

A Modular Approach to Ground Station Control Software Design

Introduction

Much of the existing satellite oriented software and hardware infrastructure is steadily aging and is becoming harder to maintain. Operating systems are dropping support for many of the tried and true interfaces and libraries that are the foundation of the bulk of today's satellite control software. At the same time, the amateur satellite community continues to innovate by squeezing more out of the current satellites and designing more features into future satellites.

The next generation of ground station control software needs to be able to meet the challenges of this changing landscape of satellite operations: new radios, new satellites, and fundamental changes in operating systems and computer hardware. Ultimately, the software should also foster and support as much experimentation as the rest of the amateur pursuit.

Designing software to meet these goals effectively requires re-architecting the classic screen or GUI-based satellite prediction and control application - not an undaunting task by any means. Fortunately, there are a few resources at hand.

First, we must learn (and perhaps borrow) as much as possible from existing open source projects. There is a lot of work that shouldn't go wasted, both in the form of code and documentation. We will, for example, be able to encapsulate a number of algorithms from sources such as Michael F. Henry's *OrbitTools* and *Predict* from John A. Magliacane, KD2BD. Additionally, Paul Williamson, KB5MU's "One True Rule for Doppler Tuning" and Anthony Monteiro, AA2TX's paper entitled "An Object Oriented Approach to Automatic Radio Tuning" both outline the established best practices for tuning and object modeling.

The second place we should look to for help, and the one on which this article focuses, is the practice of pattern-based development. Design patterns are repeatable solutions to commonly occurring design issues. Using patterns will provide tested, proven development paradigms, help to prevent subtle issues, and improve code readability for those familiar with the patterns. In

short, patterns will help us to design a piece of software that is easy to maintain and extend.

Lastly, we will need solid foundation on which to base the application. It should provide at least support for advanced design concepts, native platform independence, and hopefully has wide support base with a number of useful tools and libraries.

Application Model

The requirements outlined in the introduction could describe what, in modern software design, is called an *agile* application. An agile application is defined as a loosely coupled set of functional units tied together by an orchestration layer which is easily modified to address changing needs and is scalable by design.

For now, let's call the orchestration layer the *controller*. The controller will ultimately be in charge of two tasks. The first is bootstrapping the application by loading itself and each of the functional units into memory. Unfortunately, it does not know what type of functional units it will be loading. In fact, no one can predict what functional units may be required in the future no matter how well the application was designed. To solve this problem, we will use the *factory* pattern to load and initialize abstract classes whose actual implementation is independent of its interface to the orchestration layer.

The controller's second responsibility is managing communications between each of the functional units. Let's call these units *modules*. Each module is capable of either generating messages or receiving them. A module notifies the controller of its interest in a particular type of message by subscribing to it. Any number of modules may subscribe to one type of message so that they are all notified of the same message. This pattern is called the *message bus* and effectively isolates the operation of each module and defines their interaction through formal messages.

Figure 1 illustrates a number of possible modules and the controller as a message bus.

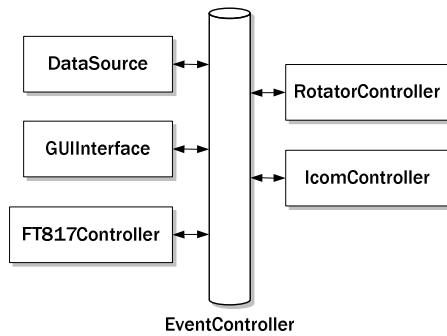


Figure 1 - Message Bus Pattern

Modules should have a well defined and concise scope of responsibility. Generating satellite data, tuning a radio, or displaying information in a GUI are examples of some tasks each module might be responsible for. Each module should operate completely independently of the other modules, generating messages only within its scope and subscribing only to messages which affect its operation. Isolating the modules in this manner will greatly simplify debugging and extending functionality.

We can now identify some of the possible message types and outline a possible message handling scenario.

Message Type	Description
Start Track	Indicates that tracking should be initiated for a given satellite. The StartTrack event includes the name of the satellite and the name of the mode.
Satellite Change	Indicates that a satellite's state has changed. Provides current azimuth, elevation, and Doppler values
Aos	Indicates that a satellite has risen above the horizon
Los	Indicates that a satellite has fallen below the horizon

Table 1 - Sample Message Types

A typical message handling scenario is as follows:

1. The DataSource module periodically computes satellite states for all of the satellites it currently knows about issues a SatelliteChange event for each update.
2. The GUI receives each SatelliteChange event and regularly updates the display based on the messages it receives.

