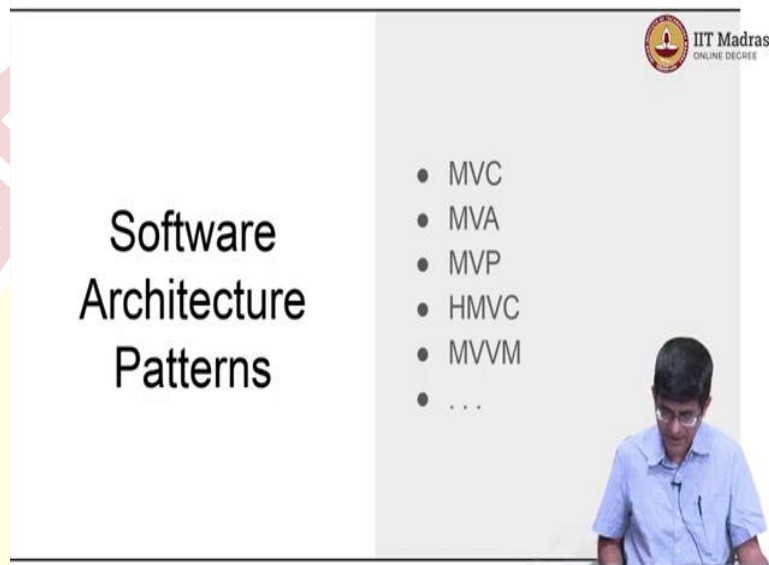


**IIT Madras**  
ONLINE DEGREE

**Modern Application Development – 1**  
**Professor Nithin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**  
**Software Architectures**

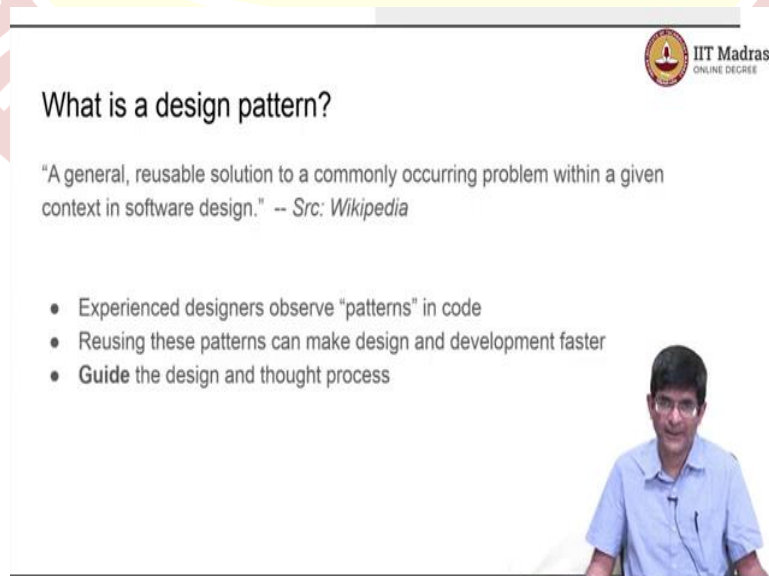
Hello, everyone, and welcome to this course on Modern Application Development.

(Refer Slide Time: 0:16)



So, now, that was about the architecture of systems. Now, we would like to look a little bit more at software architecture patterns. And as you can see on the screen, this is going to be a flood of TLAs, three letter acronyms.

(Refer Slide Time: 0:33)

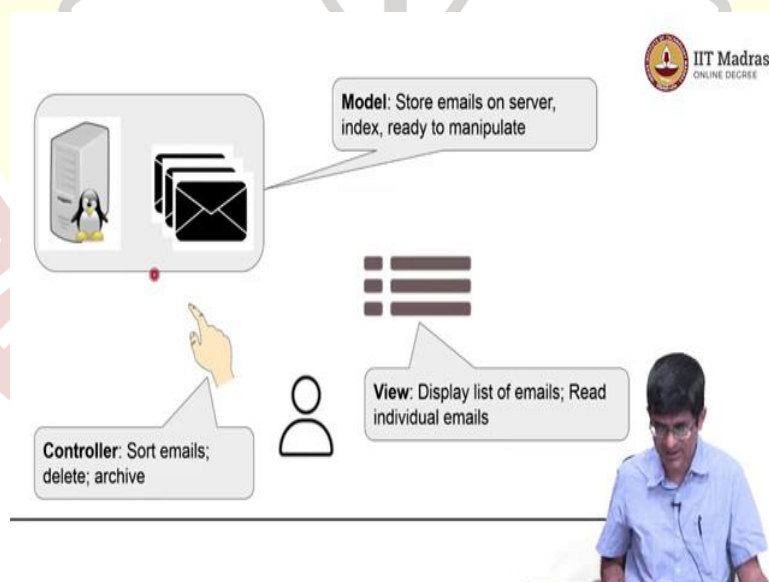


So, before we get into software architecture patterns, I want to briefly look at this concept, what is a design pattern, because the idea of a design pattern is something that you need to at least be familiar with in the context of software engineering or software design. And the basic idea is very simple. It is a general reusable solution to a commonly occurring problem. And usually within a given context in software design.

