The MVC Pattern

The model-view-controller (MVC) pattern is an architectural pattern used primarily in creating GUIs. The major premise of the pattern is based on modularity and it is to separate three different aspects of the GUI: **the data (model)**, **the visual representation of the data (view)**, and **the interface between the view and the model** or the **business logic[1] (controller)**. The primary idea behind keeping these three components separate is so that each one is as independent of the others as possible, and changes made to one will not affect changes made to the others (**Open/Closed Principle**). In this way, for instance, the GUI can be updated with a new look or visual style without having to change the data model or the controller. This also helps ensure that each class performs one very specific purpose (**Single Responsibility Principle**).

Newcomers will probably see this MVC pattern as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else (**Open/Closed Principle** again!). Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

One of the great advantages of the Model-View-Controller Pattern is the ability to reuse the application's logic when implementing a different view.

Model

Generally, the model is constructed first. The model has two jobs: it must both *store a state* and *manage subscribers*. (For now, we only care about storing state…) The state does not need to be anything special; you simply need to define how you're going to store data, with setters and getters. People sometimes use the word 'model' to refer to the 'data model' -- the architecture of the data structures in the application.

In Della's House of Bagels, we had *model* classes for each of the objects of the application. Notice that the method that calculated totals was left to the Controller which had the action listener. This is one way to implement MVC. We could also have created a model class called Calculator, which would have been invoked by the controller's listener. There are no "hard and fast" rules, but if you follow the Open/Closed and Single Responsibility principles and document well, your application should be very maintainable.

[1] Note that in some cases, business logic directly related to a Model class might be manifest there. In this case, the Controller will execute methods in a Model class.

Adapted from *Computer Science Design Patterns: Model-View-Controller,* Wikibooks, https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Model%E2%80%93view%E2%80%93controller. Emphasis and examples added.

View

Once you write a data model, the next easiest thing to write is usually a view. The view is the part of the application which subscribes to a model. Usually it presents it to a *user* alongside a *user interface*, or GUI. The GUI contains other components too, which are usually part of the *controller* and can be handled later.

When you're writing a view, there are two things to think about: "what do I do when the state changes?" and "how do I display this to the user?" Your MVC framework usually provides *editors* for various properties in the model, like date selectors, text fields, sliding bars, and combo boxes.

In Della's House of Bagels, we used the View for the panels and the frame. Note that the panels don't do much but receive information from the controller (e.g., list of bagels) and pass information to the controller (e.g., the ArrayList of all the charges). Notice that the GUI just assembled all of the charges which came from GUI components. It did not even total them although the code to do that would have been trivial. Rather it depends on the controller to perform all the calculations.

Controller

In the case of Della's House of Bagels, the controller used the model to create the bagel, coffee, and toppings objects and pass them to the view. This is a classic example of a controller mediating between the model and the view. Notice that the Model knows nothing about the View. The View knows as little as possible about the Model. The View classes only know about abstract Model superclass Item (so they can grab the name and know the cost of each item). But the View classes know nothing about what kind of Item they are dealing with.

Final Notes

A more advanced application could have the user indicate that the shop is out of onion bagels, for instance. What would this require? More than likely additional Model classes like an Inventory class. One can easily imagine that as bagels are sold, the View invokes a Controller which updates the Model. The simple application of Della's House of Bagels only featured one-way communication (Model → Controller → View) and not the reverse (View → Controller → Model).

Note also that the View can talk to the Controller, which may turn around and talk back to the View. Della's House of Bagels features one such communication pathway.

Adapted from *Computer Science Design Patterns: Model-View-Controller,* Wikibooks, https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Model%E2%80%93view%E2%80%93controller. Emphasis and examples added.