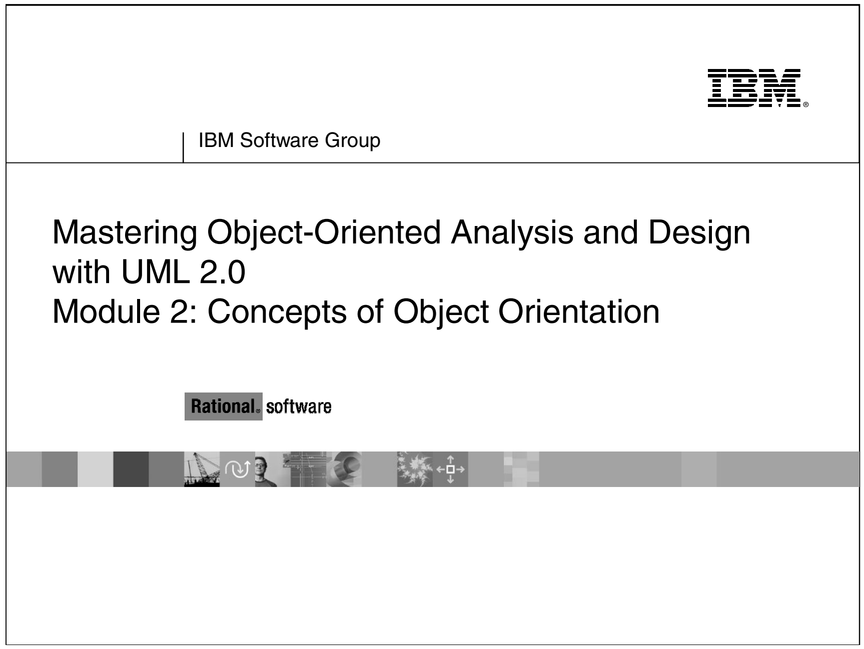
Instructor Notes:



Instructor Notes:

This is a review module of the Concepts of Object Orientation. The students should have already seen many of these slides in the Essentials of Visual Modeling course. You should be able to review these slides quickly or if you think the students already have grasped this content, feel free to skip it.

Objectives: Concepts of Object Orientation

- Explain the basic principles of object orientation
- Define the basic concepts and terms of object orientation and the associated UML notation

2

IBM

Instructor Notes:

Sell the students on the value of visual modeling.

Clarify that you are discussing formal modeling, not modeling written on a white board or on the back of a napkin at lunch.

Review: Why Model?

- Modeling achieves four aims:
  - Helps you to visualize a system as you want it to be.
  - Permits you to specify the structure or behavior of a system.
  - Gives you a template that guides you in constructing a system.
  - Documents the decisions you have made.
- You build models of complex systems because you cannot comprehend such a system in its entirety.
- You build models to better understand the system you are developing.

3 IBM

According to Booch in *The Unified Modeling Language User Guide*, modeling achieves four aims:

1. Models help you to **visualize** a system, as you want it to be. A model helps the software team communicate the vision for the system being developed. It is difficult for a software team to have a unified vision of a system that is described only in specification and requirement documents. Models bring about understanding of the system.

2. Models permit you to **specify** the structure of behavior of a system. A model allows how to document system behavior and structure before coding the system.

3. Models give a template that guide you in **constructing** a system. A model is an invaluable tool during construction. It serves as a road map for a developer. Have you experienced a situation where a developer coded incorrect behavior because he or she was confused over the wording in a requirements document? Modeling helps alleviate that situation.

4. Models **document** the decisions you've made. Models are valuable tools in the long term because they give "hard" information on design decisions. You don't need to rely on someone's memory.

Instructor Notes:

Demonstrate that there are guiding principles for visual modeling. It is not chaotic activity.

The four modeling principles are described in detail on the next four slides.

See the "UML User Guide" for a more detailed discussion on these four principles.

Four Principles of Modeling

- The model you create influences how the problem is attacked.
- Every model may be expressed at different levels of precision.
- The best models are connected to reality.
- No single model is sufficient.

4

Modeling has a rich history in all the engineering disciplines.

The four basic principles of modeling are derived from this history.

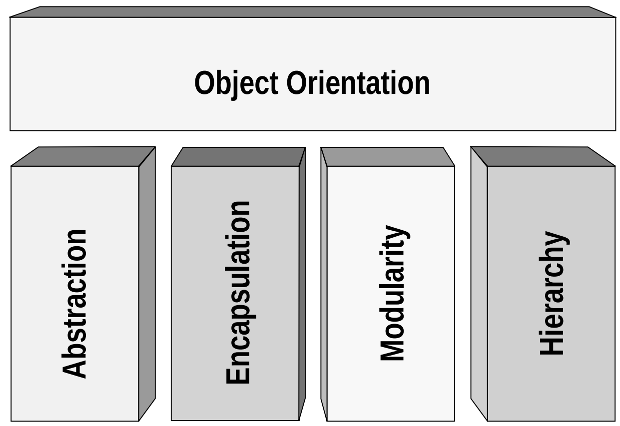
1. The models you create profoundly influence how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every non-trivial system is best approached through a small set of nearly independent models.

Instructor Notes:

Be sure the students understand objects before you begin this next section.

You've introduced objects first to help students better apply each of these principles.

Review: Basic Principles of Object Orientation

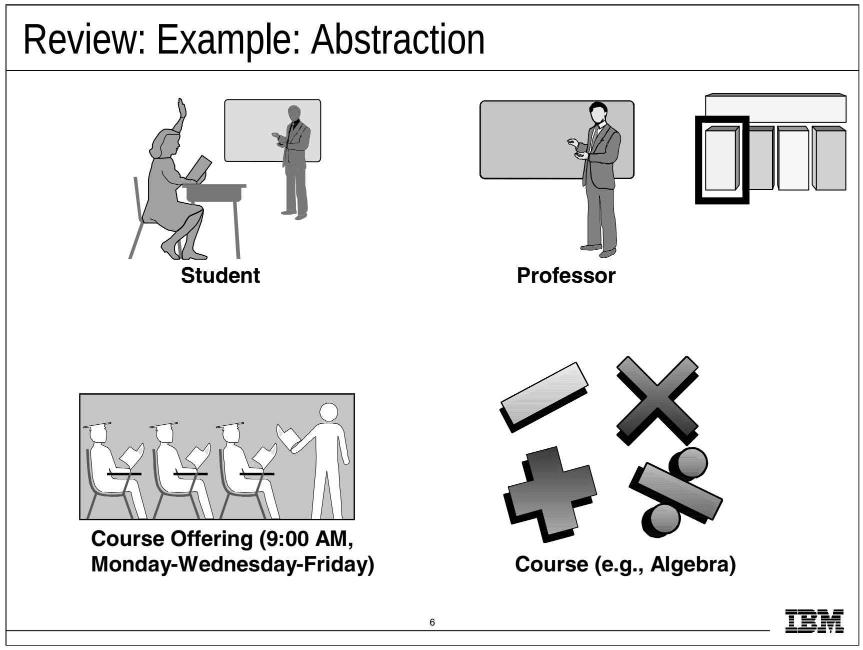


5

There are four basic principles of object orientation. They are:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Instructor Notes:



The following are examples of abstraction.

