H-MBD Architecture

MBD adoption for HMI development

FPT Global Automotive

Yakiro Nguyen

Solution Architect

FGA - Business Unit #2

Contents

1. Design Philosophy

HMI development for In-vehicle Infotainment (IVI) continues to have a bright future, but only for those who can develop profitable models. In fact, even tightly budget IVI system now have ton of features that keep growing exponentially by market and trends pressure. Things go faster, needs to be better but must be cheaper also. Anyone who want to contest now must prove the ability to survive after massive changes. Which apparently being a disaster for the legacy, unoptimized workflow, that expects nearly perfect specification and timing at first. And the truth: there is nothing that's perfect enough, so we need to pay an expensive price for that. Motivated by the MBD successful in electrical engineering field, that could be considering faces similar challenges, we now search for an adoption of MBD in HMI development, to mitigate the pain for serving the changes at that high-throughput, which usually leads to unreasonable verification and validation cost. MBD, while proven itself as no joke solution in its original domain, to locate and correct errors earlier, still not quite suits to our problem set as it sounds. The most important merit of MBD is that reduces cost for making a lot of real hardware just to be realized they are non-sense trashes. How this merit maps to the HMI development? We are facing different problem, our system is software only, of course that may depend on hardware, but not itself being the hardware. In fact, HMI development suffers from messy dependencies instead, so early/cheap rejection is very hard or sometimes impossible. Fortunately, modeling and various of other techniques still could be applied and shows its value to drastically improves the HMI development.

This section defines the most importance philosophies to drive the effort.

1.1. Rapid prototyping and iteration

A good prototype is something that allows experiment of a specific idea for our HMI system. The key points there are it must be both fast and cheap to create and shouldn't depends on bunch of constraint/prerequisite as usual. Therefore, it could be throw away easily if can't shows potential while still minimizes the gap to turns that to a real product feature if necessary. Let's define a typical process of building a product feature by: prototyping, tweaking, result validation and verification, back to re-prototyping, more tweaking etc. Each cycle can take hours, days, weeks and sometimes longer to complete. Rapid iteration is invaluable tool to minimizes the non-sense waiting times as much as possible, so we can achieve the feedback for each change that we made immediately, typically less than one second.

Benefits:

1. Save times, reduces cost.
2. More tweaks, better quality.
3. Keeps promise and time to market.
4. Enables agile development.

1.2. Provides open and common environment

Believe it or not, there are lots of bottle neck for most of legacy workflow that accidently requires specialist even for a smallest change. That results as wasted times to transferring the problem, waiting for another one who may at the end just misunderstand your intention. Instead of that, we should try our best to give the user the ability to resolves problem independently. This situation is when you made a birthday cake and eat that yourself. That becomes a paradise which give as much power as possible to everybody. There are no mystical zones, every stakeholder of the project should be able to play on the same ground to interact with the system independently, at every stages of the development if that fulfill their jobs.

1.3. TBD: Strategy question

The most confusing thing there in my opinion is “project scope”, in other word what we are going to do. Of course, this document talks about high level / generic decision a lot, but in fact it still be created to serves a concrete situation. There are so much concerns that could impacts our final decision e.g. legacy system, human resource, standard compliance, 3rd party solution or competitor, OEM force and demands etc. just somethings that I know. Some of that even already sound conflicted. First, we want to leverage 3rd party e.g. CGI Studio, Kanzi or Qt Quick to take the advantages of each, or sometimes just by the OEM demand. Next, to ensures the human resource capability, we need to add another abstraction layer to standardize the tool, process, knowledge and so on. Technically, we now building an engine that depend on 3rd rendering library. However, the scope of these 3rd parties are “a little bit” biased. They are not limited to being just a rendering library. In fact, they tend to add ton of high level feature and tool to prove themselves as an engine also, which add much more values. Furthermore, there are difference and limitation between 3rd party, so abstraction sometimes isn’t feasible as well, not to mention the maintenance effort to keep all these things up to date. Eventually we end up with a simple, stupid yet brilliant in-house rendering library.

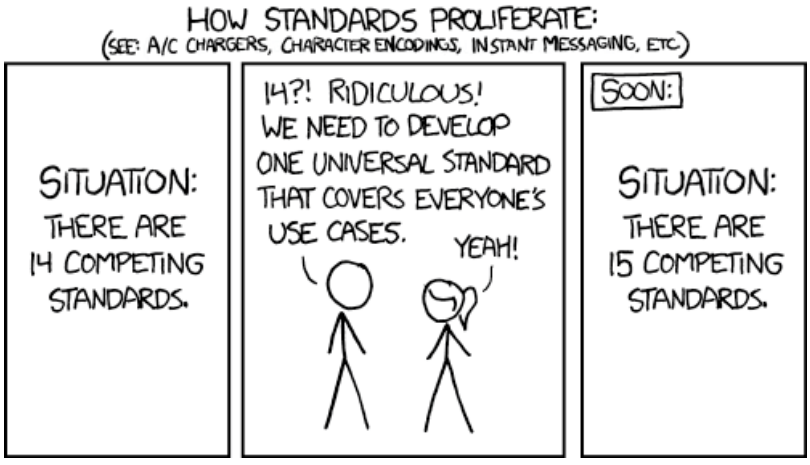

Figure 1: How standards proliferate

In my opinion, there are two ways to go for this project, either accept the fact and position ourselves as a competitor **at the end**, who made yet another better in-house engine, which feasible but may not reasonable depends on the budget and what being considered as top priority right now.

