







1-1	 <p>Object-Oriented Analysis and Design (Part 2)</p> 	<p>In this module we will continue our coverage of object-oriented analysis and design.</p> <p>As you will recall from the previous module, we are modeling the problem and the solution in terms of objects. Objects are made up of information and related operations. Objects should be cohesive and have well defined lightweight coupling to other objects. We started by taking advantage of the anthropomorphic nature of objects in CRC modeling. We then created static analysis models using UML class diagram notation.</p>
2	 <p>Overview</p> 	<p>We will finish up our coverage of object-oriented analysis with dynamic analysis modeling. We will be using the UML activity diagram notation for this.</p> <p>Then we will turn to object-oriented design. The main difference between analysis and design is that in analysis we find and specify the classes of objects inside the system. In design we describe how to implement the classes in code.</p>
3	 <p>Objectives</p> <ul style="list-style-type: none"> <li>• Here is what you should be able to do upon completion of this module <ul style="list-style-type: none"> <li>○ Perform dynamic object-oriented analysis modeling and express the models using UML activity diagram notation</li> <li>○ Create static object-oriented design models of the code using UML class diagram notation</li> <li>○ Create dynamic object-oriented design models of the code using UML sequence diagram notation</li> </ul> </li> </ul>	<p>Here is what you should be able to do once you've completed this module.</p> <p>You should be able to perform dynamic object oriented analysis, using UML activity diagram notation for your modeling.</p> <p>You should be able to create static and dynamic object-oriented design models using UML class diagrams and sequence diagrams.</p>
4	 <p>Outline</p> <ol style="list-style-type: none"> <li>1. Introduction</li> <li>2. Dynamic analysis modeling</li> <li>3. UML activity diagrams</li> <li>4. Internet storefront example – analysis activity diagram</li> <li>5. Introduction to OO Design</li> <li>6. Static OO design modeling using UML class diagram notation</li> <li>7. Implementing associations</li> <li>8. Implementing aggregations and compositions</li> <li>9. Object lookup</li> <li>10. Implementing generalization-specialization</li> <li>11. Internet storefront example – design class diagram</li> <li>12. Dynamic design</li> <li>13. UML sequence diagrams</li> <li>14. Structuring sequence diagrams</li> <li>15. Internet storefront example – design sequence diagrams</li> </ol>	<p>This looks like a lot, but most of these videos are short.</p>

5


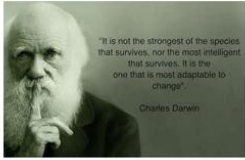
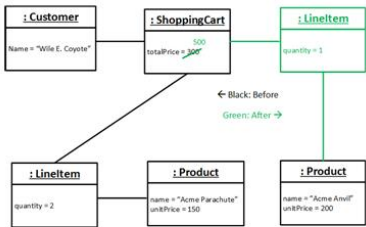
### Next


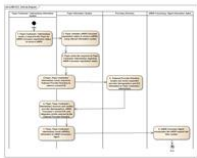
- Dynamic analysis modeling


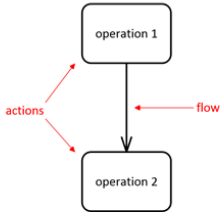
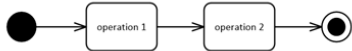


We have to finish up our coverage of object oriented analysis. We will look at dynamic analysis modeling in the next video.

2-1	<div data-bbox="347 199 680 233" data-label="Section-Header"> <h3>Dynamic Analysis Modeling</h3> </div> <div data-bbox="415 266 612 394" data-label="Image"> </div>	<p>Dynamic modeling means that we model things changing over time. There's always an element of time in dynamic modeling.</p> <p>What can change over time?</p> <p>Information inside the system can change over time.</p> <p>The outputs that the system sends to the external actors are events that occur over time.</p>
2	<div data-bbox="371 516 656 550" data-label="Section-Header"> <h3>How Did We Get Here?</h3> </div> <div data-bbox="381 590 643 751" data-label="Diagram"> </div>	<p>Up to this point, we have alternated between dynamic and static modeling at lower and lower levels of abstraction.</p> <p>We started with use case specifications, which are dynamic models of the system behavior. We treated the system as a black box and modeled the messages going into the system from actors, messages coming from the system to actors, and things happening inside the system. Those things that happen inside the system are called system operations. In the requirements the system operations are elemental units of functionality. They can't be broken down further without knowing what's inside the system.</p> <p>Then we did some static modeling by identifying candidate classes of objects that we figured would need to be inside the system to make it hold the information and provide the services specified by the system operations.</p> <p>We then did dynamic modeling using CRC cards, acting out the dynamics of the candidate objects in order to determine their responsibilities. The two kinds of responsibilities, as you will recall, are to know things and to do things.</p> <p>We took the information from the CRC cards and built a static analysis model: a Class Diagram.</p> <p>That brings us to this point</p>
3	<div data-bbox="380 1587 651 1621" data-label="Section-Header"> <h3>Where Are We Going?</h3> </div> <div data-bbox="406 1682 621 1827" data-label="Image"> </div>	<p>Now we are ready to do dynamic modeling once again.</p> <p>We will detail the specifications of the system operations by being precise as to what system state changes occur during the execution of the system operations.</p> <p>We will see how to use UML activity diagram notation to model the dynamics of the system operations.</p>

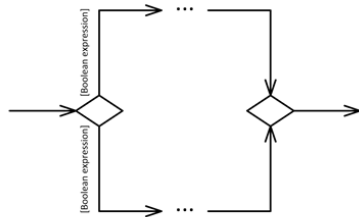
<p>4</p>	<h3>System Operations</h3> 	<p>One reason we like to model the behavior of the system in terms of system operations is that they provide a convenient level of granularity for doing dynamic modeling. Trying to model an entire use case can be rather difficult. A use case is actually a collection of system operations.</p> <p>System operations are behaviors that are visible from outside the system. They describe what the system does as a step in a use case.</p> <p>A system operation is invoked by an actor during a use case scenario.</p> <p>A system operation is an elementary unit of functionality that is performed by the system, as seen from the outside. It can't be decomposed any further at the requirements level of abstraction. The next level of decomposition requires an understanding of what is inside the system to make it work. This is the level of abstraction that we are at during analysis.</p>
<p>5</p>	<h3>State Changes</h3> 	<p>A system operation typically results in a change in state of the system. We will specify a system operation in terms of the state of the system before and after the execution of the system operation. These changes can be documented as preconditions and postconditions. What we mean by system state is the values of all of the information stored within the system.</p> <p>As you recall, there are only two ways to store information inside a system of objects: attribute values within object instances and links connecting objects to one another.</p>
<p>6</p>	<h3>Before and After</h3> 	<p>The only way to change the state of the system is to invoke a system operation. The invocation of the system operation is called a "trigger." The only kinds of system state changes that can occur are these five:</p> <ul style="list-style-type: none"> <li>• Creation of an object instance</li> <li>• Destruction of an object instance</li> <li>• Creation of a link connecting two object instances together</li> <li>• Destruction of a link</li> <li>• Modification of attribute values within an object instance</li> </ul> <p>In this diagram, we see two snapshots of the objects: before and after a system operation. The system operation here is the one we have been using as an example: Add an item to a shopping cart. The black indicates the condition of things before the system operation executes, and the</p>

		<p>green shows the state of things afterwards.</p> <p>In this case, the cart already includes one line item for two Acme parachutes before the system operation starts. Their unit price is \$150, to the total price for the cart starts at \$300.</p> <p>Now, in this system, operation, the customer will ask to add one Acme anvil. The trigger is “add item to order.” The inputs are the Product object for the anvil, and the quantity, 1.</p> <p>The system communicates with the Inventory actor. Presumably, in this scenario, there’s at least one anvil in stock.</p> <p>A new object instance of the class LineItem is created, and its quantity attribute is set to 1. The LineItem object is linked to the already existing Product object for the anvil.</p> <p>The Product object knows that the unit price of the anvil is \$200. The total price attribute of the ShoppingCart object is updated to \$500.</p>
7	<p>How Do You Get From Here to There?</p> 	<p>We need to model how the system gets from the before state to the after state (from the preconditions to the postconditions).</p> <p>As we see, there are several things that change between the before and after pictures. We can’t be sure about the exact order in which these changes occur, but we can produce models of possible ways things might happen.</p>
8	<p>Next</p> <ul style="list-style-type: none"> <li>UML Activity Diagrams</li> </ul> 	<p>We will be using the UML Activity Diagram notation to create these dynamic models. As we said, there may be many different sequences of events that can lead to the same result. We just need to pick one, keeping all of our software engineering principles in mind.</p> <p>In the next video, we’ll take a look at some of the UML notation that is used when creating activity diagrams.</p> <p>Then in the following video we will return to our Internet storefront example to see the application of the notation.</p>

