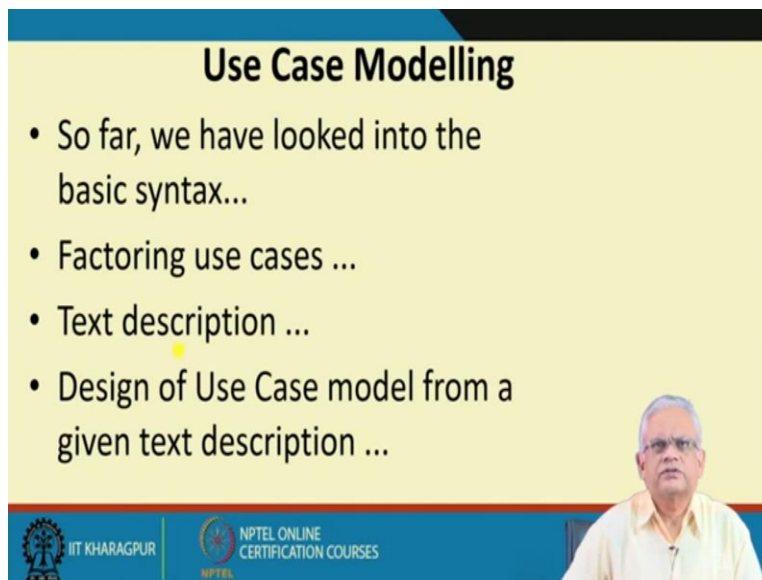


Object Oriented System Development Using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Use Case Guidelines
Lecture 06

Welcome to this lecture.

In the last lecture, we were discussing about use case modeling. We have said that use case modeling is one of the core activities in the object-oriented design process and the use case diagram is one of the most important diagram because all other diagrams are derived based on this use case diagram.

(Refer Slide Time: 0:45)



Use Case Modelling

- So far, we have looked into the basic syntax...
- Factoring use cases ...
- Text description ...
- Design of Use Case model from a given text description ...

The slide features a video inset of Professor Rajib Mall in the bottom right corner. At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL Online Certification Courses.

So far, in the use case modeling we had discussed about the basic syntax, how the use cases are represented, about the actors, communication relationship and so on. We also discussed about factoring of use cases. We had discussed that under some situations it is necessary to factor the use cases, especially when the use cases are complex and large and when there is scope of reuse the functionality across use cases. We had discussed three mechanisms for factoring the use cases: inheritance, include and extend relations.

We had also said that the diagram by itself conveys only limited information and therefore every use case diagram must be accompanied by a text description. Though there is no recommended format for text description, but we had discussed about some commonly used formats for text description. And after that, we had looked at some problems and tried to identify the use cases and construct the use case models. We had looked at how to read through the text description of the problem, identify the use cases. We also identify the actors and see what use cases they participate. In our initial model, we tried to represent those and also the event-based ones: various events and how the use cases respond to those.

(Refer Slide Time: 3:06)

- Use case name should begin with a verb.
- While use cases do not explicitly imply timing:
 - Order use cases from top to bottom to imply timing -- it improves readability.
- **The primary actors should appear in the left.**
- Actors are associated with one or more use cases.
- Do not use arrows on the actor-use case relationship.
- **To initiate scheduled events include an actor called "calendar"**
- **Do not show actors interacting with each other.**
- <<include>> and <<extend>> should rarely nest more than 2 levels deep.

Based on the discussion so far, let's look at some Style Notes recommended, proposed by Ambler in 2005 in his book. The Style Notes are about developing good quality UML diagrams. The first thing to note is that all use cases should be named with a verb form, for example, register student, register courses, et cetera. The functionality that we document here are essentially activities and should appear in verb form. Another good practice is that, even though the use cases are independent and they don't have dependency and it is not clear that which use case should be invoked first, there is no explicit notion of timing, but then we must try to document the use cases in the way they are typically invoked. For example, we have here (in the above slide use case diagram) the professor first registers for the courses and finds the students have registered, the students register for courses and they may drop courses.

In many use cases, there are more than one actor collaborating for completion of the use cases. The primary actor who invokes the use case should appear on the left. The other collaborating actor like just see in this example (in the above slide), the student drops courses, but during the dropping of the course, it is checked the calendar, that whether it is within the permitted time. Calendar here is the collaborating actor. The collaborating actors should appear on the right side of the use case diagram as much as possible. These are not rules but just general guidelines.