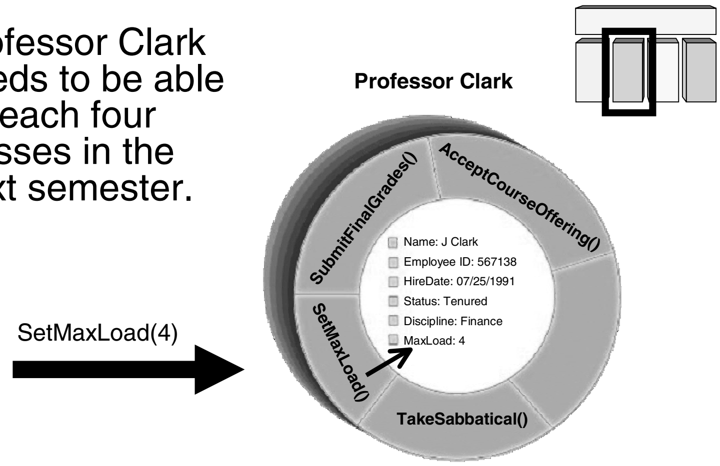
- A student is a person enrolled in classes at the university.
- A professor is a person teaching classes at the university.
- A course is a class offered by the university.
- A course offering is a specific offering for a course, including days of the week and times.

Instructor Notes:

- Note that encapsulation can also be illustrated using interfaces.
- Point out that the requesting object does not need to know the structure of the Professor object to request a state change.
- The object that owns the attributes is the only one allowed to change its own attributes.

Review: Encapsulation Illustrated

- Professor Clark needs to be able to teach four classes in the next semester.



7

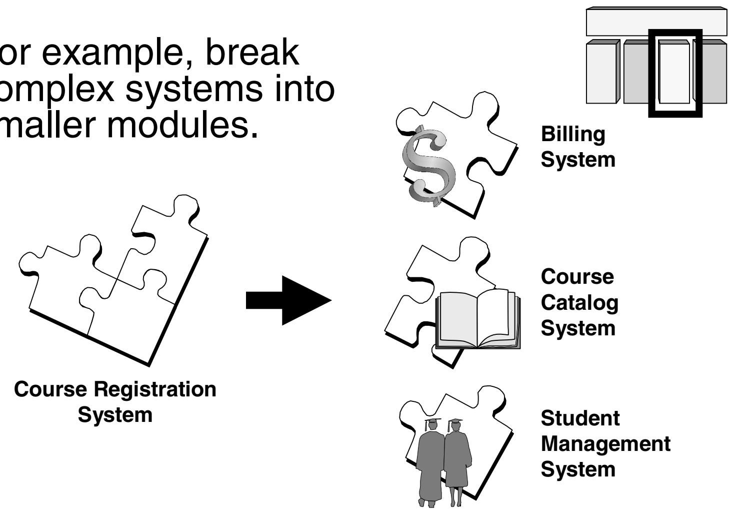
- The key to encapsulation is an object's **message interface**. The object interface ensures that all communication with the object takes place through a set of predefined operations. Data inside the object is only accessible by the object's operations. No other object can reach inside of the object and change its attribute values.
- For example, Professor Clark needs to have her maximum course load increased from three classes to four classes per semester. Another object will make a request to Professor Clark to set the maximum course load to four.The attribute, MaxLoad, is then changed by the SetMaxLoad() operation.
- Encapsulation is beneficial in this example because the requesting object does not need to know how to change the maximum course load. In the future, the number of variables that are used to define the maximum course load may be increased, but that does not affect the requesting object. The requesting object depends on the operation interface for the Professor Clark object.

Instructor Notes:

A car is an example of modularity. It is made of up of a body, chassis, engine, wheels, and so on.

Review: Example: Modularity

- For example, break complex systems into smaller modules.



8

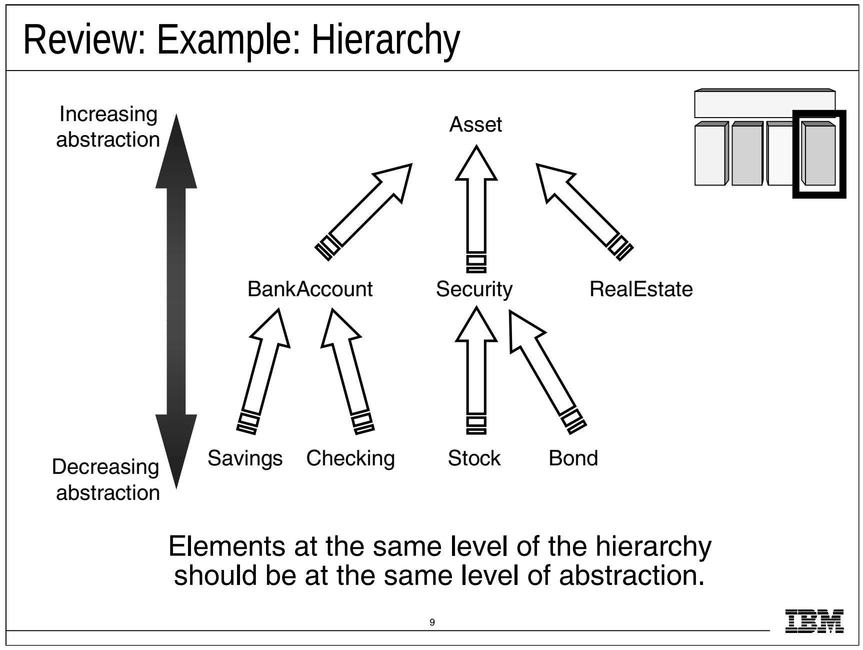
Often, the system under development is too complex to understand. To further understanding, the system is broken into smaller blocks that are each maintained independently. Breaking down a system in this way is called **modularity**. It is critical for understanding a complex system.