3-1	<p style="text-align: center;">UML Activity Diagrams</p> 	<p>In this video we will see how we can use the UML Activity Diagram notation to document a flow of events inside the system to carry out the effect of a system operation.</p> <p>The last time we did dynamic modeling was when we were acting out scenarios for our CRC modeling. However, the only things we recorded were the responsibilities and collaborator dependencies. We didn't capture the dynamics.</p> <p>Activity diagrams provide us with a notation for describing the dynamic collaborations of the objects graphically.</p> <p>Activity diagrams are good at showing the sequence of actions that occur inside the system when a system operation is being executed. The actions in the activity diagram correspond to the execution of operations on individual object instances inside the system.</p> <p>What activity diagrams do not show is how the objects communicate. Who sends messages to whom? This is not a concern during analysis. It will be one of the things we will need to figure out when we get to design.</p>
2	<p style="text-align: center;">Actions and Flows</p> 	<p>In an activity diagram, we can see that object A executes one of its operations and then object B executes one of its operations. What we can't say for sure is how control passes from object A to object B.</p> <p>An activity diagram is sort of like an old fashioned flow chart. The boxes are called actions and the directed lines connecting the boxes are called flows. (They are sometimes referred to as edges or transformations, but we will try to use the term flow here.) The arrow heads on the flows indicate the direction of flow of control. In a well-structured activity diagram there is exactly one flow into an action and one flow out.</p>
3	<p style="text-align: center;">Start and End Nodes</p> 	<p>There are two special nodes in an activity diagram. They are the start and end nodes. The start is represented as a solid circle with a flow coming out of it. The end, or activity final, node is a solid circle with a ring around it, and only one flow coming into it.</p>

4

### Decision and Merge Nodes



Just like in flowcharts, there are decision nodes in activity diagrams.

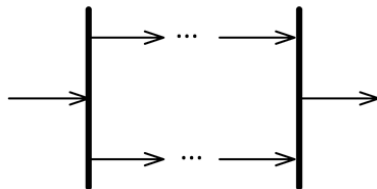
The difference here is that a decision node has nothing inside it (just like start and end nodes have nothing inside them, either). A decision node has one incoming flow and multiple outgoing flows. Each of the outgoing flows is annotated with a Boolean expression, called a guard. Guards are enclosed within square brackets and are placed near the flows emanating from the decision node. At the point in the flow when the decision is reached, the guards are evaluated. The outgoing flow corresponding to the guard that is true is the path that is taken. It is generally, then, a good idea for all of the guards to be disjoint – otherwise the flow will be nondeterministic, which is something you don't want in a computer program. It is also a good idea for there to be no gaps in the set of guards. In logic, we would say that if all of the guards are OR'ed together, they should result in a Boolean TRUE. Violating this condition might result in a premature halt in the flow of control. If the decision point is reached and all of the guards are evaluated, and none of them resulted in a TRUE condition, the flow would stop right there.

Corresponding to a decision node is a merger node. The merge is also a diamond shaped node, also with nothing inside it. The purpose of the merge node is to merge all of the paths that were created at the decision node back together into a single flow once again.

In a well-structure activity diagram, all of the flows that come out of a decision are combined back together again in a merge. There are exceptions, but this is the basic rule.

5

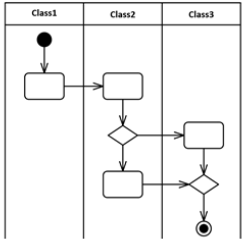
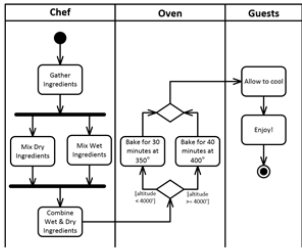

### Fork and Join




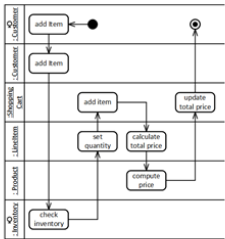
A construct that is structurally quite similar to the decision and merger is the fork and join. We use the same icon for both the fork and the join. It is a straight bold line. The fork has a single flow coming in, and multiple flows coming out. The join has multiple flows coming in, and a single flow coming out.

Unlike with the decision-merge structure, where only one flow is executed, with the fork and join, all of the flows emanating from the fork are executed. They are performed asynchronously. This means that they could be executed at the same time, or they could be executed in any order. It shouldn't matter what order they are






		<p>executed in.</p> <p>All of the flows that come out of a fork are joined back together again at the join. The behavior of the join is that it won't resume with the outgoing flow until all of the incoming flows have finished executing and control for each flow has arrived at the join.</p> <p>In a well-structured activity diagram, all of the flows that come out of a fork are combined back together again in a join. There are exceptions here also, but this is the basic rule.</p>
6	<p><b>Partitions</b></p> 	<p>A nice feature of activity diagram modeling is the partition, or sometimes called the “swim lane.” While keeping all of the flows intact, it is possible to rearrange the actions so that they are in groups based on who performs them. The “who” in this case might be actors, or subsystems, or object instances. The partitions are labeled with a name indicating who performs the actions in the partitions.</p>
7	<p><b>Example</b></p> 	<p>Here is an example of this notation. It is an activity diagram for baking a cake. It starts when an object instance of Chef executes the “gather ingredients” action. Then the chef will mix the dry ingredients and mix the wet ingredients separately. These actions may be done in either order, or at the same time. They are asynchronous. After both actions have finished, flow continues with the chef combining the wet and dry ingredients. What happens next is a decision. If the altitude is less than 4000 feet, the path on the left is taken. If the altitude is greater than or equal to 4000 feet, the one on the right is taken. In either case, the oven will bake the cake for a certain amount of time at a certain temperature. The flows merge back together and then the guests allow the cake to cool and then enjoy eating it. The activity terminates at that point.</p>
8	<p><b>Next</b></p> <ul style="list-style-type: none"> <li>Internet Storefront Example – Activity Diagram</li> </ul> 	<p>In the next video, we will continue with our Internet Storefront example. We will draw an activity diagram for the system operation “add an item to the shopping cart.”</p>



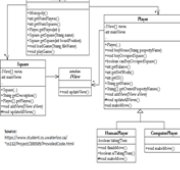
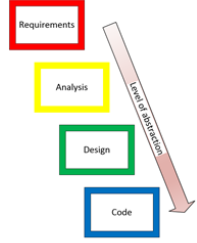
4-1	<p>Internet Storefront Example Analysis Activity Diagram</p> 	<p>In this video we will return to our Internet Storefront example. We will draw an UML activity diagram.</p>
2	<p>System Operation</p> <p>After finding the product, the customer selects “add item to cart.” The customer also indicates the quantity. The system checks inventory to see that there is sufficient quantity in stock. The item is added to the cart and the total price of the order is updated.</p>	<p>The dynamics we will show in the activity diagram are a bit more detailed than what we explored in the CRC analysis. We will find that we need to add a couple more operations to some of the classes.</p> <p>Here is the system operation specification. We’ve been through this before several times, so the behavior should be familiar to you.</p>
3	<p>Activity Diagram</p> 	<p>Here is the activity diagram. Notice that the partitions run horizontally. Sometimes things fit better on the page this way. In an activity diagram, the partitions can be either vertical or horizontal.</p> <p>Our activity starts with the customer actor requesting the operation “add item.” The customer actor communicates with the system through what is called a boundary object.</p> <p>There are two boundary objects in this diagram. One for the customer and one for the inventory subsystem. The boundary objects serve as intermediaries between the actors outside the system and the objects inside the system.</p> <p>The boundary object for the customer then invokes the operation “add item” in the object instance of the Customer class inside the system.</p> <p>Before we can process the item, we have to determine that there’s enough stock in inventory. This action is performed by the inventory boundary object, which communicates with the external inventory subsystem.</p> <p>Assuming the inventory is OK, a line item object is created and its quantity is set. In design, this will be done by a constructor. For analysis we can simply invoke a setter operation.</p>



		<p>Next, the item is added to the shopping cart.</p> <p>We go back to the line item to perform the action, calculate total price (which is done by multiplying the unit price by the quantity). The unit price is in the Product object. The quantity is in the LineItem object. So which object will perform the action? We talked about this before. One approach would be for the line item to ask the product for the unit price and multiply it by its quantity. The other would be to pass the quantity to the product object and let the product compute the total price. We chose the second approach. It may seem like an arbitrary choice, but there is an important rule of object oriented software engineering at play here. That rule says to let the object that owns the data do the computation. This rule results in better encapsulation of the data. As we discussed, there is also another advantage of having the Product object do the calculation of the total price. Suppose there is a volume discount. Only the Product should know the rule for computing this discount. The unit price would be different depending on the quantity. This knowledge should be encapsulated inside the Product object. That's why we let the Product perform the action of computing the price.</p> <p>Finally, the total price in the shopping cart is updated.</p> <p>And the activity ends.</p> <p>As you can see, activity diagrams are good at showing the sequence of actions. They are good at showing looping, decision making, and asynchronous activity.</p> <p>But they aren't so good at some other things. They don't show arguments being passed and results being returned. They also don't show who's in control. Just because one action follows another doesn't mean that the object performing the first action sends a message to the object performing the second action. There could be another object sending messages to both of the objects. Also, there is no way to show creation or destruction of objects.</p> <p>These things are mostly design level considerations, so they don't really present a big problem for us at analysis time.</p>
--	--	---

		When we get to design, we will be using UML sequence diagram notation. This will handle most of the things we just said were difficult with activity diagrams.
4	 <p>Next</p> <ul style="list-style-type: none"> <li>• Introduction to OO Design</li> </ul> 	This concludes our coverage of object oriented analysis. With our next video, we will start our discussion of object oriented design.



