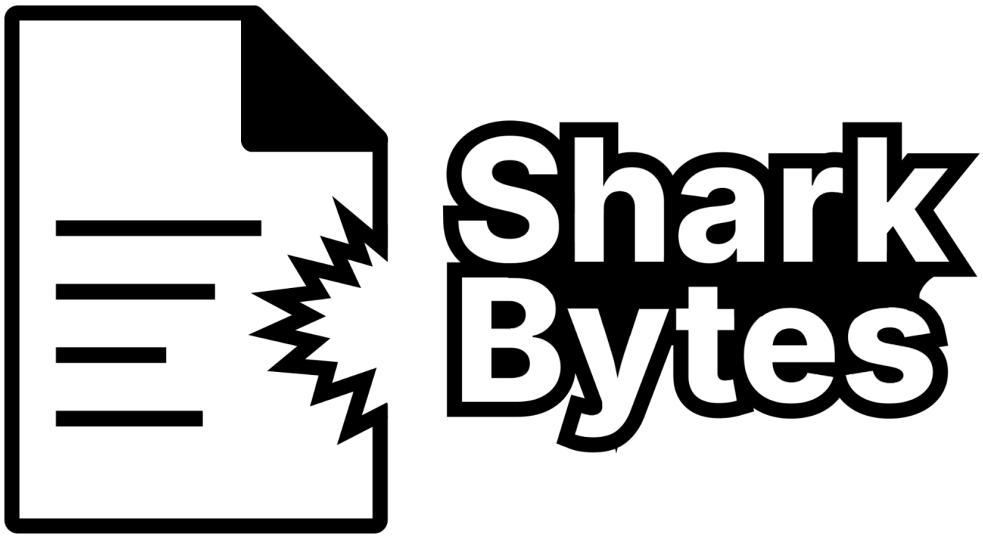
Software Design Document Ver. 2 - 02/27/2026




Clients: David Rogowski, Mara Dzul, Pilar Rinker

Mentor: Scott LaRocca

Team: Jered Angous, Daniel Arden, Alexander King, Anthony Narvaez

The purpose of this document is to discuss how our team has decided to architect our application.

# Table of Contents

# Introduction

Over 40 million people depend on the Colorado River and its surrounding ecosystems for food, water, and livelihoods, underscoring the importance of this vital resource. Conservation efforts are essential to monitoring and preserving the well-being of these resources amid growing pressures from climate change, prolonged drought, and overuse. Now, more than ever, the data collected by organizations such as the U.S. Geological Survey, Arizona Game and Fish Department, Grand Canyon Monitoring and Research Center, and U.S. Fish and Wildlife Service are critical in identifying environmental stressors, guiding resource allocation, and informing effective conservation strategies.

However, the current data collection workflow used by these organizations relies on an outdated and poorly maintained application known as SHOALS. While still essential to their operations, the use of SHOALS requires dealing with an unintuitive user interface, a total lack of documentation, and minimal maintenance. These issues have led to frustrations in the field, increased time spent troubleshooting, and a higher risk of data entry errors, all of which undermine the efficiency and effectiveness of conservation efforts.

Our team aims to address the issues with SHOALS by creating a more flexible and modern data entry system. This new application will provide a maintainable and well-documented replacement to the aging SHOALS program and allow biologists or researchers to continue providing critical monitoring data efficiently and effectively. This document intends to describe the architectural design patterns we plan to develop our new application around, as well as define the design and functionality of the major modules that comprise our solution.

Key user-level requirements of our application are:

- A customizable set of fields so users can adapt it to their use case.
- A robust backup system that saves to local storage and multiple USB drives upon every saved entry.
- Bluetooth compatibility with PIT Tag Readers.
- Easily interpretable graphical data summaries of collected data.

Functional requirements include:

- Interactive user interfaces for entering data
- Exporting data to a flat file format.
- Store entered information into a SQLite database,
- Logic for pairing to and reading data from PIT readers
- Automatic backups to multiple external storage devices every time an entry is made.

# Implementation Overview

Our project is meant to replace a piece of software named SHOALS. As such, we are designing our application to be a stand-in replacement for SHOALS, by matching all of SHOALS' key functionality. SHOALS is a Windows-based data entry application tailored for aquatic researchers who collect, scan, and process fish.

SHOALS' primary feature is that it supports Bluetooth PIT scanners, drastically speeding up data processing. It was heavily emphasized that our application needs to have the same functionality. It also supports being able to customize fields via 'control files', which denote which fields need to exist, their names, and properties such as whether they cannot be empty or not. It is key that our application supports a similar level of customization, as it enables the software to be used for multiple field study designs.

Additionally, while SHOALS only supports Windows, our clients have agreed that supporting multiple platforms would be valuable. Therefore, our team has decided to build our application using a multi-platform application framework. Our team landed on Flutter due to its higher performance than competing frameworks like Electron, support for most major operating systems, and wide community support.

# Architectural Overview

The architecture of the project is critical to provide our clients with a maintainable and scalable solution. This is why our team has decided to go with a combination of classical Clean Architecture with the presentation layer structured following the View-ViewModel pattern. The key goals behind this architectural choice were maintainability, ease of concurrent development, and scalability. These two design patterns will ensure the app continues to evolve with the needs of our clients far beyond our scheduled project completion time. We now take a domain-centric approach to explaining the components that make up the project.

