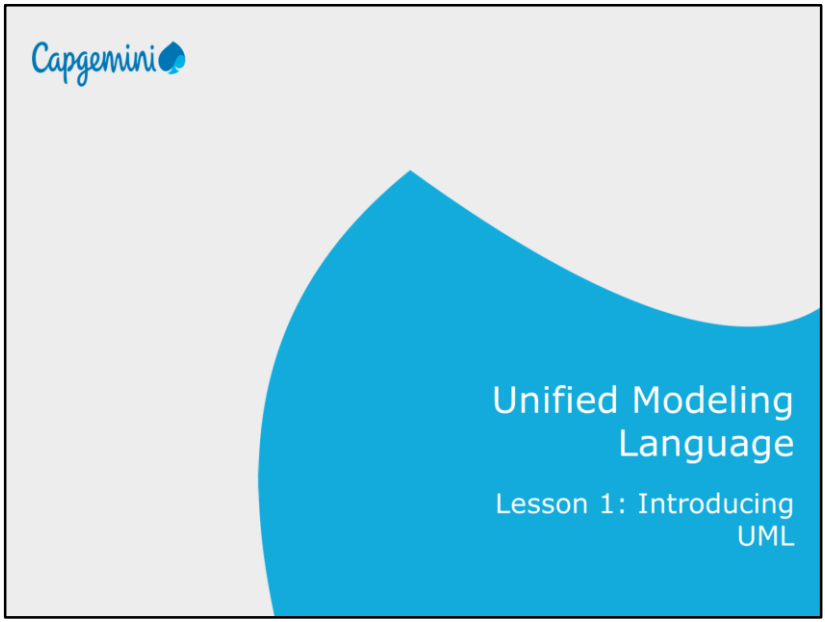
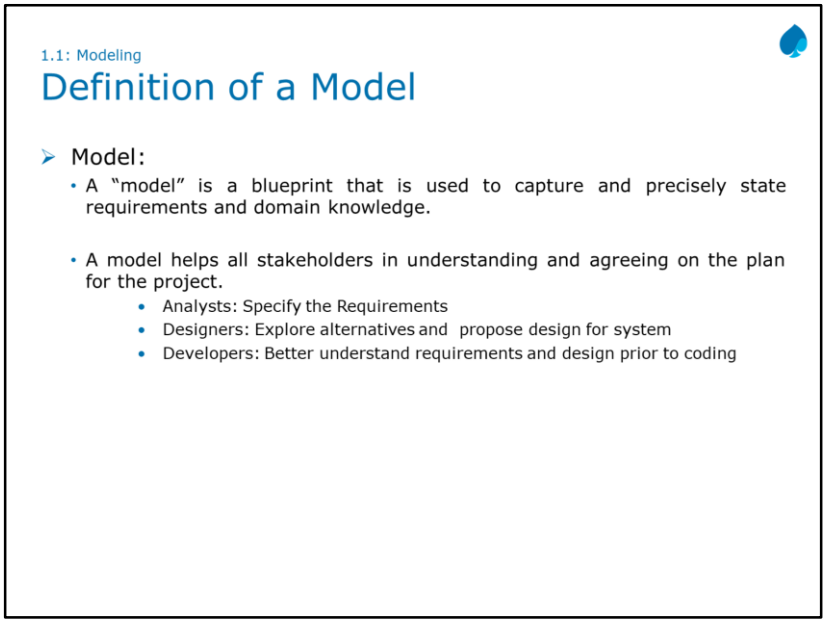

Capgemini
Unified Modeling Language
Lesson 1: Introducing UML

Lesson Objectives

- To understand the following topics:
  - Principles of Modeling
  - Basics of UML – What is UML? What UML is NOT?
  - UML building blocks
  - List of UML diagrams



Principles of Modeling: What is Modeling?

Modeling is an essential activity in many domains, including the fields of construction and engineering.

Actual building of a house is almost always preceded by a blueprint, a model, which describes the architectural layout and other details. Such a blueprint is essential for understanding the system and to convey the same to concerned parties.

The same concept applies to software systems as well. Models can be used to capture the knowledge about the system. A model helps to capture and precisely state the requirements and domain knowledge so that all stakeholders may understand and agree on the plan for the project.

Different models of a software system may capture the following:

requirements about its application domain,

the ways in which users will use the application,

its breakdown into application modules,

common patterns used in its construction, etc.

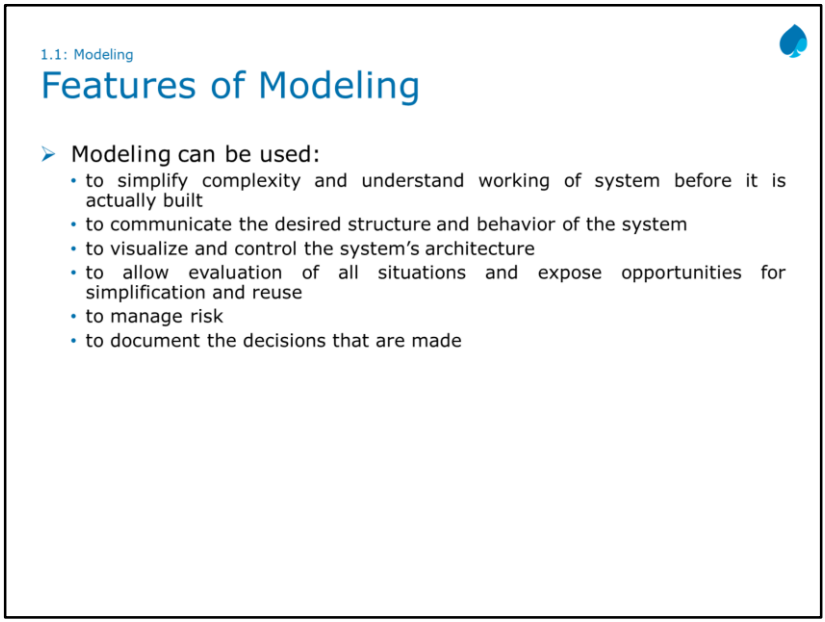
Thus modeling helps the developers easily explore several architectures and design solutions before writing code.

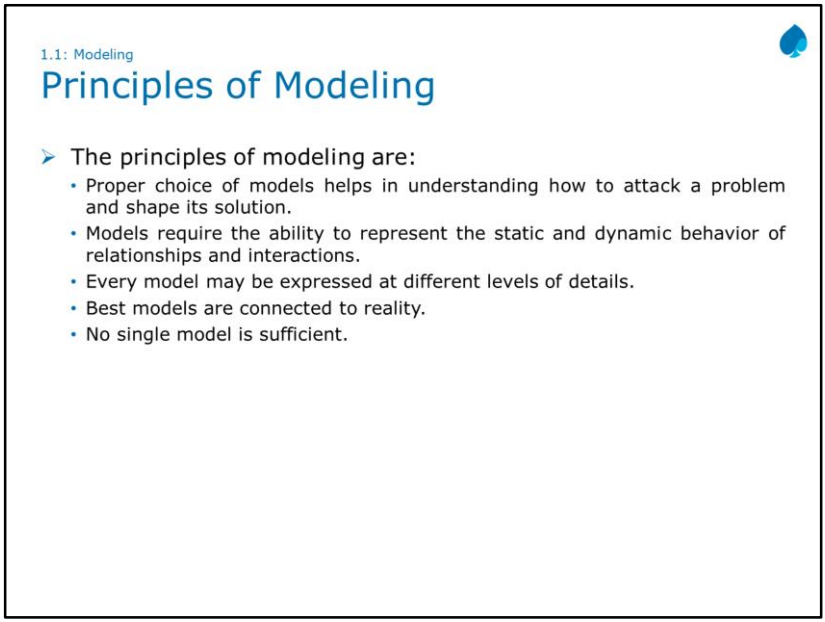
Models have two major aspects:

Semantic information (semantics)

Visual presentation (notation)

A model can tell what a function does (specification), and also how the function is accomplished (implementation).


1.1: Modeling
Features of Modeling
Modeling can be used:
to simplify complexity and understand working of system before it is actually built
to communicate the desired structure and behavior of the system
to visualize and control the system's architecture
to allow evaluation of all situations and expose opportunities for simplification and reuse
to manage risk
to document the decisions that are made



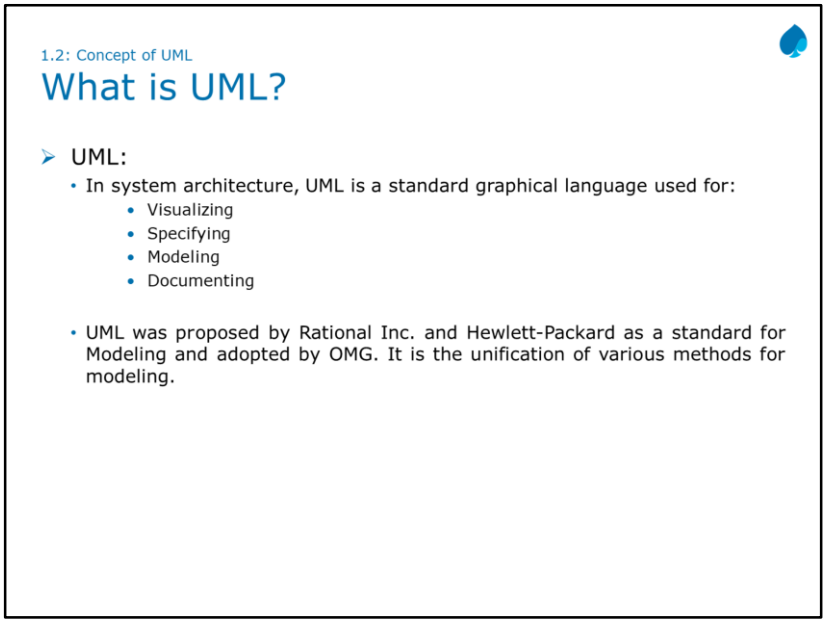
Principles of Modeling (contd.):

While modeling, the problem must always be kept in mind. Only the models that will add value to a “view” of the system must be considered. For example: If a system is supposed to be a stand alone system, having a model to depict the deployment details may not be required. However, such a model would be required for a system which is supposed to be deployed across a network.

