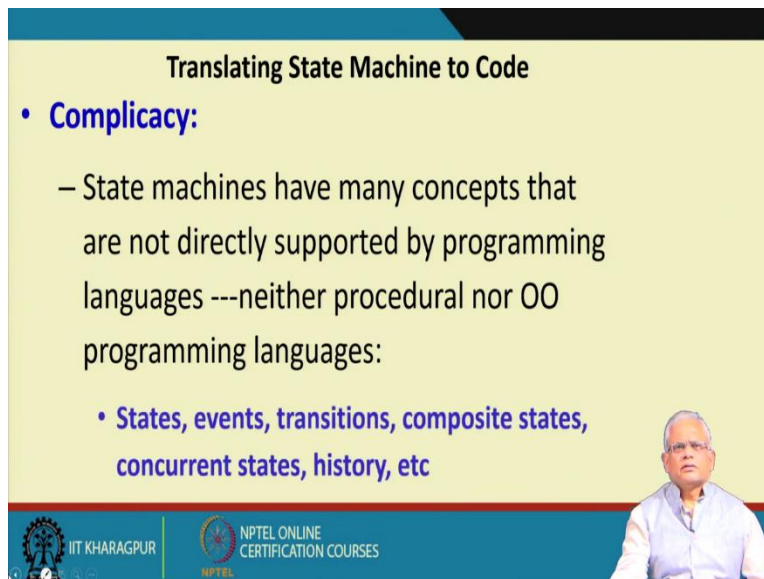


Object- Oriented System Development Using UML, Java and Patterns
Professor. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 24
Encoding a State Machine - II

Welcome to this lecture.


In the last lecture we had just started to look at once we develop the State Machine for a class, how do we mechanically or automatically generate code from the state model? And we had just about started that, let me just start from the basic concepts in generating code from a state machine diagram.



(Refer Slide Time: 00:48)



Translating State Machine to Code

- **Complicacy:**
 - State machines have many concepts that are not directly supported by programming languages ---neither procedural nor OO programming languages:
 - **States, events, transitions, composite states, concurrent states, history, etc**



 IIT KHARAGPUR  NPTEL ONLINE CERTIFICATION COURSES

The main problem with generating code on a state machine diagram is that the concepts of the state machine like the states, the transitions, the girds, starting state, end state, and so on. These are not really supported by a procedural or object-oriented language. It would have been really nice, if we had language constructs available where we can specify the states easily, the transitions, and so on. And then, it should work just like we had in the inheritance: the extends just to worked it derived class is implemented just to the key word.

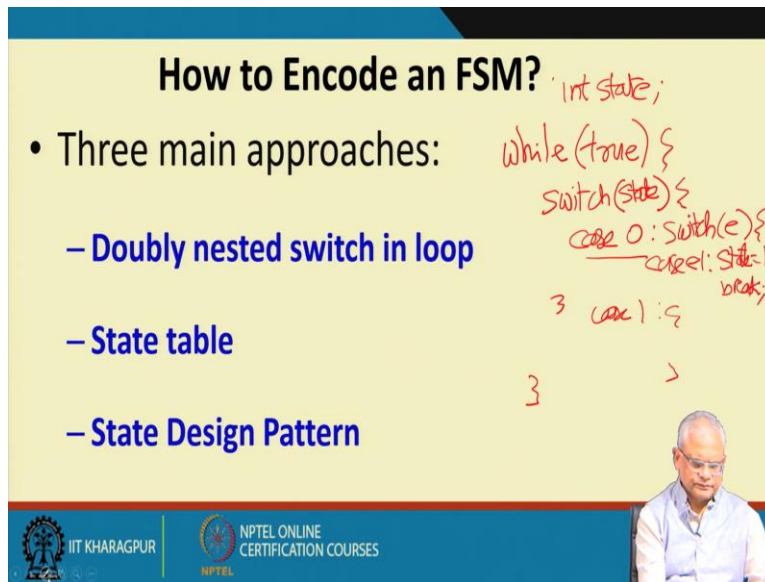
But unfortunately, for state machine implementation in object-oriented code we do not have any simple way, we have to basically understand how to generate the code and we will be able to see that it is not really very difficult.

