msdn® training

Module 4: Object-Oriented Design for Visual Basic .NET

Contents

This course is based on the prerelease version (Beta 2) of Microsoft® Visual Studio® .NET Enterprise Edition. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 2 version of Visual Studio .NET Enterprise Edition.

Microsoft®

Instructor Notes

Presentation:
75 Minutes

Lab:
45 Minutes

This module explains the basic concepts that students need to understand to take advantage of the object-oriented enhancements to Microsoft® Visual Basic® .NET. It describes how to create use cases and class diagrams to model the system. It focuses on the areas that Visual Basic developers may be familiar with but need to understand fully so they can use encapsulation, inheritance, interfaces, and polymorphism.

In the lab, students will use Microsoft Visio® to create diagrams for classes, attributes, operations, and relationships based on given use case descriptions.

After completing this module, students will be able to:

- Describe the basics of object-oriented design.
- Explain the concepts of encapsulation, inheritance, interfaces, and polymorphism.
- Create classes based on use cases.
- Model classes for use in Visual Basic .NET by using Visio.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2373A_04.ppt
- Module 4, "Object-Oriented Design for Visual Basic .NET"
- Lab 4.1, " Creating Class Diagrams from Use Cases"

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Read the instructor notes and the margin notes for the module.
- Practice the demonstrations.
- Complete the practice.
- Complete the lab.

Practice: Deriving Classes from Use Cases

This section provides suggested solutions for the student practice detailed in the module notes. These are only recommended solutions for the classes, attributes, and operations, and may be slightly different from the solutions created by some students. If this occurs, the students should be able to justify their position appropriately. The suggested solution is not intended to provide a fully completed class diagram; it is meant to be a basic, rough design of the classes.

- **Customer class**

Attributes	Operations
E-Mail	LogOn (E-mail, Password)
Password	
Full Name	
Date of Birth	
Sex	
Address	

- **Product class**

Attributes	Operations
Name	RetrieveDetails(ID)
Image	
Full Description	
Manufacturer	
Price	

- **Order Class**

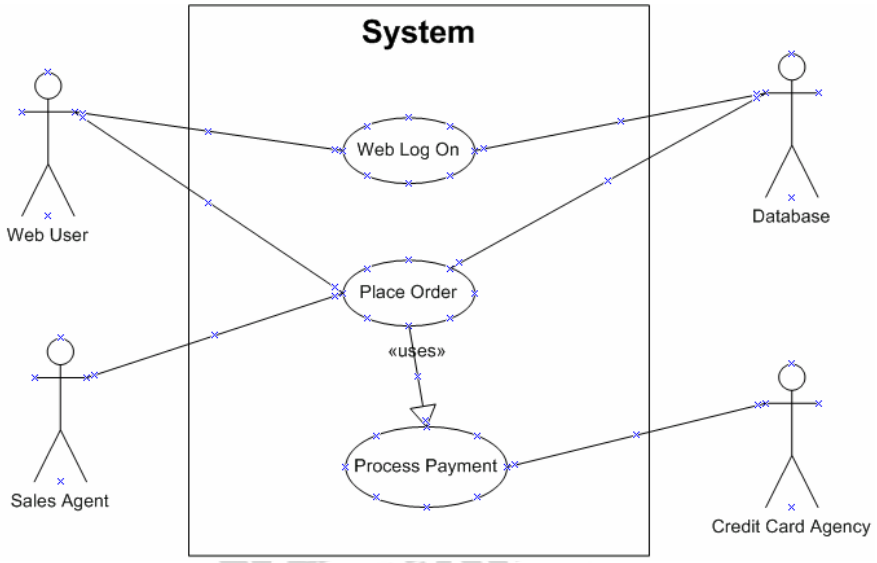
Attributes	Operations
Product ID	ConfirmOrder( )
Quantity	
Delivery Date	

Demonstrations

This section provides demonstration procedures that will not fit in the margin notes or are not appropriate for the student notes.

Creating Use Case Diagrams

The illustration provides an example of the finished diagram.



↙ To create the Visio drawing

1. Open Visio. In the **Category** list, click **Software**. In the **Template** list, click **UML Model Diagram** to create an empty drawing.
2. On the **View** menu, click **Grid** so that the grid lines are not displayed.

↙ To create the use case diagram

1. In the **Shapes** toolbox, click the **UML Use Case** tab, and then use the shapes to create the following actors and use cases. Set the name for each item by double-clicking the item and changing the **Name** property.

Type	Name
System Boundary	
Actor	**Web User**
	Sales Agent
	Credit Card Agency
	Database
Use Case	**Web Log On**
	Place Order
	Process Payment

2. Arrange the items so that only the use cases are displayed within the system boundary.

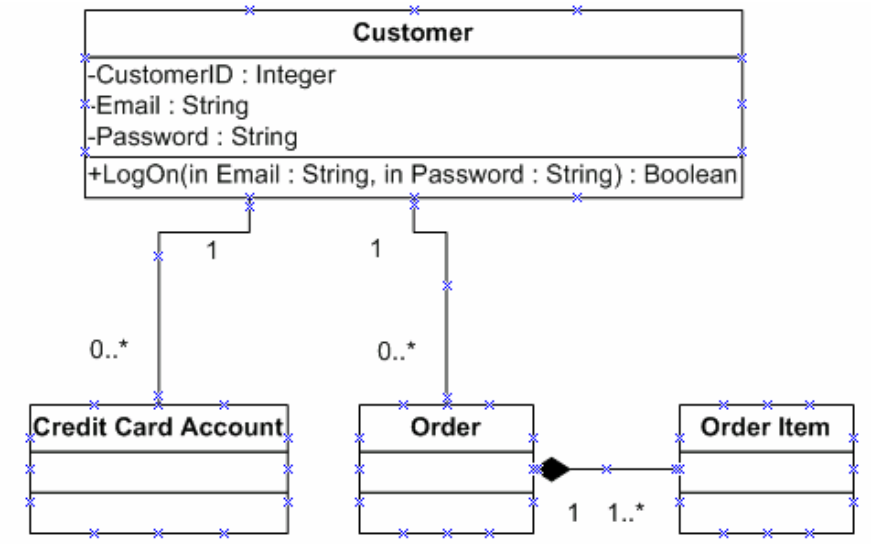
3. Use the tools in the **Shapes** toolbox to create associations between the following items.

Shape	Link from	Link to
Communicates	Web User actor	Web Log On use case
Communicates	Web User actor	Place Order use case
Communicates	Sales Agent actor	Place Order use case
Communicates	Web Log On use case	Database actor
Communicates	Place Order use case	Database actor
Uses	Place Order use case	Process Payment use case
Communicates	Process Payment use case	Credit Card Agency actor

4. Select all of the associations, right-click any of the selected associations, and then click **Shape Display Options**. Clear all of the **End Options** check boxes, select **Apply to the same selected UML shapes in the current drawing window page**, and click **OK**. This will remove the association multiplicities and end names from the diagram.
5. Double-click the **Web User** actor, and then type the following description into the **Documentation** box:
" A Web user customer who can place orders over the Internet."
6. Double-click the **Web Log On** use case, and then type the following description into the **Documentation** box:
" A customer logs on to the system using an e-mail address and password. The e-mail address and password are then validated by the database. If they are not valid, an error message is displayed to the customer. If they are valid, a welcome screen is displayed."
7. On the **File** menu, click **Save**, and then go to the *install folder*\DemoCode\Mod04\Use Cases folder. Rename the file **UseCase.vsd**, click **Save**, and then click **OK**.

Creating Class Diagrams

The following illustration provides an example of the finished diagram.



↙ To create the UML classes

1. Open Visio, and then open the drawing you saved from the use case demonstration.
2. In Model Explorer, right-click **Top Package**, point to **New**, and then click **Static Structure Diagram**.
3. In the **Shapes** toolbox, click the **UML Static Structure** tab, and then use the **Class** tool to create the following classes:
   - Customer
   - Credit Card Account
   - Order
   - Order Item

↙ To create the associations

1. In the **Shapes** toolbox, click the **Binary Association** tool to create an association from the **Customer** class to the **Order** class. Point out that the multiplicity end will be determined by the order in whichyou create the association. Double-click the association, and in the **UML Association Properties** dialog box, set the following values.

Association ends	Multiplicity value
First item in list	**1**
Second item in list	**0..***

2. Create an association from the **Customer** class to the **Credit Card Account** class. In the **UML Association Properties** dialog box, set the following values.

