

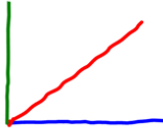







1.1	 <h2 data-bbox="300 205 727 235">Analysis and Design Fundamentals</h2>	<p>Welcome to the module on Analysis and Design Fundamentals.</p> <p>This is the basis for the specific modules on object oriented analysis and design that follow.</p> <p>We have already discussed some of this before in the course, but it is important that we all have these concepts down pat before we continue.</p>
2	<h2 data-bbox="457 520 571 550">Overview</h2>  <p data-bbox="311 697 690 718">Requirements    Analysis    Design    Code</p>	<p>Our starting point in our software development activity is the requirements specification. We view this as a black box. The requirements specification says WHAT our system must do, but not HOW.</p> <p>Our ultimate goal, of course, is to generate some code.</p> <p>Before we can code, we need to create a design. Think of the design as a high level model of the code.</p> <p>There is still quite a large semantic gap between the level of abstraction in the requirements and the level of abstraction in the design. This gap is filled in by performing software analysis.</p>
3	<h2 data-bbox="451 1056 581 1085">Objectives</h2> <p data-bbox="300 1108 690 1155">Upon completion of this module, you will be able to:</p> <ul data-bbox="300 1159 722 1302" style="list-style-type: none"> <li>• Explain the importance of each of the three bases of analysis and design: Structure, Information, and Behavior.</li> <li>• Utilize structural, informational, and behavioral models in analysis and design</li> <li>• Apply analysis and design principles such as abstraction, modularity, and encapsulation.</li> </ul>	<p>When we are finished with this module, you will be able to do these things.</p>
4	<h2 data-bbox="474 1434 558 1463">Outline</h2> <ul data-bbox="300 1474 633 1669" style="list-style-type: none"> <li>• Structure, Information and Behavior</li> <li>• Levels of development</li> <li>• Modeling structure</li> <li>• Modeling information</li> <li>• Modeling behavior</li> <li>• Modularity</li> <li>• Abstraction</li> <li>• Encapsulation</li> </ul>	<p>There are eight more videos in this module. We start with a discussion of the three dimensions of software analysis and design (Structure, Information, and Behavior).</p> <p>We will then discuss the levels of development in more detail.</p> <p>Then we will zoom in on each of the three dimensions (modeling structure, information and behavior).</p> <p>Then we will turn our focus to software analysis and design principles: modularity, abstraction, and encapsulation.</p>

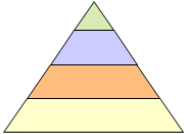
### Next

- The three dimensions of software specification: **Structure**, **Information** and **Behavior**.


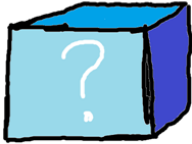




So, in the next video you will see the three dimensions of software specification: structure, information and behavior.

2.1	 <p>Structure, Information, and Behavior</p>	<p>In this video we will examine the three dimensions of software specification: structure, information and behavior.</p> <p>In a sense, we have already seen these dimensions, if you think about what we specified in the software requirements.</p> <p>To completely describe software, we need to specify all three dimensions.</p>
2	<p>Structure</p> <ul style="list-style-type: none"> <li>• What the pieces are</li> <li>• How they fit together</li> </ul> 	<p>We view our software as being made up of components, or modules. When we describe the structure of our system, we identify the modules and specify how they are connected.</p> <p>Each module will be connected to at least one other module. This is some sort of dependency relationship. The dependency may be due to the fact that one module has some information that is needed by other modules. The dependency may also be due to the fact that one module provides some services or functions that are needed by other modules.</p>
3	<p>Information</p> <ul style="list-style-type: none"> <li>• What the system remembers over time</li> </ul> 	<p>Each module may hold some information. In the requirements we discovered what information is remembered by the system. In analysis and design we will allocate the responsibility for storing that information to individual modules.</p>
4	<p>Behavior</p> <ul style="list-style-type: none"> <li>• What the system does</li> </ul> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">  <p>Imperative</p> </div> <div style="text-align: center;">  <p>Declarative</p> </div> </div>	<p>The third dimension is behavior. It is what the system is able to do. The largest part of the requirements specification was involved with describing this behavior at the system level. In analysis and design we will allocate the functional capabilities to individual modules.</p> <p>There are two main ways of describing behavior.</p> <p>Imperative descriptions of behavior always involve an element of time. They describe what changes occur in what sequence. These are the most common ways of describing behavior. Examples are use case specifications (flows of events), activity diagrams (which look like flow charts), and sequence diagrams. Activity diagrams and sequence diagrams are UML models, and we will explore UML notation in more detail when we get into object oriented analysis and design later in the course.</p>

		<p>Declarative behavior specifications are less commonly used. We have already seen some declarative techniques. State transition diagrams and matrices are declarative. So are decision tables. Declarative techniques don't indicate the order in which events occur, but rather, they specify rules that govern the behavior of software.</p>
5	<div> <div></div> <div>Next</div> <div> <ul style="list-style-type: none"> <li>• Levels of development</li> </ul>  </div> </div>	<p>In the next video, we will go into more detail on the levels of abstraction in software development.</p>

