Design Specification Document

Team Android Optimizers

Jose Franco Baquera
Atiya Kailany
Jared Peterson

CS 448

December 7, 2019

Table of Contents

1. Introduction

The purpose of this document is to outline in a clear and coherent manner the design specification of the *Autonomous Robotic Vehicle (ARV) Application* project. More specifically, this document will describe in detail how the system was built and how it operates to implement the requirements mentioned in the *Requirements Specification* document. Special emphasis is placed on the system's publish-subscribe architecture with *Spitfire*, as well as on the user interface's design and the influences that drove specific design choices. Someone unfamiliar with the system, but familiar with the problem, will understand the system's design after reading this document. Important information that will be covered and described in the following sections include platforms, languages, environments, system architecture, system design, user interface design, deployment design, and coding standards.

1.1 Design Drivers

The main design drivers for the *ARV Application* project are modifiability, portability, user friendliness, and *ROS* connectivity. These design factors are described in more detail in the following four subsections.

- *Modifiability* – The application must be designed with modifiability in mind since it will be eventually used for other research projects that involve prototyping wearable interfaces. Furthermore, the application might be modified and used to control physical drones in the near future when more sophisticated technology emerges in the market. Ultimately, the ease of which the application can be modified to improve performance or adapt to changing environments will be of great concern for this project. Modifiability will be accomplished by several means, such as modular software.

- *Portability* – The application must be designed with portability in mind since it is expected that users will be operating it in various types of environments (i.e. both indoors and outdoors). This rules out designing the project as a computer application since computers are less portable than tablets. Furthermore, an Android application was preferred over an iOS one since Android applications are capable of running on more devices from different manufactures, prices, and even API levels. This directly supports portability since the application can be used from tablet to tablet.

- *User Friendliness* – The application must be designed with user friendliness in mind since it is expected that users will not need complex trainings or tutorials to learn how to use the application. Furthermore, the application cannot be confusing and must be easy to use. This rules out using complex buttons and visual design patterns that are used in more intricate video games. The application must also be designed for tablets and not smartphones since the latter does not provide enough screen size to add all the required functionalities.

- *ROS Connectivity* – The application must be designed with *ROS* connectivity in mind since it has to connect with *Spitfire*, a *ROS*-implemented server. It is important to note that communicating with a *ROS* server forces the application to be implemented as a publish-subscribe and client-server architecture (see section 3 for more information). Because of the previously mentioned observations, the application cannot be designed with other system architecture types or server protocols.

1.2 Acronyms, Abbreviations, and Definitions Used Throughout the Document

Term	Definition
Android API Level	An integer value representing the Android operating system version
Android Device	Physical Android device
Android OS	Android operating system
Android Studio	Official integrated development environment for Google's Android operating system
API	Acronym for *Application Program Interface*; a set of routines and protocols
ARV	Acronym for *Autonomous Robotic Vehicle*
Danger Zone	Drone and human danger zones that can be encountered by a drone while in-flight
EventBus	Library that implements a publish-subscribe event bus for Android applications
Game Topic	A *Spitfire* topic that can be subscribed to; examples include drone positions, game/user score, danger zones, and hidden objects
Hidden Object	Drone and human hidden objects that can be found by a drone while searching
iOS	A mobile operating system created and developed by Apple Inc.
JSON	Acronym for *JavaScript Object Notation*; An open-source, lightweight file format that allows for data interchanging
NMSU	Acronym for *New Mexico State University*
Parsing	Splitting a file or String into smaller pieces of data that can be easily manipulated
Publish	Occurs when *Spitfire* returns messages to ROSBridge
Register	Means to register a class to the *EventBus*
ROS	Acronym for *Robot Operating System*; An open-source operating system for robots that provides services such as hardware abstraction and message-passing between processes
ROSBridge	Protocol used to communicate to a ROS-implemented machine or server
Spitfire	The NMSU server that will be responsible for providing relevant game information
UML	Acronym for *Unified Modeling Language*
Virtual Drone / Drone	A drone object on the Android application user interface; does not imply an actual physical drone
XML	Acronym for *Extensible Markup Language*

Table 1. A list of the most widely used acronyms, abbreviations, and definitions in this document.

1.3 References

Khalaf, A., Pianpak, P., Alharthi, S., NaminiMianji, Z., Torres, C., Tran, S., Dolgov, I., and Toups, Z. (2018, May). "An Architecture for Simulating Drones in Mixed Reality Games to Explore Future Search and Rescue Scenarios". New Mexico State University.

Preece, J., Rogers, Y., & Sharp, H. (2015). *Interaction design: beyond human-computer interaction*. Chichester, West Sussex: John Wiley & Sons Ltd.

2. Platform, Languages, and Environment

This section will describe the platforms and environments chosen for our system, as well as the programming languages that we used for implementation. The following subsections lists this information in more detail.

2.1 Platforms

- *Operating System* – The application will be run on the Android 4.0 (Ice Cream Sandwich) operating system within an Android tablet. We chose this operating system because it is compatible with over 99% of Android devices. Operating systems with higher API levels will be chosen only if there is an increase in performance with such change. It is worth mentioning that the application will be able to run on any compatible Android device, but will be specifically designed for Android tablets only.

2.2 Languages, Build Tools, and Libraries

- *"Java"* – The programming language used to implement the system is Android Studio's Java. This programming language was selected, instead of Kotlin, since it is the most widely used for Android development. However, it is imperative to note that the Java in Android Studio is "Java-like" with small differences from the original Java programming language. For example, Android Studio's Java bytecode cannot be run by a Java Virtual Machine (JVM) but can with an Android Runtime (ART).

- *XML* – The application will use the XML markup language to implement some of its user interface. This particular language was chosen since it makes designing the user interface in Android Studio much easier.

- *Build Tool* – Within Android Studio, the application is being built with Gradle, which is an open-source build-automation system. We use Gradle because it is the default build tool used by Android Studio. After the build is complete, the application is then downloaded and installed on an Android tablet for use.