Association ends	Multiplicity value
First item in list	**1**
Second item in list	**0..***

3. Use the **Composition** tool to create an aggregation between the **Order** and **Order Item** classes. In the **UML Association Properties** dialog box, set the following values.

Association ends	Multiplicity value
First item in list	**1**
Second item in list	**1..***

4. Explain that this multiplicity setting is appropriate because an **Order** must contain at least one **Order Item** to be valid.
5. Click the **Customer-Credit Card Account** association, hold down the SHIFT key, and click the **Customer-Order** association. Right-click any of the selected associations, and then click **Shape Display Options**. Clear the **First end name** and **Second end name** check boxes, select **Apply to the same selected UML shapes in the current drawing window page**, and then click **OK**.
6. Repeat this process for the composition association.

↙ To modify the Customer class

1. Double-click the **Customer** class, and in the **UML Class Properties** dialog box, type the following description into the **Documentation** box:
   "The Customer class stores information about a particular customer and allows logging on to the system."
2. In the **Categories** list, click **Attributes**, and then click the first line in the list of attributes. Use this list to add the following attributes and set the following properties.

Attribute	Type	Visibility
CustomerID	**VB::Integer**	private
Email	**VB::String**	public
Password	**VB::String**	public

3. In the **Categories** list, click **Operations**, and then click the first line in the list of operations. Use this list to add an operation called **LogOn**, specifying a return type as **VB::Boolean**. Click the **Properties** button to get to the **Documentation** box, and then type the following text:
   "The LogOn method validates an e-mail address and a password for a customer. If the validation is successful, the attributes of the Customer object will be retrieved from the database. The method returns a success or failure Boolean flag."
4. In the **Categories** list in the **UML Operation Properties** dialog box, click **Parameters**. Create new parameters based on the following values.

Parameter	Type	Kind
Email	**VB::String**	In
Password	**VB::String**	In

5. Point out that there is much more that could be done to complete this class diagram.
6. On the **File** menu, click **Save As**, and go to the *install folder*\DemoCode\Mod04\Class Diagrams folder. Rename the file **ClassDiagrams.vsd**, click **Save**, click **OK**, and then quit Visio.

Module Strategy

Use the following strategy to present this module:

- Designing Classes

  This lesson introduces the concept of use cases and how they can be used to model a system. It also shows how classes, attributes, and operations can be derived from these use cases.

  It should be noted that many of the examples used throughout the module are open to different interpretations. Instructors should be able to discuss various interpretations from students.

- Object-Oriented Programming Concepts

  This lesson introduces the basic concepts of object-oriented design, including encapsulation, association, aggregation, and class diagrams.

  Some of the topics covered may be relatively basic for some students; however, these areas are essential for those new to class design. Cover this lesson quickly if the student knowledge level is appropriate.

- Advanced Object-Oriented Programming Concepts

  This lesson covers more advanced object-oriented concepts, such as inheritance (or generalization), interfaces, and polymorphism.

  Some students may already be familiar with these advanced topics, but these topics are essential for using Visual Basic .NET to its full capability, so this lesson should not be rushed.

- Using Microsoft Visio

  This lesson shows how Visio can help you model a system, particularly with use cases and class diagrams.

  Students who have used other tools such as Visual Modeler will find some familiar features, but they will also find Visio to be a more powerful tool. The module focuses on the creation of use case diagrams and class diagrams (also known as static structure diagrams), but there are many other areas of interest that you can discuss if time permits.

  Note that code generation and reverse engineering are supported in this version of Visio, but that neither are demonstrated.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about object-oriented design and how it can help you to create your Visual Basic .NET applications.

- **Designing Classes**
- **Object-Oriented Programming Concepts**
- **Advanced Object-Oriented Programming Concepts**
- **Using Microsoft Visio**

Delivery Tip
This module does not cover all aspects of object-oriented programming. It concentrates on areas Visual Basic developers need to know to apply object-oriented design techniques when building applications.

Given this, you should refer students to other reference books on object-oriented design.

Developers using Microsoft® Visual Basic® version 4.0 and later have had some exposure to the benefits of an object-oriented approach to programming, but it is only now that you can take full advantage of an object-oriented paradigm if you so choose. To use these new capabilities, you must be familiar with object-oriented programming concepts. This module explains the areas that you must understand to create object-oriented solutions in Visual Basic .NET.

In this module, you will learn how to begin the class design process by using use cases. You will then learn about some common object-oriented programming concepts, including inheritance, interfaces, and polymorphism. Finally, you will learn how to document your system design by using Microsoft Visio® to build use cases and class diagrams.

After completing this module, you will be able to:

- Describe the basics of object-oriented design.
- Explain the concepts of encapsulation, inheritance, interfaces, and polymorphism.
- Create classes based on use cases.
- Model classes for use in Visual Basic .NET by using Visio.

◆ Designing Classes

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
This lesson discusses use cases and how they can be used to begin the class design process.

- **Use Case Diagrams**
- **Use Case Diagram Example**
- **Use Case Descriptions**
- **Extending Use Cases**
- **Converting Use Cases into Classes**

You can use the Unified Modeling Language (UML) to help you analyze requirements by graphically showing interactions within the system.

After completing this lesson, you will be able to:

- Design classes for use in applications created in Visual Basic .NET.
- Begin the process of designing classes by using use cases.
- Derive classes based on an existing use case.

Use Case Diagrams

Topic Objective
To examine use case diagrams.

Lead-in
Use case diagrams are the first step in designing classes for your solution.

- **Use Cases**
  - Provide a functional description of major processes
  - Use a non-technical language to describe the process
  - Show a boundary around the problem to be solved
- **Actors**
  - Graphically describe who or what will use the processes

A *use case* is a collection of possible sequences of interactions in a system. Use case diagrams are an integral part of designing a modern software application. You can use the Unified Modeling Language (UML) to help you analyze requirements by graphically showing interactions within the system. Use case diagrams consist of individual use cases and actors.

What Are Use Cases?

You can employ use cases to:

- Provide a functional description of major processes.

  Use cases usually represent common interactions between a user and a computer system. A use case describes a process that is needed in order for a system to fulfill its requirements.

- Use a non-technical language to describe the process.

  Each individual use case will describe a particular process in a non-technical language that can be understood by the intended application user.

  The use case description should concentrate on the sequence of events and decisions that must be made in the process rather than on an overly detailed look at how things will be implemented.

- Show a boundary around the problem to be solved.

  This boundary helps the designers and developers to concentrate on individual processes without getting lost in the detail of the communications between actors and the processes at this stage.

Creating Use Cases

When you create a use case, you will often focus on the people or actors who will be using the system. If you are working with an existing system, you will typically meet with users to discuss how they use the existing system to complete their tasks. You might also observe intended users of your system to record their processes. Each of these approaches leads to the creation of large descriptions that you can then distill into smaller individual use cases.

Actors

An actor graphically represents who or what will use the system. An actor is often a role that a person plays when interacting with a particular process of the system. Actors request functionality of the system by way of a use case and can use many different use cases. Multiple actors can also use the same use case.

Four categories of actors include:

- Principal actors

  People who use the main system functions.

- Secondary actors

  People who perform administration or maintenance tasks.

- External hardware

  Hardware peripheral devices that are part of the application domain and must be used.

- Other systems

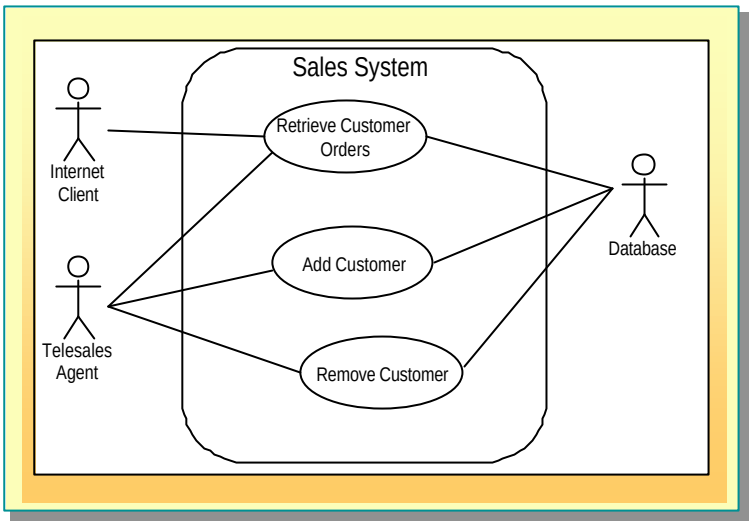
  Other systems with which the system must interact.

