftCharacteristics to compute the variables needed for accurate navigation.

Sounds – This subsystem is one of the simpler components of FlightGear's architecture, as it mainly manages the generation of sounds. It comprises of the Radio and Sound components, is depended on by Navigation and UserRequests, and has mutual interactions with Images and the overall flight simulation (where it outputs sound).

Time – This subsystem only contains the Time component of FlightGear's source code. It is also one of the simpler subsystems, as it does not depend on any other subsystems other than FlightSimulation. Its primary functionality is managing the time of the whole simulation.

User Requests – UserRequests handles inputs from the user from external interfaces and their interactions with the graphical user interface (GUI) and air traffic systems. As such, the Input, GUI, and ATC components from the source code are applied here and determine the behaviour of a wide number of subsystems within the architecture [3].

Weather – In our proposed categorization, the Weather subsystem is only comprised of the Environment component and relates to any natural factors that may affect the aircraft's behaviour and the rendering in the simulation [3]. It depends on input from Aerodynamics, Images, and Time, and has a mutual dependency with Navigation and the overall flight simulation program.

Flight Dynamic Model Analysis:

Based on the concrete architecture given by the dependencies and method invocations given in FlightGear's source code, the Aerodynamics subsystem is responsible for managing the Flight Dynamic Model (FDM) component. As stated earlier, the FDM is used to calculate the physics and aerodynamics variables observed during aircraft flight and environmental changes, like drag and lift [5]. The JSBSim and YASim directories determine the significant outputs of the subsystem using relevant computations and routines for speed, atmospheric forces, propulsion, etc. [6, 7]. The LaRCSim and UIUCModel components perform similar functions to those mentioned but were replaced by the JSBSim and YASim models. The SP directory houses different flight dynamics models that are not predominantly used throughout the simulation. Finally, AIWake manages wake turbulence, and ExternalNet and ExternalPipe manage any external flight dynamics models introduced to the system.

The Aerodynamics component (in other words, the flight dynamics model) interacts with the Navigation, Weather, AircraftCharateristics, Images, UserRequests, and FlightSimulation subsystems. It is depended on by Environment and Navigation, specifically to share flight model parameters, determine environmental factors, and determine variables of the head-up display, as seen in the flight.hxx source code. Likewise, it depends on the GUI of UserRequests, Images for the management of 3D models and scenery, and AircraftCharacteristics for the aircraft's control interface. Finally, it interacts mutually with the overall flight simulation system to display and receive changes to Main, the multiplayer functionality and its network, and any AI-controlled objects. Figure 2 provides an in-depth look into which components the FDM subsystem specifically interacts with.

When comparing the use of this subsystem between our conceptual and concrete architecture, we notice that the conceptual architecture is somewhat accurate to the actual interactions of Aerodynamics. As seen in our proposed conceptual Publisher-Subscriber architecture in Figure 3, we included Aerodynamics to be an individual subsystem that only has one mutual interaction with the overall flight simulation and no other components. Due to the nature of the Publisher-Subscriber architecture style, we assumed that Aerodynamics/ the FDM received information from other subsystems through the broadcasting system, but we did not consider what exactly that information was and how it affected the subsystem. Therefore, it is not only faster for the FDM to receive information of necessary subsystems through direct method invocations, but also provides a clearer idea of what subsystems actually interact with the FDM.

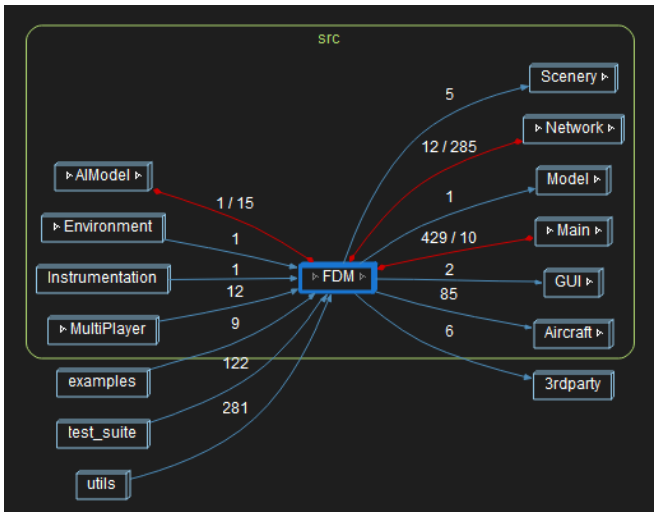

Figure 2. Dependency graph of the FDM component from FlightGear's source code.