So, once again, this is just the definition as seen on Wikipedia. But what does that mean? It basically, means that people have been writing programs for a long time, there are some designers who are highly experienced, they have observed certain kinds of patterns in the way they write code that may make it useful for other newcomers people who are new to the system to absorb and start using.

So, what they say is, if you reuse those patterns, you can probably save a lot of time, simply because they have been found to be good patterns by people with a lot of experience. Does that mean that fundamentally these are the only way of doing things? No, of course not. These are just good ways. They are thought of known good ways to do certain things. They guide your design, and also the process in which you think, and that is the primary purpose of a design pattern.

(Refer Slide Time: 1:56)



So, let us look at this with an example. Let us say, I am a user who wants to check email. What would be the steps that I follow over here? I find that there is a server that basically has my email, and can help me to read the email. Now, how do I go about actually requesting this? I have two parts of information. One is the server now needs to also store the emails.

And these emails are stored in a form that is indexed and ready to manipulate. So, both of these together essentially constitute the so called model of data.


What is the data that I am dealing with my actual emails, but the model is a little bit more than that it contains the email, as well as information about the email, the so called metadata. So, what is metadata? It basically says, who sent the email? To whom was it sent? When was it sent? How big is it? Does it have attachments? All of that is metadata about the email. The server probably indexes all of those to make its life easier later on.

Along with that, you also have the view. And the view essentially says, it is based on the user. The user basically wants to view something. So they say, show me a list of emails. And, typically, what that would mean is, I would send a request to the server saying, show me a list of emails, the server would need to pull out the all emails addressed to me, sort them in some order, and finally display it, send it back to my device for display.

Now, that is, this right hand side is one part of the story. It basically says how I can use the model to view it. Then comes the question, what do I do with those emails, I might want to sort them or I might want to delete certain emails, archive some of them, sort them into folders, things of that sort, that is another interface back from the user back into the model, or the server. And this part is what we call the controller.

So, if you think about it, pretty much any kind of application that you can think of can finally be broken down into something, at least resembling this may not be exactly the same, but to a large extent does resemble this.

(Refer Slide Time: 4:27)




### M-V-C paradigm

- **Model:**
  - Core data to be stored for the application
  - Databases; indexing for easy searching, manipulation;
- **View:**
  - User-facing side of application
  - Interfaces for finding information, manipulating
- **Controller:**
  - "Business logic" - how to manipulate data

User  
uses **controller** ->  
to manipulate **model** ->  
that updates **view** ->  
that user sees

Hardly new: origins in Smalltalk language: 1979



And because of that, this MVC or the model view controller is considered a very good paradigm to understand how applications are built. Nowadays, of course, if you go search on Google, you will find a whole bunch of articles basically trashing the MVC paradigm saying that, it is not good for this. It is not good for that. It is a bad way of looking at things. You have to take that with a pinch of salt.

The way that you need to really think about any paradigm, not just MVC is that what is it telling you? It is basically telling you about how to think about the problem. Is it saying this is the only way to solve the problem? That cannot be it, that cannot be right. And that is the approach that we are going to take in this course, I am going to use the MVC paradigm to a large extent, simply because it helps to clear up the thought process of how to go about designing an application.

Is it necessarily the best way of doing things, is it the only way of doing things? Certainly not. Is it the best? In some cases, yes. In others, maybe not. Maybe there are better ways of looking at the problem. Why is it useful? Because it helps to separate out three sort of fundamental concepts over here, the first is the model which is the core data that needs to be stored for the application, emails and the indexes.

And it also says that this can be done using databases. What kind of database should I use? Should I use a relational database like MySQL or SQL lite? Should I use a no SQL system? Should I use just plain CSV files, various other alternatives are available. And you can just focus on that while looking at the model.

Then comes the part that faces the user. The view. And the view basically says, what should the user be looking at finally? And it also provides for the interfaces? If I have a keyboard, how should I interact with the view? If I have a touchscreen, how should I interact with the view? If I have only a voice interface, how should I interact with the view? Think, one thing to keep in mind over here is a view need not necessarily be something that you see.

Even let us say that you are dealing with a completely audio based interface, a hands free interface that you are designing for your car, you speak to the device, it then maybe reads out your SMS messages. That is the view. And your interaction with it is also by speaking. So, there is no touch, there is no visual input at any point, but we still call it a view. And finally comes the control of how to manipulate the data.

So I can, for example, send an instruction to the server telling it put this email into this folder, or delete this email or archive it. All of those are controller actions, they are telling the system, how to actually manipulate the data. Now, this MVC paradigm is not new. It has its origins in the Smalltalk programming language, which is from 1979 or so.