5-1	 <p>Introduction to OO Design</p>	<p>For the rest of this module we will be discussing object oriented design</p>
2	<p>Change in Focus</p> <p>What ➡ How</p>	<p>In design, our focus changes. When we were doing analysis, we were trying to get an understanding of what classes needed to be built and how the object instances of those classes interacted to carry out the system operations.</p> <p>In design, we are concerned about how those classes should be implemented in code.</p> <p>Here's another way of looking at this. The requirements models specify the system as a black box. The analysis models specify the classes within the system as smaller black boxes.</p> <p>Design, on the other hand, is an abstraction of the code. It models the implementation at a higher level of abstraction.</p> <p>Analysis is a more concrete view of the requirements while design is a more abstract view of the code.</p>
3	<p>Mapping</p> <p>Analysis ↪ Design</p>	<p>We will be talking about some techniques for mapping the analysis model to the design model.</p> <p>There were a few occasions when we were doing the analysis modeling that we said that we would defer certain decisions until design time. Well, now's that time.</p> <p>During analysis we weren't too concerned with control strategy. We were more interested in understanding the behavior of the objects themselves. In design, control is a big issue because it affects the code.</p> <p>At analysis time, we modeled relationships between the classes, but didn't get too concerned about how those relationships would be implemented in code. Well, now we will be.</p> <p>In the design there will be a lot more classes than in the analysis. In analysis, we only modeled just the domain, or entity classes. In the design, we</p>





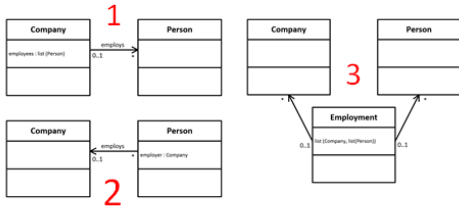
6-1	 <p>Static OO Design Modeling using UML Class Diagram Notation</p>	<p>In this video we will begin our discussion of the design class diagram. Even though we use UML notation for both the analysis and design class diagrams, they are distinct models. For one thing, we will use more UML notation in design than we did in analysis. We'll talk about this additional notation in later videos.</p>
2	 <p>Levels of Abstraction</p>	<p>Most of the classes that we modeled in the analysis will reappear in the design class diagram, but there may be some differences, which we'll discuss in a second.</p> <p>The overall level of abstraction in the design class diagram is much lower. We are creating models of the code now.</p> <p>Some of the issues we will get into will seem to many people to be coding level issues. However, since the design is an abstraction of the code, these are really design level decisions. It just turns out that most of the time, people skip the design step and go right into coding. Of course, they seem like coding decisions, since they haven't done the design. This doesn't change the fact that they are truly design decisions.</p> <p>In fact, many of the decisions that we are forced to make in programming are in fact design decisions. It is my contention that many decisions that we make in coding should be made in the design modeling instead. These decisions have to be made sometime, and it's better to make them and model them in a more abstract notation like the design class diagram or sequence diagram than it is to try to make these decisions when we are at the much lower level of abstraction of the programming language. If the decision needs to be changed, it is much easier to do in the UML than it is in the code.</p>
3	<p>Most Analysis Entity Classes are also in the Design</p> <p>(but some may not be)</p>	<p>Most of the entity classes in the analysis class diagram will appear in the design class diagram as well. However, there may be some good reasons why one or more of the analysis classes is not part of the design.</p> <p>It may turn out that one of the classes in the analysis model is better implemented as two or more smaller classes in the design.</p> <p>Or we may go the other way, and combine two or</p>

		<p>more smaller classes in the analysis into one in the design.</p> <p>Another possibility is that what we thought of as a class in analysis is actually more appropriately modeled as an attribute in the design.</p> <p>Maybe what we thought of as a class in the analysis is actually an external subsystem, and wouldn't appear in the design at all.</p> <p>What looked like inheritance at analysis time might actually be better realized as a flag attribute in the design.</p> <p>If we used boundary and controller classes in our analysis modeling, these may not be necessary in design.</p>
4	<div>Classes Introduced at Design Time</div> 	<p>We may introduce new classes into the design.</p> <p>Even if we didn't use controllers at analysis time, we may wish to create controller objects in the design. This is especially true if we are using a pattern such as the model-view-controller. We'll talk about this and other patterns in another module.</p> <p>In many cases, we will need to introduce objects to represent the information that is modeled as links in the analysis. When one object is linked to another object, the information about the two objects being linked needs to be stored somewhere. It may be stored in either of the linked objects (or both, even). But another approach would be to create a third object that contains the information about the two objects being linked. We'll see more on this later.</p> <p>We may need to introduce objects to improve performance, provide for persistence, concurrency, interfacing to other subsystems, etc.</p>
5	<div>Creating the Design Class Diagram</div> 	<p>Even though we use much of the same UML notation in the design class diagram that we did in the analysis model, it is a good idea to start fresh with the design class diagram rather than to try to evolve the analysis class diagram.</p> <p>You probably will need to develop the dynamic model at the same time as you are working on the static model. This is an iterative process. Neither the static nor dynamic part can be done in isolation. That said, we are forced to cover most of the notation involved in the static modeling in this and the next few videos, and the dynamic</p>





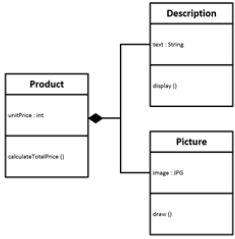
		<p>modeling in later videos. That's just for pedagogical reasons.</p> <p>As we identify classes that belong in the design, we add them to the class diagram.</p> <p>Start with the classes in the analysis model.</p> <p>Eliminate any objects that are simply attributes of another object.</p> <p>Eliminate any objects that are outside the system.</p> <p>Split any really complicated classes into multiple, more cohesive classes. We'll discuss ways of doing this later on.</p> <p>Introduce coordinator and controller objects that may be needed in the dynamic models.</p>
6	<div>  <p>Next</p> <ul style="list-style-type: none"> <li>Implementing Associations</li> </ul>  <pre> classDiagram     class Company     class Person     Company "0..1" -- "1" Person : employee </pre> </div>	<p>In the next video, we will discuss techniques for implementing relationships between classes.</p>



7-1	<div data-bbox="354 199 678 231" data-label="Section-Header"> <h2>Implementing Associations</h2> </div> <div data-bbox="394 258 638 283" data-label="Section-Header"> <h3>In the Design Class Diagram</h3> </div> <div data-bbox="391 306 638 428" data-label="Diagram"> <pre> classDiagram     class Company {         employees : List (Person)         hire ()         terminate ()     }     class Person {         name : String         ssn : SocSecNo         work ()         drinkCoffee ()     }     Company "0..1" -- "1" Person : employs   </pre> </div>	<p>In this video we will begin our discussion of implementing class relationships in the design class diagram.</p>
2	<div data-bbox="354 514 678 546" data-label="Section-Header"> <h2>Links = Object References</h2> </div> <div data-bbox="300 569 500 594" data-label="Section-Header"> <h3>Referential Attributes</h3> </div> <div data-bbox="341 617 686 787" data-label="Diagram"> <pre> classDiagram     class Company {         employees : List (Person)         hire ()         terminate ()     }     class Person {         name : String         ssn : SocSecNo         work ()         drinkCoffee ()     }     Company "0..1" -- "1" Person : employs   </pre> </div>	<p>When we created the analysis class diagram, we showed relationships between classes. As you recall, most of these relationships represent information about the linking of objects... information that needs to be remembered over time. We said that this information is shared by the two objects being linked. We specifically said that the linkage information is not stored in either of the objects being linked.</p> <p>That was analysis. This is design. In design, ALL information, whether it is attributes or link information, all information is stored as attributes inside objects. There is no information floating around outside of the objects.</p> <p>So, linkage information must be stored as attributes in the design model. The question comes down to, where should this linkage information be stored?</p> <p>There are two objects being linked. That means that the linkage information could be stored in either of those objects. Or, it could be stored in some other object. As long as the information is stored. That's what is important.</p> <p>The linkage information is stored as object references or pointers. The precise terminology depends on the programming language. In this course, we will try to use the term object reference.</p> <p>An object reference is a piece of data. It can be stored in a variable. The data type of that variable is the class of the object being referenced. More precisely, the data type of the variable can be either the class of the object being references, or any of the ancestor classes up the inheritance hierarchy.</p> <p>Also, since an object reference is data, it can be passed around as an argument or return result in</p>