(Refer Slide Time: 02:10)

How to Encode an FSM?

- Three main approaches:
 - Doubly nested switch in loop
 - State table
 - State Design Pattern

```
int state;  
while (true) {  
    switch (state) {  
        case 0: switch (e) {  
            case e1: state = 1;  
            break;  
        }  
        case 1: {  
        }  
    }  
}
```



Let see the approaches to encode a state machine. Of the available approaches three are most prominent: one is the doubly nested switch in a loop, we have state table, and a state design pattern. The design pattern, we will just postpone till we start discussing about the design patterns and we will see that state design pattern is a prominent design pattern. We will discuss it later.

Right now, we will look at the doubly nested switch in loop and state table approaches. The first one as it says, doubly nested switch in a loop. So here, we will have a loop (while(TRUE)) or something, and then we will have the first switch which is switch based under state, and we will have a state variable (the state encoded as a variable int state). There can be different states, let say there are four states and we encode them as zero, one, two, and three.

And then we will say, case0 which is state0, we will have a switch event which occurred. And let say if event e1 occurs, then it transits to state-1. We will say case e1, the state = s1, so transited to the new state the state variable from zero it has changed to one. And we can write and break here, and similarly the other events that occur in the state we can write state change for them. And for the other states, case1, case2 and so on we can write. So, this is our simple encoding the doubly nested switch in a loop. And then the next is the state table, we will just have a look at that.

(Refer Slide Time: 05:39)

The slide is titled "Three Principal Ways" in a yellow box on the right. It lists three methods for implementing state machines:

- **Doubly nested switch in loop:**
 - Use scalar variables to store state --- Used as switch discriminator for first level switch
 - Event type is argument to second nested switch
 - Harder to handle concurrent states, composite state, history, etc.
- **State table:** Straightforward table lookup
- **Design Pattern:**
 - States are represented by classes
 - Transitions represented as methods in classes

The slide also features the IIT Kharagpur logo and NPTEL Online Certification Courses logo at the bottom left, and a small portrait of a man in a white shirt at the bottom right.

Now, the doubly nested switch we use a scalar variable to store the state, the state is a scalar variable like int or something. And then we use this as the discriminator of the switch, first level switch and then depending on the state, that is case S0, S1, etc. we write the second level switch where the argument to the second level switch is the event. But then, we will have to use ingenuity or cleverness in programming to handle concurrent states, composite states, history, etc.

The second approach is a state table, here in this approach we first develop a state table from the state machine diagram and the state table we store in a data structure. And then when an event occurs at a certain state, we index into the state table and then look up what state transition should occur, which new state it should go, and what action should take place in the process and that we implement the code.

State table is also straight forward, both these doubly nested switch in loop and state table are straight forward. And then the design pattern which we said that we will see little later once we start discussing about the design patterns. Here we will see that each state is represented as a class and the transitions are represented as methods in the class.

(Refer Slide Time: 07:50)

```
int state, event; /* state and event are variables */
while(TRUE){
    switch (state){ /* Wait for event */
        Case state1: switch(event){
            case event1: state=state2; ... break;
            case event2:...
            default:
        }
        Case state2: switch(event){...
    }
}
```

Doubly Nested Switch Approach

Handwritten notes:
while (true) {
switch (state) {
case S1 : switch (e) {
case e1 : state = S2;
break;
case S2 : switch (event) {

The diagram shows two circles representing states S1 and S2. An arrow labeled 'e1' points from S1 to S2. Another arrow labeled 'e2' points from S2 to S1. There are also arrows pointing to each state from below, labeled 'e1' and 'e2' respectively.

