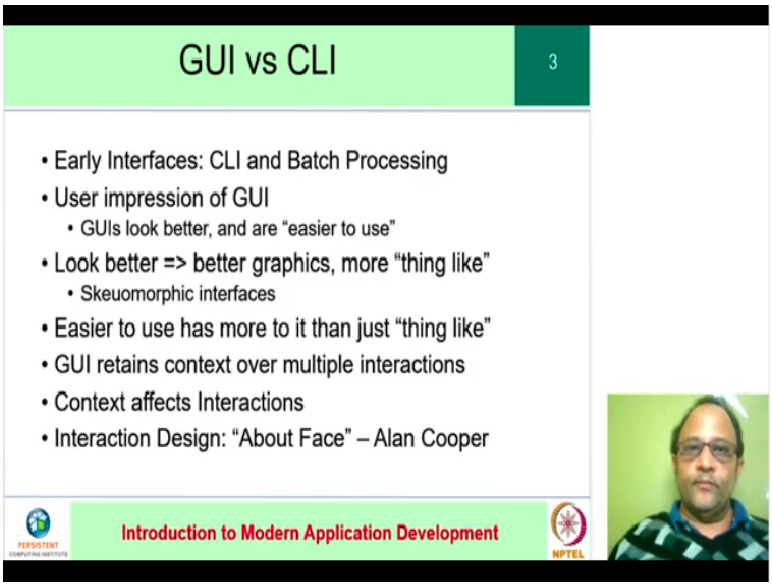
Introduction to Modern Application Development

Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Abhijat Vichare
Persistent Computing Institute
Madhavan Mukund
Chennai Mathematical Institute

Lecture-10
Comparing CLI, GUI, and Web Interfaces

Hello, welcome to modern application development. In this week, we will study the transition from command line interfaces to graphical user interfaces. And as part of this transition, we will learn lessons about what kinds of new ideas need to be involved when we transition from our basic batch processing command line applications to interactive applications with graphical user interfaces, and later on those lessons will translate over to the world of web application development.

(Refer Slide Time: 00:48)



Once upon a time everybody used command line interfaces because that was the only technology that was around. As our technology progress we invented the better idea of a graphical user interface. At this point, if you had asked our users exactly in what way our graphical user interface

is better, then they might have said something like how will they look better, and that they are easier to use.

But as programmers, we need to unpack these statements to understand precisely what it is that makes graphical user interfaces better. And what it is that makes graphical interfaces easier to use. The *"look better"* part to some extent comes from the sheer beauty of the graphics, especially in our day, where graphical interfaces have become extraordinarily elegant and aesthetically satisfying. But there is another side to it.

Besides, the looks being pure and simple, graphical user interfaces are more thing like they feel closer to our real-life experience in using the things that are around us. For example, buttons are a classic example of something that translates very nicely from the real world into the user interface. Ever since the days of electric light bulbs, buttons have been ubiquitous, and so they are very natural for us to use.

The buttons have seemed natural before we were used to electrical systems does not seem necessarily true to me. This idea is sometimes is called **skeuomorphism,** *the idea that you are mimicking things in the real world and using them as interfaces to things that are not real on the computer*. So that is one sense in which graphical user interfaces are easier. But there is more to it than even just the looks or the similarity to real world things.

What we find is that the nature of interaction, the very experience that the user has of what happens when they use graphical interfaces is different from that of using a command line interface. For one thing with GUIs we can interact multiple times with a program without having to start and stop that program all the time. And this means that the app can now retain context because it learns from history, it remembers what was going on, and you can use this information to improve user interaction.

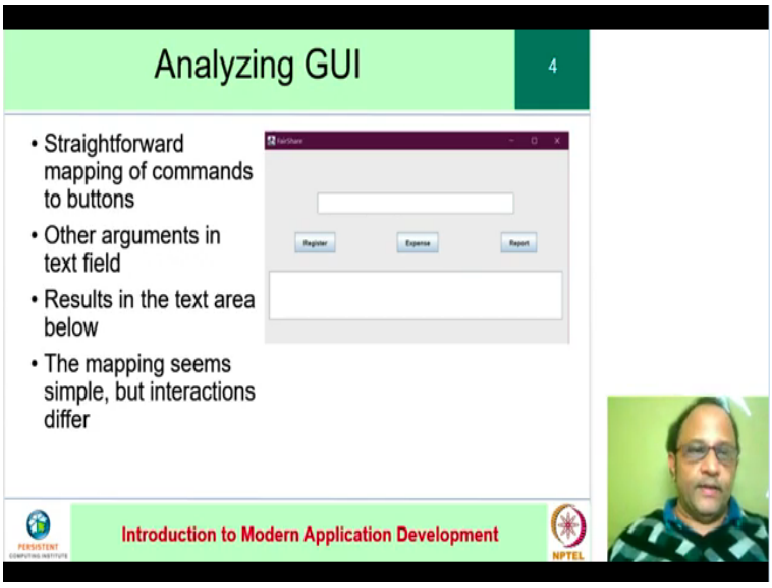
We will see a simple example of how managing the historical context makes things easier, starting with a simple version of our app. But the general, more real-life, version of the simple example that we will see is comes under a topic called interaction design. This is not just about how an

application looks but how is it that a user can find the application natural to use without too much interaction.

A famous book called ***About Face by Alan Cooper*** first made the idea of interaction design popular, the first edition of this book in the mid 90s, and maybe that book talked about desktop apps and the latest edition in around 2014 now covers, for example, things like mobile applications. Our example will be very simple. But when it comes to complex applications, it is worth keeping in mind the distinctions between various aspects of app design.

One aspect of app design is the internal software design of the program, which governs things like how easy it is for a programmer to understand and change application. Another different dimension is visual design, which looks at whether or not we like the looks of the app. This is the more artistic side of development. And a third kind of design is interaction design, which talks about how user experiences their interactions with your application. Now let us go on to our simple example.