High-Level Architecture Reflexion Analysis:

Initially, FlightGear was conceptualized using a Publisher-Subscriber model, shown in Figure 3, due to its real-time communication and responsiveness capabilities which are important for a realistic flight simulation environment. Different components of the simulator needed to exchange information and updates efficiently, and the Publisher-Subscriber model appeared to facilitate this requirement. However, further analysis revealed a shift towards an Object-Oriented architecture, as illustrated in Figure 4. This shift arose from observing subsystems interacting directly, contrary to the Publisher-Subscriber model's centralized communication. These are some of the major factors that highlighted the benefits of changing to an Object-Oriented architecture:

1. Complexity Management: The Object-Oriented architecture allows encapsulation, inheritance, and polymorphism, Publisher-Subscriber model struggles with this due to its less direct method handling of interactions and modifications as seen in Figure 3. By structuring the simulation with classes and objects that mirror various elements of the flight environment, developers could more easily expand and maintain the system. Looking at Figure 4, the direct interactions between subsystems like 'Aerodynamics' and 'Navigation' show the necessity for a more cohesive structure that Object-Oriented design offers and Publisher-Subscriber does not.

2. Modularity and Reusability: The conceptual architecture of FlightGear as shown in Figure 3 displays modularity through its clear division of subsystems like 'Weather', 'Navigation', and 'AircraftCharacteristics'. These components can be independently developed and tested, showcasing the modularity in an Object-Oriented architecture. This modularity is important and makes the most sense for an open-source project like FlightGear, where contributions come from a diverse community of developers. However, Figure 4 the Publisher-Subscriber architecture where modules like 'Weather' and 'Time' are interlinked with numerous other components, indicating a web of dependencies. This may get in the way of individual component testing and reuse, in turn affecting modularity and reusability negatively.

3. Direct Interactions for Real-Time and Fast Performance: The real-time nature of flight

simulation needs immediate feedback and adjustments based on user inputs and environmental conditions. The direct method calls and data access provided by an Object-Oriented architecture allows these quick interactions, offering a more responsive and realistic simulation experience. Figure 3 illustrates this principle with direct dependencies (blue arrows) such as the connection between 'navigation' and 'Sound', In contrast, Figure 4 we do not see any direct relationship between the two components. In Publishers and Subscribers architecture publishers and subscribers don't talk directly to each other, and it can cause delays. These delays can make a simulation slower and less realistic.

2nd level subsystem Reflexion analysis:

1. Aerodynamics:
    - Conceptual: Initially assumed to interact with the flight simulation through a Publisher-Subscriber model.
    - Concrete: Direct interactions are noted with UserRequests, Images, and AircraftCharacteristics, influencing the Navigation and Weather subsystems and interacting mutually with FlightSimulation.
    - Rationale: The direct method seen in the concrete architecture provides faster and more direct communication between subsystems which is crucial for real-time simulations, which the Publisher-Subscriber model could not efficiently provide due to its broadcast nature.
2. Aircraft Characteristics:
    - Conceptual: Had a more isolated role with fewer dependencies.
    - Concrete: Found to have multiple dependencies and interactions with Navigation, FlightSimulation, Aerodynamics, and UserRequests.
    - Rationale: The Object-Oriented architecture revealed the interconnectedness and the complex role of Aircraft Characteristics in the simulation, contrary to the conceptual view which did not account for such extensive interactions.
3. Flight Simulation:
    - Conceptual: Considered the core system managing real-time updates and state changes in a Publisher-Subscriber model.
    - Concrete: Acts as a central hub with mutual interactions with almost all subsystems, suggesting a more integrated system than initially conceptualized.
    - Rationale: The shift towards an Object-Oriented design became necessary to handle the intricate dependencies and to ensure the seamless incorporation of community contributions and new features.
4. Images:
    - Conceptual: Thought to be a separate subsystem handling visual aspects with limited interactions.
    - Concrete: Involved in numerous mutual interactions/dependencies, indicating a more central role in the simulation process.
    - Rationale: The interconnection between the Images subsystem and others like Navigation and UserRequests highlighted the need for a tightly integrated architecture to manage the dynamic visual aspects of the simulation.
5. Navigation:
    - Conceptual: Expected to function primarily on inputs from the central broadcast system.