NPTEL ONLINE CERTIFICATION COURSES
IIT KHARAGPUR

```
int state, event; /* state and event are variables */
while(TRUE){
    switch (state){ /* Wait for event */
        Case state1: switch(event){
            case event1: state=state2; ... break;
            case event2:...
            default:
        }
        Case state2: switch(event){...
    }
}
```

Doubly Nested Switch Approach

Handwritten notes:
S2 : switch (event)

A red bracket highlights the inner switch statement within the 'Case state1' block. A green box highlights the code inside this inner switch: 'case event1: state=state2; ... break;', 'case event2:...', and 'default:'. A red arrow points from the 'Case state1' label to the inner switch.

NPTEL ONLINE CERTIFICATION COURSES
IIT KHARAGPUR

23

Let see examples of the doubly nested switch (in the above slide). The Doubly nested switch approach, here as I was saying that both state and event are integer (ints), or it can be any scalar variable like char or something. This is the first loop here (first slide of the above two slides), and then inside there is switch on state, that is what is the current state. And then it switches to the current state may be state S1 and then here we have another switch within first switch and the event. And then depending on the event which has actually occurred, let say event1 has occurred then we encode the transition that should occur. For event1, it should go to state2 (state = state2),

and then if any action to be taken we just write the action and then break, wait for the next event to occur and so on.

So, this fairly straight forward approach to translate a state machine to a C code. Just to try out let say we have, a state S1 and state S2 and there is a transition occurring here on event e1, transition occurring here on e2, and on e3 terminates, and starts with by default on S1 (as shown in the above slide hand drawing).

Now how do we encode this, of course we will have the while loop, while(TRUE), and then switch state: case S1, switch event, and case e1: state = S2, then break. Because that is the only event to which this state S1 response to. And state S2: there are two events to which it responds to one is e2 and another is e3. At case S2: we will write again switch event. And here case e2: we will set state to S1 (state = S1) and on case e3: exit. The hand written code shown on the above slide.


So, we can see it is very straightforward translation: given a state machine diagram how to translate it into C like code. We will see like how to translate it into a Java like code (2nd slide of the above two slide). So, here also inside while loop the first switch is the state and then the second switch is the event. Based on the specific state we switch in the event. So, it is also very straight forward translation.

(Refer Slide Time: 11:59)

State Table Approach

- From the state machine, we can set up a **state transition table...**

Present state	Event	Next state	Actions
S1	e1	S3	Open Valve
	e2	S2	set red LED flashing
S2	e1	S1	Close Valve
	e2	S4	reset red LED flashing



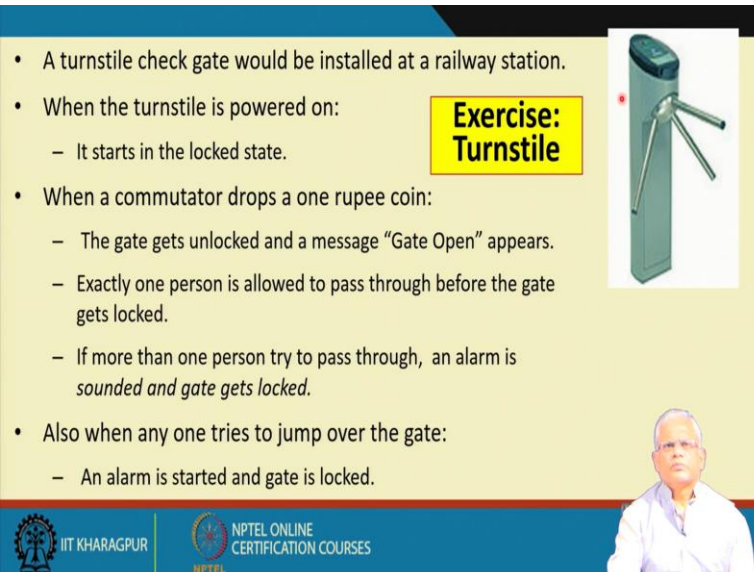
Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES is visible at the bottom of the slide.