Besides, every system has static as well as dynamic aspects. So the models must be capable of depicting the same.

Further, based on the view and the people it is intended for, models may be at different granular levels. Each user may require different degrees of details.

It is unlikely that one model gives the complete system description. This is where multiple models, each giving a different (but relevant) view of the system, becomes important. So proper choice of models is important, which will best help in understanding how to attack the problem and shape its solution.



What is UML?

OMG specification states: “The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system”. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions, as well as concrete things such as programming language statements, database schemas, and reusable software components.

UML is used for the following tasks:

Visualizing - Visual Model helps better communication and goes beyond what can otherwise be textually described.

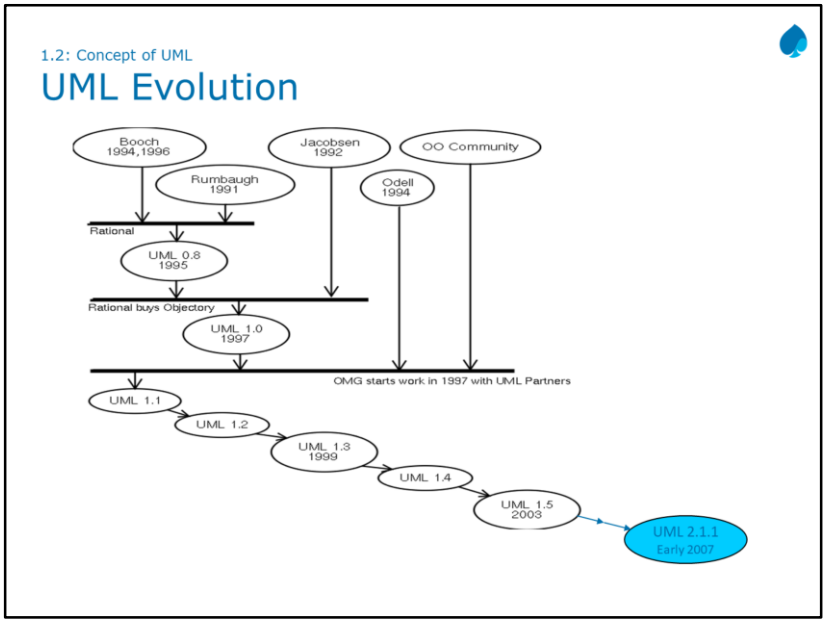
Specifying - UML can help in specifying all important analysis, design, and implementation decisions.

Modeling - Allows for construction of the system from the various models.

Documenting - Models can help in documenting all decisions taken during the entire system development lifecycle.

Prior to UML, there were many methods with similar modeling languages having minor differences in overall expressive power. However, there was no single “leading” modeling language. Lack of disagreement on a general-purpose modeling language discouraged new users from adopting the OO approach.

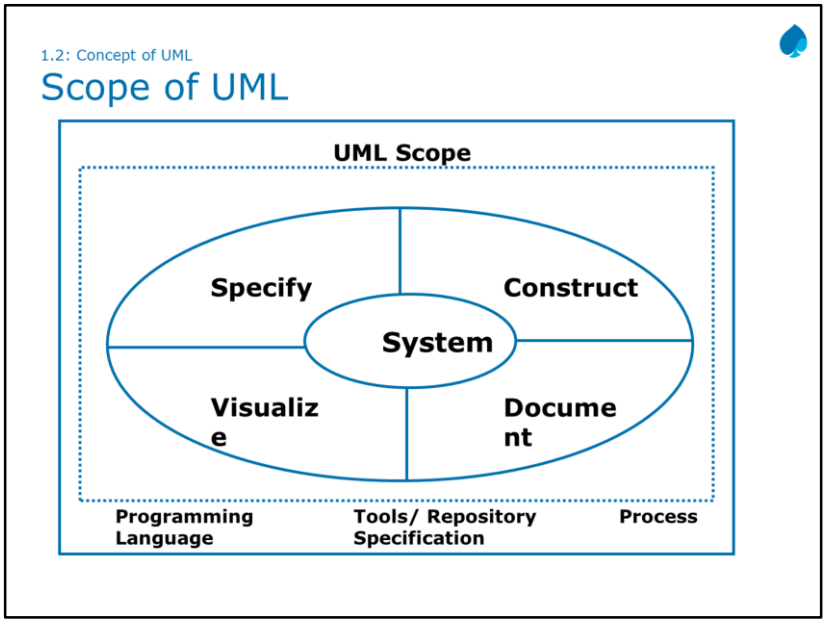
contd.



Here is a figure illustrating how UML has evolved over a period of time. Ver 2.x

is the current industry standard. We are dealing with Ver 1.4 in our current

course.



Scope of UML:

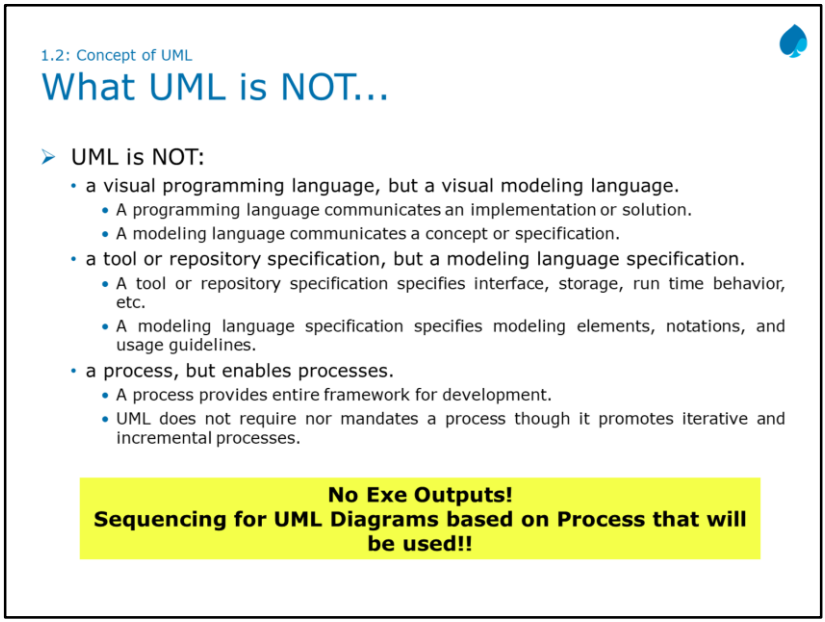
The figure shown in the slide illustrates what is within the scope of UML, and what is outside of the UML scope. While it provides adequate notation and semantics to address contemporary modeling issues, there are some items outside the scope of UML as explained in subsequent slides.

Some of the things that are outside the scope of UML are:

Programming Language: UML is a “visual modeling language”. It is not intended to be a visual programming language. It is a language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system. It does have a close mapping with OO languages.

Tools: Language standards form the foundation for tools and process. UML defines a semantic meta-model, and not a tool interface, storage, or run-time model.

Process: Software development process will use UML as a common language for its project artifacts, but will use the same type of UML diagram in the context of different processes. UML is process independent.



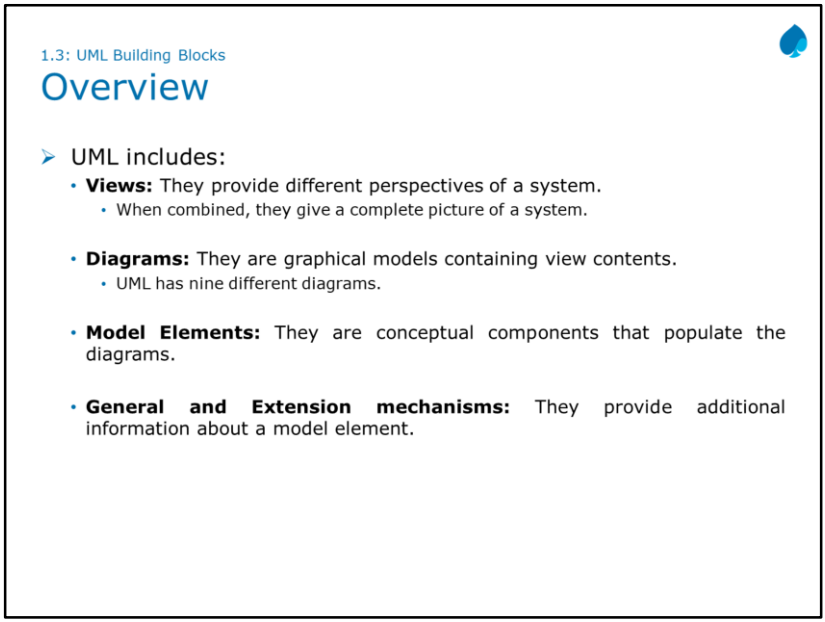
And What UML is NOT …

UML is not meant to be a programming language, rather it is a language meant for modeling. By using UML, one can convey a concept or a specification but not a solution (which a program does).