- *EventBus Library* – The application will heavily use the *EventBus* Android library, which is a library that implements a publish-subscribe event bus for

Android applications. Furthermore, the library version used is 2.4.1. This library and its version number are preset by ROSBridge Android and cannot be changed unless a different API is used.

2.3 Environments

No specific application frameworks are used to implement the system.

3. System Architecture

This section will document the high-level architecture of the system and includes an architecture diagram with accompanying text. The system is based on a hybrid between publish-subscribe and client-server architectural styles. In addition, there exist three main system components: a physical (or emulated) Android tablet, the *Spitfire* server, and the Robot Operating System (ROS). It is important to mention that a separation was made between *Spitfire* and ROS since the original architectural diagram provided to the team made this distinction clear (please refer to our *Requirements Specification* document for more information). A more in-depth reason as to why this separation was made, as well as additional information about all components, can be found in the following bullet points. In addition, we note that the Google Maps' servers were not included in the diagram and system architecture description since emphasis was placed on system components that can be directly modified by the Android application.

- *Android Tablet* – The first component in the system is the physical (or emulated) Android tablet. The chosen tablet will have the application installed in it and will display the user interface, as well as take user input.

- *Spitfire Server* – The second component in the system is the *Spitfire* server, which will store game information and will serve as an "intermediary" between the Android tablet and ROS. The reason why this is the case is because *Spitfire* is composed of not only ROS, but also other components. However, our project only heavily uses ROS and is the only relevant *Spitfire* component that is inside our scope of work. It is worth noting that the game engine is not included in the diagram since the Android application will never interact with it directly (i.e. *Spitfire* will also serve as an "intermediary" between the Android tablet and game engine).

- *ROS* – The third and last component in the system is ROS, which is responsible for simulating drone altitude, drone position, and other game information. In addition, ROS will provide simulated hidden objects and danger zones every time the game engine publishes new and relevant events.

As previously mentioned, the system is based on a hybrid between publish-subscribe and client-server architectural styles. The system uses the publish-subscribe architectural style since the Android application installed in the tablet subscribes to specific Topics provided by

the *Spitfire* server. *Spitfire* then publishes game information and other events to the Android tablet once ROS (or the game engine) finishes simulating that particular Topic. Such message passing between the Android tablet and *Spitfire* is handled by ROSBridge (see section 4 for more information).

The system also uses the client-server architectural style since the Android tablet serves as a client while *Spitfire* serves as a server. This architectural style becomes more apparent when user input is used to manipulate the game. For example, when a user wishes to send a drone to a specific location, the Android application will request such a service to *Spitfire. Spitfire* will then handle the request using ROS and will deliver the updated drone map coordinates by publishing appropriate events. In other words, *Spitfire* is responsible for hosting, delivering, and managing the resources and services to be consumed by the client, which, in this case, is the Android tablet. Figure 1 illustrates how the system uses both architectural styles. It is worth noting that blue lines represent the client-server architectural style, red lines represent the publish-subscribe architectural style, and green lines represent a combination of both architectural styles.

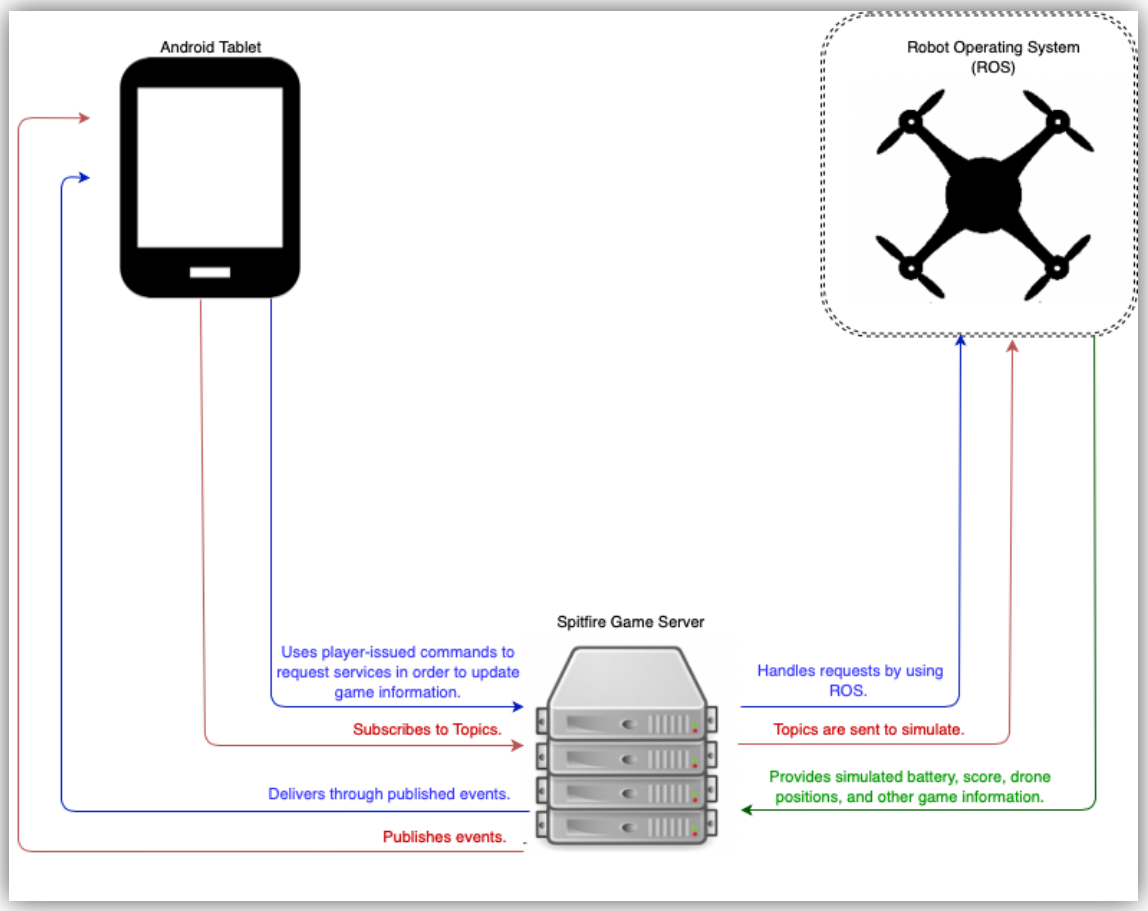

Figure 1 – Diagram illustrating the system architecture. Blue represents the client-server architectural style, red represents the publish-subscribe architectural style, and green represents a combination of both architectural styles.

