MVC

It allows you to make your program more modular. Programmers might use the term loosely coupled. Essentially, you are breaking your program into separate pieces that don't rely upon each other to function. The idea is that you can switch out each piece without affecting the others. The pieces are "M" for Model layer which contains classes used to create objects (i.e. a student, a car, etc.). The "V" is for View layer, which is the interface to the program. This is how you program will receive input and send output. Finally, there is "C" for Controller layer, which controls the flow of data from the view layer to the model layers.

A good example is if you had an application that was built using forms for the UI, and the MVC design pattern was used, then you could easily change the UI to a web based UI.

[https://github.com/hord-brayden/CIT-360/tree/master/14 Topics/Afghan-Trail-master/Afghan\_Trail/src/byui/cit260/afghan\_trail](https://github.com/hord-brayden/CIT-360/tree/master/14%20Topics/Afghan-Trail-master/Afghan_Trail/src/byui/cit260/afghan_trail)

MVC is a good process to adopt. There are a couple of other design ideas, but as I continue to implement it, I find that it works pretty well.

One of the principles I like, is decoupling of the View. This is because the View (what the user sees and interacts with) can change so frequently. The view could be a web page, an android or iphone interface, or a windows universal app. All of them simply need to communicate with the controller.

The controller does all the heavy middle man stuff. Long long time ago, I used to do all the work on the same page or form that the user interfaced with. When that format changed, it was a huge problem.

Now I can simply ask for an object, the controller contacts the model, the model might retrieve it from the database (or wherever) then passes it back to the controller and the controller serves it up to the view.

Rick

MVC is a schema of organization. It's just separating different features and functions into the model, view, and controller folders respectively.

Here's my link to my MVC, [https://github.com/hord-brayden/CIT-360/tree/master/14%20Topics/Afghan-Trail-master/Afghan\_Trail/src/byui/cit260/afghan\_trail (Links to an external site.)Links to an external site.](https://github.com/hord-brayden/CIT-360/tree/master/14%20Topics/Afghan-Trail-master/Afghan_Trail/src/byui/cit260/afghan_trail)

Each one of the three serves a functional purpose. Here's a VERY SIMPLIFIED version of what they do!

**The View** is what the user sees, **the Controller** handles input, and **the Model** contains the background data being used and calculations.

I personally only started using it when working with a team, because it keeps everyone on the same page with organization and contributions.

<https://alvinalexander.com/uml/uml-model-view-controller-mvc-diagram>

I am still struggling. In several of the posts I find autonomy between the MVC components. The controller is loosely coupled to the models and views. However, in others I see where they have a strong relationship. What I mean is, I have seen some where they create extensions and observers of the classes (loose) but in others I see where instances are created (In the view there is an instance of controller, in the controller there is an instance of model and view, in the model there is an instance of controller and view and in main there is an instance of all)...

Is there a definitive way to handle this? I am struggling with my class creations...

Model has my object - student with several attributes (Name and Age)  
View has my prompts for the student's name and age  
Controller has called the view to prompt, calls the model to process data and calls view to display results

But... how do I return the students name and age (multiple values) without an object? How can I return these multiple values without having the controller as an instance then using getters and setters from view to talk back to the controller?

[ReplyReply to Comment](https://byui.instructure.com/courses/36268/discussion_topics/411064?module_item_id=2442376)

[Collapse Subdiscussion](https://byui.instructure.com/courses/36268/discussion_topics/411064?module_item_id=2442376)[Drazen Lucic](https://byui.instructure.com/courses/36268/users/14826)

## [Drazen Lucic](https://byui.instructure.com/courses/36268/users/14826)

1:36pmJan 19 at 1:36pm

[Manage Discussion Entry](https://byui.instructure.com/courses/36268/discussion_topics/411064?module_item_id=2442376)

Shane, I was pondering the same problems and so far concluded the following:

* + Model, view, and controller should be separated but they also must exchange data. This is unavoidable.
  + Creating instances of classes in one layer and storing them in another layer seems to me like a tight coupling. However, perhaps we can use generic collections as data carriers. Map<String, Object> looks like a good candidate, something like JSON but in Java code.
  + The entire system somehow must be initialized. This can be done in main, that is, there is nothing wrong for main to keep some references and render the first (or home) view.

The MVC pattern works well for web applications. However, let us not forget that web applications need an infrastructure to run. There is a client, which is usually a web browser of some sort. Also, there is a web server and a database. I recall from previous courses that the client sends requests (GET, POST and others) to the server and receives responses, which is usually a  view rendered as a web page. Perhaps we should implement some equivalent of a browser and server. The browser will create a request object with data and a command, and send it to the server object. The server object will check the command and route the request to the appropriate controller (in this case, the server is actually a server-side router). The controller will "talk" to the model, do its part, create a response object, return it to the server, which will then return it to the client. The response object will contain the name of the next view. The client-side router will use the next view's name to fetch and render the view.

In summary, besides MVC classes, we must also have some utility classes such as Request, Response, ClientRouter, ServerRouter, and a class that contains the main method to initialize all those components and start the application. The data between layers can be exchanged by using collections (e.g. Map<String, Object>) or [POJO (Links to an external site.)Links to an external site.](https://en.wikipedia.org/wiki/Plain_old_Java_object)s or both (probably both).