UML comprises model elements, each with its own associated notation and semantics. UML is not meant to specify a tool or repository in terms of interfaces, storage, or run time behavior.

Similarly, UML is not a process. A process will provide guidance regarding order of activities, and spell out the work products that have to be developed. They are usually domain specific.

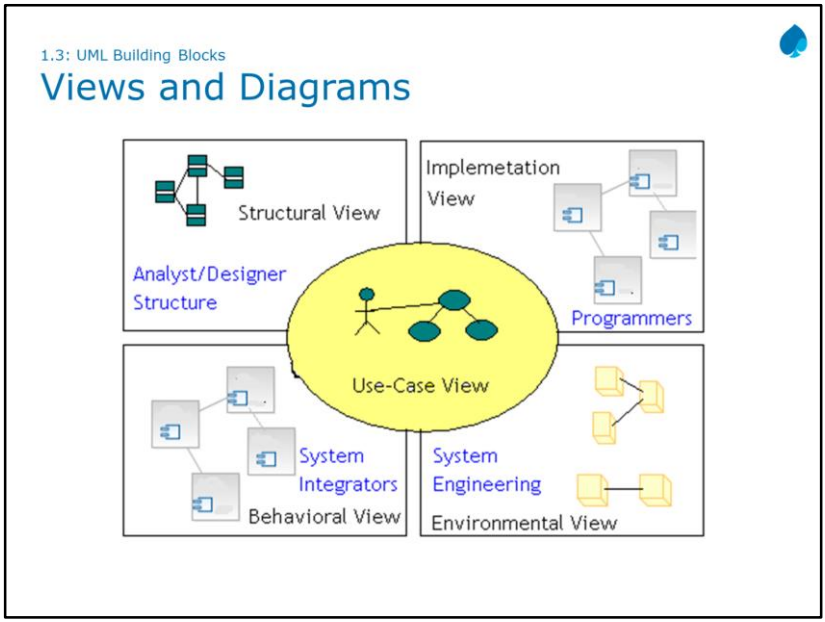
UML does not require a process. However, it enables and promotes Object-Oriented and component-based processes.



UML Building Blocks:

Let us look at the building blocks of UML. UML offers different views of the system, each view containing different diagrams. The diagrams are made up of specific modeling elements.

In addition to existing modeling elements, UML allows extending available notation and semantics by the use of extension mechanisms.



UML Building Blocks – Views and Diagrams:

For the end user, the User view is useful to understand the functionality that will be provided by the system. Various views are:

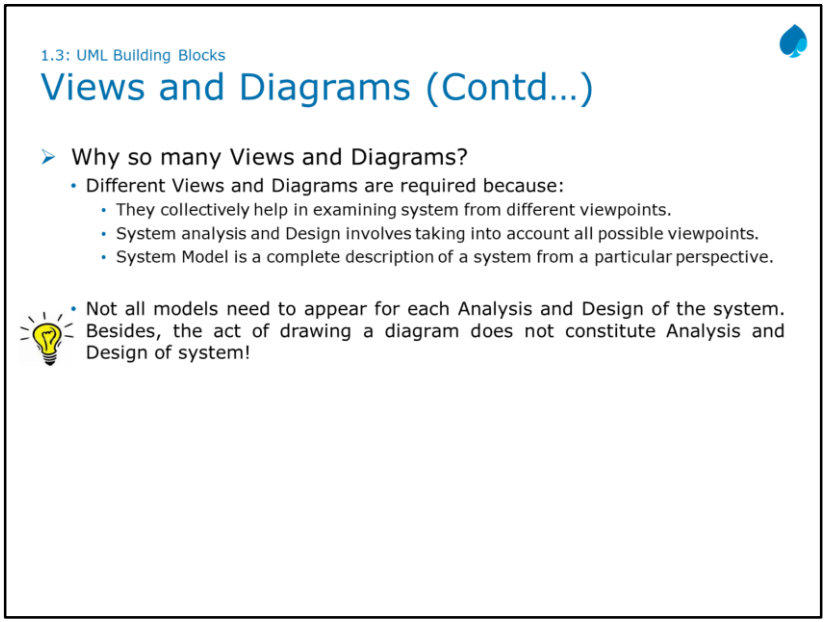
Structural view - Structure diagrams define the static architecture of a model. They are used to model the "things" that make up a model. They are used to model the relationships and dependencies. Analysts and Designers can get the view of the structural aspects of the system through the Structural view.

Behavioral view - It can give important inputs in terms of performance, scalability, and throughput, which can be used by the system integrator.

Implementation view - This view is helpful for programmers.

Environment view – This view can convey decisions relating to system topology, delivery mode, installation, and communication.

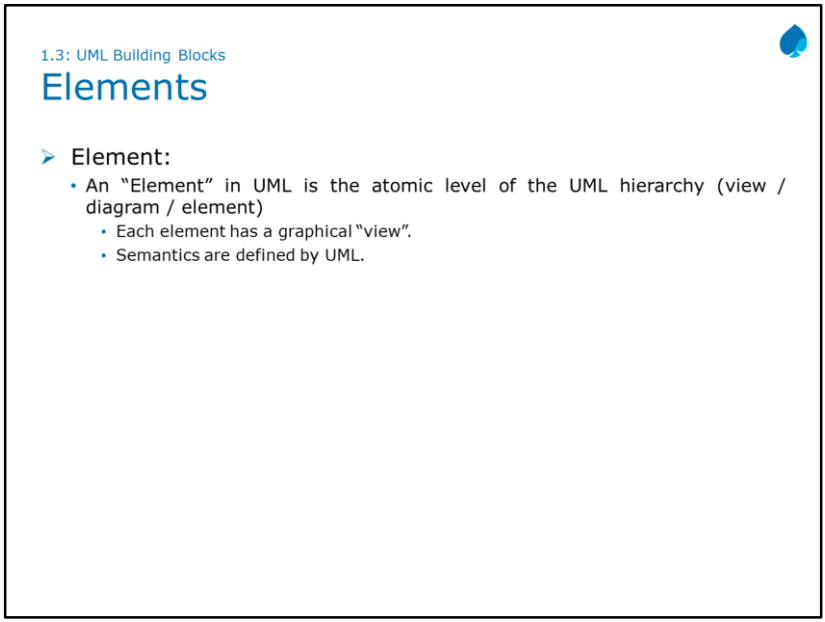
contd.



UML Building Blocks – Views and Diagrams (contd.):

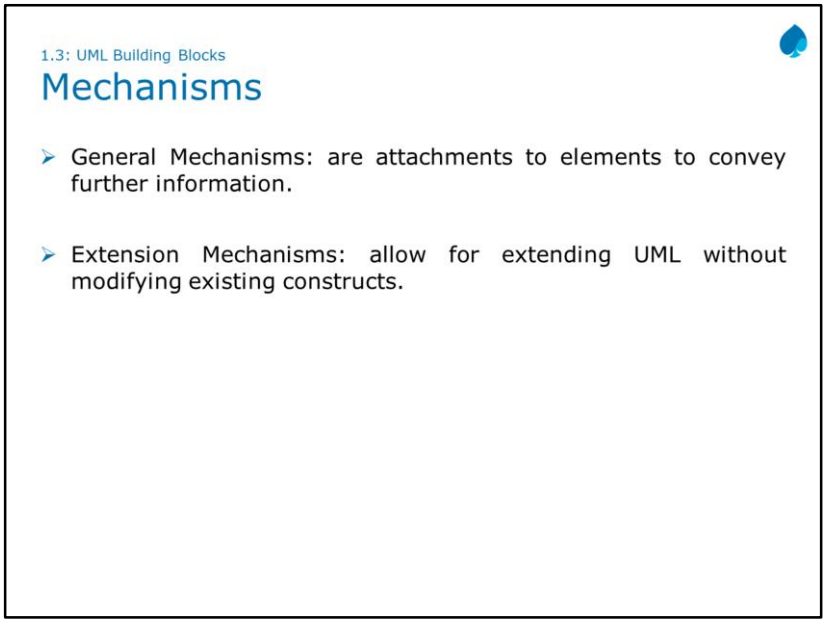
Often one has to decide which views / diagrams are required for the system under consideration. While deciding on this, consider the “reason for communication” of models. Depending on what aspects of the system need to be emphasized on, the views / diagrams can be chosen. (To that extent, each view / diagram is an independent entity in itself). Hence, it may not be required to have all models for each Analysis and Design.

It is important to note that the activity of drawing diagram by itself is not Analysis and Design. Rather, the diagrams are a means of representing and conveying the “Analysis and Design decisions”.