(Refer Slide Time: 05:02)



Here we have a screenshot of a straightforward way to map our simple command line application to a graphical format. Let us consider what the features of such app might be. For example, like

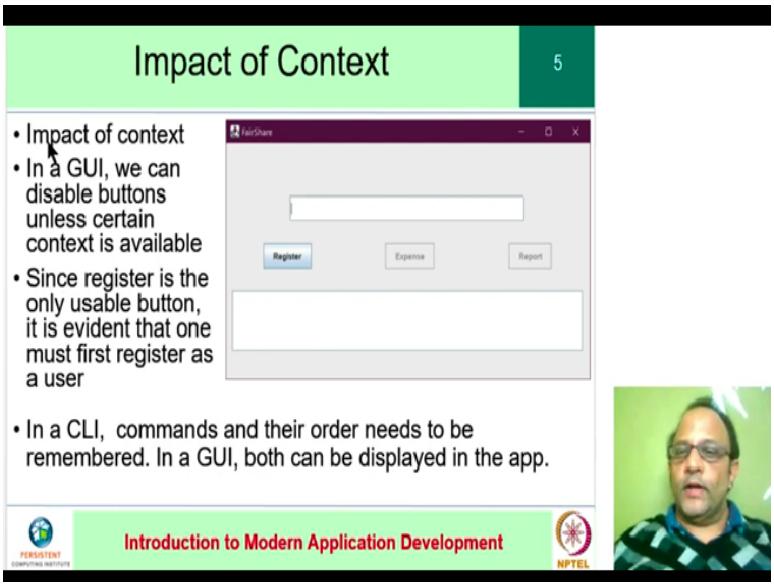
the CLI, we can register a user. And it seems clear that the way we register should be to type the name of the user in the top text will hit the register button, and the user should get registered. Obviously, the user name goes into the text field.

Similarly, it seems clear that we should enter expenses into the text field and then hit the expense button. The lower text area is where we will get responses such as what just happened and when you press the report button, that is where we will see the output. How many of you would agree that it seems natural that the roles of the text field and the larger text area are obvious. Is not it interesting that too many of us, even non technical users will be able to make this case about what the app is doing, and therefore, give us a pretty easy way for app to be able to be become usable, without ever having to read a manual, or to instruct people in how to do these things.

These idioms of how our user interface should be organized or by now so natural to us, that they feel obvious. And there is even more that we would consider obvious. For example, we might expect the app to remember who the current user is, and not require us to again and again specify the user which is something we have to do in case of the command line application, for example. And even our non technical friends, if we show them this app and they have to enter their name again and again, they will definitely make a suggestion that would not your app, just remember our name. Another way we could have designed the app is to just simply copy whatever it is our command line app did, and require the user to typing the entire command into the text field. But that would not feel as different as an app which remembers the context.

So, in this sense, this appearance of context, which persists across multiple interactions is what is unique about graphical user interfaces, as compared to textual user interfaces offered by command line applications.

(Refer Slide Time: 07:42)



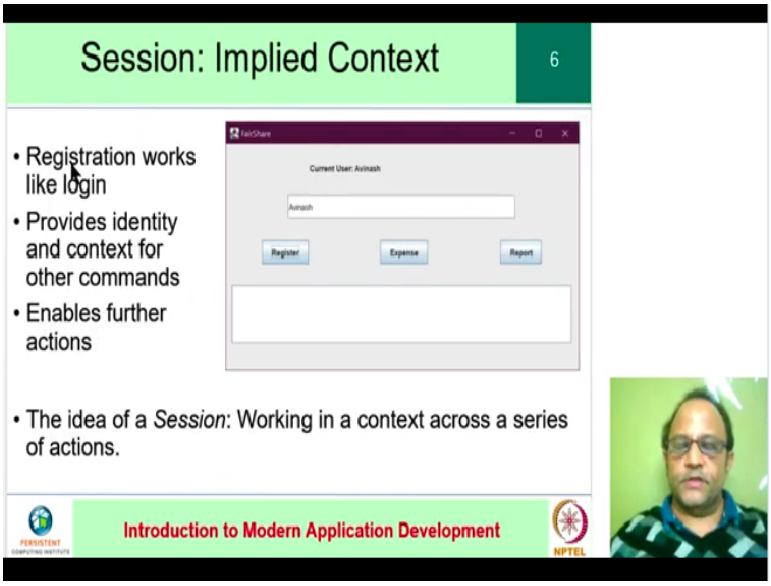
Now let us see what happens when we start using context in an even more, shall we say, inventive or energetic way. As an example of what can happen, in the simple app that we consider all 3 buttons of the app were possible to be pressed as soon as the app open. But that is not actually very useful because until user has been registered, it is not as if you can do anything with the other 2 buttons. But this knowledge tells us that we should actually have some sort of way to tell the user that certain buttons are of no use right now. And we do understand how to do such a thing. So, the usual idea here is to gray out certain buttons, if currently they do not have any use. This graying out and features like this switch start becoming context sensitive, or one of the key sources of complexity of graphical user interfaces. For instance, in the command line app, all command line strings are considered valid when enter.

And we will announce that there is an error if it so happens that the user enters some sort of illegal string. Now, the question is what shall we do in the graphical user interface case. One answer is as we have seen: graying out of the buttons. But decisions like this are not really clear. Some interaction designers might feel that the error message style is in fact superior. Because if you press a button that is not supposed to be used, then the programmer could offer a nice error message that clearly explains why this button is not available right now, and what the user might do instead.

In this way, context changes many of the possibilities of interaction. As history gets deeper, and the amount of features the application offers become more complicated, the interaction between what has happened so far and therefore what can happen the future becomes harder to understand. Unfortunate or fortunately as users, rather than as programmers, it turns out that small nuances like this are really appreciated by users.

And therefore, it turns out that many interaction designers, and even us experienced as users of other applications, will say that it is a good idea to have such context sensitive features. The conditions in the app core however, as we add these things start ballooning, and it requires a good dialogue between interaction designers and programmers to figure out what is viable and what is sustainable over the long term.

(Refer Slide Time: 10:38)



Reliance on the context to talk about what an app should and should not do or can and cannot do, naturally leads to the idea of a session. A session begins when an app is launched. There are a series of interactions and then the app is terminated. ***A session maintains context in memory*** especially for desktop tools. And it is quite natural to be able to easily know what is going on in an app in which such information can be kept directly in memory.

The most common sort of context, for example is user identity. In our app registration works sort of like login, and we can tell the current user, after they register, that the rest of the operations will happen in the context of their identity. As shown in the screenshot, our application can arrange easily to remember the registered identity. It is important to show this to users because among other things it may be possible that multiple users might use this app.

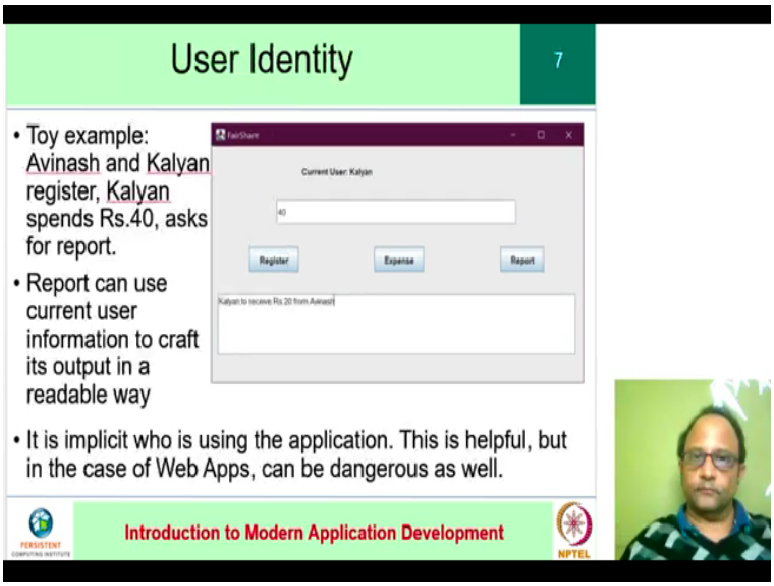
And even if that does not happen in very common context, it is also important to signal to the users that some aspect of the previous interaction, and exactly what aspect, will be reflected in the application. Because we are so familiar with sessions, the idea of a session might seem to be simple. And of course, as users it is a simple idea. However, for an application implementer, sessions are not that easy to implement.