Now let's look at the state table approach. In the state table approach, based on the state machine we generate this table. We can manually generate this table or it can be generated automatically based on an analysis of the table. Here the first column we will have the states here and then we look at what are the rest events to which it responds (this is the second column). The third column encodes that for specific event what's the state it transits to and then the corresponding action.

Let say we have state S1, which response to e1 and e2 and on event e1 it transits to S3 and then the action that takes place is open valve. And here, on an event we will first check the current state possibly, we will switch on the current state, and then index into the table for that state. We do not need a switch actually, based on the state we will index into the table.

And then for that specific state we check under second column, the specific event to which it responds. Here, with responds to e1, we set the state variable to S3 (state= S3) and call the method or the function to take the action which is 'open valve' here. So here, it is very simple code based on an event. First index into the table based on the current state, match the event here, and then for the event we set the state variable to the corresponding state here, and then call a method for the corresponding action.

(Refer Slide Time: 14:39)



The slide contains a list of events and actions for a turnstile system. The events are: 'When the turnstile is powered on:', 'When a commutator drops a one rupee coin:', and 'Also when any one tries to jump over the gate:'. The actions are: 'It starts in the locked state.', 'The gate gets unlocked and a message "Gate Open" appears.', 'Exactly one person is allowed to pass through before the gate gets locked.', 'If more than one person try to pass through, an alarm is sounded and gate gets locked.', and 'An alarm is started and gate is locked.' There is an image of a turnstile and a small portrait of a man in the bottom right corner.

- A turnstile check gate would be installed at a railway station.
- When the turnstile is powered on:
 - It starts in the locked state.
- When a commutator drops a one rupee coin:
 - The gate gets unlocked and a message "Gate Open" appears.
 - Exactly one person is allowed to pass through before the gate gets locked.
 - If more than one person try to pass through, an alarm is sounded and gate gets locked.
- Also when any one tries to jump over the gate:
 - An alarm is started and gate is locked.

Now let's try to do one simple exercise (in the above slide), develop its state model and also generate the code. This problem is about railway station in which we want to install a check gate

like this (in the above slide image). And here, once the check gate is powered on it is locked, nobody can enter the station. And a commutator needs to drop a one-rupee coin here, and once the one-rupee coin is dropped (passenger entry fee), the gates get unlocked and one person can pass through here. And then, again the gate gets locked and again the next person has to drop another one-rupee coin. But if more than one person tries to pass through, then the sensor checks that two persons are passing through and then alarm is sounded and the gate is locked. Also, when anybody tries to jump over the turnstile, then the sensor checks somebody jumping over the gate and alarm is sounded and the gate gets locked.

(Refer Slide Time: 16:19)

- When the alarm is ON:
 - A message **“Please wait: Gate temporarily blocked”** is displayed.
 - If any one still inserts a coin, it return the coin without unlocking the gate.
 - The alarm is reset when an attendant swipes a card and the gate starts at the locked state.
- When any one inserts a coin when the gate is already open:
 - The coin is returned.
- If there is a power failure any time:
 - The gate gets locked and **“Power fail: inoperative”** message is displayed and coins are not accepted.

Turnstile Exercise

Also, when the alarm is on a message gets displayed, “Please wait: Gate temporarily locked”. But then if anyone still inserts a coin, the coin is returned without unlocking the gate. The alarm can get reset when the attendant swipes the card and the gate start in the locked state for the passengers or the commutators to insert one-rupee coin and then the gate gets unlocked and so on. Also, another feature is that when the gate is already open and somebody inserts a coin, then the coin is returned.