Identifying actors can help you understand what the system should do and who can use each section of the system. Actors provide a starting point for designing security permissions for the different roles that interact with the system. An actor can also represent another software application or even a hardware device that interacts with the new system.

Use Case Diagram Example

Topic Objective
To examine an example of a use case diagram.

Lead-in
A simple use case diagram is made up of one or more use cases, actors, and the communications between them.



A simple use case diagram is made up of one or more use cases, actors, and the communications between them.

Use Case Example

The example on the slide shows a sales system use case diagram that represents three different processes (or use cases) and the three actors that interact with them.

The illustration shows the system boundary that encapsulates the use cases within its walls. It also shows external parts of the system, such as a database, as external to the system boundary, or outside the walls.

Each actor communicates with one or more of the use cases. These actors help to establish the roles in the system and where security boundaries will need to be set later in lifetime of the application. In the example, you can see that two of the actors are people and that one is a database that is considered another part of the system. The Internet Client actor will only interact with the Retrieve Customer Orders use case, while the Telesales Agent will interact with all three use cases.

Delivery Tip
Point out the different actors and what they represent. Note that the Database actor is not a person but that this is acceptable.

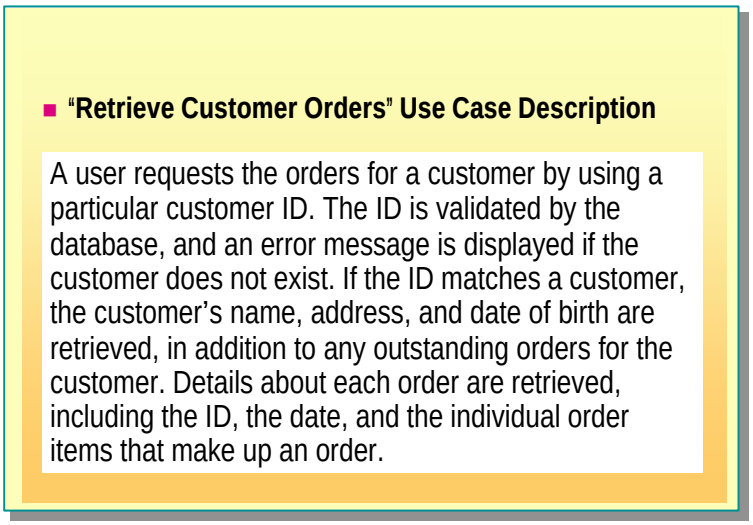
Also point out and briefly describe each use case. Don't give precise details about the use cases because this is covered in more detail later in this module.

These use cases are not identical to the cargo system and are meant to be fairly generic.

Use Case Descriptions

Topic Objective
To examine a particular example of a use case description.

Lead-in
Use case descriptions provide information about a particular scenario. Here is an example of one of those scenarios from the previous use case diagram.



Use case descriptions provide information about a particular scenario. Here is an example of a scenario from the use case diagram in the preceding topic .The Retrieve Customer Orders use case description reads as follows:

Delivery Tip
Read through the use case description slowly.

" A user requests the orders for a customer by using a particular customer ID. The ID is validated by the database, and an error message is displayed if the customer does not exist. If the ID matches a customer, the customer's name, address, and date of birth are retrieved, in addition to any outstanding orders for the customer. Details about each order are retrieved, including the ID, the date, and the individual order items that make up an order."

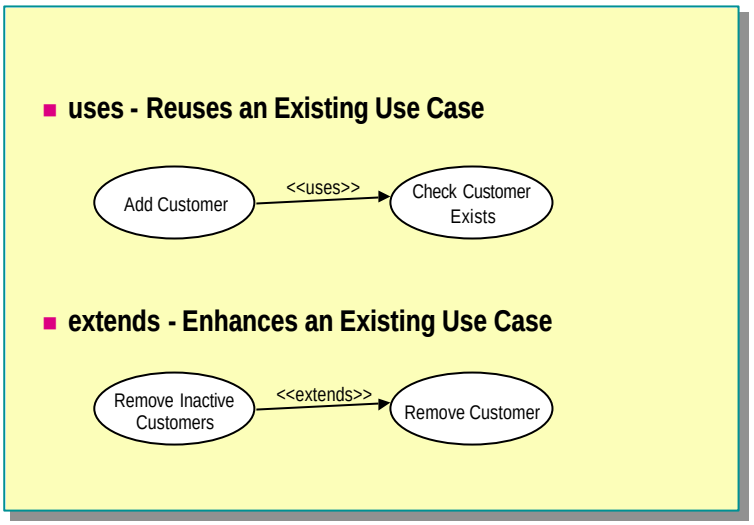
By working through this use case description, you can see that it starts with a request from a user to perform a certain action. Some validation then takes place, and an error message is displayed if appropriate. If the validation succeeds, information about the customer, the customer's orders, and the order items for an order are retrieved and displayed to the user.

Notice that the precise details of how information is retrieved and displayed are not mentioned. The point of a use case description is simply to describe the business process, not provide all information for the developer. From this use case description, various classes can be derived that will become the first version of the detailed design for the system solution.

Extending Use Cases

Topic Objective
To explain how use cases can be extended.

Lead-in
Use cases can be reused and extended with two keywords.



Instead of developing a separate use case for every process, you can reuse use cases. This is a less repetitive approach and can save you a lot of time.

Two keywords that you can use to extend use cases are as follows:

Delivery Tip
The **uses** syntax is straightforward, but the **extends** syntax is more complex.

Point out that **extends** is mainly used to give a more precise description of a general use case. It is particularly useful if the extending use case deals with an unusual situation.

- **uses**

  The **uses** keyword allows an existing use case to be reused by other use cases. This means that if a use case needs to perform some sort of action that has been created elsewhere, there is no need to duplicate the effort. The slide shows how the Add Customer use case might use the Check Customer Exists use case that has been defined for reuse. This means that the Check Customer Exists use case can be called directly from an actor or reused from other use cases.

- **extends**

  The **extends** keyword allows a use case to describe a variation on normal behavior in an existing use case. In other words, it allows a new use case to perform a similar action to an existing use case, except that the new use case builds on the existing one, performing a more complex action. The illustration shows how the Remove Customer use case can be extended to remove the inactive customers who have not placed orders within two months.

Converting Use Cases into Classes

Topic Objective
To describe the process of designing initial classes based on a use case description.

Lead-in
Use case descriptions can be used to create initial class designs.

- **Use Case Descriptions Provide the Basis for Initial Class Design**
  - Nouns = classes or attributes

A user requests the orders for a customer by using a particular customer ID. The ID is validated by the database, and an error message is displayed if the customer does not exist. If the ID matches a customer, the customer's name, address, and date of birth are retrieved, in addition to any outstanding orders for the customer. Details about each order are retrieved, including the ID, the date, and the individual order items that make up an order.

  - Verbs = operations (methods)

    Example: ValidateCustomer, RetrieveOrders, RetrieveOrderItems

---

Delivery Tip
Creating classes, attributes, and operations based on use case descriptions can be a subjective process. Make it clear to the students that this is only a preliminary draft of a class design.

You can create initial class designs by finding the nouns and verbs in use case descriptions. You can begin to identify classes or attributes for classes by finding the nouns in the use case description. You can begin to identify processes that can become operations or methods in a class by finding the verbs in the use case description.

Identifying Classes and Attributes

Using the Retrieve Customer Orders use case description as an example, you can identify several nouns that may lead to classes or attributes of those classes.

"A **user** requests the **orders** for a **customer** by using a particular **customer ID**. The ID is validated by the database, and an error message is displayed if the customer does not exist. If the ID matches a customer, the customer's **name**, **address**, and **date of birth** are retrieved, in addition to any outstanding orders for the customer. Details about each **order** are retrieved, including the **ID**, the **date**, and the individual **order items** that make up an order."

Based on the use case description, you might conclude that the following classes and attributes are possible.

Class	Attributes
User	<Unknown at this stage>
Customer	CustomerID
	Name
	Address
	Date of Birth
	Orders
Order	ID
	Date
	Order Items
Order Items	<Unknown at this stage>

These are only possible classes and attributes based on the use case description, and they may be removed or modified in later design stages.