An actor can be associated with one or more use cases. We don't have to draw the actor again and again for every use cases it invokes. We can show the each and every interaction with use cases with communicate relations. These communicates relations are just lines, these are not directed lines so need to use arrow direction. A scheduled event in use case diagram may be done based on the internal clock but it is a good idea to have an actor named as calendar and this helps in the later design. In this it may appear slightly artificial because the calendar is actually part of the system, it is a system clock basically. But then it's a good idea to have the calendar mentioned here as actor instead of system clock because it helps in the later design. The actors interact with the system, but if there is an interaction between the actors themselves, this is not a part of the system that we are modeling. We are modeling only the computer interaction of the actors but the actors collaborating with themselves is not represented. For example, the student telephoned the professor which is an interaction among actors which is not represented here. So, the direct communication between the different actors is not represented; only the interaction of an actor with the system is represented.

We had discussed about factoring the use cases, using the include, extend and generalization. This factorization often leads to good quality design during the design process, but then this should be either a single level or at most two levels. Typically, it should be single level. In this example (in the above slide), single level is shown. But in more complicated cases, maybe two levels use but not more than that because it unnecessarily complicates the diagram and designing becomes difficult.

(Refer Slide Time: 9:02)

Effective Use Case Modelling

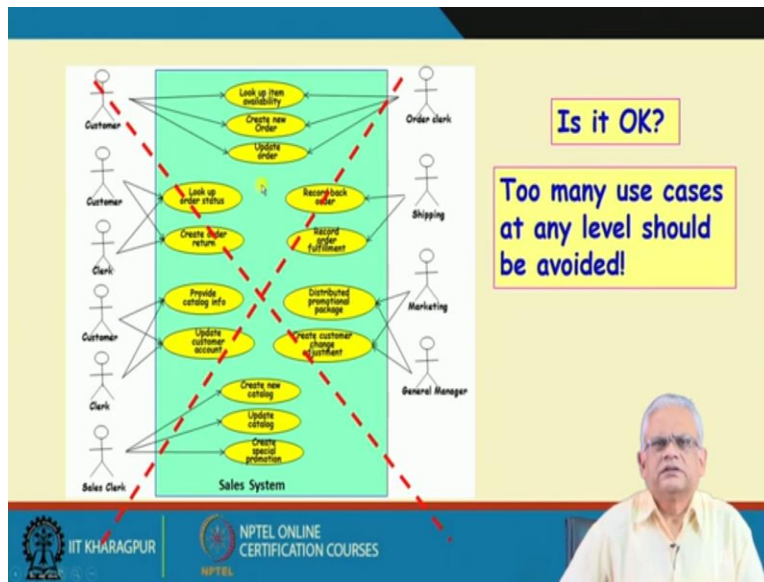
- Use cases should be named and organized from the perspective of the users.
- Use cases should start off simple and at as much higher view as possible.
 - Can be refined and detailed further.
- Use case diagrams represent functionality:
 - **Should focus on the "what" and not the "how".**

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

There are some more guidelines, the names of the use cases should be as given by the users. The technical names should not be used, for example, register courses is the terminology that everybody uses and we should not give another terminology. The use cases at the top level should be as simple as possible and then we may split these into more detail ones as the design proceeds.

Remember the use case diagrams represent functionality and these should focus only on what is the system is supposed to do and not how the system is supposed to do. For example, it should focus on what are the functionality supported by the system, what it should do, not how it will: will it access the data base, will it send a message to another process, semaphores, et cetera. So, these how aspects are not represented in use case modelling. Only the what aspects, what functionality to be supported is represented in use case modelling.

(Refer Slide Time: 10:38)