3.1	<div data-bbox="368 201 656 233" data-label="Section-Header"> <h3>Levels of Development</h3> </div> <div data-bbox="386 254 636 428" data-label="Diagram"> <pre> graph TD     A[Problem Statement] --&gt; B[Requirements Specification]     B --&gt; C[Analysis Model]     C --&gt; D[Design Model]   </pre> </div>	<p>In any software effort except the most trivial, there is a huge semantic gap between the description of the problem to be solved, and the code that will hopefully solve it.</p> <p>If you try to jump the semantic gap in one step, you are applying what is sometimes called the big bang approach to software development. It almost never ends well.</p> <p>It is much better to take a careful step by step approach to the development of software. We do this one level of abstraction at a time.</p> <p>We see four main levels of abstraction in this development process. They are illustrated in this graphic. Let's look at each level individually.</p>
2	<div data-bbox="389 751 634 783" data-label="Section-Header"> <h3>Problem Statement</h3> </div> <div data-bbox="293 808 641 865" data-label="List-Group"> <ul style="list-style-type: none"> <li>• What is the problem to be solved?</li> <li>• Avoid "jumping to solution."</li> </ul> </div> <div data-bbox="469 879 553 997" data-label="Image"> </div>	<p>We start with a description of the problem to be solved by the software system.</p> <p>A typical mistake that a lot of people make is to do what I call "jumping to solution." Instead of describing the problem to be solved, we describe solutions. I know I am guilty of this in my everyday life. I see the ads for new computers, and think I'd like to replace my old laptop with a new one. I find the one I want to buy and say, "I need this computer." I have just proposed a solution to an unstated problem. What I really should have said was "I WANT this computer." I have jumped to solution. What is the problem I'm trying to solve here? If I'm honest with myself, I probably don't need such a fancy machine. In fact, I probably could get by just fine with the computer I have now.</p> <p>When customers bring us in to help them solve a problem, they frequently suggest solutions. Our first job should be to ask a simple question: "WHY?" What is the problem you are trying to solve?</p> <p>Other times we will try to force fit a solution to the customer's problem. We don't take the time to understand the problem completely. We make assumptions that may or may not be accurate. We have jumped to solution.</p> <p>Things are not going to go very well if we proceed to writing requirements for a system that doesn't solve the customer's real problem.</p>

3	<div data-bbox="272 98 760 130" data-label="Page-Header">  </div> <div data-bbox="347 149 683 180" data-label="Section-Header"> <h3>Requirements Specification</h3> </div> <div data-bbox="298 205 612 228" data-label="List-Group"> <ul style="list-style-type: none"> <li>• How will we solve the problem?</li> </ul> </div> <div data-bbox="420 273 610 415" data-label="Image">  </div>	<p>Once we understand the problem to be solved, we can specify the solution as a requirements specification. In the requirements, we view the system as a black box. We know there is going to be some code in there eventually, but our job in requirements is to describe WHAT the structure, information and behavior are, rather than HOW they are implemented. (That's analysis and design.)</p>
4	<div data-bbox="272 457 760 489" data-label="Page-Header">  </div> <div data-bbox="420 516 607 548" data-label="Section-Header"> <h3>Analysis Model</h3> </div> <div data-bbox="298 573 677 651" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Identify and specify the main data and functional components <ul style="list-style-type: none"> <li>◦ Noun-verb analysis</li> </ul> </li> </ul> </div> <div data-bbox="464 680 561 758" data-label="Image">  </div>	<p>During analysis, we open the covers of the black box and peer inside. Our job is to find the data and functional components inside the system.</p> <p>We view analysis as a discovery process. There are two ways of gaining new information. One is invention and one is discovery. Thomas Edison invented the light bulb. It never existed until he invented it. George Priestly discovered oxygen. It always existed, but no one knew about it until he discovered it.</p> <p>The data and functional components that we identify during analysis are discovered rather than invented. Different teams doing analysis starting with the same set of requirements, will typically discover the same set of information and functional components. There's not a lot of room for creativity in analysis. Our job is to find the components inside the system rather than to invent them.</p> <p>A technique that is sometimes useful is called noun-verb analysis. The nouns that appear in the requirements and problem statement indicate the things in the problem. Frequently these things reflect the names of the components in the analysis. They also may indicate the information that is managed by the components inside the system. The verbs that we use indicate the functions performed by the components inside the system. This technique works best when used informally. I wouldn't be too pedantic about it. You will find that in a typical set of requirements there are way too many nouns and verbs. You can easily get swamped if you try to deal with every noun and verb.</p>

5	<div data-bbox="365 142 662 178" data-label="Section-Header"> <h3>Model the Relationships</h3> </div> <div data-bbox="321 199 698 367" data-label="Diagram"> <p>Static Models</p> <p>Dynamic Models</p> <p>[1] <a href="http://www.pcdix.com/term.asp?id=895">http://www.pcdix.com/term.asp?id=895</a>  [2] <a href="http://oneflyandsports.blogspot.com/2009/12/saints-beat-patriots-on-line-of-them">http://oneflyandsports.blogspot.com/2009/12/saints-beat-patriots-on-line-of-them</a></p> </div> <td data-bbox="771 98 1451 667"> <p>During analysis, after finding the components we model their relationships. There are two types of relationships between components. Static relationships show the structure of the solution. Dynamic models show what happens when the system is working.</p> <p>I'm using a football analogy here. The static model shows where each player is positioned at the start of the play. There is no movement here. It is purely structural.</p> <p>The dynamic model shows what happens over time. The diagram shows where each player is supposed to move, and who has the ball when, and whether the ball is carried or passed and received.</p> </td>	<p>During analysis, after finding the components we model their relationships. There are two types of relationships between components. Static relationships show the structure of the solution. Dynamic models show what happens when the system is working.</p> <p>I'm using a football analogy here. The static model shows where each player is positioned at the start of the play. There is no movement here. It is purely structural.</p> <p>The dynamic model shows what happens over time. The diagram shows where each player is supposed to move, and who has the ball when, and whether the ball is carried or passed and received.</p>
6	<div data-bbox="430 716 597 751" data-label="Section-Header"> <h3>Design Model</h3> </div> <div data-bbox="435 783 597 976" data-label="Diagram"> <pre> graph TD     Requirements[Requirements] -- "Refinement of" --&gt; Analysis[Analysis]     Analysis -- "Abstraction of" --&gt; Design[Design]     Design -- "Abstraction of" --&gt; Code[Code] </pre> </div> <td data-bbox="771 667 1451 1451"> <p>We view the analysis model as a lower level of specification. If the requirements specify the system as one large black box, the analysis models identify the components within the system and show their specifications. The requirements show WHAT the structure, information and behavior are for the entire system. The analysis identifies WHAT the components inside the system are, WHAT their structural relationships are, and WHAT their information and behavioral properties are. So the analysis is a refinement of the requirements. But the analysis is still a specification model.</p> <p>Design should be thought of as an abstract model of the code. It describes HOW the code works.</p> <p>Design is an invention process. For each component specification in the analysis, we need to invent an implementation. There may be many choices for implementing each analysis component. Different teams may come up with different designs given the same analysis model.</p> </td>	<p>We view the analysis model as a lower level of specification. If the requirements specify the system as one large black box, the analysis models identify the components within the system and show their specifications. The requirements show WHAT the structure, information and behavior are for the entire system. The analysis identifies WHAT the components inside the system are, WHAT their structural relationships are, and WHAT their information and behavioral properties are. So the analysis is a refinement of the requirements. But the analysis is still a specification model.</p> <p>Design should be thought of as an abstract model of the code. It describes HOW the code works.</p> <p>Design is an invention process. For each component specification in the analysis, we need to invent an implementation. There may be many choices for implementing each analysis component. Different teams may come up with different designs given the same analysis model.</p>
7	<div data-bbox="300 1501 727 1537" data-label="Section-Header"> <h3>The Importance of Analysis and Design</h3> </div> <div data-bbox="321 1585 698 1717" data-label="Text"> <p>"The beginning of wisdom for a computer programmer is to recognize the difference between getting a program to work, and getting it RIGHT."</p> <p>-- Michael A. Jackson</p> </div> <td data-bbox="771 1451 1451 1946"> <p>Here is a good quote from Michael Jackson (not that Michael Jackson). He is a British software engineer.</p> <p>The beginning of wisdom for a computer programmer is to recognize the difference between getting a program to work and getting it RIGHT.</p> <p>Anyone can hack at a program long enough to eventually get something to work well enough to get it out the door. But it probably won't be easy to understand, or maintain, or use. It will probably have low reliability. The customers will most likely be dissatisfied. (Remember our discussion of the software crisis?)</p> </td>	<p>Here is a good quote from Michael Jackson (not that Michael Jackson). He is a British software engineer.</p> <p>The beginning of wisdom for a computer programmer is to recognize the difference between getting a program to work and getting it RIGHT.</p> <p>Anyone can hack at a program long enough to eventually get something to work well enough to get it out the door. But it probably won't be easy to understand, or maintain, or use. It will probably have low reliability. The customers will most likely be dissatisfied. (Remember our discussion of the software crisis?)</p>

