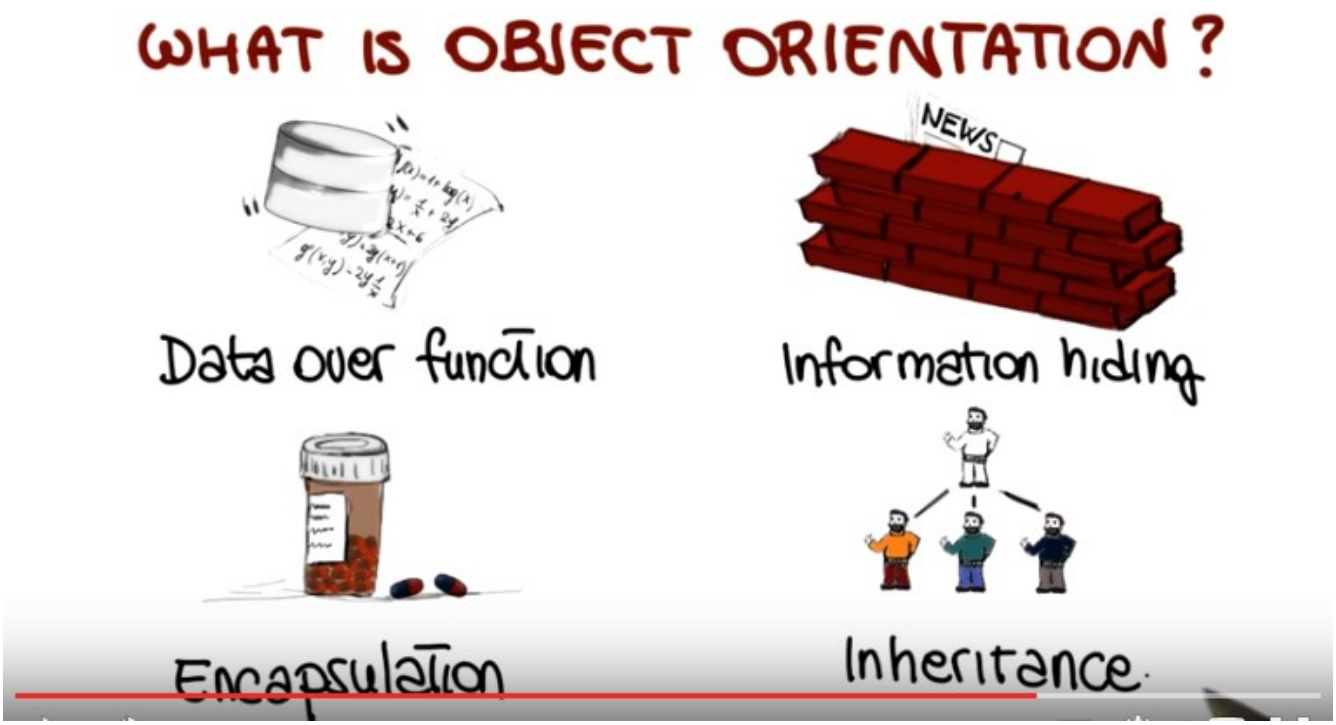
一個很好的講 UML 的網頁(已弄成 course note 8-extra):
http://www.classdraw.com/help.htm

1. Hi. In the last lesson we discussed requirements engineering. This lesson is about object orientation and other related concepts. The lesson is split in two main parts. In the first part, we will provide a quick introduction to object orientation and object oriented analysis and design. In the second part, we will cover the essential of UML, which is the notation that we will use in the rest of the course and also in our projects.

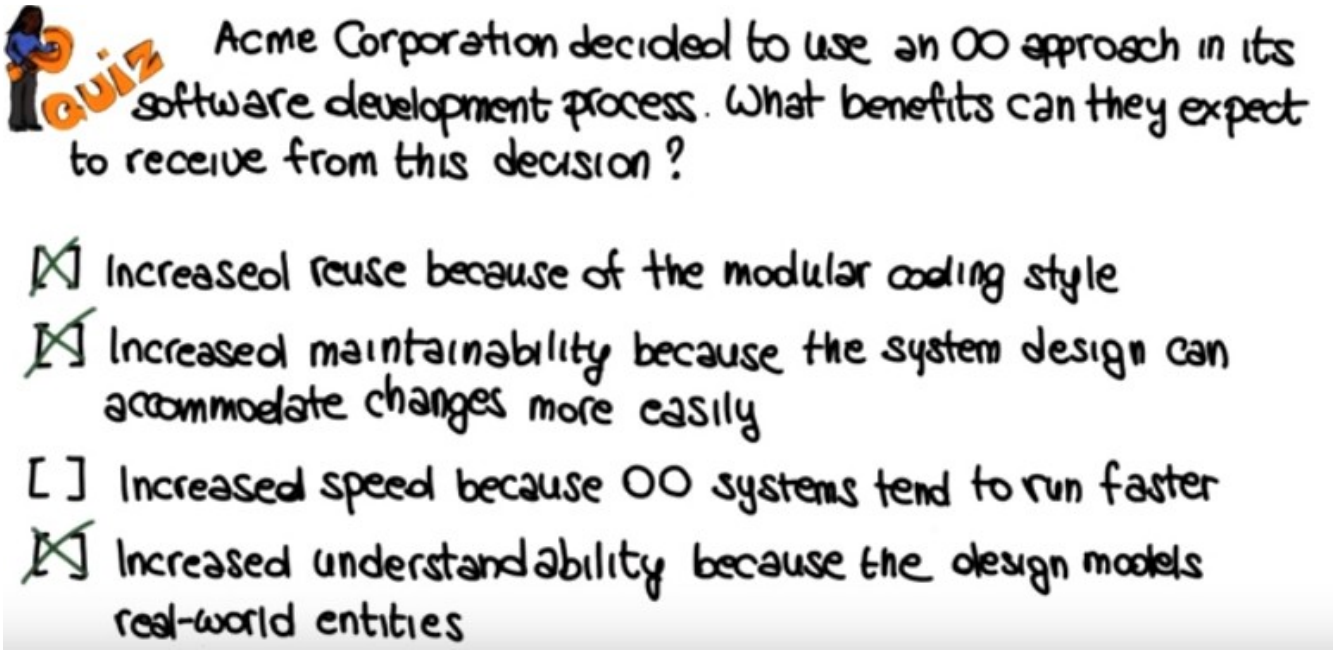
2. Let's start with a quick introduction to object orientation and the fundamental concepts behind it. And let's start by discussing what exactly is object orientation? If you're younger than me, it could be that the first programming language you learned was already an object-oriented language. But things were not always like this. Before object orientation became prevalent, people were not used to thinking in terms of objects. So what happened afterwards? And what does it mean to think in terms of objects and to follow an object-oriented approach? First of all, it means to give precedence of data over function. Did items rather than functionality become the center of development activities? This also allows for enforcing the very important concept of information hiding, which is the encapsulation and segregation of data behind well-defined and ideally stable interfaces. In order to be able to hide the design and also implementation decisions. And note that the terms encapsulation and information hiding are often used interchangeably, although some people prefer to think of information hiding as being the principle and encapsulation being the technique to achieve. Information hiding. The key concept though, no matter which term you use, is really to gather, to seclude this data behind sort of a wall and give access to the data only through interfaces that you, the developer define. And why is that important? Oh, for many reasons, and one of the main ones is that it makes code more maintainable. Because the rest of the code, the rest of the system doesn't have to be concerned on how the implementation details or the design are defined. And therefore, any change that happens behind this wall doesn't concern the rest of the system. And, doesn't affect the rest of the system, as long as you keep your interfaces consistent. Another advantage of focusing on objects and encapsulating the information into cohesive entities is that it allows the reuse of object definitions by incremental refinement. Which is what we normally call inheritance. And inheritance is definitely a fundamental concept in object orientation. For example, we can define a car as a refinement of the vehicle. That there's some additional characteristics with respect to a generic vehicle. And then we can use the car wherever a vehicle can be used, which is what we call polymorphism. And we'll continue this discussion for a very long time. Because there's so many things that could be discussed when we talk about object orientation, its characteristics and its advantages. But in the interest of time, let's for now just stop here. And start talking about two key concepts in object orientation.



3. Let's start with objects. An object is a computing unit organized around a collection of state or instance variables that define the state of the object. In addition, each object has associated with it a set operations or methods that operate on such state. So what that means is that operations and methods read and write instance variables. And in traditional object orientation, we say that operations are invoked by sending a message to the appropriate object, which is what we call normally a method implication. So now that we define what an object is, state variables are attributes and operations or methods, let's see what a class is. A class is basically a template. A blueprint, if you wish, from which new objects, which is what we call instances of the class can be created. And notice that the fact of having a blueprint for objects that allows us to create as many objects as we want can further reuse, and also contribute to make the code more readable, understandable, and therefore ultimately more maintainable.

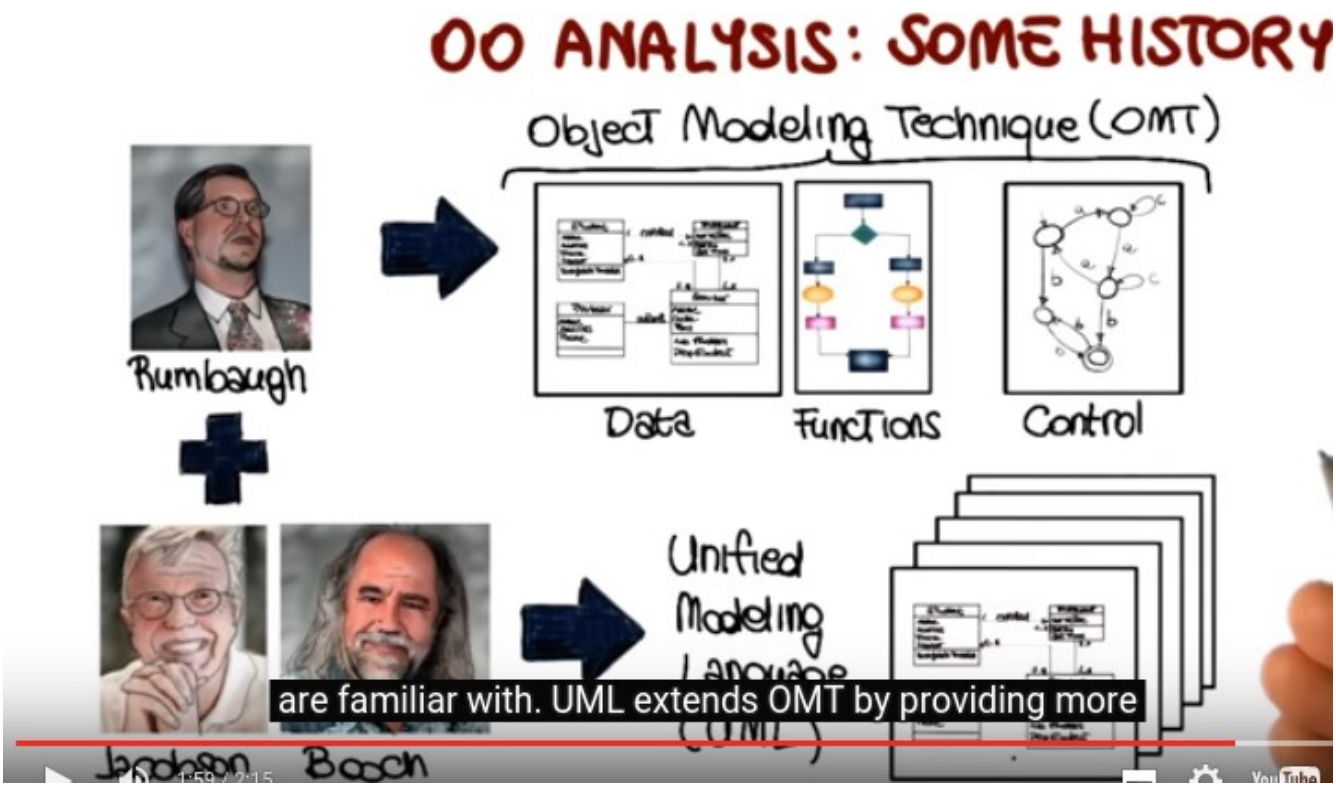
4. So in more general terms, why do we want to use object orientation? The first reason is that object orientation can help reduce long-term maintenance costs by limiting the effects of changes. As we saw, the effect of using encapsulation and information hiding makes it easier to modify parts of the system without affecting the rest of the system. Object orientation can also improve the developing process by favoring code and design reuse. In general, object orientation helps enforce good design principles. Principles such as the ones that we saw in encapuslation, information hiding, high cohesion, low coupling and we will discuss these aspects more extensively in the next mini course which is centered around design concepts.

5. Now let's make sure that we understand the benefits of object orientation through a quiz. Imagine that acme corporation decided to use an objetory entered approach in its software development process. If this is the case what benefits can they expect to receive from this decision. And here I'm listing some possible benefits. Increased reuse because of the modular cooling style. Increased maintainability because the system design can accommodate changes more easily. Increased speed because object oriented systems tend to run faster. And increased understandability because the design models real world entities. So, I would like, as usual, for you to mark all that apply.

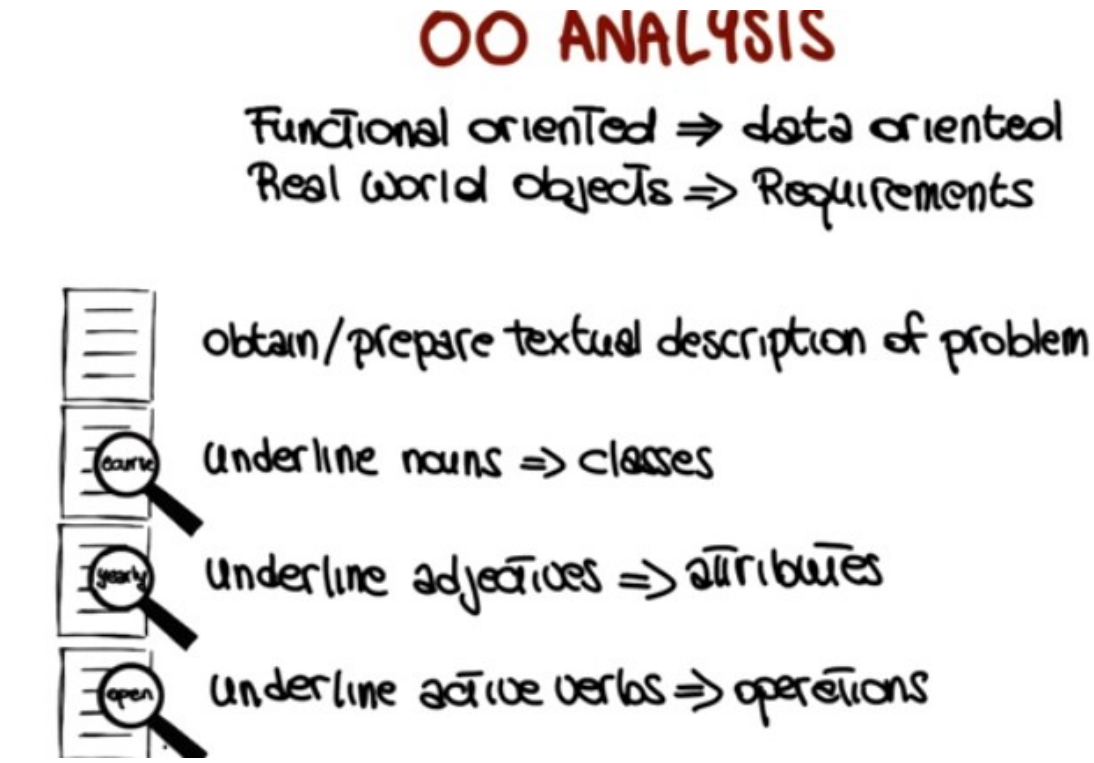


6. So, now let's see which ones of these benefits can actually be expected. Definitely, the first one. The modular coding style typical of object-oriented approaches can, and normally does, increase reuse. Similarly, because of the characteristics of typical object-oriented systems, these systems are normally easier to change. Because they're more modular, they're more cohesive, they're more decoupled. And therefore, all of this contributes to increase the maintainability of the resulting systems. So we're going to mark this benefit as well. There's really nothing about object-oriented systems. That make them run faster and, therefore, this is not a benefit that we can expect from the use of an object-oriented approach. Conversely, the last one is an expected benefit because normally, the fact of designing real-world entities, which is one of the characteristics of the object oriented approaches, does increase understandability. It's easier to understand the system because we can relate. To the system because we can recognize in the systems, real world entities that we are used to see and that we understand.