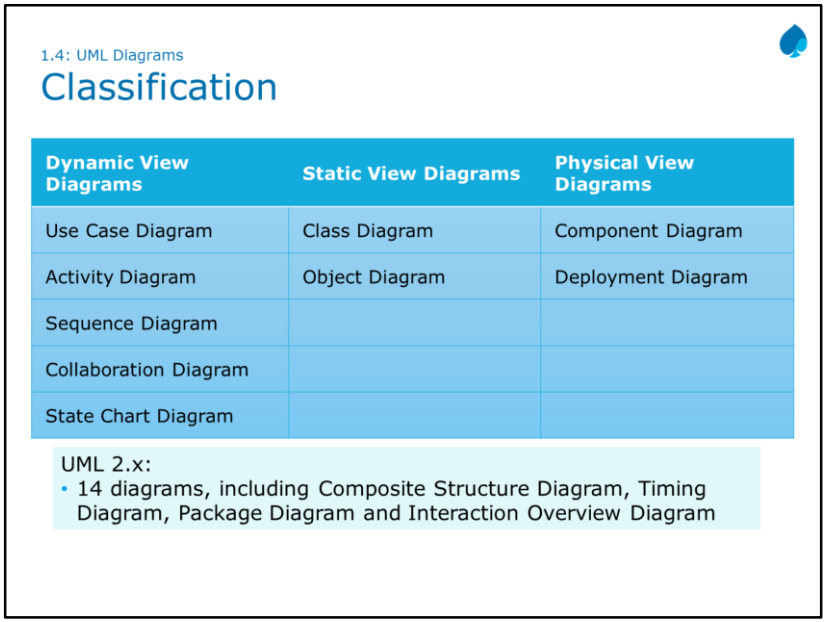
UML Building Blocks – Elements:

Elements form the atomic level of the UML hierarchy. Each element has a predefined “meaning” and a “graphical notation” associated with it.



UML Building Blocks – Mechanisms:

There are some mechanisms available to add on to the expressive power of UML. They are broadly categorized as General mechanisms and Extension mechanisms.



UML Diagrams:

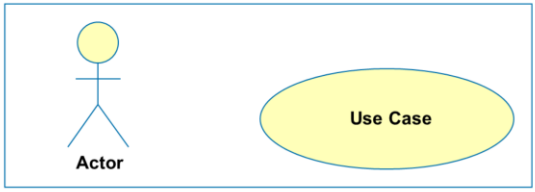
The slides shows a list of the nine diagrams in UML 1.4. Also mentions the additional diagrams of UML 2.x.

In the subsequent sections we will be looking at each UML 1.4 diagram in detail. The sections provide the notations and associated semantics for the constituents of each diagram.

1.5: Use Case Diagrams

Use Case Diagrams - Features

- Use Case Diagrams model the functionality of a system by
- using Actors and Use Cases:
  - Actor is a user of the system.
  - Use cases are services or functions provided by the system to its users.



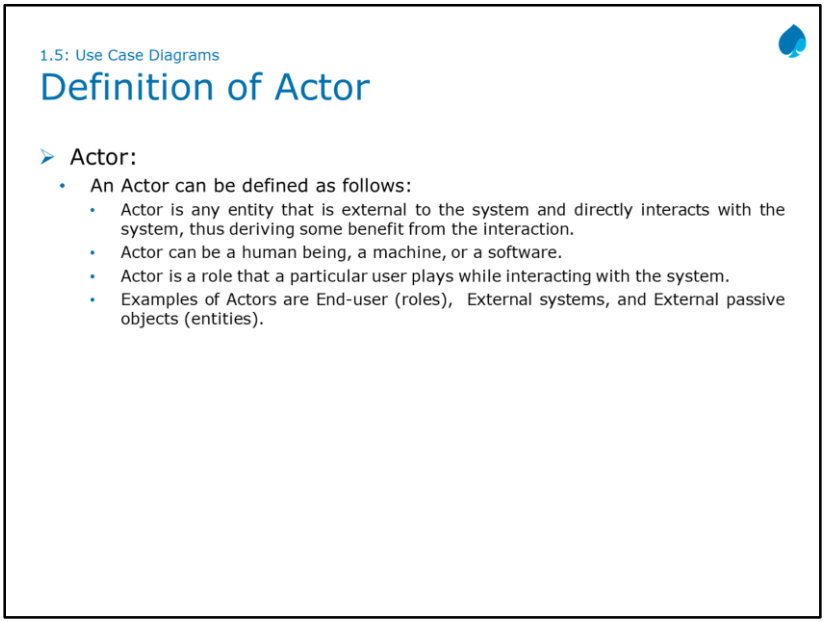
Use Case Diagrams:

Use Case is a description of a system's behavior from a user's point of view. It is a set of scenarios that describe an interaction between a user and a system. It also displays the relationship among Actors and Use Cases.

Two main components of Use Case diagram are Use Cases and Actors.

> Use case diagrams, which render the User View of the system, describe the functionality (Use Cases) provided by the system to its users (Actors).
>
> An Actor represents a user or another system that will interact with the system you are modeling.

An Use Case is an external view of a system that represents some action that the user might perform in order to complete a task.



Actor:

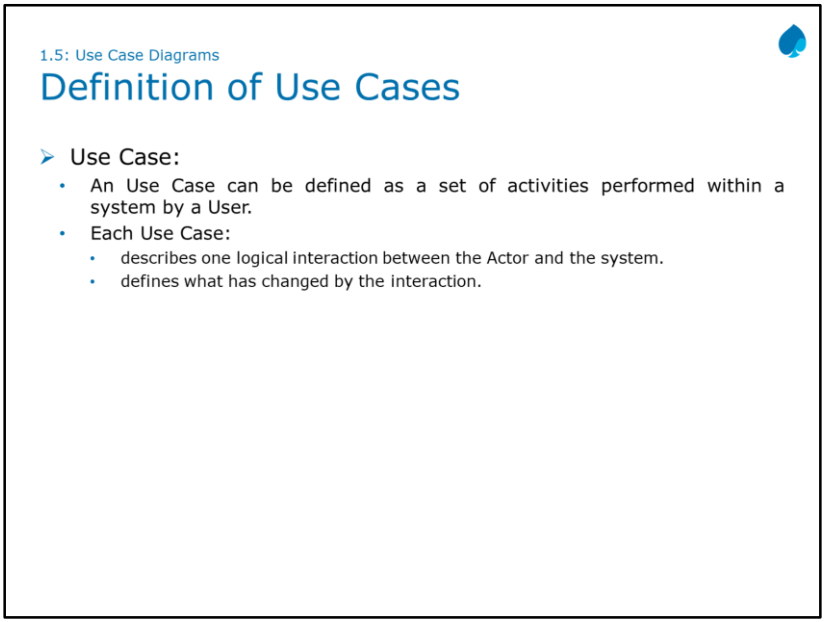
Actors are people, organizations, systems, or devices which use or interact with our system. The system exists to support that interaction. Therefore, the important part of the project is to identify the Actors and find out what they want from the system.

Actors are characterized by their external view rather than their internal structures. It is a role that the user plays to get something from the system.

Role and organization Actors only require logical interactions with the system. Ask who wants what from our system, rather than who operates the system.

For example: ABC and XYZ are users who wish to buy from an online store. For the online stores system, they play the role of a customer, and hence customer is the Actor for the system. The database for this system may already be existing, and hence this may be another Actor (note that user in this case is not a human).

The Actors will finally be used to describe classes, which will interact with other classes of the system.



Use Cases:

The Use Cases define “units of functionality” provided by system. They model “work units” that the system provides to its outside world.

A Use Case is one usage of the system. It is a generic description of a use of the system. It allows interactions in a specific sequence.

At the lowest level, they are nothing but methods which need to be implemented by various classes in the system.

Use Cases determines everything that the Actor wants to do with the system.

A Use Case performs the following functions:

Defines main tasks of the system

Reads, writes, and changes system information

Informs the system of real world changes

A Use Case needs to be updated / informed about system changes.

1.5: Use Case Diagrams

Drawing the Use Case Diagram

- A Use Case diagram has the following elements:
  - **Stick figure**: It represents an Actor.
  - **Oval**: It represents a Use Case.
  - **Association lines**: It represents communication between Actors and Use Cases.

Drawing the Use Case Diagram:

The Use Case Diagram has the following elements:

A stick figure, which represents Actors (sometimes stereotyped classes, as explained later, are also used to represent Actors). They differ from tool to tool.

Ovals or ellipses, which represent Use Cases

Association lines, which indicate interactions between Actors and Use Cases.

Use Cases will have description of what the Use Case is supposed to do when it is used.

An example of use case description is given.

1.5: Use Case Diagrams

Use Case Specification

- Each Use Case would have a Use Case Specification associate with it
- No standard template used by UML for this, but typical formats would include
  - Name and Brief Description
  - Invoking Actor
  - Flow of Events including Basic and Alternate Flows
  - Special Requirements
  - Pre-Conditions, Post Conditions & Extension Points

See the notes pages for an example of Use Case Specification (or Description Document)

Example:

Use Case Specification: To make a Reservation

This use case describes how a reservation is made in the Airlines Reservation System.

Invoking Actor:

Customer (passenger)

Flow of Events:

Basic Flow:

The use case begins when a customer wishes to make a reservation.

The system displays a reservation form.