The reason is that the idea of a session, and the increasing amount of different contexts that could exist, start adding more and more dimensions to the internal details of a program. For example, suppose that in addition to the identity, we acquire a notion of the current project that an application has to deal with, once there is a current project now for every single data structure in the program, here to think about for which identity and for which project.

Soon the cases start multiplying and then some features are allowed for some identity, some are allowed for some projects, some are for some combinations, and some are disallowed. And when these kinds of options multiply, the complexity keeps on increasing. For example, think of the distinction between the administrator of a server versus the user of a service. These 2 identities, even if they are the same person at different times, can have radically different consequences.

And so, the internal logic of the application must cope with these kinds of variations. But particular property that is commonly associated with sessions adds further complexity. And this is unfortunately, the very common property of identity.

(Refer Slide Time: 13:41)



Here is a small example of how we might use identity. Suppose there are 2 users Avinash and Kalyan and then Kalyan spends 40 rupees and then asks for a report. Report can use the information about the current user to craft its output in a nice readable way. It is implicit who is using the application, and this is helpful. But as we will see, in the case of the web, this can make things pretty difficult.

Most of us are so used to logging into our machines, and we take it for granted that our identity will be implicit in all of our interactions. We are also used to thinking of logging in, logging out of shared machines as the very definition of sessions. But identity is hard even for desktop applications. As an example, once upon a time, personal computers were thought to be truly personal, the way phones are, and so they were designed with one person anyway.

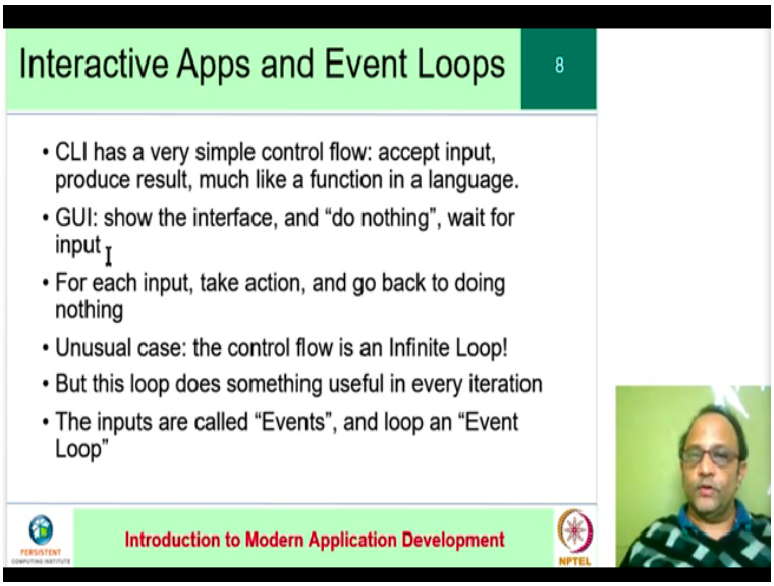
But it happened that family started sharing them and institutions started offering shared PCs. And now the idea of a personal computer broke down. The transition from a single user to multiple user required deep changes to the system. For example, we had to add the idea of owner to every single file system. And so, something that was used only for storage without thinking much about who it is for all of a sudden had to deal with the dimension of identity.

If identity is hard for desktop apps, it is even harder for web applications. Because identity in the case of web causes issues such as impersonation of users and now you have to deal with not merely the identity as used internally to the app to distinguish among things, but also to deal with validating user IDs, determining possible fakes and things like that. In the case of local CLI and GUI apps since our access is protected by logging on a physical machine.

The distinction I am making about may not seem important, but once you become a designer of applications, it turns out that many websites are under constant attack by various people who would like to get in and use your facilities for all sorts of strange things, including such things as mining bit coins on your servers. In the case of the web, impersonation becomes a very serious issue. A decade or so ago, it was possible to impersonate another user merely by sharing correctly crafted URLs.

Nowadays, most websites have defenses against such things, but even if you are careful you may open up your site to attacks inadvertently, and we will spend some time trying to understand the security ramifications when they arise. For now, let us move on to the other aspects of interactive applications that we need to understand. The key difference besides session and identity is the notion of an event loop, which changes the nature of control flow that is available to an application.

(Refer Slide Time: 17:02)



A command line application has a very simple control flow, accept input and produce the result. This is much like how functions work in a programming language. In contrast, the control flow of an interactive GUI application is more confusing. There is nothing like it in usual programming languages that we can rely on for some experience. The default behavior of GUI when it started is quite different from the default behavior of a command line app.

Whereas a command line app is immediately doing exactly what you told it to and nothing more, the default behavior of the GUI when it started is to show the interface and then do nothing except wait for our input. When an input is received, it executes the command and goes back to doing nothing. In general, surprisingly on our machines, the CPU the disk etc. are idle most of the time, unless we are programs such as clients that check whether you have received an email or WhatsApp message or something like that, other than that machine has nothing to do except to do whatever it is that the user wants.

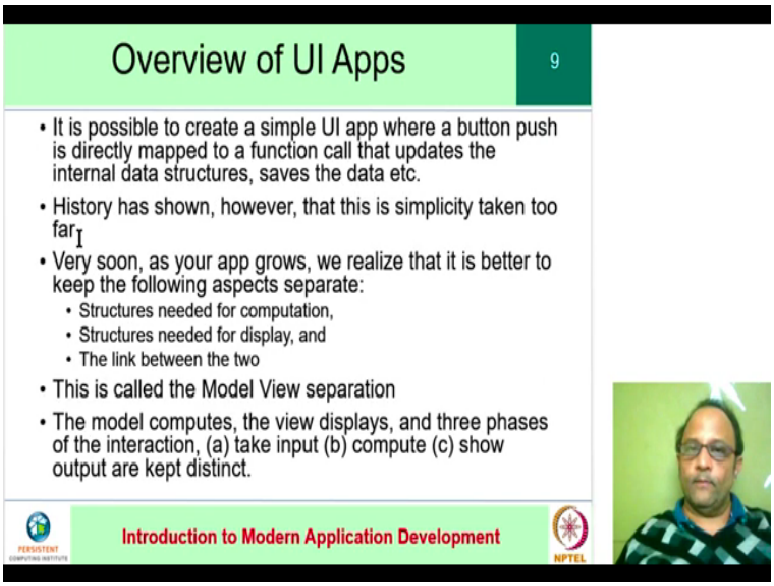
So such a waiting do is surprisingly enough an infinite loop, this is one of the rare cases where an infinite loop is exactly what you want, normally and an infinite loop in normal programming, such as a mistake, but in this case, it is exactly what you want indeed, the program should loop forever,

just wait for user input, and every time it receives input, do something useful, such inputs that are received by this infinite waiting loop are known as events. And so, the loop is called an event. At the lowest the event loop or the events that loop might be interested in are things like mouse movements and keystrokes and GUI it will get together with operating systems determines whether the mouse that is clicked in a particular part of the screen should be interpreted as having triggered a button.