For example, the system under development is a Course Registration system. The system itself is too large and abstract to allow an understanding of the details. Therefore, the development team broke this system into three modular systems, each independent of the others.

- The Billing System
- Course Catalog System
- Student Management System

Instructor Notes:

Hierarchy is a taxonomic organization. The use of hierarchy makes it easy to recognize similarities and differences.



Hierarchy can be defined as:

Any ranking or ordering of abstractions into a tree-like structure. Kinds: Aggregation hierarchy, class hierarchy, containment hierarchy, inheritance hierarchy, partition hierarchy, specialization hierarchy, type hierarchy. (*Dictionary of Object Technology*, Firesmith, Eykholt, 1995)

- Hierarchy organizes items in a particular order or rank (for example, complexity and responsibility). This organization is dependent on perspective. Using a hierarchy to describe differences or variations of a particular concept provides for more descriptive and cohesive abstractions and a better allocation of responsibility.
- In any one system, there may be multiple abstraction hierarchies (for example, a financial application may have different types of customers and accounts).
- Hierarchy is not an organizational chart or a functional decomposition.
- Hierarchy is a taxonomic organization. The use of hierarchy makes it easy to recognize similarities and differences. For example, botany organizes plants into families. Chemistry organizes elements in a periodic table.

Instructor Notes:

Emphasize that state, identity and behavior are the key characteristics of an object.

Take a moment to explain the graphic on this slide.

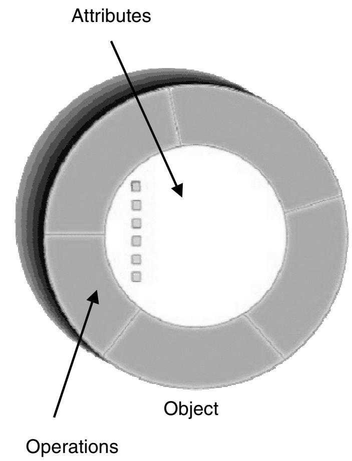
Attributes are documented on the inside of the doughnut.

Operations are documented on the borders, which become clearer to the student as you discuss topics like encapsulation.

Note that the doughnut is not part of the UML notation. UML notation is discussed later.

Review: What Is an Object?

- An object is an entity with a well-defined boundary and *identity* that encapsulates *state* and *behavior.*
  - State is represented by attributes and relationships.
  - Behavior is represented by operations, methods, and state machines.



10

An **object** is an entity that has a well-defined boundary. That is, the purpose of the object should be clear.

An object has two key components: attributes and operations.

Attributes and relationships represent an object's state. Operations represent the behavior of the object.

Object behavior and state are discussed in the next few slides.

Instructor Notes:

Review the definition of the terms attribute and operation for the students. Relationships will be covered shortly.

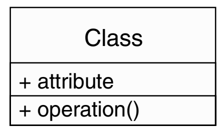
An **attribute** is a named property of a class that describes the range of values that instances of the property may hold.

In the UML, methods and **operations** are NOT synonymous and have distinct definitions.

An operation is simply the advertisement of a service that is offered by a class. A method is the actual code that realizes that operation.

Review: What Is a Class?

- A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and semantics.
  - An object is an instance of a class.
- A class is an abstraction in that it
  - Emphasizes relevant characteristics.
  - Suppresses other characteristics.



11

A **class** can be defined as:

A description of a set of objects that share the same attributes, operations, relationships, and semantics. (*The Unified Modeling Language User Guide*, Booch, 1999)

- There are many objects identified for any domain.
- Recognizing the commonalties among the objects and defining classes helps us deal with the potential complexity.
- The OO principle of abstraction helps us deal with complexity.

An **Attribute** can be defined as:

A named property of a class that describes the range of values that instances of the property may hold. (*The Unified Modeling Language User Guide*, Booch, 1999.)

•A class may have any number of attributes or no attributes at all. At any time, an object of a class has specific values for every one of its class's attributes.

An **Operation** can be defined as:

A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

•The operations in a class describe what the class can do.

Instructor Notes:

Note that the class graphics used throughout this course were created in Rational Rose.

Make sure the class understands that the visibility symbols next to both attributes and operations are Rose-specific and not UML notation.

Tell the students that the UML represents public visibility with a plus (+) symbol and private visibility with a minus (-) symbol. We do not discuss protected visibility.

Review: Representing Classes in the UML

- A class is represented using a rectangle with compartments.

Professor
- name - employeeID : UniqueID - hireDate - status - discipline - maxLoad
+ submitFinalGrade() + acceptCourseOffering() + setMaxLoad() + takeSabbatical() + teachClass()

Professor J Clark

12

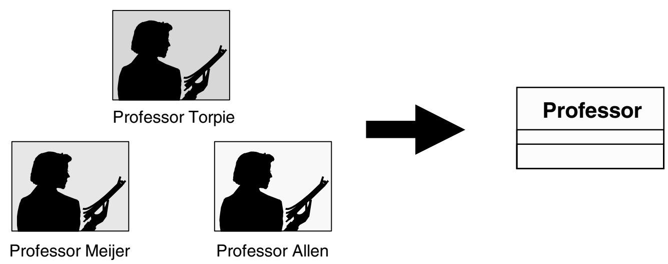
- The UML notation for a class permits you to see an abstraction apart from any specific programming language, which lets you emphasize the most important parts about an abstraction — its name, attributes, and operations.
- Graphically, a class is represented by a rectangle.

Instructor Notes:

A class is to an object what a cookie cutter is to a cookie.