The customer enters the reservation details. Reservation details include:

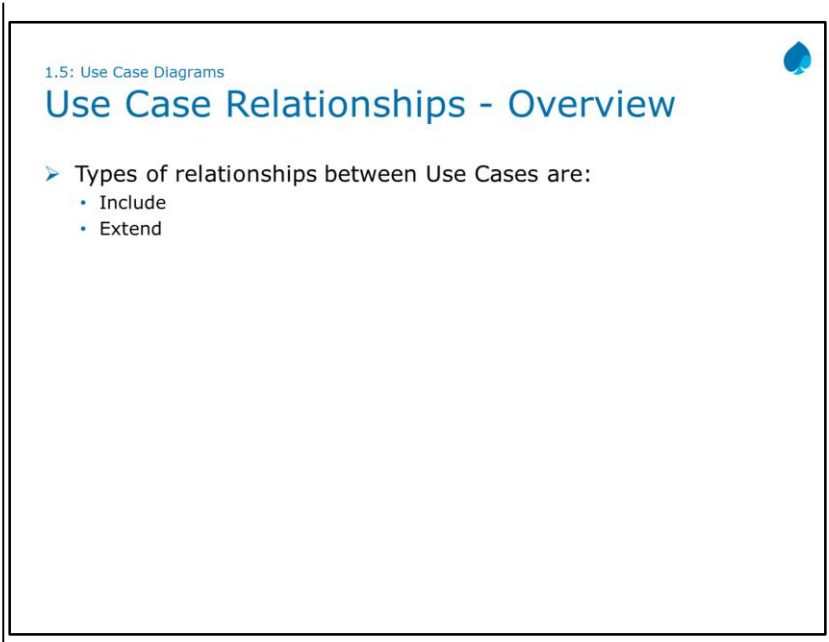
Round Trip or One Way, Origin, Destination, Departure and Arrival Dates, Number of Passengers (Adults and Children).

The system retrieves Flight Details based on reservation request.

The customer selects the desired flight.

The system calculates and displays the total fares applicable inclusive of taxes.

The system requests passenger details (First and Last Name) and Contact Information (Phone No., Email Id and Mobile No.)



Use Case Relationships:

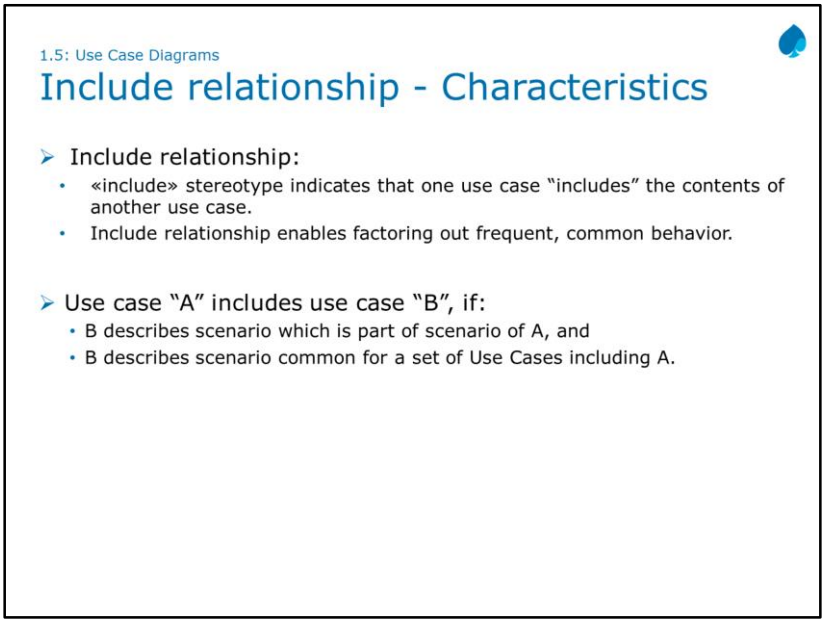
Relationships help us connect the model elements.

After finding out the primary Use Cases, one can start looking “into” the system to see if there are any relationships between the Use Cases.

The following types of relationships can exist between the Use Cases:

include

extend

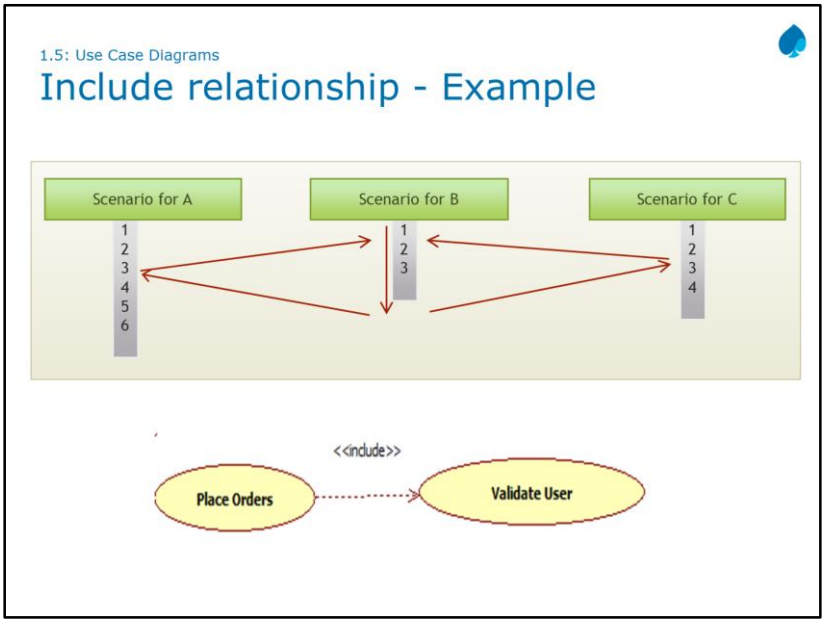


Use Case Relationship – Include:

In an Include relationship, one Use Case includes behavior specified by another Use Case. If there are common steps in the scenarios of many Use Cases, they can be factored out into a separate Use Case. This Use Case can then be included as part of the "Primary Use Case".

The above arrangement helps us segregate and organize common sub-tasks. An "Included Use case" is not a complete process. Extra behavior is added to the base Use Case.

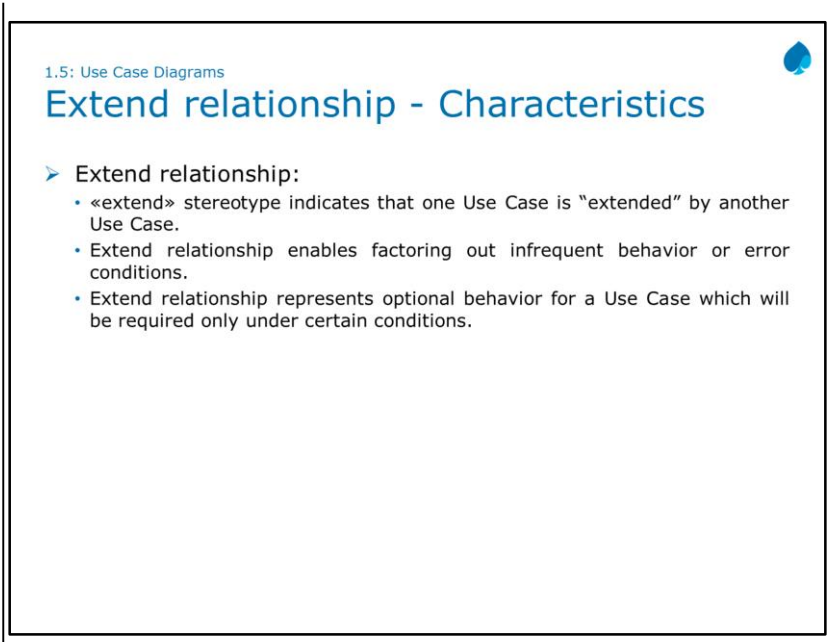
This may also be used in case of complex Use Cases (where there is too much functionality in one Use Case). In such cases, the primary functionality can be distributed across Use Cases, and a primary Use Case can then include the remaining secondary Use Cases.



Use Case Relationships – Include (contd.):

As illustrated in the figure shown in the slide, scenario of Use Case B is required by Use Case A and Use Case C. Hence both Use Cases A and C can include Use Case B.

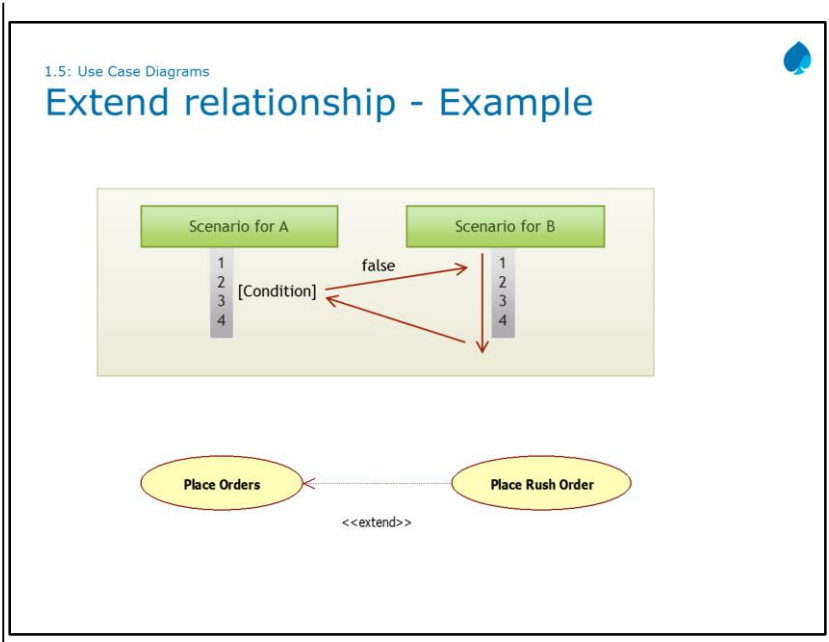
After completing the scenario of Included Use Case B, the Use Cases A and C will continue with their respective scenarios.