4. System Design

This section of the document will detail the low-level design of the system, such as its data and architecture component designs, as well as important architecture component interfaces, interaction sequences, object state diagrams, and Topic classes.

4.1 Data Design and Structures

The primary structure that the application will be processing is the messages returned from *Spitfire* when the application subscribes to specific Topics. Depending on the point of execution, these messages will be represented as either String, *PublishEvent*, or JSON objects (see Figure 2). Nevertheless, the message will be parsed only when it is converted to a JSON data structure. All messages returned from *Spitfire* will be handled in the following manner:

- *Spitfire* will return a String representation of the message to ROSBridge
- ROSBridge will redirect this string into Android's *EventBus* library
- Android's *EventBus* library will convert this String into a *PublishEvent* object
- Any class that is registered to the *EventBus* library will be able to convert the *PublishEvent* object to a JSON object

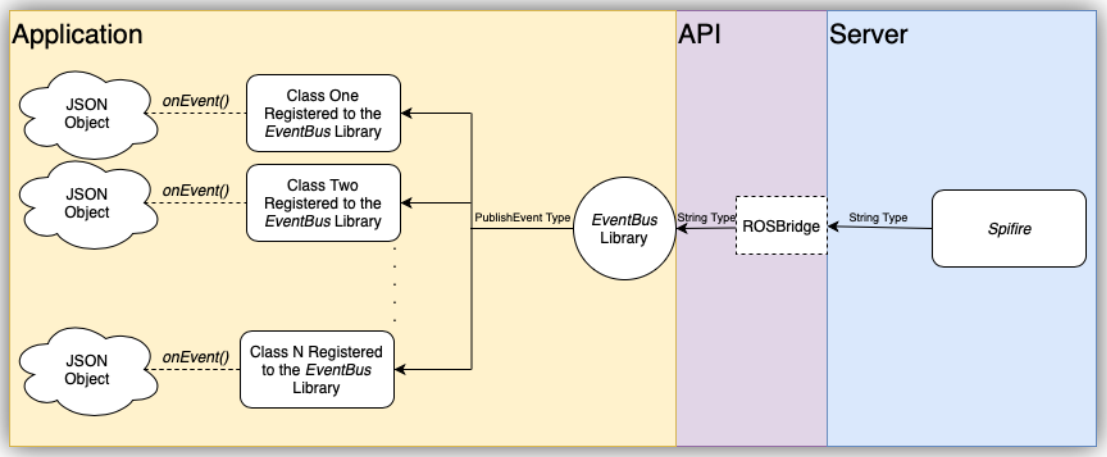

Figure 2 – Diagram illustrating how the messages from *Spitfire* change in type during different times of execution.

It is worth mentioning that any class that is subscribed to the *EventBus* library will call its corresponding *onEvent()* function whenever a new event is published. Here, the class can then create JSON objects, which are the main data structures used in this system, and parse the message accordingly. An example of how the messages are published for a drone position Topic and how they can be parsed using JSON can be found in Figure 3. We note that every different Topic will have a different JSON data structure.

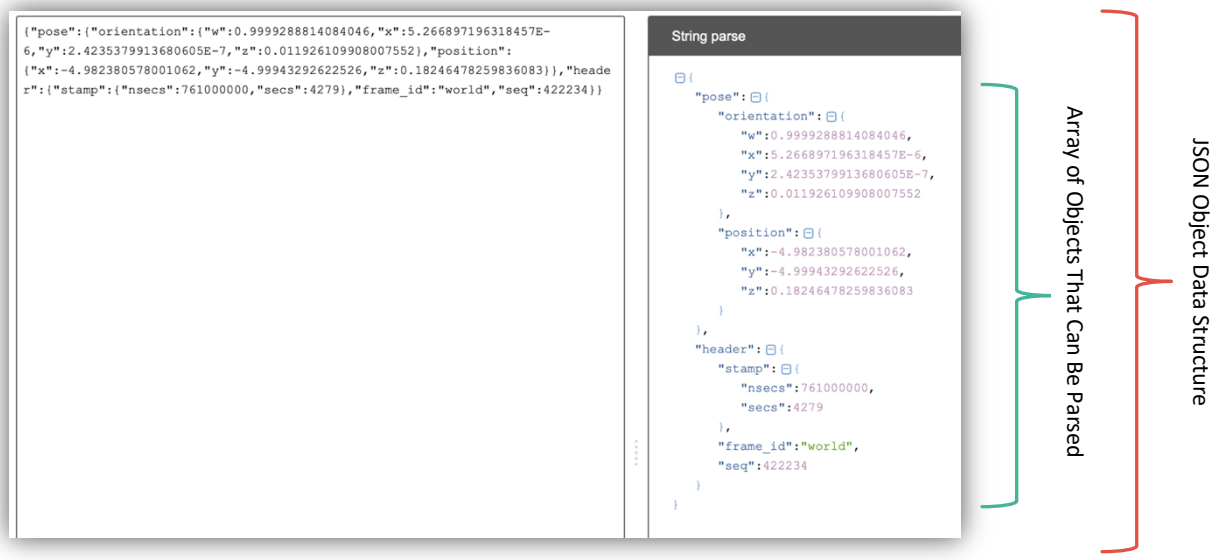

Figure 3 – Example of how published messages can be parsed using JSON. Here, we look at the drone position Topic. JSON objects are the primary data structures used in the system.

