SL2.0 GUI Flux Concepts

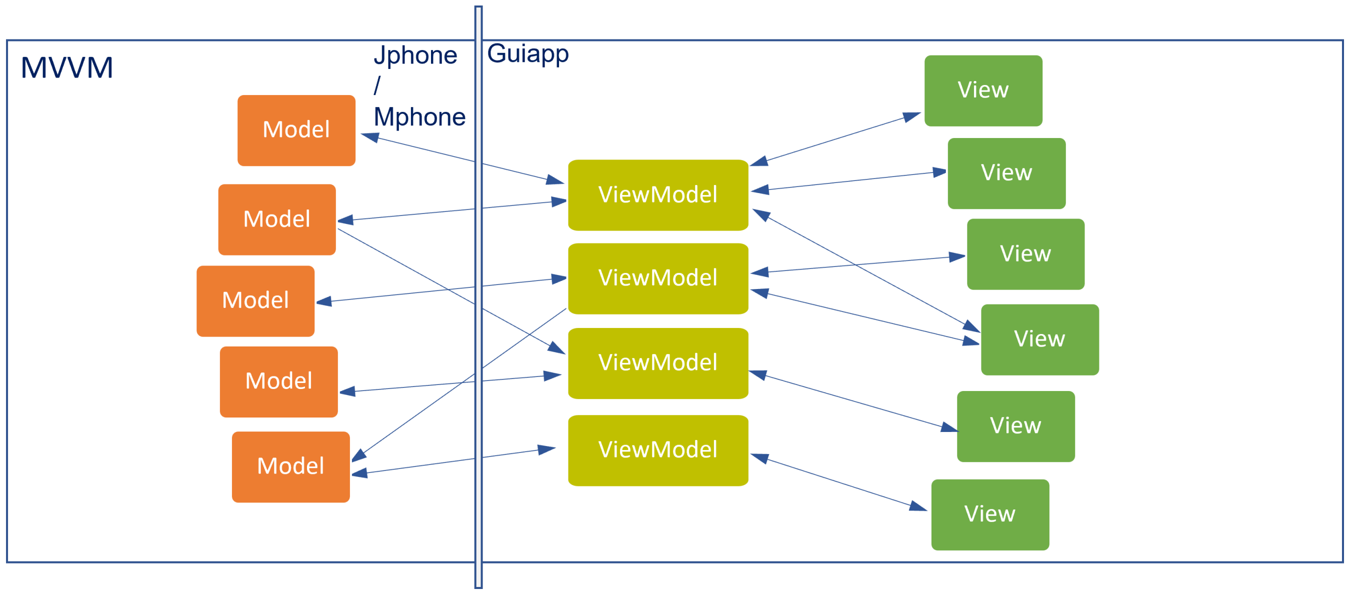
GUI application design patterns

Design patterns are typical solutions to common problems in software design. There are two well-known design patterns for GUI applications: Model-View-Controller (MVC) and Model-View-ViewModel (MVVM). The design patterns specifies how to structure the code and how to separate different layers like GUI and business logic.

If you ask *n* people about the difference between MVC and MVVM, you will get *n* different opinions. There are various implementations even with just one pattern like MVC, for instance.

The SynergyLite platform (SL) uses MVC / HMVC (Hierarchy MVC) on the Mphone/Jphone side and MVVM on the Guiapp side. The overall pattern is more like MVVMC (MVVM-Controller) which is a mixed and more complex pattern.

MVVM Pattern



Picture 1 MVVM pattern diagram

The SL2.0 platform uses Qt as the GUI framework. The Qt QML application is a perfect fit for MVVM pattern.