Use Case Relationships - Extend :

In an Extend relationship, an Use Case may be required by another use case based on “some condition”, or due to “some exceptional situation”.

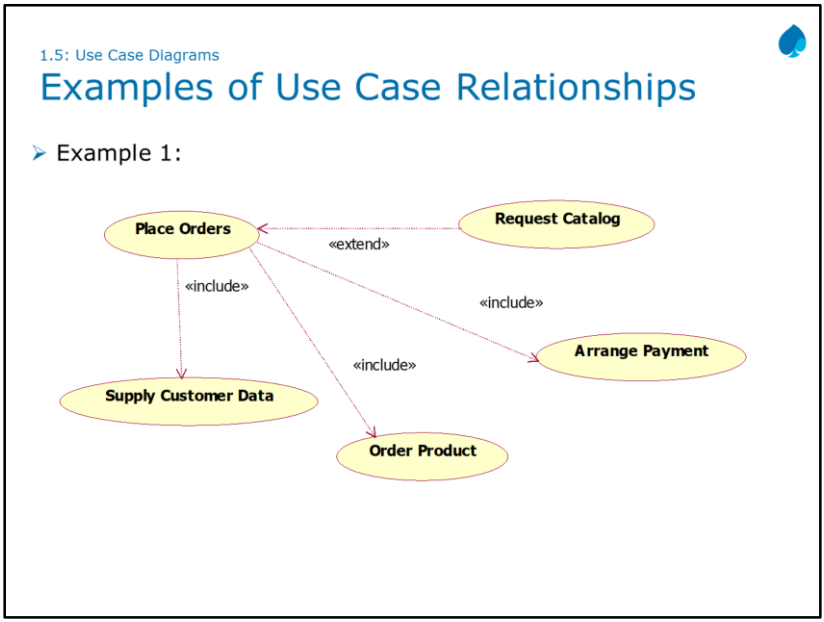
Again, upon completion of extension activity sequence, the original Use Case will continue.



Use Case Relationships – Exclude (contd.):

As illustrated in the figure shown in the slide, in Use Case A when the condition becomes false, the scenario of Use Case B is invoked.

Note that Use Case B is said to extend Use Case A. The stereotyped generalization arrow with keyword “extend” depicts the extend relationship between the Use Cases.



Example of Use Case Diagram:

The slide shows an example where we are looking at the Use Case relationships (though the Actors and system boundary has not been shown here, let us assume that they exist. They have been left out so as to focus on the relationships).

The Request for Catalog may not always happen when an Order needs to be placed. Hence, Extend is the relationship used between the two Use Cases of “Place Order” and “Request Catalog”.

“Place Order” being a complex Use Case, it is broken down into secondary use cases of:

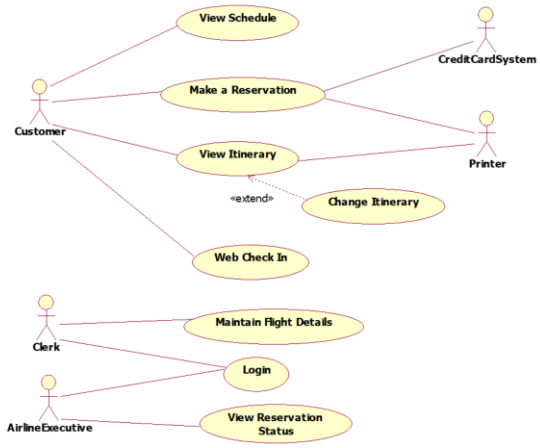
- Supply Customer data
- Order Product
- Arrange Payment

It is important to note that the diagram by itself does not indicate any kind of ordering (i.e. first invoke order product, and then arrange payment, etc). The ordering has to be taken care of as part of the Use Case scenario.

1.5: Use Case Diagrams

Examples of Use Case Relationships Contd...)

Example 2:

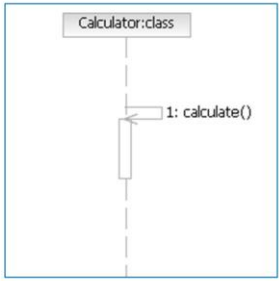


How do you interpret this diagram?

1.6: Sequence Diagrams

Direction of Arrows

- Direction of Arrows:
  - Direction indicates which object's method is being called by whom.
  - A circulating arrow on the Object Lifeline is for a self method - called within the object by itself.



Sequence Diagram: Notations (Directions and Branches):
Direction of arrows:

Every message has a "sender" and "receiver" and this is depicted by the direction of the arrow head. Control gets passed from the sender to receiver.

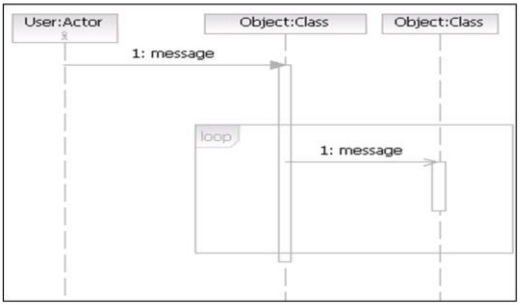
"Implicit returns" are assumed in case there are no return messages to the sender.

It is possible that there is a "call to object's own method". A "circulating arrow" is used to depict such a case. In the example shown on the slide, Calculator object calls its own method calculate().

1.6: Sequence Diagrams

Branch Conditions

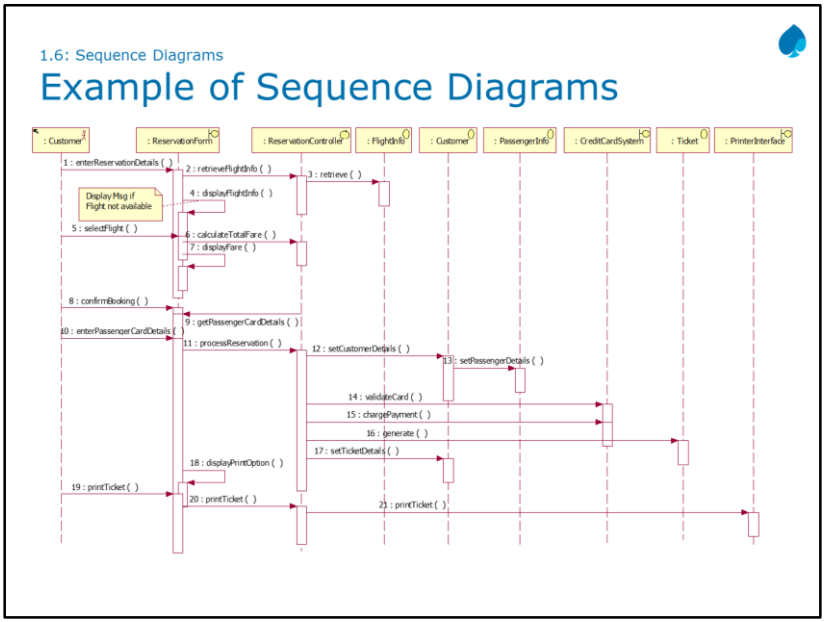
- Branch Conditions:
  - Branch Conditions are depicted as "Guard Conditions" within Square Brackets.
  - Repetition or Looping depicted as a rectangle, with condition for exiting the loop placed at the bottom left corner.



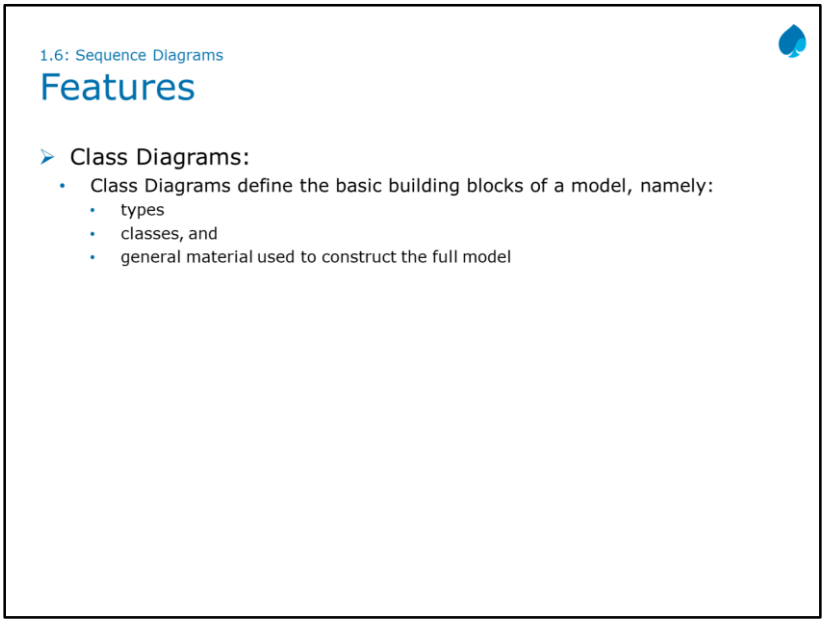
Sequence Diagram: Notations (Directions and Branches):
Branch Conditions:

Branching involves multiple messages originating at same time to different objects, based on some condition called as the "Guard Conditions". These Guard Conditions are given in square brackets.