		the invocation of an operation.
3	<p>Directionality</p> 	<p>In the design class diagram, when we make the decision about where the object reference is to be stored, we will represent that information by putting an arrow head on one end of the association line. This directed association looks like a pointer from the class where the object reference is stored to the class being referenced.</p> <p>In UML this is sometimes referred to as navigability, as if the association is a path to be navigated. I think this is the wrong way to look at associations. As we have pointed out before, associations represent information. They are not to be viewed as conduits for sending messages. Thus the idea of navigating associations doesn't make sense. In order to send a message to an object, all you need is the reference to that object. The reference doesn't need to be stored as an attribute of the sending object. It could just be a local variable in the operation sending the message.</p> <p>We will discuss how the object reference could be put into this local variable in the video on dynamic modeling in the design.</p>
4	<p>Determining the Directionality</p> 	<p>That said, navigability does play a role in deciding where to store the object reference. If the object reference is stored as an attribute of an object, a message can easily be sent to the referenced object. The object being referenced can't send a message to the referencing object very easily because it doesn't have a stored reference.</p> <p>So, if we want to optimize message sending in one direction, we can use that decision to tell us where to store the object reference.</p>
5	<p>Three Choices</p> 	<p>So, here are the three choices we have. A could store the reference to B, B could store the reference to A, or a third object, C, could store a reference to both A and B.</p> <p>How do you decide? It depends on which way you want to optimize the navigation. This, of course, depends on a good understanding of the system operations, and their probabilities of occurrence.</p>




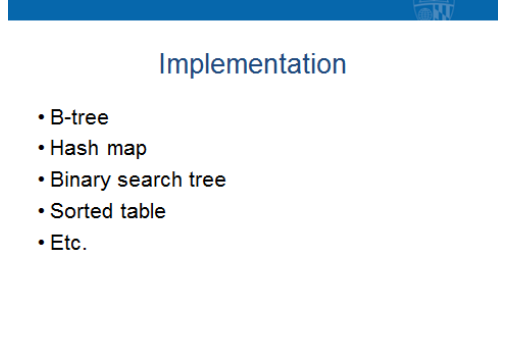
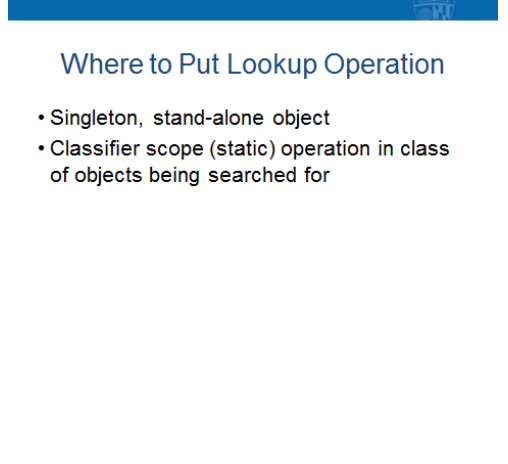
6	<div data-bbox="397 142 633 178" data-label="Section-Header"> <h3>Multiplicities Matter</h3> </div> <div data-bbox="305 220 711 373" data-label="Diagram"> </div>	<p>If the multiplicity on each end of the association is 1, then storing the link is easy. You just put a referential attribute variable in the referencing class.</p> <p>But if the multiplicity is greater than one, an object instance of the one class has to be able to point to many object instances of the second class. This means that the object must store a collection of references. This collection is a set, by default. There is no left to right order of the links.</p> <p>If you wanted the links to be ordered in some way, then you would have placed a constraint such as {ordered} or {FIFO} on the association. Maybe you didn't think to do this in analysis, but now, in design, you realize that you should put the links in some kind of order. You can go back and change the analysis model.</p> <p>If the association in the analysis class diagram has an {ordered} constraint, the collection used in design would be some kind of ordered collection such as a list or vector.</p>
7	<div data-bbox="483 982 544 1018" data-label="Section-Header"> <h3>Next</h3> </div> <div data-bbox="300 1039 617 1092" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Implementing Aggregations and Compositions</li> </ul> </div> <div data-bbox="438 1102 587 1255" data-label="Diagram"> </div>	<p>In the next video, we will take a look at implementing aggregations and compositions.</p>

8-1	<p>Implementing Aggregations and Compositions</p> 	<p>In this video, we will continue our discussion of implementing relationships between classes. If we see an aggregation or composition relationship between two classes in the analysis class diagram, how should we implement that relationship in the design class diagram?</p>
2	<p>Aggregations</p> 	<p>Let's take the easier one first: aggregation.</p> <p>As we discussed in our analysis module, aggregation is just a special kind of association that has the inherent meaning, "contains" or "part of." It has a special symbol, an open diamond on the aggregator end of the association. Beyond that, it should be treated as a regular association in terms of its implementation in the design class diagram.</p> <p>First let's talk about directionality. Where should the linkage information be stored? In the aggregator, or in the part? Most people would say, "Put the links in the aggregator class," and leave it at that. But the answer isn't always so easy. We really have to look at the dynamic models to see whether they would be more optimized by storing the links in the aggregator or the part. If the links are stored in the parts, each part object is aware of its aggregator, but the aggregator doesn't know its parts. If the links are stored in the aggregator, it can see its parts, but the parts can't see their aggregator.</p> <p>Then there's multiplicity. The multiplicity on the diamond end of the aggregation is usually one, but not always. It may be possible to have a situation where an object can be part of multiple aggregations at the same time. Think of an employee object. It could be part of a department aggregation. It also could be part of a union aggregation at the same time.</p> <p>So the way we deal with multiplicity is exactly the same as for regular associations. If the multiplicity on the far side of the relationship is greater than one, then you'll need a collection of object references. Once again, by default this collection is an unordered set. If you need order, then you can use an ordered collection, such as a list.</p>

3	<h3 style="text-align: center;">Compositions</h3> 	<p>Compositions are the relationships with the solid diamond on the composite end of the relationship. Compositions are much more highly constrained than aggregations. We discussed these constraints when we talked about compositions in the static analysis module. These constraints make the choices for implementing compositions a bit more limited than for aggregations.</p>
4	<h3 style="text-align: center;">Composition Constraints</h3> <ul style="list-style-type: none"> <li>• The composite             <ul style="list-style-type: none"> <li>◦ Encapsulates the parts                     <ul style="list-style-type: none"> <li>◦ No linking to parts by any object other than composite</li> <li>◦ All communication to parts controlled by composite</li> </ul> </li> <li>◦ Controls the lifetime of the parts                     <ul style="list-style-type: none"> <li>◦ Cascading delete</li> </ul> </li> </ul> </li> </ul>	<p>Let's recall the constraints implied by a composition relationship.</p> <p>The composite fully encapsulates the parts. All communication to the parts is controlled by the composite. Usually this means that all messages that are sent to the parts are sent by the composite. If you need to get a message to a part, you must send it to the composite, which will, in turn, send a message to the part.</p> <p>This also means that no object other than the composite may be linked to the part objects. A part may only be contained in one composite at any point in time.</p> <p>The lifetime of the parts is completely controlled by the composite. This means that part objects are only constructed by or on behalf of the composite. The parts are also destroyed by or on behalf of the composite only. This is sometimes referred to as the "cascading delete rule." If the composite goes away, so do its parts. When we say "destroyed," it means different things in different languages. In some languages, you must send a destructor message to an object to have it clean itself up, and return its memory to the free space list. In other languages, this is done by a garbage collector when all references to the object are deleted. This is partly the reason for the exclusive ownership rule. If the composite is the only object containing links to the part, if the composite goes away, so do the links to its part objects. Since these are the only links to these objects, the objects can be garbage collected. In a language without garbage collectors, you must make sure the composite sends the destructor messages to all of its parts as part of its own destructor method.</p> <p>This also means that the directionality of the composite relationship must always be from the composite to the parts. The links are stored in</p>

		the composite. Usually the multiplicity on the part end of the relationship has a multiplicity of greater than one. This, of course, means that the links are stored in a collection. And, again, by default this is a set. If you need order, then you'll have to use an ordered collection.
5	<div>  <p>Next</p> <ul style="list-style-type: none"> <li>• Object lookup</li> </ul>  </div>	In the next video, we will discuss how to find an object you need to send a message to in the case that you don't have a stored reference to it.