### Next

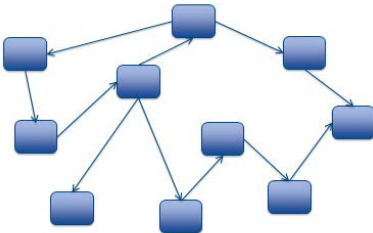
- Modeling the structure



In the next three videos, we will discuss modeling structure, information, and behavior.



4.1	<div data-bbox="399 199 630 233" data-label="Section-Header"> <h3>Structure Modeling</h3> </div> <div data-bbox="428 258 589 401" data-label="Diagram"> </div>	<p>We are now going to look in more detail at how we model structure, information and behavior. Let's start with structure modeling. We'll see that there are several stereotypical structures that appear over and over in software (and in real life).</p>
2	<div data-bbox="440 514 587 546" data-label="Section-Header"> <h3>Hierarchical</h3> </div> <div data-bbox="323 554 722 808" data-label="Diagram"> </div>	<p>Perhaps the most ubiquitous structure in the hierarchy. We can view this structure as a graph in which there are no cycles. In other words, there is no path from any node back to itself. This is referred to as a lattice structure in graph theory. It is possible to draw a lattice so that all of the edges point in the same direction. In the diagram shown here, the edges all point downwards. In doing so, the nodes fall into tiers or layers.</p> <p>There is a special case of a lattice in which each node has only one parent. This is called a tree structure.</p> <p>We see hierarchies all over. A hierarchy could represent the dependencies between functions. It could represent the inheritance structure in an object oriented system. It could represent part-whole data structures. Directory structures are hierarchical.</p> <p>What makes hierarchies so appealing is that they mimic structures we encounter every day. Typical organizational structures in, for example, the military, are hierarchies. The dependencies all flow in the same direction. We frequently organize information in a hierarchical fashion. For example, recall from biology, we organize properties of living things into kingdom, phylum, class, order, family, genus and species.</p>
3	<div data-bbox="462 1512 565 1543" data-label="Section-Header"> <h3>Layered</h3> </div> <div data-bbox="302 1577 730 1793" data-label="Diagram"> </div>	<p>If we view a system as being made up of components or modules, it might be useful to organize those components into layers based on their reuse characteristics. We see at the base layer a lot of rather small components that do very specific things. These are the capabilities that are typically found in programming language class libraries or in the operating system. They are not specific to any particular domain and are highly reusable. Examples of base layer components might be collection classes, UI builder components, or networking capabilities.</p> <p>The components in the next layer up are what we might call common services. These components still have quite broad reuse potential. They don't come</p>

		<p>from languages or operating systems, so they must be built on top of the language and operating system facilities. Examples of common services might be message queueing facilities or database access capabilities.</p> <p>The next layer up consists of domain specific components. These are built especially for implementing systems in particular domains. Much of the time these components are harvested from existing systems and are sort of polished up to make them more reusable. The reuse potential for one of these components is not as great as for components in lower layers, but the return on investment of its reuse is quite high. We might see domain specific components for specialized domains such as banking or insurance.</p> <p>The top layer is the application layer. These are not really reusable components. They represent the systems that are built using the capabilities of all of the components in the layers below them.</p> <p>Note that, just like in the hierarchical organization, all dependencies are downward. You would never have a component in a lower layer be dependent on one in a higher layer.</p> <p>Layered architectures are quite common in computing. Think about TCP-IP or the OSI framework, for example.</p>
4	<p style="text-align: center;">Network</p>  <pre> graph TD     A[ ] --&gt; B[ ]     A --&gt; C[ ]     A --&gt; D[ ]     B --&gt; E[ ]     C --&gt; F[ ]     D --&gt; G[ ]     E --&gt; H[ ]     F --&gt; I[ ]     G --&gt; I[ ]     H --&gt; J[ ]     I --&gt; J[ ]   </pre>	<p>A more general graph structure is what we might call a network. It is different from a lattice in that it can't always be structured in layers. It may even have cyclic dependencies.</p> <p>Network structures are more common than you might think. All of the UML models that have nodes and edges (such as state transition diagrams, activity diagrams, and class diagrams) are networks. We will look in more detail at these models later in the course.</p> <p>The obvious example of a network in computing systems is the Internet. Computing nodes are connected by communication links.</p>

5



## Next

- Information Modeling



Next, we will discuss ways in which we may model information.