But you will find that in a lot of cases, I mean, there are a lot of the core underlying concepts are here, in many of the technologies that we use today, were developed quite a while ago, that does not necessarily mean that people are just sitting around reinventing the wheel, it means that you have now found a new context in which to apply a certain idea where it makes more sense than perhaps it did several years ago.

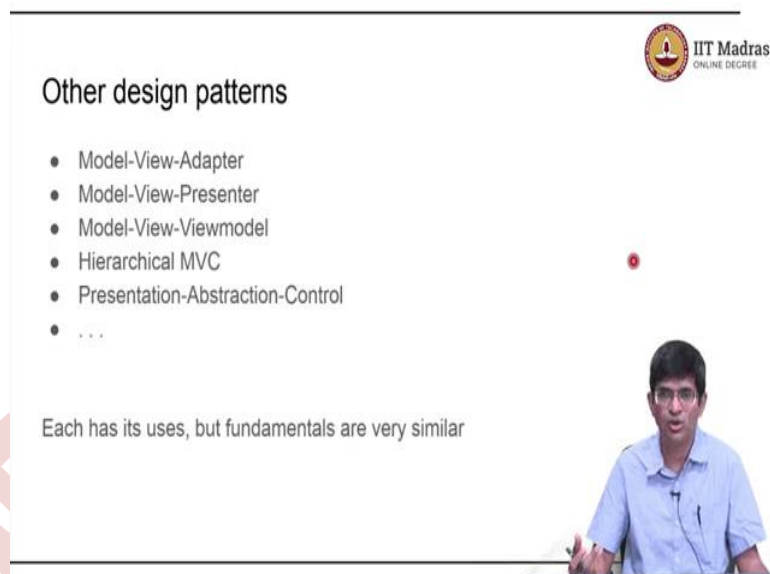
So, even though MVC was developed in 1979, for a long time, people did not think in that way while developing applications, because of the rest of the technology. But once it got to a point where applications developed into a certain level of complexity, you had to make that break and say, let me start separating these things out.

So, one way to look at the MVC paradigm is what is shown on the right hand side over here. Basically, the user uses the controller to manipulate the model, which then updates the view that once again, the user sees, so it is sort of a closed feedback loop that we have over here.

Every time I want to make a change of some form, I use the controller to manipulate the model, the view cannot manipulate the model, the view comes from the model back to the user. And the information from the user goes back to the model through the controller, that sort of separation of functionality helps to clean up a lot of designs. And that is the main purpose for which we will be sort of using this model and trying to understand it in context.



(Refer Slide Time: 8:57)



The slide is titled "Other design patterns" and features a list of design patterns. In the top right corner, there is a logo for "IIT Madras ONLINE DEGREE". A small red dot is visible on the right side of the slide. In the bottom right corner, there is a video feed of a man in a light blue shirt speaking. The slide content is as follows:

### Other design patterns

- Model-View-Adapter
- Model-View-Presenter
- Model-View-Viewmodel
- Hierarchical MVC
- Presentation-Abstraction-Control
- ...


Each has its uses, but fundamentals are very similar

There are a number of other design patterns, you just go search for MVC alternatives and a whole bunch of things pop up. So, this Model View Adapter, Model View Presenter. I, for the most part, we will not be looking into it, many different models moving forward. Each of them has its own uses. At some level, the fundamentals are very similar. The point is hierarchical MVC, for example allows you to build up one further level of abstraction that makes it easier to understand certain problems.

The Model View presenter has a slightly different sort of division of what goes into the view versus what goes into the controller, which in some cases can make the way that you partition your code a little bit easier to understand. But at the end of the day, it also depends on your programming language. It depends on your platform. It depends on the way you think.

For the purposes of our course, we are just going to use the simple MVC model. We will point to other things where necessary or where they are found to be useful. But keep that in mind MVC is not the only one. It is not necessarily the best one. But it is a good enough alternative that it makes sense to understand it clearly.


(Refer Slide Time: 10:10)



### Focus of this course

- Platform:
  - Web-based
- Architecture:
  - Client-server
- Software architecture:
  - Model-View-Controller

How to build apps (or applications) that are web-based with central servers, use hypertext markup to control and manipulate display



So, to summarize everything that we have looked at so far, the focus of this course is going to be to use the web as the platform for building applications or apps. The architecture that we will be looking at will be the client server architecture, we are not going to be looking at distributed applications in this case, although, once you have the basic ideas on how to develop a client server based system, you will realize that it is not that much of a jump to a distributed system.

In some sense, one way of looking at it would be that you are combining clients and servers on the same piece of software, and then having them interact with, each other each software could now be divided into a client portion and a server portion. Anyway, for us, the abstraction we are going to stick to is there is a client and there is a server, how do we communicate between them.

And the software architecture will be the model view controller. So, the focus of this course to summarize will be how to build apps that are web based, rely on central servers, and use hypertext markup to control and manipulate the display of information on your screen.