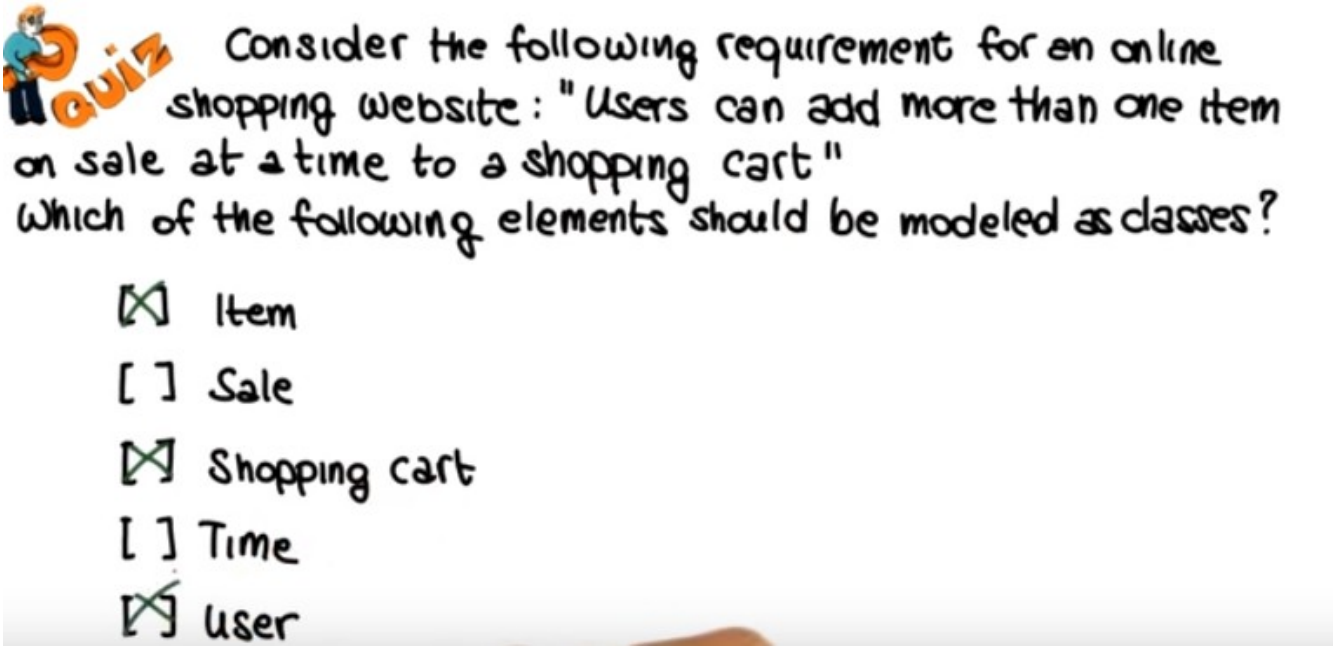
7. The use of object orientation and object oriented concepts led to what we call OOAD, object oriented analysis and design. OOAD is a software engineering approach whose main characteristics is to model a software system as a group of interacting objects, and we'll see what that means. In particular, in this lesson we will specifically focus on the first part of this, object oriented analysis, which is a requirements analysis technique that concentrates on modeling real world objects. And as I usually like to do, I would like to start by providing some historical perspective on object oriented analysis to better understand how we went from a function-centric world to a data-centric world. And several people contributed to this shift in perspective, but I'd like to mention a few that were particularly influential. Starting from James Rumbaugh, which in the 90s developed an integrated approach to object oriented modelling with three main aspects. A data aspect, so the modelling was based on using an extended version of entity relationship diagrams (這不就是 6400 Dabase 裡講的那個嗎) to describe classes and inheritance. So that's what was called the object model. And the second aspect has to do with functions. So data flow diagrams were used to represent the functional aspects of the system, where each function was then becoming a method in a class. So this is what is called the functional model. So object model and functional model. The third model in Rumbaugh's methodology had to do with control. So it was representing the dynamic aspects of a system. And it uses state machines, which we'll cover in more detail, to represent how a system would evolve going from one state to the other based on what happened to the system. These three models together represented what was called the Object Modeling Technique, or OMT. And OMT combined with contributions from several people, and in particular Jacobson and Booch, evolved into what we call the Unified Modeling Language, which is UML, which is probably the modeling language that most of you are familiar with. UML extends OMT by providing more diagrams and a broader view of a system from multiple perspectives. So, in the second part of the lesson, we will cover some of these diagrams in details, but before that, I'd like to talk a little bit more about object oriented analysis, and how we can perform it.



8. So let's look at the object-oriented analysis in a little more detail. As I said earlier, traditional analysis and design techniques were functionally oriented. What that means is that they concentrated on the functions to be performed, whereas the data upon which the functions operated were secondary to the functions themselves. Conversely, object oriented analysis, is primarily concerned that with a data objects, so we went from a functional oriented view to a data oriented view, what that means is that during the analysis phase, we define a system first in terms of the data types and their relationships, and the functions or methods are defined only later and associated with specific objects which are sets of data. So let's see how we can perform object orientated analysis in practice, so the basic idea is to be focused on the objects of the real world. So to go from a real world objects to a set of requirements. And we can describe this as a four-step process. The first step is to obtain or prepare a textual description of the problem to be solved. So obviously, we need to start from some description of the system that we need to build. And this is a very practical oriented approach, so that the next thing we do is that we take the description and we underline nouns. In this description. And the nouns that we underline will become classes in my analysis. We then look at adjectives in the document. We underline those, and we use that information to identify the attributes of the classes that we've previously identified. At this point we focus on active verbs in the description, and the analysis of the active verbs will give us the operations that we'll need to define for our classes. So, again, underline nouns, and those will become the classes in my system. Then, objectives. And, those will be the attributes of the classes. And, finally, active verbs that will become the operations of my classes. And of course, this is a high level view to take this with a grain of salt. But we will see that it's a very good pragmatic approach to identifying requirements, starting from a description of the system to be built.



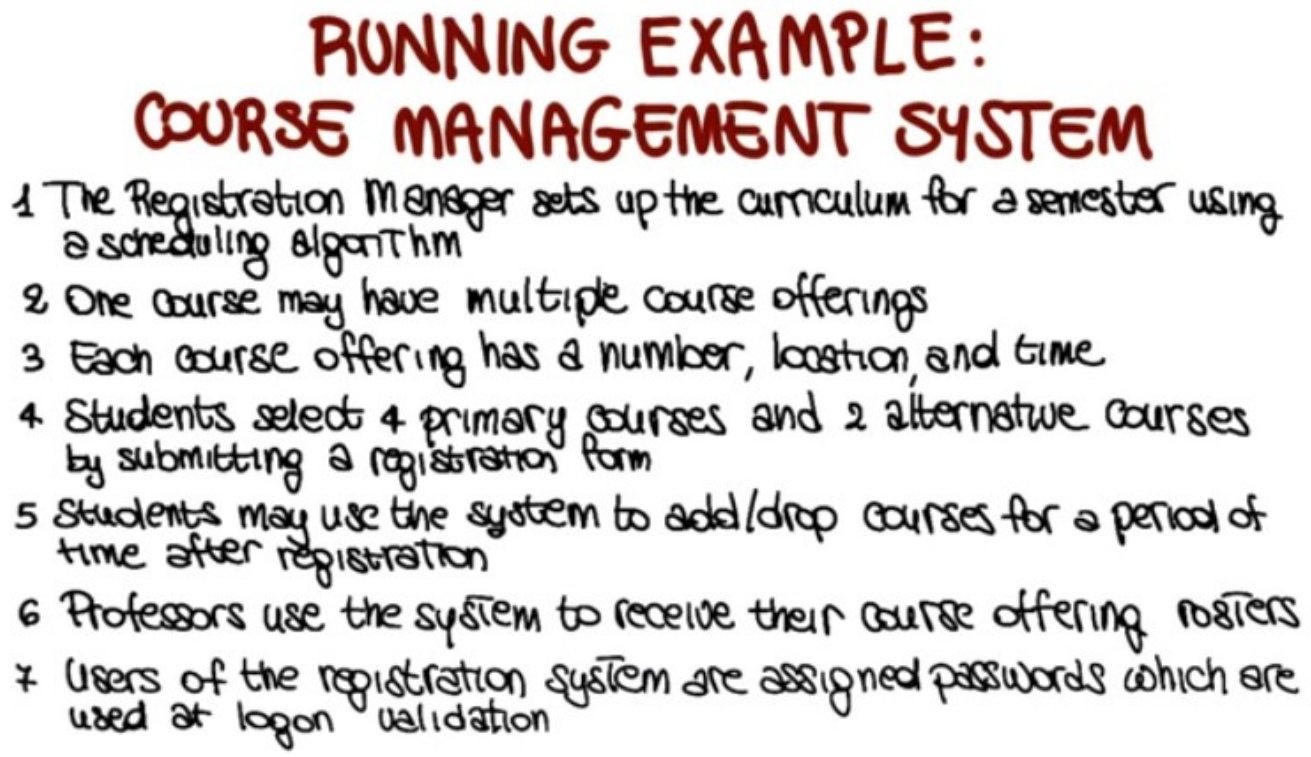
9. Now let's see how object oriented analysis might work in practice by considering the following requirement for an online shopping website. The requirement says that users can add more than one item on sale at a time to a shopping cart. So, looking at this requirement I would like you to tell me which of the following elements should be modeled as classes. And the elements are: item, sale, shopping cart, time and user. So mark all that apply.



10. Looking at the requirements, item is definitely a relevant element for my system, so it is appropriate

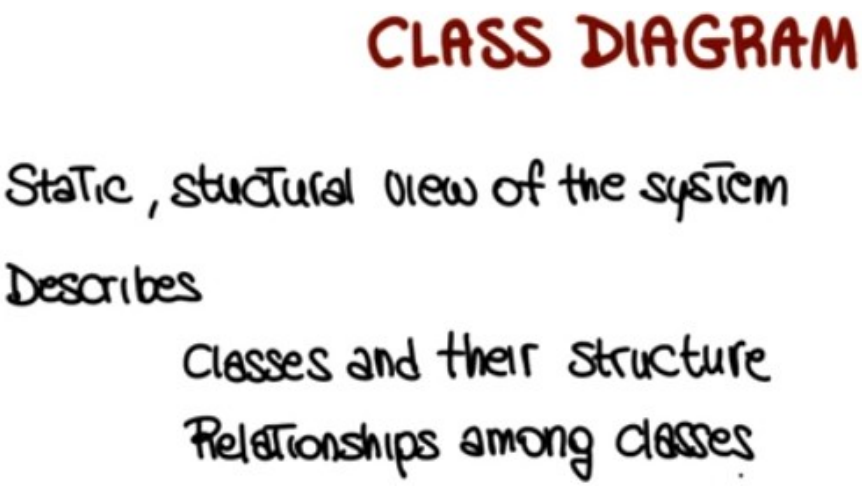
to model item as a class. Sale, on the other hand, is more of a characteristic of an item, an attribute of an item, rather than a class in itself. So, we're not going to mark this one. Shopping cart sounds, as well, as an important element for my system. Time can be an important system in some contexts, but in this case we're measuring time just because more than one item at a time can be added to the shopping cart. So, time really doesn't have any reason for being modeled as a class. And finally, user also seems to also have an important role to play in the system, and therefore we will model user as a class as well.

11. This concludes the first part of this lesson in which we discussed the basic object-oriented concepts. And, we started to look at how to perform object-oriented analysis. In the second part of the lesson, I will introduce UML, and we will perform the object-oriented analysis steps that we just saw using an example. A course management system so before getting to the second part, let me introduce the example. As we mentioned before, the first step is to start from a textual description of the system the we need to analyze and that we need to build. So that's exactly what I'm going to do. I'm just going to read through this description then we'll reuse throughout the rest of the lesson. The registration manager sets up the curriculum for a semester using a scheduling algorithm and the registration manager here is the registrar. So we will refer to the registration manager both as registration manager and as registrar in the rest of the lesson. One course may have multiple course offerings, which is pretty standard. Each course offering has a number, location, and a time associated with it. Students select four primary courses and two alternative courses by submitting. A registration form. Students might use the course management system to add or drop courses for a period of time after registration. Professors use the system to receive their course offering rosters. Finally, users of the registration system are assigned passwords which are used for login validation. So, as you can see, this is a kind of a high-level description of a standard course management system. So, if you ever used a course management system, you'll recognize some of the functionality described here.



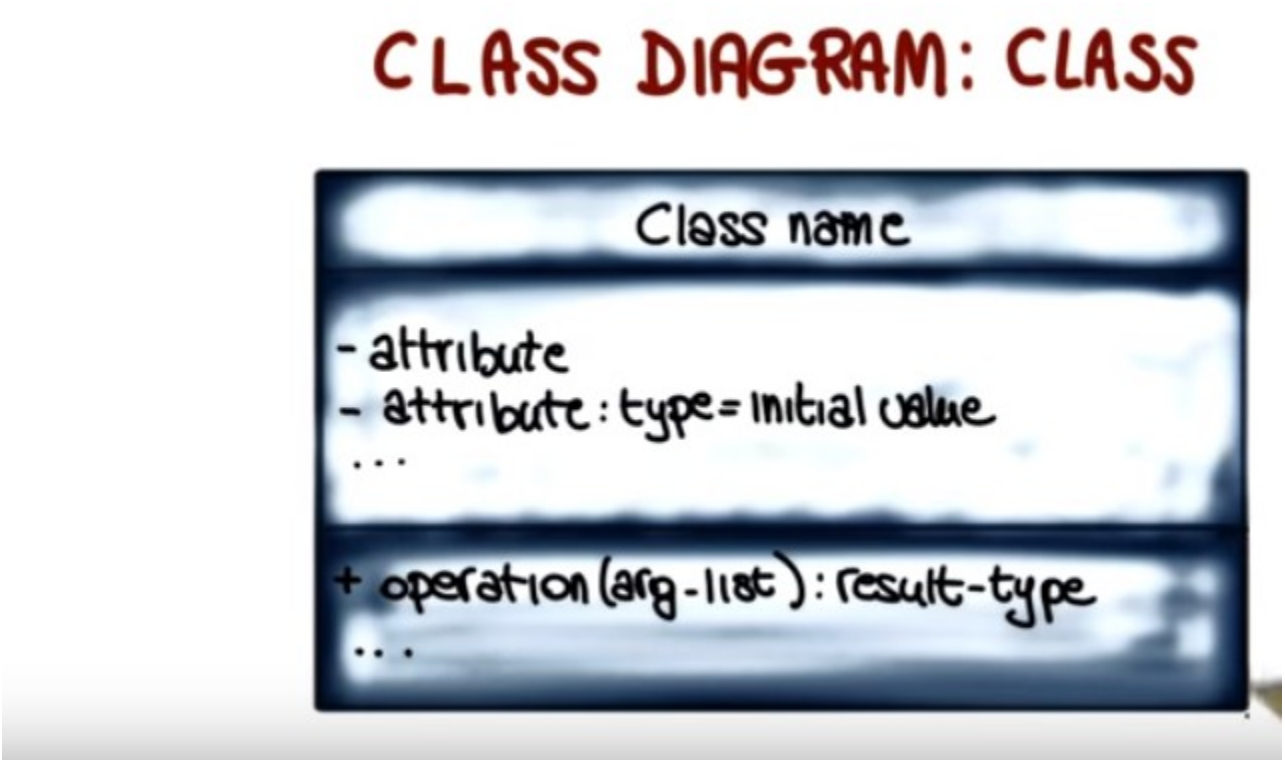
12. Let's now start talking about UML, the Unified Modeling Language. And we are going to start by

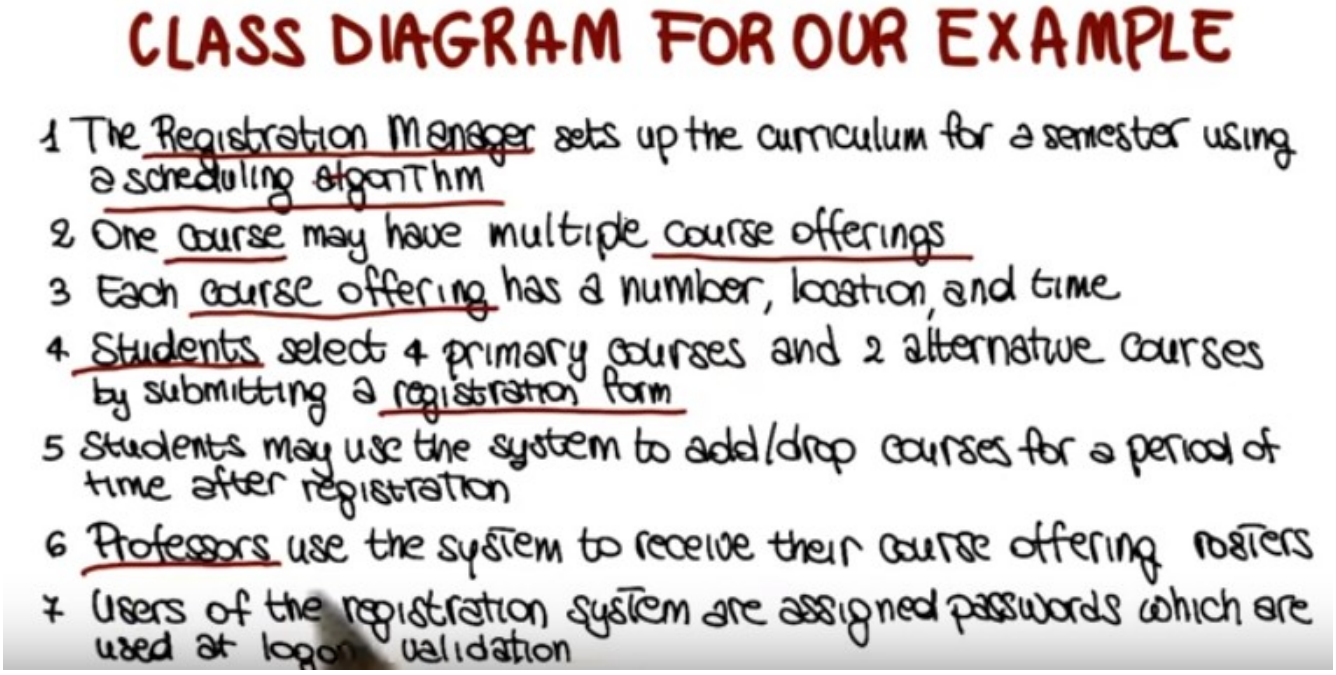
looking at UML structural diagrams. This are the diagrams that represent static characteristics of the system that we need to model. This is in contrast with dynamic models which instead behaviors of the system that we need to model. And we will also discuss dynamic models, later on in the lesson. We're going to discuss several kinds of diagrams, starting from the class diagram, which is a fundamental one in UML. The class diagram represents a static, structural view of the system, and it describes the classes and their structure, and the relationships among classes in the system.

