

Milestone 1

Flight Data Management System



October 15, 2021

StudentS: Jayson Biswas, Shaq Purcell, Taimoor Salam, Jaydan Zabar

SENG3020 – Advanced Software Quality

Table of Contents

[FDMS Design Overview 2](#_Toc85222339)

[High level Design Summary 2](#_Toc85222340)

[Class Diagrams 2](#_Toc85222341)

[USE case Diagrams & Descriptions 4](#_Toc85222342)

[Ground Terminal Software Design 6](#_Toc85222343)

[Modules 6](#_Toc85222344)

[Design Patterns 7](#_Toc85222345)

[SOLID 8](#_Toc85222346)

[GUI 9](#_Toc85222347)

[Ground Terminal Database Design 10](#_Toc85222348)

[Architecture 10](#_Toc85222349)

[Database Engine 10](#_Toc85222350)

[Table Definitions 10](#_Toc85222351)

[Connection 11](#_Toc85222352)

[Aircraft Transmission Design System 12](#_Toc85222353)

[Design Pattern 12](#_Toc85222354)

[Module DescrIption 12](#_Toc85222355)

[SOLID 13](#_Toc85222356)

[Diagram 14](#_Toc85222357)

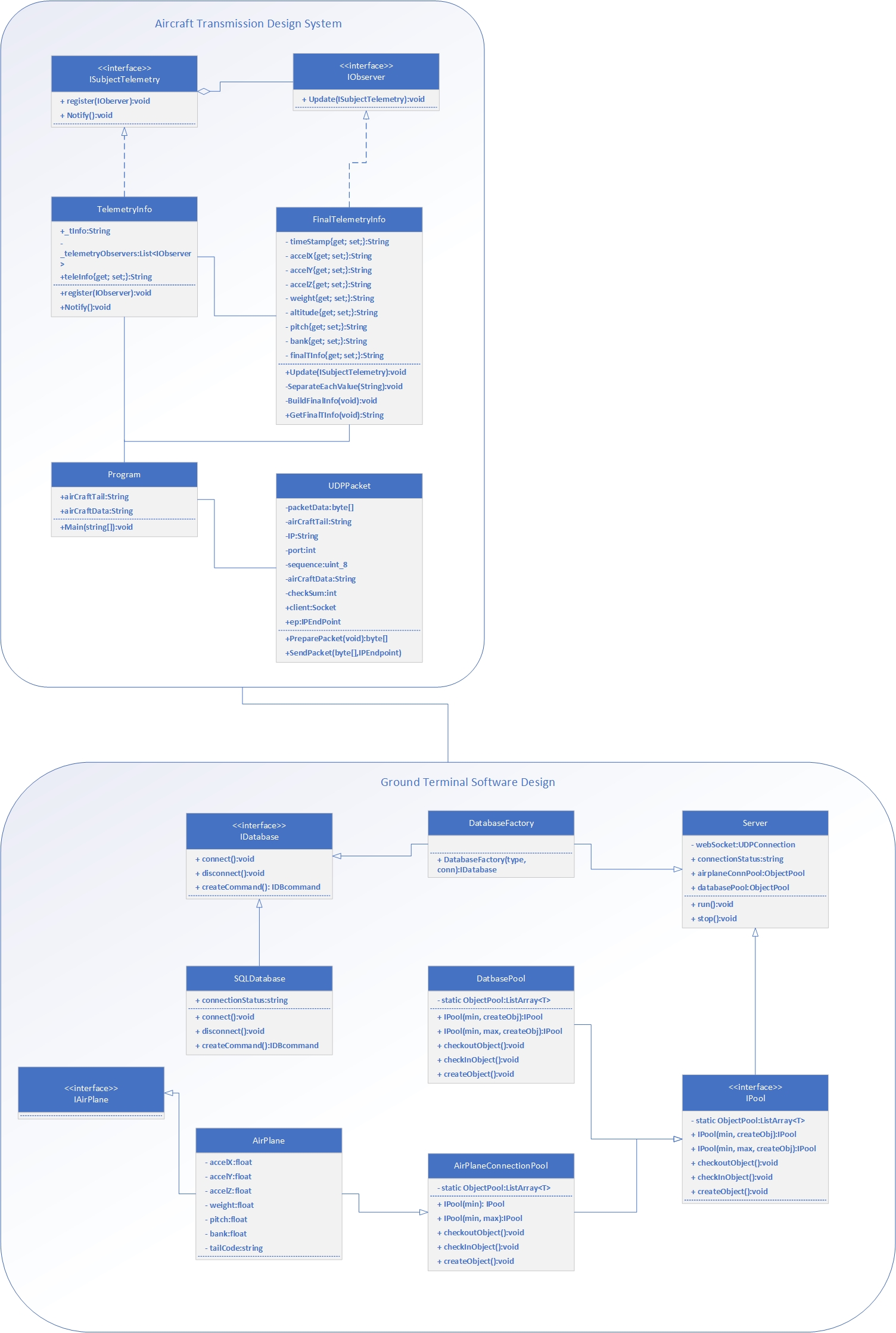
[References 15](#_Toc85222358)