Or move on with 3rd party and accept there are fragmentation.

In the following sections, we favor the 1st option.

2. System Architecture

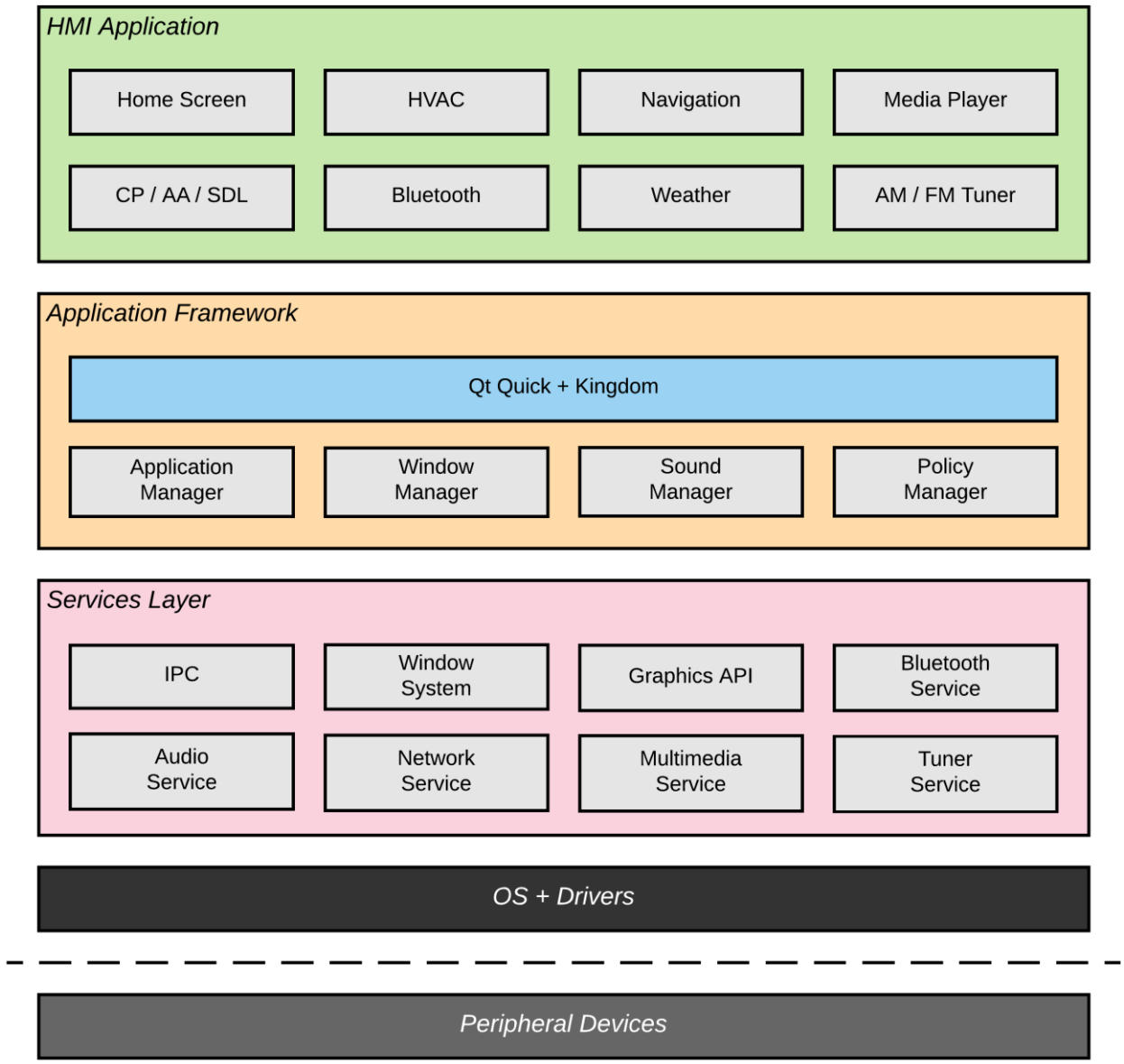

Figure 2: System architecture

Concerns:

1. Prototype with/without services.
2. Prototype with/without assets.
3. Prototype with/without boards.
4. Prototype with/without peripherals.
5. Multiple application (wayland, ivi-shell?).
6. QEMU, virtualized GPU & peripherals.