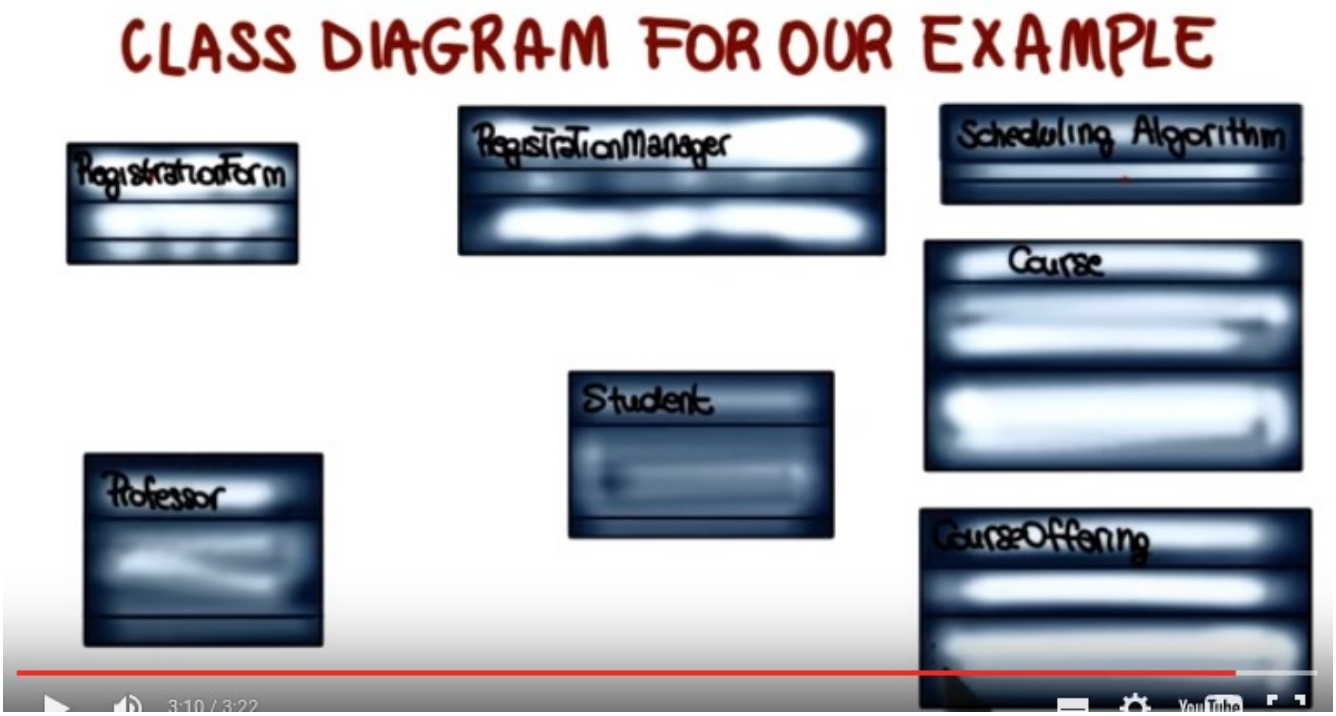


13. So how do we represent a class in a class diagram? Class is represented as a rectangle with three parts. The first part is the class name. Classes should be named using the vocabulary of the domain, so we should pick names that make sense. And the normal naming standard requires that the classes are singular nouns starting with a capital letter. The second part of the class are the attributes of the class. With the set of attribute for the class, we find the state for the class. And, we can list an attribute simply by name, or we can provide the additional information. For example, we might find the type of the attribute, and we might also find the initial value for the attribute. Finally, the third part of the class consist of the operations of the class. And normally, the operations of the class are represented by name, with a list of arguments. That the operation will take as input, and with a result type. So the type of the result produced by the operation. Something else you can notice in this representation is the fact that there is a minus before these attributes and a plus before this operation. This indicates what is called the visibility of these class members. So the minus indicates that the attributes are private to the class. So only instances of this class, roughly speaking, can access these attributes. And notice that this is what allows to enforce the information hiding principle, because clients of the class cannot see what's inside this box, what are these attributes. The plus conversely indicates that this is a public operation. So something that is visible outside the class. And, in fact, normally, this is what we use to define the interface for my class. So we encapsulate the state of the class and we make it accessible. To the extent that we want and that is needed through a set of public operations. Last thing I want to note is the use of these ellipses that we can utilize if we want to indicate that there are more attributes for example, or more operations. But we just don't want to list them now. Okay now that we know what a class is, and how is this represented, let's start our analysis of our course management system. By identifying the relevant classes in the system, we need to bring back the description of our system. And what we want to do, is that we want to go through the description and underline the relevant nouns in the description. And here I encourage you to stop the video and to do the exercise of underlying such now yourself before listening to my explanation into how I do it. For example in this case I may going to underlined the registration manager which is a noun and probably a relevant one. The scheduling algorithm, also seems like a relevant concept, so is the course. The course offerings, again, course offerings over here.

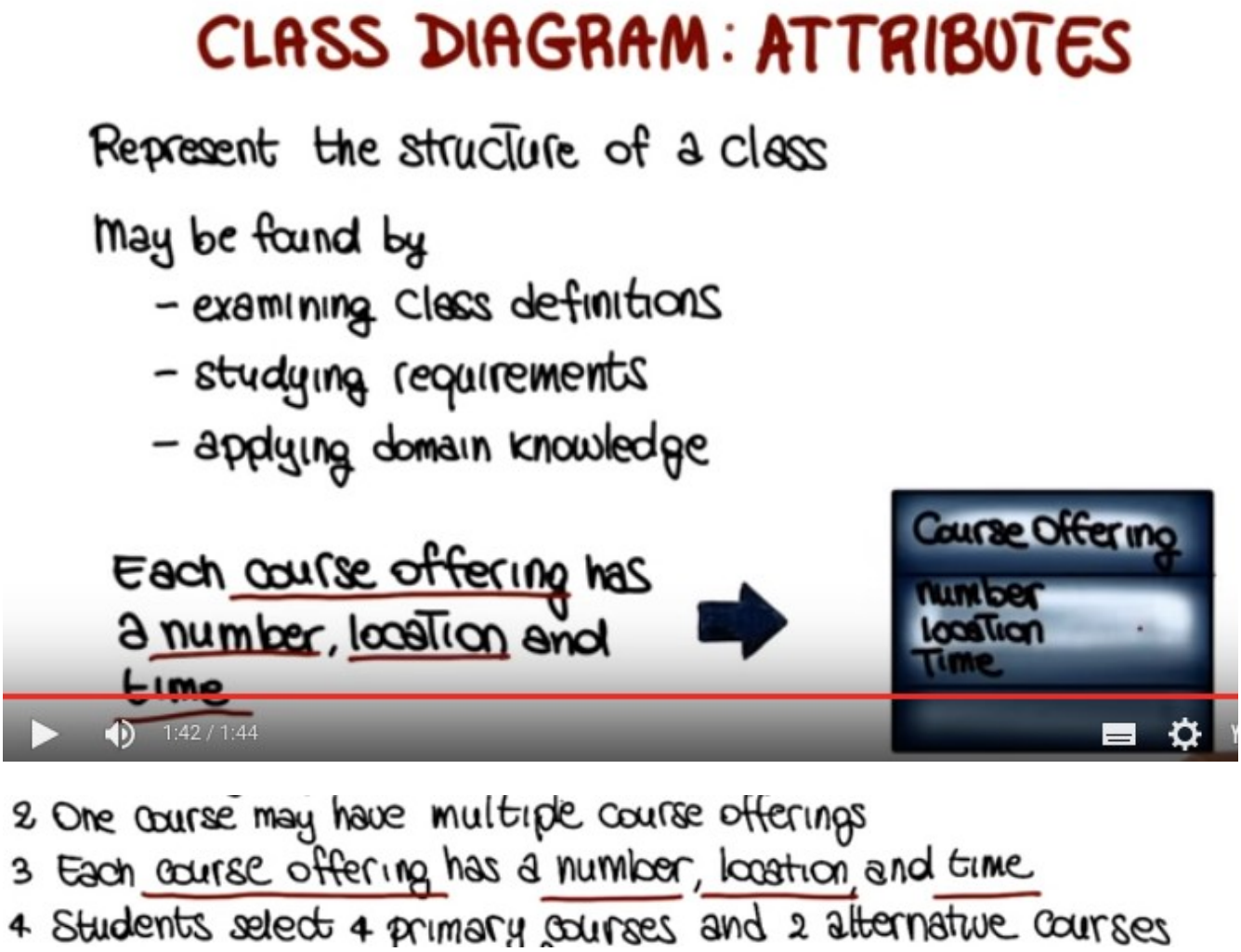
Definitely, the students seem to be a relevant noun and so is probably the registration form and the professors. Okay, so, at this point, I identified seven possible classes for my system. So, what I'm going to do is simply to create classes for each one of this nouns. So my initial class diagram looks exactly like this, with the seven classes where for each class, I picked the name that is representative of the domain. So, in this case, it's pretty straightforward. The registration form is called registration form, the student is called student and so on and so forth. But you can already see how this analogous method is starting from a description of the real world and And it's just identifying objects or classes in the real world and transforming them into entities in my analysis document.



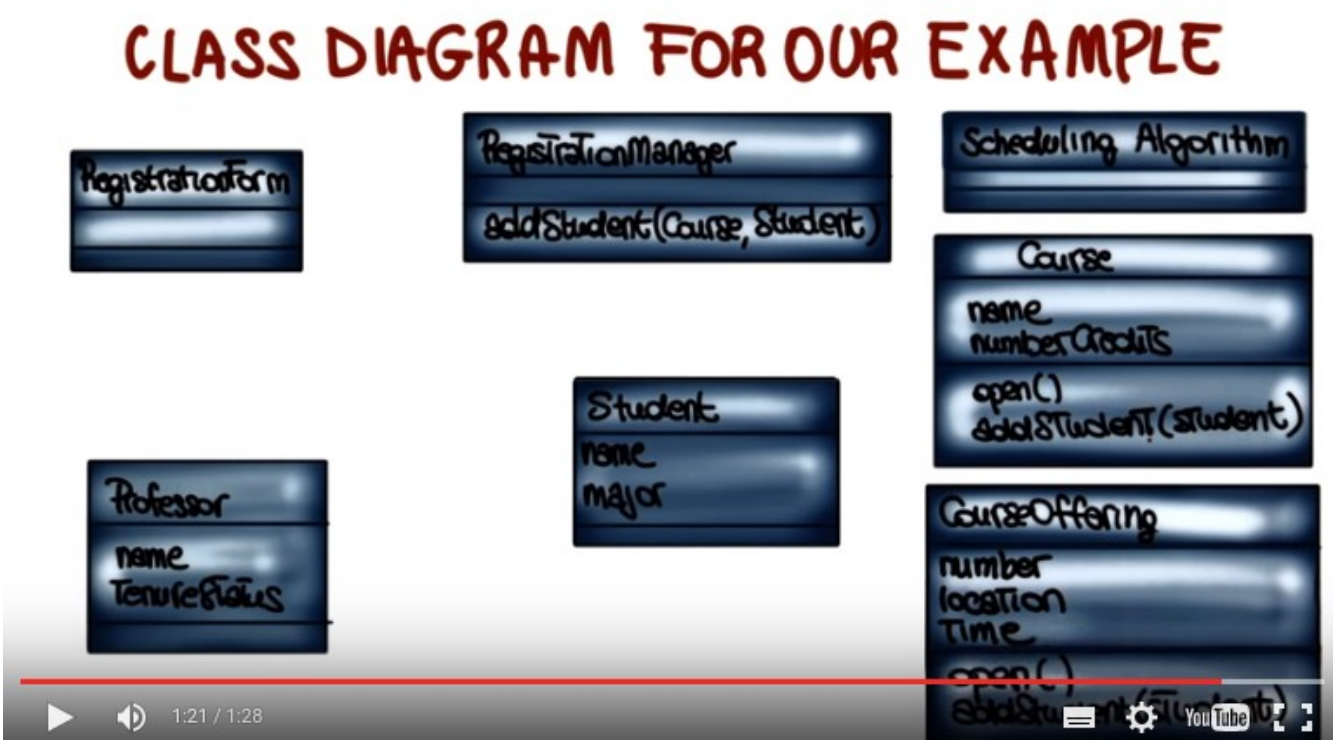




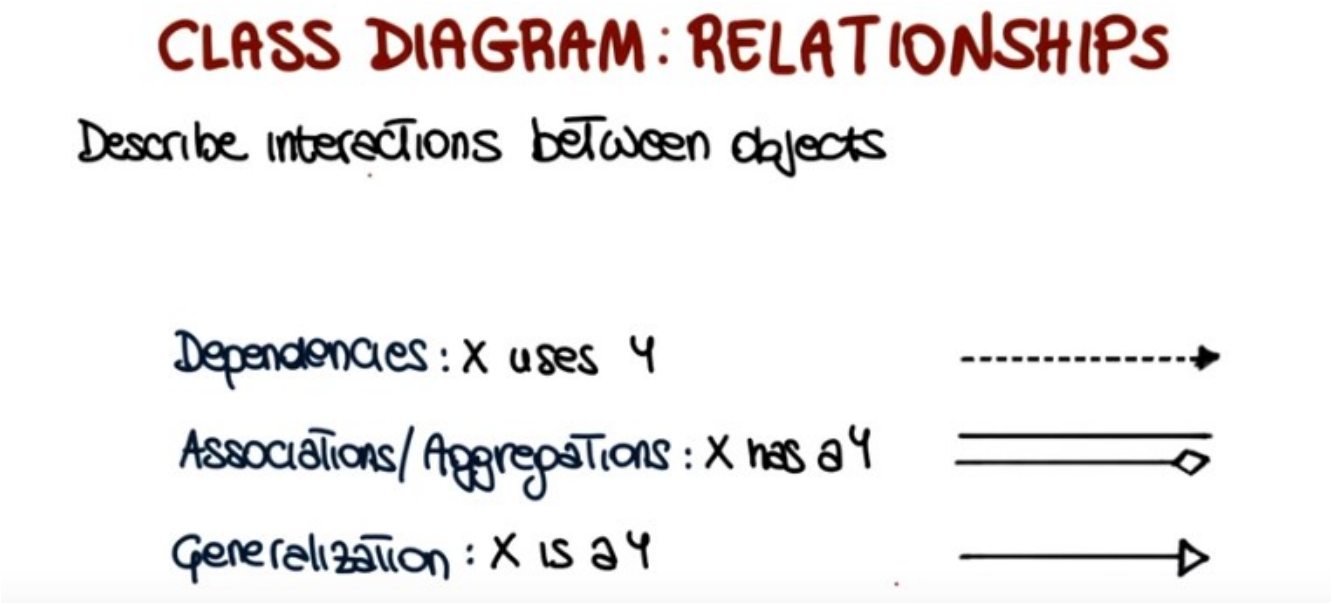
14. Now that we identify the classes in the system, let's see how we can identify the attributes for these classes. First of all let's recall what attributes are attributes represent the structure of a class the individual data items that compose the state of the class. So how do we identify these attributes? Attributes may be found in one of three ways. By examining class definitions, by studying the requirements, and by applying domain knowledge. And notice that I want to stress, that this is always a very important aspect. No matter what kind of system you're developing. Domain knowledge tends to be fairly important to identify things which might not be provided in the descriptions of the system that tend to be incomplete. And that you can derive by the fact that you are familiar with the domain. So always keep in mind the domain knowledge is important for analysis, for design, for requirements gathering and so on. So now let's go back to our description of the system. As I said, I will bring you back for each step of our analysis. And in this case, we're going to focus on course offering. And we can say that the course offering, according to the description, has a number. A location, and a time. So this is a pretty clear indication that these are important aspects of the course offering. So they probably should become attributes of the course offering class. So now if we report here that sentence, and once more, we underline the information that we underlined in the description. We can clearly see how this can be mapped into the definition of the class. So our class course offering after this step the analysis will have 3 attributes: number, location, and time. And as you can see here, I'm not specifying the type or any other additional information. So in this first step I'm just interested in having a first draft. of the class diagram, that I can then refine in the next iterations of my analysis.



15. At this point we have our classes, our attributes, what we're missing is the operations for the class. Let me remind you that operations represent the behavior of a class, and that they may be found by examining interactions among entities in the description of my system. So once more, let's bring back our description, and let's in this case focus on this specific item. That says that the students may use the system to add courses. So this is clearly indicating an action that the students should be able to perform. But notice that this doesn't mean that this is an operation that should be provided by the student's class. It rather means that there should be, somewhere in the system, the possibility of performing this operation. So let's see what this means for our example. This might mean, for example, if we focus on the RegistrationManager, so that there should be an operation in the RegistrationManager that allows me to add a student to a course. And this, in turn, means that both Course and CourseOffering should provide a way to add a student. And therefore, I add this corresponding operation to the RegistrationManager, to the Course, and to the CourseOffering. So after doing that we will continue and populate in a similar way, the other classes in the system. So let me recap. Now we saw how to identify classes. How to identify members of the classes, and particular attributes, and operations. There is one thing that we're missing, a very important aspect of the class diagram which is the relationships between these classes.