9-1	 <p>Object Lookup</p> 	<p>Up to this point in this module, we have mainly been talking about ways to implement things we see in the analysis class diagram in the design class diagram. Things such as associations, multiplicity, aggregations, compositions.</p> <p>We'll deal with some more of that later, when we get into implementing generalization/specialization relationships.</p> <p>But we can't go too far without discussing a very important topic: object lookup.</p>
2	 <p>Why Do We Need to Look Up Objects?</p> <ul style="list-style-type: none"> <li>• Translate keys to object references</li> </ul>	<p>In many systems, involving multiple subsystems that have to communicate with each other by sending messages, these messages may contain references to specific objects. However, unless some sort of object-oriented inter-process communication mechanism is used, in which serialized objects can be sent from one subsystem to the other, these object references are typically in the form of keys.</p> <p>Each object instance of a particular class will have a unique key value associated with it, and stored as an attribute. The object instance can then be located by using that unique key.</p> <p>We will need an operation to translate a key value into an object reference.</p>
3	 <p>Implementation</p> <ul style="list-style-type: none"> <li>• B-tree</li> <li>• Hash map</li> <li>• Binary search tree</li> <li>• Sorted table</li> <li>• Etc.</li> </ul>	<p>The lookup may be implemented in many ways. You studied these techniques in the Data Structures class. Fortunately most modern languages and environments support efficient mechanisms such as hash maps that make the implementation of the lookup relatively easy.</p>
4	 <p>Where to Put Lookup Operation</p> <ul style="list-style-type: none"> <li>• Singleton, stand-alone object</li> <li>• Classifier scope (static) operation in class of objects being searched for</li> </ul>	<p>In order for other objects in the system to be able to invoke this lookup operation, it needs to be visible to all of them. The best way to accomplish this is to make the lookup operation globally visible. We could do this in two ways. One would be to put the lookup operation in a singleton object. One thing about singletons is that they are globally visible.</p> <p>The other option, and the one we will choose here, is to add the lookup operation as a static operation to the class of the objects we want to look up. As you know, static operations are invoked by messages sent to the class rather</p>

		than an object instance. Thus static operations have the same visibility as classes, which make them globally visible. Remember that we indicate a static operation with underline font.			
5	<div>Example</div> <ul style="list-style-type: none"><li>From storefront application</li></ul> <table><thead><tr><th>Product</th></tr></thead><tbody><tr><td>sku : int {unique} lookup:map(sku,Product)</td></tr><tr><td><u>getProduct(sku:int):Product</u></td></tr></tbody></table>	Product	sku : int {unique} lookup:map(sku,Product)	<u>getProduct(sku:int):Product</u>	<p>Consider our Internet storefront system. The system operations for finding products in the catalog and then adding the product to an order are quite loosely coupled. In some implementations, there may some kind of session memory that is retained between the two system operations. The two system operations may communicate through the session, in which objects can be shared. In many cases, though, these two system operations are not coupled through shared session memory. Thus if one system operation needed to send an object reference to another system operation, it would have to rely on key values.</p> <p>In our storefront system, the system operation, Add Item to Order, has two inputs: the Product to be added to the order, and the quantity of this product.</p> <p>The quantity is an integer. No problem. The question is, how do we transmit the Product object instance from the client to the server? If there's no object-oriented inter-process communication, this is usually done by keys. So, the client passes a key value that corresponds uniquely to the object instance of the Product that is to be added to the order. It is now up to the server to translate that key value into an object reference to the correct object instance of Product. We need a Product lookup operation. There will need to be a static operation in the Product class to map the key to the object reference. This operation name usually has the verb get or find or lookup in it. The argument to this operation is the key value. The result is an object reference to an object instance of Product.</p> <p>The mechanism for the lookup may be a hash table or b-tree or some other data structure that is optimized for searching. It would have to be stored in the Product class as a static attribute.</p> <p>There will probably need to be operations for managing the lookup information, too. You might see an operation to add a new object, as well as an operation to delete an object from the lookup.</p>
Product					
sku : int {unique} lookup:map(sku,Product)					
<u>getProduct(sku:int):Product</u>					

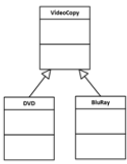
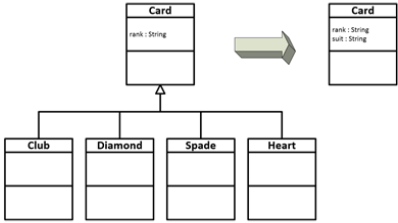
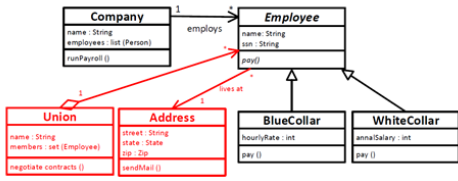
6

## Next

- Implementing generalization-specialization

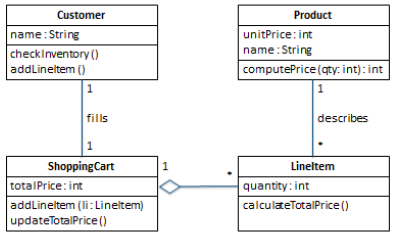
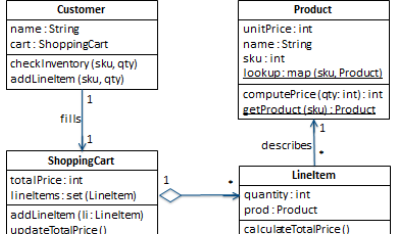


In the next video we will look at what to do about generalization specialization relationships in the design.

10-1	<p>Implementing Generalization-Specialization</p> 	<p>In this video we will discuss options for handling generalization/specialization, or inheritance, in design</p>
2	<p>Flag Attributes</p> 	<p>A lot of the time, when we see generalization specialization relationships in the analysis model, it is because the analysts noticed that there was what they might call a “kind-of” relationship between classes. Concepts like polymorphism aren’t usually employed in the analysis model. You see polymorphism more in the design model.</p> <p>So, frequently, when we have one of these “kind of” relationships in the analysis, and there are no overridden operations, implementing this as inheritance in the design might be overkill. A better approach might be to just use a flag attribute.</p>
3	<p>Quiz</p> 	<p>How would you handle this situation in design? There are two kinds of employees: blue collar and white collar. Blue collar employees are paid by the hour and get paid for overtime. White collar employees get an annual salary and are not paid for overtime. All employees have lots of attributes in common, such as name, social security number, hire date, address, etc. All employees have a pay() operation. There may be other objects that are linked to and from employee objects. In this diagram, they are shown in red.</p> <p>In analysis, when we hear that there are two “kinds of” employees, our first thought is to model this “kind of” relationship as generalization-specification. You can see this here. This is actually the design model in which the decisions about implementation of the links have been made</p>

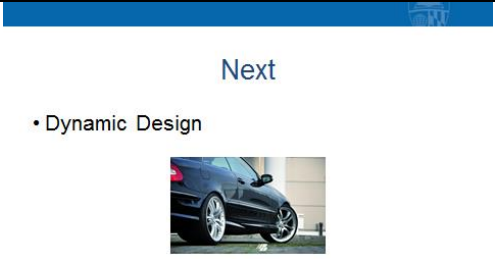
4	<p style="text-align: center;">The Problem</p>	<p>But there is a problem with this approach. Say we want to promote an employee from blue collar to white collar.</p> <p>You can't convert an object instance of one class into an object instance of another class.</p> <p>If you wanted to promote a blue collar employee object, you'd have to create a completely new white collar employee object instance, copy the attributes over from the old blue collar employee object instance, link the new white collar object in where the blue collar one was, and destroy the blue collar object. This is especially difficult if other objects point to the employee object (such as the Union object in our illustration here). We'd have to send a message to the union object to tell it that it is pointing to the wrong object, and tell it which new object to point to,</p> <p>In the instance diagram shown here, things in red are to be discarded. The things in green are to be created.</p> <p>This isn't a very elegant way to approach this.</p>
5	<p style="text-align: center;">A Better Way</p>	<p>Here's a better way. Instead of using generalization specialization to represent what is shared between white collar and blue collar classes, we use aggregation.</p> <p>The aggregate class (in this case, Employee) holds the information that doesn't change if the employment type changes.</p> <p>The Employee class delegates to the Contract class, which uses generalization-specialization to represent what is different between the two types of employment.</p> <p>This use of aggregation rather than generalization is called the state pattern. We'll see more patterns later in the course.</p>
6	<p style="text-align: center;">Much Easier to Change</p>	<p>Here's how it works.</p> <p>We want to change AI from a blue collar to a white collar employee. All of AI's information stays the same. All of the links to and from the Employee object are unchanged.</p> <p>We create a new instance of a WhiteCollar contract, point to it from the Employee object, and destroy the old BlueCollar contract object. No need to move data. No need to swizzle links.</p>