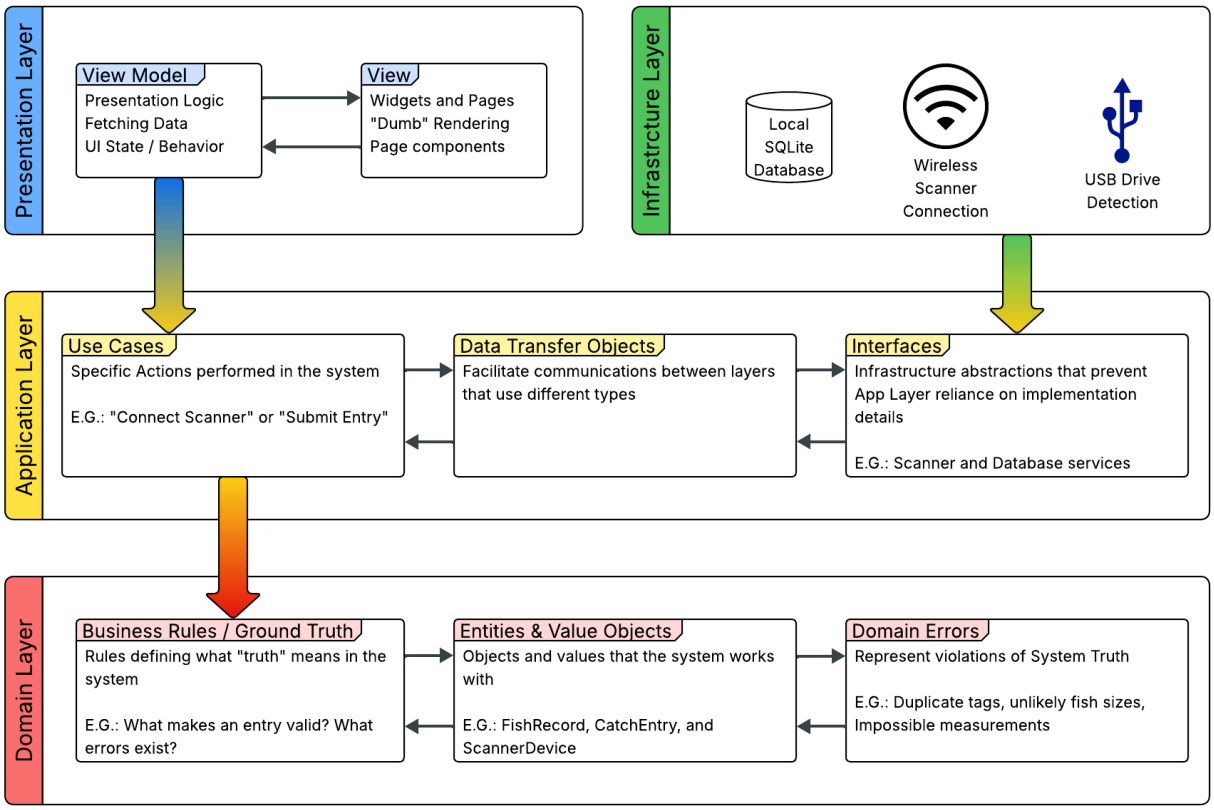



*Figure 1: High-Level Architectural Organization*

Starting with the Domain, we have “ground truth” for our application. All of the business rules, logic, and objects that the system directly works with are modelled here. Entities and Value objects are classes that represent real concepts like fish records, catch entries, and scan events. These follow business rules that ensure the system never enters a state in which truth is violated. In such a scenario, Domain-level errors are generated and handled throughout the app. An example of this would be entering a Fish Record with attributes that cannot physically exist, or identifying a duplicate record. The domain has no dependencies and is not concerned with the implementation of any other parts of the application. Its purpose is to ensure core business logic stays the same even if other parts of the application change.

Moving up a layer, the application layer enforces the rules found in the domain. Here, we define specific workflows, or use cases, needed by the application. These handle the way data moves through the application and how the application interacts with external modules. Importantly, code in this layer cannot depend on upper layers like the Presentation layer and the Infrastructure layer. In order to access dependencies in the Infrastructure layer, interfaces are used. Interfaces provide a way for the application layer to communicate with external resources like the main database, external tools like PIT scanners, and other hardware like backup USB drives. These interfaces are implemented as abstract classes within the application layers that expose methods for data retrieval. The concrete implementation of these classes is found within the infrastructure layer and is what directly interacts with the aforementioned external dependencies.

This is a powerful design choice as it allows for easy swapping of external technologies in the future. The database could easily be switched to something other than SQLite, different types of scanners could be added, and support for other platforms could be implemented simply by changing code in the infrastructure layer alone. The core interaction between these external devices and the application is represented by the interfaces and usually remains the same. Even if these

interactions did need to change, the code for them is conveniently found in the interfaces directory of the application layer.

Finally, we have the presentation layer, which handles all of the rendering and UI that the users interact with. Due to the presence of UI state and the need of the application to access lower layers, it is helpful for this layer to be organized in its own specific way. This is what the ViewModel architecture aims to help with. Using this organization, the presentation is split into two parts, the View and the ViewModel. The View focuses on all of the static components of the UI. It contains pages, widgets, and other navigational components that don't change and are not tightly bound to state. However, in order to detect when a user pressed a button or when the user types into a field, we need the ViewModel. This component handles the state of the UI and controls how the UI reacts to specific changes. This component is also responsible for storing the text written by the users and processing it by calling on use cases found in the application layer.

Clean Architecture, combined with VVM, was evaluated by the team to provide the largest benefit as development continues. With it our team will be able to work simultaneously with little to no conflicts while providing a future-proof and maintainable product to our clients. We are excited to continue developing the application according to this structure.