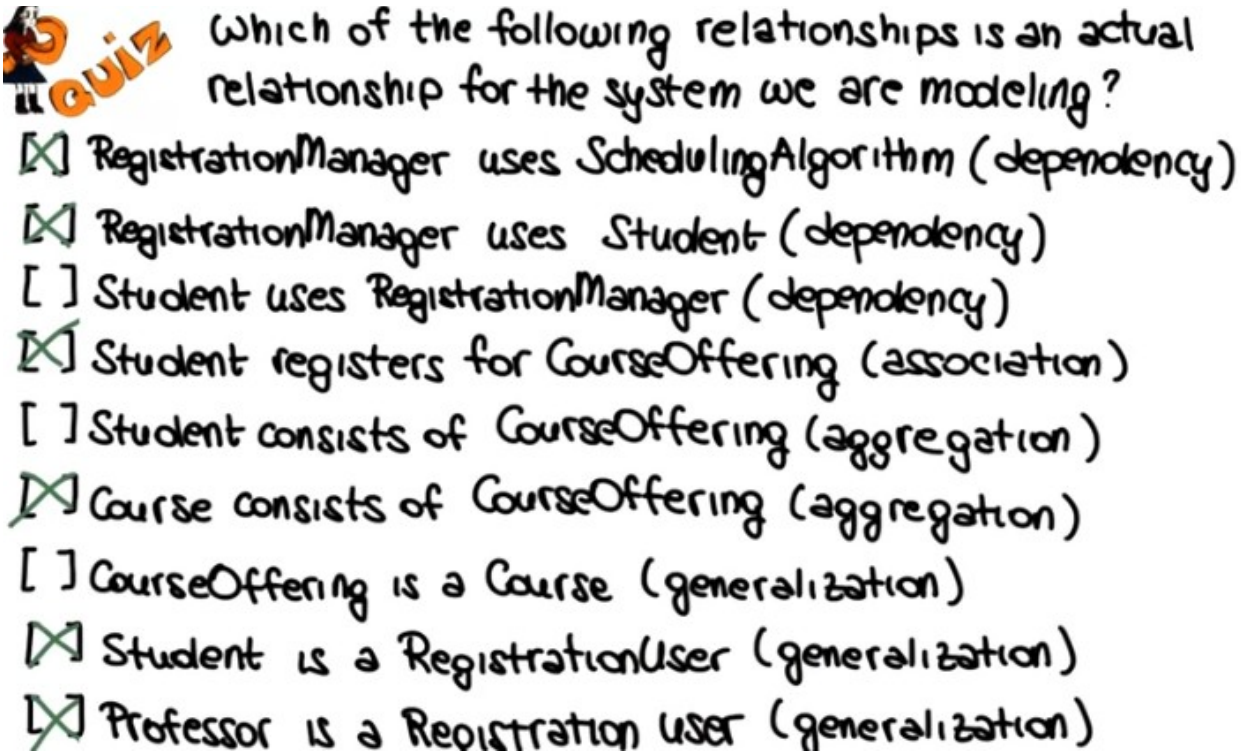


16. And that's exactly what we're going to look at next, relationships in the class diagram, how they're represented and what they mean. First of all relationships as the name says, describe interactions between classes or between objects in my system. And we will describe three main types of relationships. The first one is called a Dependency relationship. And we can express that as X uses Y and we represent it with a dashed directed line. So when we have such a line between two classes that means that the first class uses the second one. And we're going to provide an example of a dependency in a minute. The second type of relationship is an association that can also be an aggregation. We'll see what the distinction is. But basically, what this means is that we can express that as a X has a y. So x contains a y. And if it is in association, we indicate it with a solid undirected line. If it's an aggregation, we indicate it in the same way, but with a diamond at one of the ends. Finally, the third type of relationship is what is called Generalization. And this can be expressed as x is a y. So this is the relationship that expresses inheritance. Specialization between two classes. It's represented with a solid directed line with a large open arrow head at the end. Going from the more specialized class to the less specialized class. So going from the subclass to the super class. So now let's look at each relationship in more detail using our example, our course management system.



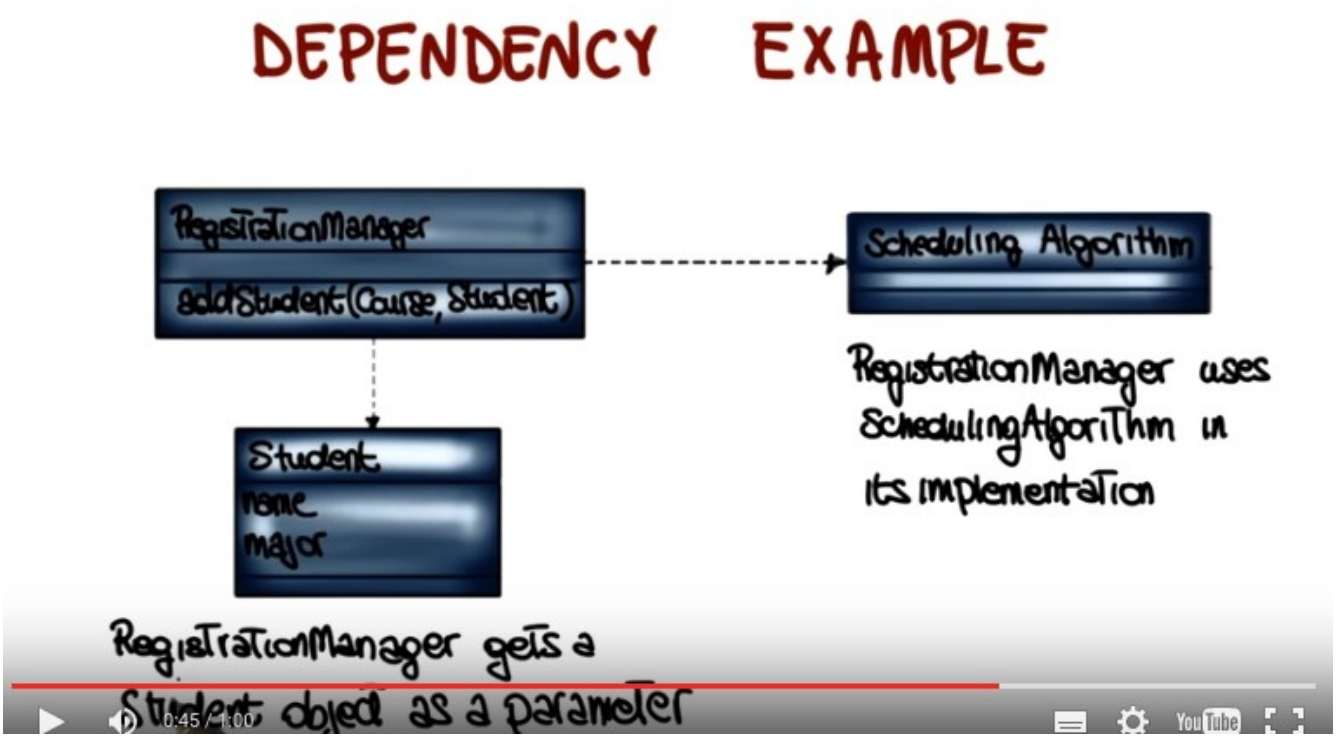
上圖中, Associations 即那條直線, Aggreations 即那條帶菱形的直線. 且 X has a Y 是僅指 Aggregations, 不指 Associations.

17. Before doing that, though, now that we discuss the different kinds of relationships among classes in the class diagram. I would like to ask you to look at a list of relationships that I'm providing here and mark the relationships that you think actually hold for the classes in the system that we are modeling. So here I have a list of possible relationships for each relationship. I'm first defining what the relationship is and then what kind of relationship that is, for example, for the first one I'm saying that the registration manager uses the scheduling algorithm, which is a dependency relationship. And similarly for the other ones. So like for you to go back to the example, look at the classes that we defined, think about the requirements, and identify which ones of this relationships you think hold in the system.

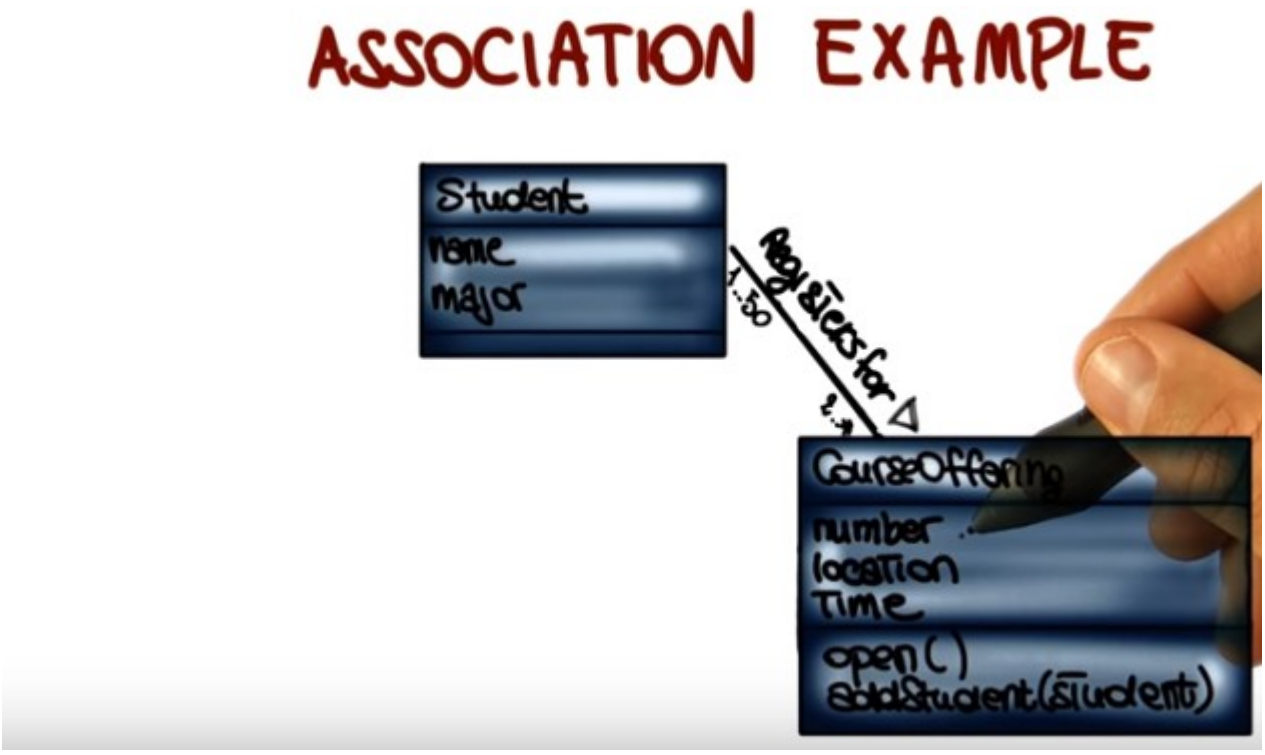


18. So, I'm going to start by marking the relationships that actually hold in the system. Which are this ones. And then what I'm going to do, I'm going to explain this answers. Not here but in the next part of this lesson. By looking at the different relationships in the context of our example. Which will make the explanation much clearer.

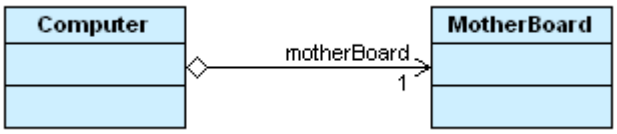
19. So let's start with the dependency example. A dependency, as we said, expresses the relationship between a supplier and a client that relies on it. There is a dependency because changes in the supplier can affect the client. Here in this example I am showing that a dependency example involving the registration manager and the scheduling algorithm. As you can see the, the dependency is indicated with a dashed line pointing from the client to the supplier. And here it's pretty clear why the RegistrationManager is dependent on the Scheduling Algorithm. It's because the RegistrationManager uses this Scheduling Algorithm. And therefore, if the Scheduling Algorithm changes, the RegistrationManager might be affected by that change. Another less obvious example is the dependency between the Registration Manager and the Student. In this case, because the Registration Manager gets a Student object as a parameter here there is a dependency between the two. Again, if the Student class were to change the Registration Manager might be affected because it's relying on the Student for it's behavior.



20. The next example of relationship we're going to look at is the association relationship. This is a relationship in which objects of one class are connected to objects of another. And it's called an has a relationship. So it means that objects of one class have objects of another class. So let's see what that means. Let's do it by considering two classes in our example system. The student class and the course offering class. In this case, there is an association between the student and the course offering, because the student is registering for the course offering. So, in a sense, the course offering has students. Contains students, to indicate this fact we add a solid line between the student class and the course offering. And the fact that having a solid line doesn't really tell us much about the nature of the relationship, so to clarify such nature we can use what we call adormants that we can apply to associations we can add to associations to clarify. Their meaning. In particular we can add a label to an association and the label describes the nature of the relationship. In this case, for example, it clarifies that the student registers for course offering. We can also add a triangle to further clarify the direction of the relationship. So in this case, the triangle will indicate that it's the student That registers for the course offering, and not the other way around. Another important adorment or limitation that we can put on an association, is multiplicity. Multiplicity defines the number of instances of one class that are related to one instance of the other class. We can define multiplicity as either end of the relationship. In this case, for instance, we can say that if we look at the student, the student can register for two or more course offerings. Whereas, if we look at the course offering, we can say that each course offering can have or can enroll between 1 and 50 students. So as you can see by adding a label, a direction, and multiplicity, we make it much clearer what the relationship is and what it means and what are its characteristics.

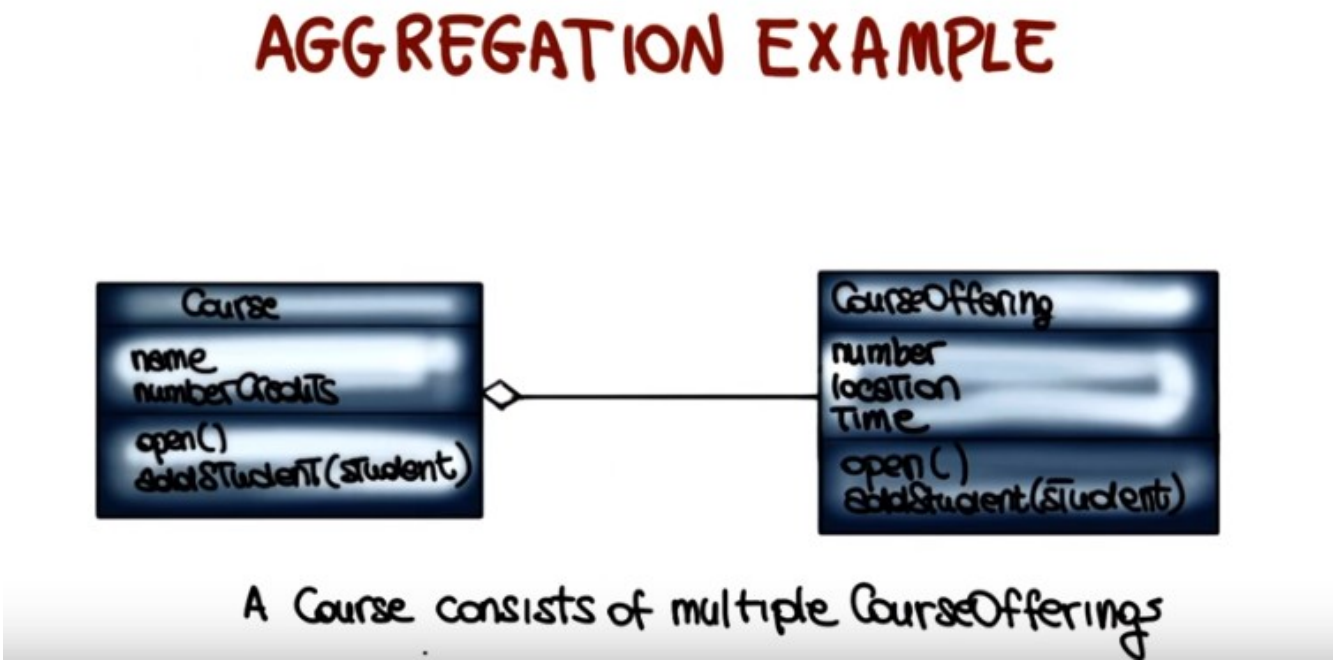


關於 aggregation, 最前面給出那個網頁講得好:
Aggregation is a relationship where one class is part of another class. In basic aggregation, the class that forms part of the whole class can exist independently, so the life of an instance of the part class is not determined by the whole class. Basic aggregation is represented using an empty diamond symbol next to the whole class. In the example, a computer in a warehouse contains a motherboard, but although the motherboard is part of the computer, it can exist as a separate item. In this system, Computer knows about Motherboard but Motherboard doesn't know about Computer, so the aggregation is unidirectional. In a program, this relationship could be implemented as a member variable in the Computer class which is a reference to a Motherboard class.

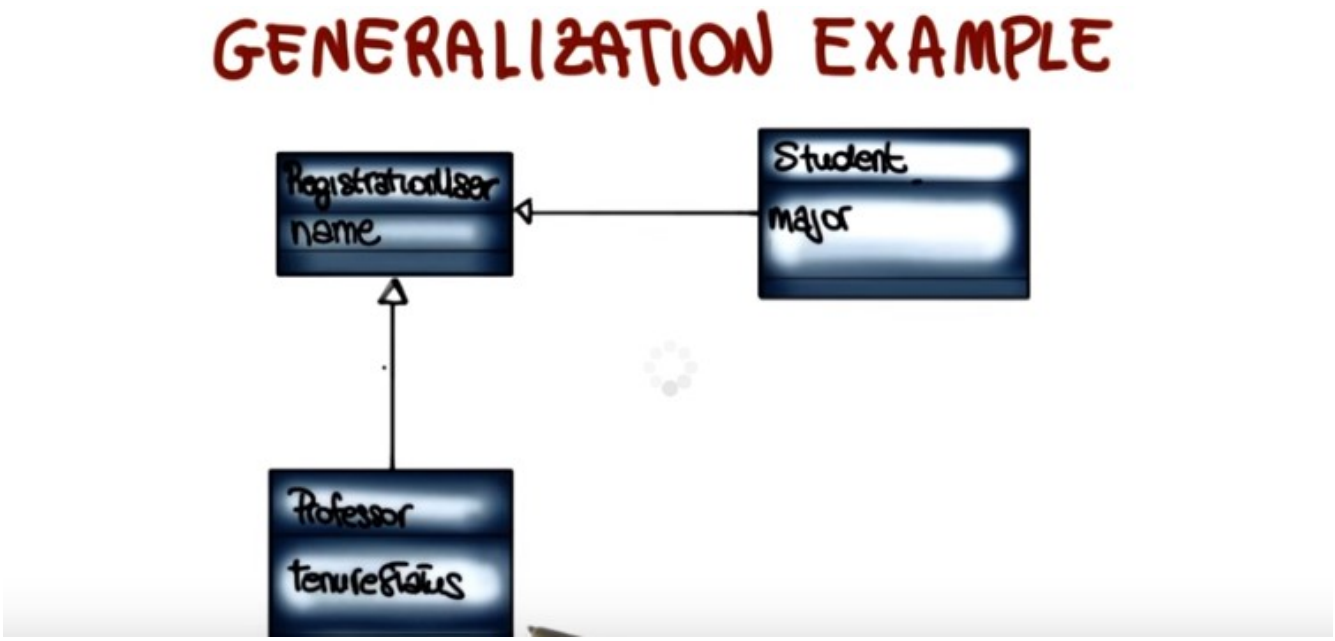


以下 Alex 給的例子不是操蛋, 後面的課(P3L2)用到了一個類似的關 aggregation(Title 和 Item)
As we saw when we introduced relationships, there is a different kind of association, kind of a specialized one, which we call aggregation. So here we're going to look at an example of an aggregation. So first of all what is an aggregation? An aggregation is a relationship between 2 classes in which 1 represents a larger class like a whole which consists of smaller classes which are the parts of this whole. So lets look at an example in the context of our system, lets consider. A course and the course offering. And in this case, we can see that the course consists of multiple course offerings. So in a sense, a course is a whole and the course offerings are the parts of this whole (我的理解: 同一門課, 可以有不同的 course offering, 這些 course offering 的時間和地點可能不同). So this a perfect case in which we will use an aggregation to express this relationship. So we will add. A solid line with a diamond on the side of the whole class (即 diamond 指向的是 whole class) to indicate that the course of multiple