- Concrete: Shows mutual dependencies with Aerodynamics, Sounds, and Weather, indicating a complex web of interactions for managing flight paths and traffic.
- Rationale: The detailed dependency mapping in the concrete architecture shows that Navigation requires direct and immediate data from various subsystems, which would be less efficient in a Publisher-Subscriber model.

6. Sounds:
   - Conceptual: Considered a simpler subsystem with primary responsibilities for generating sounds.
   - Concrete: Interacts mutually with the Images and FlightSimulation subsystems, pointing to a more complex role.
   - Rationale: The Object-Oriented approach revealed that the Sounds subsystem has a broader impact on the user experience, influencing and being influenced by graphical changes and user interactions.
7. Time:
   - Conceptual: Not explicitly detailed in terms of interactions.
   - Concrete: Identified as having a simple, unidirectional dependency on FlightSimulation.
   - Rationale: The role of Time in the concrete architecture shows its specialized function in the simulator, which was underrepresented in the conceptual model.
8. User Requests:
   - Conceptual: Expected to be an interface for user inputs, with minimal detail on how it interacts with other subsystems.
   - Concrete: Involved in complex interactions with Navigation, Images, and Sounds, showcasing its central role in processing user inputs.
   - Rationale: The complexity and importance of the User Requests subsystem in processing and channeling user interactions across the simulation were underestimated in the conceptual model.
9. Weather:
   - Conceptual: Originally thought to be a stand-alone subsystem influencing the flight environment.
   - Concrete: Interacts with Aerodynamics, Images, and Time, and has mutual dependencies with Navigation and FlightSimulation.
   - Rationale: The intricate role of the Weather subsystem in affecting and being affected by various simulation parameters required an Object-Oriented approach to manage these dynamic interactions, contrasting with the conceptual isolation.

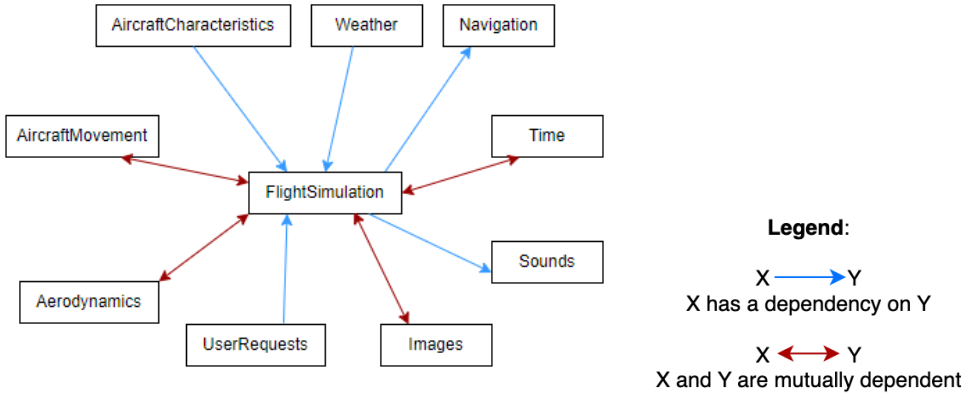

Figure 3. The proposed Conceptual Architecture of FlightGear.