5.1	<div data-bbox="378 199 643 233" data-label="Section-Header"> <h2>Information Modeling</h2> </div> <div data-bbox="410 254 617 409" data-label="Image"> </div>	<p>The second dimension of software specification is information. Let's look at some ways to model information within the system.</p>
2	<div data-bbox="337 512 685 546" data-label="Section-Header"> <h2>Entity Relationship Diagram</h2> </div> <div data-bbox="386 573 634 791" data-label="Diagram"> <p>Source: Roger Pressman, Software Engineering, A Practitioner's Approach, 3rd Edition, page 201</p> </div>	<p>A course grained view of the data can be represented by an entity relationship diagram, or ERD. The notation shown here was developed by Peter Chen in 1976, and is sometimes called Chen notation. The boxes represent entities, about which the system must store information. The diamonds are called relationships. Each relationship relates two entities. The symbols on the ends of the lines denote cardinality. The variant shown here was developed by James Martin for a technology called Information Engineering. For example, on the relationship, "Builds," one manufacturer builds multiple cars. Each car is built by one manufacturer.</p> <p>This notation has been superseded by the Unified Modeling Language for the most part. So let's look at the same diagram in UML.</p>
3	<div data-bbox="386 1119 636 1152" data-label="Section-Header"> <h2>UML Class Diagram</h2> </div> <div data-bbox="329 1180 696 1396" data-label="Diagram"> </div>	<p>This is a UML class diagram. The relationships are designated by lines connecting the boxes. The boxes represent the classes of information. The cardinalities are called multiplicities in UML, and are represented by numbers. A star indicates a multiplicity greater than one. If you look at the "builds" relationship between Manufacturer and Car, you can see that one manufacturer can build many cars, but each car is built by one manufacturer.</p> <p>Some additional information is included in the UML class diagram. For each class, not only do we see its name, but we also see its attributes and operations. In Chen notation, attributes would be represented as boxes with an elliptical shape attached to the entities. The operations are only shown on UML class diagrams, and are not present in entity-relationship diagrams.</p>

## Data Models

	Homogeneous	Duplicates	Ordered	Access
Set	Yes	No	No	Random
List	Yes	Yes	Yes	First, Last
Array	Yes	Yes	Yes	Indexed
Record	No	Yes	Yes	By Field Name
Map	Yes	Yes	No	By Key Value
Scalar	--	--	--	--

On a lower level, we need to be able to describe the properties of the information contained within a module or component, without getting into the design of the data. We will use mathematical models of the data.

For example, if the information contained in a module is a collection that doesn't need to be ordered, the collection can usually be modelled as a set. The elements within the set would all be of the same type (or class), so the set is homogenous. One thing about a set in mathematics is that there are no duplicate values within the set. So, for instance, if we have a set of integers, there could be at most one five in the set. Of course there is no left to right ordering of the elements of a set. The elements are either in the set or they're not. One way to access the elements of a set would be to randomly select an element.

A list is like a set except it is ordered. The elements can be put in left to right order. Also, duplicates are allowed. If we have a list of integers, there may be any number of fives in the list. We may access the elements of a list by the first or last element. Some notations allow you to represent the first element and the remainder of the list after the first element.


An array is a fixed size collection. Access to the elements is by an index value. While sets and lists have dynamic size: you can add or remove elements, an array is fixed. You can have slots in the array that don't have any values in them, but there are still slots.


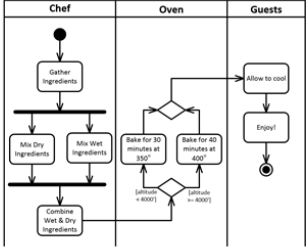
A record is a collection of elements of possibly different types. Each element is accessible by a field name. We find records in older languages like Pascal or Ada. A record is similar to a struct in the C programming language.

A map is a key transformation data structure. There are two data types involved with a map. The first one is the key data type. The second one is the value data type. Sometimes we use the term hash map, but this implies a particular implementation, which is what we're trying to avoid here. A typical use of the map data model is where you might have a collection of objects that are accessible by a unique name.

A scalar is a data model that has no sub-parts. An integer is a scalar. So is a string in many languages.

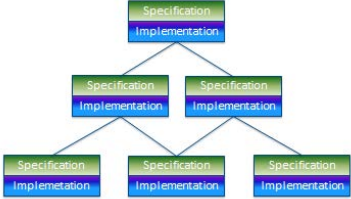

There are other models that you might find in particular programming languages or design

		notations.
5	<div><div></div><div>Next</div><div><ul style="list-style-type: none"><li>• Behavior Modeling</li></ul></div><div></div></div>	<p>The last dimension of our modeling is the behavior dimension. That's coming up in the next video.</p>