Iteration involves sets of messages sent multiple times. A rectangle enclosing the set of messages indicates looping.



How do you interpret this diagram?



Class Diagrams: Features:

Class Diagrams can be used to model classes, and the relationships between classes.

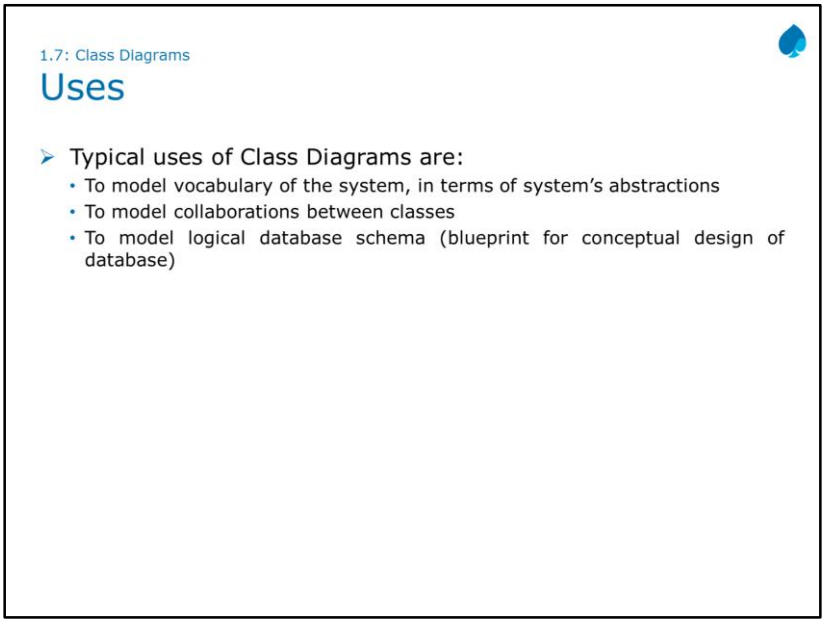
When drawn during the analysis stage, only the names of the classes maybe represented.

During further refinements in the detailed analysis or design stage, details like “attributes” and “services” get added to each class, and are depicted in the Class Diagram.

1.6: Sequence Diagrams

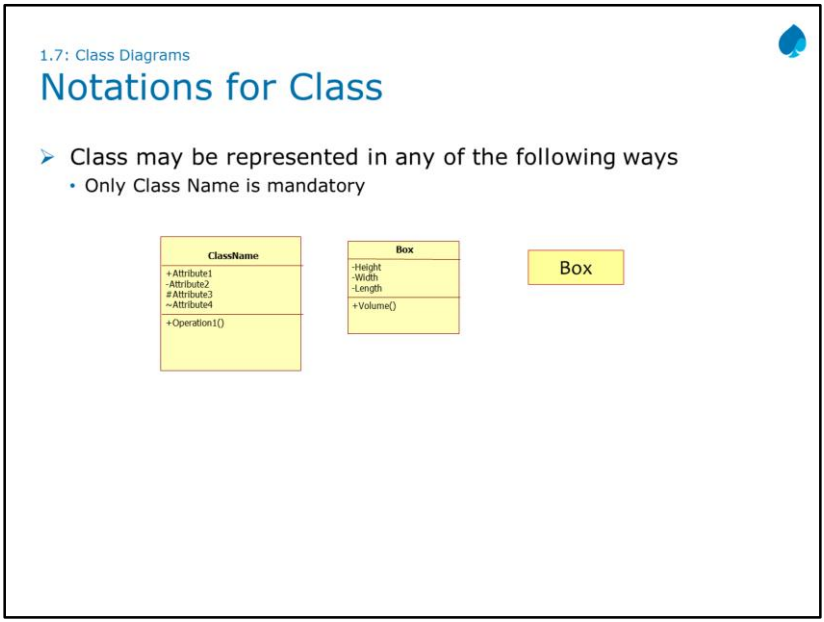
Functions

- Class Diagrams have the following functions:
  - They describe the static structure of a system.
  - They show the existence of classes and their relationships.
  - Classes represent an abstraction of entities with common characteristics.
  - Relationships may be:
    - Generalization
    - Association
    - Aggregation
    - Composition, or
    - Dependency



Uses of Class Diagram:

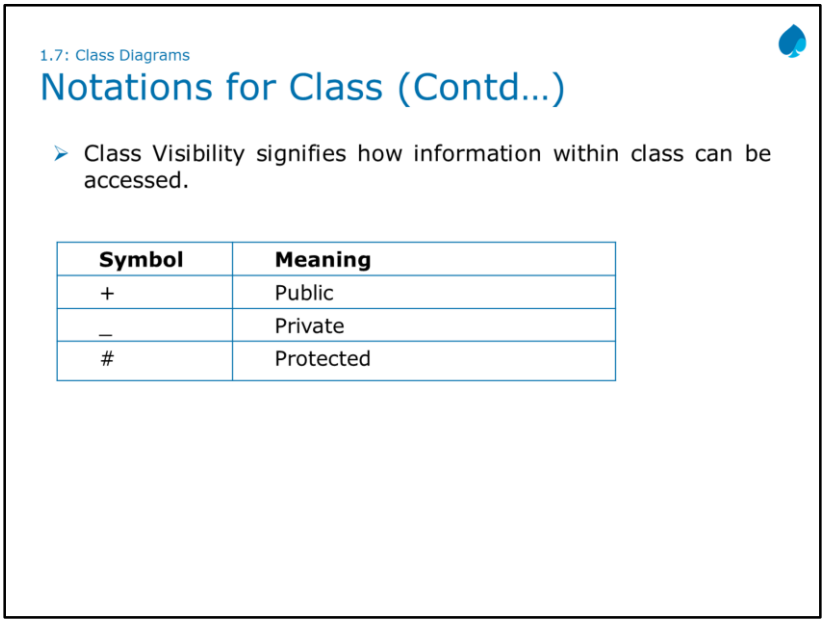
The importance of the Class diagram is that it gives a view of all the classes that are required to make up the system. It also conveys the collaborations that exist between classes to give the system behavior.



Notations for Class:

Classes are denoted as rectangles, with compartments for name, attributes, and operations. There is optionally a last compartment that can be used for specifying responsibilities, variations, business rules, etc.

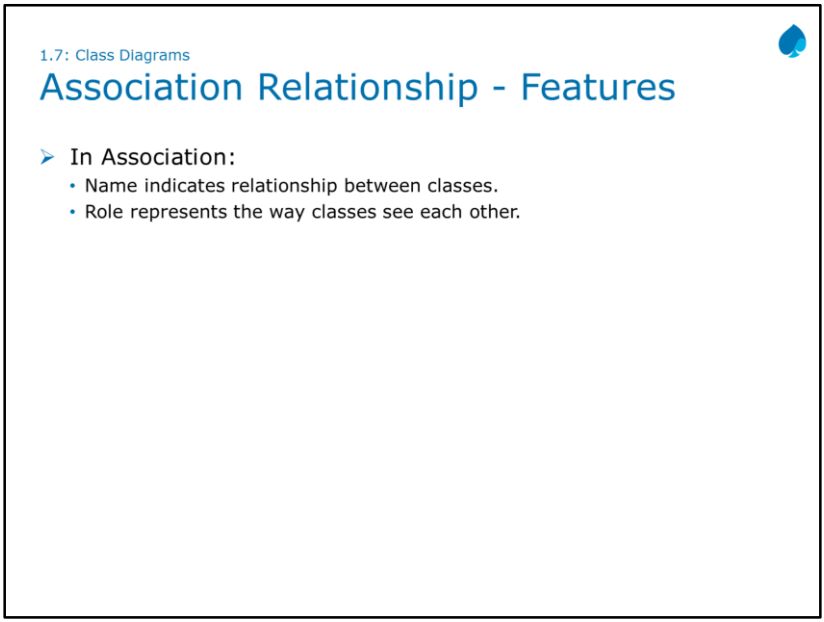
The name is the mandatory part. Other compartments may be included based on the amount of details required to be communicated. The representations of classes that do not have all compartments are known as “elided notations for class”.



Notations for Class: Class Visibility:

Information about visibility of attributes and operations can sometimes be represented by using symbols like + for public, - for private, or # for protected.

These symbols may vary from tool to tool.



Relationships: Association:

Associations may be characterized by the following:

Name: The name signifies purpose of association, and is written along with the line indicating association, role and direction of association.

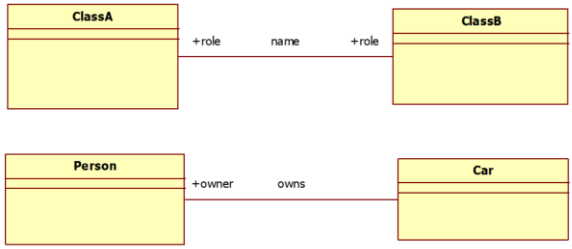
Role: In case there are specific roles played by classes in the association, then it is indicated by the role name, which is written near the class.