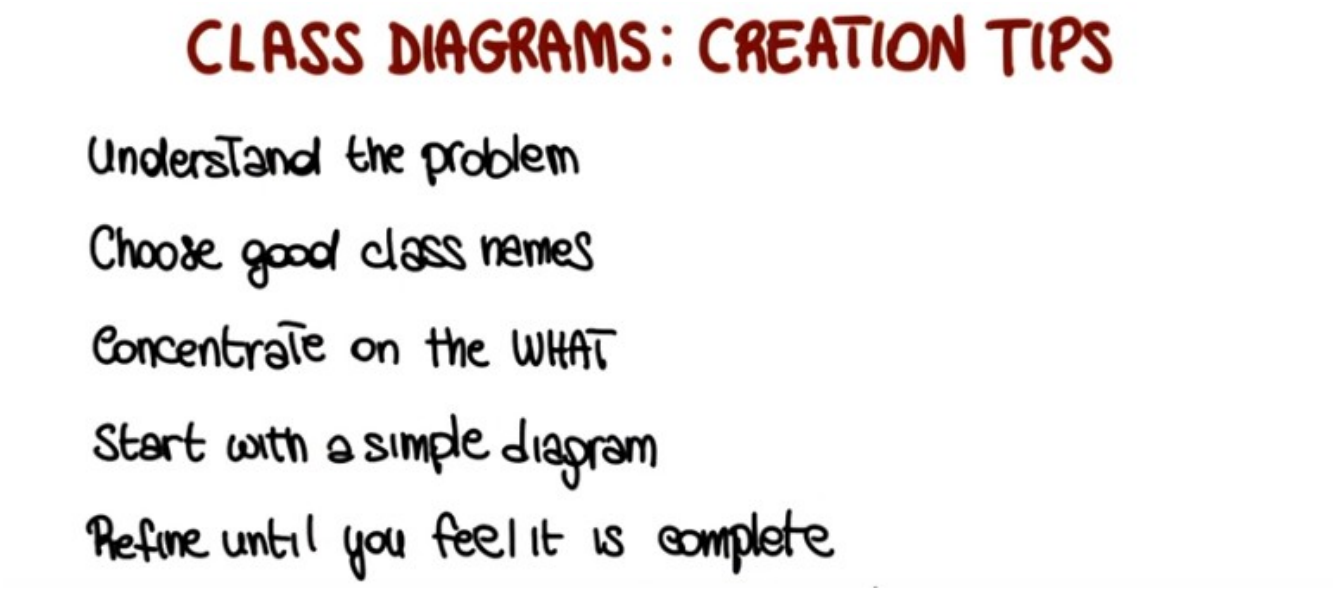
course offerings, and as we did for associations even though we are not going to do it for this specific example, we could also in this case add multiplicity information on the aggregation to indicate how many classes of the two types are involved in the relationships.



21. The third type of relationship that we saw, is Generalization. Generalization is a relationship, between a general class, which we normally call super-class and the more specific class, a class the refines the super-class and that we normally call sub-class. It's also known as a kind of or is a relationship because we can say that the subclass is a super class and it's expressed with a solid line with a big arrow head at the end (arrow 指向的是 superclass). So let's see an example of that. In this case I'm going to indicate. Two of this kind of relationships. The first one involving the registration user and a student. And the second one, a registration user and the professor. So, basically what we're sh, showing here, is a typical case in which the registration user is a more general concept. And the student and the professor. So both the student and the professor are registration users. So there is a relationship, the professor is a registration user and the student is a registration user. And therefore we indicate that using the generalization relationship in our class diagram.

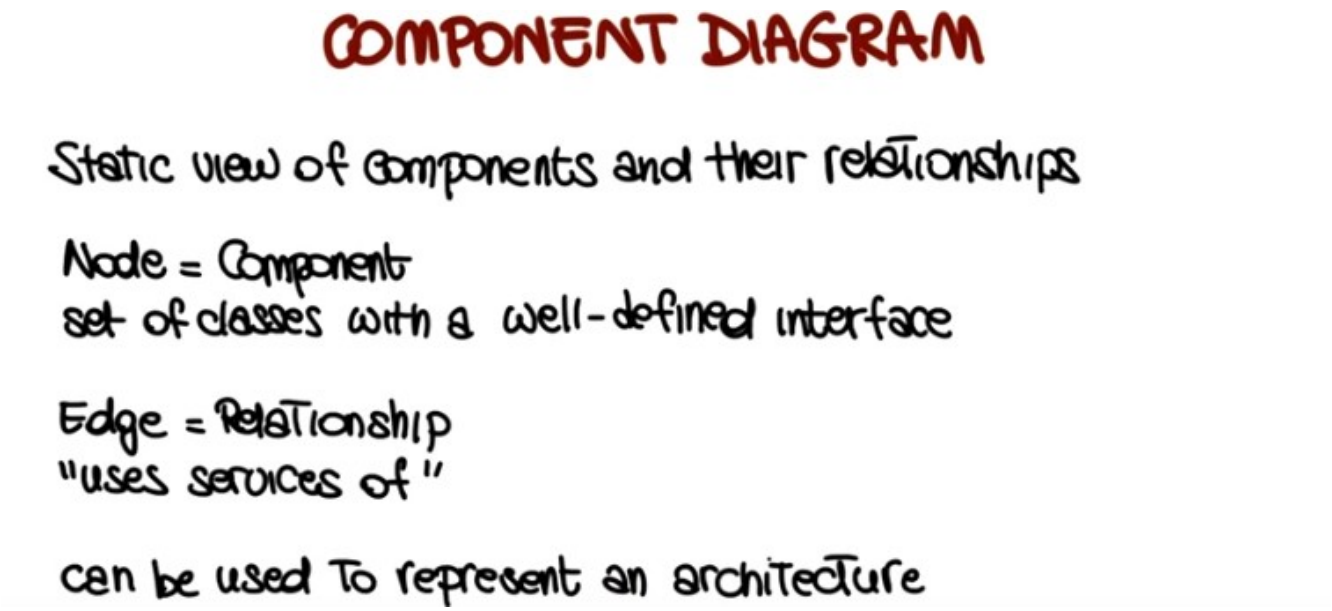


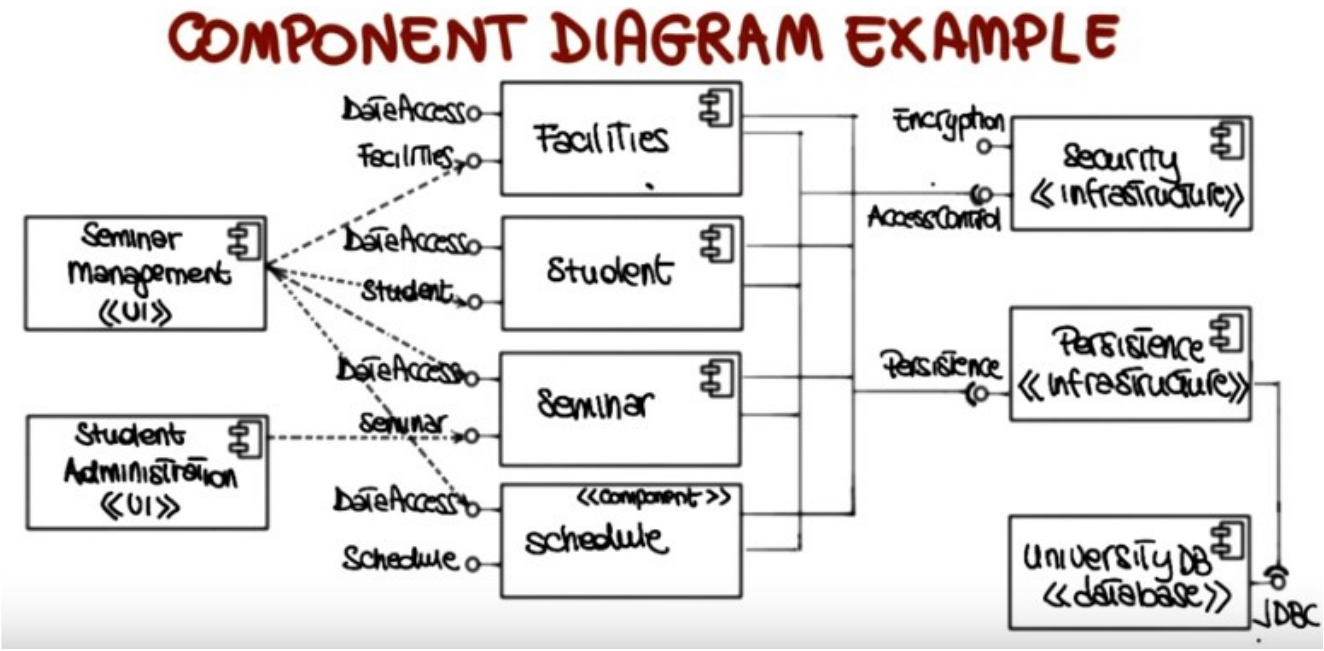
22. The last thing that I want to mention about class diagrams is some creation tips. So something I know based on my experience and the experience of others, I can recommend to do when creating a class diagram. So the first tip is to understand the problem. So take the time to look at the description of the system that you have to build, to make sure that you understand the domain. That you understand what you are supposed to build. Because that is going to save you time later. It's going to help you identify from the beginning, a more relevant set of entities in the description of the system. This one might seem trivial but is very important to choose good class names. Why? Because class names communicate. The intent of the class, and clarify what the class refers to. So having good class names allows you, makes it easier, to create the mapping between the real-world object and the entities in your model. And of course, it also makes it easier to understand the system, after the system is built. Third tip, concentrate on the what. So here, in the class diagram, we're just representing the structure of the system. We're representing what is in the system. What are the entities? What are the characteristics of the entities? We are not focusing at all, on how things are done. So, be careful. Don't think about the how, just think about the what. Proceed in an itinerary way. So, start with a simple diagram and refine it. There is no need to identify, right away, all of the details of the system you need to build. It is much easier to look at the description, identify an initial rough class diagram and then refine it, because in this way, you'll also gather more understanding of the system as you build it, and you'll most likely end up with a better product at the end. And if you proceed in this way, then make sure to refine until you feel the class diagram is complete, until you feel that you represent the system that you're supposed to build. So your final goal should be to have a class diagram that is complete. So it represents all of the relevant entities in the system and their characteristics, and it's correct so it represents them in the right way



23. There's two more structural diagrams that I want to mention before we move to the behavioral ones. The first one's the component diagram. A component diagram is a static view of components in a system and of their relationships. More precisely, a node in a component diagram represents a component where a component consists of one or more classes with a well-defined Interface. Edges conversely indicate relationships between the components. You can read this relationship as compotent A uses services of component B. And that's the component diagrams can be used to represent an architecture, which is a topic that we will cover extensively in the next mini-course. So let's illustrate this with an example. So, what I'm representing here is a component diagram for our example system, the course management system. And as you can see, it's slightly more complex than the other diagrams that we saw. But there's really no need to go through all the steps and all the details. Important thing is to point out some key aspects of this diagram. So the first one is that these rectangular nodes are the nodes in the system, so are my components. For example, student is a component, schedule is a component, and so on. And as far as edges are concerned, I'm representing two kinds of edges. The first kind of dashed edges which were part of the original uml definition and indicate use. So an edge, for example, between this compnent and this compnent indicated that the seminer management uses the facilities component. More recently, in UML two, a richer representation was introduced, which is the one that I'm also showing here. So if we look at this part of the diagram, you can see this sort of you now, lollipop(棒棒糖) socket(穴,插座) representation. And in this case, what this represents, is that a lollipop indicates a provided interface. So an interface that is provided by the component. So, for example, this security component provides encryption capabilities. The socket, conversely, indicates a required interface. So, for example, in this case, it's saying that the facilities component is needing access control capabilities, which, by the way, is provided by the security component. So in a sense this sockets and lollipop indicate interfaces between a provider of some of functionality, and the client of that functionality and you can look at those as basically APIs. So sets of methods that provide a given functionality. To give you another example, if we look at the persistence components the persistence component provides. Are surprisingly persistent serivces. And those persistent services are required by several other components in the system. And in turn, the persistent components rely on the University database component to provide such services. So, there's the University DB components provide these sort of low-level database services that are used by the persistence component To in turn provided services. Last thing I want to note is that components or relationships can be annotated(如<<UI>>), so, for example if we look at the seminar management and the student administration components we can

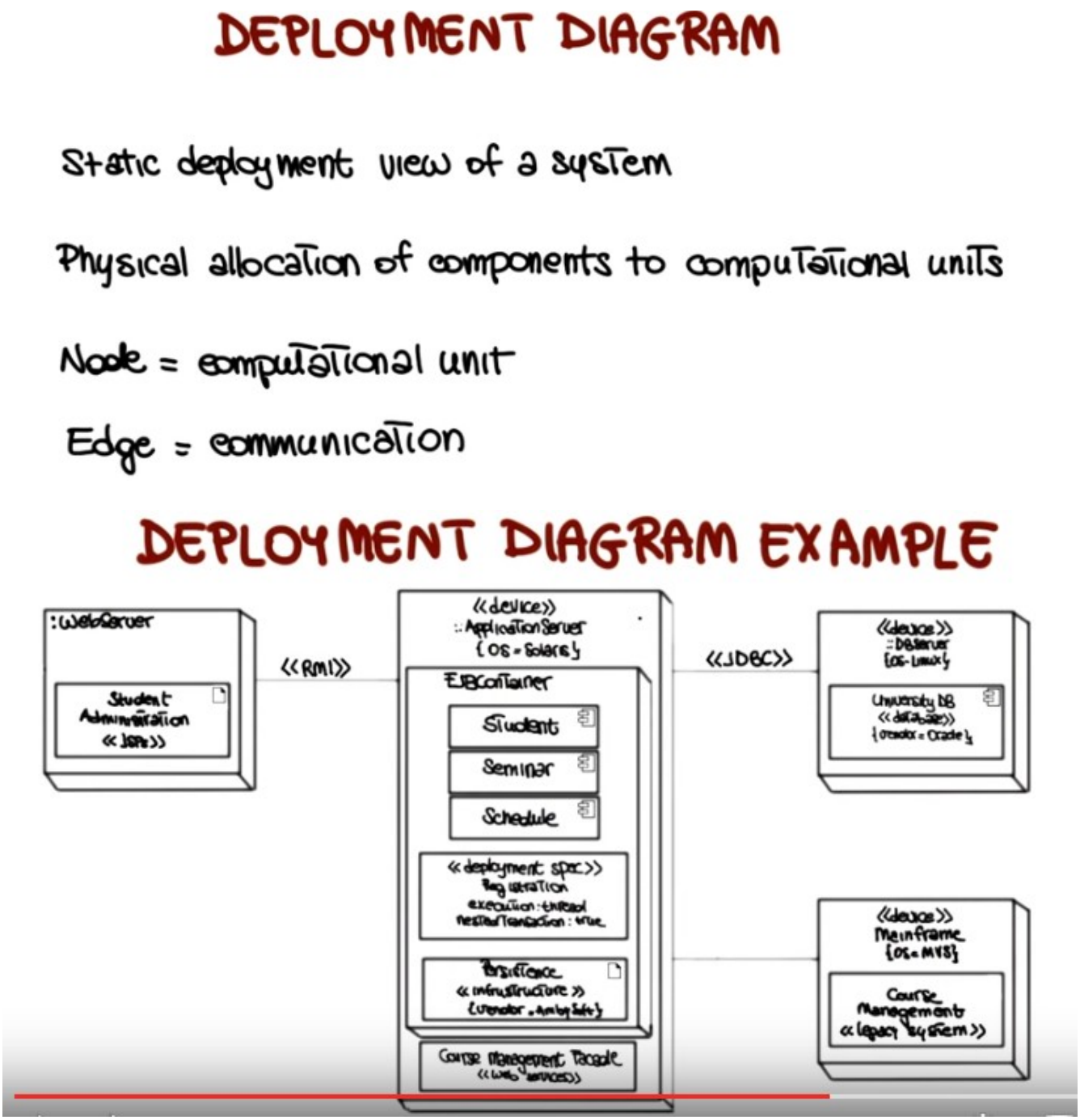
see that they are annotated here to indicate that they are user inferfaces. So that's all I wanted to say on the component diagrams, but again the key piece of information is that they represent components in the system where a component consists of one or more classes indicate the interfaces that these components provide or require. and describe the interactions between these components.