3. The user watches the display, and uses the interface to manually select a satellite to begin tracking. The GUI module issues a StartTrack event.

4. The three modules which are subscribed to the StartTrack event (FT817Controller, IcomController, and RotatorController) each note the satellite to begin tracking and at this point begin listening to the appropriate SatelliteChange events.

5. The RotatorController simply watches the updated satellite data for azimuth and elevation data as the satellite passes overhead and matches the values.

6. The radio controller modules note the current Doppler correction values and adjust the uplink and downlink frequencies accordingly.

7. As the satellite passes below the horizon, a LOS event is triggered and the three controller modules return to their initial state until another StartTrack event starts the process over again.

Data Model

Just as the application design is based around an agile architecture, the data store should be robust and flexible. We need to anticipate the fact that the future needs of the application may differ from those of today and ensure that today's decisions will have minimal negative impact on the evolution of the application. The data model is similar to what referred to as the object model, but extends to represent how the data is persisted and presented in a human-readable form.

The hierarchical nature of object oriented programming lends itself towards a hierarchical data model. It is possible to visually represent the underlying object model and is similarly extensible. Many of today's runtime frameworks also support the direct creation of in-memory objects from persisted data in hierarchical formats such as XML.

Storing satellite configuration data in a hierarchical format as described in Figure 2 easily maps to a data structure which allows the application to not only know the satellite's current state, but also which modes are available as well as the mode's frequencies and emission type.

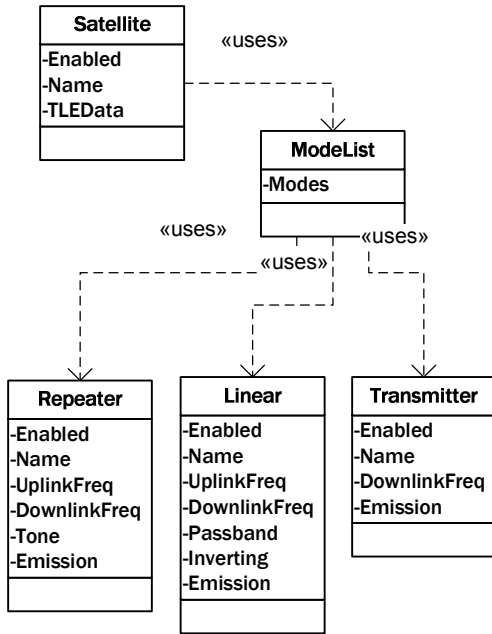


Figure 2 –Satellite Data UML Diagram

The framework should provide not only a common data repository, but a common mechanism for multiple modules to access the same data concurrently. This is achieved by using a thread-safe implementation of the *singleton* pattern. The singleton pattern ensures that only one instance of the data is available, and the thread-safe implementation ensures that multiple modules may access the data concurrently without causing unwanted side-effects.

Summary

This brief article describes the architecture of a basic system meant to stand as the framework for a flexible ground station software package. Unfortunately foundational work like this does not have the mass appeal of applications with flashy user interfaces and other bells and whistles, but by building on top of a solid groundwork, the end product will be capable of more features, each with a faster turnaround time than if the application were more monolithic.

It should be possible to see how a modular approach simplifies experimentation by lowering the initial learning curve of learning the intricacies of orbital mechanics or the finer details of CAT programming. The developer, instead, is left to concentrate only on the realm of the individual module and its public interface. Some

other ideas for simple modules which could operate only on the messages named in Table 1 include:

- A data/audio recorder which automatically records from AOS to LOS using standard data ports.
- Support for a USB jog/shuttle wheel which allows tuning around the passband of a linear transponder.
- An auto-tracker which automatically issues StartTrack messages for a number of satellites as they pass overhead and alerts the operator to upcoming passes.
- A GPS module which automatically updates the base location for mobile operation

The true strengths would come from implementing useful modules that meet use cases from real operators. I hope that some readers may be curious enough to download the prototype and be inspired to help develop, test, or write use cases. Much more information and downloads can be found at <http://code.google.com/p/satcontroller/>