Design 922
Individual Reflective Report
Xiang Wei

One of the biggest reflections this project brought to me was a careful look back at my process of learning a new programming language and programming with it. When doing programming tasks or learning a new language, I was a bit too spoiled by google. Since I already have some basic programming skills and knowledge on the mainstream programming languages, I tend to start with reading sample code. Find the main function, analyze the structure, then go to a rough interpretation of the code by analogizing known similar languages. I would cut a big paragraph of code into small blocks to quickly get its meanings. Once I get each block's job and the connection of these blocks, I make a backup and then start modifying the sample code. I will try to first clear up the code, delete the most irrelevant functions, comment on all the potential reusable functions and only leave the main structure in the code. Afterwards, I gradually added the functions to validate my interpretations. If it doesn't work as I expected, I'll study all the code here in detail. I go back to the code and find out the part that might cause this error or I didn't fully understand. In order to understand, I need to study the syntax. This is one of the few situations that can drive me to open the documentation. I was so bad at reading documentation. One of the reasons is that most of them are quite tedious. I understand that with such rich information, it's impossible to shorten them. Luckily at this point, I can only care about the usage of that function. Sometimes, I will also pay attention to the parameters' meanings, if they are helpful for me to understand the connection of the code blocks. So once I validate all the parts of the existing code, I start to build my application.

The general idea is to first build the big structure of my app. After understanding the existing structure, I will comment down all the ideas and potential features at its position that I would like to implement later. I tend to keep the code clean and finish all features step by step. That can help me avoid importing conflicts and porting code.

To implement new features, depending on the difficulties, I would either manually code it, or directly search for an existing solution on google (for a normal popular language). Then I adapted that piece of code into mine. In this specific situation (working with GUIT), I would make the demo code as my small google database. Although it's a bit hard, because we can not use natural language to specify the task to locate the working part of code, I can still abstract a similar function by playing with the executable. And the small application in the demo app covers almost all the basic tasks that one might need. By finding the piece of code that can be reused, I go to the documentation, check the input and output data type, and adjust it into my own code. At some point, I even feel it's more convenient than google. Because everything is more integrated and unified, which makes adapting easier.

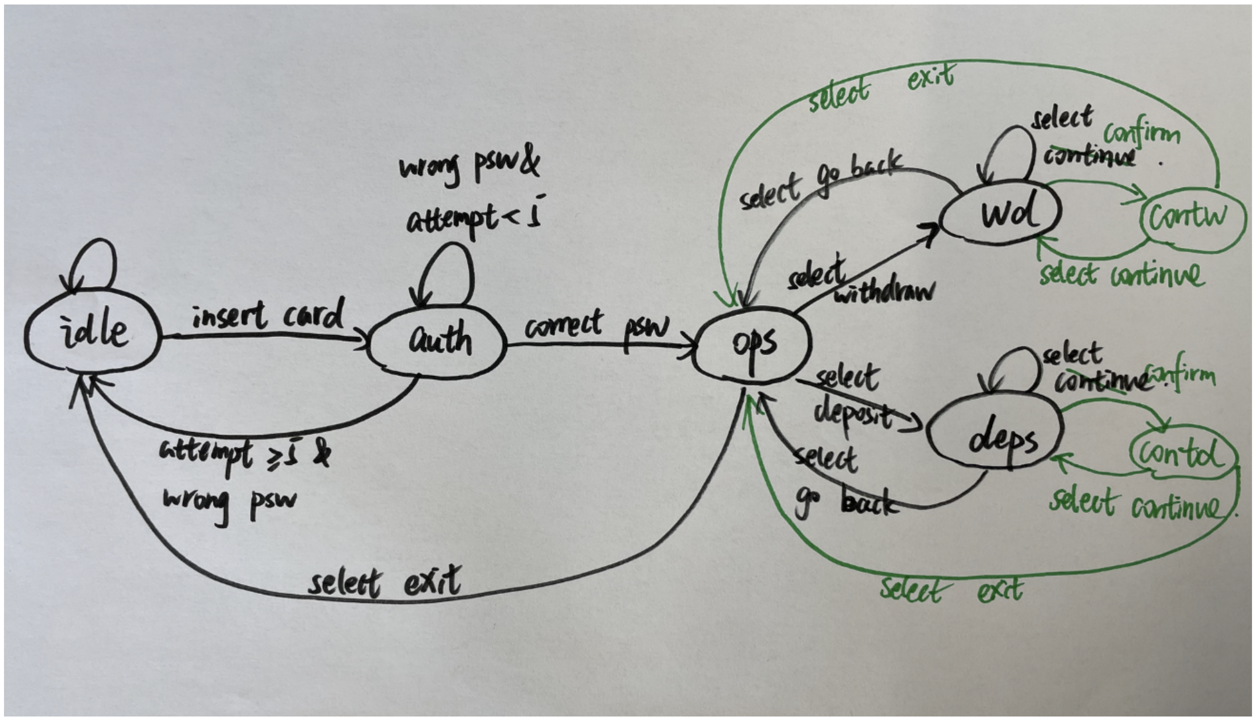
In our ATM project, I would like to start with having the interface, then add interactive components. To build a layout, it's always better to have a good reference, so that we can easily abstract the layout model that we need. And that's the reason why I drew this down. Same for the state machine later. It really released a lot of workload for me to later iternarate again and again. I found the layout gadgets in GUIT are super convenient and intuitive. Simply with just two gadgets, "VBox" and "HBox", I structured my entire interface. But there is also the negative side. With these "smart" gadgets, we don't know how much control we still have on them. What's more, layout issues can always be tricky, especially when you incorporate it with a new language. Some solutions on layout do need some creative and great amount of testing to come out. Sometimes we can't adapt available solutions, sometimes we even can't find useful samples. Therefore, the layout sample code lets us play with is a great idea. Yet, from the playground to the code, there is a big gap which makes me struggle for a while. I highly suggest we can specify this connection inside the demo. For example, replacing the `x-axis : <drop down menu> y-axis : <drop down menu> ` by the pseudo code `<< (HBox("{align: <drop down menu> drop down menu>  ;}")  `. I think it eases the users' efforts to self build this linkage.