Identifying Operations and Methods

The name of the use case—Retrieve Customer Orders—gives you an idea for an initial operation or method that begins the business process. The operation ValidateCustomer can be derived from the statement "the id is validated." The purpose of this operation is to check the validity of a customer ID. The verb "Retrieve" can also be used to derive an operation called RetrieveOrders on the **Customer** class.

Although there may not be any other specific verbs in this use case description example, you can see how verbs can be used to produce method names.

Note Using use case descriptions for initial class design is a subjective process. You may have identified classes, attributes, or operations that differ from those that the example shows. This is not unusual because this is only a first stage in the class design process, and the differences will often disappear with further design. However, there is generally more than one correct answer.

Practice: Deriving Classes from Use Cases

Topic Objective
To practice deriving classes and attributes from use cases.

Lead-in
This practice allows you to work in pairs to derive some classes and attributes from a use case description.

In this practice, you will work in pairs to select classes and some attributes or operations based on the following use case descriptions. When you have finished, the instructor and class will discuss the results as a group.

Delivery Tip
Students will work in pairs and then discuss the outcome as a group. While there should not be too many variations in the answers, you should expect different interpretations and be able to explain why a given student answer is correct or incorrect.

Expected answers are given in the Instructor Notes for this module.

Customer Log On Use Case Description

" A customer logs on to the system by using an e-mail address and password. If the e-mail address or password is not valid, a message stating that information was incorrectly entered is displayed to the customer. If the e-mail address and password are valid, a welcome screen is displayed, showing the customer's full name, date of birth, gender, and address."

Place Order Use Case Description

"The customer selects the product to add to the order by using a product name. To confirm that the customer has selected the correct product, an image, a full description, a manufacturer, and a price is displayed to the customer. The customer must enter a quantity value for the order and press a confirmation button to continue the order process. A delivery date for the order must also be entered by the customer."

◆ Object-Oriented Programming Concepts

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
There are several object-oriented programming concepts that you must be familiar with when developing classes for Visual Basic .NET.

- **Comparing Classes to Objects**
- **Encapsulation**
- **Abstraction**
- **Association**
- **Aggregation**
- **Attributes and Operations**

This lesson introduces several important concepts of object-oriented design that will improve the way you design your Visual Basic .NET solutions.

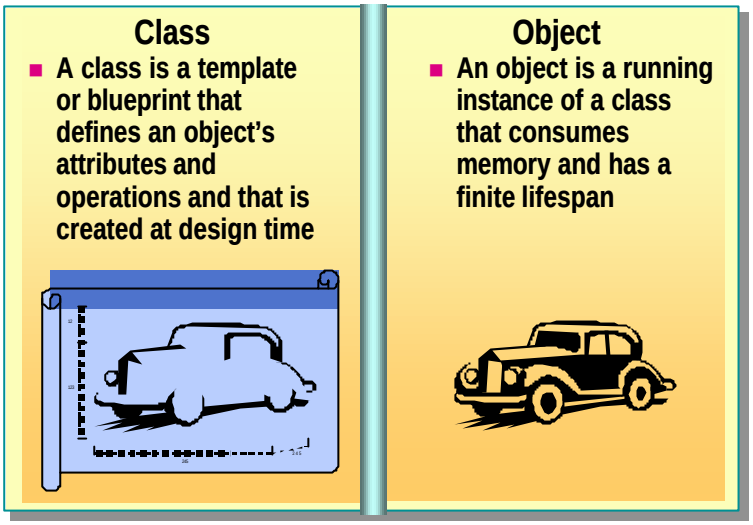
After completing this lesson, you will be able to:

- Distinguish between objects and classes.
- Describe encapsulation, association, and aggregation.
- Explain how properties and methods are used to define a class and the different levels of scope that make them accessible or inaccessible.
- Explain how classes are represented in class diagrams along with their relationships to each other.

Comparing Classes to Objects

Topic Objective
To examine the differences between a class and an object.

Lead-in
Classes and objects are closely related. We cannot talk about an object without regard to its class. However, there are some important differences between them.



Delivery Tip
It may be useful to discuss the differences between objects and classes with another example that is relevant to Visual Basic developers.
A button is defined internally in Visual Basic as being rectangular, as having a caption, and as having the ability to be clicked. This is an example of a class.

You can place as many instances of a button on a form as you want, and changing the caption of one doesn't affect the others. These are examples of objects.

Likewise, clicking one instance of a button does not cause the others to be clicked.

The object-oriented terms *class* and *object* often create some confusion because they can be easily misused by developers when describing a system.

Class

A class is an abstract data type containing data, a set of functions to access and manipulate the data, and a set of access restrictions on the data and on the functions.

You can think of a class as a template or a blueprint for an object. This blueprint defines attributes for storing data and defines operations for manipulating that data. A class also defines a set of restrictions to allow or deny access to its attributes and operations.

A car is an example of a class. We know that a car has attributes, such as the number of wheels, the color, the make, the model, and so on. We know that it also has operations, including unlock door, open door, and start engine.

Object

Objects are instances of classes. A single blueprint or class can be used as a basis for creating many individual and unique objects.

If you consider classes and objects in Visual Basic terms, a class is created at design time and will exist forever, whereas an object is instantiated at run time and will only exist as long as required during the application execution.

For example, an instance of a car would contain specific information for each attribute, such as number or wheels equals four, color equals blue, and so on.

Objects exhibit three characteristics:

- Identity
- Behavior
- State

Identity

One object must be distinguishable from another object of the same class. Without this characteristic, it would be impossible to tell the difference between the two objects, and this would cause great difficulties for developers. This difference could be a simple identifier such as a unique ID number assigned to each object, or several of each object's attributes could be different from those of the other objects.

A particular car could be the same make, model, and color as another car, but the registration numbers cannot be identical. This difference provides a way to distinguish two otherwise identical cars.

Behavior

Objects exist to provide a specific behavior that is useful. If they did not exhibit this characteristic, we would have no reason to use them.

The main behavior or purpose of a car is to transport people from one location to another. If the car did not provide this behavior, it would not perform a useful function.

State

State refers to the attributes or information that an object stores. These attributes are often manipulated by an object's operations. The object's state can change by direct manipulation of an attribute, or as the result of an operation. A well-designed object often only allows access to its state by means of operations because this limits incorrect setting of the data.

A car keeps track of how far it has traveled since it was created in the factory. This data is stored internally and can be viewed by the driver. The only way to alter this data is to drive the car, which is an operation that acts upon the internal state.

Encapsulation

Topic Objective
To explain the concept of encapsulation.

Lead-in
Encapsulation is one of the key concepts of object-oriented design.

- **How an Object Performs Its Duties Is Hidden from the Outside World, Simplifying Client Development**
  - Clients can call a method of an object without understanding the inner workings or complexity
  - Any changes made to the inner workings are hidden from clients

---

Encapsulation is the process of hiding the details about how an object performs its duties when asked to perform those duties by a client. This has some major benefits for designing client applications:

Delivery Tip
To continue with the Visual Basic button example, we do not know how a button receives a click event from Microsoft Windows , just that it does and that we can use it.

- Client development is simplified because the clients can call a method or attribute of an object without understanding the inner workings of the object.
- Any changes made to the inner workings of the object will be invisible to the client.
- Because private information is hidden from the client, access is only available by means of appropriate operations that ensure correct modification of data.

Example

Driving a car is an example of encapsulation. You know that when you press the accelerator the car will move faster. You do not need to know that the pedal increases the amount of fuel being fed into the engine, producing more fuel ignition and thereby speeding up the output to the axle, which in turn speeds up the car's wheels, which has the final effect of increasing your speed. You simply need to know which pedal to press to have the desired effect.

Likewise, if the car manufacturer changes the amount of fuel being mixed with oxygen to alter the combustion, or creates a drive-by-wire accelerator pedal, you do not need to know in order to increase your speed. However, if the manufacturer replaces the accelerator pedal with a sliding throttle device, similar to what you would find in an aircraft, you may need to know about it!

Abstraction

Topic Objective
To explain the concept of abstraction.

Lead-in
Abstraction allows you to concentrate on what is important in your objects and ignore what is not.