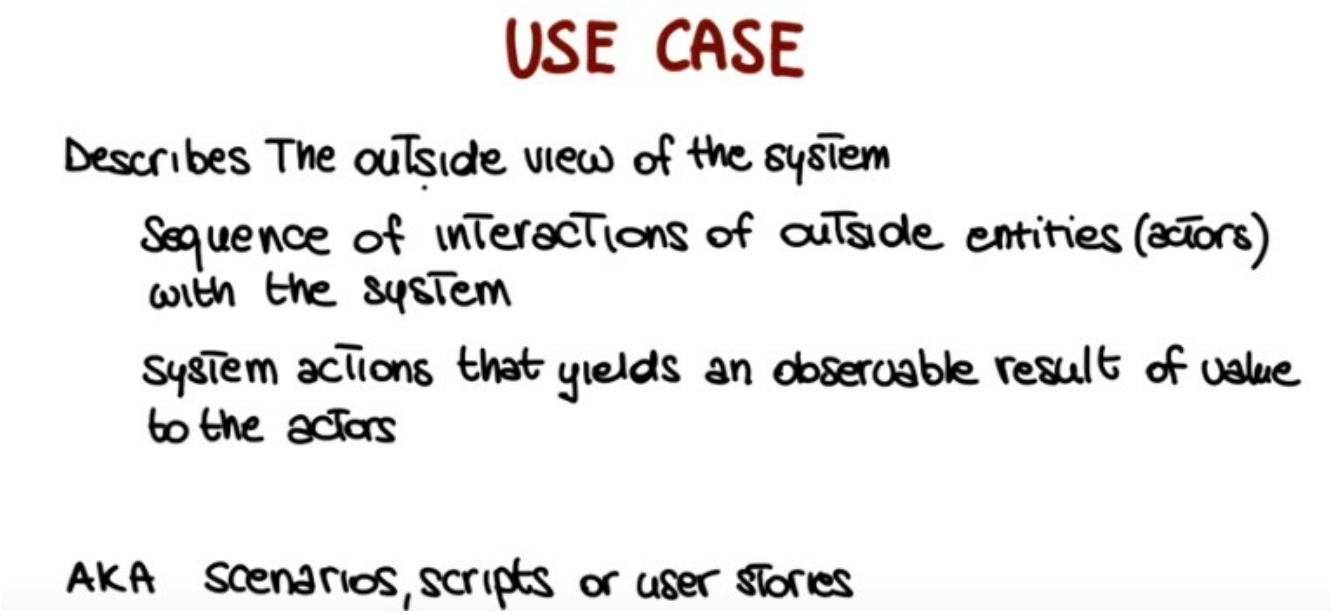


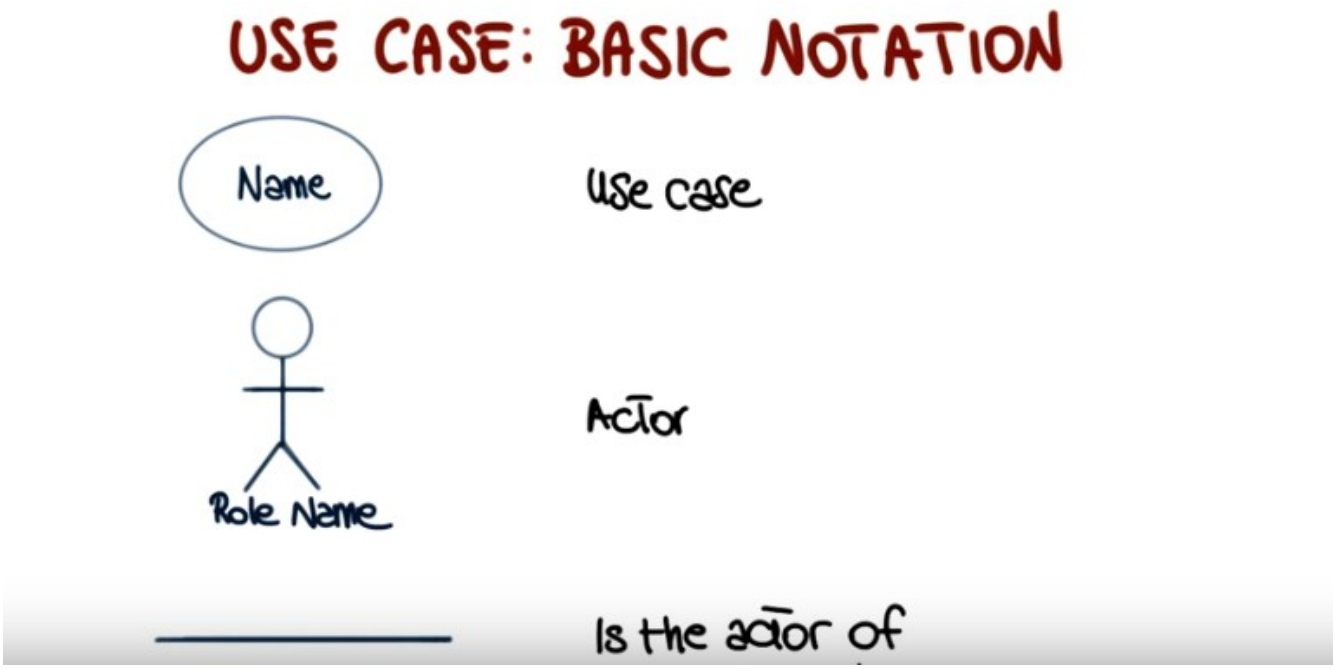
24. The last UML structural diagram I want to discuss is the deployment(部署;配置) diagram. The deployment diagram provides a static deployment view of a system, and unlike previous diagram, it is about the physical allocation of components to computational units. Think, for example, of a client-server system in which you'll have to define which components will go on the server and which component will go on the client. For deployment diagram, the nodes correspond to computation unit; for example, a specific device. And the edges indicate communication between these units. Also in this

case, I'm going to illustrate deployment diagrams using an example for our course management system. And also in this case, I'm going to use a slightly more complex diagram than usual. But I don't want you to look at all the individual details. Instead, I would like to focus on a few main aspects. So, if you look at this diagram, there are three things that you should clearly see. First, you should see how the system involves four nodes, a web server, an application server, a DB server, and a mainframe. Second, you should see which components are deployed on which nodes. For example, the student component is deployed on the application server. And finally, you should see how the nodes communicate with one another. For example, you can see that the application server and the university database communicate using a JDBC protocol.



25. We now discuss UMLs behavioral diagrams. Those diagrams that have to do with the behavior, the dynamic aspects of the system, rather than the static ones. The first behavioral diagram I want to discuss is a very fundamental one, the Use Case Diagram. So, let's start by seeing what a Use Case is. A use case represents two main things. First the sequence of interactions of outside entities which is what we normally call actors with the system that we're modelling and the second thing is the system actions that yield an observable result of values to the actors. And basically these two things, and nothing else that the outside view of the system. So the view of the system in which we look at the interaction between this system, and the outside world. If you want to parallel, think about designing a house. Considering how you would use the house. And you might have seen use cases called with different names. So for example, they're also called scenarios, scripts or user stories, but in the context of UML, we'll call the use cases, now let's look at the basic notation for a use case, which is fairly simple. We have a use case which is represented by an oval, with a name, which is the name of the use case. We have an actor, which is represented by this icon and is normally identified by a role name. And finally we have an edge which is a solid edge that connects actors and use cases and indicates that an actor is the actor of a given use case and just for completeness let me note there are some additional notational elements but now for simplicity we'll just use these ones.





26. Now, let's look at use cases in a little more detail. And start by defining exactly what an actor is. An actor represents an entity, which can be a human or a device, that plays a role within my system, so that interacts with my system. It's some entity from the outside world, with respect to my system, that interacts with my system. It is important to clarify that an entity can play more than one role. For example, you might have somebody working in a bank that can be both an employee of the bank, or a customer of the bank, depending on how it interacts with the banking system. And, obviously, more than one entity can play the same role. Using the same example, we can have both an employee of the bank and just a regular customer, playing the role of the customer. So again, it all depends on what the entity does, how the entity interacts with the system, what kind of functionality of the system the entity uses. And finally, actors may appear in more than one use case. So it's fairly normal for the same actor to interact with the system in different ways. And therefore, to appear in more than one use case. Just think about the use cases in scenarios of usage. If the same actor can interact with the system in different ways, that actor will appear in multiple use cases. Now let's go back to the description of our course management system, and see how we can identify actors in the system. And as we did for the class diagram before, I encourage you to stop the video and try to identify the actors in the system yourself, before I do it. If we look at the description, we can see that, for example, the Registration Manager is clearly an actor for the system. Students are actors for the system. Professors are actors for the system. And notice that we're not doing the same thing that we were doing when identifying classes. Here we're identifying entities that are from the outside world, and have an active role in interacting with my system. Again, Registration Manager, that we will just call registrar for simplicity, students, and professors. So once we have identified the actors for our example, we can simply draw them, using the notation that we just introduced. So we have the registrar, and notice how for every actor we clarify the role that the actor plays. We have the professor, and we have the student. So here, these are the three actors that we identified for our system.

USE CASE: ACTOR

Entity : human or device

Plays a role

- An entity can play more than one role
- more than one entity can play the same role

May appear in more than one use case

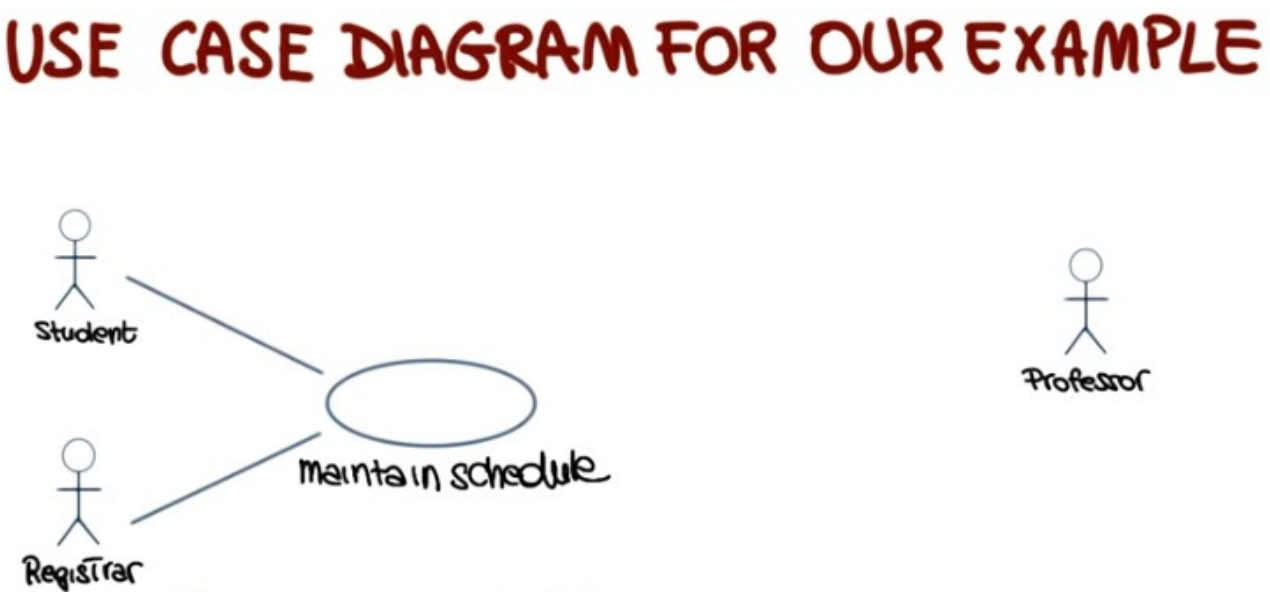
ACTORS FOR OUR EXAMPLE

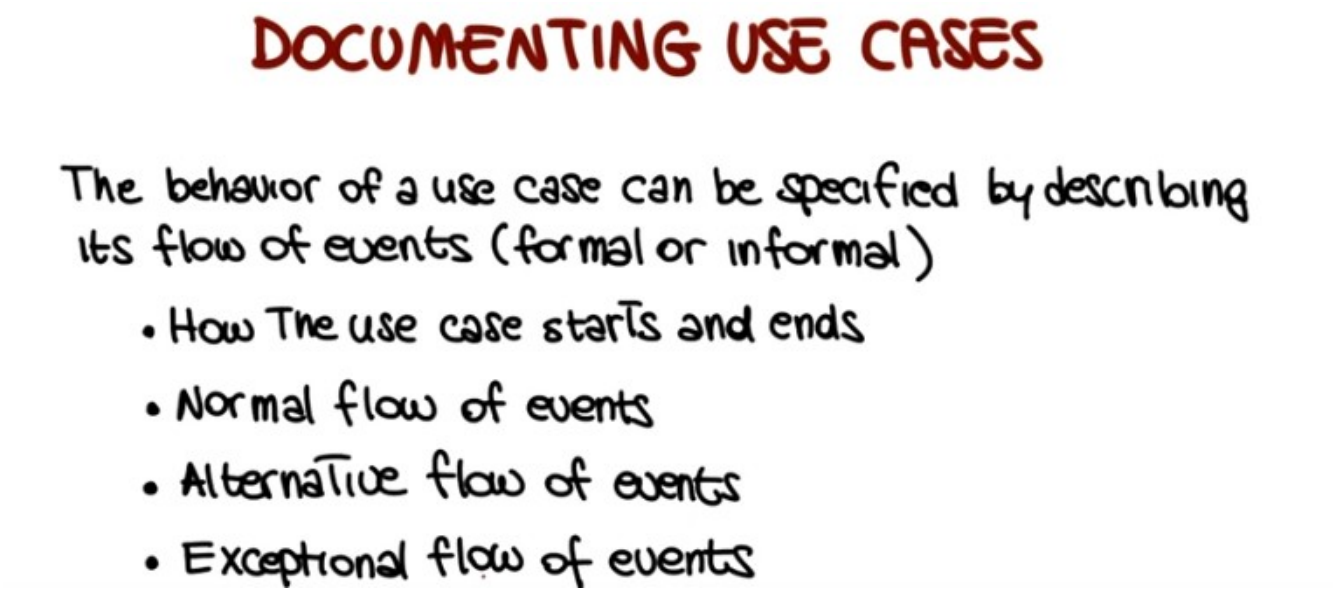
1 The Registration Manager sets up the curriculum for a semester using a scheduling algorithm
2 One course may have multiple course offerings
3 Each course offering has a number, location, and time
4 Students select 4 primary courses and 2 alternative courses by submitting a registration form
5 Students may use the system to add/drop courses for a period of time after registration
6 Professors use the system to receive their course offering rosters
7 Users of the registration system are assigned passwords which are used at logon validation

ACTORS FOR OUR EXAMPLE

27. Now if you want to build a use case diagram for our example, we have to add the use cases for

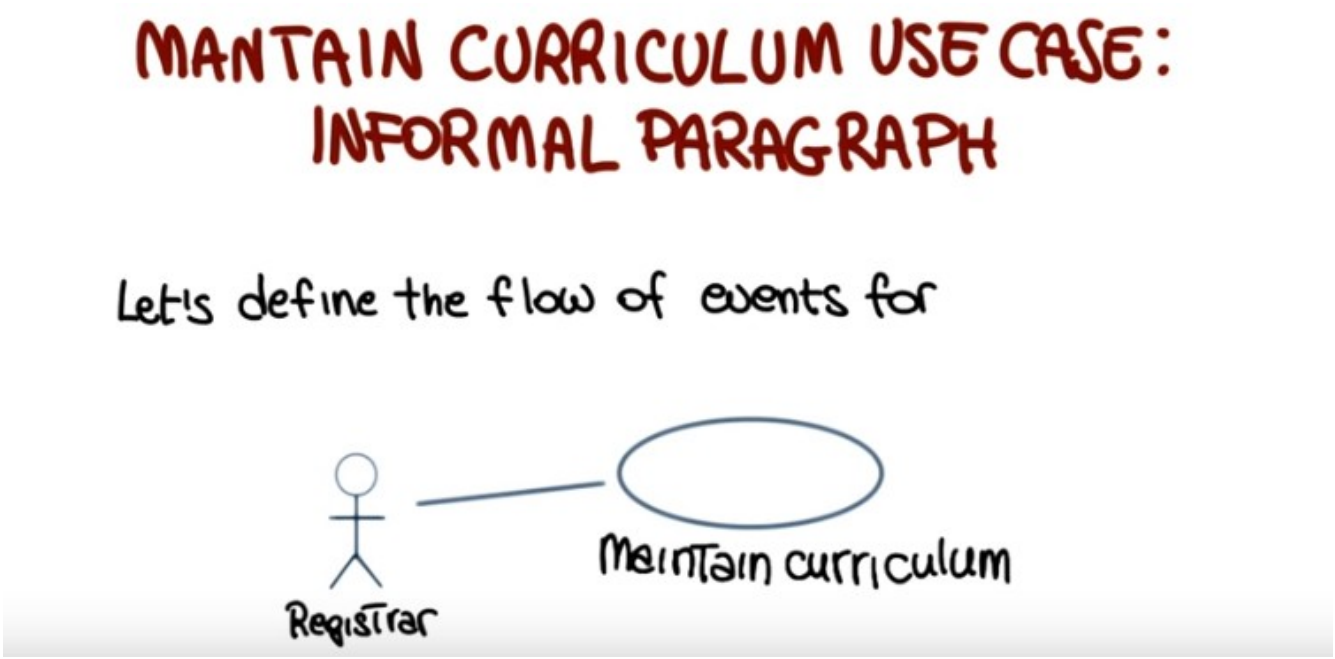
these different actors. For instance, if we consider the student and the registrar, they might be both interacting with the maintain schedule system, the registrar by updating the schedule and the students by using the schedule that has been updated by the registrar. As you can see, different roles for the same use case. Another possible use case is the request course roster. And on this case, the professor will request the roster by interacting with the system. We will continue in this way by further refining and by further adding use cases as we identify possible interactions of the actors that we identified with our system. So in summary, what the use case diagram is doing is to show the actors and their interaction with the system through a set of use cases. At this point, it should be pretty clear that sure, this gives us an idea of the interactions but we don't really know how these interactions occur. So there is one piece that is missing, which is how do we document the use cases, how do we describe what happens and what these interactions actually are. And that's exactly what we're going to discuss now, how to document use cases. So the behavior of a use case can be specified by describing its flow of events. And it is important to note that the flow of events should be described from an actor's point of view, so from the point of view of the external entity that is interacting with my system. So the description should detail what the system must provide to the actor when the use case is executed. In particular, it should describe how the use case starts and ends. It should describe the normal flow of events, what is the normal interaction. And in addition to the normal flow of events, it should also describe possibly alternative flows of events. For example, in the case in which there are multiple ways of accomplishing one action or performing a task. And finally, it should also describe exceptional flow of events. For example, assume that you are describing a use case for withdrawing money from an ATM. You may want to describe the normal flow of events in which I insert my card, I provide my pin and so on. An alternative one in which, in addition to withdrawing cash, maybe I'll also first ask for some information about how much money is in my account. And finally, I may want to also describe an exceptional flow of events in which I get my pin wrong and, therefore, I'm not able to perform the operation. One more thing I want to mention, when we talk about documenting use cases, is the fact that the description of this information can be provided in two main ways, in an informal way or in a formal way. In the case of an informal description, we could just have a textual description of the flow of events in natural language. In the case of a formal or structure description, we may use, for example, pre and post conditions, pseudo code to indicate the steps. We could also use the sequence diagrams, which is something that we will see in a minute.