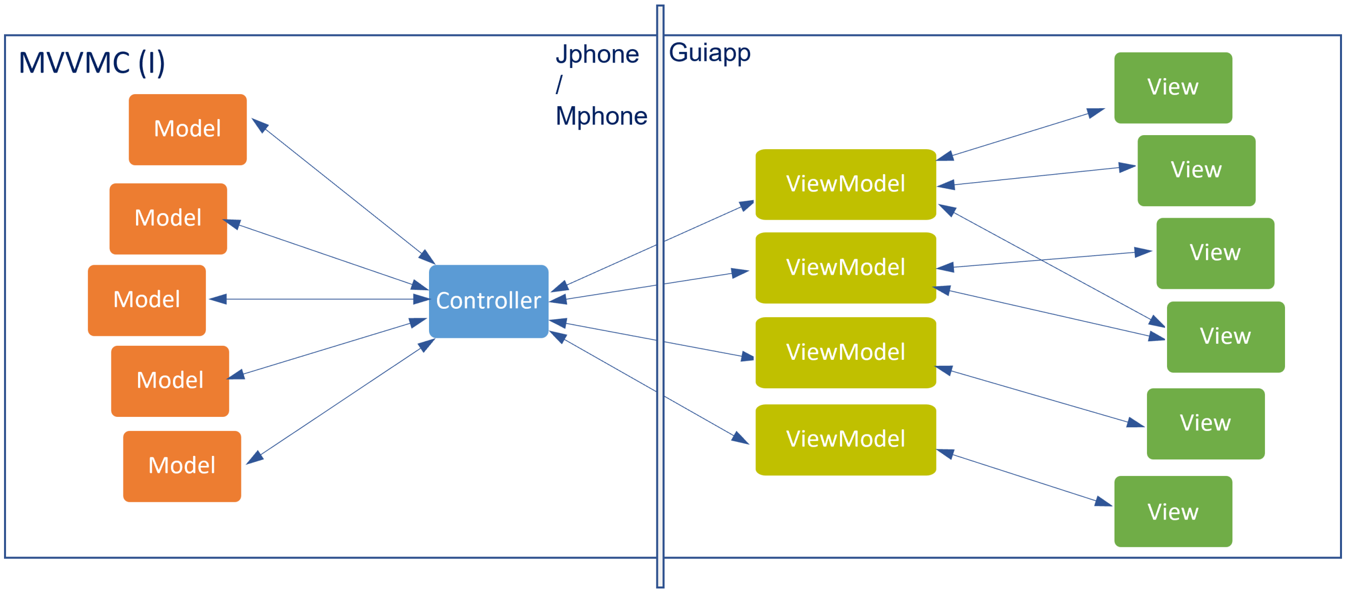
The model is responsible for business logic and manages the data. The model is implemented by Jphone in the on-prem mode and by Mphone in the MPP mode.

The view is accountable for displaying the data from the view model. The view is also the only user interactive component. The view can be rendered with QML and JavaScript.

The view model is accountable for getting the data from the model and operating the model. Typically, the view model can be implemented as an QObject written by C++. Some functions of the view model can also be written in JavaScript code too (like those onXXX code blocks).

Note: The MVVM pattern is one of the desired patterns for SL2.0. However, the actual GUI application pattern in current in SL2.0 is the so-called MVVMC pattern. With the pure MVVM pattern, the functions of view model will become rich.