- **Abstraction Is Selective Ignorance**
  - Decide what is important and what is not
  - Focus on and depend on what is important
  - Ignore and do not depend on what is unimportant
  - Use encapsulation to enforce an abstraction

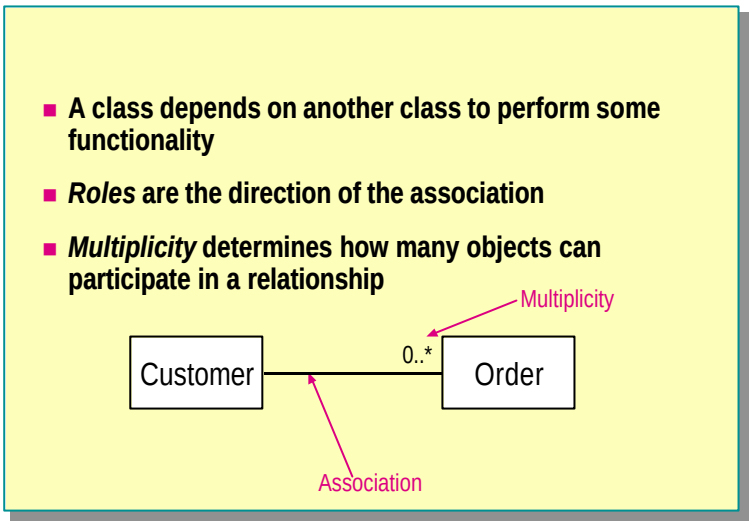
Abstraction is the practice of focusing only on the essential aspects of an object. It allows you to selectively ignore aspects that you deem unimportant to the functionality provided by the object. A good abstraction only provides as many operations and attributes as are required to get the job done. The more operations and attributes provided, the more difficult to use the object becomes. If an object is simple to use because it includes only the essential operations, there is a greater possibility that it can be reused by other applications.

A good abstract design will also limit a client's dependency on a particular class. If a client is too dependent on the way an operation is performed by an object, any modification to the internal aspects of that operation may impact the client, requiring that additional work be completed. This is often known as *the principle of minimal dependency*.

Association

Topic Objective
To use association to examine relationships between classes.

Lead-in
Classes rarely exist in isolation and are often dependent on other classes for certain functions. Association is one way to link classes together.



An *association* is a relationship between two classes. It represents a dependency, in that one class needs another class to accomplish a specific function.

The slide shows an example of an association between a **Customer** class and an **Order** class. In this relationship, it does not make sense to be able to create an order that does not belong to a particular customer, so we would specify that the **Order** class is dependent on the **Customer** class. The association shows this relationship.

Roles

A *role* is the direction of an association between two classes. The illustrated association between **Customer** and **Order** contains two inherent roles: one from **Customer** to **Order**, and another from **Order** to **Customer**. Roles can be explicitly named by using a label, or implied if not included on a diagram like that of the class name.

Multiplicity

Multiplicity is used to define a numeric constraint on an association that restricts how many objects exist within the relationship. If no multiplicity values are specified, there is an implicit one-to-one relationship between the classes.

In the illustration on the slide, a customer can place many orders. This relationship is signified by the 0..* multiplicity range on the **Order** end. As no multiplicity value is specified at the **Customer** end, an implicit value of one is assumed, signifying that an order can only have one customer.

The following table lists the other possibilities for multiplicity.

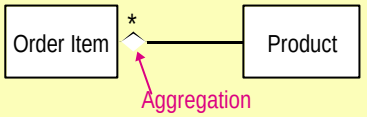
Symbol	Meaning
*	Many (zero or more)
0..1	Optional (zero or one)
1..*	One or more
2-3, 6	Specific possibilities
{constraint}	Rules including well-known constraints like Order or Mandatory, or other unique business rules specific to your solution.

Aggregation

Topic Objective
To examine the concept of aggregation.

Lead-in
Some classes are more complex than others. Aggregation describes a complex class that is made up from other simpler classes.

- **A complex class containing other classes**
- **A "part-of" relationship**
- **Example:**
  - An **Order Item** class contains a **Product** class
  - A **Product** class is a "part of" an **Order Item** class



Delivery Tip
This topic may cause some discussion because the formal definitions of aggregation and composition cause much disagreement between UML experts.

Do not go into too much detail because the differences are not significant to Visual Basic .NET development. In addition, be aware that leading UML experts still debate the definitions of association, aggregation, and composition.

Aggregation represents a relationship where simple objects form parts of a more complex whole object.

This type of relationship is often used when an object does not make any sense in its own right, such as **Order Item** in the example on the slide. An **Order Item** needs to exist as part of a more complex object, such as an **Order**. The **Order** itself is only useful as a complete object that includes individual **Order Items**. The **Order Item** class can be referred to as the *part classifier* and the **Order** class as the *aggregate classifier*.

You can also specify the number of parts that make up the whole by using multiplicity on an aggregation relationship. The slide shows that an **Order** can be made up of one or more **Order Items**.

The words *aggregation* and *composition* are sometimes used as though they are synonyms. In UML, composition has a more restrictive meaning than aggregation:

- Aggregation

  Use aggregation to specify a whole/part relationship in which the lifetimes of the whole and the parts are not necessarily bound together, the parts can be traded for new parts, and parts can be shared. Aggregation in this sense is also known as *aggregation by reference*.

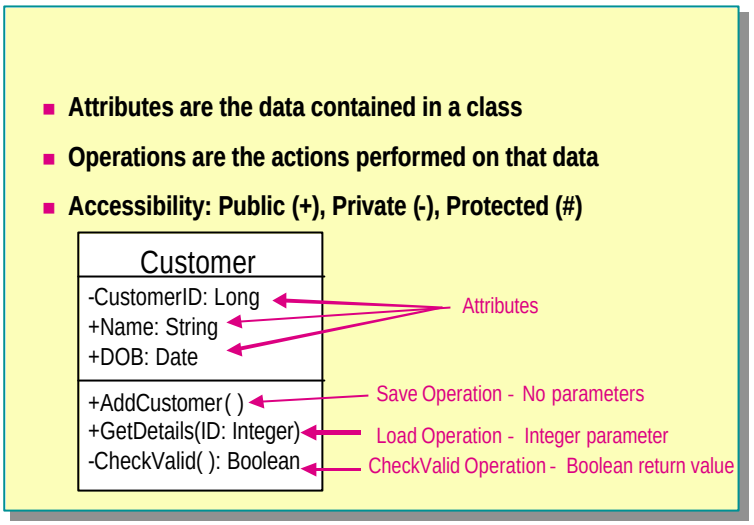
- Composition

  Use composition to specify a whole/part relationship in which the lifetimes of the whole and the parts are bound together, the parts cannot be traded for new parts, and the parts cannot be shared. Composition is also known as *aggregation by value*.

Attributes and Operations

Topic Objective
To explain class attributes and operations.

Lead-in
Classes usually contain data and functions that act upon this data. These are known as attributes and operations.



Classes are usually made up of data and actions performed on this data. These are known as *attributes* and *operations* respectively, but developers also call them *properties* and *methods*. These attributes and operations are also defined with an accessibility setting.

Attributes

Attributes are the data members of the class. They can be of any data type, including String, Decimal, or even another class. Each attribute will also have an accessibility option specified, as shown in the following table. In Visual Basic .NET, public attributes will be implemented as either class-level variables or, more appropriately, as class properties that encapsulate internal variables.

Operations

Operations are the actions performed on internal data within the class. They can take parameters, return values, and have different accessibility options specified in the same way that attributes can. In Visual Basic .NET, these are implemented as either functions or subroutines.

Accessibility

Attributes and operations can be defined with one of the access modifiers in the following table.

Value	Meaning
Public (+)	Accessible to the class itself and to any client of the class.
Protected (#)	Only accessible to a child class when used for inheritance. (Inheritance will be covered later in this module.)
Private (-)	Only accessible by code within the class that defines the private attribute or operation.

◆ Advanced Object-Oriented Programming Concepts

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
There are some other important concepts that you must understand to take full advantage of the object-oriented capabilities of Visual Basic .NET.

- **Inheritance**
- **Interfaces**
- **Polymorphism**

This lesson introduces some advanced concepts of object-oriented design: inheritance, interfaces, and polymorphism.

After completing this lesson, you will be able to:

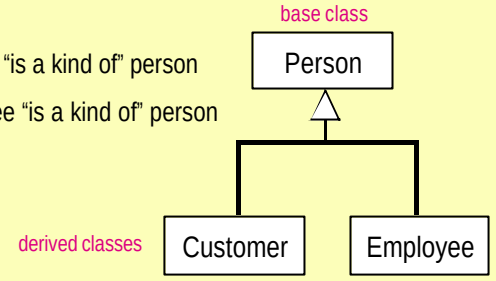
- Explain inheritance.
- Define interfaces.
- Define polymorphism

Inheritance

Topic Objective
To explain the concept of inheritance.