Review: The Relationship Between Classes and Objects

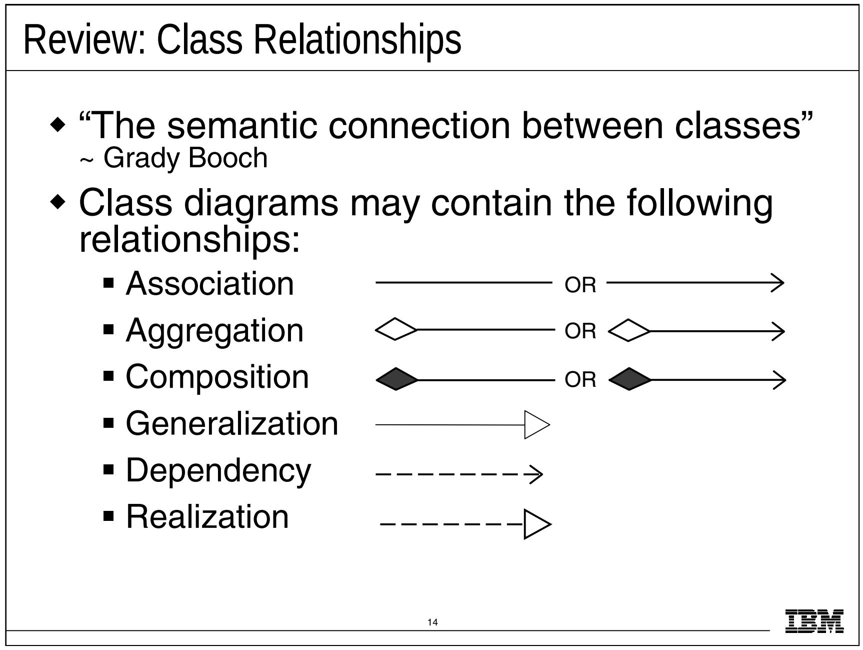
- A class is an abstract definition of an object.
  - It defines the structure and behavior of each object in the class.
  - It serves as a template for creating objects.
- Classes are not collections of objects.



13

- A class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.
- An object is defined by a class. A class defines a template for the structure and behavior of all its objects. The objects created from a class are also called the **instances** of the class.
- The class is the static description; the object is a run-time instance of that class.
- Since we model from real-world objects, software objects are based on the real-world objects, but they exist only in the context of the system.
- Starting with real-world objects, abstract out what you do not care about. Then, take these abstractions and categorize, or *classify* them, based on what you *do* care about. Classes in the model are the result of this classification process.
- These classes are then used as templates within an executing software system to create software objects. These software objects represent the real-world objects we originally started with.
- Some classes/objects may be defined that do not represent real-world objects. They are there to support the design and are "software only."

Instructor Notes:



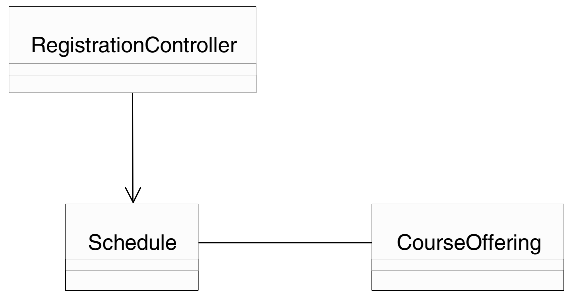
The next several slides will explain and define each of these relationships.

Instructor Notes:

Double-headed arrows are also valid for bi-directional associations.

What Is Navigability?

- Indicates that it is possible to navigate from a associating class to the target class using the association



15

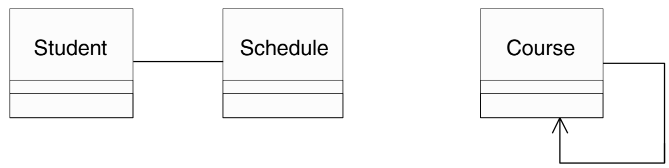
- The **navigability** property on a role indicates that it is possible to navigate from a associating class to the target class using the association. This may be implemented in a number of ways: by direct object references, associative arrays, hash-tables, or any other implementation technique that allows one object to reference another.
- Navigability is indicated by an open arrow placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true (associations are bi-directional by default).
- In the course registration example, the association between the Schedule and the Course Offering is navigable in both directions. That is, a Schedule must know the Course Offering assigned to the Schedule, and the Course Offering must know the Schedules it has been placed in.
- When no arrowheads are shown, the association is assumed to be navigable in both directions.
- In the case of the association between Schedule and Registration Controller, the Registration Controller must know its Schedules, but the Schedules have no knowledge of the Registration Controllers (or other classes). As a result, the navigability property of the Registration Controller end of the association is turned off.

Instructor Notes:

Draw these examples on the board as objects to help the students understand the concept.

Review: What Is an Association?

- The semantic relationship between two or more classifiers that specifies connections among their instances
  - A structural relationship, specifying that objects of one thing are connected to objects of another



16

An **association** can be defined as:

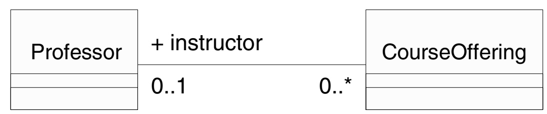
The semantic relationship between two or more classifiers that specifies connections among their instances. In other words, an association is a structural relationship that specifies that objects (instances of classes) are connected to other objects.

- The way that we show relationships between classes is through the use of associations. Associations are represented on class diagrams by a line connecting the associating classes. Data may flow in either direction or in both directions across a link.
- Most associations are simple. That is, they exist between exactly two classes. They are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible.
- Sometimes, a class has an association to itself. This does not always mean that an instance of that class has an association to itself. More often, it means that one instance of the class has associations to other instances of the same class.
- This example shows that a student object is related to a schedule object. The course class demonstrates how a course object can be related to another course object.

Instructor Notes:

Review: What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course Offering, there may be either one or zero Professor as the instructor.



17

Multiplicity can be defined as:

The number of instances of one class that relate to one instance of another class.

- For each role, you can specify the multiplicity of its class and how many objects of the class can be associated with one object of the other class.
- Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges.
- It is important to remember that multiplicity is referring to instances of classes (objects) and their relationships. In this example, a Course Offering object can have either zero or one Professor object related to it. Conversely, a Professor object can have zero or more Course Offering objects related to it.
- Multiplicity must be defined on both ends of the association.

Instructor Notes:

The use of “N” instead of “*” is from Booch, not UML. For example, the use of “0..N” and “N” is not UML.

The multiplicity specified for a relationship is for all instances of that relationship, not simply for a particular use-case realization or for a particular point in time.