6.1	<h2 style="text-align: center;">Behavior Modeling</h2> 	<p>We saved the most powerful notation for last. Behavior modeling. When most people think about analysis or design, they generally think about modeling the behavior of the system. We just want to make sure you also model the structure and information as well.</p> <p>We will explore some techniques for modeling the behavior of the system.</p> <p>There are two types of behavior models: Imperative and Declarative.</p>
2	<h2 style="text-align: center;">Imperative Models</h2> <ul style="list-style-type: none"> <li>• How things happen</li> <li>• Examples:             <ul style="list-style-type: none"> <li>○ Use Cases Specifications</li> <li>○ Activity Diagrams</li> <li>○ Sequence Diagrams</li> </ul> </li> </ul>	<p>Imperative models describe how things happen. We describe change over time. Usually this change is the change in state of the system, but it also may include input and output events.</p> <p>We will look at three examples of imperative models. There are certainly more than these three, but these are the most popular.</p>
3	<h2 style="text-align: center;">Use Case Specification</h2> <p>[from ATM system: "Withdraw" use case]</p> <p><i>Precondition:</i> Customer has been authenticated</p> <p><i>Flow of events:</i></p> <ul style="list-style-type: none"> <li>○ Customer selects "Withdraw" and enters the desired account and the amount.</li> <li>○ [debit account] System debits the customer's account.</li> <li>○ [hardware operational] System opens the cash drawer and dispenses the proper currency.</li> </ul>	<p>We have already seen use case specifications when we did our requirements modeling. The flow of events is an imperative model describing a sequence of events. These events can be inputs from the actors to the system, outputs from the system to actors, or things that happen inside the system. We also describe alternate flows to take care of all of the variations in behavior of the system.</p> <p>This notation could also be used to describe what happens inside the system during analysis or design, but typically it is reserved only for requirements specification.</p>
4	<h2 style="text-align: center;">Activity Diagram</h2> 	<p>An activity diagram is a Unified Modeling Language diagram that looks like a traditional flowchart. The rectangular boxes with the rounded corners are called actions. Think of an action as some behavior that is performed by a component.</p> <p>The components that perform the actions are frequently shown at the tops of partitions, or so-called "swim lanes." Here we see three things that can perform behaviors: Chef, Oven and Guests. This is the activity diagram for baking a cake.</p> <p>The actions are connected by arrows that indicate the order in which the actions are performed. The actions are shown in the swim lane of the component that performs them.</p> <p>Diamonds indicate that the flow may take different</p>

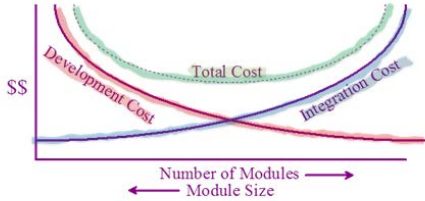
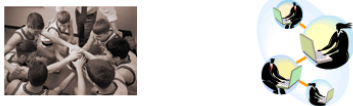
		<p>paths depending on the value of the conditions that annotate the flows. These conditions are called guards. So we will bake the cake at different temperatures and for different times depending on the altitude.</p> <p>The bold bars indicate asynchronous activities. In this model, the activities of mixing the dry ingredients and mixing the wet ingredients may be done in either order, or, in fact, at the same time if there is more than one chef or the chef is ambidextrous.</p>
5	<h3>Sequence Diagram</h3> <pre> sequenceDiagram     participant C as :Customer     participant O as :Order     participant P as prod:Product     participant PC as Product:Class     C-&gt;&gt;O: addItem(ku,quantity)     O-&gt;&gt;P: getProd(ku)     P-&gt;&gt;PC: calcTotalPrice(quantity)     PC--&gt;&gt;P: totalProductPrice     P--&gt;&gt;O: totalOrderPrice     O--&gt;&gt;C: totalOrderPrice   </pre>	<p>The sequence diagram is one of the more powerful notations for illustrating dynamic behavior.</p> <p>The boxes at the top are the components that perform the activities. The lines below the components represent that component over time. An arrow between two timelines represents a message from one component to another. We can show returns as dashed lines. The order of the arrows, top to bottom, indicates the sequence in which the messages are sent or returns occur.</p> <p>There is a lot more to sequence diagram notation. We will see a lot of it when we get into object oriented analysis and design.</p>
6	<h3>Declarative Models</h3> <ul style="list-style-type: none"> <li>• <b>What happens</b></li> <li>• <b>Examples:</b> <ul style="list-style-type: none"> <li>○ Pre/post conditions</li> <li>○ Decision tables</li> <li>○ State Transition Diagrams</li> </ul> </li> </ul>	<p>Declarative models don't describe how things happen, but they only specify rules for what will happen in various circumstances.</p> <p>We are already familiar with decision tables and state transition diagrams from our discussion of requirements modeling. These models can also be used during analysis and design.</p> <p>Pre and post conditions can be used to declare what changes but not how it changes. We saw their use in use case descriptions. They may also be used to describe the behavior of an operation in an individual component. (For example, a method in a class.)</p>
7	<h3>Stepwise Refinement</h3> <ul style="list-style-type: none"> <li>• <b>Divide and conquer</b></li> <li>• <b>Repeat as long as there are specifications to be implemented</b> <ul style="list-style-type: none"> <li>○ Choose a specification to be implemented</li> <li>○ Write the implementation, invoking lower level, invented components (if necessary)</li> <li>○ Write specifications for these components</li> </ul> </li> </ul>	<p>A useful technique for creating a modular structure is called stepwise refinement. It is similar to the old divide-and-conquer approach to problem solving. If you are faced with a problem that is too hard to solve, we break it into a collection of smaller problems. Maybe we can solve those. If not, we can repeat the process to break these problems down into even smaller problems. Eventually we can solve the whole thing by combining all of the solutions together.</p> <p>In stepwise refinement, our goal is to implement a specification. The starting specification is the requirement specification for the system. For any</p>




		<p>specification, we devise an implementation that typically involves the use of lower level components, whose specifications must be invented. Thus we end up with more specifications to implement in subsequent steps of the process.</p>
8	<p>Stepwise Refinement</p> 	<p>Here is a pictorial representation of stepwise refinement. We start with a specification. We implement it in terms of lower level modules, which we specify. Now we have more specifications to implement. We implement the specifications using this same process, one at a time. At the lowest level, we may decide that the capability can be achieved without inventing other lower level modules, and the stepwise refinement process stops.</p>
9	<p>Next</p> <ul style="list-style-type: none"> <li>• Analysis and design principles <ul style="list-style-type: none"> <li>◦ Modularity</li> <li>◦ Abstraction</li> <li>◦ Encapsulation</li> </ul> </li> </ul> 	<p>That covers the three dimensions of specification: structure, information and behavior.</p> <p>Next we will revisit some of the analysis and design principles we introduced in the first module of this course.</p>