# FDMS Design Overview

## High level Design Summary

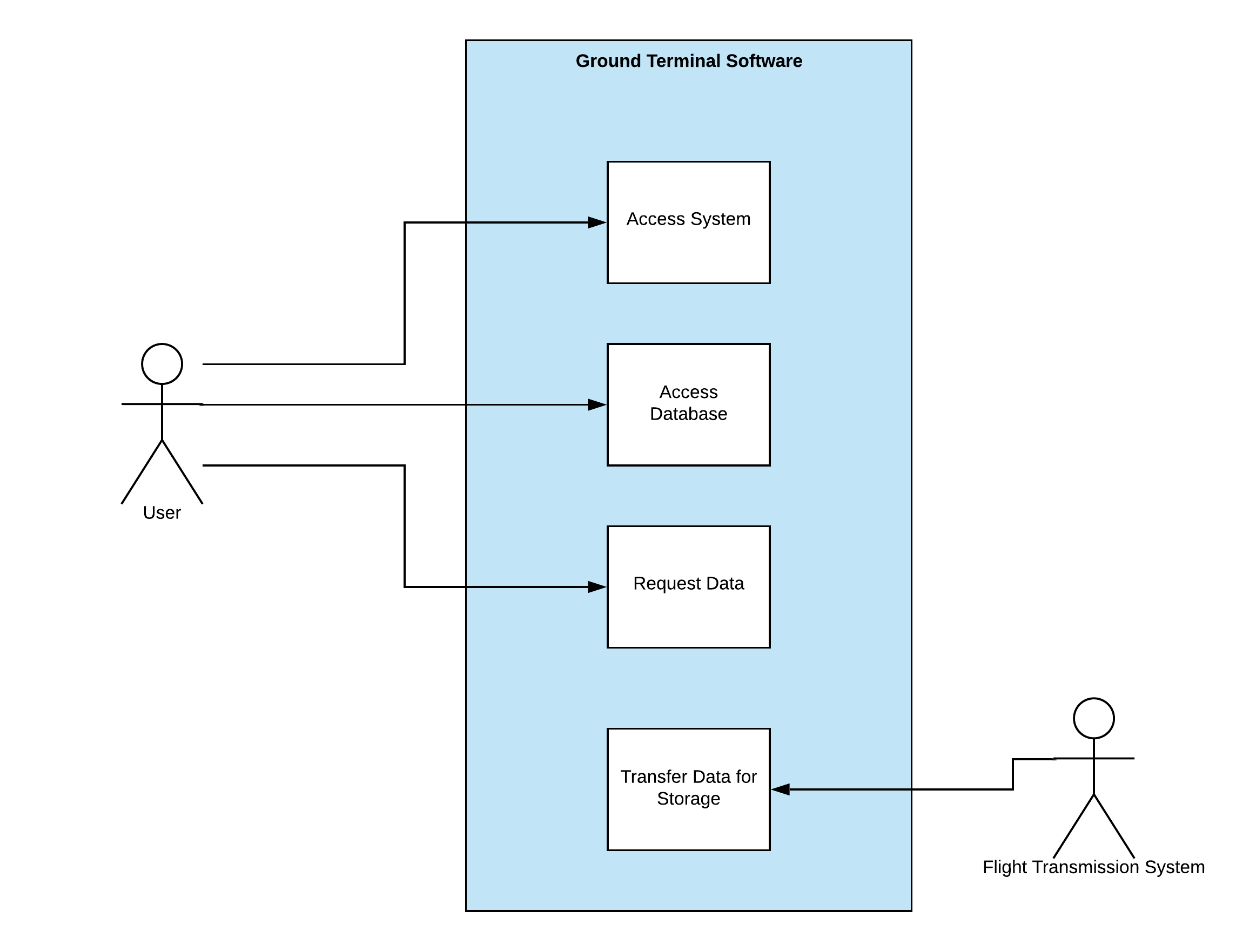
Our Flight Data Management System (FDMS) will be designed in C# for the client side and ASP.NET for the server side. The design patterns we are utilizing are Factory, Object Pool and Observer, which are categorically different patterns being used for different components. To specify, the Aircraft Transmission system will be a simple Windows program that will run constantly, collecting data to send to the Ground Terminal Software. This collected data is the aircraft telemetry that includes the timestamp at which the telemetry is received, the G-Force and altitude parameters. This system utilizes the Observer pattern. The processed string is then sent by the UDPPacket class in a packet to the Server classes webSocket contained in the Ground Terminal Software. This can be accessed by the User, to which he/she can access the Database in which all flight Data is constantly stored and updated. The User can request a report of any flight data he/she desires with the flight ID. The Ground Terminal Software then utilizes the Object Pool and Factory Pattern. The combinations of these design patterns offer a clean design approach that we are trying to achieve with our version of the FDMS.