# Component-Level Design

Having considered the high level organization of our project, we now take a closer look at the major components within the application and how their internal structure is organized. It is important to note that, for clarity, the domain layer was omitted from the diagrams as the vast majority of the functionality of the major components in the application can be summarized via the upper layers. It is also worth mentioning that most of these components span most, if not all, of the layers in the architecture. We begin with the scanner connection system.

## Scanner Connection System

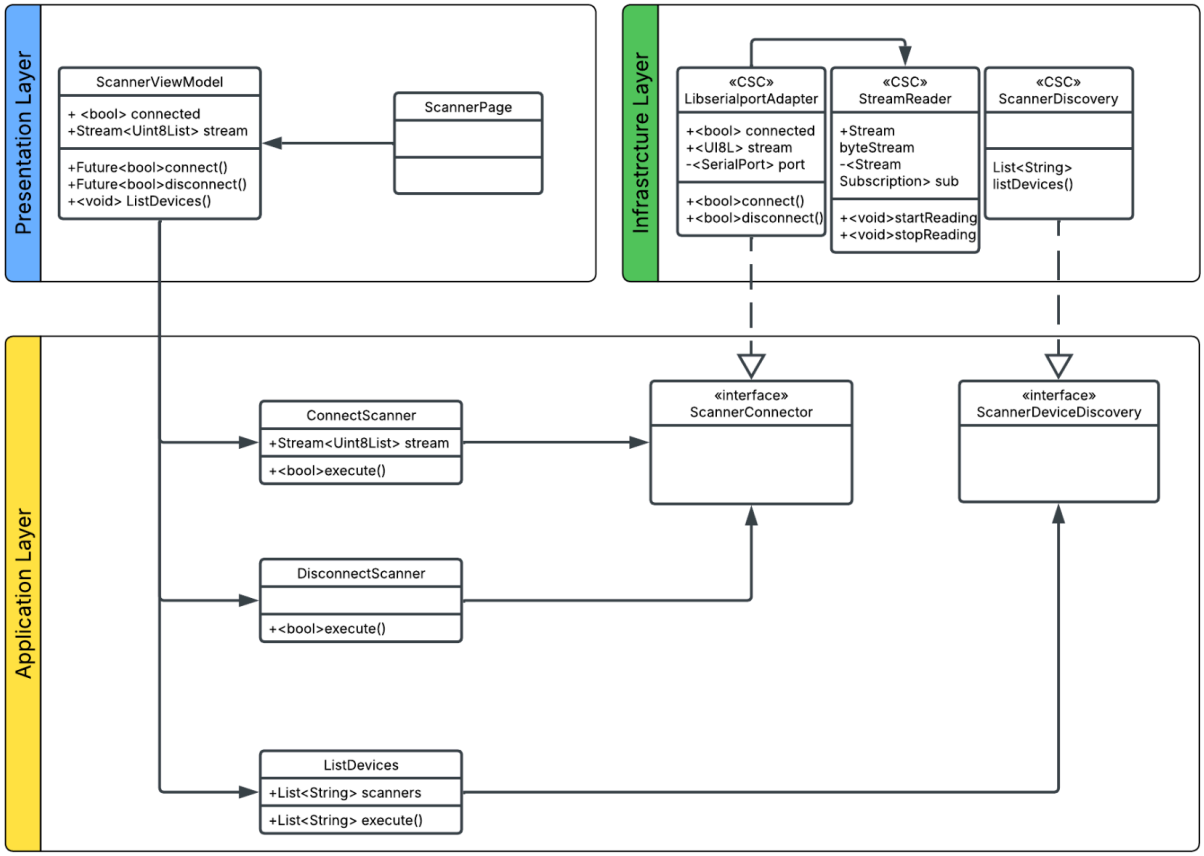



*Figure 2: Scanner Connection Component Diagram*

The scanner connection module is what allows for wireless connection to Biomark PIT scanners via Serial Port Profiles (SPP). The module is responsible for establishing a connection, returning an event stream used to read data, and tearing down the connection once the app is closed or when the user decides to do so. This component is accessible via the ScannerViewModel class in the presentation layer and provides access to connect, disconnect, and list functionality. The latter of these allows for the detection of possible scanner devices already connected to the computer. These use cases depend on interfaces to the infrastructure layer for dependencies such as

the connector and discovery implementations. This structure, which is followed by most of the components in the application, neatly separates the concerns of each class and provides modularity.