2.1. Facing the absence of dependency

Above concerns already shows a few real-world usecases that usually prevents prototype making, especially at the early stages of the development when most of dependencies are not available. In case of simple and mostly static resources e.g. GUI layout, sound, button skin etc. that could be easily overcome by just move on and make a dedicated preview tool to be used by the non-programmer users. Perhaps it's better than nothing and save us for a long time. However, things not always going on this direction, resource that requires dynamic behavior or may be non-trivial dummy data to be preview e.g. animation, 3D content, state transition etc. becomes popular. Our tiny yet promise tool now needs to be a monster with lot of features, and it's still not being the worst situation since there are features that even impossible to be implemented as separated tools. Furthermore, it already violates the philosophy by introducing a different working environment just for the non-programmer. Which is, at the end may be the same as our final product that ship to the customer. Well, we shouldn't try to cheat by provide a simulated, dedicated environment that so much different from the target hardware for them to working on at the mid/late stages of the development. It's best to experiments directly on the real environment if possible, at the target FPS, includes bug and issue as well, so everyone knows exactly what we are shipping right now. Not to mention who on the paradise want to get to the nightmare by maintaining a separated source code that is dedicated for the toolset, which basically do just the same work as the target runtime, then of course have bugs also.

2.2. Tool architecture

What we are searching here is a way to leverage the engine runtime to powering our toolset and provides additional running modes which may be crafted to aware about the situation. It should run fine without dependencies, provides reasonable interaction and feedback capability, while not limited itself to switching to real mode when the dependencies become available, so we are free to choose how best to do the iteration at each stages of development that suits our needs. On the other hand, tools should not be strongly coupled to the engine runtime to improves the freedom and flexibility for both of that. The tools can freely be written in any language and technology e.g. WPF, Web

based, Qt etc. The same story also applied for engine, so changing runtime data formats will not impact the tools.

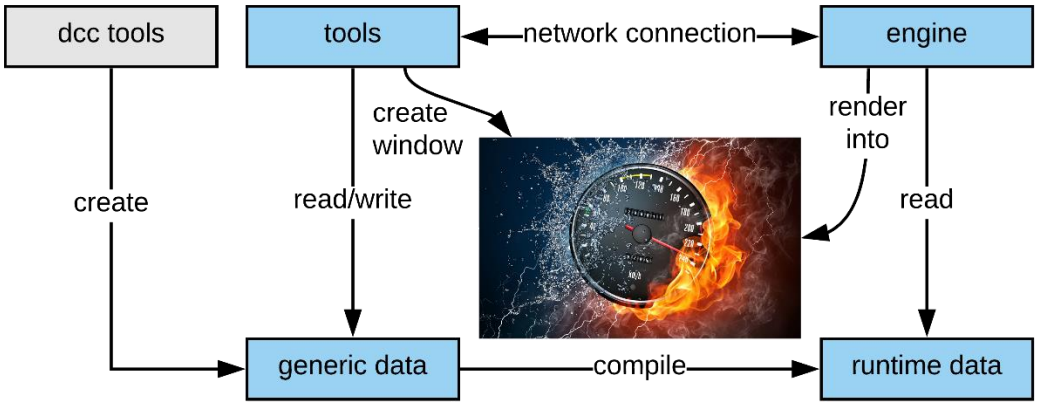

Figure 3: Tools architecture

3. UI Design Pattern

Let's answer a simple question: how to iteration with absence of almost all non-HMI component e.g. service, hardware, driver etc. Furthermore, how to iteration even without programming. Apparently, our engine and high level logical code base must be portable at first, that no doubts what we should achieve. Next, we use/create reusable model e.g. GUI layout, skin, state transition etc. then experiment what we made. For missing dependencies, either making a stub component or switch to a special running mode so that dependency isn't a dependency anymore. We just pick MVVM pattern as the core UI design pattern here and describe it more later.