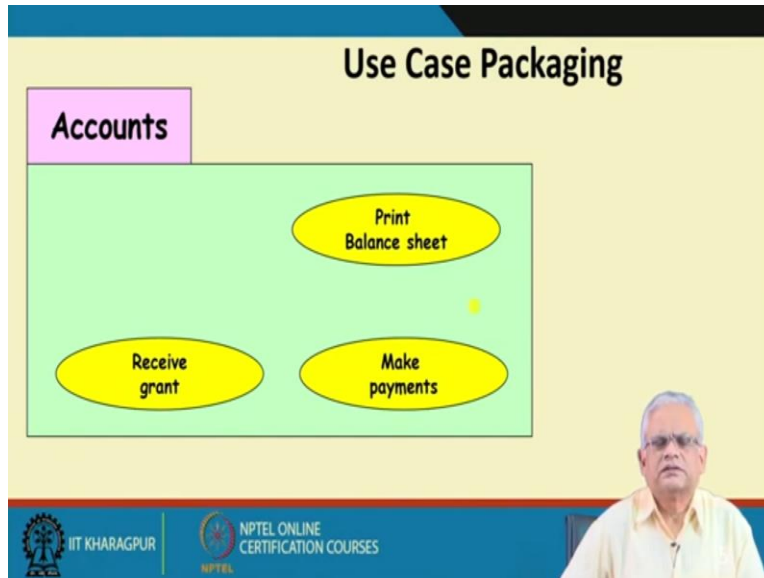


Now let's observe this example of use case model (in the above slide). There are lot of actors here and lot of use cases. Is it okay?

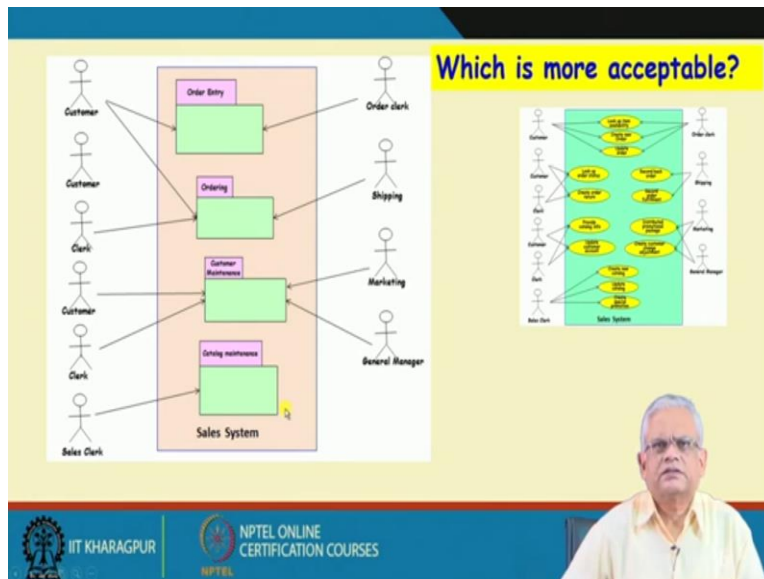
No, it is not okay. There are too many use cases, too many actors, we should avoid it. At the top level, we must have a simple diagram, but how do we achieve it?

We will see that we can achieve this even for a very complicated system by packaging. So, we should not use this kind of diagram where there are dozens of use cases, it makes it extremely difficult to understand this and proceed with the design. When we have many use cases, complicated use cases, we use packaging. A package is like a folder. This is the representation of a package (in the below slide), name of the package is 'Accounts' and looks like a file folder. We can see in the 'Accounts' package there are three use cases: Print Balance sheet, Receive grant, and Make payments. And we might have actors interacting with this package and that's the way we simplify the top-level diagram using packages.

(Refer Slide Time: 11:40)



(Refer Slide Time: 12:40)



So, the diagram that we have drawn earlier was this one (right side of the above slide). We can see, there are too many use cases. If we use packaging, it appears like this (left side of the above slide). Each package contains a set of coherent use cases, and they confirm to the name of the package. For example, one of the names of a package here is 'order entry'. The use cases Look up item availability, Create new order, Update order all these are part of this package naturally. We can see that rather too many use cases, its more acceptable to have the use cases organized into packages and later in other associated diagrams will elaborate the packages.

(Refer Slide Time: 13:54)

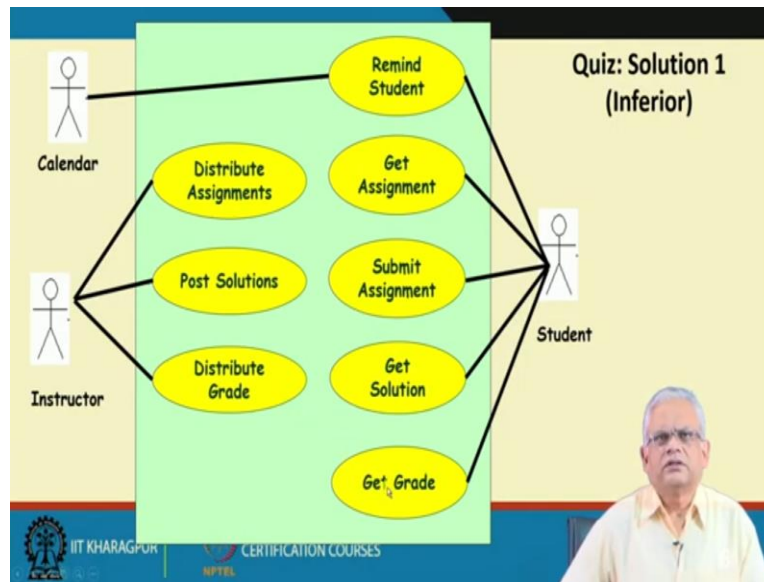
Quiz: Home Assignment System - Use Case Model