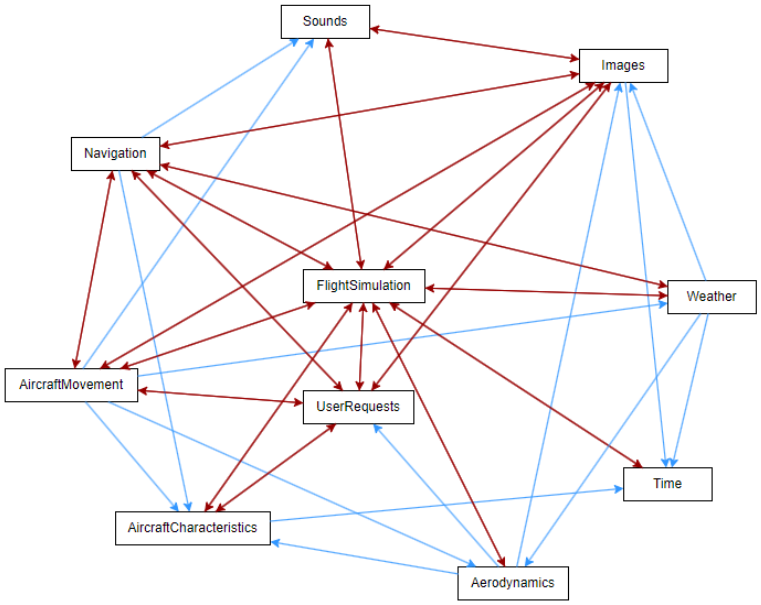

Figure 4. Diagram of the dependencies when directly referencing the Publisher-Subscriber architectural style, not including the subsystems 3rdparty, utils, test_suite, docs-mini, examples, and scripts.

External Interfaces

FlightGear utilizes many different external interfaces for functionalities that are not handled by the internal system. As can be seen in the source code, FlightGear contains many different application program interfaces for other external interfaces. As was discussed in the conceptual report, some of these are used for OpenGL and OpenAL, application programming interfaces that send graphical or auditory information to the graphics and audio drivers of the computer, assisting the rendering process. OpenAL is mainly used by the sound subsystem of FlightGear, as it is the subsystem that is responsible for rendering audio. OpenGL is used by the images subsystem for a similar reason, the images subsystem is responsible for rendering graphics, however it is also partially used by other subsystems, such as the environment subsystem, which plays a role in determining which graphics should be rendered.

Another interface being used by FlightGear is structured query language (SQL), a programming language widely used for database management. SQL allows storage of data into tables, sorted and joined by rows and columns. SQL can be used for many different purposes, however it seems that FlightGear mainly uses it to store information about navigation in a file written in C++, called NavDataCache.cxx. In this file, many SQL queries can be found inserting, removing, and deleting information to and from an SQL database, containing lots of properties of different navigational information.

Use Cases

Case 1: Decreasing velocity during flight simulation

Assume a user is already running a flight simulation, and they choose to decrease the velocity at which their plane is flying. They will make this change through the user interface, which will interact with the user requests subsystem, which is responsible for all inputs through the GUI. This subsystem will process the request and first communicate it with the navigation

subsystem. Upon receiving this request, the navigation subsystem will need to update certain values such as the estimated time to land or the current flight path. In order to update this information, it will call upon three other subsystems, those being aerodynamics, sounds, and aircraft characteristics. The aerodynamics subsystem will provide information about the physics of the simulation that will be useful towards the calculations being performed by navigations. The aircraft characteristics will also provide useful information about the way that this specific aircraft will react to a decrease in velocity. Finally, the sound subsystem needs to be updated as well in order to correctly reflect the sounds that a slowing aircraft would make. Once this information has been updated, the navigation subsystem will be able to finish the functions it was performing, and then return the results to user requests. User requests will call the images subsystem, which is responsible for generating textures for the environment. The subsystem will update the rate at which it generates these graphics, and then return to user requests, which will also return its results to the user.