When moving to the interaction part, here join in more logical and unique stuff. Here, we relied less on others' solutions but more on ourselves. Therefore the most important thing is to make the language syntax sense-making. A big factor to make it sense-making is the sense of familiarity. If it aligns to some other language (including the natural language that we use) that I've seen before, I could quickly adapt to it, such as 'Python'. Syntax in 'GUIT' meets this requirement perfectly. "<<" means add to, which is also used in natural language, especially when taking notes. The callback function is similar to lambda expressions in C++. But I'm not so used to C++, so it took me a while to master it.

After working for a while, I started to try some more advanced features, like using a state machine to make my code more clear. There is a great amount of code from the microwave application that I can port. But to fully understand and be flexible enough to use it, I asked for some external help. Because it integrates with many concepts that I skipped. I don't know if it is a good idea to adding more tutorial inside the code. But for me, I do take the two lines of explanation super helpful:

```
// Guard => like a if when the button is clicked
// On => activated when the value of auth or idle changes
```

It also builds a bridge between the code and the demo, like the layout demo. Using a state machine is also like drawing the layout. It requires a global design. A good design that takes the pressure off a lot of iterative work. But all success does not come overnight. In the actual implementation, we might modify the original design. Some are due to unsupported technology, while others are due to factors that were not considered in the design. For example, in my cash machine project, I added two more states (green in Figure) to the state machine I originally designed. Because I found out that I need an extra state to show UI hints.



I took this ATM machine as my project topic was inspired by the microwave machine in the demo application. I also think about vendor machines. I pick ATMs because it's more complicated and it's quite the opposite of a microwave and a vendor machine. Because these two machines are notorious for confusing interface designs, although they have tried their best to clear all the features with limited buttons in limited space. While the key idea of ATMs is to avoid ambiguity. No one wants to risk money. And ATMs don't have any restriction on buttons. In addition, I was also thinking about the outcome, a complete individual ATM, perhaps can be made into a sample code in the tutorial.

On the other hand, I think the ATM machine can be a good task for pedagogical purposes. It can be divided into several individual steps. Such as mine : First, code the layout; Second, use statuses to build a procedure; Third, replace statuses with a state machine. It's self-complete on each step, with some predictable problems, i.e about syntax, CSS, etc. And the final outcome is more pleasant for students, to have a fully worked independent executable. Besides, with the follow-up methods, it leaves students big space to explore and study themselves. In that way, students can learn how to self-teach a new language. And it also trains the students abilities to write and read documentation.