Only the message attribute of the *PublishEvent* object will be converted to a JSON object. However, there are other attributes stored in the *PublishEvent* object that are of great interest. These attributes are discussed in more detail in the following table:

Attributes of a *PublishEvent* Object	
Attribute	**Description**
id	Unique ID number that maps to a specific published event.
op	Specific operation that was performed by *Spitfire*. For published events, this will always be "publish."
name	Name of the Topic that was subscribed to that particular event.
msg	The main message that can be parsed by JSON (see Figure 3).

Table 2. A list of the *PublishEvent* attributes that can be extracted from published messages.

The Topics that will subscribe to *Spitfire* will be game/user score, drone locations, hidden human/drone objects, and human/drone danger zones. Lastly, all other data in the system (e.g. map location coordinate chosen by a user) will be String type and will be contained within the application. Figure 5 illustrates how the data inputted by a user (through button presses) will be processed. More information about this interaction can also be found in sections 4.2 and 4.3.

4.2 Architecture Component Interfaces

The system will have various architecture component interfaces that will be interacting with each other. In this section, we describe in detail these interfaces and the interactions between the components. Furthermore, the section also supplements the architecture diagram in section 3 (i.e. Figure 1).

4.2.1 Android Application Interfacing with ROSBridge (Android) API

The Android application will interface with ROSBridge, an API that allows a system to communicate with a ROS-implemented machine. The particular version and type of ROSBridge that is used in this system can be found in the appendix section of the document (section 8). With this API, the application will be able to connect with *Spitfire* (via *connect()*), subscribe to Topics (via *subscribe()*), and use services to manipulate specific abstract objects or attributes stored in *Spitfire,* such as a drone's location. In order to use this particular version of ROSBridge, the following two steps need to be taken:

- The *implementation 'com.github.xbw12138:rosbridge:1.0'* line must be added to the app's build Gradle under dependencies.

- The *maven { url 'https://jitpack.io' }* line must be added to the project's build Gradle under repositories.

4.2.2 Android Application Interfacing with Google Maps API

The Android application will interface with Google Maps' API in order to allow users to choose map coordinates, as well as to display an aerial image of NMSU's campus. The application will request this aerial view from Google servers and will display it with:

- The correct longitude and latitude coordinates of NMSU's campus
- Map bounds
- No map labels

Furthermore, when a user wishes to send a drone to a particular map location, the application will request the Google servers for the appropriate map coordinates that were chosen (i.e. in latitude and longitude format). The application will also need to interface with Google Maps' API when certain objects need to be displayed on top of the map (e.g. drones, danger zones, building overlays, etc.). More information on how to use Google Maps' API can be found in the appendix section of the document.

4.2.3 Android Application Interfacing with Android Tablet (Hardware/OS)

The Android operating system will, essentially, run the application on the chosen physical (or emulated) Android tablet. More specifically, the application will use the touch screen display of the tablet to display the map user interface (see section 5 for more detailed information). The map user interface displays:

- The appropriate buttons, such as those that allow a user to send a drone or to search an area
- Game/user score

- The drone selector
- The High/Low altitude setting
- A legend
- The virtual drones
- Drone altitude
- Battery level
- Hidden objects (when found)
- Danger zones (when found)

Users will interact with the tablet's touch screen to manipulate the application, unless it is run on an Android Emulator. If this is the case, then the user will interact with it using the mouse/mousepad of the computer running the Android Emulator. More information about Android Emulators can be found in the appendix section of the document.

4.2.4 ROSBridge Interfacing with *Spitfire* Server

ROSBridge will interface with NMSU's *Spitfire*, which is the server responsible for handling requests and publishing events. This API allows the Android application to subscribe to specific Topics and request services that are provided by *Spitfire*. ROSBridge is also responsible for handling messages returned from *Spitfire* by publishing *PublishEvent* objects to the *EventBus* library (see sections 4.1 and 4.3 for more detailed information).

4.2.5 ROSBridge Interfacing with *EventBus* Library

Every time that *Spitfire* publishes a message, ROSBridge will also register a corresponding *PublishEvent* object to the *EventBus* (see section 4.1 for more detailed information).

4.2.6 Android Application Interfacing with *EventBus* Library

The android application will keep track of which class is subscribed to the *EventBus* and will call each of the classes' *onEvent()* function every time *Spitfire* publishes a message. It is important to note that in order to register a class to the *EventBus,* the following three steps need to be taken:

- Import the *de.greenrobot.event.EventBus* library into the chosen class
- Register the class using the following command: *EventBus.getDefault().register(this);*
- Implement the *onEvent()* function and handle/parse incoming messages appropriately (see section 4.1 for more detailed information)

The class in question will be able to handle all of the messages being published by *Spitfire* only when these three criteria are met.

4.3 Architecture Component Designs

Ideally, each architectural component would have its own internal design. However, designing or modifying ROSBridge, Google Maps' API, the game engine, and *Spitfire* is outside our scope of work. Because of this, we can only provide a low-level description of the Android application that implements the user interface, as well as how it communicates with *Spitfire*. The Android application's main components are illustrated in Figure 4 and are discussed in more detail below.

4.3.1 MapActivity

This component will communicate with Google Maps' API to get an aerial view of NMSU's campus, as well as user selected map coordinates. The component is also responsible for displaying the correct game information onto the tablet's touch screen, such as drones, hidden objects, danger zones, and game score. That is, *MapActivity* will be the only component that will be responsible for implementing the user interface. The component will accomplish this by registering itself to the *EventBus* library and implementing the *onEvent()* function in order to parse all published messages from *Spitfire. MapActivity* will also pass important user actions to the *Drone* component and will use the *GameScore* and *ServerCommunication* components to accomplish specific tasks. For example, if a user wants to send a drone to a particular map location, the component must request this service from the *ServerCommunication* component, which will in turn request this service from *Spitfire* using ROSBridge. *MapActivity* will also connect to *Spitfire* by creating an instance of the *ServerCommunication* class and will use this instance to subscribe the system to all the required Topics.