- HAS will be used by an instructor to:
 - Distribute homework assignments,
 - Review students' solutions,
 - Distribute suggested solution,
 - Assign a grade to each assignment.
- Students can:
 - Download assignments
 - Submit assignment solutions
- System:
 - Automatically reminds the students a day before an assignment is due.

Now, let's do some quiz. You must try to do with pen and paper before we discuss the solution. Here the problem is about a home assignment system (in the above slide), it is used by the instructors while taking the class. The instructor distributes home assignments and uses the home assignment system to download the students' solutions and look at solutions and evaluate them. Instructors also uses the home assignment system to post the suggested solutions and also for each submitted assignment by a student assigns a grade to the assignment. The student is another actor, the student can download the assignments and then upload or submit the assignments solutions on the home assignment system. The system clock triggers to automatically to remind the students, a day before an assignment is due. It sends out a mail to all students saying that: 'tomorrow is the last date for submitting the assignment'.

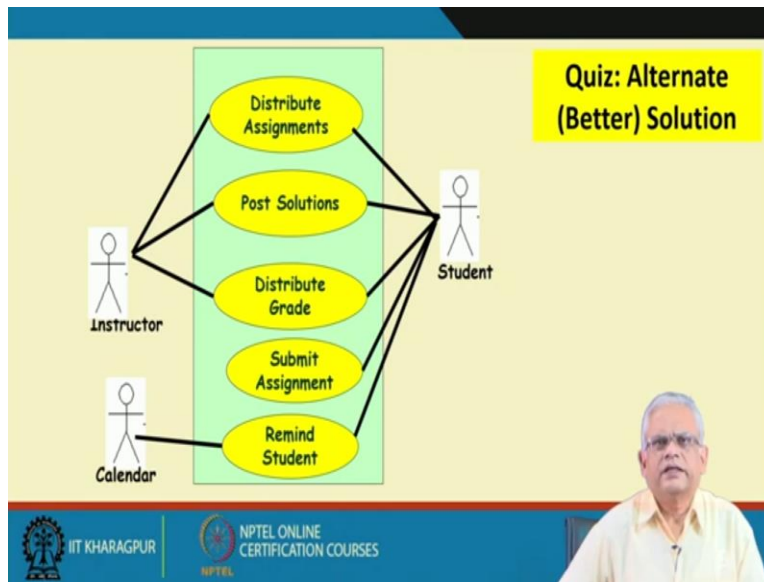
If we read through this, it is organized in a way which makes it very easy to draw the use case model, typically it is not organized in this structured way. It's more like an essay text description but here it is under each actor what functionality invokes is mentioned and a straight forward model development. So, we can have an instructor as the actor. The instructor functionality or the use cases invoked are distribute assignments, review and distribute solutions, and assign grades four use cases. That a naive solution, we just draw one use case for each. Another actor is the student who can download the assignment, submit assignment solution, get solution and the system reminds the student a day before an assignment is due.

(Refer Slide Time: 16:58)