Current SL2.0 MVVMC pattern



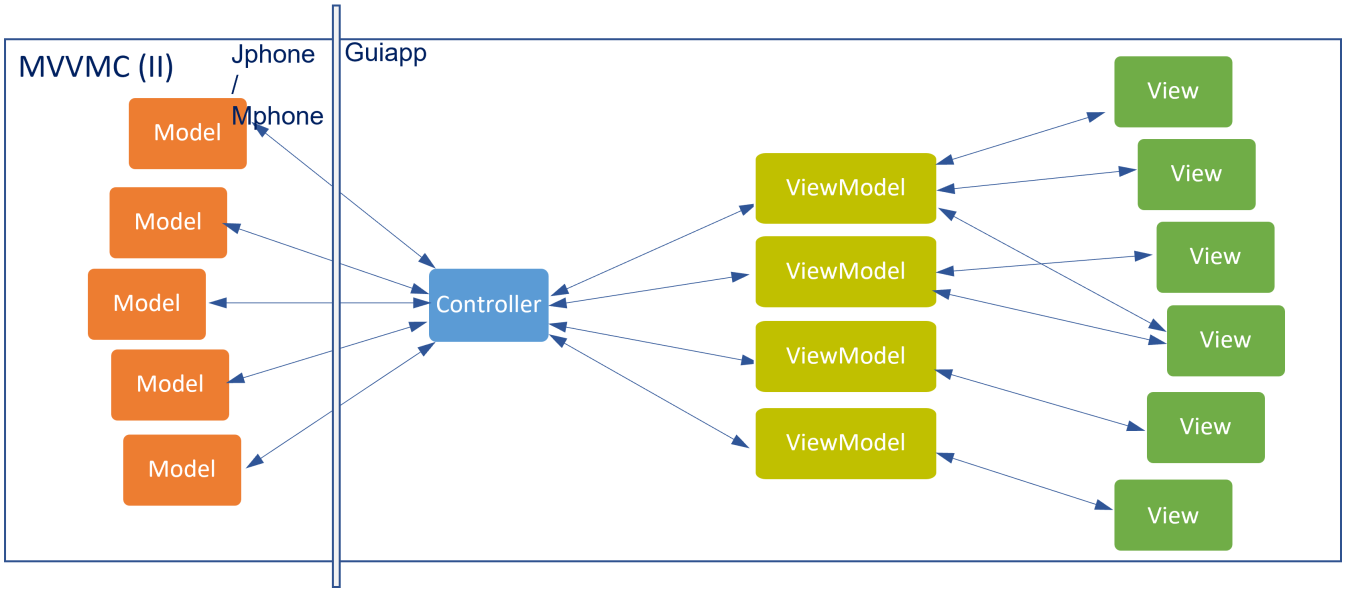
Picture 2 MVVMC (I) pattern diagram

The "C" in MVVMC here stands for "Controller". Although this pattern looks very confusing, it's the pattern used since SL1.0. The reason behind this is the evolution of SL GUI applications. Jphone (or Mphone) used the MVC pattern from the beginning. Later, the view part was replaced by Guiapp.

The controller parts are retained in Jphone and Mphone, respectively.

The biggest problem of this pattern is that there are two different controller implementations for Jphone and Mphone. There is duplication of effort and it is difficult to keep both controllers consistent in behavior.

Improved MVVMC pattern



Picture 3 MVVMC (II) pattern diagram

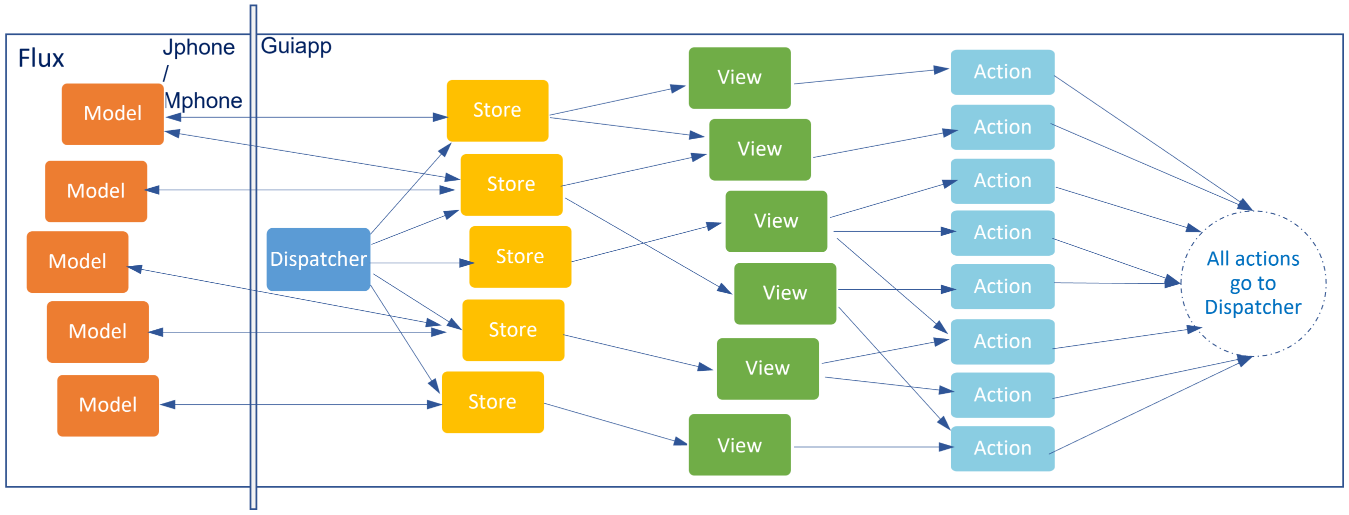
We can consolidate the two controllers by rewriting a single controller in Guiapp. The model part is not necessary to be moved. One reason is the business logic is too complicated to be rewritten. Another reason is that the models of Jphone and Mphone are different indeed.