4.3.2 GameScore

This component will communicate with *MapActivity* since it will provide the current game/user score to it. The component will also register itself to the *EventBus* library and implement the *onEvent()* function in order to parse all published messages from *Spitfire.*

4.3.3 Drone

There will be a total of four instances of this component declared in the *MapActivity* component since there are a total of four virtual drones. Each individual *Drone* component will provide the x (longitude), y (latitude), and z (altitude) position coordinates to *MapActivity* in order for it to display each individual virtual drone correctly on the user interface.

4.3.4 ServerCommunication

This component will have the following three important responsibilities:

1. Connecting with *Spitfire* by creating a new *ROSBridgeClient* instance and calling the *.connect()* function.

2. Implementing several functions that will subscribe to specific Topics when called by *MapActivity*. In other words, *ServerCommunication* will be responsible for creating all relevant Topics and subscribing to them once *MapActivity* calls specific functions (see Table 3 to see all relevant Topics that will be created and subscribed to).

3. Requesting services from *Spitfire.* We note that requesting services from *Spitfire* can be done easily by simply sending a String in the correct JSON message format to the server.

It is important to make the distinction that this component will not register itself to the *EventBus* since messages will be parsed only by the *MapActivity, Drone,* and *GameScore* components.

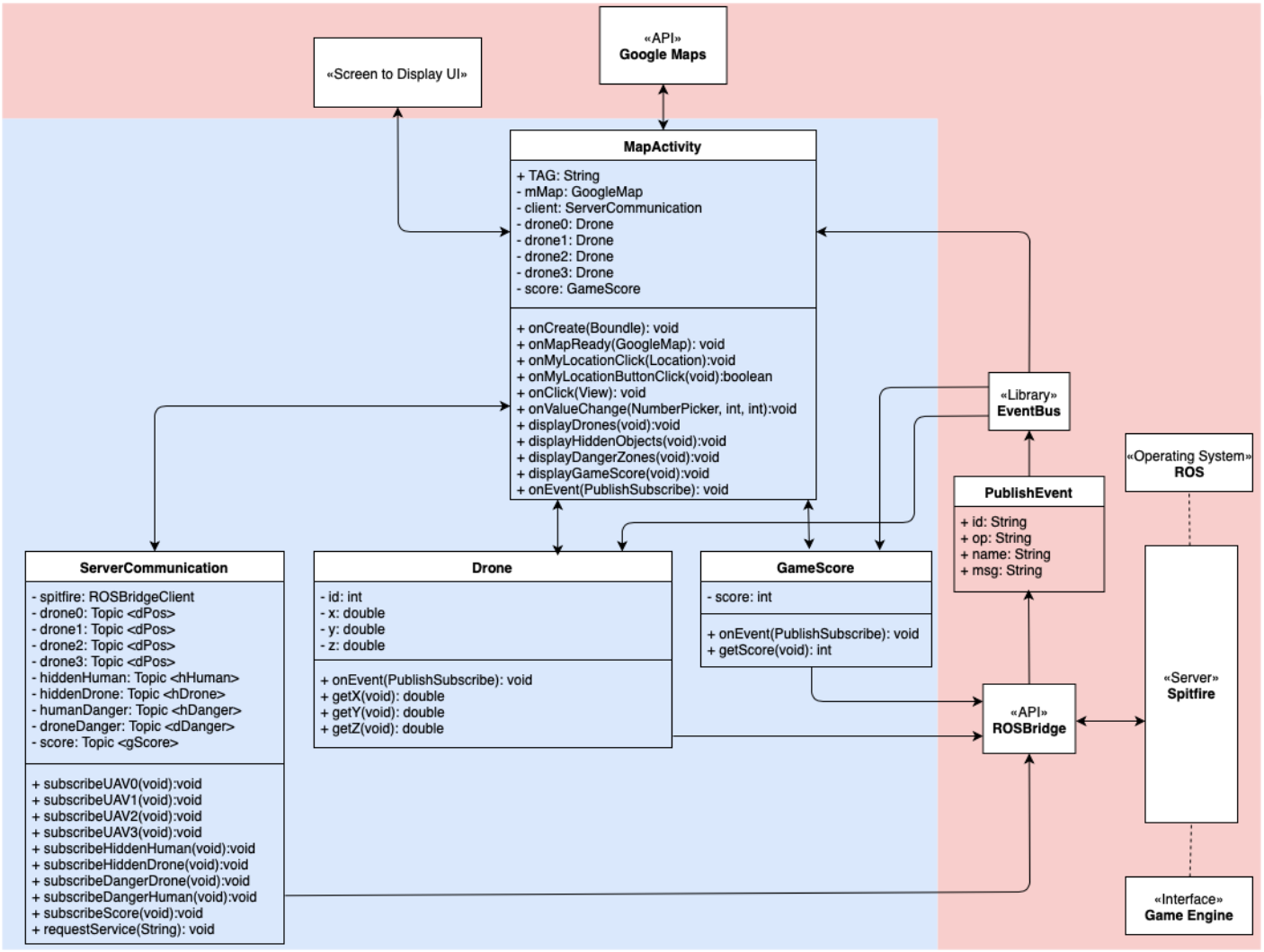

Figure 4 – UML class diagram of the Android application and the interfaces it interacts with. We note that everything that lies outside the blue area is outside the scope of work for this project.