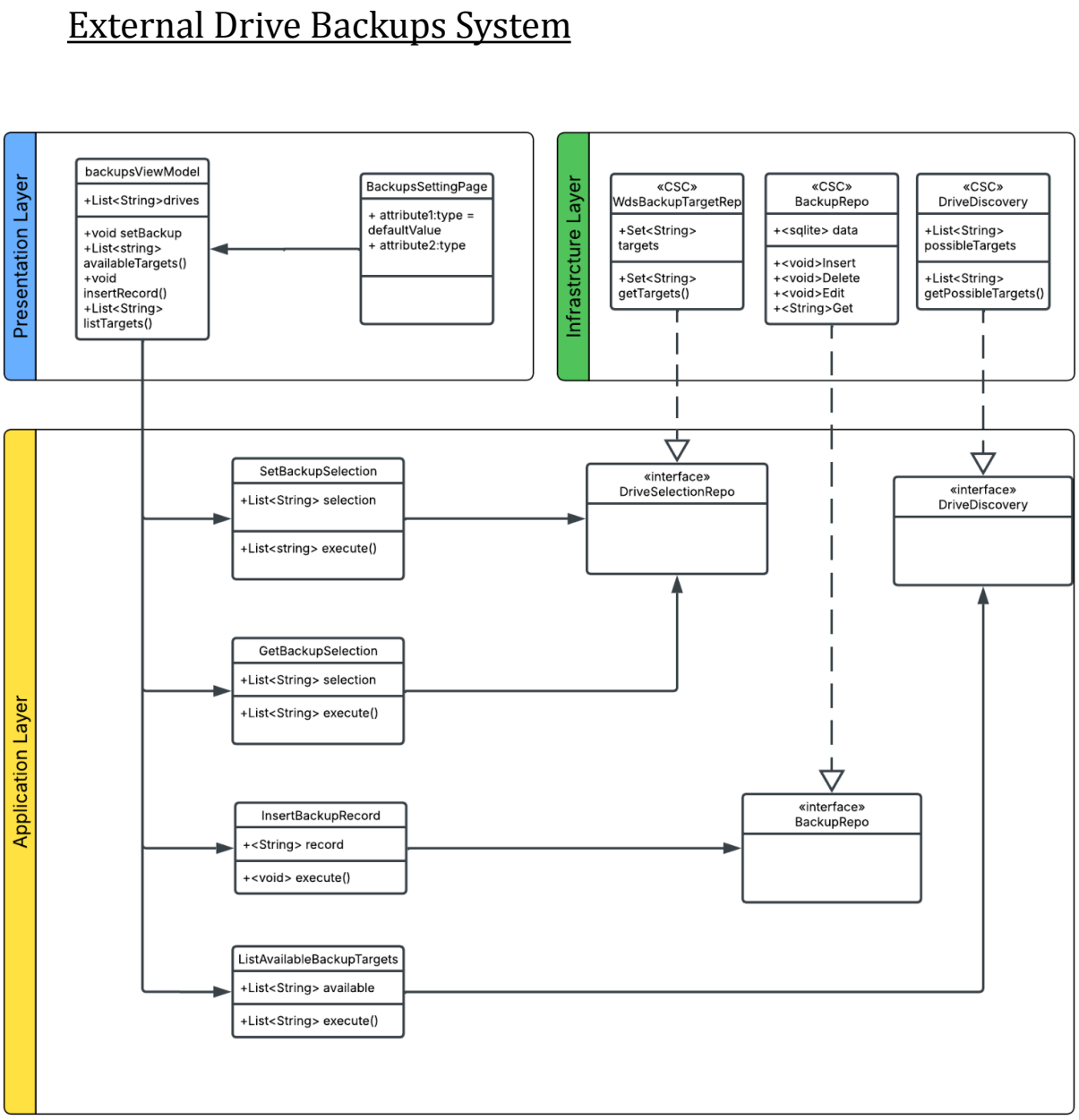



*Figure 3: Component Diagram for the External Backup System*

This system tackles the tasks of identifying connected external drives, displaying them, allowing for a selection of these drives to be made, and routing backups to those specific drives. This is crucial to maintain the integrity of the data throughout the trip. The module begins in the presentation layer, providing a UI for drive selection and using the BackupViewModel class for access to app-layer use cases. These use cases include actions such as selecting specific backup drives, getting a list of selected drives, inserting a backup record, and listing available backup target drives. These use cases depend on interfaces not only for the data that is to be backed up, but also for identifying what drives are present in the first place, and keeping track of what drives have been selected by the user. These interfaces are implemented in the infrastructure layer using SQLite tools to handle the data, and other OS-dependent libraries to detect the drives.