There are still problems in this improved MVVMC pattern.

- The overall problem is its complexity. The responsibilities of the controller and the view model are difficult to be separated.
- Another common problem of MVVC pattern is a lot of fragmented code across all the view models. The data flow between the view and the view model are bidirectional. The view and the view model are not necessarily one-to-one relationship. One view can trigger update to multiple view models, which makes it hard to track where and when data is changed. One way to solve this is to extract a global common view model. But one view may be mapped to multiple view models.

- There will be vast size of view mode code. It requires view model abstraction to maintain clear data flows. It is usually a challenge to remove duplicate code in view models. The relationship between view model and view will get more complicated as new features are introduced.

Flux pattern



Picture 4 Flux pattern diagram

The basic concept of the Flux pattern can be found here. Let's ignore the complexity of the relationship between model and store for now. The core concept of the Flux pattern is the unidirectional data flow. Traditional MVC pattern has two-way data bindings in the view. The model data may be updated in multiple GUI components and there may be cascading data update, which leads to unpredictable results.

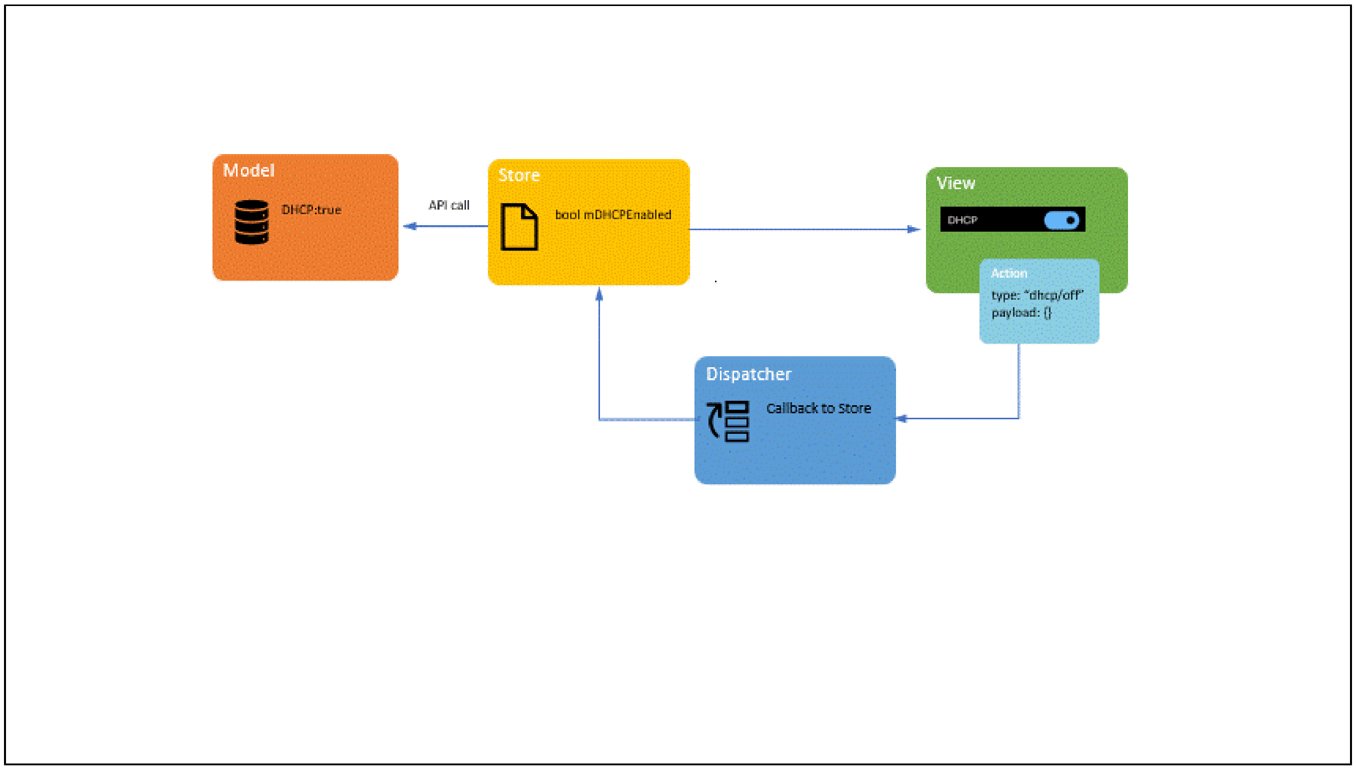
The Flux pattern has some advantages. The view can be better decoupled from the model. The view displays the data but it does not updates the data directly.

The dispatcher is a central hub. All of the actions flow through the dispatcher, such that we can easily track the data flow.

In Flux concept, the store contains the application data and logic. The store registers callbacks to the dispatcher for actions. The store also emits data changed to the view. For SL platform, the store follows the Flux pattern. With a slight difference, the store still keeps the model layer of the MVVM pattern. We want to keep the vast model layer code in Jphone/Mphone. The store can talk with the existing model layer and does not necessarily have many business logic. The Qt/QML framework somehow requires a view model layer, so the store will also work as the view model.