## Class Diagrams



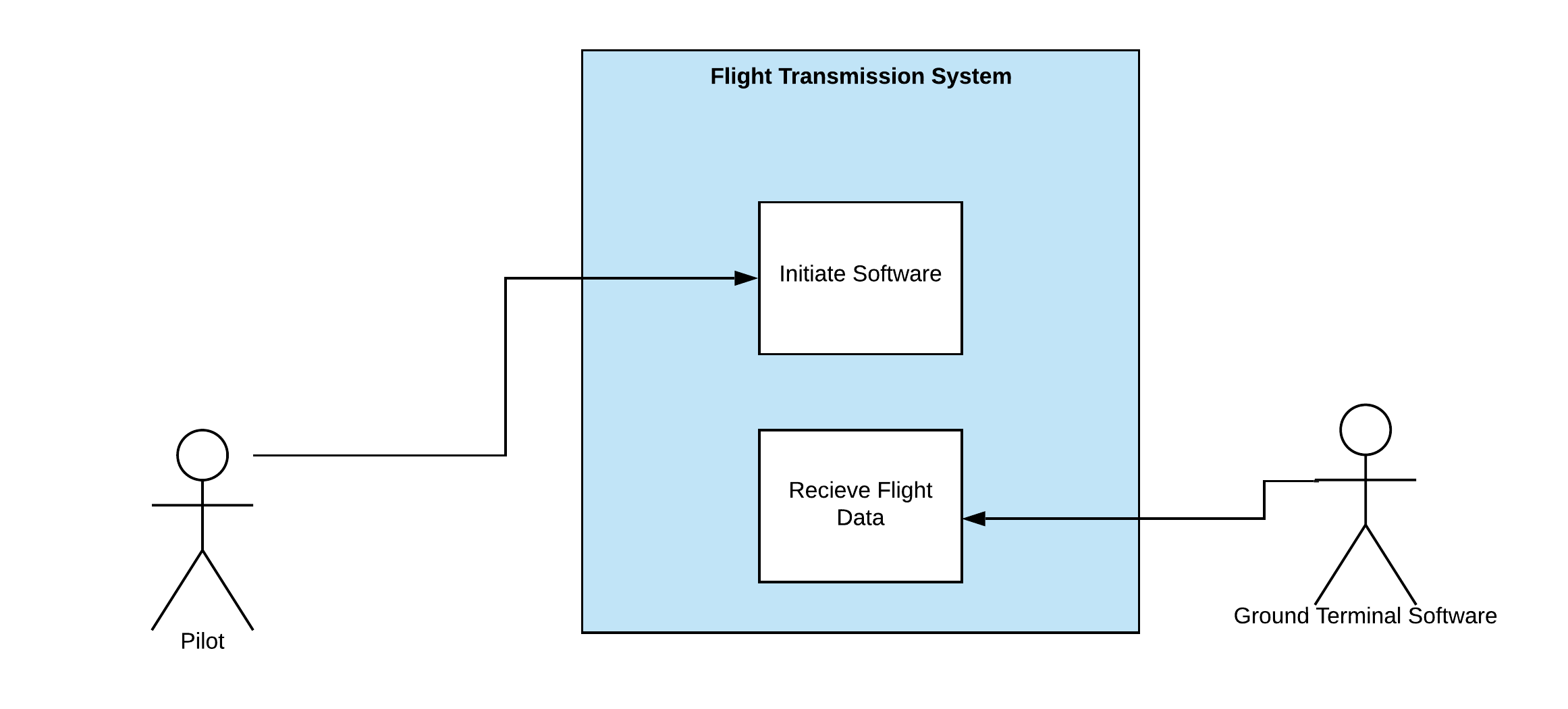
**Figure 1: Class Diagram of Entire System**