Another important aspect that is worth discussing is all of the relevant topics and topic types that are supported by *Spitfire.* A list of them can be found in the following table:

Topic Name	Topic Type	Short Description	Topic Class
/uav0/ground_truth_to_tf/pose	geometry_msgs/PoseStamped	Provides drone0's simulated map location	dPos
/uav1/ground_truth_to_tf/pose	geometry_msgs/PoseStamped	Provides drone1's simulated map location	dPos
/uav2/ground_truth_to_tf/pose	geometry_msgs/PoseStamped	Provides drone2's simulated map location	dPos
/uav3/ground_truth_to_tf/pose	geometry_msgs/PoseStamped	Provides drone3's simulated map location	dPos
/w_player_score2	std_msgs/String	Provides the game/user score	gScore
/w_hccoordinates	std_msgs/String	Provides hidden human objects	hHuman
/w_dccoordinates	std_msgs/String	Provides hidden drone objects	hDrone
/w_hdzcoordinates	std_msgs/String	Provides human danger zones	hDanger
/w_ddzcoordinates	std_msgs/String	Provides drone danger zones	dDanger

Table 3. A list of the most important Topic names and Topic types provided by *Spitfire.*

4.4 Important Interaction Sequences

This section will describe and illustrate especially important object interaction sequences that occur to implement some particular behavior using UML message sequence charts. For this system, the most important interaction sequence is users selecting which drone should be manipulated and what manipulation should take place. Figure 5 depicts this important interaction between the user and system components. As the figure illustrates, the *MapActivity* component will act as an intermediary between the user and other components. That is, the *MapActivity* component will be responsible for getting user input, handling this input by calling other components, and displaying appropriate updates to the user through the user interface. It is important to note that even though ROS is part of *Spitfire's* implementation, it was separated in order to make a clear distinction of which component is communicating results back and which one is doing the actual required simulations.

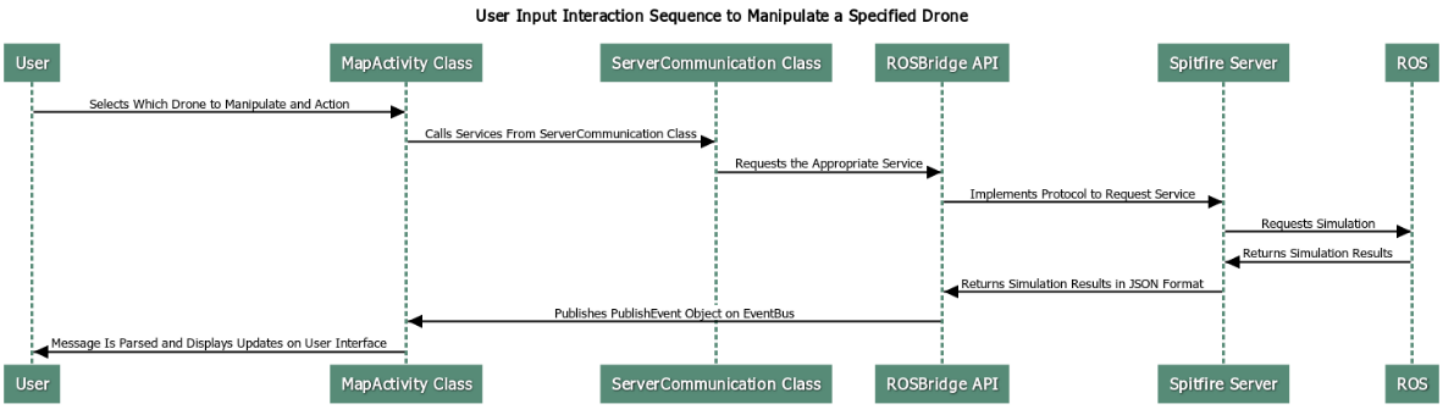

Figure 5 – Diagram illustrating the UML message sequence chart for user input when manipulating a specific drone.

4.5 Important Object State Diagrams

This section will document objects in the system that have regular and important state progressions using UML state diagrams. It is important to note that this particular system only has one object with regular state progressions: the drone selector and its accompanying drone actions. That is, selecting which drone to manipulate and how to manipulate it on the user interface has a series of discrete states. Figure 6 illustrates this point and emphasizes that the drone selector object can only be in one of the following four states: 1, 2, 3, or 4. That is, 1 maps to drone0, 2 maps to drone1, and so forth. Furthermore, a manipulation of a selected drone can only be in one of the following four states: send, land, search, or altitude change.