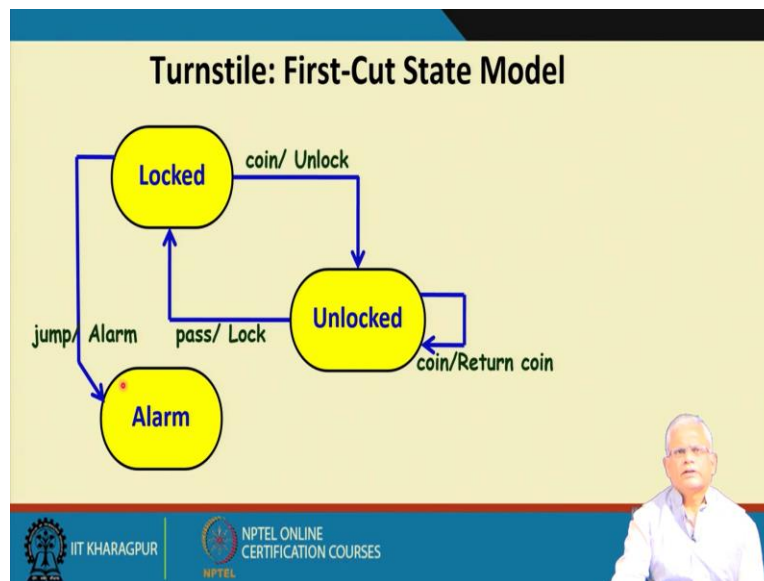
If there is a power failure at any time then the gate gets locked and power fail inoperative message is displayed and coins are not accepted. Here as we can see that the states of the turnstile is locked state, unlocked state, and the alarm state and we can identify the transitions that occur from lock to unlock. To start with it is in the locked state, and then as the passenger

drops a one-rupee coin it goes to unlocked and a person is allowed to pass as long as a person passes the gate again got locked.

In the locked state or in the unlocked state if somebody jumps over, then the alarm state is entered. The alarm state to locked state transition occurs when the card swipe occurs. And then we have the final feature here, that the power failure at any time: in this case the gate gets locked and “Power fail: inoperative” is displayed. That means from any of the states if there is a power failure it goes to the power fail state or the off state.

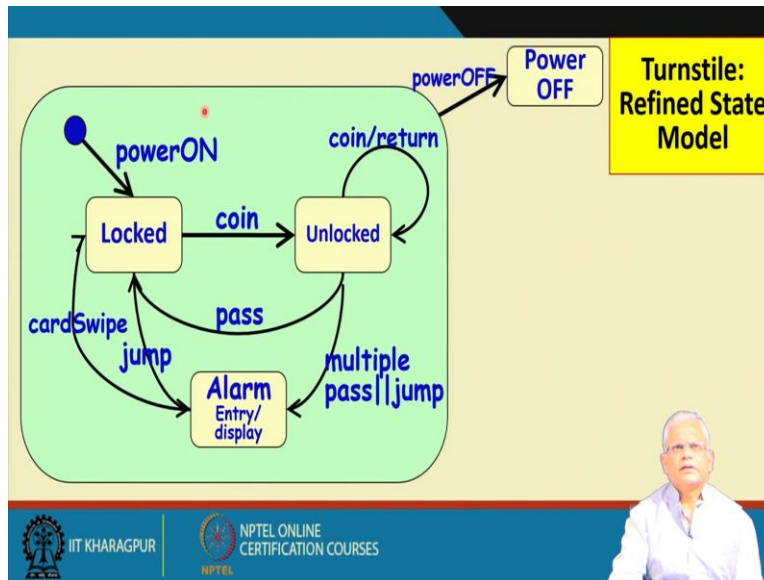
And we know that from any of the state gives us indication that this will be a or state, a composite or state and then there is a power fail from any state we indicate that with a transition from the composite state; that means that from all the state inside it. And this should be the diagram (in the below slide), and then we will generate the code from here.