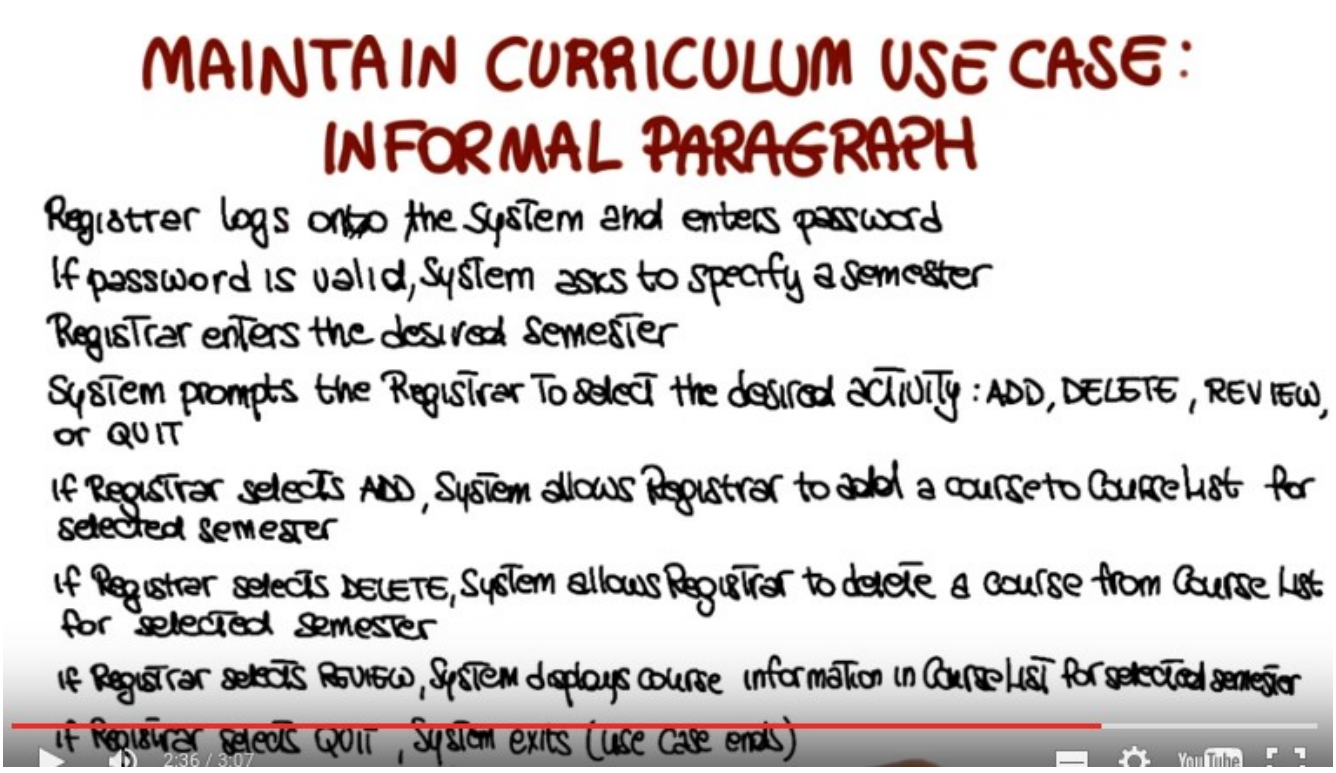




28. So, as we did for the previous cases, now let's look at an example. Let's consider a specific use case, maintain curriculum, in which we have the registrar that interacts with the system to do operations for maintaining the curriculum. And, let's define the flow of events for this use case. To do this, we're going to go back, as usual, to the description of our system. So this is the one that you already saw several times, but I would like for you to do something. I would like for you to stop the video, look back at the spec, the one that is shown here. And write on your own, what you think is the informal flow of events that categorizes the interaction of the registration manager with the system. And it is very important that you keep in mind something as you're doing that. You should keep in mind that, as it always happens, when extracting requirements from an initial specification, in particular an informal one like this one, a high-level one, you will have to be able to read between the lines and fill in the blanks. That is, you have to provide the information for the missing parts using your domain knowledge. So try to do that exercise. Read the description, and see how you will define the steps, the flow of events for the maintain curriculum use case. If you're done with that, now let's see the possible informal paragraph that describes that flow of events. And the one I'm providing now is just one possibility, based on my experience and based on the way I see this possible flow of events. So yours might look different, of course. In my case, because the description was measuring the fact that every user has got a log-in and a password. I decided that the first step should be that the registrar logs onto the system and enters his or her password. As it normally happens with password protected systems, if the password is valid, the registrar will get into the system. And the system at this point should ask to specify a semester for which the maintain curriculum activity has to be performed. The registrar will therefor enter the desired semester. The interface I envisioned is one in which the system will prompt the registrar to select the desired activity. Add, delete, review, or quit. And if the registrar selects add, the system will allow the registrar to add a course to the course list for the selected semester. Similarly, if the registrar selects delete, the system will let the registrar delete a course from the course list for the selected semester. And again similarly, if the registrar selects review, the system will simply display the course information in the course list for the selected semester. And finally, if the registrar selects quit, the system will simply exit and our use case will end. So, again, there's the main knowledge that goes into this. But this is a good example of how you can refine the initial description to identify these scenarios that then you will use to specify and implement your system. And as we discussed a few minutes ago, we provided the information that is requested for for use case. How the use case starts, by logging into the system. And how it ends, by selecting quit. We described the normal flow of events.

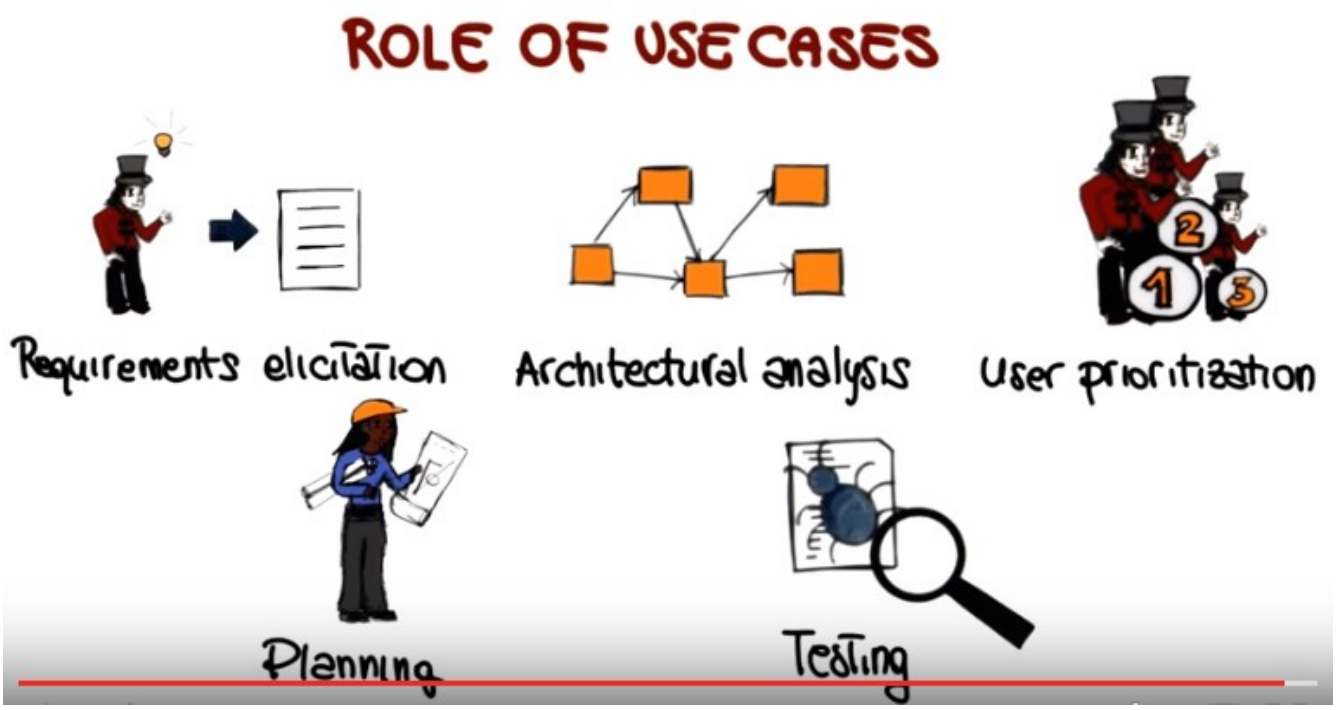
And, of course, these flow of events could be improved, because right now even though we described how the use case starts and ends, we just described one possible flow of events. But there's many alternative ways we could provide and we do not describe any exception of flow of events. So this could be the starting point for multiple use cases, or for use cases just richer and contains more information, more steps to a richer flow. But you should have gotten the idea of what a use case should be.





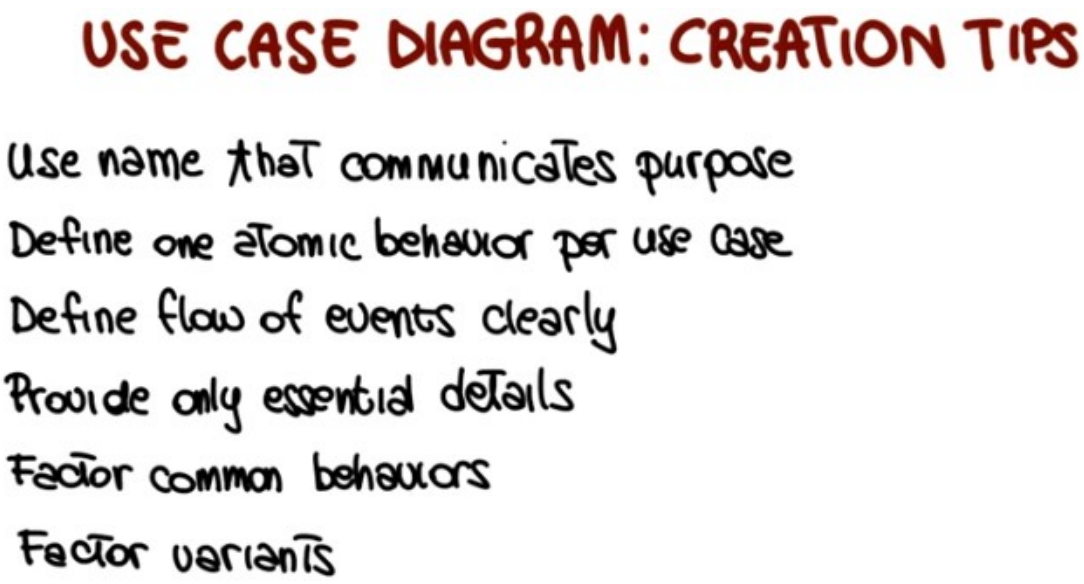
29. As I mentioned when we started talking about use cases, use cases are fundamental in UML, and in

general. So, now I would like to discuss why they're so important and what are the different roles that use cases can play. The first obvious one is for requirements elicitation. It is much easier to describe what the system should do if we think about the system in terms of scenarios of usage. Rather than trying to describe the whole functionality of the system at once. So, use cases can help performing a more effective requirement solicitation. As we will see when we discuss the unified software process, they can be used for architectural analysis. So, use cases are the starting point for the analysis of the architecture of the system that can help identify the main blocks of the system. And therefore, can help define in the initial architecture. And as I said, we'll talk more extensively about that. They can be used for user prioritization. For example, imagine to have multiple actors in the system, and you might want to prioritize some of them. For instance, using again the banking system example, we might want to first provide functionality for the administrators of the bank. And only in a second time provide functionality for the customers, because of course, if the administrator cannot perform any operation, the customers cannot use the system. So again, they can be used to prioritize the users. Or the actors, and therefore define which part of the system should be built in which order. Related to this point, they can be used for planning. If I know which pieces of functionality I need to build and in which order, I can better plan the development of my system. And again, we will see how this becomes very important in many different software life cycles. So, both in the unified software process, for instance, but also in more agile development processes. And finally, use cases can be used for testing. If I have an early description of what the system should do, what are the main pieces of functionality of the system. And I know how the interaction between the actors and the system is, I can easily define test cases, even before writing the code, even before defining my system. And when we discuss testing, we will get back to this and talk a little more extensively about this, as well.



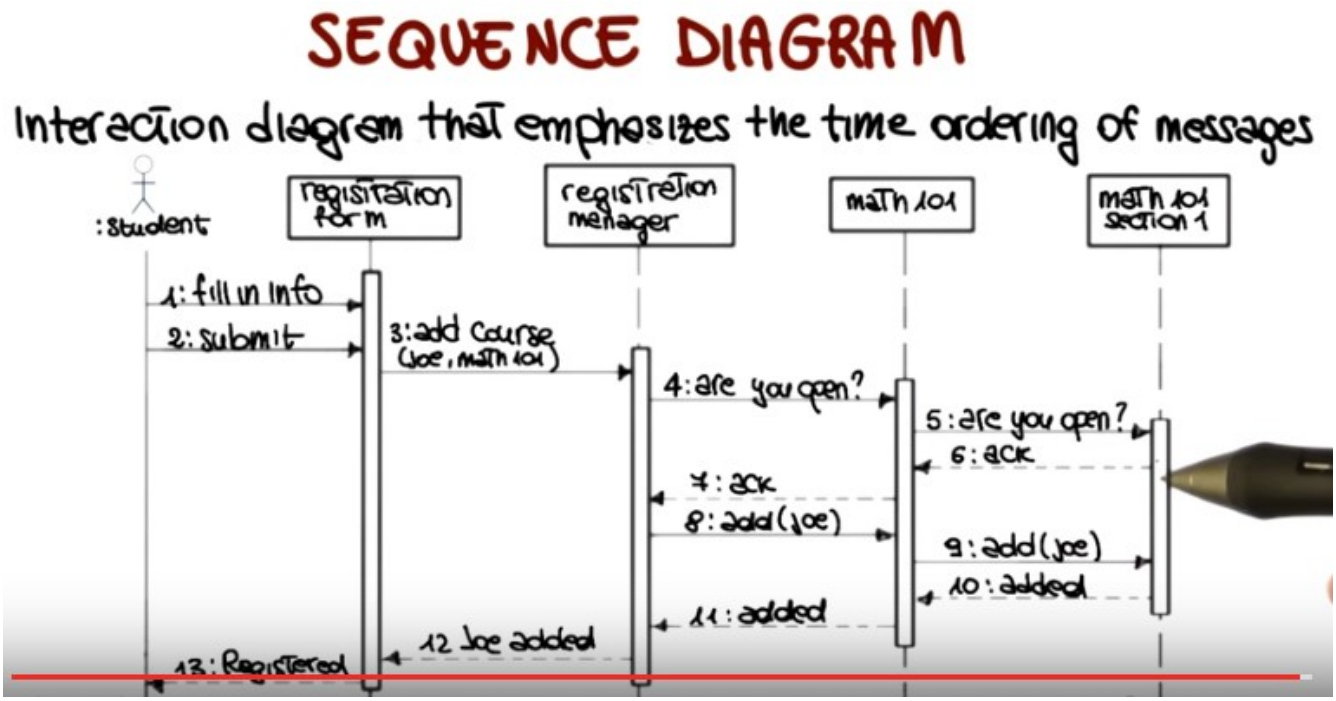
30. Now, as we did for the class diagram, let's look at some creation tips for use case diagrams. The first tip is that when you define a use case, use a name that communicates purpose. It should be clear what the use case refers to by just looking at the name of the use case. Second tip is to define one atomic behavior per use case. So try not to put more than one specific scenario into a use case. Why?

Because these will make the use cases easier to understand and better suited for their roles that we just discussed to define these cases, to do planning, to define an architecture and so on and so forth. Define the flow of events clearly. So again, do it from the perspective of an outsider. An outsider should be able to read the description of the flow of events and understand exactly how the system works or how that specific piece of functionality works. As we suggested for the class diagram, provide only essential details. So there is no need to provide all the nitty gritty details about the use case, just provide enough details so that the use case is complete and understandable. And finally, even though we didn't cover that, there is a way to factor common behaviors and factor variants when defining use cases. So I will encourage you to look at how to do that. For example, by looking at the additional UML documentation and to try to factor out this common behaviors and variants. Typical example would be a system that requires login, like the one that we just discussed, will probably require an initial login step for each use case. It is possible that instead of describing the same steps, or same sub-steps, for each use case, you can factor that out. And create a use case that you should then include in your own use cases. As I said, we didn't cover this for simplicity, but feel free to further read about UML and to see how you can actually factor out behaviors and factor variants. Which can be very useful in practice.