## USE case Diagrams & Descriptions



**Figure 2: Ground Terminal Software Use Case Diagram**

|  |  |
| --- | --- |
| Goals: | User (Ground Operator) requests data report |
| Actors: | User |
| Trigger: | User opens up the system |
| MAIN: | 1. User enters security Code 2. System verifies that the User has clearance 3. System presents User with Control Panel 4. User selects the Database 5. System presents the Database 6. User selects desired flight ID 7. System presents flight data and diagnostics 8. User requests data for analysis 9. System transfers data as a report |
| EXCEPTION: | 2.1. **IF** user code is invalid **THEN** revoke access  2.2. **IF** user does not enter code **THEN** revoke access  2.3. **IF** user enters invalid code 5 times **THEN** lock out of system **AND** end use  6.1. **IF** user selects/enters invalid ID **THEN** prompt error **AND** return user back to Main Step 5 |



**Figure 3: Flight Transmission System Use Case Diagram**

|  |  |
| --- | --- |
| Goals: | User (Pilot) sends flight data |
| Actors: | User |
| Trigger: | User initiates aircraft |
| MAIN: | 1. System collects all flight data 2. System sends data to Ground Terminal Software 3. System stores data into database |
| EXCEPTION: | 2.1. **IF** System detects change in telemetry **THEN** return to Main Step 1 |

# Ground Terminal Software Design

## Modules

**IDatabase:** Interface for how a database object should be created. Each database object needs to be able to produce is own commands using a Command Pattern, and connect via connection string

**SQLDatabase:** The concreate version of IDatabase, this maintains connection to the main Ground Terminal Software Database.

**DatabaseFactory:** This class creates a database class to manage its own connection based on the type and connection string passed. Class supports SOLID design principles by loosely coupling database types to the database used

**Server:** A singleton class containing the webSocket used to facilitate communication between airplane clients and the ground terminal server. This class also contains object pools that manage all connections to the server. This is also the starting execution point of the application.