(Refer Slide Time: 20:17)



So, we have the three states locked, unlocked, and alarm as we have drawn. If a coin is inserted then we call the unlock method which unlocks the gate, this is the action part and the gate gets unlocked. If anybody drops a coin when it is in unlocked then the coin is returned, the coin event occurs in unlocked then the coin is returned. If a person passes through then the turnstile gets locked, if somebody jumps in the locked state there is an alarm.

(Refer Slide Time: 21:02)



And we need to also draw the power off from any of the states goes to the power off state and power failure is displayed here (in the above slide).

Now our work is to write the C code for this first and then the java code. Here the states are three actually, in the turnstile operational state or power on state of the turnstile. And let's first write the turnstile power on.

(Refer Slide Time: 21:50)

```
enum State {Locked, Unlocked, Alarm, PowerOFF};
enum Event {Pass, Coin, multiplePass, jump, cardSwipe};
static State s = Locked;
void Transition(Event e){
    switch(s){
        case Locked:
            switch(e){
                case Coin:
                    s = Unlocked;
                    Unlock();
                    break;
                case Jump:
                    s=Alarm;
                    Alarm();
                    break;
            }
            break;
        case Unlocked:
            switch(e){
                case Pass:
                    s=Alarm;
                    Alarm();
                    break;
                case multiplePass:
                    s=Unlocked;
                    break;
                case jump:
                    s=Alarm;
                    Alarm();
                    break;
            }
            break;
        case Alarm:
            switch(e){
                case cardSwipe:
                    s=Unlocked;
                    Unlock();
                    break;
            }
            break;
        case PowerOFF:
            break;
    }
}
```

The number of states are four: locked, unlocked, alarm and power off. The events are pass, coin, multiple pass, jump, etc. So, use the enumerated type (enum Event {Pass, Coin, multiplePass,

jump, cardSwipe}, enum State(Locked, Unlocked, Alarm, PowerOFF)) and the state start with locked. This implements this transition here, the pseudo transition to the locked state, and then in this program whenever an event occurs the event handler calls the transition function with the event type as the parameter. Because an event typically occurs as an interrupt and this will be interrupt handler or the event handler.

And then we switch on state and then in case of locked state and the coin event occurs, then the state gets unlocked the state becomes unlocked (S = Unlocked), and we call the unlock method and break. Similarly, in the jump event the state becomes alarm (S= alarm), and the alarm is sounded by the alarm function, and then break, and so on.

(Refer Slide Time: 23:30)

```
enum State {Locked, Unlocked, Alarm, PowerOFF};
enum Event {Pass, Coin, multiplePass, jump, cardSwipe};
static State s = Locked;
void Transition(Event e){
    switch(s){
        case Locked:
            switch(e){
                case Coin:
                    s = Unlocked;
                    Unlock();
                    break;
                case Jump:
                    s=Alarm;
                    Alarm();
                    break;
            }
            break;
        cont...
```

The diagram illustrates a state machine with four states: Locked, Unlocked, Alarm, and Power OFF. The initial state is Locked. Transitions are as follows: Locked to Unlocked on 'coin'; Unlocked to Locked on 'coin/return'; Unlocked to Alarm on 'multiple pass' or 'jump'; Alarm to Alarm on 'pass'; Alarm to Power OFF on 'powerOFF'; and Power OFF to Power OFF on 'powerOFF'. There is also a 'cardSwipe' transition from Locked to Alarm.

C Code

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

We can write the other part also for the case unlocked, and so on. It is straight forward, you can just read through the code here (in the above slide), and the finally the case alarm.

(Refer Slide Time: 23:49)

```
public class Turnstile{
enum state{ Locked, Unlocked, Alarm}
public Turnstile(){ state=Locked;}
public pass(){
    if(state== Unlocked) state=Locked;}
public coin(){
    switch(state){
        case Locked: state=Unlocked; break;
        case Unlocked: returnCoin(); break;
    }
}
public jump(){...}
...}

```

Java Code

NPTEL ONLINE CERTIFICATION COURSES

The java code is also not very difficult (in the above slide). Here we have the class turnstile and the turnstile has locked, unlocked, and alarm as state. And then, the constructor of the turnstile sets the state to locked, that is to start with once we create the turnstile object. The state starts in locked and on the pass event the method pass gets called by the handler routine, and once the pass method is called then we set the state to be locked.