Review: Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional scalar role)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

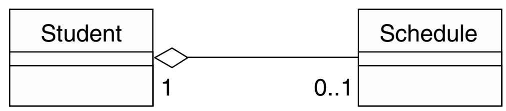
18

- Multiplicity is indicated by a text expression on the role.
- The expression is a comma-separated list of integer ranges.
- A range is indicated by an integer (the lower value), two dots, followed by another integer (the upper value).
- A single integer is a valid range, and the symbol “*” indicates "many.” That is, an asterisk “*” indicates an unlimited number of objects.
- The symbol “*” by itself is equivalent to “0..*” That is, it represents any number, including none. This is the default value.
- An optional scalar role has the multiplicity 0..1.

Instructor Notes:

Review: What Is Aggregation?

- An aggregation is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - An aggregation is an “Is a part-of” relationship.
- Multiplicity is represented like other associations.



19

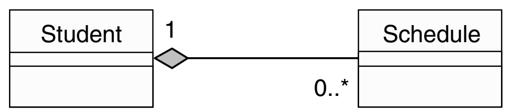
An **aggregation** can be defined as:

A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.

- Aggregation is used to model relationships between model elements. There are many examples of aggregation: a library contains books, departments are made up of employees, a computer is composed of a number of devices. To model an aggregation, the aggregate (department) has an aggregation association to the its constituent parts (employee).
- A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.
- An aggregation relationship that has a multiplicity greater than one for the aggregate is called **shared**. Destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph or a tree with many roots. Shared aggregations are used where there is a strong relationship between two classes. Therefore, the same instance can participate in two different aggregations.

What Is Composition?

- A composition is a stronger form of association in which the composite has sole responsibility for managing its parts – such as their allocation and deallocation.
- It is shown by a diamond filled adornment on the opposite end.



20

The relationship from Student to Schedule is modeled as a composition because if you got rid of the Student, you would get rid of any Schedules for that Student. Here are some general rules for when to use composition.

Use composition when:

- Properties need independent identities
- Multiple classes have the same properties
- Properties have a complex structure and properties of their own
- Properties have complex behavior of their own
- Properties have relationships of their own

Otherwise use attributes

Instructor Notes:

Composition and attributes are semantically equivalent. It just depends on what you need to model to make your system understandable. Classes are capable of expressing structure, relationships, and more complex behavior. If you need any of these, use a class. The next slide shows an example of composition within a structured diagram.

Instructor Notes:

The concept of capsules in Rose RealTime was a leading influencer for the UML2 structured class.

One of the most important modeling features added to UML2.

What is a Structured Class?

- A structured class contains parts or roles that form its structure and realize its behavior
  - Describes the internal implementation structure
- The parts themselves may also be structured classes
  - Allows hierarchical structure to permit a clear expression of multilevel models.
- A connector is used to represent an association in a particular context
  - Represents communications paths among parts

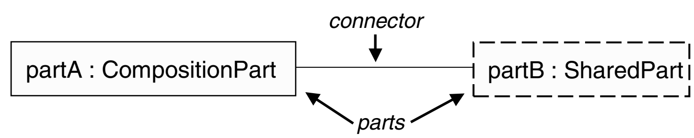
21

A **role** is a constituent element of a structured class that represents the appearance of an instance (or, possibly, set of instances) within the context defined by the structured class.

Instructor Notes:

Structured Class Notation

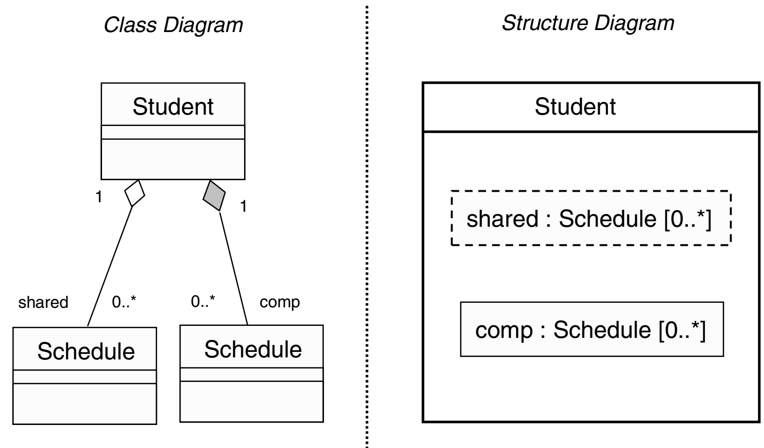
- A part or role is shown by using the symbol for a class (a rectangle) with the syntax:
  - rolename : Typename [multiplicity]
- All three may be omitted.
  - If multiplicity is omitted, it defaults to one.
- A reference to an external object (one not owned by the enclosing object) is shown by a dashed rectangle.



22

Instructor Notes:

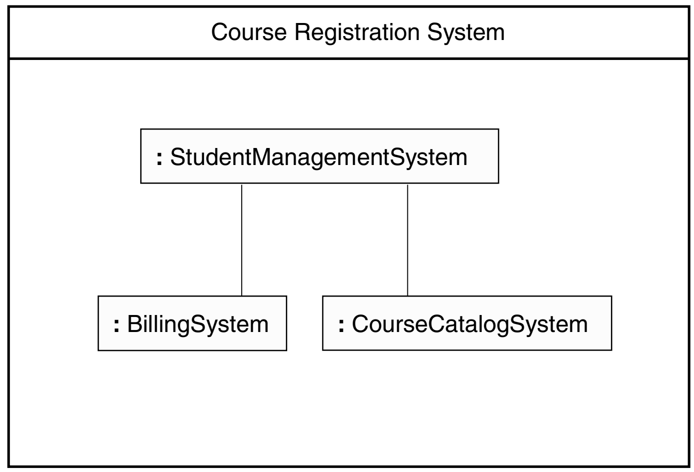
Class Diagram versus Structure Diagram



23

Instructor Notes:

Example: Structure Diagram



As the system is further decomposed, each of the parts may be a structured class which contains parts themselves. This is a very effective method to visualize the system architecture.

Instructor Notes:

Generalization relationships are also permitted between packages. However, packages do not have semantics. Therefore, generalization between packages is not common.

According to Grady Booch:

The terms "inheritance" and "generalization" are, practically speaking, interchangeable. The UML standardized on calling the relationship "generalization" as not to confuse people with language-specific meanings of inheritance.

To confuse matters further, some call this an "is-a" or a "kind of" relationship (especially those into conceptual modeling in the cognitive sciences).

So, for most users, it's fair to use either term. For power users, people who care about things like the UML metamodel and specifying formal semantics of the same, the relationship is called "generalization," and applying such a relationship between, for example, two classes, results in the subclass inheriting the structure and operations of the superclass. (Inheritance is the mechanism.)