7	<div data-bbox="315 142 714 180" data-label="Section-Header"> <h3>Introducing Inheritance in Design</h3> </div> <div data-bbox="371 201 652 428" data-label="Diagram"> <p>The diagram illustrates the process of introducing inheritance. On the left, a single class box labeled 'Class' contains two sections: 'generalStuff' and 'specificStuff' in the top section, and 'generalBehavior' and 'specificBehavior' in the bottom section. A green arrow points to the right, where a new hierarchy is shown. The top box is labeled 'G-Class' and contains only 'generalStuff' and 'generalBehavior'. The bottom box is labeled 'S-Class' and contains only 'specificStuff' and 'specificBehavior'. An inheritance arrow points from 'S-Class' up to 'G-Class'.</p> </div> <td data-bbox="771 98 1445 531"> <p>Sometimes we can introduce inheritance at design time. During analysis, we identify classes in the problem domain. At design time, consider dividing some of the classes into two parts, a general part and a specific part.</p> <p>The general part becomes an abstract superclass. The specific part becomes a concrete subclass.</p> <p>More subclasses can be easily added later if you discover other specializations.</p> </td>	<p>Sometimes we can introduce inheritance at design time. During analysis, we identify classes in the problem domain. At design time, consider dividing some of the classes into two parts, a general part and a specific part.</p> <p>The general part becomes an abstract superclass. The specific part becomes a concrete subclass.</p> <p>More subclasses can be easily added later if you discover other specializations.</p>
8	<div data-bbox="479 579 547 613" data-label="Section-Header"> <h3>Next</h3> </div> <div data-bbox="292 636 732 690" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Internet Storefront Example – Design Class Diagram</li> </ul> </div> <div data-bbox="431 720 594 842" data-label="Image"> <p>A blue icon of a shopping cart on a computer screen, representing an Internet Storefront.</p> </div> <td data-bbox="771 531 1445 894"> <p>In the next video we will look at building the design class diagram for our Internet Storefront example.</p> </td>	<p>In the next video we will look at building the design class diagram for our Internet Storefront example.</p>

11-1	<div data-bbox="349 199 682 289" data-label="Section-Header"> <p>Internet Storefront Example</p> <p>Design Class Diagram</p> </div>	<p>In this video we will re-visit our running example, the Internet storefront.</p>
2	<div data-bbox="370 514 657 546" data-label="Section-Header"> <p>Analysis Class Diagram</p> </div>  <pre> classDiagram     class Customer {         name : String         checkInventory()         addLineItem()     }     class Product {         unitPrice : Int         name : String         computePrice(qty : Int) : Int     }     class ShoppingCart {         totalPrice : Int         addLineItem(l : LineItem)         updateTotalPrice()     }     class LineItem {         quantity : Int         calculateTotalPrice()     }     Customer "1" -- "1" ShoppingCart : fills     Product "1" -- "*" LineItem : describes     ShoppingCart "1" o-- "*" LineItem </pre>	<p>Here is what we have to work from. This is the analysis class diagram. We have to decide how to implement each of the three relationships. Two of them are associations, and one is an aggregation.</p> <p>We also made a decision when we were doing the dynamic modeling to add an operation to the Product class to calculate the total price. We thought it would be better for Product to have this responsibility because of the principle that says keep the operation with the data. Also, if perhaps we wanted to give a quantity discount, the information that would support that behavior would be best kept in the Product object.</p>
3	<div data-bbox="381 1018 649 1050" data-label="Section-Header"> <p>Design Class Diagram</p> </div>  <pre> classDiagram     class Customer {         name : String         cart : ShoppingCart         checkInventory(sku, qty)         addLineItem(sku, qty)     }     class Product {         unitPrice : Int         name : String         sku : Int         lookup : map(sku, Product)         computePrice(qty : Int) : Int         getProduct(sku) : Product     }     class ShoppingCart {         totalPrice : Int         lineItems : set(LineItem)         addLineItem(l : LineItem)         updateTotalPrice()     }     class LineItem {         quantity : Int         prod : Product         calculateTotalPrice()     }     Customer "1" -- "1" ShoppingCart : fills     Product "1" -- "*" LineItem : describes     ShoppingCart "1" o-- "*" LineItem </pre>	<p>Here are the four classes we saw in the analysis class diagram.</p> <ol style="list-style-type: none"> <li>1. Let's start with the Customer class, and add attributes and operations that we identified in our analysis modeling. Customer has a name</li> <li>2. There is an operation to check inventory</li> <li>3. There is an operation to add a line item.</li> <li>4. In the shopping cart, there is an attribute for storing the total price.</li> <li>5. We have the add line item operation in the shopping cart</li> <li>6. And the update total price operation.</li> <li>7. In the line item object, we will store the quantity.</li> <li>8. We also have the operation calculate total price in the line item.</li> <li>9. In the product object is where we decided to store the unit price</li> <li>10. We added the name attribute to the product class to help us with our instance diagrams</li> <li>11. There is the operation, compute price, in the product object</li> <li>12. Now let's work on the associations.</li> <li>13. There is the "fills" association between Customer and Shopping Cart.</li> <li>14. The shopping cart is filled by only one</li> </ol>



		<p>customer</p> <ol style="list-style-type: none"> <li>15. The customer fills only one shopping cart at a time.</li> <li>16. Let's store the references in the Customer object. We create an attribute called cart, which is a reference to an object instance of Shopping Cart.</li> <li>17. So the association becomes a directed association pointing from customer to shopping cart</li> <li>18. Now let's look at the aggregation between ShoppingCart and LineItem.</li> <li>19. Each line item is part of only one shopping cart.</li> <li>20. A shopping cart can contain many line items.</li> <li>21. The place to store this linkage information is the shopping cart. Since the collection of references is unordered by default, we will use a set. Later on, if we decide that we need to keep the references in some sort of order, we can easily change this to a list.</li> <li>22. We indicate our decision by putting an arrow head on the end of the association line.</li> <li>23. For the association between line item and product</li> <li>24. We said that the product describes line items.</li> <li>25. Each line item is described by one product</li> <li>26. A product can describe many line items.</li> <li>27. We should store this linkage information in the line item object, so we add a referential attribute called prod to the line item class.</li> <li>28. To indicate this choice, we add an arrow head on the association line.</li> <li>29. The last decision we have to make is how to find a product object instance based on a key value. Let's use the sku number of the product as its key. Sku stands for "stock keeping unit."</li> <li>30. We need a lookup operation. Let's call it getProduct. We pass in the sku value and it returns a reference to an instance of Product. This is a classifier scoped (or static) operation. We indicate this by using underlined font.</li> <li>31. To support this lookup operation, we need to store the lookup information in the class. The data model we will use is a map from sku number to product. Again, since this map is stored in the class, we</li> </ol>
--	--	---

		use underlined font to indicate that this is a classifier scoped attribute.
4		In the next video we will turn our attention to dynamic modeling at the design level.

12-1	<div data-bbox="406 199 617 241" data-label="Section-Header"> <h2>Dynamic Design</h2> </div> <div data-bbox="409 264 617 384" data-label="Image"> </div>	<p>In this video we will begin our investigation into dynamic modeling at the design level.</p>
2	<div data-bbox="357 514 665 556" data-label="Section-Header"> <h2>Design Models the Code</h2> </div> <div data-bbox="417 569 602 802" data-label="Diagram"> </div>	<p>Recall that in analysis, our main concern was to specify the classes inside the system and to understand how they collaborate to carry out the system operations. In design, our focus is on how this all happens in the code. The level of abstraction is higher than the code, so we won't be bogged down in code level details.</p> <p>As far as the dynamics are concerned, we are interested in not only what the operations are, and in what order they are executed, but also how they get executed. We care about an object having a reference to another object when it needs to send a message to that object. So we will need to design the access to the object reference in addition to the use of the reference to send the message.</p> <p>We will be using the UML Sequence Diagram notation to design these dynamics. This is a powerful notation that is more effective in modeling code level logic than activity diagrams are. The thing that made activity diagrams useful for analysis was that they weren't at the level of coding. They enabled us to stay at the proper level of abstraction of analysis. But in design, we need a notation that is closer to the level of abstraction of code, and that's the sequence diagram.</p> <p>Recall that design is an abstract view of the code. Certainly there are a lot of decisions to be made about how the objects behave to carry out the system operations. Many times these decisions are left until coding time. In other words, many people skip over the design step and make what we could call design decisions when coding.</p> <p>The problem with this approach is that in code you have a lot of other things to worry about. The level of abstraction is too low to make intelligent decisions about what are really design issues.</p>

	<p>In the previous video, we modeled the static aspect of the object relationships. Things such as referential attribute placement are appropriate at the design level and shouldn't be left to coding time.</p> <p>In this module we will be modeling the dynamics of the objects using the sequence diagram notation. As we get into this, you may begin to think that drawing sequence diagrams is more tedious than coding. There are two things I'll say about this.</p> <p>One is that, sure the diagrams are harder to create than code. But that's not necessarily a bad thing. Diagramming the dynamics forces you to think about the communication paths more clearly and you have a better chance of getting things right. Also, drawing sequence diagrams gets easier with practice.</p> <p>The more important thing about sequence diagrams is that they are drawn at a higher level of abstraction than code. In code you have to worry about a lot of low level things such as syntax, exception handling, and so forth. These kinds of details can be abstracted out in a sequence diagram. This allows you to concentrate on getting the messaging between the objects right.</p> <p>If you make a mistake, it is much easier to go back and fix things in the sequence diagrams than it is trying to fix the code. Programmers have a great reluctance to change code once it's written. In design we can easily make changes to our sequence diagrams. These same changes, if made at the coding level, would seem very huge. So we get things right in the design modeling, then it is relatively easy to write the code from the models that we have created.</p> <p>If all the hard decisions are made in the design, then coding should be easy.</p>
--	---

### Invoking the System Operation



For each system operation, we need to identify a controller. This controller may be one of the existing entity classes, or it may be a made up class, especially constructed for the purpose of controlling the system operation.