The action entity has two parts: an action type and a payload. The action type is semantic and descriptive. It can contain a resource and/or an operation, for instance, "settings", dhcp/on" and "dhcp/off". The optional payload may contain all the data that the action needs to take care of. The payload can appear as a key-value map.

Flux data flow example



Picture 5 Flux data flow

This picture shows the data flow to update the DHCP enabled state from the view.

Qt C++ vs. QML

The most popular options of Qt applications are C++ and QML. While the Qt framework is C++ based, we can also code with QML and JavaScript. Coding in QML has several advantages over development with C++. QML is a declarative language which is cross-platform and can help to develop GUI application quickly.

For the dispatcher and the store, we will still choose C++ over QML for the following reasons:

Existing functionalities and libraries
From the technical perspective, C++ has a lot of built-in functionalities as well as rich libraries. We don't have reinvent the wheel for most cases. It's much easier to implement multi-threading programming and mix C++ with native code.

Performance
C++ is a compiled language while QML is an interpreted one. Executing the machine code is generally much faster than running the code through an interpreter.

Stability
C++ code is type-safe. For parts where stability and security are important, using C++ helps to make your app less error-prone.

Code readability
QML and JavaScript are accountable for displaying the data. There are a lot of code related with item rendering, such as size, position and layout. If more and more code goes to QML and JavaScript, the code will be messed up and hard to maintain.