**IPool:** An interface for ObjectPools, this class determines the functionality of a pool. A pool must have a list of objects that it manages and a way to checkout and check in the objects that are being used elsewhere in the application

**DatabasePool**: A class that keeps track of all the database connections to the main application.

**AirplanePool**: A class that keeps track of all the airplane connections to the main application.

**Airplane:** A class that represents a record from the database it is a combination of the tables G-Force Parameters and Altitude Parameters.

**IAirplane:** An interface for that describes the parameters and functions of an Airplane.

A picture containing timeline

Description automatically generated

**Figure 4: Ground Terminal Software Class Diagram**

## Design Patterns

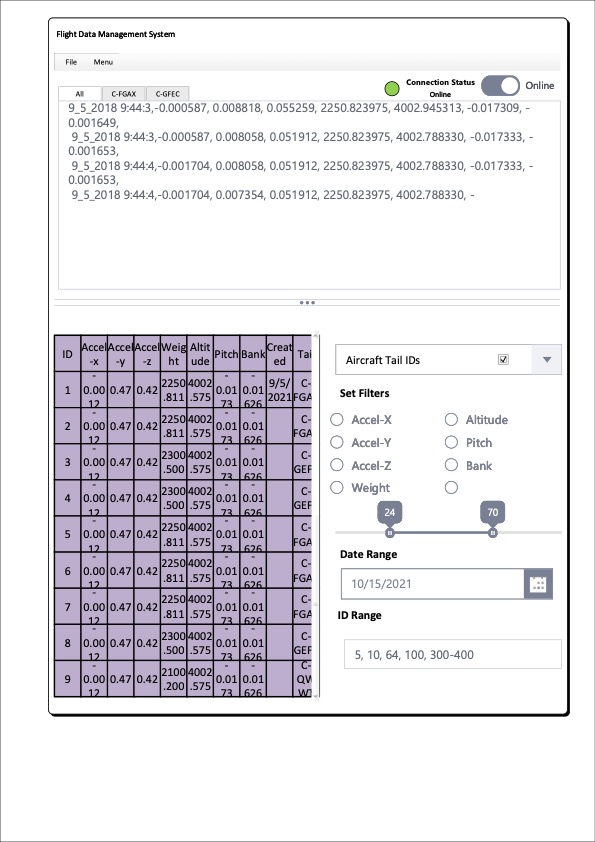
For the Ground Terminal Software, we have utilized two main patterns: The Factory pattern and the Object Pool pattern. The Factory pattern is used to manage the types of databases that connect to the application. Even though the requirements state that we are using Microsoft SQL Server, this allows the application to be extended to other database connections if the requirements ever change. The Object Pool pattern is used to manage all connections that come within the server. This allows for expensive connections objects to be instantiated once, and then released from the object pool to be used by the application when needed.

## SOLID

This is not entirely SOLID. But this uses most of the SOLID principles. Here is how:

1. Classes within the application are small and have one purpose.
2. There are interfaces to segregate the business logic from the design patterns in code, satisfying the I in the SOLID acronym (IAirplane, IDatabase)
3. The children of interfaces, are open for extending but are not easily modifiable
4. Since we have used concrete subject class and concrete observer class, the Dependency Inversion Principle is not followed.

## GUI



**Figure 5: UI Mock-up Flight System GUI**

# Ground Terminal Database Design

## Architecture

The database itself is relatively simple as shown in the figure below and is only composed of three tables. There are two tables that hold the G-Force and Altitude parameters are named respectively. All possible air tail codes have been added to their own table, ‘Aircraft Tails’, this normalizes the database to second normal form, which optimizes the database without making it harder to manage. This means that the data held in both Altitude and G-Force parameters

Every record entered into the database will be stamped with the time and date that it is was created using the CURRENT\_TIME function, this value is saved in column dateCreated. Date values are also stored when a record is updated or deleted. Values that have been deleted from the database are set with a date, but the values are kept, by setting the dateDeleted acts as a Boolean.

## Database Engine

The database engine in use is Microsoft SQL server as per the application requirements. The storage engine that is used is InnoDB. Other viable storage engine choices were analyzed, all providing less functionality than InnoDB (, no foreign keys and the lack of transactional functions).