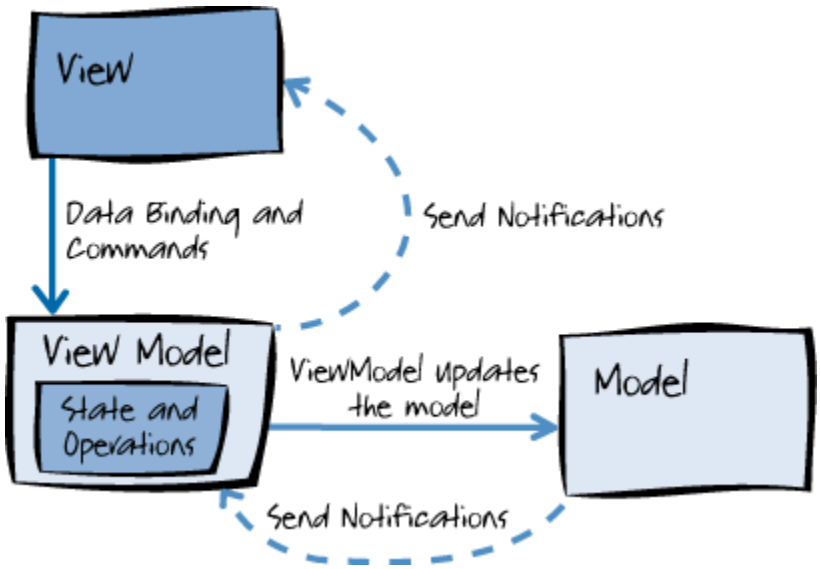

Figure 4: MVVM design pattern

3.1. Let's stub together

In the constitution of the United Paradises states that everyone joins free and equal, have the right to make stub for whatever HMI system they want. That make service/peripheral stub only solution apparently being an unforgivable violation, we must provide an alternative solution for ct. Phuong[1], who may just have enough ability to calculate the Fibonacci number in JavaScript, or risk part of our life in the Freedom Jails of Paradise (FJP). In this case, "Visual Stub-io: Service Edition" may suits perfectly to the paradise laws as usual, so we could forget about it now and let's stub the Model or ViewModel instead. Figure 4: MVVM design pattern shows exactly what we want, provided tools to define model of the Model (data) and ViewModel (event & state) so programmer and ct. Phuong could sit down to easily define these parts together. After that, she can interact with the system via sample data and event injection.

3.2. MV-VM benefits

VM stands for ViewModel, it is the **model of a View**. Yes, you read that right, even just looking at the pattern name, we already see that so much relevant. While this conclusion is a little bit hilarious, that is exactly what we are searching for.

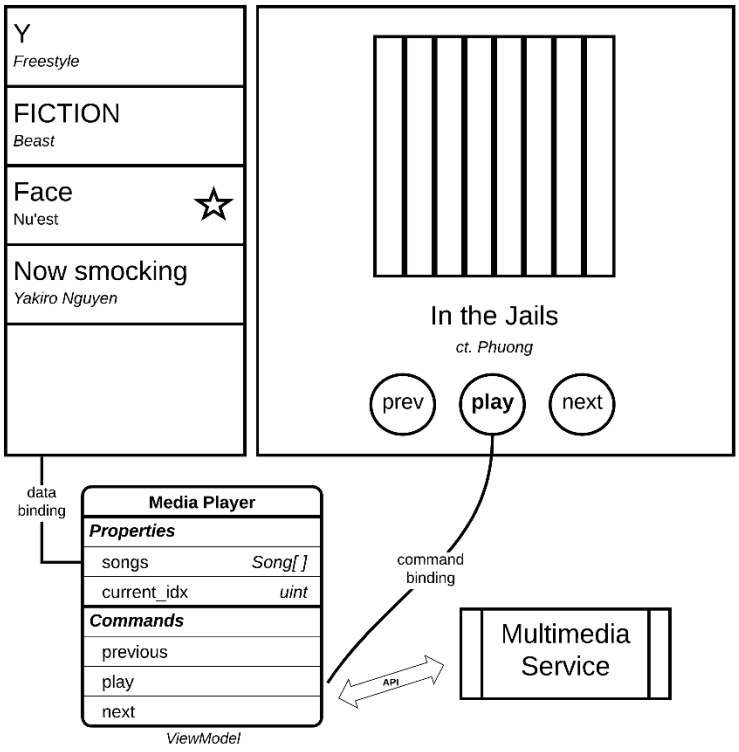

Figure 5: MVVM media player

[1] ct. Phuong: aka comtor Phuong, sometimes be misunderstand as captain Phuong though. ct. Phuong is a mystic talent bachelor of Japanese communication, she's working for an automotive software company in Vietnam. She has zero experience about programming, but still have great passion about IVI hacking from her childhood dream.

ViewModel describes how the View functions, and what information it provides to the user. They are being loosely coupled by property and command binding. We are free to change each of that component if we keep these interfaces. Furthermore, ViewModel basically knows nothing about the View, which is the most advantage of MVVM over MVC. In MVC the View is just a passive dumb that rely on the Controller to update its content, if we change the View, we must change the Controller also. That is so much pain and not ct. Phuong friendly as well, a typical freedom violation that could be leads to many years in the FJP, by precedent. Another benefit is ViewModel could be written once, easy to test and reuse later despite many cosmetic changes in the Views. Finally, we need a data driven / visual way to define ViewModel interface and may be even simple behavior, so ct. Phuong will not get angry and throw us into the FJP. The best scenario is she and programmer could define a stub ViewModel first, so she could use that while waiting for Service and programming part ready.