CS 246E Fall 2021 — Tutorial 9

November 9, 2021

Summary

1 Model-View-Controller (MVC)

In class, we discussed the single responsibility principle which states that each class should have one job. Keeping this in mind, consider how we should implement an application which interacts with a user. Logically, if the program interacts with a person, it must be able to accept input and display information to the user. Additionally, aside from input and output, there must be some interpretation of what the input does and how that effects the display. Thus, we've come up with three responsibilities: input, output, and logic.

Model-View-Controller (MVC) is a design pattern which explains how these responsibilities should be divided amongst classes and how these classes should interact. Specifically, we should separate these jobs into classes:

- Model: The "logic" behind the application. This stores the state of the application and logic of how interactions should be handled.
- View: The "ouput"/user interface. Whatever the person using the application "sees". This could be a terminal, a GUI, sound, etc.
- Controller: The "input". How the user interacts with the application. This could be any input object: clicking with a mouse, typing on a keyboard, a touch screen, a joy stick, voice control, etc.

Each of these classes should be responsible for one thing: the model is the maintainer of logic and state, the view presents data to the user, and the controller receives input from the user.

Consider how information would flow between these classes:

1. The model will receive input from the controller.
2. Determine if the input logically makes sense given the current state and update the state.
3. Notify the view to update to represent the change of state.
4. Repeat.

Note that step 3 of the flow was to "notify" the view. The relationship between the model and the view is typically implemented using an observer pattern: when something happens which results in the state of the model changing, the view is notified to reflect this change. Note that we could have easily have multiple views being present at the same time showing data in different ways to interpret the data.

In theory, this works great and it works well when interacting doing simple tasks such as interacting via a command line: we can setup the controller to read from std::cin and the view to be storing and printing some visual representation to std::cout which results in low coupling.

However, imagine that the input is received via the user pressing buttons on a window. We can imagine in this situation the controller and the view are heavily coupled: the controller will need to be updated to know

which buttons exist in the window.

Example: "Maze"

2 NCurses

NCurses is a C library which allows the programmer to "print" dynamically to the terminal. While we won't be showing you how to interact with ncurses, the final example of the maze used ncurses and you must use it (or a similar tool) in your final project. It allowed for input to be read and printed without needing to use cin or cout.

There are many webpages you can find to teach you how to program using ncurses. For example,

- http://www.cs.ukzn.ac.za/ hughm/os/notes/ncurses.html#init
- http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/index.html

While both are a bit dated in appearance, they are good.