If we choose an existing entity class, it usually is one with the same name as the actor that initiates the system operation. For example, in our Internet storefront example, a customer actor sends a message to the system to add an item to an existing shopping cart. In our static modeling, we have identified a class called Customer. We could include an operation called “AddItem” in the Customer class to serve as the controller.

It is important to note that the controller is an operation, not an object or a class. Thus, if we choose the option of creating a special controller class, it probably would only have just that one operation, and no attributes. It would just be a holder for the controller operation.

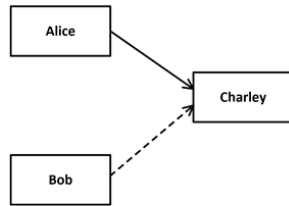
Next we have to concern ourselves with how this controller operation gets invoked. What happens when the customer actor invokes the addItem system operation?

There are two main ways the customer actor can communicate with the system. One is for the actor to send the actual customer object instance as a serialized object. This could be done with some kind of remote procedure call or remote method invocation mechanism. We would need a boundary class to handle communication to and from the actor. The boundary class would reconstitute the customer object and then execute the addItem operation.

More frequently, the communication between the actor and the boundary object is done via text messages. There are a number of Internet protocols that enable this. HTTP is probably the most common protocol. The actor issues a Get request that is executed by the boundary class in the server. The Get operation will parse the text and pull out the command and the parameters. In this case, the command would be addItem, and the parameters would include the unique identifier of the item and its quantity. We have used the stock-keeping unit, or SKU as the unique identifier of items in our catalog.

4	<div data-bbox="383 142 646 176" data-label="Section-Header"> <h3>Finding the Controller</h3> </div> <div data-bbox="423 237 602 350" data-label="Image"> </div>	<p>OK, the server has received a message from the customer actor asking to add an item to an existing shopping cart. We have decided that the Customer object will be the controller for this system operation. How does the system know which instance of the Customer class owns the shopping cart that we want to add the item to? There are two options.</p> <p>One option is that the boundary object receives the customer id from the actor and the specific instance of the Customer class must be located using a lookup operation, as we discussed earlier in static design.</p> <p>The other option is that the system has saved the reference to the specific customer object instance in something called a session object. The session object is created when the customer first logs in. The session object has its own unique identifier. Every time the customer actor interacts with the system, it passes this unique session id to the system. The boundary class can use the customer object reference in the session to send messages directly to the correct customer object instance.</p>
5	<div data-bbox="383 1018 643 1052" data-label="Section-Header"> <h3>What Happens Next?</h3> </div> <div data-bbox="298 1077 701 1125" data-label="Text"> <p>Sequence diagrams model the code more closely than activity diagrams do.</p> </div>	<p>OK, now we have figured out how the addItem operation will be invoked. How do we model what happens next? In analysis, we used activity diagrams, which were good at showing the sequence in which the objects performed their operations, and what these operations are. The notation did not support any good way to show parameters being passed or results being returned when these operations were being invoked. Sequence diagrams are good at this.</p> <p>We know that in order to send a message you must have the reference to the object you are sending it to, unless you are invoking a classifier scoped operation, which has global visibility. In analysis, we didn't concern ourselves with how these references are obtained. In design, this is a major concern.</p> <p>Sequence diagrams are very useful for modeling these kinds of details as well.</p>

### Obtaining an Object Reference



There are several ways in which a reference to an object instance may be obtained.

If the reference is stored as a referential attribute in an object, it may be used to send a message. Frequently the multiplicity of these referential attributes is greater than one. This means that there is a collection of referential attributes. You'll need some kind of key to enable a search for the correct reference.

In the picture, Alice knows Charlie's phone number. She can call Charlie whenever she needs to.

Sometimes the reference is returned to you as a result of a message you send to someone else. The message may be to an object instance that is able to provide the reference. Or maybe the reference is returned from the invocation of an object lookup operation, as we discussed earlier.

In our example here, Alice asks Bob to do a job that involves calling Charlie. Bob doesn't know Charlie's number, but Alice does. Let's assume Bob has caller ID and can call Alice back and ask for Charlie's number. He can then call Charlie and complete the job.

Another way to obtain a reference is to have it passed in as an argument when you are sent a message.



Alice asks Bob to do the job. She knows he will need to talk to Charlie so she gives Bob Charlie's number.



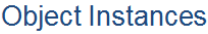
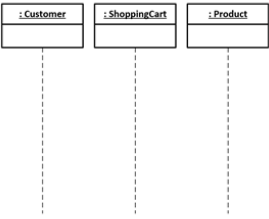
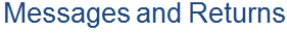
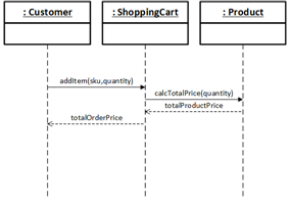
The fourth way to obtain the reference is if it is globally visible. One way this can be accomplished is to use the singleton design pattern. This is one of the patterns we will cover later in the course. In a nutshell, a singleton object or value can be stored as a static attribute of a class. The class will provide a static operation that will return the singleton reference or value.

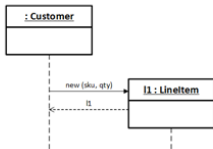
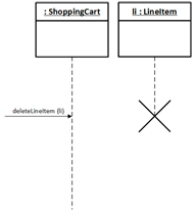
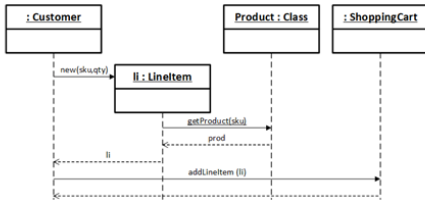
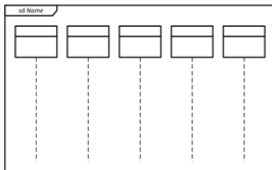
Suppose in our example, Charlie is really directory assistance. This service can be reached everywhere by dialing 411. It's globally visible. You don't need to look it up.




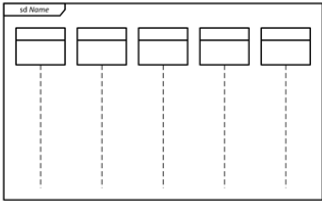

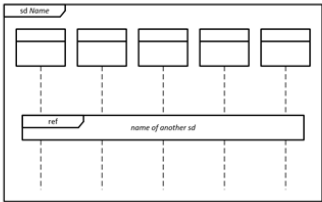

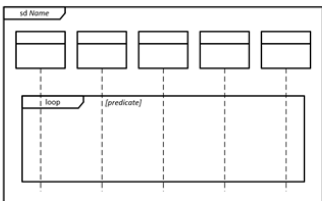

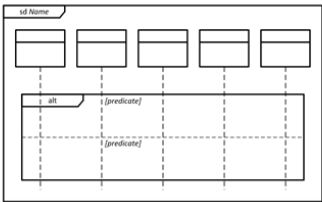
Of course there are other ways for Bob to get a message to Charlie. Since Alice knows how to

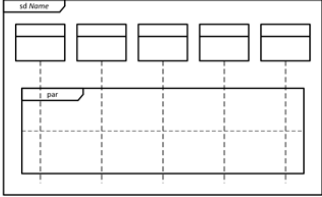
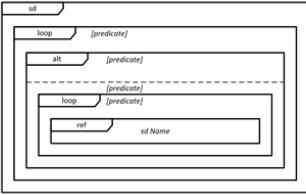





		<p>contact Charlie, Bob could ask Alice to forward a message to Charlie.</p> <p>Another mechanism to get a message to Charlie would be for Bob to post the message on the bulletin board. Charlie checks the board frequently, and receives the message from Bob eventually.</p>
7	 <p>Next</p> <ul style="list-style-type: none"> <li>• UML Sequence Diagrams</li> </ul> 	<p>In the next video we will cover some of the basics of UML sequence diagram notation.</p>