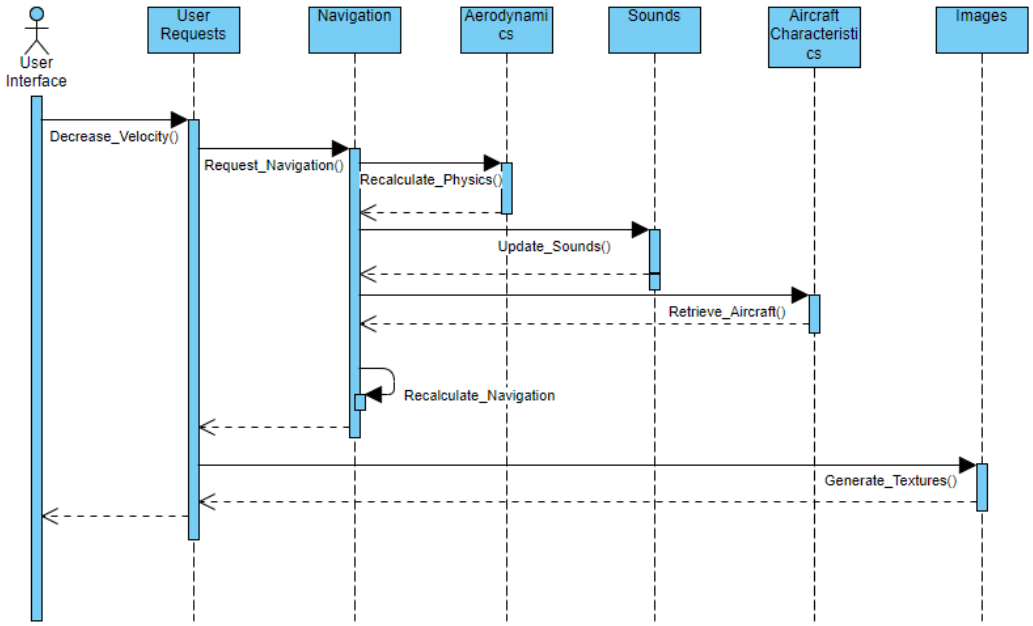

Figure 5. Sequence diagram for use case 1.

Case 2: The user sets the weather to rainy

Assume a user is already running another flight simulation, and they go into the settings and choose to set the weather to rainy, as opposed to sunny. Similar to the previous scenario, this case will start by the user interacting with the user requests subsystem through the user interface. Weather will be the first affected subsystem, however the user requests subsystems does not have any dependencies on the weather, meaning that it will first interact with the flight simulation subsystem, which will then submit a request to the weather subsystem. In the weather subsystem, functions will be called to update variables about the environment, and then a call to the images subsystem will be made so that these changes in weather will be reflected graphically. After this, the system will return to the flight simulator which will call upon the navigation subsystem, similarly to the last use case. Navigation is heavily affected by the weather, and so it is important

that the calculations are made to update the estimated flight path and other important variables. Just like in the last case, navigation will call on aerodynamics to update physical facts in reaction to the new change in weather, sound to add the sounds of rain, and aircraft characteristics in order to retrieve important information about how the in use aircraft reacts to changes in weather. Once all this has been done, the navigation subsystem will return to the flight simulator. After the flight simulator has been entirely updated, similarly to the previous case these changes will be reflected in the user interface.