<p>7.1</p>	<div data-bbox="321 199 703 235" data-label="Section-Header"> <h3>Analysis and Design Principles</h3> </div> <div data-bbox="329 254 482 331" data-label="List-Group"> <ul style="list-style-type: none"> <li>• <b>Modularity</b></li> <li>• Abstraction</li> <li>• Encapsulation</li> </ul> </div> <div data-bbox="561 252 683 375" data-label="Image"> </div>	<p>Recall that in the first module of this course we introduced several software engineering principles. We'd like to focus in on three main principles that will help us with analysis and design.</p> <p>Modularity means breaking something complicated into pieces, making things more manageable.</p> <p>Abstraction is where we model the essential properties of something that might otherwise be quite complex.</p> <p>Encapsulation is the separation of information about a module into two parts, a specification, which is an abstraction of what its capabilities are, and an implementation, which hides the details of how it works.</p> <p>Let's look at modularization first. One way to look at modularization is decomposing complex things into a number of simpler pieces. This is the traditional divide-and-conquer approach to problem solving that we mentioned in the previous video. The other way to view modularization is composing big things out of components. Think of the way that we put together personal computers these days. They are made up of components such as memory boards, hard drives, video boards, keyboards, monitors, and so forth. Each of these components is designed to fit together with the other components to build a bigger piece of equipment. We can think of this analogy to building software out of reusable components.</p>
<p>2</p>	<div data-bbox="350 1285 672 1318" data-label="Section-Header"> <h3>Benefits of Modularization</h3> </div> <div data-bbox="293 1341 633 1463" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Reduces complexity</li> <li>• Results in easier implementation</li> <li>• Facilitates change</li> <li>• Encourages parallel development</li> </ul> </div> <div data-bbox="462 1495 561 1558" data-label="Image"> </div>	<p>When we consider software being assembled out of components, we see that there are several benefits that accrue.</p> <p>First, instead of thinking of our software as being made up of thousands or millions of lines of code, we can view it as being made up of orders of magnitude fewer pieces – components or modules, each of which is made to fit together with other modules, much like Legos fit together. If you remember our discussion of complexity in the first module, you can see that modularization will result in a reduction in software complexity (which was one of the leading causes of the so called software crisis).</p> <p>As we discussed earlier in the course, the cost of developing a piece of software goes up exponentially with the size of the software. Thus, it is easier to build each of the components separately and then integrate them together to build the system, than it is to try to build the entire system as one giant whole.</p>

		<p>If we apply the principles of abstraction and encapsulation when designing our system of modules, then later on, it should be easier to make changes to the software. The system is less complex because there are a small number of well-defined dependencies among the components. This also means that a change to one component may not result in the typical ripple effect we used to see when we visualized our system as being built out of a whole bunch of lines of code.</p> <p>When we are designing our system, if we specify each of the modules with an eye to making sure they will fit together properly when we do the integration, then we should be able to go off and implement each of the modules separately. We could even have different teams of people work on different modules.</p> <p>Think of the way cars are designed and built. Various subsystems, such as engines, transmissions, carburetors, seats, radios, etc., all have standard specifications. In this way, the manufacturer can buy the subsystems from various suppliers and integrate the subsystems together in the assembly line. There may actually be various vendors that can supply the same part. Later on, if one of the parts has a problem, we can just go to the auto parts store and buy a replacement part, possibly made by another manufacturer. It fits because there is a standard specification for the part. This is the way we should be building software – out of pieces with well-defined specifications.</p>
3	<div> <div>What is a Module?</div> <div> <ul style="list-style-type: none"> <li>• Component</li> <li>• Class</li> <li>• Method</li> <li>• Package</li> <li>• Subsystem</li> </ul> </div> <div>  </div> </div>	<p>What, exactly, is a module? It could be something as small as a method or function. It could be as large as a subsystem. In object-oriented systems, the modules are usually classes. At a higher level, a package is a collection of related classes.</p> <p>The unifying thing about these various kinds of modules is that a module specifies a well-defined interface. The modules are connected together by these interfaces. Think of the analogy of a computer being made up of components all with standardized interface specifications.</p>

4	<h3>How Big is a Module?</h3> 	<p>If we view a system as a collection of modules, we have to weigh the cost of specifying and building the modules against the cost of integrating them.</p> <p>As you will remember from our discussion of cost estimation, the amount of effort to build a module is exponentially related to the size of the module. That way it would make sense to build the system out of a whole lot of really small, simple modules. We just add up the costs to build all of the modules. The sum of the costs to build the modules would be less than the cost to build a smaller number of very large modules.</p> <p>On the other hand, there is a cost to integrating the modules together. This would encourage us to build the system out of a small number of modules, each of which would be large. The integration costs also go up exponentially with the number of modules.</p> <p>If we add together the development costs and integration costs, we see that what we have is sort of a bathtub effect. There is a region of lower total cost. We could call this the baby bear effect. Not too many really small modules and not too few really big ones. A moderate number of moderate size modules is just right.</p>
5	<h3>How Do You Decide?</h3> <ul style="list-style-type: none"> <li>• High Cohesion</li> <li>• Low Coupling</li> </ul> 	<p>Of course, we shouldn't determine the ideal module size by counting things like lines of code or anything.</p> <p>There is a "right size" for a module that is determined by its properties. The two main properties that matter are high cohesion and low coupling.</p> <p>If you remember, cohesion refers to how well a module "hangs together." We know that a module holds information and performs behaviors. There is a right way to partition out the information and behaviors to modules and there is a wrong way. The right way is to put things that are related into a module. Adding unrelated information or behavior to a module reduces its cohesion. Removing essential information or behavior from a module also reduces its cohesiveness. Some modules will naturally be large and some will naturally be small. It is the cohesion that matters, not size.</p> <p>The other property that we should be concerned with is the coupling between modules. We should design the modules so that they have standardized interfaces, much like the way we design the components that make up computers or cars, for example. This tends to reduce the complexity of the</p>

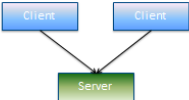

		system by reducing the strength and number of dependencies.
6	<div><div></div><div>Next</div><div><ul style="list-style-type: none"><li>• Abstraction</li></ul></div><div></div><div><p>Jackson Pollock, "No. 5, 1948"</p><p><a href="https://en.wikipedia.org/wiki/File:No._5,_1948.jpg#mediaview:File:No._5,_1948.jpg">https://en.wikipedia.org/wiki/File:No._5,_1948.jpg#mediaview:File:No._5,_1948.jpg</a></p></div></div>	In our next video we will explore the principle of abstraction.