13-1	 	<p>In this video we will cover some of the most commonly used aspects of the UML sequence diagram notation.</p>
2	 	<p>Sequence diagrams show the object instances that are involved in the system operation. Object instances that exist at the start of the system operation are shown along the top of the diagram.</p> <p>To represent an object over time, we extend a dashed line, called a lifeline, below it. Time goes down the page.</p>
3	 	<p>A message is represented as a solid line with an arrow head from the lifeline of the sender to the lifeline of the receiver.</p> <p>The message is annotated with the name of the operation being invoked in the receiver object. Optionally, you can include arguments being passed to the operation.</p> <p>Usually we show the operation signature on the line, including the formal parameter names and their types.</p> <p>Sometimes a sequence diagram can be used to model a specific scenario. In this case, actual parameter values are shown on the messages.</p> <p>Returned results are indicated by a dashed line with an arrow head.</p> <p>Some people (and some CASE tools) place a little box on the lifeline between the invocation arrow and the return arrow. This is called a focus of control box. It isn't really needed if you show all returns. I personally prefer to show the returns because you can trace the flow of control through the scenario. You can easily see if there are any breaks in the sequence. There should be a continuous path from the beginning of the sequence to the end, if every invocation has a return.</p>


		<p>In some languages you can have operations that don't return a value. They just return control to the invoker. You would show this as a dashed arrow with no annotation on it.</p>
4	<p><b>Object Creation</b></p> 	<p>We can show the creation of an object instance by drawing the constructor message being sent to the object instance box at the point in the timeline when the object is being created.</p>
5	<p><b>Object Destruction</b></p> 	<p>We can show the destruction of an object by terminating its lifeline with a large X. In the example shown, the Shopping Cart receives a message to delete a specific line item. It does this by deleting the link to the line item object from its list. In a language like C++, the shopping cart object would need to send the destructor message to the line item. In a language like Java, if this was the only reference to that line item object, it would get garbage collected.</p>
6	<p><b>Classifier Scope Operations</b></p> 	<p>Since the lifelines all represent object instances, how do you show a message being sent to a class?</p> <p>We can do this if we realize that a class is technically an object instance of the meta-class, Class.</p> <p>Here we see how we can invoke the static operation getProduct by sending the message to the Product class.</p>
7	<p><b>Next</b></p> <ul style="list-style-type: none"> <li>• Structuring sequence diagrams</li> </ul> 	<p>In the next video, we will see how we can use frames to structure our sequence diagrams.</p>

14-1	 <h2>Structuring Sequence Diagrams</h2> 	In this video, we will see how to structure sequence diagrams through the use of frames.
2	 <h2>Frames</h2> 	A frame is a rectangle containing part (or all) of a sequence diagram. It has a pentagon in the upper left corner. This pentagon is called the heading. It indicates the purpose of the frame. There are many kinds of frames. Some of the more popular types are reference, looping, decision, and parallelism. Let's look at these in more detail. Sd stands for sequence diagram. This frame holds the entire diagram.
3	 <h2>Reference</h2> 	The frame with ref in the pentagon is a reference frame. It contains the name of another sequence diagram. It serves the same purpose as the rake in our activity diagrams.
4	 <h2>Looping</h2> 	To indicate looping, we use a loop frame. The top of the frame contains a predicate in square brackets. You can use keywords such as for and while in the predicate. Whatever messages are inside the frame are repeated based on the predicate.
5	 <h2>Alternation</h2> 	The word in the pentagon is alt, for alternative behaviors. There may be any number of parts. Each part has a predicate. You use the keyword else on the last one if you want. It is a good idea for all of the predicates to be disjoint and to cover the entire domain (i.e., OR together to form the Boolean TRUE).

6	<p style="text-align: center;">Parallel</p> 	<p>The keyword here is par, for parallel. There are multiple parts. Each part is executed in parallel with the others. The rules for parallel execution here are similar to those in activity diagrams. Each of the parallel regions must wait for all of the others to terminate before the frame terminates.</p>
7	<p style="text-align: center;">Nesting</p> 	<p>Frames may be nested to any depth. Frame borders may not cross – that is, a frame must be fully contained within another frame. This leads to a nice hierarchical structure in the diagrams.</p>
8	<p style="text-align: center;">Next</p> <ul style="list-style-type: none"> <li>Internet Storefront Example – design sequence diagram</li> </ul> 	<p>In the next video, we will develop the sequence diagram for the addItem system operation in our Internet storefront application.</p>

15-1	<p>Design Sequence Diagram for Internet Storefront Example "Add Line Item" System Operation</p> 	<p>In this video we will review the sequence diagram for the addItem system operation in our Internet storefront example.</p>
2	<p>Add Line Item</p> 	<p>You may want to print out the sequence diagram as some of the text may be too small to read on the screen.</p> <p>This sequence diagram incorporates a number of the concepts we discussed earlier in this module.</p> <ol style="list-style-type: none"> <li>1. The sequence begins with the Customer actor sending a message to the system asking it to add an item to an existing shopping cart. It passes in the sku for the item desired, the quantity, and the session id. This message is received by the Customer boundary object. The boundary object controls things for a while.</li> <li>2. The first thing the boundary object needs to do is to find the session object instance corresponding to the session id that was passed in from the actor. It sends the lookup message (getSession static operation) to the Session class, passing in the session id and getting back the reference to the object instance of Session that contains the objects pertaining to the current customer.</li> <li>3. Then the boundary object sends a message to the session object to get the reference to the correct instance of the Customer class.</li> <li>4. Now that we have the reference to the customer object, we can send it the message addItem, passing in the sku and the quantity. From this point, the flow is quite similar to what we discussed earlier.</li> <li>5. The first thing the customer object will do is to find out if there is sufficient quantity of the product in stock. In our design, we are using a proxy object to help us communicate with the external Inventory subsystem. A proxy is an object that stands in for something else that may be otherwise quite complicated to</li> </ol>

		<p>communicate with. The message we send to the proxy is "isSufficientQuantity," passing in the sku and quantity. The proxy talks to the actual inventory subsystem to determine whether there is enough stock to fulfill the order. It returns a Boolean result.</p> <ol style="list-style-type: none"> <li>6. If there is sufficient quantity, we execute the activity shown in the upper part of the "alt" frame. If not, we would execute the lower part of the frame.</li> <li>7. In the upper part of the frame, the first thing that we do is to obtain the reference to the object instance of Product, based on the sku. The lookup operation is a static operation in the Product class. We pass in the sku, and it returns a reference to the corresponding object instance of the Product class.</li> <li>8. We'll use this product reference to construct a new instance of the LineItem class. We pass the Product object reference and the quantity to the constructor.</li> <li>9. The line item object is created. We see that the box representing this object instance appears at the point in the timeline when it is created. The constructor returns a reference to the line item object.</li> <li>10. Now we add the line item to the shopping cart. The customer object already has a link to the shopping cart object, since the cart was created in a previous system operation, and there was a precondition on the current system operation that the cart exists. So the customer object sends the message addLineItem to its shopping cart. It just passes in the reference to the line item object on this message.</li> <li>11. The shopping cart adds the reference to the line item to its list of line items, and returns.</li> <li>12. The next thing that happens is the pricing. The customer object asks the shopping cart to update its total price.</li> <li>13. The shopping cart will ask the line item to calculate its total price.</li> <li>14. As we discussed before, we are going to let the Product object compute the total price because we decided to put the information about unit price in the product, and so the job of multiplying the unit price by the quantity goes to the product object.</li> </ol>
--	--	--

		<p>15. The price information is returned to the line item, which returns it back up to the shopping cart. The cart adds that price to its total price, and then returns back to the customer object.</p> <p>16. The customer then returns control back to the customer boundary object, with a status of "ok."</p> <p>17. In the else part of the frame, there is not sufficient quantity of the product in stock. The customer object returns control back to the customer boundary object with the status, "fail."</p> <p>18. In either case, the boundary object returns back to the actor, forwarding the status it received from the customer object.</p> <p>And that completes the sequences of actions for the design of the system operation for adding a line item to an existing shopping cart.</p>
3	 <h3>End of Module</h3> <ul style="list-style-type: none"> <li>• You should now be able to do these things: <ul style="list-style-type: none"> <li>○ Perform dynamic object-oriented analysis modeling and express the models using UML activity diagram notation</li> <li>○ Create static object-oriented design models of the code using UML class diagram notation</li> <li>○ Create dynamic object-oriented design models of the code using UML sequence diagram notation</li> </ul> </li> </ul>	<p>And we've come to the end of the module. You should now be ready to do these things:</p> <p>You should be able to perform dynamic object oriented analysis, using UML activity diagram notation for your modeling.</p> <p>You should be able to create static and dynamic object-oriented design models using UML class diagrams and sequence diagrams.</p> <p>If you haven't already started, you should join the forum to discuss this week's question.</p> <p>You can now take the quiz for this module.</p> <p>You can work with your team to perform dynamic object-oriented analysis with activity diagrams.</p> <p>Then you can work with your team to perform object-oriented design by creating design class diagrams and sequence diagrams for the important system operations. Look at the team exercise instructions for more detail.</p>