So, for example, it converts effectively the lower level mouse pressed event into a higher-level button pressing, and GUI and web programs are written in terms of such high-level events. So, not only is the control flow unusual, we need a new kind of terminology to deal with this, which is the notions of events and event loops and responding to events. An event loop in a GUI program is relatively simple to understand, and so we will spend some time talking about it. But event loops in web programs can exist on the server, or the client the browser, or both. Furthermore, event loops in the web systems can simultaneously serve multiple users, unlike the usual GUI event loops that serve only one user.

In the following slides, we will see how sessions events, and GUI code looks like in our app. We will then build on this knowledge to study similar constructions in web applications.

(Refer Slide Time: 20:24)



Let us start with the overview of the basic structure of user interface application, it is possible to create a simple UI app where the push of a button is directly mapped to a function call. And this function call like any other function call in a normal program would go around updating various data structures. Various simple UI building tools do take this approach because this approach is easier to get started with.

However, experience has shown that while it is easy to get started with a program, where UI controls like buttons and text fields directly trigger functions, such apps become unmanageable as new requirements arise. What happens is that the program for every button is thinking in terms of that button, and not in terms of the big picture. If a different programmer later wants to understand what is going on, some features might arise accidentally, as a result of strange interactions between different input and output elements of a program.

And you really need more organization to try to make sense of what is actually going on in the user interface application. This happens very quickly, even for simple seeming apps. So, it is worthwhile to take a more structured approach for building a user interface application. This solution, or a solution to this problem of structuring, is something that most user interface programs

will have to deal with at one point or another. And the solution is simple enough that it is better to use it right from the start. As we will see, it can be used in a very lightweight way and does not have to be so complicated at all.

What we realized historically, was that this sort of separation needs the following 2 elements. We structured the application into mostly independent parts, usually, at least 2 parts and often 3 or even more parts.

One way to separate out the complexities of what an application was do versus how to present that information is to consider that there are 3 aspects. There are structures that are needed for computation. This part which is responsible for computation is called a model. Another part called a view, manages all the code direct to create the user interface and so becomes responsible for display.

And there has to be a third part which connects the model and the view, so that we know which element of the view affects which elements of the model and we can connect them together. This component is sort of like a method of wiring which connects a button to a lamp. If there are multiple lamps and multiple buttons than it is the wiring that decides which button affects which lamp in this metaphor.

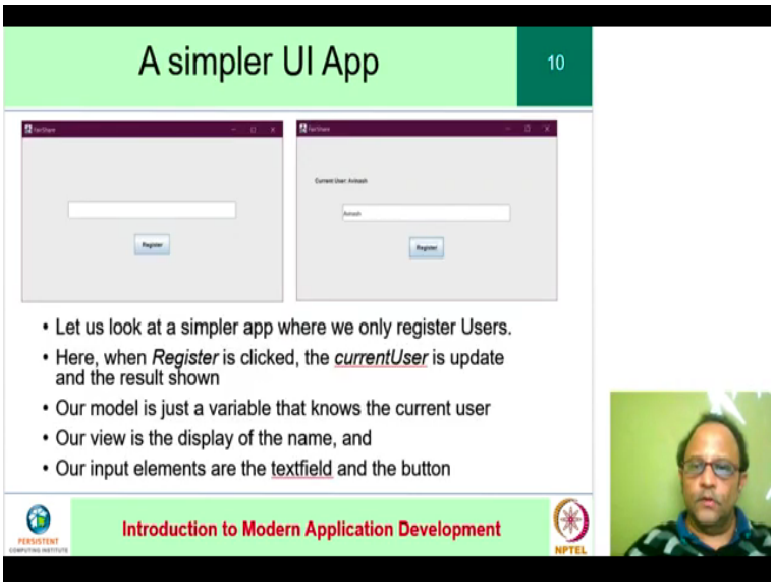
Sometimes this wiring is programmed explicitly and in other styles called reactive programming and many other kinds of things. The wiring is inferred from the structure of the program. For our purposes, we will remain with a simple explicit wiring style of programming. So, once these 3 parts are in place, what is happening? The idea is that the user of an application who is mostly interested in the display side, and not in the internal structuring, sees the view and interacts with it. But the data that the view shows comes from the underlying model. Effectively, the view offers a way for a user to manipulate the underlying model. So, that is about the static structure.
What about the dynamic case, what about the behavior? For that behavior, we also organize into a sequence of 3-phases:

1. In the first phase a user works with the view and the view accepts user input,
2. it then determines which part of the model should be affected by the current input,

3. and then the wiring conveys the meaning of the user gesture, eventually to the user via the view.

Once the model knows what the user wants, the computation occurs in different parts of the modern data updated, then changes to the view computed based on what changes occur in the model. This 3-phase way of thinking helps us clearly organize the program. To understand this, we will start off with an even simpler version of the application we already have, and see how the model restructuring works.

(Refer Slide Time: 25:35)



This slide shows a simple user application and even simpler one, which simply registers users and does nothing else. Here when the register button is clicked, the current user is updated and the result is show. Our model here is just a simple variable that knows the current user. So:

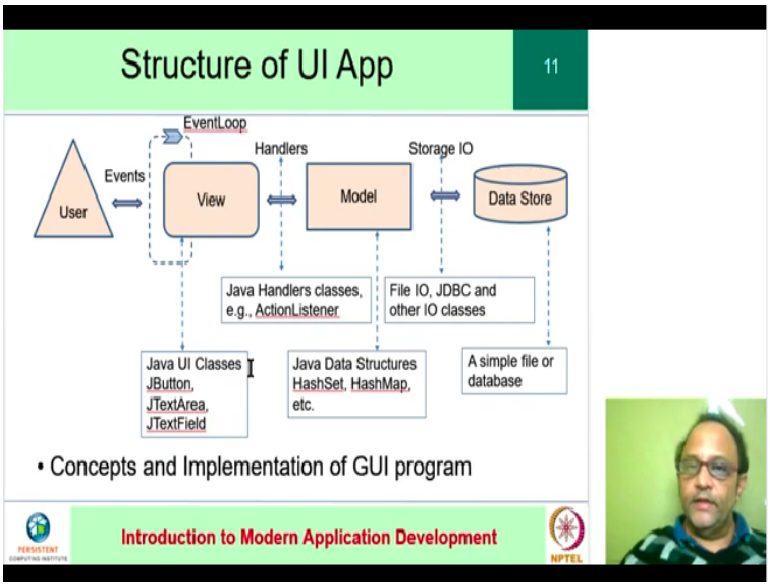
- In the first image that you see, the application is how you see it when you launch it.
- And the second is after you register a single user, you can see the current user reflected in the label that we have shown.