Lead-in
Inheritance is an important concept for object-oriented developers because it allows great flexibility in solution design.

- **Inheritance Specifies an "Is-a-Kind-of" Relationship**
- **Multiple classes share the same attributes and operations, allowing efficient code reuse**
- **Examples:**
  - A customer "is a kind of" person
  - An employee "is a kind of" person

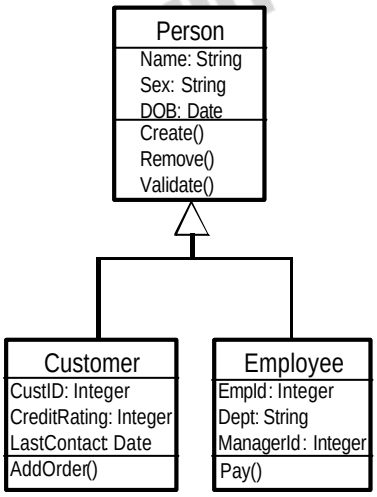


Delivery Tip
Inheritance is a key new feature of Visual Basic .NET, so it may be worthwhile to give as many examples as required for students to understand the concept.

Other examples include the following:
Animal - Dog and Cat.
Vehicle - Car and Boat.
Company - Vendor and Supplier

Inheritance is the concept of reusing common attributes and operations from a base class in a derived class. If the base class does not contain implementation code and will never be instantiated as an object, it is known as an abstract class.

The example below shows an inheritance relationship between the **Customer**, **Employee**, and **Person** classes. The **Person** superclass has attributes defined as **Name**, **Gender**, and **Date of Birth** and contains the operations **Create**, **Remove**, and **Validate**. These are all attributes and operations that could equally be applied to a **Customer** or **Employee** subclass, providing a good deal of reuse.



Specialization occurs when a subclass is created and includes attributes or operations specific to that class.

The **Customer** class has the extra attributes **CustomerID**, **CreditRating**, and **LastContacted** in addition to the inherited ones from the **Person** superclass. It also defines its own operation named **AddOrder** that is specific to the **Customer** class. Having an operation called **AddOrder** would not make sense for either the **Person** class or the **Employee** class.

The **Employee** class has the extra attributes **EmployeeID**, **Department**, and **Manager**. It also defines a unique operation named **Pay** that would not be required in either the **Person** superclass or the **Customer** subclass.

If a superclass is not an abstract class and contains some implementation code, the subclass can inherit the code from the superclass or override it by providing its own code. This reuse of code is known as *implementation inheritance* and is the most powerful form of reuse.

Although implementation inheritance is very useful, it can lead to class diagrams and code that are complex and difficult to read. You should ensure that implementation inheritance is used appropriately and not overused.

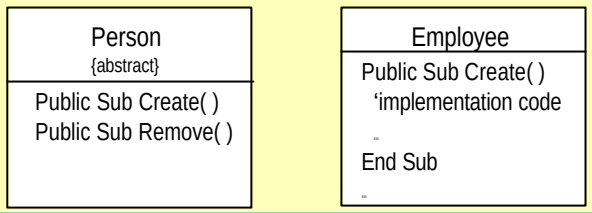
Note Visual Basic 6.0 does not support implementation inheritance, but Visual Basic .NET does. For more information, see Module 5, " Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Interfaces

Topic Objective
To explain interfaces from an object-oriented point of view.

Lead-in
Interfaces are similar to abstract classes in that they contain no implementation code.

- **Interfaces only define the method signatures**
- **Classes define the implementation of the code for the Interface methods**
- ***Interface inheritance* means only the Interface is inherited, not the implementation code**



Interfaces are similar to abstract classes. They define the method signatures used by other classes but do not implement any code themselves.

Delivery Tip
Do not try to explain interfaces from a COM point of view. Concentrate on the fact that an interface defines only the method signatures and not the implementation code.

Interface inheritance means that only the method signatures are inherited and that any implementation code is not. You would need to create separate code in the appropriate inherited method of each derived class to achieve any required functionality. Reuse is therefore more limited in interface inheritance as compared to implementation inheritance because you must write code in multiple locations.

Polymorphism

Topic Objective
To explain the object-oriented concept of polymorphism.

Lead-in
Polymorphism is an object-oriented concept that some Visual Basic developers will be familiar with.

- **The same operation behaves differently when applied to objects based on different classes**
- **Often based on Interface inheritance**
  - Classes inherit from interface base class
  - Each derived class implements its own version of code
  - Clients can treat all objects as if they are instances of the base class, without knowledge of the derived classes

---

Polymorphism is the ability to call the same method on multiple objects that have been instantiated from different subclasses and generate differing behavior. This is often achieved by using interface inheritance. If two subclasses inherit the same interface, each of them will contain the same method signatures as the superclass. Each one will implement the code in an individual way, allowing different behavior to be created from the same method.

Customer
Inherited Sub Create() ‘do specific customer ‘code … End Sub

Employee
Inherited Sub Create() ‘do specific employee ‘code … End Sub

In the above example, the **Customer** and **Employee** classes have inherited from the **Person** superclass. Each class implements its own version of the **Create** method differently, but, because they both inherit the same interface, a client could treat both classes the same.

◆ Using Microsoft Visio

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
Visual Studio .NET includes a built-in modeling tool: Visio.

- **Visio Overview**
- **Use Case Diagrams**
- **Class Diagrams**
- **Creating Class Diagrams**

This lesson introduces the Visual Studio .NET modeling tool: Visio.

After completing this lesson, you will be able to:

- Use Visio to help you design your system solution.

Visio Overview

Topic Objective
To give an overview of the capabilities of Visio.

Lead-in
Visio provides many ways to help design and document your system.

- **Supports:**
  - Use case diagrams
  - Class or static structure diagrams
  - Activity diagrams
  - Component diagrams
  - Deployment diagrams
  - Freeform modeling

Visio allows you to design and document your solution from the initial analysis and design stages all the way to the final deployment of your enterprise system.

Delivery Tip
Use cases and class diagrams have already been discussed, so briefly review these models but concentrate on the other models.

It supports many different models, including the following.

Use Case Diagrams

As you have seen previously in this module, use cases are created to document the interactions that take place between the actors and the processes in the system. Visio allows you to model these diagrams and document the use case descriptions within these diagrams.

Class or Static Structure Diagrams

This UML diagram provides a view of some or all of the classes that make up the system. It includes their attributes, their operations, and their relationships. Visio supports all aspects of class diagrams, including attribute visibility, association roles, and interfaces.

Activity Diagrams

This UML diagram provides a view of the system's workflow between activities in a process. They can be used to model the dynamic aspects of the system, usually based on one or more use case descriptions. You can use initial states, transitions, decisions, and final states to model this view.

Component Diagrams

This UML diagram allows you to model physical aspects of a system, such as the source code, executables, files, and database tables. You can use interfaces, components, packages, and dependencies to model this view.

Deployment Diagrams

This UML diagram gives a view of the physical nodes (computational devices) on which the system executes. This type of diagram is especially useful when the system will involve more than one computer, such as in an enterprise solution. Nodes, component instances, and objects are the main shapes used in this diagram.

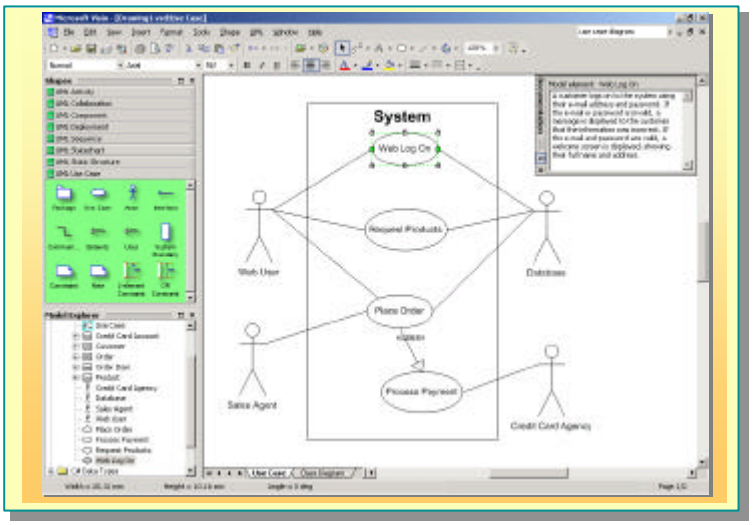
Freeform Modeling

Visio allows you the flexibility to create freeform models that do not need to adhere to the UML standards. This allows you to create a diagram that incorporates common UML shapes such as classes and components in addition to non-UML shapes such as COM and flowchart shapes. You have the option to validate all or part of your model to see whether it conforms to UML semantics.