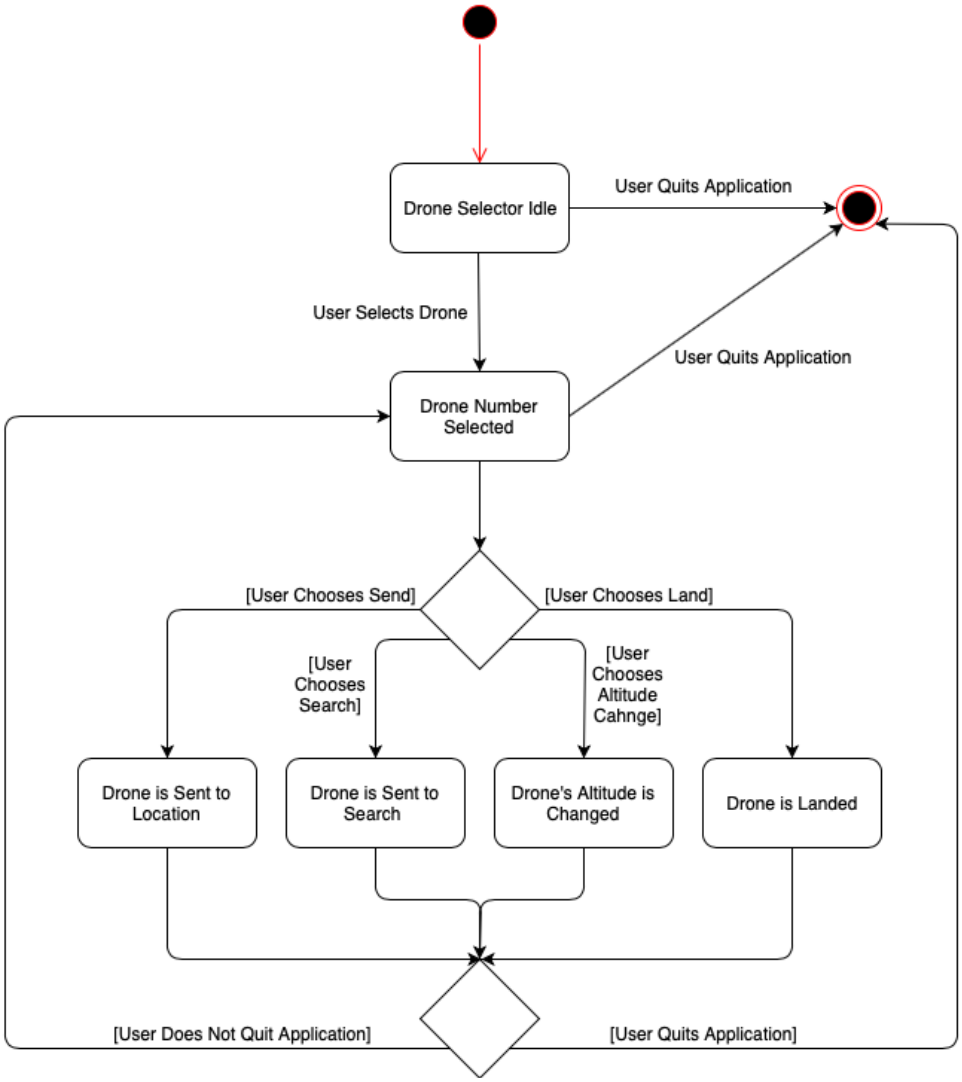

Figure 6 – UML state diagram illustrating the states of the drone selector and drone actions.

4.6 Algorithms

Figures 2, 4, and 5 provide an overview of how the system was implemented and how the system components work in conjunction with one another. However, there were no specific "algorithms" used to implement the system.

4.7 Design Patterns

A publish-subscribe design pattern was used since the system uses a publish-subscribe architectural style. More information can be found in sections 4.1, 4.2, and 4.3.

4.8 Distinction Between Registering and Publishing Events

An important distinction must be made between registering and publishing events. That is, registering events to the *EventBus* can only be done by ROSBridge while publishing events can only be done by *Spitfire.* Figure 2 visually represents this concept.

5. User Interface Design

This section of the document will explain in detail the user interface design. The user interface for the Android application is designed with XML. During the design phase of the user interface, an emphasis was made to make it as simple and clean as possible, while still maintaining all the required functionality. The user interface encompasses the entire screen of the physical (or emulated) Android tablet and displays an aerial view of NMSU's map (see Figure 7). The following list of buttons and features are placed on top of the map:

- Search **(1)** – Sends the chosen drone to search a location within the map.
- Send **(2)** – Sends the chosen drone to a location within the map.
- Land **(3)** – Lands the chosen drone.
- Ascend / Higher Altitude **(4)** – Increases the altitude of the chosen drone.
- Descend / Lower Altitude **(5)** – Decreases the altitude of the chosen drone.
- Legend **(6)** – Legend of how hidden objects and danger zones will be displayed.
- Game Score **(7)** – Current game/user score.
- Battery Life **(8)** – Battery life of chosen drone.
- Current Drone Image / Altitude **(9)** – Image used to represent the chosen drone. Current altitude of chosen drone.
- User Location **(10)** – Displays the current location of the user/tablet on the map.

It is important to note that the Drone Number Selector **(11)** was implemented using a "carousel method" instead of a button since this will make it easier for users to choose which drone to manipulate. Furthermore, all buttons and the Drone Number Selector are placed near the corners of the user interface and are designed to be large and bold. Such design decisions were made in order to comply with Fitts' law, which suggests that buttons are better if they are bigger and placed next to the edges of the screen since they are easier and quicker to reach by users (Preece, Rogers, & Sharp, 2015). Android Studio's Java adds functionality to the buttons and interprets user input. Furthermore, each button has an action/*onClick/onValueChange* listener.

The user interface displays building outlines, four different colored drones, danger zones, and hidden objects. The game/user score and battery life are also displayed. We observe that the battery level changes depending on which drone is selected. All relevant information is also displayed near the edges of the screen in order to comply with Fitts' Law and to make the user interface less cluttered. It is important to mention that the user interface will allow users to manipulate only one drone at a time since it makes the system not only perform better, but also simpler and easier to use. Lastly, the user interface will display different colored "imaginary square" areas whenever a drone is chosen to search.

Figure 7 – Screenshot of the user interface currently being implemented by the *MapActivity* component. *We note that the numbers are placed only in this document and not in the final implementation of the user interface.*

6. Deployment Design

This section of the document will outline how the application will be delivered and maintained, as well as any external components that are needed in order to run it.

6.1 Delivery

The *ARV Application* will be delivered through a private GitHub repository. This repository will be managed by Ahmed Khalaf (khalaf@nmsu.edu) and/or Dr. Toups (z@cs.nmsu.edu), the two project sponsors. As such, access to the source code and other relevant documentation must be requested to either Ahmed or Dr. Toups. Because the application will be delivered only to a few people for evaluation and research, large scale deployment will not be considered and is outside the scope of this project.

A user with granted access to the source code can download the required files from the private repository and use Android Studio to run the application. Android Studio can install and run the application on either an Android Emulator or a physical Android tablet

(see section 8 for more resources on how to accomplish this). It is worth mentioning that the Android Emulator simulates an Android device on the computer screen. This means that the functionalities of the application will stay the same, but how the user interacts with the interface will change (e.g. using a touchpad instead of a physical touch screen to navigate through the user interface).

6.2 Maintenance

The application will be maintained by Ahmed and Dr. Toups, as well as other individuals authorized to modify the project, after December 14, 2019. Maintenance will consist of making sure that the application runs and meets the performance requirements described in the *Requirements Specification* document. Furthermore, the application must be constantly tested on several different types of tablets and API levels. If new technologies (such as more advanced physical drones) or ROSBridge APIs (such as a more efficient ROSBridge API for Android) emerge, then the responsible individuals for maintaining the application must make appropriate modifications. All new changes and modifications must be pushed to the private repository with the permission of Ahmed or Dr. Toups.