# Data Visualization

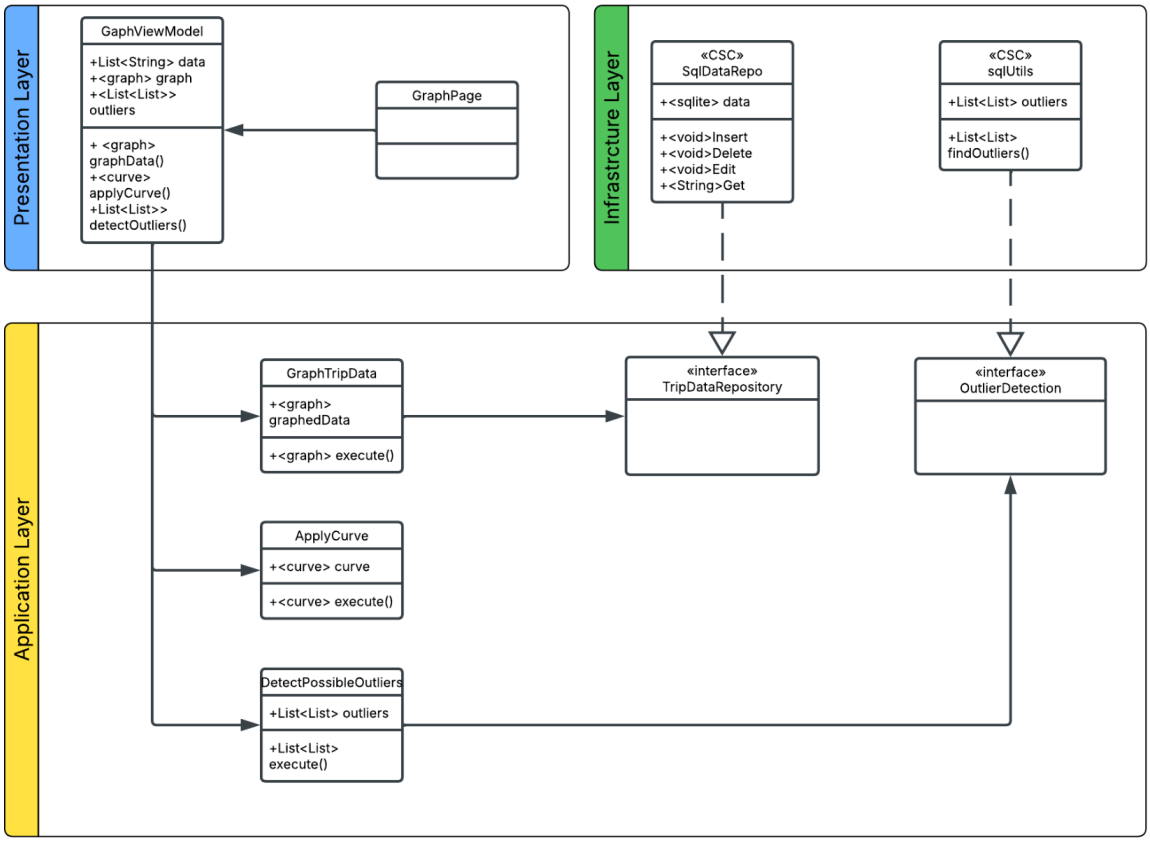



*Figure 4: Component Diagram for the Data Visualization System*

The data visualization system is a feature designed to take the data collected during the trip and generate a graph, properly apply any desired curve fit, and detect any possible outliers. This component lives primarily on the presentation layer since a lot of the graphing components are very tightly related to flutter's UI tools. The component still communicates with the App layer to get access to use cases like graphing the data, applying curves, and detecting possible outliers. The two interfaces used by this module involve the database for access to the data that is to be graphed, and the sqlite tools needed for outlier detection. These aspects are organized according to Clean

Architecture and can thus swap out the sqlite tools for any other external tool to identify outliers or to add functionality.

## Customizable Fields

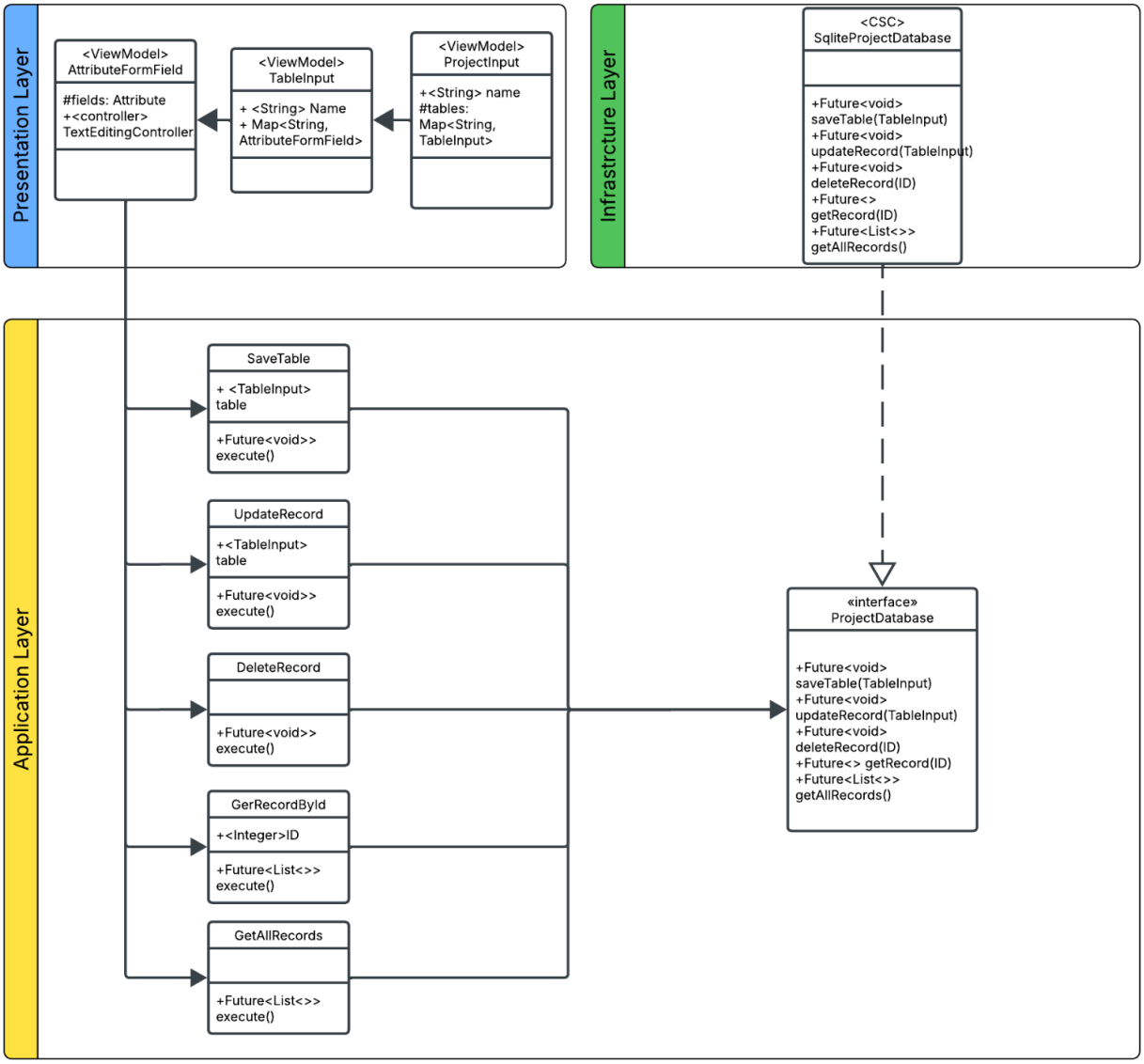



*Figure 5: Component Diagram for the Dynamic Field Generation System*

The customizable fields system, known internally as the FlexField system is a crucial module that allows the user to define the specific fields they intend to fill out upon the capture of a fish. This allows different agencies to use the same app to collect different types of data. This system interacts with the main database of the app via use cases and interfaces in the application layer. The project sql table is identified, the columns are extracted, and the fields are generated based on these. The data access part of that process happens in the infrastructure, but the actual logic for field generation takes place in a use case at the Application layer. The front end then uses these fields to provide an interface to the user in which they can perform standard CRUD operations on the trip data.