Review: What Is Generalization?

- A relationship among classes where one class shares the structure and/or behavior of one or more classes
- Defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses
  - Single inheritance
  - Multiple inheritance
- Is an "is a kind of" relationship

25

IBM

Generalization can be defined as:

A specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). (*The Unified Modeling Language User Guide*, Booch, 1999)

- The subclass may be used where the superclass is used, but not vice versa.
- The child inherits from the parent.
- Generalization is transitive. You can always test your generalization by applying the "is a kind of" rule. You should always be able to say that your specialized class "is a kind of" the parent class.
- The terms "generalization" and "inheritance" are generally interchangeable. If you need to distinguish, generalization is the name of the relationship, while inheritance is the mechanism that the generalization relationship represents/models.

Inheritance can be defined as:

The mechanism by which more-specific elements incorporate the structure and behavior of more-general elements. (*The Unified Modeling Language User Guide*, Booch, 1999)

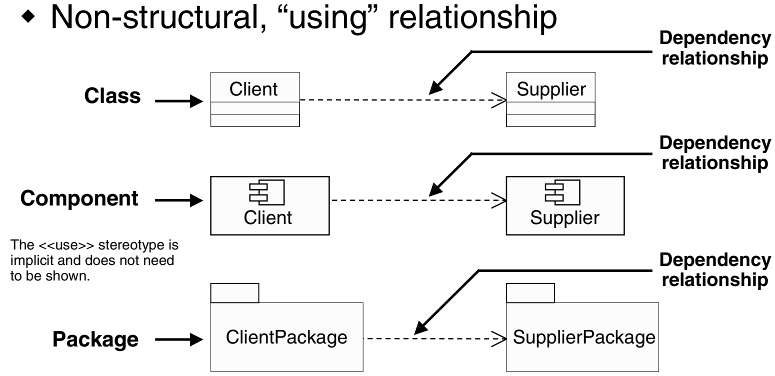
- Single inheritance: The subclass inherits from only one superclass (has only one parent).
- Multiple inheritance: The subclass inherits from more than one superclass (has multiple parents).

Instructor Notes:

Structural versus non-structural relationships will be discussed in depth in the Class Design module.

Relationships: Dependency

- A relationship between two model elements where a change in one may cause a change in the other
- Non-structural, “using” relationship



26

A **dependency** relationship is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier.

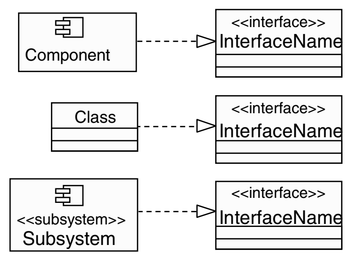
A dependency relationship denotes a semantic relationship between model elements, where a change in the supplier may cause a change in the client.

Instructor Notes:

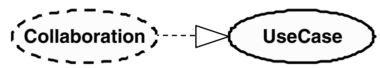
We discussed subsystems earlier in this module. We will look at interfaces and the realization relationship in more detail in the Identify Design Elements module.

Relationships: Realization

- One classifier serves as the contract that the other classifier agrees to carry out, found between:
  - Interfaces and the classifiers that realize them



  - Use cases and the collaborations that realize them



27

Realization is a semantic relationship between two classifiers. One classifier serves as the contract that the other classifier agrees to carry out.

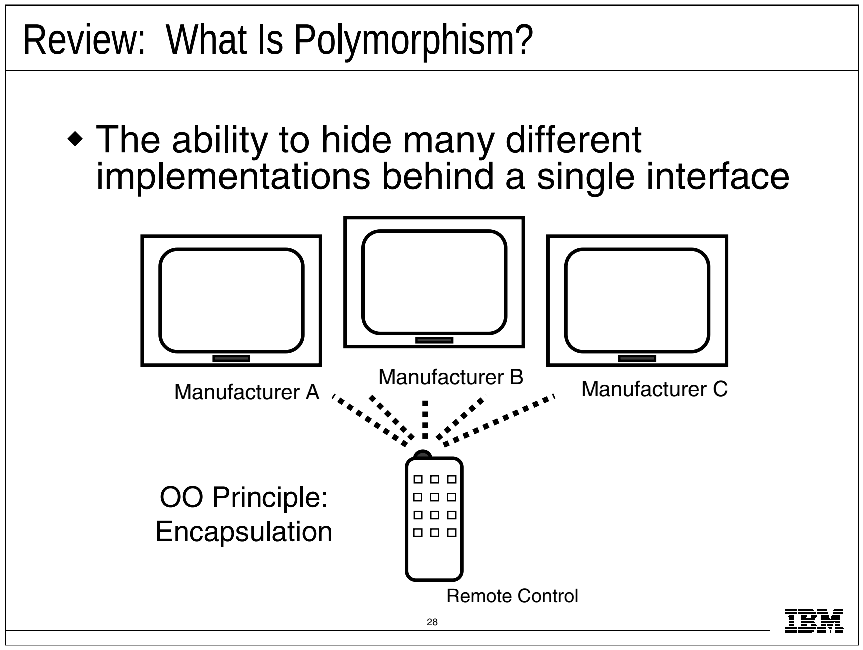
The realizes relationship is a combination of a dependency and a generalization. It is not true generalization, as only the “contract” (that is to say, operation signature) is “inherited.” This “mix” is represented in its UML form, which is a combination of dependency and generalization.

The realizes relationship may be modeled as a dashed line with a hollow arrowhead pointing at the contract classifier (canonical form), or when combined with an interface, as a “ball” (elided form).

Again, from *The Random House Collegiate Dictionary*:

- Elide: to pass over; omit; ignore.
- Canonical: authorized; recognized; accepted.

Instructor Notes:



The Greek term *polymorphos* means “having many forms.” There may be one or many implementations of a given interface. Every implementation of an interface must fulfill the requirements of that interface. In some cases, the implementation can perform more than the basic interface requirements.

For example, the same remote can be used to control any type of television (implementation) that supports the specific interface that the remote was designed to be used with.

Instructor Notes:

Interfaces are not abstract classes, as abstract classes allow you to provide default behavior for some/all of their methods. Interfaces provide no default behavior.

What Is an Interface?