## Table Definitions

Table

Description automatically generated

**Figure**

## Connection

The database will be hosted remotely using Azure. The Azure service being utilized is the `SQL Database’, which spins up a database in the cloud using the aforementioned database engine (Microsoft SQL Server 2019). The connection being opened using the Open Database Connectivity (ODBC) this will allow for interoperability between another Microsoft application (Excel, Word, Visio, etc.), if the feature requirements extend.

# Aircraft Transmission Design System

## Design Pattern

For the Aircraft Transmission System, we have used the Observer Design Pattern. Because the Observer design pattern allows the program to change variables and notify in real time. This means, every time the telemetry information changes, the program can detect it and act accordingly. This design pattern also allows future development as it has the scope to add more observers. This means, in future, if there is another ground application along with the current one, where this program needs to send data, it can easily do that with simple code extension.

## Module DescrIption

**Program:** This is the main class where the application will run from. The Main() function is here. This function creates instances of TelemetryInfo and FinalTelemetryInfo to get the airCraftData. The airCraftData consists of the “Body” of the final packet. This function takes the txt file, reads it and changes the value of “String teleInfo” in TelemetryInfo class. TelemetryInfo class notifies the current observer (which is FinalTelemetryInfo), that the information has been changed. Then the FinalTelemetryInfo takes the string and builds the body of the packet. This string is called from the main function via a get method. The Main() function then saves the airCraftData and sends it to the UDPPacket class. Thus, each line of the txt file is sent as a packet.

**TelemetryInfo:** This module is a concrete subject class that can register new Observers and notify once the “\_tInfo” variable is updated or changed. This class implements “ISubjectTelemetry” interface. The Main() function directly changes the “\_tInfo” variable. Once that happens, the Notify() method is used to send notification to the FinalTelemetryInfo observer class.

**ISubjectTelemetry:** This interface stands between IObserver and TelemetryInfo. The Notify() method uses Update() method that belongs to the FinalTelemetryInfo class. So, to communicate between ISubjectTelemetry and FinalTelemetryInfo, we created IObserver interface.

**IObserver:** This interface has the Update method to be implemented by FinalTelemetryInfo.

**FinalTelemetryInfo:** Once the Update method is triggered, the FinalTelemetryInfo class takes the given string, separates values from the string, builds the final body of the data and saves it in a variable called finalTInfo. The value of this variable is then called by the Main() function via GetFinalTInfo() function.