Here as I said, the model is actually a very simple thing. It is merely whatever it is that holds the name of the current user. The view consists of the button, the text field and the label. And now 2 connections exist between the view: the text field and the button and another part of the view, which is the label.
What sorts of connections are these? the text field and the button act as input devices that changed the currently held value of the model. And the label acts as a device that shows you what the current value of the model is.

So, one pair of these 2 elements, the TextField and the button are acting as input systems and the label is acting as output system. The 3-phase flow in this case is particularly simple. In the input phase, the user fills the textfield and presses the button. In the computing phase, the model stores the current value of the textfield, because that is all the computation that is needed in this case.

And then there is nothing much going on other than that, the output has to be shown whenever the model changes. So, this is the core example. Even in something as simple as this, this discussion of the distinction between the view and the model and the 3-phase input output system, does it help a little bit to understand how one might go around structuring even the simple UI program like this. Okay, now on to something more complicated, where we will take a look at how these ideas work out when we look at greater details.

(Refer Slide Time: 28:11)



So, let us take a look at the conceptual structure of what is happening in the case of a model view application. In this slide, we will study how the code is actually organized. So, if you take a look, you will see that at the top there is the triangle represent in the user. There is a rounded rectangle which represents the view and around this view there is a looping construct called the event loop as we talked about, the view in turn interacts with the model via elements called handlers, and the model interacts with the data store via storage IO.

The user interacts with the event loop via events, and the event loop organizes the user interaction and provides them one after another to the view. So, when we built our actual user interface, we have elements in the program that correspond directly to these conceptual blocks. Those elements of the program have been listed below. And since we are building this in Java, most of the elements refer to the Java classes that are used to build these systems.

So, corresponding to the view, we have Java UI classes, such a JButtton, JTextArea and JTextField corresponding to the handlers, the Java handler classes such as ActionListener are used to connect the view to the model. The model for the kinds of systems that we are looking at are simply data structures like HashSet and HashMap, and of course any other classes that you might wish to design and use for computation.

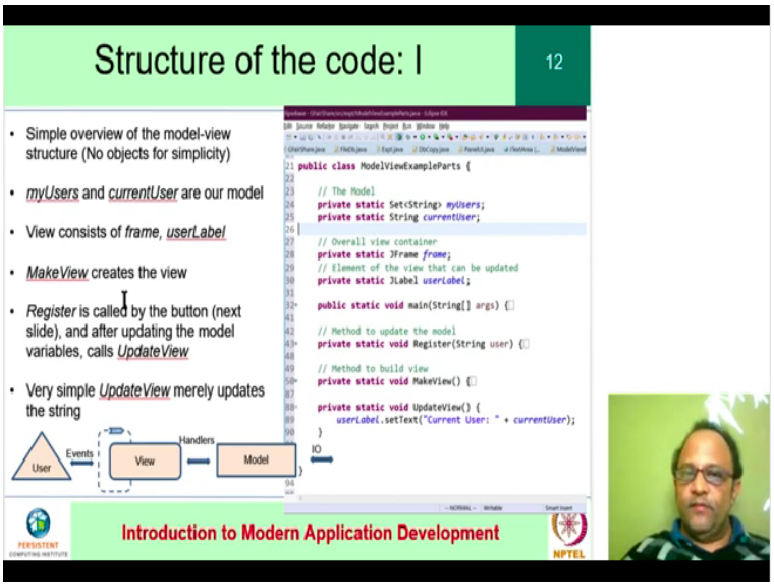
The storage IO part is handled by file IO, JDBC and various other IO classes, some of which you have already encountered in writing the command line application. And the data store for our use case anyway, is just a simple database in file. Somehow, then all the real code we write has to fit into this kind of a model to have the structure that we would like, so that we can understand what the dependencies are and what – as the application grows, how to manage the distinction between what is needed for viewing and what is needed for modeling.

Let us see how the 3-phase of the input computation output loop fit into this model. So, in this case, what we have is that there is the user who types in something, and when the user does something, it kicks off the event loop, which then drives the ActionListener. Then this is the part where we have accepted the input.

At this point the handlers get triggered, and the model gets caught. This is the compute part. And once the computation of the new state of the model is complete, in turn, the model tells the view that there are some aspects of the view that the view objects should be updated. And this is the part the where the control flow lays out on top of the data structures that we have seen. In the rest of this, we would not talk much about the data store, it will just be there in the background.

Because there is nothing new that is going on here compared with the command line application. Next, let us take a look at the code which corresponds to the conceptual structures that we have seen.

(Refer Slide Time: 31:56)



So, let us see. Okay, here we go okay. So, slide number 12. In this slide, we are looking at parts of the code that were written to create the application that we have already seen. In this thing, as you look to the right, I am using Eclipse, but there is no particular reason why I mean I am not making any recommendation about what IDE you should use, use whatever you are comfortable with. Here, we see those parts of the application that correspond to the conceptual things that we have seen.

And as you can see the elements that are used to create the model and the view are very, very simple. We do not even have much by way of objects, of course, things like Sets, and so forth, and there are Java objects in the libraries. But this design is not particularly object oriented. The reason was that I did not want any extra structure. I just wanted to show that even simple structures can map very clearly into the model view structure that we have already seen.

For ease of reference, I have replicated the model view diagram on the bottom left of this slide. So, let us do a quick walkthrough of the code that we see here. So, the class is called `ModelViewExampleParts`, and it begins by showing you the model. The model in this case consists of really 2 variables, one of which is a set, and the other one, which is a string. `myUsers` is all the users that have registered, and a string `currentUser` is the user that we will actually show in the label.