Use Case Diagrams

Topic Objective
To point out the main elements in a use case diagram in Visio.

Lead-in
Let's take a look at a use case diagram inside Visio.



Visio allows you to create fully featured use case diagrams that include:

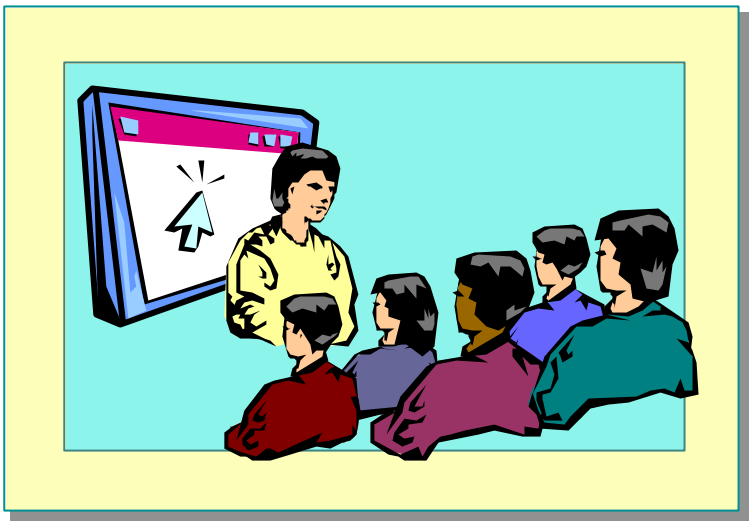
- Actors.
- Use cases.
- Relationships, including association, dependency, and inheritance. These relationships include attributes such as multiplicity, navigability, and stereotype.
- Notes to help store additional information.

You can add descriptions to all of the objects listed above to fully document your model. You can also add any non-UML shapes from the various stencil tabs to create a freeform model.

Demonstration: Creating Use Case Diagrams

Topic Objective
To demonstrate how to create use case diagrams by using Visio.

Lead-in
Creating use cases is one of the initial design steps you will take in Visio.



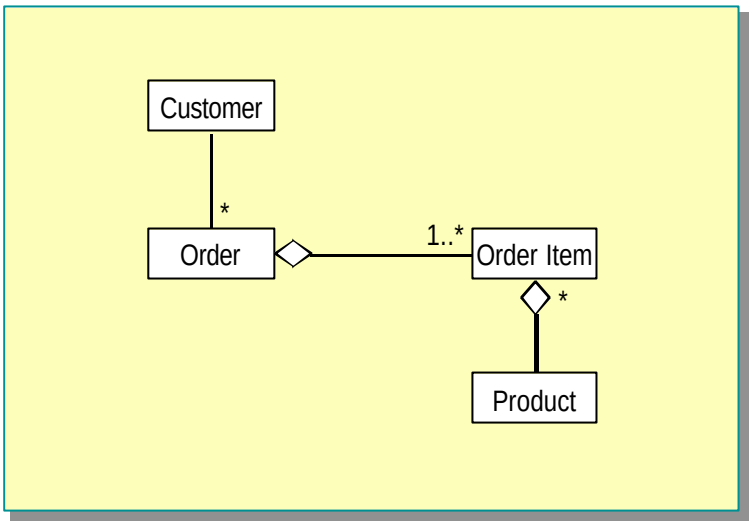
Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to create a use case diagram in Visio. Note that the use cases created in the demonstration do not represent a completed model; this exercise is for demonstration purposes only.

Class Diagrams

Topic Objective
To examine an example of a class diagram.

Lead-in
We can put all of these relationships together to form a class diagram.



Delivery Tip
Explain the different parts of the class diagram, pointing out that you could argue that a customer does not exist unless it has at least one order.

The various relationships may also provide discussion because some could be designed as composition, aggregation, or association.

Such discussion should be limited because the purpose of the illustration is to examine notation and the overall look of a class diagram.

Class diagrams allow you to graphically put all of these relationships together in a single place. With some refinement, class diagrams will help a designer or developer model the system in enough detail to enable code to be generated and development to begin.

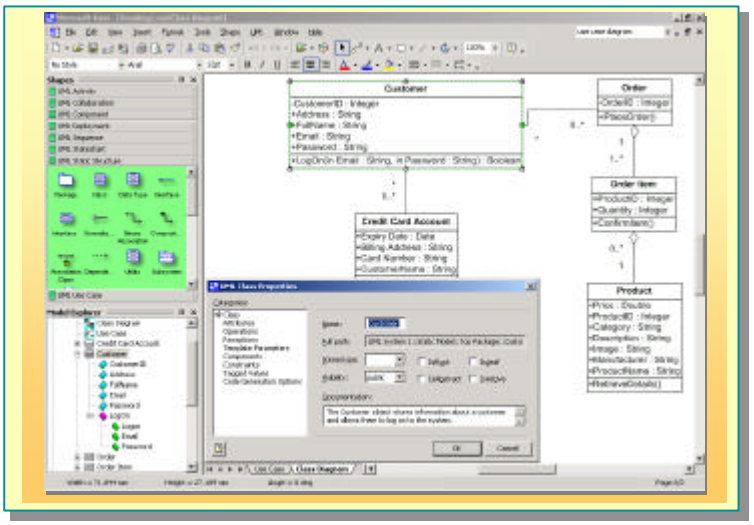
The ability to interpret a class diagram is vital in modern software development. The slide shows that a single **Customer** class is associated with between zero and many **Order** classes. An **Order** class is made up of (or aggregated with) **Order Item** classes. The multiplicity for this relationship shows that there must be at least one **Order Item** for an **Order** to be valid, but that there can be an unlimited number of **Order Items** on the **Order**. Finally, each **Order Item** is associated with a single **Product**, but a **Product** can be associated with many **Order Items**.

Note Class diagrams are also known as *static structure diagrams* in Visio and in some other UML modeling tools.

Creating Class Diagrams

Topic Objective
To point out the main elements of a class diagram in Visio.

Lead-in
Let's take a look at a class diagram inside Visio.



Visio allows you to create extensive class diagrams that include the following elements:

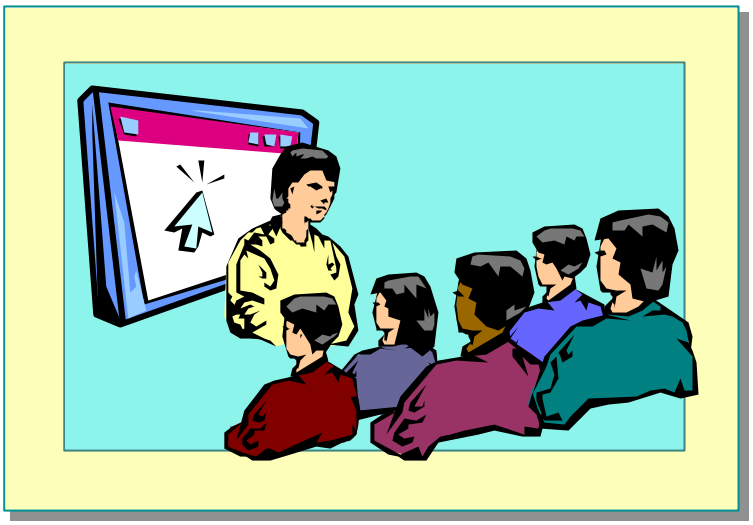
- Classes.
- Abstract classes or interfaces.
- Class attributes with accessibility (private, protected, and public), initial value, and stereotype (data type).
- Class operations with accessibility and parameters (detailed in the following text).
- Operation parameters with direction (in, out, inout, and return) and stereotype (data type).
- Relationships between objects, including association, dependency, aggregation, composition, and inheritance. These relationships include attributes such as multiplicity, navigability, and stereotype.
- Notes to help store additional information.

You can add descriptions to all of the objects listed above to fully document your model. You can also add any non-UML shapes from the various Toolbox tabs to create a freeform model.