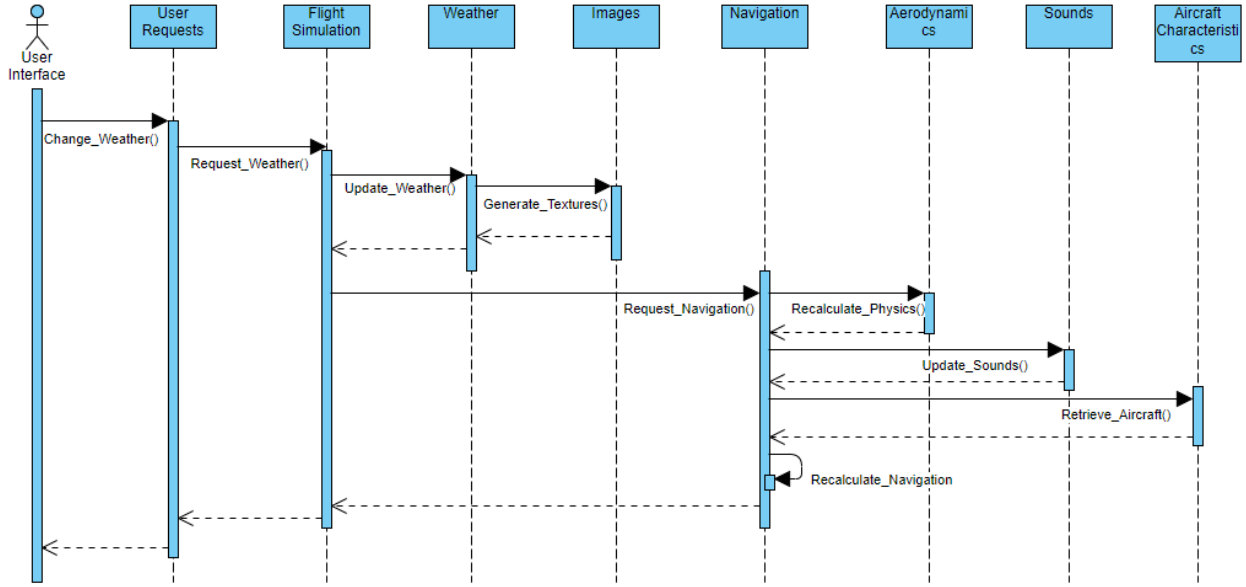

Figure 6. Sequence diagram for use case.

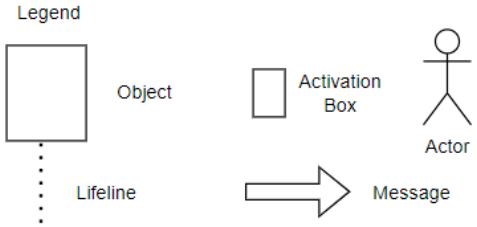

Figure 7. Legend for sequence diagrams.

Data Dictionary

Adaptability. The capability of adjusting to new conditions.
Component. A software object that encapsulates a certain functionality or a set of functionalities, capable of interacting with other software objects.
Dependency. A relationship between software components where one component relies on the other to work properly.
Directory. A hierarchy of entries, each containing a set of attributes, used to store data.
Encapsulation. The bundling of data and methods operating on this data into one unit.
Flexibility. The capability of adapting to different situations, environments, and needs, without requiring significant changes to code or structure.

Inheritance. The mechanism of basing an object or class upon another object or class, retaining similar implementation.
Modularity. A logical partitioning of components that makes otherwise complex software manageable in terms of implementation and maintenance.
Object-Oriented. A design paradigm based on the division of responsibilities for an application or system into individual reusable and self-sufficient objects.
Open-source. Denoting software for which the original source code is made freely available and may be redistributed and modified.
Pipe-and-Filter. An architectural style that divides responsibilities for an application or system into individual reusable and self-sufficient objects.
Polymorphism. The phenomenon in which something occurs in several different forms.
Publisher-Subscriber. Also known as the "implicit invocation" architecture style, it consists of a loose collection of components that do not communicate with each other individually, but rather take part in the system through a connecting procedure they can subscribe to.
Scalability. The capability to increase or decrease system performance and cost in response to changes in demand.
Subsystem. A computer system that is part of a larger computer system.

Naming Conventions

ATC. ATC stands for "air traffic control."
FDM. FDM stands for "Flight Dynamic Model."
GUI. GUI stands for "graphical user interface."
OpenAL. OpenAL stands for "Open Audio Library."
OpenGL. OpenGL stands for "Open Graphics Library."
SQL. SQL stands for "structured query language."