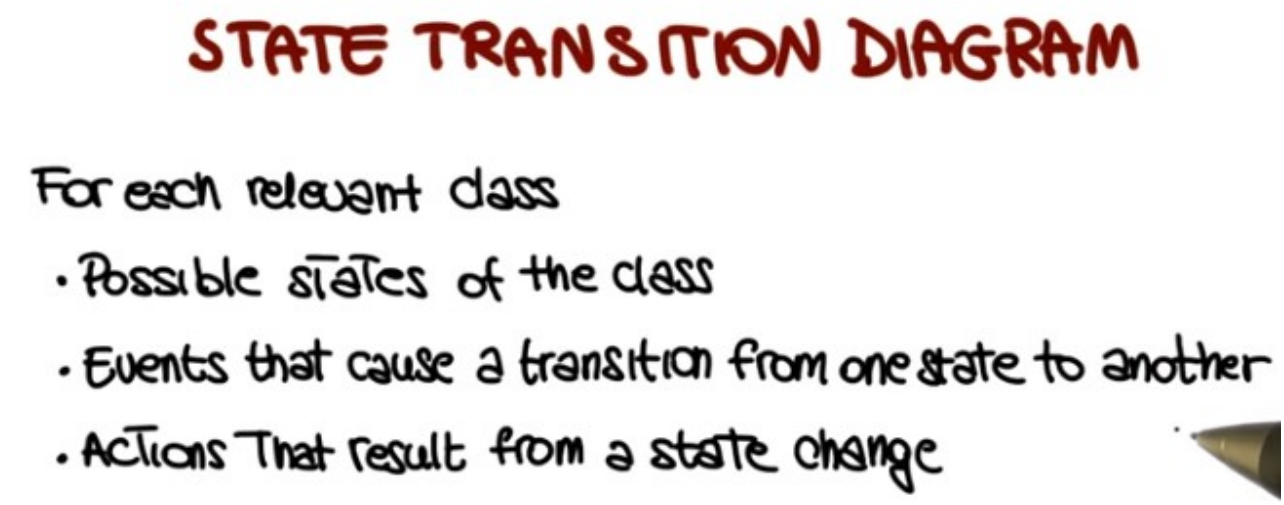
31. Now that we have seen use cases, the next behavioral diagram I want to discuss is the sequence diagram. So what is a sequence diagram? It is an interaction diagram that emphasizes how objects communicate and the time ordering of the messages between objects. To illustrate sequence diagrams in a practical way, and hopefully in a clear way, I will introduce them by creating an actual sequence diagram using an example taken from our course management system. So let's see what are the steps needed to build such a sequence diagram. The first thing we want to do is place the objects that participate in the interaction at the top of the diagram along the x-axis, and you also want to place them in a specific way. You want to place objects that initiate the interaction at the left, and place increasingly more subordinate objects to the right. So basically, this should reflect the way the events will flow for the majority of the interactions in the system. Next thing you want to do is to add what is called the object lifeline. It's a vertical line that shows the existence of objects over a period of time. And it's normally represented with a dashed line, except for the outermost object for which it is a solid line. Now that you have your object lifeline you can start placing messages that these objects send and receive. You want to put them along the y-axis in order of increasing time, from top to bottom. And you can also put a number on the message to further clarify the sequence. So in this case what we're

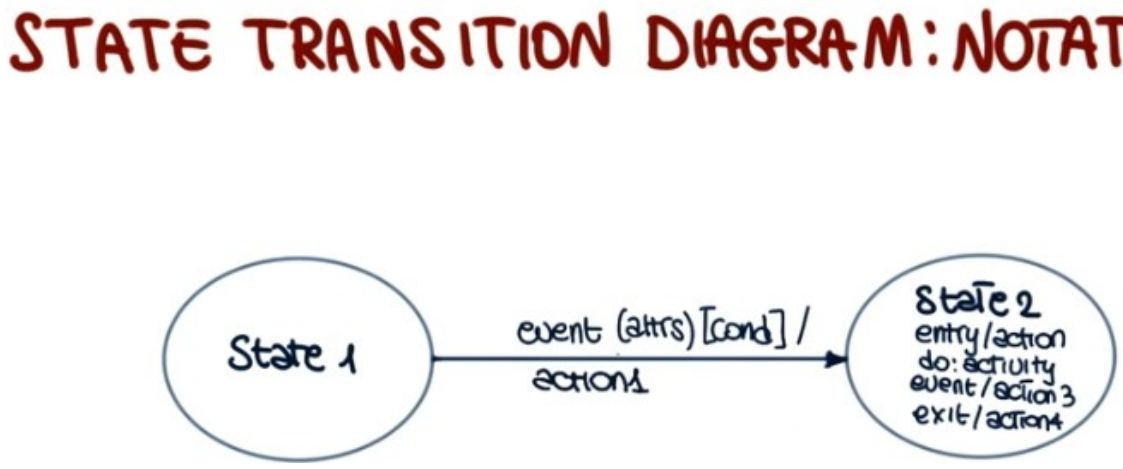
showing is that the student will send the fill in info message to the registration form. And this is the first message in the sequence diagram, the first interaction. Then the student might submit the form and this is also a message that goes to the registration form. At this point, when the submission takes place, the registration form will send the message, so it will invoke some functionality in the registration manager. Specifically you will invoke the add course functionality and pass Joe, the name of the student and Math 101 which is the specific course for which Joe is registering. Then the registration manager will ask the Math 101 course whether it accepts registrations, and the interaction will continue. So that Math 101 will actually check for a specific offering, if everything goes fine, you will receive an ack, you'll send back the act to the registration manager and so on. Until at the end, Joe will be registered for Math 101. As you can see, it is very easy to see how the interaction occurs between these different objects at run time, dynamically. So what the behavior of the system is for this specific scenario. So the last notational element that I want to add to this diagram is the focus of control. Which is this tall thin rectangle, that shows the period of time that an object is performing an action, either directly or indirectly. So if we look at the registration form, this is telling us that the registration form is active for this amount of time. And the same thing we can do for the registration manager, the Math 101 course offering, and the Math 101 specific section.



32. The very last diagram that I want to discuss is the state transition diagram. The state transition diagram is defined for each relevant class in the system and basically shows the possible live history of a given class or object. So what does it mean to describe the life history? It means that it describes the possible states of the class as defined by the values of the class attributes. And it also describes the events that cause a transition from one state to another. Finally, it describes the actions that result from a state change. So if you put all of this together you can see how this can represent the whole history of the class, from its creation to its destruction. So let me discuss the transition diagram in more detail, and also provide information about the notation used to represent them. We have states, that are represented by ovals with a name. And we have transitions marked by the event that triggers the transition. What transitions indicate is the passage from one state to another state as the consequence of some external stimuli. Notice that not all events will cause a state transition. So for example, some

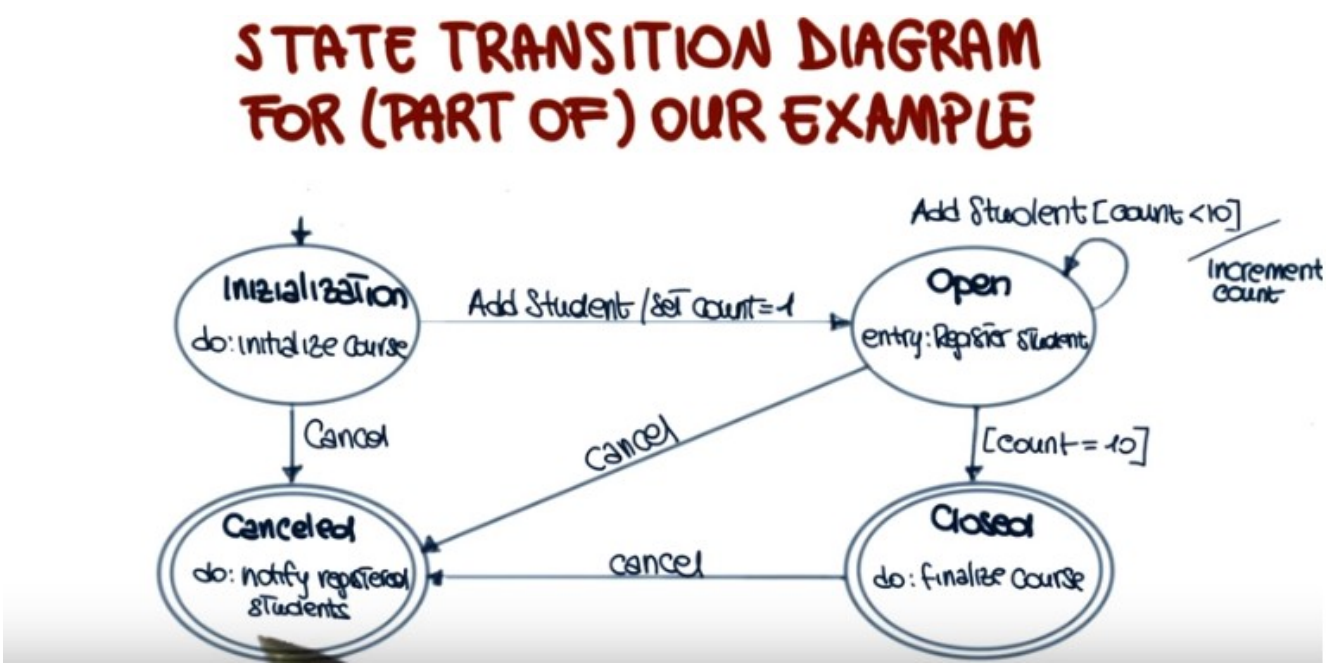
events might be consumed within a single state. And we'll get to that in a second. But in most cases, an event will trigger some state transition. Events may also produce actions and they may also have attributes which are analogous to parameters in a method call. And Boolean conditions that guard the state transition that is prevented from happening in the case the conditions are not satisfied. States may also be associated with activities and actions. Specifically, activities are operations performed by an object when it is in a given state and that takes time to complete. Actions, conversely, just like the actions corresponding to an event are instantaneous operations that are performed by an object. And can be triggered on entry. So, when the object reaches a given state, when the object exits that state, and also when a specific event occurs. And in this case, this notation is basically a shortcut for any event that will cause a state transition that will bring the object back into the same state. Since we have several actions and activities, it is probably worthwhile clarifyinig the ordering of such actions and activities. So the way in which these actions and activities occur is, first of all, we have the actions on the incoming transition, so this is performed first. Then if there is an entry action that is the next action that would be performed, then we have activity and event actions as appropriate. And finally exit actions.



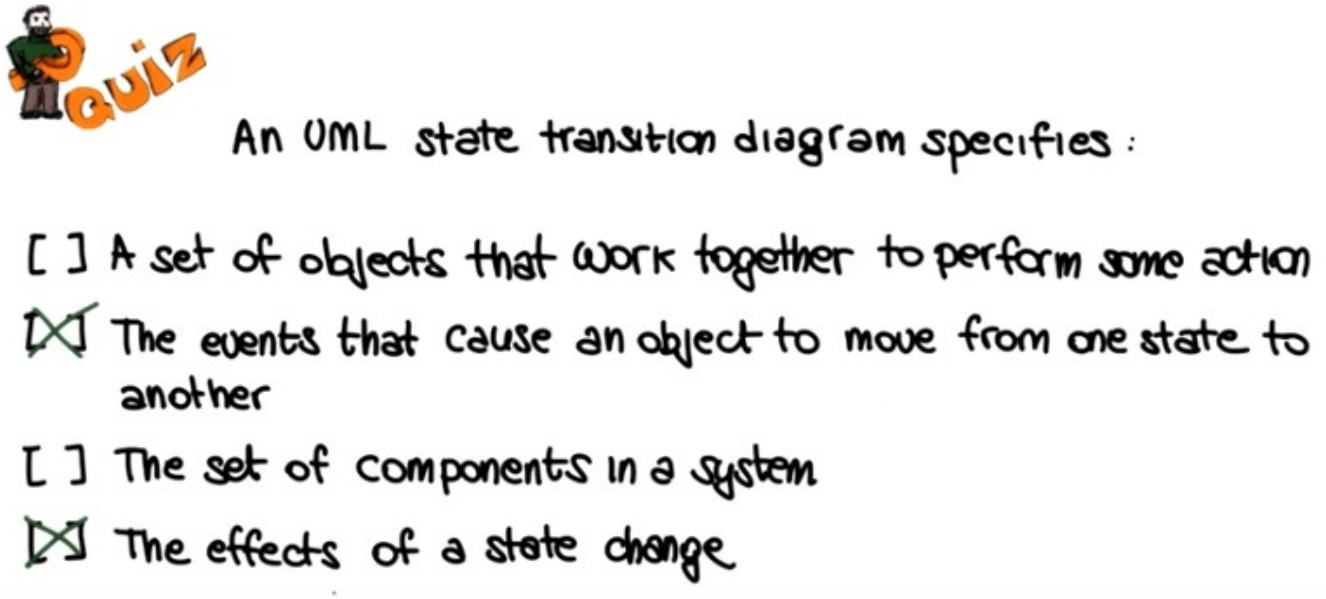


33. As usual were going to illustrate this kind of diagrams by using an example. In particular we are going to describe the state transition diagram for part of our example, for part of our course management system. And we're going to focus on the course offering class. When the class is created, it

enters the initialization state, in which the activity performed is to initialize the course. At this point, a simple case is the case in which the class is cancelled, so there is a cancel event. And if that happens, the class is simply cancelled. So it gets into this final state, which is the state cancelled. And the activity in this case is to notify the registered students. Obviously if this is the flow there will be no registered students. However something else can happen when we are in this initial state. So what can happen is that a student can register, so in this case the add student event is triggered. And the corresponding action is to set the count, in this case it would be the count of students for the course offering, to one. And there will be a change of state and the course offering will get into this open state. And the action that will be performed on entry will be to register the student. At this point more students may register. So other student events marker and notice that we have a guard here that tells us this event will trigger this transition only if the count is less than 10. So we're assuming that we're not going to have more than 10 students just for lack of a better number in our course offering. So if that happens, if the count is less than ten so the count is incremented so the increment count action takes place. And the system goes back into the open state, and the new student is registered. Now here we have an interesting transition, because there's no event triggering the transition, but simply the fact that the count is equal to 10. So you can imagine this as being a transition that is always enabled so can always be triggered, but will be guarded. By the fact that the count has to be exactly ten. So basically this transition will take place only when enough students are added, such we get to count them. Being incremented and being equal to ten and then the transition will occur. And we will get into the closed state, in which the class is no longer open because there are enough students registered. And at this point, what will happen is that the course will be finalized. So there will be this activity which performs some operation that is needed to finalize the course. Another possibility is that when we are in the open state, the course is cancelled. And if the course is cancelled, in this case, we go again to the cancel state. But here, the activity of notifying registered students makes more sense. Because we will have at least one registered student in this state, and therefore we'll need to notify such student that the course offering has been cancelled. Finally, is it also possible also to cancel a course after it has been closed? And in this case again, the same thing will happen. The class will reach the cancelled state and all the students, in this case ten students, that are registered for the course will be notified that the course has been cancelled. So, if we look at this state transition diagram, you can see that it's pretty easy to see what the evolution of objects of this class can be. How they can go from their initial state to various final states depending on what are the external events that reach the system.



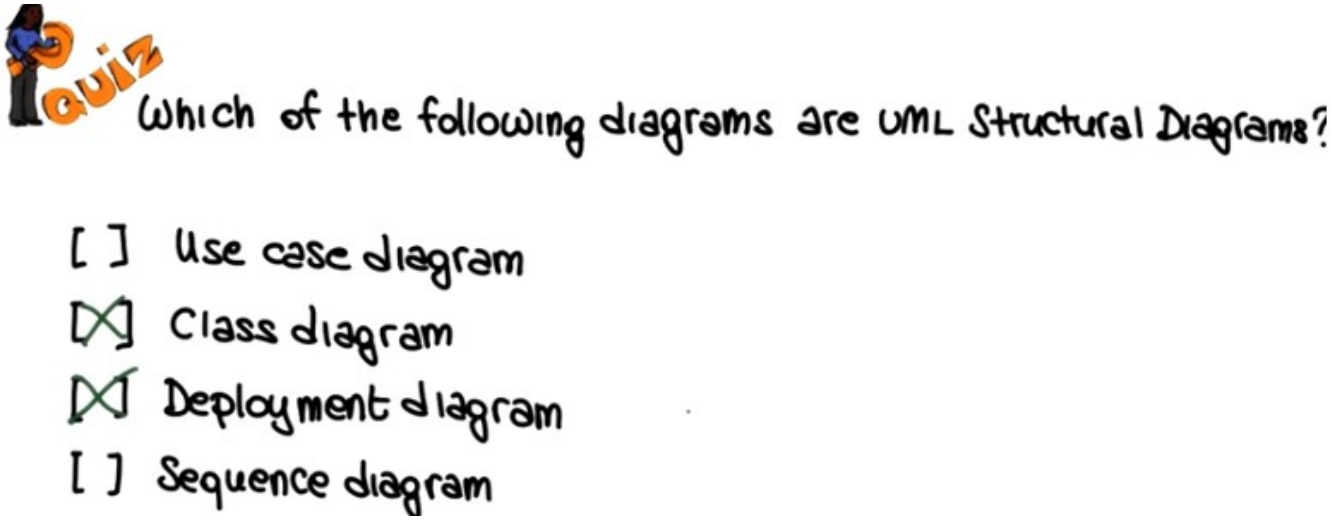
34. I'd like to conclude this lesson with a couple of quizzes. Just to recap what we saw. And, make sure that everybody's on the same page. In the first quiz, I want to know whether an UML state transition diagram specifies a set of objects that work together to perform some action. The events that cause an object to move from one state to another. The set of components in a system, or the effects of a state change. And as usual, you should mark all that apply.



35. A UML state transition diagram does not specify a set of object that work together to perform some action, because this is what a sequence diagram does instead. Conversely, the second one is correct. As we said, a state transition diagram describes the events that cause a transition from one state to another. Again, a UML state transition diagram does not specify the set of components in a system, because this is what a component diagram does, not a state transition diagram. As for the last one, this is correct,

because, as we also discussed, a state transition diagram describes the actions that result from a state change, that is, the effects of such state change.

36. And for the last quiz, I want to know which of the following diagrams are UML Structural Diagrams? Use case diagram, class diagram, deployment diagram and sequence diagram. Again, mark all that apply.



37. And in this case, the correct answer is that class diagram and deployment diagrams are the only two UML structural diagrams among these four. So the answer to this quiz was probably pretty obvious, but I wanted to use it also to stress, once more, the difference between structural and behavioral diagrams. Structural diagrams provide the static picture of the system being modeled, presented from different perspective. For example, from the perspective of the class diagram and of the deployment diagram. Behavioral diagrams, on the other hand, provide information on the dynamic behavior of the system being modeled, also presented from different perspective. So it's important to be able to distinguish between these two types of diagrams. This concludes this lesson on object orientation and UML. But I encourage you to look at the references provided in the class notes, in case you want to know more about object orientation, object oriented analysis, and UML.