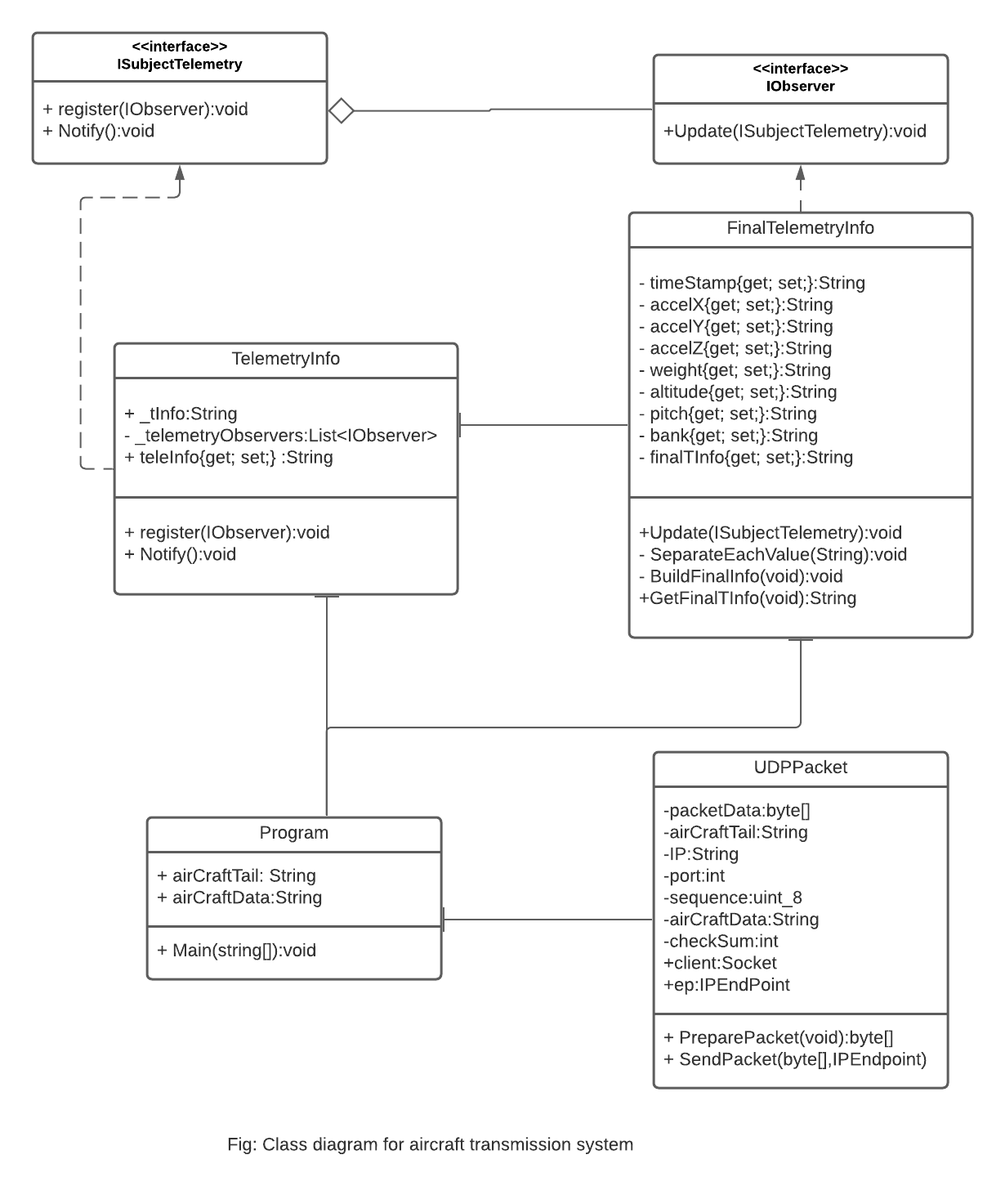
**UDPPacket:** This is an independent class. This takes the value of the airCraftTail and airCraftData from the Main() function. Then it prepares the packet with header, body and trailer. After that, it sends the packet via UDP.

## SOLID

This is not entirely SOLID. But this uses most of the SOLID principles. Here is how:

1. Each method does only one thing. Small methods are being used.
2. New observer can be created without changing the code. In short, code can easily be extended without any issue.
3. Since we don’t have a child class in this design, there is no possibility of breaking Liskov Substitution Principle. But child classes can be created here without breaking parent’s type definition.
4. Interfaces have been used to connect the functionalities of classes. Here, we used ISubjectTelemetryInfo and IObserver interfaces in order to enable TelemetryInfo class to notify FinalTelemetryInfo class about the variable change.
5. Since we have used concrete subject class and concrete observer class, the Dependency Inversion Principle is not followed.

## Diagram



# References

Source Making. (2007-2021). *Design Patterns & Refactoring*. https://sourcemaking.com

Source Making. (2007-2021). *Observer Design Pattern*.  <https://sourcemaking.com/design_patterns/observer>

Observer Design Pattern (C#). <https://www.youtube.com/watch?v=8nICu7ZSPtE&t=338s>

UML Use Case Diagram. LucidChart. <https://www.lucidchart.com/pages/uml-use-case-diagram>

UML Class Diagram Tutorial. LucidChart. https://www.youtube.com/watch?v=UI6lqHOVHic&t=499s&ab\_channel=Lucidchart