Lessons Learned

One of the initial steps in our project was to look into FlightGear's extensive source code. This was both insightful and challenging. We learned the importance of familiarizing ourselves with the architecture and coding conventions of an existing system, which required patience and much attention to detail. Understanding the codebase was very important for correctly identifying the architectural style FlightGear uses and for mapping out the interactions between various subsystems.

A major obstacle we faced was related to downloading the FlightGear system and effectively utilizing the Understand tool for architectural analysis. The tool was a great help in generating dependency graphs and exploring as well as understanding subsystem interactions, but we had various technical issues from downloading the software to compatibility issues. Also once we were finally able to access the systems we faced another challenge in familiarizing ourselves with its features and capabilities, it required a steep learning curve.

Navigating through FlightGear's design, we initially thought that the Publisher-Subscriber model would be a good fit to model the system. The system's concrete design suggested a more interwoven architecture than we first imagined. We then thought about the possibility of a Client-Server or even a Repository model, which on the surface seemed to fit with the structure of the system's data management and operations. However as our analysis deepened, we found these styles did not accommodate the direct and intricate communication evident in the code. In

the end, as reflected in the report we chose the Object Oriented architecture that we thought best modeled the system.

Conclusion

To conclude, it is clear that the path from conceptualization to realization within software development is one marked by continuous learning, adaptation, and refinement. The exploration of FlightGear's architecture, from its initial Publisher-Subscriber framework to the adoption of an Object-Oriented design, underlines the importance of flexibility and adaptability in the era of evolving project requirements and technological landscapes.

The process of dissecting FlightGear's architecture has been invaluable, providing profound insights into the complexities and intricacies of developing a high-fidelity flight simulation platform. Furthermore, through the analysis of subsystem interactions, the critical role of the open-source community in shaping FlightGear became evident, highlighting the symbiotic relationship between architectural design and community engagement. As we delved into this connection we found many lessons to be learned, extending beyond the specifics of FlightGear's architectural style. The challenges encountered in aligning theoretical models with practical realities reaffirmed the necessity for a deep understanding of the underlying system, the significance of community contributions, and the need for a robust and flexible architecture capable of accommodating both current functionalities and future expansions. Moreover, this report serves as a testament to the collaborative effort of our team, reflecting our collective commitment to unravelling the nuances of FlightGear.

All in all, FlightGear's architectural journey from a conceptual model to a concrete implementation is a vivid illustration of the dynamic nature of software development. It showcases how theoretical architectural principles must be tempered with practical considerations to create a system that is not only technically sound but also responsive to the needs of its users and adaptable to the inevitable technological advancements. As FlightGear continues to evolve, it stands as an icon for open-source development, demonstrating the power of community collaboration and the value of architectural adaptability.

Works Cited

[1] Garlan, D., & Shaw, M. (1993). An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, *vol. 1*, pp. 1-39. onq.queensu.ca/d2l/le/content/861602/viewContent/5155035/View.

[2] Allerton, D. (2022). *Flight Simulation Software : Design, Development and Testing*. John Wiley & Sons Inc.

[3] *FlightGear Source Code.* Github. https://github.com/FlightGear/flightgear/tree/next/src.

[4] *AI Systems*. (2019, Nov. 21). FlightGear Wiki. Retrieved Mar. 21, 2024, from https://wiki.flightgear.org/AI_Systems.

[5] *Flight Dynamics Model*. (2022, Nov. 9). FlightGear Wiki. Retrieved Mar. 22, 2024, from https://wiki.flightgear.org/Flight_Dynamics_Model.

[6] *JSBSim*. (2022, Mar. 31). FlightGear Wiki. Retrieved Mar. 21, 2024, from https://wiki.flightgear.org/JSBSim.

[7] *YASim*. (2023, Jul. 21). FlightGear Wiki. Retrieved Mar. 21, 2024, from https://wiki.flightgear.org/YASim.