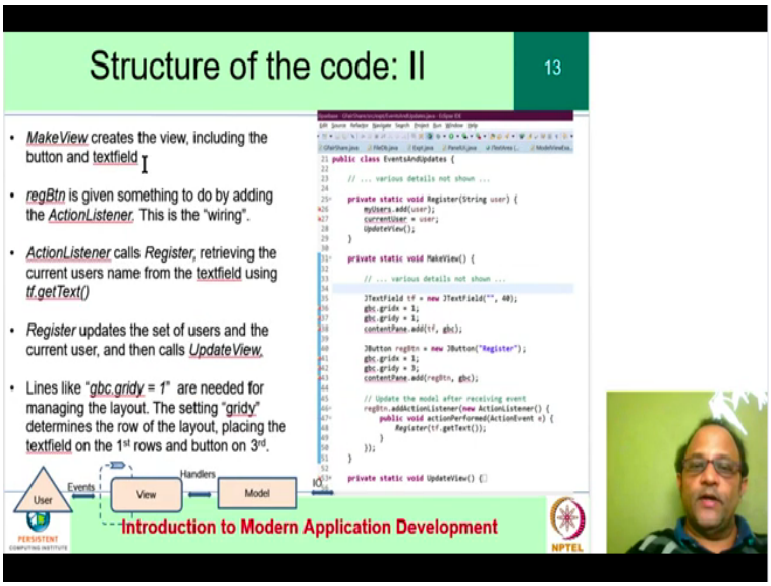
Below that is the overall view contain, it is a frame, which is one of the Java standard style of organizing GUI in Java code. And the element of the view that we are interested in is of course, just a JLabel. All of this is driven the usual sort of main program. And then you have the following 3 methods, which constitute the elements that lead to the 3-phase control flow that we have discussed.

First of all, let us start with the method `MakeView`. `MakeView` is the method that lays out the structure of the view, and makes it ready for presentation to the user. When the user does something, it is `MakeView`. It is components created in make view that will receive the event and enter the trigger the method `Register`, which goes and changes the model, and `Register` in turn triggers `UpdateView`, which then goes and updates the element.

As you can see, update here is pretty simple. It just says user label dot set text and change the current user to whatever the user now is supposed to be. So, it is not the case that just because you have ideas called model and view, they should necessarily correspond to objects called model and view, you just need clarity in what you use as models and what you use as views. Of course, as applications grow more complicated, it does become more natural to use objects to represent these kinds of things, and say, be able to manage and control the structure in a better way than just writing a bunch of static functions that I have done here. But for purposes of illustration, this is good enough. This is not a course particularly on object-oriented programming. And I am going to assume that should the need arise you will be able to study if required, and perhaps you have already seen object-oriented programming enough that you can do a decent job with it.

So now that we have seen the overall structure of the code, and we understand how the 3-phase execution cycle maps, namely, you have the view, then from the view, the model gets called and in the inter the model causes updates to the view. Once we have this picture in mind, let us take a look at how the wiring part of the code works.

(Refer Slide Time: 36:18)



So here I have extracted the parts that are needed in yet another class just to have something isolated to talk about, but they are the exactly the same methods that we have seen before. As the comments say, there are many details here that have said nothing about and so let us take a look at what is left. So first we said that `MakeView` creates the view. If you look at the structure of `MakeView` you will see that it creates a new JTextField and a new JButton.

And there are elements like setting up a grid and a content pane and a bunch of other things that are needed via the Java libraries, reg button is the registration button, this button is given something to do by adding a class called ActionListener. And this part constitutes the wiring of the view to the model. When ActionListener is invoked by the event loop, then ActionListener calls register, and register gets information about what the name should be, from the textfield, this is expected by the ActionListener, it is not as if register calls this particular TextField directly.

In fact, it does not even necessarily know that such a thing exists. It only knows that it gets a username. The process is simply as follows:

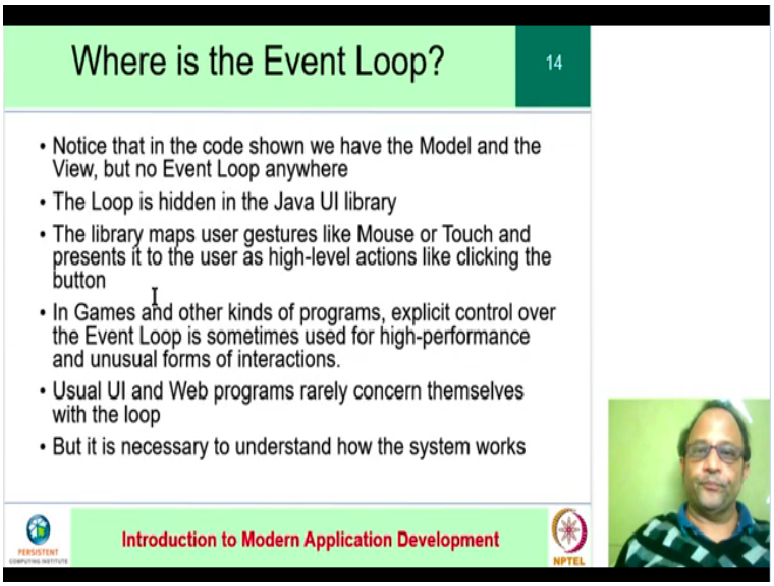
- When Register gets a username, it adds the user to the collection and to the currentUser.
- And once that is done, it updates the view by calling UpdateView.

Lines like `gbc.gridwire` in the code are needed for managing the layout. The setting grid wire determines the role of the layout, placing the textfield on the first row and the button on the third row. But there is no particular reason why this particular layout has to be done this way, it is just something that I picked.

In general, doing layouts in desktop programs is a harder problem than doing it for the web, especially nowadays. So, when it comes time to replicate something like this on the web, at some level for the layout part will actually have an easier time of it. When we look at more complex programs, it is essentially the same style, just more elaborate constructions.

And as you can imagine, the ideas of model view organization are pretty simple. As long as you have clear in your head that the separation exists and whether or not it is evident in the code directly, at least you should have a plan in mind and clearly separate the wiring and have a 3-phase control flow cycle, that is the crucial part alright.