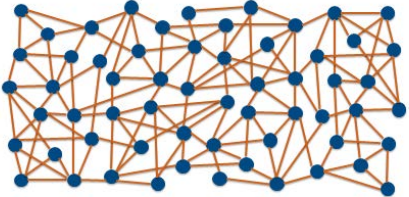
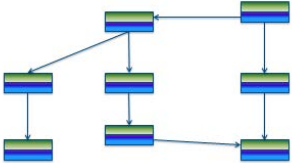
8.1	<div data-bbox="323 201 701 233" data-label="Section-Header"> <h2>Analysis and Design Principles</h2> </div> <div data-bbox="336 279 477 352" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Modularity</li> <li>• <b>Abstraction</b></li> <li>• Encapsulation</li> </ul> </div> <div data-bbox="537 243 626 390" data-label="Image"> </div> <div data-bbox="423 390 740 436" data-label="Caption"> <p>Marcel Duchamp, "Nude Descending a Staircase, (No. 2)" The Philadelphia Museum of Art <a href="http://www.philamuseum.org/collections/permanent/51449.html">http://www.philamuseum.org/collections/permanent/51449.html</a></p> </div>	<p>Abstraction is the second of the three analysis and design principles that we will be discussing here. As you know, abstraction means to highlight the essential properties of something. Of course, there may be many abstractions for something, depending on the properties one wants to highlight.</p> <p>You might wonder if abstract art applies this same principle. Yes, it does. An abstract painting highlights the properties of something that the artist thinks are important. When this picture was shown at the 1913 Armory show in New York, it caused quite a stir. It is an example of Cubism, which is an art form that abstracts out essential properties as geometric shapes.</p>
2	<div data-bbox="440 682 584 711" data-label="Section-Header"> <h2>Abstraction</h2> </div> <div data-bbox="297 739 574 793" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Highlight the essential</li> <li>• Suppress unneeded details</li> </ul> </div> <div data-bbox="448 831 568 947" data-label="Image"> </div>	<p>If abstraction means that we highlight the essential properties of something, then it follows that the inessential details are suppressed, or hidden. This hiding of the details is called encapsulation. We'll talk about that later. Let's focus on how we highlight the essential properties of something.</p> <p>We frequently use models to do this. Consider, for example, the atom. In 1913, the Danish physicist, Niels Bohr, devised the familiar model of the atom as a nucleus with protons and neutrons with electrons spinning around in orbits. It appealed to the notions of Newtonian mechanics that governed things like planetary motions. This model was pretty much determined to be inaccurate based on the later understanding of quantum mechanics, which Bohr himself contributed to. So, if this model is not really the way the atom works, why do we still teach it in Chemistry classes? The reason is that it helps us to understand basic chemical reactions, why certain things combine chemically and why other things do not. We can use it to explain the organization of the periodic table of the elements.</p> <p>So, a model doesn't really need to be transparent at all. It is simply an illusion that helps us to understand something that might otherwise be quite complex.</p>
3	<div data-bbox="339 1644 688 1675" data-label="Section-Header"> <h2>Modules Employ Abstraction</h2> </div> <div data-bbox="297 1701 714 1887" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Specification <ul style="list-style-type: none"> <li>◦ What the module behavior is</li> <li>◦ What information is managed by the module</li> <li>◦ How to invoke the services provided by the module</li> </ul> </li> <li>• Implementation <ul style="list-style-type: none"> <li>◦ How all this works</li> </ul> </li> </ul> </div> <div data-bbox="561 1808 660 1887" data-label="Diagram"> </div>	<p>As we have discussed earlier in this module, analysis and design are modeling exercises. We use models of data and models of behavior to explain the essential properties of software modules.</p> <p>We can think of a module as having two parts: a specification and an implementation. The specification answers the WHAT question: What does the module do? What information does it manage? What is the interface for invoking its behavior?</p>

		<p>The implementation answers the HOW question: How does it work? We will discuss the implementation in the next video, on encapsulation.</p> <p>The important thing to remember is that the specification is an abstraction that makes it easier to understand the module and how to use it.</p>
4	<div> <div>Abstract is a Relative Term</div> <div> <ul style="list-style-type: none"> <li>• writeDocument()</li> <li>• writeChapter()</li> <li>• writePage()</li> <li>• writeParagraph()</li> <li>• writeSentence()</li> <li>• println()</li> <li>• Sequential file I/O services</li> <li>• I/O Control block services</li> <li>• Channel control services</li> <li>• I/O Controller chip commands</li> </ul> </div> <div> <div>Increasing levels of abstraction</div> <div>Decreasing levels of abstraction</div> </div> </div>	<p>I just want to point out that abstraction is a relative term. You can't just say that something is abstract. You can only say that something is more (or less) abstract than something else.</p> <p>If I ask you, "Is println abstract?" Your answer would depend on who you were. As you can see here, we can implement higher and higher levels of abstraction on top of the println. In that case, it's pretty concrete. On the other hand, if you were a systems programmer, say implementing a compiler or operating system, println would be an abstraction. You would have to make it work in terms of operating system details.</p>
5	<div> <div>Next</div> <div> <ul style="list-style-type: none"> <li>• Encapsulation</li> </ul> </div> <div>  </div> </div>	<p>The third principle that we will discuss is encapsulation, also sometimes referred to as information hiding or implementation hiding. The point is that we are separating out the WHAT from the HOW.</p>