- A declaration of a coherent set of public features and obligations.
  - A contract between providers and consumers of services. Examples of interfaces are:
    - Provided interface - The interfaces that the element exposes to its environment.
    - Required interface - The interfaces that the element requires from other elements in its environment in order to be able to offer its full set of provided functionality.

29

IBM

Interfaces are the key to the "plug-and-play" ability of an architecture: Any classifiers (for example, classes, subsystems, components) that realize the same interfaces may be substituted for one another in the system, thereby supporting the changing of implementations without affecting clients.

Interfaces formalize polymorphism.They allow us to define polymorphism in a declarative way, unrelated to implementation. Two elements are polymorphic with respect to a set of behaviors if they realize the same interfaces. In other words, if two objects use the same behaviors to get different, but similar results, they are considered to be polymorphic. A cube and a pyramid can both be drawn, moved, scaled, and rotated, but they look very different.

You have probably heard that polymorphism is one of the big benefits of object orientation, but without interfaces there is no way to enforce it, verify it, or even express it except in informal or language-specific ways. Formalization of interfaces strips away the mystery of polymorphism and gives us a good way to describe, in precise terms, what polymorphism is all about. Interfaces are testable, verifiable, and precise.

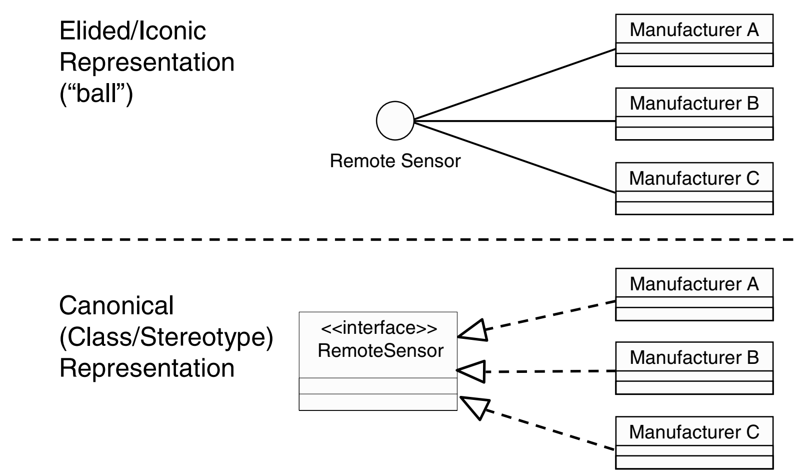
Instructor Notes:

According to the *UML User Guide,* there are two ways to represent a realizes relationship with an interface. The elided form ("ball") is useful when you want to expose the seams of the system. However, there is a limitation in that you can't visualize the operations on the interface.

The second way, canonical, allows you to visualize the operations on the interface.

The implementation dependency from a classifier to an interface is shown by representing the interface with a circle or *ball*, labeled with the name of the interface and attached by a solid line to the classifier that implements this interface.

Example: A Provided Interface



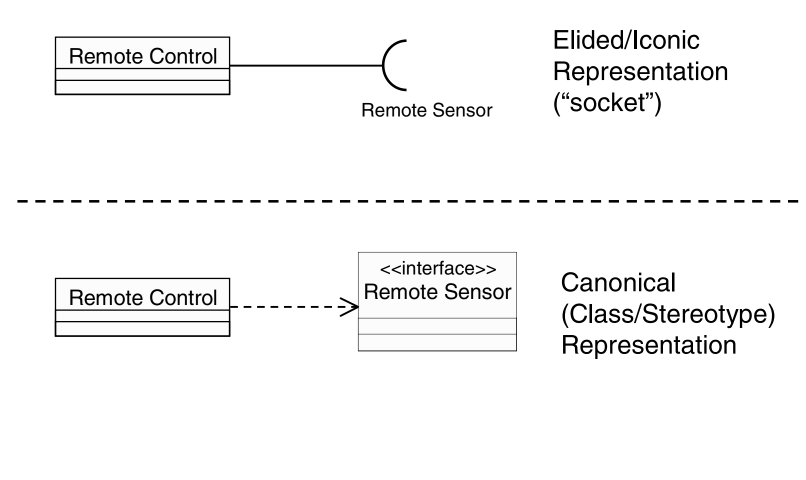
30

The ball notation is best used when you only need to denote the existence of an interface. If you need to see the details of the interface (for example, the operations), then the class/stereotype representation is more appropriate.

From *The Random House Collegiate Dictionary*:

- Elide: to pass over; omit; ignore.
- Canonical: authorized; recognized; accepted.

Example: A Required Interface



Instructor Notes:

According to the *UML User Guide,* there are two ways to represent a realizes relationship with an interface. The elided form ("ball") is useful when you want to expose the seams of the system. However, there is a limitation in that you can't visualize the operations on the interface.

The second way, canonical, allows you to visualize the operations on the interface.

The usage dependency from a classifier to an interface is shown by representing the interface with a half-circle or *socket*, labeled with the name of the interface and attached by a solid line to the classifier that implements this interface.

The ball notation is best used when you only need to denote the existence of an interface. If you need to see the details of the interface (for example, the operations), then the class/stereotype representation is more appropriate.

From *The Random House Collegiate Dictionary*:

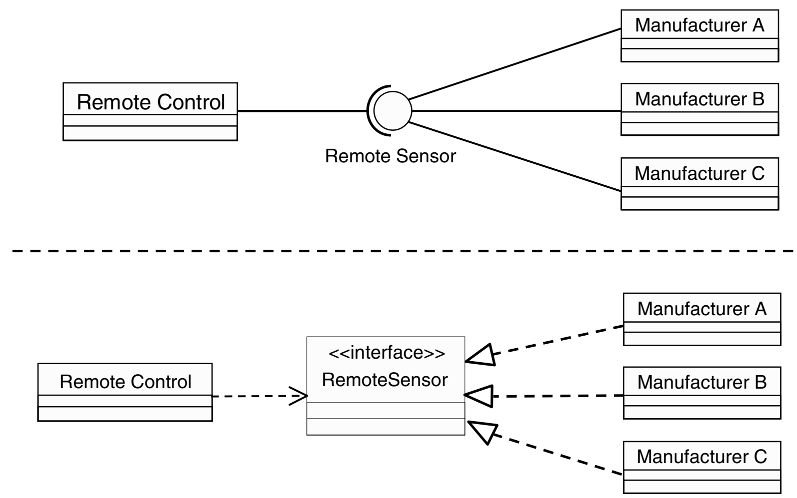
- Elide: to pass over; omit; ignore.
- Canonical: authorized; recognized; accepted.