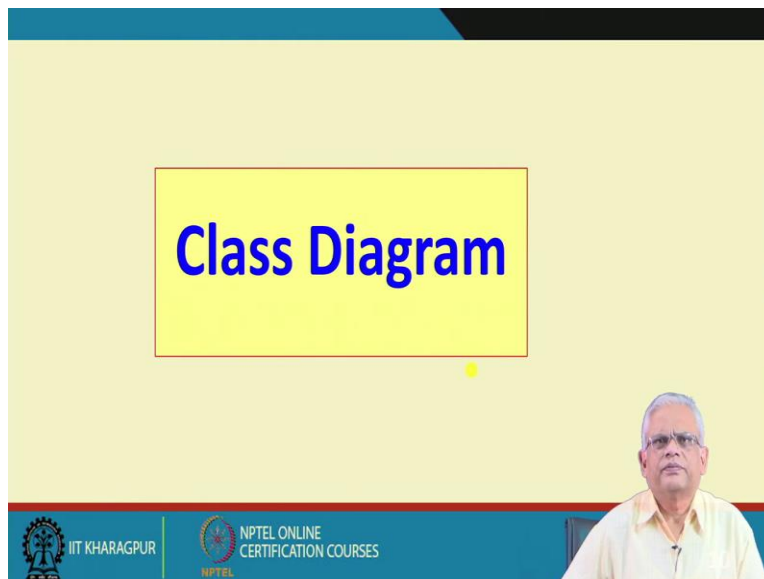
If we represent this in the form of a use case diagram, we get a straight forward diagram like this (in the above slide). Here, the calendar reminds the student, the instructor distributes grade, posts solutions and distribute assignments. The student gets assignment, submit the assignment, gets the solution and gets the grade. But this is not the best model that is possible. It should have been improved because see here, in the distribute assignment, there are two actors participating: the instructor distributes the assignment and the student gets the distributed assignment. Similarly, the instructor posts the solution and the student gets the solution. These are basically the same use case where both of them are participating. The instructor distributes the grade and the student gets the grade.

(Refer Slide Time: 18:14)



A better way to have this use case model is something like this (in the above slide). That the instructor distributes assignment and the student participates in getting the assignment, the instructor posts the solution, the student participates to download the solution. The instructor distributes the grade and the student gets the grade. The student submits the assignment and the calendar reminds the student and student is a participant here because the student gets reminded. So, this is a better diagram where less number of use cases are used.

(Refer Slide Time: 19:04)



We have so far looked at some very basic aspects of the use case model and I am sure that based on our discussion so far, you can develop a use case model for a given small problem.

Now let's look at the class diagram.

(Refer Slide Time: 19:31)

The slide features a yellow background with a blue header and footer. A yellow box in the top right corner contains the text 'Class: A Fundamental Object-Oriented Concept'. The main content consists of three bullet points: '●Template for object creation:' followed by '- Instantiated into objects' (with a small image of four blue elephants), '●Examples: Employees, Books, etc.', and '●Sometimes not intended to produce instances:' followed by '- Abstract classes' (in blue text). A small image of a man in a white shirt is visible in the bottom right corner of the slide area. The footer contains the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

A class is a fundamental object-oriented concept. Whenever we discuss anything with to do with object orientation, a class is the first thing that often comes up. A class to think of it, is a template for object creation because once we define a class, we use it to create many objects. We say that the class is instantiated into objects. We can use a class template to create objects. Examples of objects are employees, books, library members, et cetera. Those who are familiar with the programming languages like Java and C++, they know that some of the classes, they don't really produce objects, that are the abstract classes.

Question that naturally arises here is that if we have a class definition and we don't use it for object creation, what is it useful for? Let just repeat this question, if we define an abstract class, what is the use of it because we cannot create any objects from an abstract class?

The answer to this question is that yes, we cannot instantiate an abstract class but then, it helps us to reuse the definitions provided in abstract class because we can define concrete classes based on the abstract class which will reuse the functionality defined once in the abstract class and we

can derive several concrete classes from abstract class. We define the functionality only once in the abstract class and reuse this in the concrete classes.

(Refer Slide Time: 22:09)

The slide contains the following text and diagrams:

- Entities with common features are made into a class.
- Represented as solid outline rectangle with compartments.
- Compartments for **name, attributes, and operations.**
- Attribute and operation compartments are optional ... used depending on the purpose of a diagram.

Class Diagram

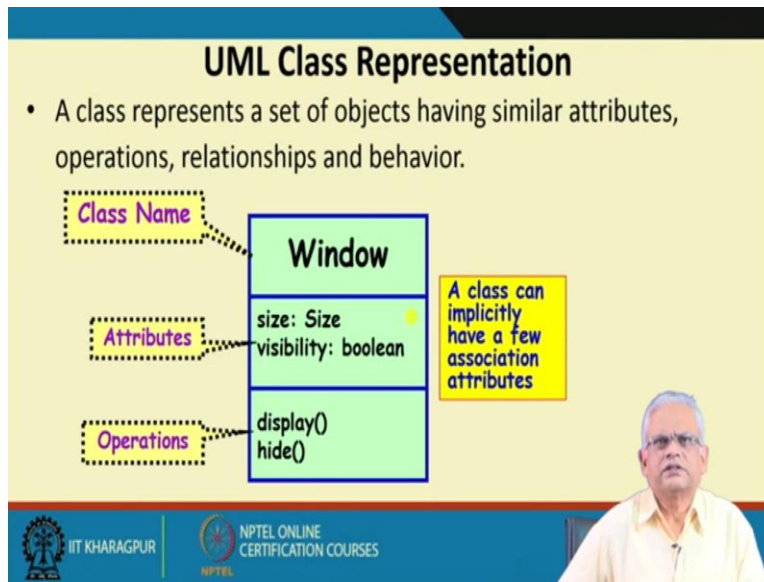
Window
size: Size visibility: boolean
display() hide()

Window

The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, and a small video inset of a speaker in the bottom right corner.