<p>9.1</p>	<div data-bbox="321 199 703 233" data-label="Section-Header"> <h2>Analysis and Design Principles</h2> </div> <div data-bbox="332 277 477 352" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Modularity</li> <li>• Abstraction</li> <li>• <b>Encapsulation</b></li> </ul> </div> <div data-bbox="547 264 703 363" data-label="Image"> </div>	<p>The third analysis and design principle that we want to cover is encapsulation. So far we have talked about modularization, where we break the system up into modules, abstraction where we provide a model of what each module does. Now we will discuss how the module hides the details of how it works.</p> <p>Think of something that has an abstract interface and a complex implementation. I have a microwave that has a really simple interface. I can operate by pushing a single button. Now I'm not an electronics engineer, but I know that there's a lot of very complicated equipment inside the microwave that makes it work. To the average person, a microwave is something we use to heat things. Very few people know how microwaves really work. Is that a problem? NO. We don't need to know how a microwave works in order to use it. That's due to a combination of encapsulation (hiding the complexity from the world) and abstraction (giving us a simplified interface so we can use the microwave).</p>
<p>2</p>	<div data-bbox="371 892 649 926" data-label="Section-Header"> <h2>Implementation Hiding</h2> </div> <div data-bbox="293 949 583 1031" data-label="List-Group"> <ul style="list-style-type: none"> <li>• Each module has two views <ul style="list-style-type: none"> <li>◦ Specification: the "what"</li> <li>◦ Implementation: the "how"</li> </ul> </li> </ul> </div> <div data-bbox="461 1054 561 1098" data-label="Image"> </div>	<p>So, just so we're clear: encapsulation is the separation of the specification from the implementation. Abstraction is the way we describe the specification.</p> <p>In encapsulation, then, a module has these two views: the outside view, or specification, that describes WHAT the module does and how to use it, and the inside view, or implementation, that describes HOW the module does what we say it does.</p> <p>One of the original proponents of this idea was David Parnas, who wrote the seminal papers on modularization in the early 1970's. I should also point out additional work that was done on cohesion and coupling by Glenford Myers later in that decade. They're both in Wikipedia.</p> <p>One of the things that Parnas talks about is that the implementation of a module should hide what he refers to as secrets. These secrets don't need to be exposed to users of the module. One of the benefits of this is that if the details of one of these secrets changed, it would only affect the module that encapsulated that secret.</p>



3	<div> <div>Another Quote</div> <div> <div>"Software is a lot like magic."</div> <div>-- Sam Schappelle</div> </div> </div>	<p>This brings us to this quote. This is something that I've often said. Software is a lot like magic. When I was a kid I was interested in performing magic tricks. I had my little magic set and put on shows for the neighbors. Software works like magic because there's the illusion and then there's the secret trick of how it's done.</p> <p>I put together a little sidebar video to explain this.</p>
4	<div> <div>Client-Server Paradigm</div> <ul style="list-style-type: none"> <li>• Client knows the server</li> <li>• Server doesn't know the clients</li> <li>• Client doesn't know how the server works</li> <li>• Server doesn't know how the clients work</li> </ul>  </div>	<p>One of the architectural patterns involves the relationships between clients and servers. This actually applies to any relationship between clients and servers.</p> <p>Clients know who the servers are. They know how to access the services of the servers. This is advertised by the servers and is publicly visible.</p> <p>The servers don't know the clients. The behavior of a server should not in any way depend on who the client is. Servers should be equal opportunity providers.</p> <p>Clients don't know how the servers work. This is encapsulation, pure and simple.</p> <p>Servers don't know how the clients work, either. This sort of goes along with the second point. Servers don't know anything about the clients. This is like encapsulation in both directions.</p>
5	<div> <div>Benefits of Encapsulation</div> <ul style="list-style-type: none"> <li>• Client doesn't need to understand the complexity of the server's implementation</li> <li>• The server can protect its internal state</li> <li>• The implementation of the server may be modified without affecting the clients</li> </ul>  </div>	<p>There are at least three benefits from encapsulation.</p> <p>First, since the complex details of how the server works are encapsulated and the module provides a simple abstract specification, the server appears to the client to be simple. This results in simple clients.</p> <p>The internal secrets in the server can be protected because they are hidden. If the server encapsulates information, the information can't be accessed directly by other modules. They must use the services of the module doing the encapsulating.</p> <p>Since the module encapsulates a secret implementation, any modifications to that secret implementation are not visible to the outside. As long as the module provides the behavior as advertised in the specification, the implementation can be changed without affecting any of the clients.</p>

6	<div> <div>Correctness</div> <div> <ul style="list-style-type: none"> <li>Does the implementation do what the specification says it will?</li> </ul> </div> <div> <div>Specification</div> <div>Implementation</div> </div> </div>	<p>One more thing. We know we have a correct module if the implementation does what the specification says it will. This can be the basis for inspections or testing.</p>
7	<div> <div>From This</div>  </div>	<p>Remember our discussion of the software crisis in the first module of this class? We said that the main cause of the problems we were facing was the overwhelming complexity of the software, and that the complexity continued to grow as time went on.</p> <p>We said that software was complex because any line of code could theoretically have a dependency on any other line of code.</p>
8	<div> <div>To This</div>  </div>	<p>There are two ways we can reduce this complexity. One is to break the software lines of code into modules, thus reducing the number of nodes of our dependency graph. The other is to minimize the number of possible dependencies between modules.</p> <p>Modularization, abstraction and encapsulation taken together achieve this effect. They provide our best hope of controlling the software crisis.</p>
9	<div> <div>End of Module</div> <div> <ul style="list-style-type: none"> <li>You should now be able to: <ul style="list-style-type: none"> <li>Explain the importance of each of the three bases of analysis and design: Structure, Information, and Behavior.</li> <li>Utilize structural, informational, and behavioral models in analysis and design</li> <li>Discuss the meaning and importance of analysis and design principles such as abstraction, modularity, and encapsulation.</li> </ul> </li> </ul> </div> </div>	<p>This is the end of the module.</p> <p>You should be able to participate in the discussions for this module, and to apply what you learned to the case study. Specifically, you should be able to do these things:</p>