6.3 Requirements to Run Application

In order to install and run the application, it is assumed that the potential user will have the following external/internal components and requirements:

- Internet access in order to connect with *Spitfire*
- A physical Android tablet or an Android Emulator
- Android Studio
- A desktop or laptop computer in order to run Android Studio
- Access to the private GitHub repository
- Connection to NMSU's secure network (see section 8 for more resources)

7. Coding Standards

The project will abide to the coding standards that are discussed in more detailed in the following subsections.

7.1 Naming Conventions

All functions and variables should start with a lowercase letter, followed by camel-case (e.g. *temporaryVariable*). Class names should start with an uppercase letter, followed by camel-case (e.g. *DronePosition*). All constants should be uppercase. Constants with multiple words should use underscores (e.g. ROS_CLIENT). All variable, function, class, and constant names should be meaningful. That is, no generic variables such as *x* or *y* shall be used. Variables that are too long or too short will be avoided unless the context in which they are being used recommends it (e.g. for loops using an *i* variable to iterate is permitted).

7.2 Spacing

Indentation must be done by either three spaces or one tab. That is, indentation might be different from class to class, but each class shall have a consistent indentation of either three spaces or one tab for each line of code. A space shall also be used before and after operators (+, =, *, etc.) in order to make the code less cluttered. Excessive spacing will be avoided at all times since it makes code harder to understand and trace.

7.3 General Comments

Comments should be frequently used throughout the program to accurately describe segments of code that are complex or not intuitive. However, too many comments should be avoided since they make the code difficult to read. A good rule of thumb to follow is that for every 5 to 8 lines of code, there should be between 1 to 2 lines of comments; no more, no less. Programmers should use "//" for short comments while "/* */" for more long and intricate comments.

7.4 Error Handling Messages

All programmer-issued error messages should be meaningful and have enough detail to give debuggers clues as to what is causing the error. It is important to note that modifying system errors that are provided by Android Studio is outside the scope of work.

7.5 Program Statements

There will be only one statement per line. The following example illustrates what is not acceptable:

```
int drone0Position = 0; int drone1Position = 2;
```

Instead, this line should be broken down into two one per line statements, such as the following:

```
int drone0Position = 0;

int drone1Position = 2;
```

7.6 Control Statements

Control statements (e.g. *if*, *then*, and *else*) with one inside statement shall have curly brackets, even though they are not needed. The following example illustrates this point:

```
if ( drone0Position == 0 ) {
        drone0Position++;
}
```

7.7 Nested Loops

If nested while or for loops are used, then a meaningful comment next to each ending curly brackets is needed. The following example illustrates this point:

```
while ( drone0Position > 10 ) {

        squareMiles = 10;

        while ( squareMiles >= drone0Position ) {

        .....

        } /* end inner while loop */

        squareMiles = 10;

        drone0Position--;

} /* end outer while loop */
```

7.8 Header Information

Every file shall have the following list of information in its header as comments:

- Team name
- Last day modified
- Project name
- Authors

7.9 Line Length and Wrapping

Each individual line of code shall not exceed more than 70 characters. Furthermore, conditions should not be wrapped into multiple lines since it makes it harder to read the code.

7.10 Function Comments

Every complex function that is declared within a class shall have the following list of information before it is declared:

- Short description of what the function does
- Names of authors that wrote or helped contribute to the function
- Pre-conditions and post-conditions (if applicable)
- Information about exceptions that might be triggered by the function

Simpler and less complex functions that are self-explanatory (e.g. *print()* or *get()* functions) do not require these function comments. However, all functions must end with a meaningful comment next to its end curly bracket.

8. Appendices

This section will detail information that was not appropriate to put in-line in any other section.

- *Android Studio* – The following link can be used to download Android Studio: https://developer.android.com/studio

- *Android Studio and Android Emulator*: The following link provides useful information and resources on how to run an application using Android Studio and an Android Emulator: https://developer.android.com/studio/run/emulator

- *Android Studio and Physical Android Device*: The following link provides useful information and resources on how to run an application using Android Studio and a physical Android device, such a tablet: https://guides.codepath.com/android/Running-Apps-on-Your-Device

- *GitHub Link* (NOTE: Only those who are granted access can view the source code.) – The following GitHub link can be used to access the project's source code: https://github.com/PIxLLab/MapInterface-AndroidApp

- *Google Maps' API for Android* – The following link can be used to access documentation relating to Google Maps' API for Android: https://developers.google.com/maps/documentation/android-sdk/intro

- *Requirements Specification* – The following link can be used to access the project's *Requirements Specification* document*:* https://eltnmsu-my.sharepoint.com/:b:/g/personal/jose5913_nmsu_edu/EQbkIc6P1K9CsQx2hrEtrv8BMS_H2WcuIi5bZ-Tplt_r4w?e=Zq1vTi

- *ROSBridge Library* – The following GitHub link can be used to access the ROSBridge API that was used in this system: https://github.com/xbw12138/rosbridge

- *Statement of Work* – The following link can be used to access the project's *Statement of Work* document: https://eltnmsu-my.sharepoint.com/:b:/g/personal/jose5913_nmsu_edu/EZ-cpMLMQypNppmWFABqVagBhKiNphEzzEbQVM9Ruo0Vcg?e=W4KQwa

- *User Interface Design Practices* – Practices on how to design effective user and game interfaces will be taken from the following book: Jenny Preece, Yvonne Rogers,

Helen Sharp. *Interaction Design: Beyond Human-Computer Interaction*, 4th Edition. Wiley. 2015.

- *VPN Connection* – A successful connection between the application and *Spitfire* requires a NMSU secure internet connection. Users can accomplish this by either connecting on-site to a secure internet connection or using a VPN client to connect while off-site. The following link can be used to access documentation on how to install an appropriate VPN: https://kb.nmsu.edu/page.php?id=70045