Instructor Notes:

According to the *UML User Guide,* there are two ways to represent a realizes relationship with an interface. The elided form (“ball”) is useful when you want to expose the seams of the system. However, there is a limitation in that you can’t visualize the operations on the interface.

The second way, canonical, allows you to visualize the operations on the interface.

Example: Connecting Interfaces



The ball notation is best used when you only need to denote the existence of an interface. If you need to see the details of the interface (for example, the operations), then the class/stereotype representation is more appropriate.

From *The Random House Collegiate Dictionary*:

- Elide: to pass over; omit; ignore.
- Canonical: authorized; recognized; accepted.

Instructor Notes:

What is a Port?

- A port is a structural feature that encapsulates the interaction between the contents of a class and its environment.
  - Port behavior is specified by its provided and required interfaces
- Permits the internal structure to be modified without affecting external clients
  - External clients have no visibility to internals
- A class may have a number of ports
  - Each port has a set of provided and required interfaces

33 IBM

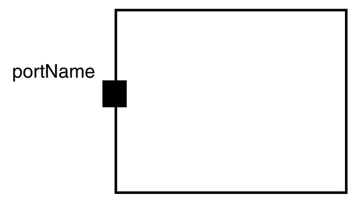
Since the port is a structural element, it's created and destroyed along with its structured class.

Another class connected to a port may request the provided services from the owner of the port but must also be prepared to supply the required services to the owner.

Instructor Notes:

Port Notation

- A port is shown as a small square with the name placed nearby.



- Ports may be public, protected or private

Instructor Notes:

Port Types

- Ports can have different implementation types
  - Service Port - Is only used for the internal implementation of the class
  - Behavior Port - Requests on the port are implemented directly by the class
  - Relay Port – Requests on the port are transmitted to internal parts for implementation

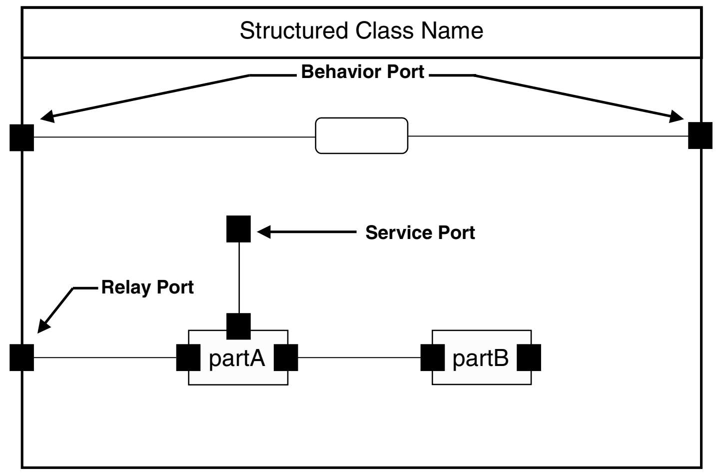
35

The use of **service ports** are rare because the main purpose of ports is to encapsulate communication with the environment. These ports are located inside the class boundary.

Behavior ports are shown by a line from the port to a small state symbol (a rectangle with rounded corners). This is meant to suggest a state machine, although other forms of behavior implementation are also permitted.

Instructor Notes:

Example: Structure Diagram with Ports



36

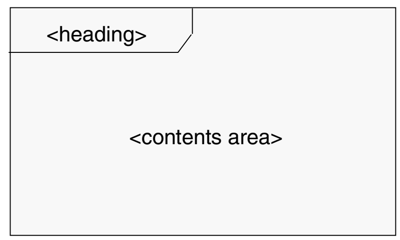
Instructor Notes:

The UML 2 specification does not have consistent usage of <kind>, for example, the interactions chapter uses **sd** for all kinds of interaction diagrams (including timing), which is clearly erroneous.

The frame is primarily used in cases where the diagrammed element has graphical border elements, like ports for classes and components (in connection with composite structures), entry/exit points on state machines, and gates/fragments on sequence diagrams.

Review: Diagram Depiction

- Each diagram has a frame, a heading compartment in the upper left corner, and a contents area.
  - If the frame provides no additional value, it may be omitted and the border of the diagram area provided by the tool will be the implied frame.



37

A heading compartment is a string contained in a name tag (a rectangle with cutoff corner) in the upper leftmost corner with the following syntax:

[<kind>]<name>[<parameters>]

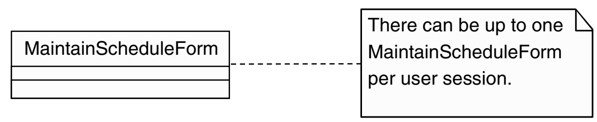
This <kind> can be:

- activity - activity diagram
- package - class diagram, package diagram
- communication - communication diagram
- component - component diagram
- class - composite structure diagram
- deployment - deployment diagram
- intover - interaction overview diagram
- object - object diagram
- state machine - state machine diagram
- sd - sequence diagram
- timing -timing diagram
- use case - use case diagram

Instructor Notes:

What Are Notes?

- A comment that is added to include more information on the diagram
- May be added to any UML element
- A “dog eared” rectangle
- May be anchored to an element with a dashed line



38

Instructor Notes:

1. The four basic principles of object orientation include: abstraction, encapsulation, modularity and hierarchy.

An **abstraction** is any model that includes the most important aspects of something while suppressing or ignoring less important details. **Encapsulation** is the physical localization of features that hides their implementation behind a public interface. **Modularity** is the logical and physical decomposition of things into small, simple groupings, which increase the achievements of software-engineering goals. **Hierarchy** is any ranking or ordering of abstractions into a tree-like structure.

2. A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and semantics.

An object is an instance of a class. It is defined by a class.

3. A class relationship is the semantic connection between classes. Examples of relationships include: association, aggregation, composition, generalization, dependency, and realization.

4. Polymorphism is the ability to hide many different implementations behind a single interface. An interface is a collection of operations that are used to specify a service of a class or a component.

5. An interface is a declaration of a coherent set of public features and obligations.

Review: Concepts of Object Orientation

- What are the four basic principles of object orientation? Provide a brief description of each.
- What is an object and what is a class? What is the difference between the two?
- What is a class relationship? Give some examples.
- What is polymorphism?
- What is an interface?

39

Instructor Notes:

40