Demonstration: Creating Class Diagrams

Topic Objective
To demonstrate the creation of class diagrams in Visio.

Lead-in
Class diagrams are particularly helpful for creating Visual Basic .NET enterprise solutions.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to create a class diagram in Visio. Note that the classes created in the demonstration do not represent a completed model. This exercise is for demonstration purposes only.

Lab 4.1: Creating Class Diagrams from Use Cases

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will look at the use cases for the Cargo system. You will then create class diagrams in Visio for those use case descriptions.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create classes based on use cases.
- Use Visio to create class diagrams.

Prerequisites

Before working on this lab, you must have:

- Knowledge of use cases and class diagrams.
- Familiarity with modeling tools.

Scenario

The Cargo system will provide the basis for many of the remaining lab exercises in this course. It is based on a simple Internet system for collecting and delivering customer packages. Web customers can log on to the system and create a delivery from one location to another. Customers can also contact the company by means of a telephone sales operator who can process the order in a similar fashion. More information about the system is detailed in the use case diagram provided to you. However, while the labs are based on this scenario, you will not create the completed system during this course.

In this lab, you will view the descriptions of existing use cases to help you understand the system. You will then create classes based on those descriptions.

Estimated time to complete this lab: 45 minutes

Exercise 1
Viewing the Cargo Use Case Diagram

In this exercise, you will investigate the use case diagram for the Cargo system, the individual actors, and the use case descriptions.

↙ To open the CargoModeler drawing

1. Open Visio.
2. On the **File** menu, click **Open**. In the *install folder*\Labs\Lab041\Ex01 folder, click the **CargoModeler.vsd** drawing, and then click **Open**.

↙ To view the diagram documentation

1. Double-click each actor individually, and read the associated documentation in the **UML Actor Properties** dialog box.
2. Double-click each use case individually, and read the associated documentation in the **UML Use Case Properties** dialog box.

Exercise 2
Creating the Cargo Classes and Relationships

In this exercise, you will create the Cargo system's classes and relationships, based on the use case descriptions from the previous exercise.

↙ To create the Cargo classes

1. On the **File** menu, click **Open**. Browse for the *install folder*\Labs\Lab041\Ex02\Starter folder, click the **CargoModeler.vsd** drawing, and then click **Open**.
2. In Model Explorer, right-click **Top Package**, point to **New**, and then click **Static Structure Diagram**. Rename the diagram **Classes** in the Model Explorer.
3. In the **Shapes** toolbox, click the **UML Static Structure** tab. Click the **Class** tool and drag it to the drawing. Double-click the new class to display the properties, and rename the class **Customer**.
4. Repeat step 3 to create the following classes: Delivery, Package, Special Package, Invoice, and Payment.

↙ To create the class relationships

1. Click the **Binary Association** tool, and drag it to create an association from the **Customer** class to the **Delivery** class.
2. Double-click the new association to display the **UML Association Properties** dialog box. In the **Association Ends** list, change the **Multiplicity** value in the first line to **1**. Change the **Multiplicity** value in the second line to **0..***. (Note that the order of association ends is related to the order in which you created the associations.)
3. Repeat steps 1 and 2 to create the following relationships and multiplicity values.

From class	To class	Multiplicity value
Customer	**Invoice**	First line: **1**
		Second line: **0..***
Delivery	**Invoice**	First line: **1**
		Second line: **1**
Invoice	**Payment**	First line: **1**
		Second line: **1**
Delivery	**Package**	First line: **1**
		Second line: **1..***

4. Create a generalization relationship from the **Special Package** class to the **Package** class by using the **Generalization** tool.

↙ To hide the association end names

1. Select all the binary associations, right-click any of the selected associations, and then click **Shape Display Options**. Clear the **First end name** and **Second end name** check boxes, select **Apply to the same selected UML shapes in the current drawing window page**, and click **OK**. This will remove the end names from the diagram.
2. Save the drawing.

Exercise 3
Creating the Customer Class

In this exercise, you will create the **Customer** class attributes and operations.

↙ To create the Customer class attributes

1. If you have not completed the previous exercise, use the starter code found in the *install folder*\Labs\Lab041\Ex03\Starter folder.
2. Double-click the **Customer** class to display the **UML Class Properties** dialog box.
3. In the **Attributes** section, click the first line in the list of attributes. Use the information in the following table to add the class attributes.

Attribute	Type	Visibility
CustomerID	VB::Integer	private
Email	VB::String	public
Password	VB::String	public
FirstName	VB::String	public
LastName	VB::String	public
Address	VB::String	public
Company	VB::String	public

↙ To create the LogOn operation

1. In the **Operations** section, click the first line in the list of operations. Add an operation called **LogOn** with a return type of VB::Boolean.
2. Click **Properties**, and in the **Documentation** box, add the following description:
   " Attempts to log on a customer and retrieve his or her details based on e-mail address and password."
3. In the **Parameters** section, create new parameters based on the following values.

Parameter	Type	Kind
Email	VB::String	in
Password	VB::String	in

4. Click **OK** to return to the **UML Class Properties** dialog box.

↙ To create the AddCustomer operation

1. Create the **AddCustomer** operation with a return type of VB::Integer.
2. Click **Properties**, and in the **Documentation** box, add the following description:
   " Adds a new customer to the database based on the input parameters."
3. In the **Parameters** section, create new parameters based on the following values.

Parameter	Type	Kind
Email	VB::String	in
Password	VB::String	in
FirstName	VB::String	in
LastName	VB::String	in
Company	VB::String	in
Address	VB::String	in

4. Click **OK** to return to the **UML Class Properties** dialog box.

↙ To create the GetDetails operation

1. Create the **GetDetails** operation. Click **Properties**, and, in the **Documentation** box, add the following description:
   "Returns the customer details based on the CustomerId received."
2. In the **Parameters** section, create new parameters based on the following values.

Parameter	Type	Kind
CustomerID	VB::Integer	in

3. Click **OK** to return to the **UML Class Properties** dialog box, and then click **OK** to return to the drawing.

↙ To generate Visual Basic .NET source code

1. On the **UML** menu, point to **Code**, and then click **Generate**.
2. Select **Visual Basic** as the **Target Language**. Select **Add Classes to Visual Studio Project**, and then in the **Template** list, click **Class Library**. Rename both the project and the solution as **Cargo**.
3. Click **Browse**, locate the *install folder*\Labs\Lab041\Ex03\Starter folder, and then click **OK**.
4. Select the **Customer** class only for code generation, and click **OK**.
5. Save the drawing and quit Visio.

↙ To view the code

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and then click **Project**. Set the folder location to *install folder*\Labs\Lab041\Ex03\Starter, click **Cargo.sln**, and then click **Open**.
3. View the code for Customer.vb.
4. Quit Visual Studio .NET.

If Time Permits
Viewing the Cargo Design Solution

In this optional exercise, you will investigate the class diagram for the completed Cargo system.

↙ To open the CargoModeler drawing

1. Open Visio.
2. On the **File** menu, click **Open**. Browse for the *install folder*\Labs\Lab041\Ex04 folder, click the **CargoModeler.vsd** drawing, and then click **Open**.

↙ To view the classes

- Investigate each attribute and operation, including the parameters, for each class on the diagram.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.

- Designing Classes
- Object-Oriented Programming Concepts
- Advanced Object-Oriented Programming Concepts
- Using Microsoft Visio

---

1. An actor must be a person that interacts with the system. True or false?

   False. An actor can also be another system or a part of a system.

2. Define the object-oriented term *encapsulation*.

   Encapsulation is the hiding of the details about how an object performs various operations.

3. Define the object-oriented term *inheritance*.

   Inheritance is the reuse of the methods and attributes of a general class in more specialized classes.

4. Describe freeform modeling.

   Freeform modeling is the ability to use UML and non-UML shapes in a Visio diagram.

5. In the following use case description, what are the likely classes and attributes?

   A user requests a listing of grades from a school based on a particular student ID. The ID is validated by the database, and an error message appears if the student ID does not exist. If the ID matches a student, the student's name, address, and date of birth are retrieved, in addition to the grades. The user is prompted to verify the information, and the grades are displayed if the verification succeeds. An error message is displayed if the user is unable to verify the information. Three verification attempts are allowed before the user is automatically logged off. The user is automatically logged off after five minutes of inactivity.

Class	Attributes
User	**<Unknown at this stage>**
Student	**StudentID**
	Name
	Address **Date of Birth**
	Grades
Grades	**ID**