If the state is unlocked, then the state gets locked. In the case of coin, the coin can occur in multiple states like locked state and unlocked state, so we use a switch here or we can use an if then else and here if it is the locked state the state gets unlocked, and in the unlocked state return coin.

In the Java code for a state machine, we have the events appearing as methods and that's possibly the only different from C code. Here, we do not have a doubly nested switcher because each event becomes a method and inside this, we use a simple switch on the state and then for the specific event we check what are the state transition and the action part. Here, state is a class variable and, in the constructor, we take care of setting at the default state or the state at which it starts (public Turnstile () {state= Locked;}).

(Refer Slide Time: 26:02)



The slide features a yellow background with a blue header and footer. A yellow box in the top right corner contains the text "Assignment: Course Registration Software". The main content area contains a bulleted list of four points. The footer includes the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, along with a small portrait of a man in a white shirt.

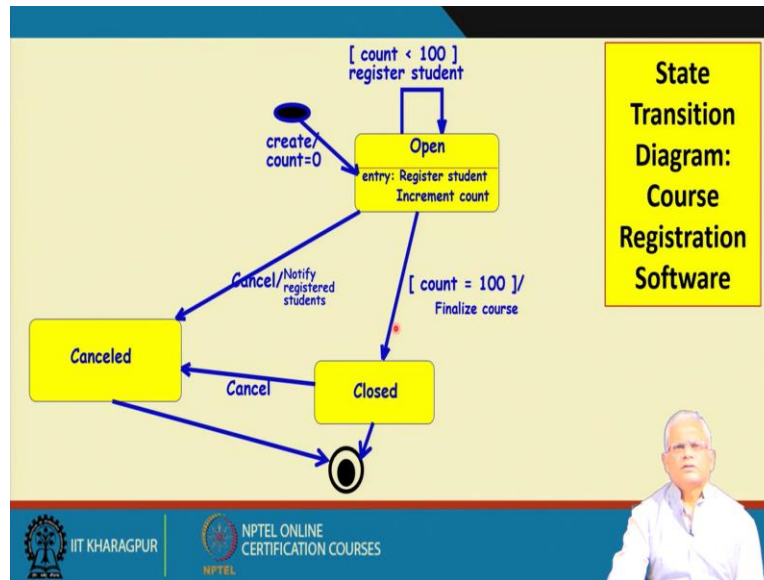
- When a course is created, it is initialized to open, and student registration made 0.
- If the course is open and a student registers, the count incremented.
- A course can be cancelled anytime: the registered students are notified and their registration cancelled.
- If registered student number reaches 100, the course is closed for registration, and it is allocated a room and a time slot.

Now just displaying one more problem (in the above slide), we will not solve it here we will just show the problem, please try to solve it on your own.

This is academic software of student registration in a course, this software will handle student registrations. When the course is created it is initialized to open and the student registration is zero. When a course is open the students register and each time a student registers the count is incremented. And if the count reaches hundred the course gets closed. Hundred is the maximum capacity of a course. And then, a room is allocated for the course. A course can get cancelled anytime by the administrator. So, the cancel event in that case the registered students are notified and then their registration is cancelled.

So, this is a simple state model first draw the state model of this and then need to write the C code which is straight forward once you developed the state model and also the java code. We will not really walk you through how to identify the states, the transitions, and so on please try to read here and identify the states, transitions, and actions.

(Refer Slide Time: 27:53)



So, this is a state model for that (in the above slide). Please draw on your own and check if you, it matches with it. And also, please try to write the C code and Java code.

We will stop here and in the next lecture, we will start with the interaction diagrams.

Thank you.