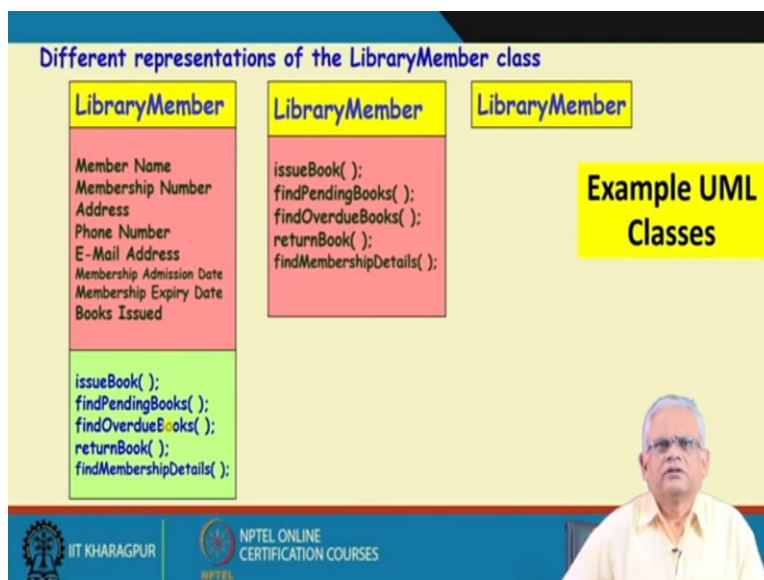
Now, let see how we can draw a class diagram. A class as we know that these are entities or objects with common features. For example, the books are the objects and these objects are constituting the book class. A class is represented as a solid outline rectangle with three compartments: the first compartment here is the name of the class and then we have the attributes in the second compartment and the methods are in the third compartment. For example, in the above slide, we can see that window class has three compartments. But it is not necessary to always have the attribute and the operations segments or the compartments of the class, we can simply write a class name with a rectangle. Like window class (yellow color) shown in above slide without second and third compartment. So, we can also alternately represent the class, just by writing the name of the class without giving the attributes and the operations. The attributes and operations are optional, the name of the class is mandatory. So, as the attributes and operations are optional, we can use these attribute and operations compartments depending on the purpose of the diagram. It will become more clear as we proceed with our discussion that under what situations, we need the attribute and the method segments.

(Refer Slide Time: 24:17)



In the above slide, we can see basic representation of class diagram in UML. In the rectangle, the first segment is the name of the class and then the second and third segments respectively are the attributes and the operations. A class can have some implicit attributes due to associations with other classes but here we have not shown. This will be clear in our next lecture where we discuss about the association relationship between classes. We will see there that whenever there is a class associated to another class, it will have some attributes appearing implicitly on account of the association relation.

(Refer Slide Time: 25:08)



Now let's look at another example of UML class (in the above slide). The same LibraryMember is represented just by its name (the right one in the above slide), name and the operations (the middle one in the above slide), and the name, the attributes and the operations (the left one in the above slide).

So, the question is, when do we use each of these? Or is it that we can use it as we like?

No, not really. The design processes are the one using which we will have a step by step method of given a problem description. We will come up with object-oriented solution, and there in almost every design process, we will see that at the start of the process, we identify only the classes and we represent them by just a class name within a rectangle. Next, as we proceed in our design process, we identify the methods, we populate the classes with the methods and therefore towards the middle of the design process, we get class diagram with class name and methods. And subsequently in the design process later, we get kind of representation where we have name of the class, the attributes and the operation and it becomes feasible to implement this kind of representation.

There are many case tools that can generate code directly based on the complete class diagram (the diagram with all three segments). For example, these tools can create complete code from our LibraryMember class diagram (the left one of the above slide) just by a click of a key. The code will have the class definition, the attributes defined and the method prototypes defines.

We will stop here. We will continue from this point in the next class.

Thank You.