# Data Exporting System

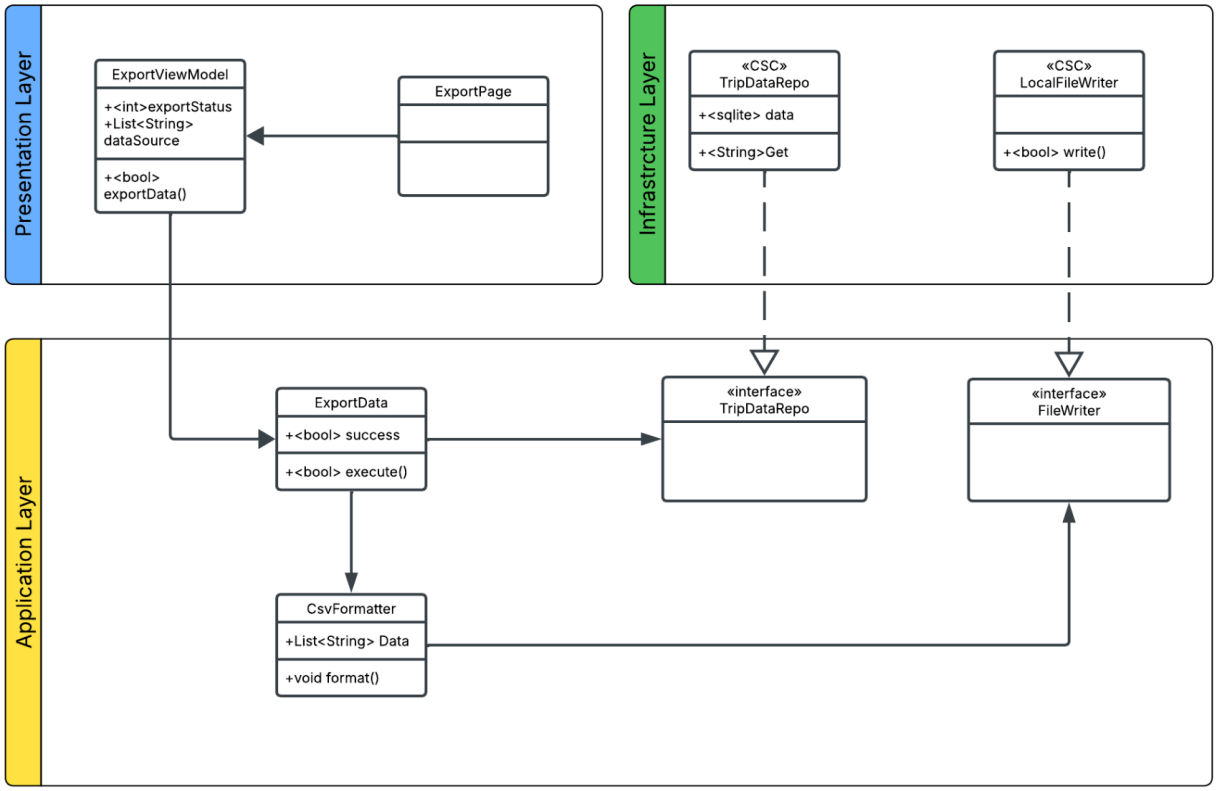



*Figure 6: Component Diagram for the Data Exporting System*

The data exporting system is a relatively small component that is in charge of converting the SQLite-formatted data into a CSV file to allow for easier integration into the main database. This format was what the old Shoals application used, and so it is important to maintain this functionality. Again, the component starts in the presentation layer, as it gives a UI for the user to export the data to a specific location. The ExportViewModel is the class responsible for connecting the UI to the App-layer functionality, which in this case, is just the ExportData use case. This use case depends on a CSV formatter service that takes the data from the interface, that is from the app's main database and translates it into CSV. This CSV is then written to a file using the FileWriter

interface. This interface is necessary because different operating systems have different ways of formatting and writing files, so it counts as an external dependency.