(Refer Slide Time: 39:19)



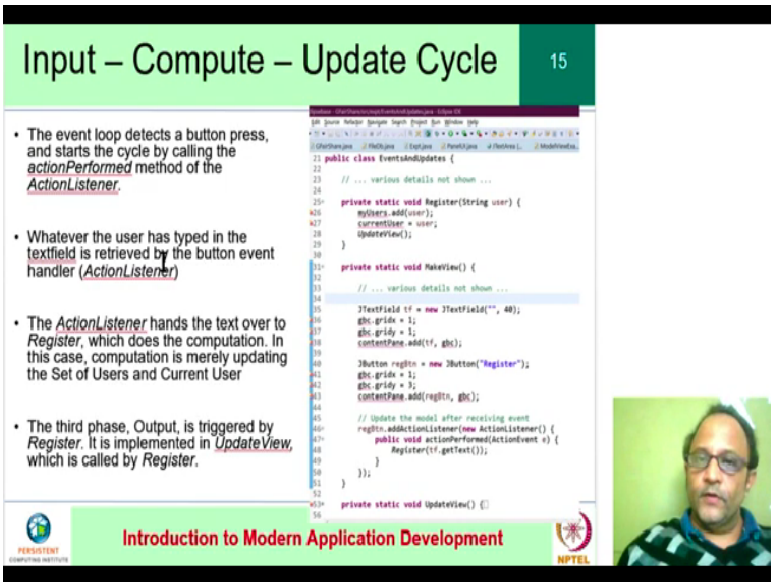
Now in all this you would notice that the event loop is nowhere to be seen. So, what is going on, although we are shown it in the model view diagram. So, what is going on is that the event loop is actually hidden by the Java UI libraries. The event loop is a sort of a 2-level structure. At one level,

the lowest level, the event loop interacts directly with the raw input devices such as keyboards and mouse. Then when it knows where the mouse is, it, with the help of the operating system and under tool kit, does some calculations to figure out if a mouse clicked in a particular place on the screen corresponds to a button press. And if it is then it triggers the button pressed event. So, this is as if the raw event loop is converting low level events into high level events that are suitable for our application. And indeed, we can pretend that the low-level event loop in fact does not exist at all as far as the application is concerned.

Rather, there is a high-level event loop which talks in application specific events. Now, event loops are usually not of much concern in desktop applications, and even in web applications because they are part of the infrastructure unless you are writing games or some other special purpose programs. Explicit control over the event loop is rarely needed.

However, to understand what is triggering the system and what is going on, you need a clear idea of the event loop, which is why we spend so much time on it.

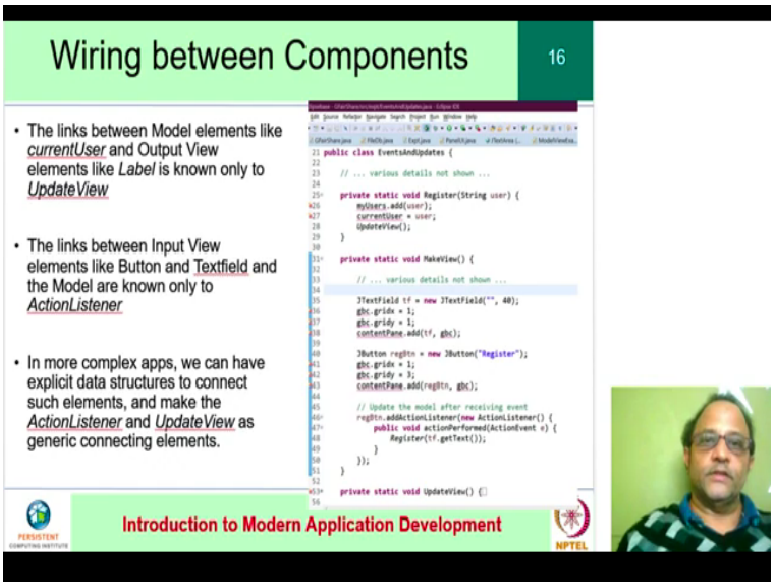
(Refer Slide Time: 41:13)



So, after this, what we are going to look for is to understand better how the *input - compute - update cycle* works. To make it clear the connection to the event loop, I am going to repeat this information again. So, as we can see in reference to the code, the event loop detects a button press starts the cycle by calling the action performed method on the ActionListener. Whatever the user has typed is then retrieved by the button handler.

And in fact, it is the only one who knows where to retrieve it from, it then hands over the text to the register, because as far as the application is concerned, it does not care how you get the username, only that you get it somehow. And then it does the computation which is updating the set of users and the current user. The third phase output is triggered by the register function. And it then is actually implemented by the `UpdateView`.

(Refer Slide Time: 42:22)

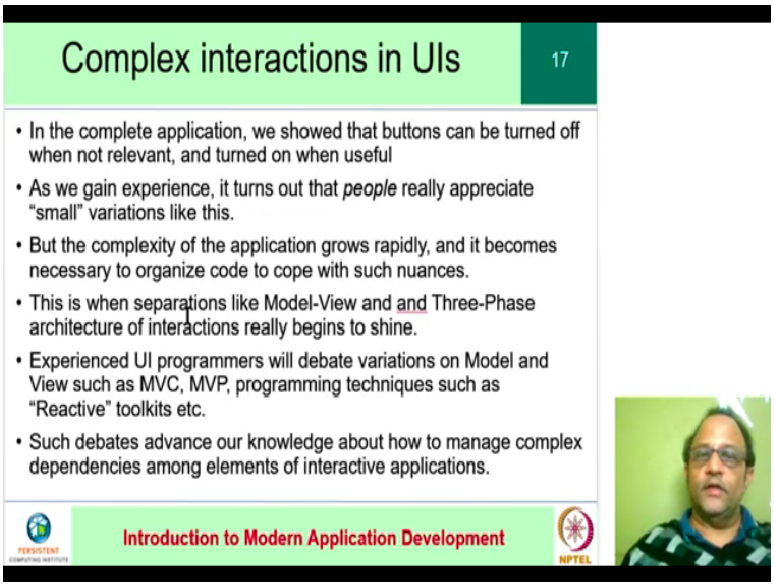


By arranging this system in this particular style, what we have ensure is that the knowledge in the application is clearly spread into manageable pieces. So, for example, looking at the wiring what we see is that the links between model elements like current user and output view elements like label are known only to the `UpdateView`. Nowhere else in application that is the knowledge of exactly which part of the view is connected with which part of the model.

The output portions of the view are known only to update, whereas the input portions of the view are known to the ActionListener. In this simple way we have separated 2 parts of the input-output loop cycle. In more complicated applications, we can have explicit data structures to connect such elements and turn ActionListener and `UpdateView` into general purpose systems for connecting various elements and managing their dependencies.

In a simple application like this model view separation doesn't seem that necessary and you could perhaps get things working without it if you really wanted to. But once complex interactions are involved, then the model view separation really starts mattering.