## External Drive Synchronization System

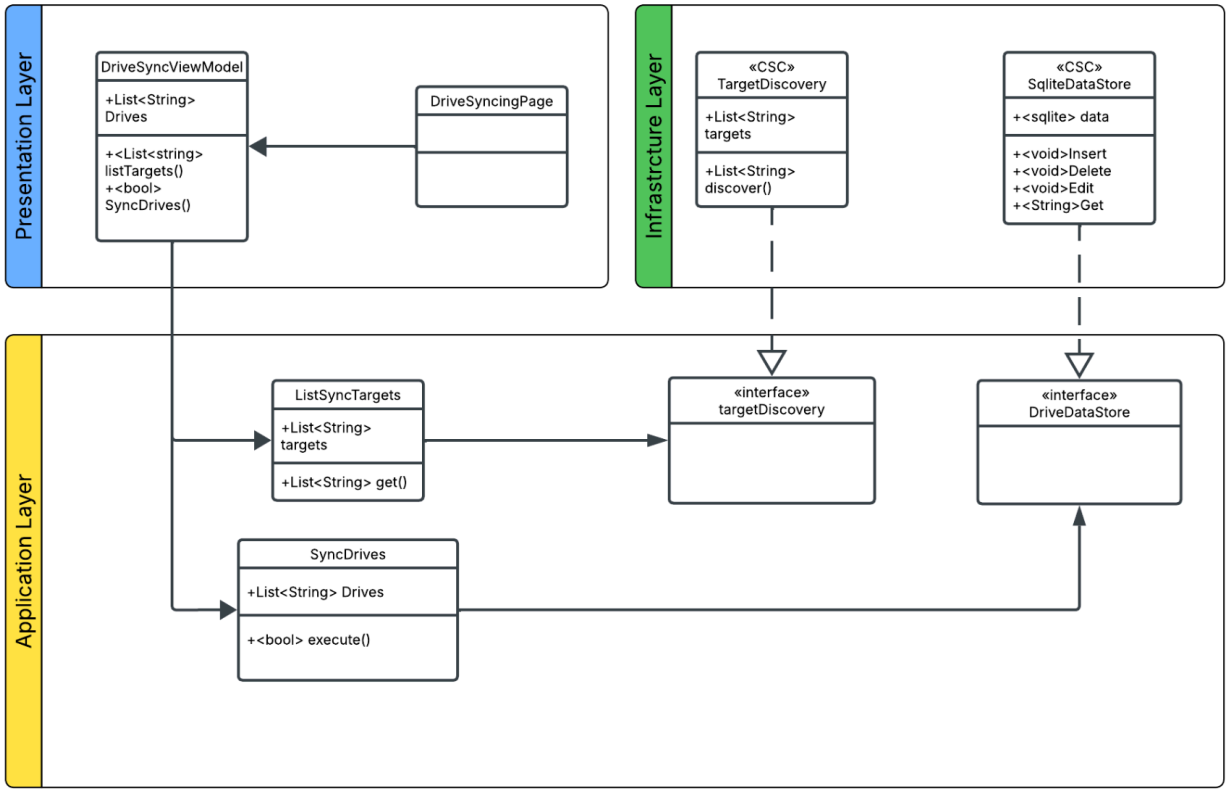



*Figure 7: Component Diagram for the Drive Synchronization System*

The external drive synchronization module is responsible for keeping different drives synchronized with the same data while out on the field. This helps further ensure that the data is kept safely as it is in multiple drives now. It also allows the team to be on the same page in general. This module, also relatively small in comparison the first three, depends on app layer use cases like listing the synchronization targets, and synchronizing the drives in the first place. Both of these use cases depend on interfaces like targetDiscovery to identify the drives connected to the system and

to store the ones selected by the user, as well as DeviceDataStore for the comparison of data between drives. Since the data is formatted into sql tables, sql tools can be used for comparison of the two drives and to identify what data needs to be copied from one drive to another and vice versa. These services are used by the presentation layer, specifically the DriveSyncViewModel class to provide a user interface for synchronization and to alert the user to the status of said operation.

# Implementation Plan

Now that we have established the underlying architectural principles we plan on following during the development of the application, we define a clear roadmap towards the completion of the project. The development and subsequent testing will be divided into phases as follows:

## Graphical Visual

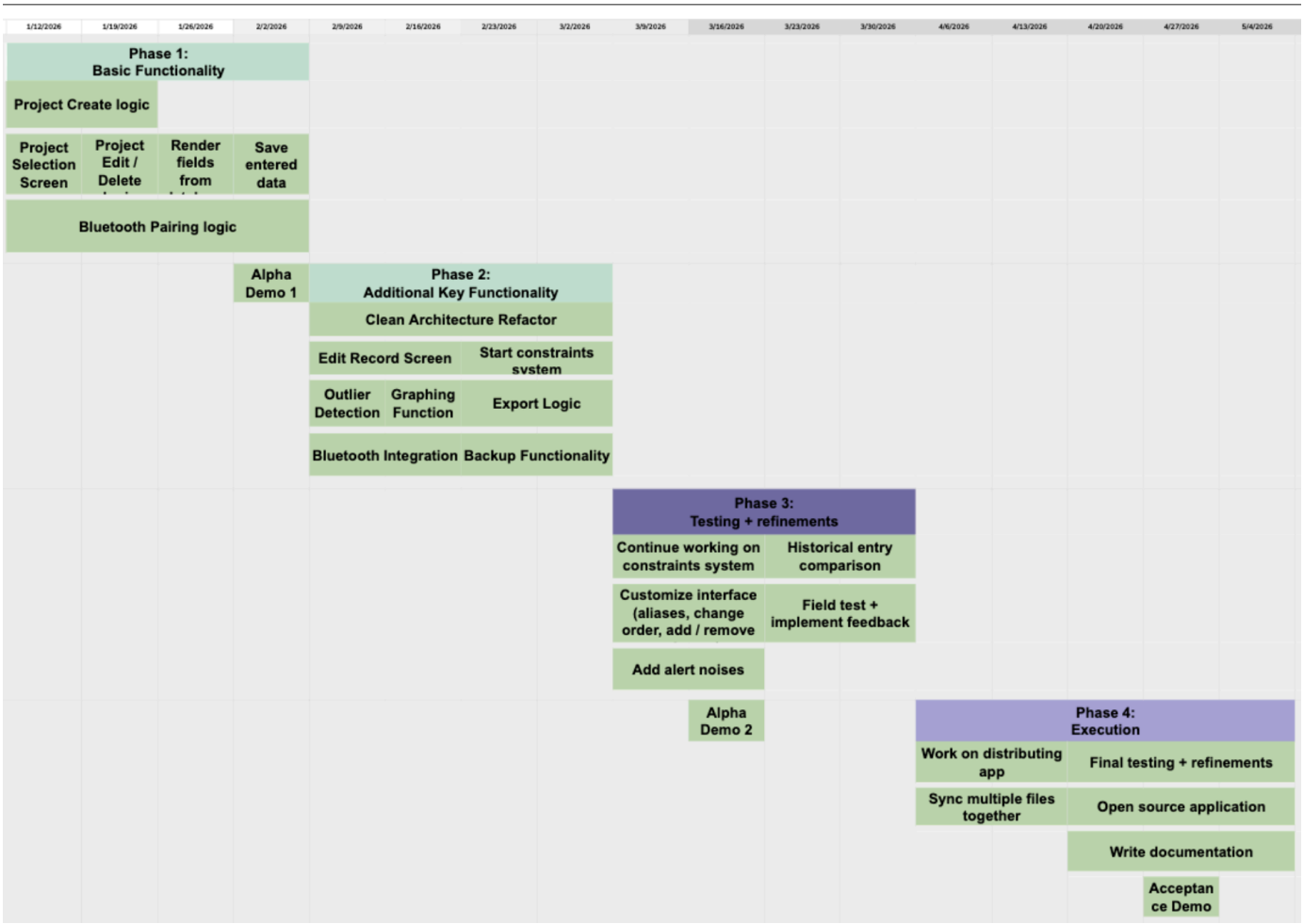



*Figure 8: Graphic Visual of our Implementation Plan*

# Development Phases

We plan on tying our phases to our demonstrations. Phase 1 corresponds to demonstration 1, phase 2 corresponds to demonstration 2, and phase 3 corresponds to the acceptance demo.

Phase 1: Basic functionality

- By this phase, we intend for our application to have basic database, Bluetooth, and backup functionality. Our application will be able to read and render fields from the database and save entered information to the database. We also intend on having the program be able to read information from a Bluetooth PIT scanner to a selected field, and back up this information to multiple external drives.

Phase 2: Additional key functionality:

- This phase will start implementing the additional key requirements. IT will consist of features like a Bluetooth scanner automatically entering PIT Tag information into the rendered fields, being able to edit entries after saving to correct data in the field, and also alerting the users of possible incorrect data entries using outlier checks on the entered fields' data.

Phase 3: Testing + refinements

- By this phase, we intend to have our application nearly fully functional. We aim to have quality of life features such as a constraints system, alert noises, a graphing page, syncing multiple data files together, and customizable fields working. We also intend to get the export functionality working by this point as well.

## Technical Risk

- Technical debt: Our current prototype is a monolithic architecture with tightly coupled modules. This creates a lot of technical debt, as minor refactors extend into many parts of the application's codebase. Our team plans to mitigate this by pivoting to a clean architecture. Clean architecture separates logic into multiple layers: the application layer, domain layer, presentation layer, and infrastructure layer. By separating logic and creating abstractions via interfaces, our application becomes more modular and maintainable.
- Library risk: Our most important dependencies are Riverpod, SQLite, and flutter_libserialport. Riverpod is a state management platform that makes it easy to access and change values that work across multiple widgets. SQLite3 is a library for interacting with the embedded ACID-compliant database SQLite, which will be used to store the records entered and saved by the user. Flutter_libserialport is a library used to interact with serial port devices over USB or Bluetooth. This is how our application interacts with the Bluetooth PIT scanner. Flutter_libserialport poses the highest risk for failure or obsolescence, being a small library for a niche use case. If the current maintainer stops maintaining it, there is a likelihood that future Dart versions may break it. Riverpod and SQLite3 are both very widely used, and therefore, it is likely that if the current maintainer steps down, someone new will begin maintaining it.
- Scalability constraints: Scalability is not a major concern for our application as it works entirely offline. As long as the application can run on the researchers' laptops, e.g., Windows and Linux, then we do not need to worry about scalability.

# Conclusion

This software design document presents a comprehensive blueprint for how our team will replace the SHOALS system with an application that is modern, maintainable, and usable. By improving all the bad aspects in SHOALS and keeping everything that was liked by the clients, we will make the biologists' conservation efforts easier. The new proposed system directly addresses the usability, reliability, and maintainability challenges that currently hinder biologists in their efforts to monitor aquatic life along the Colorado River.

The document outlines a clear architecture approach, with a domain layer at the center for the rules, and around that will be an application layer to see if any rules were broken. There will also be an infrastructure layer that will hold the implementations that the application will use, and finally, there will be a presentation layer, which is what the user will see upon opening the application. Some of our components in our plan include having customizable fields, having a backup system for the drives, a connection to the Bluetooth scanner, and outlier visualization and detection.

The implementation plan further shows the execution of our design. We plan to follow our demonstrations with three phases. Phase one will focus on basic functionality, where our team will work on creating a basic demo that shows the client what we have so far. Our second phase will focus on implementing key requirements to improve upon the basic functionality that we succeeded in completing in phase one. In our final phase, we will fix any mistakes that show up and add quality of life features so our app is ready for deployment with everything that was asked by our clients.

Overall, this software design document serves our team as a practical and actionable foundation for our development. It provides structure for implementation while being flexible at the same time, so that if a different development adds to the application, it will have the ability to

evolve, so that this application will contribute to bettering the systems of one of the most important rivers in the United States.