(Refer Slide Time: 43:40)

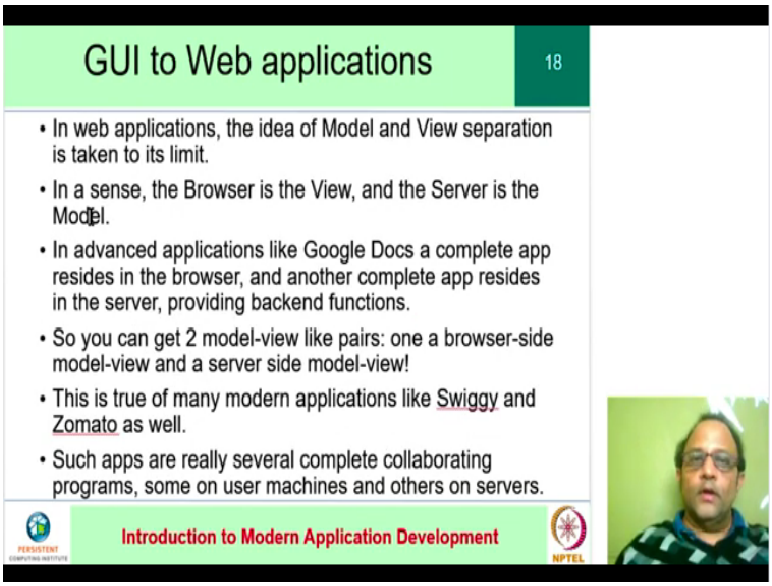


For example, in the complete application, we show that buttons can be turned off when they are not relevant and turn on when they are again useful. As we gain experience, it turns out that people really appreciate small variations like this, but then the complexity of the application because quite rapidly and it becomes essential to organize code in a clear way so that you can cope with such sort of nuances.

At that point model view in the 3-phase architecture of interactions really begins to shine. As you gain experience, you will find out that programmers will often debate variations on model view, such as MVC, MVP programming techniques like reactive toolkits and so on. Most of these come up because of the complexity of applications and such debates and inventions that go along with them advance our knowledge of how to manage ever more complicated applications as simply as we possibly could.

Okay, so at this point, we have taken a good long look at desktop applications. Now let us start making a connection with web applications.

(Refer Slide Time: 44:56)



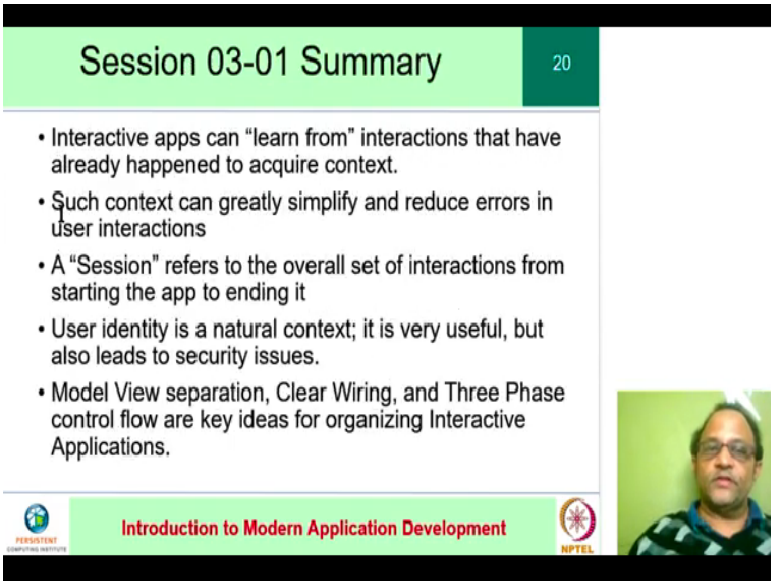
So, the first thing to notice is that as we go from GUI applications to web applications, the idea of a model view separation, in some sense gets taken to its limit. In one sense, the browser is actually the view component, and the server is the model component. In really complex and advanced applications, like Google Docs, for complete application resides inside the browser, and another complete application resides inside the server providing backend functions.

So now what happens is you get to model-view like pairs, the one pair sitting in the browser, and another pair sitting in the server. This is true of many modern applications, for instance, Swiggy and Zomato most likely have a similar architecture. This is because these applications in some sense are actually multiple collaborating programs, some of which happened to run on user machines, and some of which happened to run on server.

Now our goal here is to not produce something that complicated, we are looking at a very simple web application in which the browser will play the role of view and the server will play the role of model, more or less. These 2 will interact over the network, which in many ways is very similar to making simple function calls like register. But it is not quite that simple. It also needs a lot more dealing with differences that arise because of the separation across the network.

And because of the fact that the state does not readily reside inside memory, the way to reside inside desktop applications. But the key lessons of the model will separation will carry over to the server browser separation. And this is the link between GUI applications and web applications okay.

(Refer Slide Time: 47:02)



Now, let us summarize what we have seen so far:

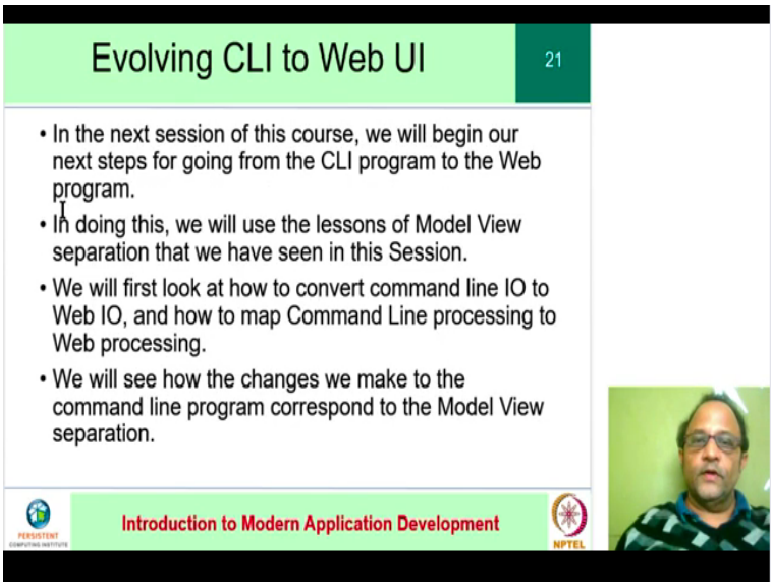
- Interactive applications can learn from interactions that have already happened in order to acquire context. Such context can greatly simplify and reduce errors in user interactions. Those two are the key elements in understanding how our interactive application differs from a command line application.

- The next thing is that we have introduced the idea of a session, which refers to the overall set of interactions from starting the app to ending it. But not just that, it also refers to the information that is collected and the context that is created during the session. So, we will use this single world in some sense to stand both for the information and for the actual working of the session from the beginning to the end. Of all the context that gets created,

a natural context is user identity. It is very helpful, but it also leads to various issues. So those are the key ways in which the behavior of interactive applications differs from command line applications.

- Lastly, we talked about the change in the internal architecture of GUI applications, where we have studied the principles of model view separation of clear wiring between different parts, and 3-phase control flow to organize the interactive application control flow.

(Refer Slide Time: 48:40)



In the next part we will start looking at how to go from a command line application to the web UI. In doing this we will use the lessons of model view separation which we have seen. Our first step will be to convert command line IO to Web IO, and to see how to map command line processing to web process. We will make changes to our command line program to correspond to the model new separation. And in doing so get closer to creating a full web application.

Thank you. See you in the next session.