Arrow: Arrows may be used to indicate whether the association is uni-directional or bi-directional. Absence of arrows implies that no inferences can be drawn about the navigability.

The example in the slide shows an association relationship between a class Person and a class Car. The class Person plays the role of an owner.

1.7: Class Diagrams

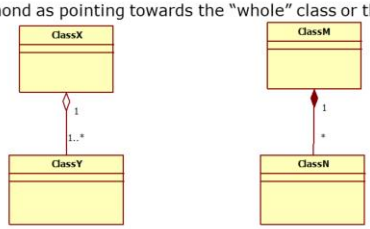
Association Relationship - Example



1.7: Class Diagrams

Relationships - Features

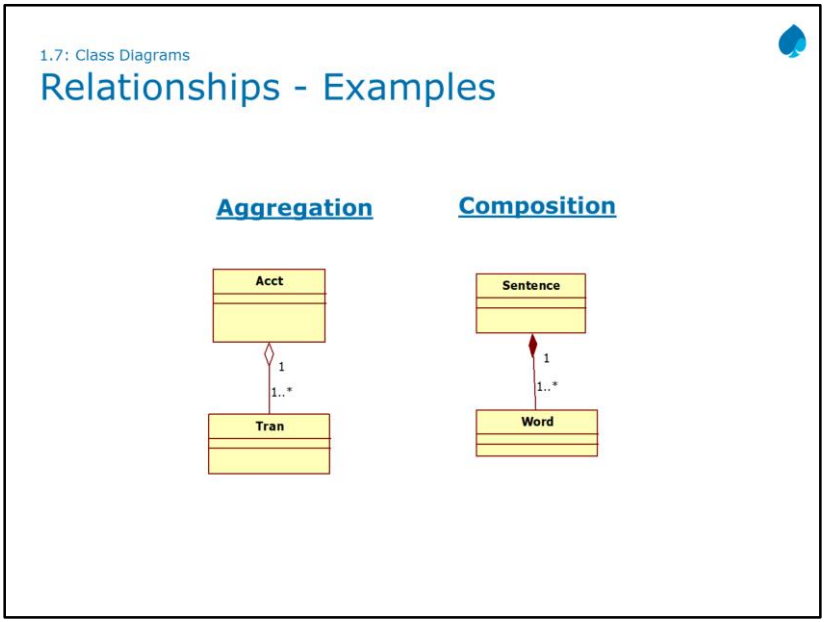
- Aggregation and Composition:
  - The following Class Diagram, possessing Composition and Aggregation, displays:
    - Aggregation as indicated by a hollow diamond.
    - Composition as indicated by a filled diamond.
    - Diamond as pointing towards the "whole" class or the aggregate.



Relationships: Aggregation and Composition:

Composition and Aggregation are modeled with filled diamond and hollow diamond, respectively, on the "Whole" part.

Roles and multiplicity, if required, can be mentioned here, as well. Typically they are done for the "Part" part of the "Whole" part.



Examples of Aggregation and Composition:

In the examples shown in the slide,

the relationship between a Sentence and a Word is represented as a “Composition” (Word is a part of a sentence).

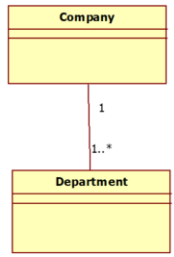
the relationship between Account and Transaction is represented as an “Aggregation”.

1.7: Class Diagrams

Definition of Multiplicity

- Multiplicity:
  - Multiplicity indicates the "number of instances" of one class linked to "one instance" of another class.

Symbol	Meaning
1	Exactly one
0..1	Zero or one
*	Many
0..*	Zero to many
1..*	One to many

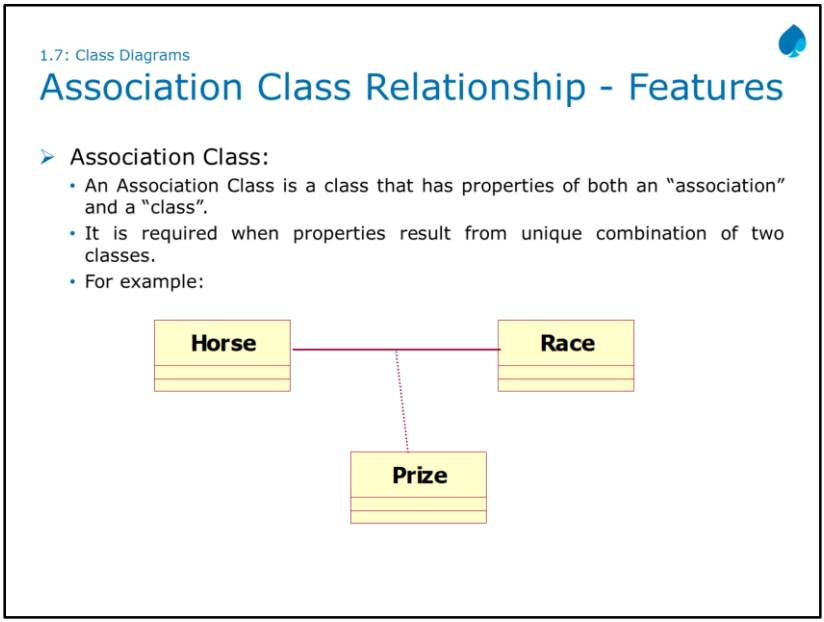


Multiplicity:

Multiplicity attached to a class denotes the possible cardinalities of objects of the association.

For example: The above figure depicts that "One company has one or more departments, and a department is associated with one company".

Multiplicity values can be indicated in association, aggregation, and composition relationships.



Association class:

An Association class is a class required as the result of association between two classes.

For example:

The Prize class is a result of association between the Horse class and the Race class.

For each Horse placed in a Race there is a prize.

The amount of prize depends on the race.

The Prize class could not be associated with the Horse class alone because a Horse might have many Prizes, and the relationship between the Prize and Race would be lost.

Similarly, Prize class cannot be associated with Race class alone because a Race has many Prizes, and the relationship between the Prize and the Horse would be lost.

Similarly, result of a student (in terms of marks in assignments, test, and grade) in a course is a unique combination of an individual student, and a particular course. So we can have an association between Student and Course Classes, with Result being an Association Class.

1.7: Class Diagrams

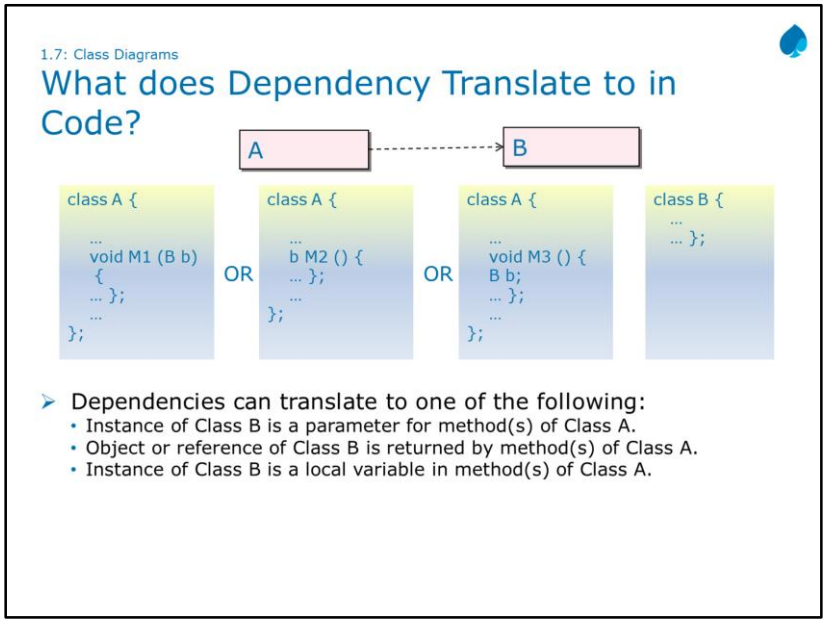
Dependency - Features

- Dependency:
  - Dependency is a "using" relationship within which the change in the specification of one class may affect another class that uses it.
  - For example:

Dependency:

The dependencies are denoted as dashed arrows with arrow head pointing to the independent element.

In the example shown in the slide, the structure and behavior of the Window Class is dependent on the structure and behavior of the Event Class.



UML Relationships: What does Dependency translate to in code?

In a dependency relationship, an instance of the independent class (B) will be used in the dependent class (A) in one of the following manners:

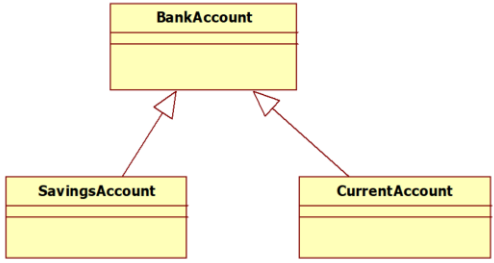
- Instance of B can a parameter to one or more methods of Class A.
- Instance of B can be returned by one or more methods of Class A.
- Instance of B can be a local variable in one or more methods of Class A.

Note that dependency is non-structural, that is, instance of Class B does not come as an attribute of Class A and hence not part of the structure of Class A. Class A is aware of existence of Class B only when the concerned method is called. The relationship is temporary in nature too…once the method invocation is completed, Class A need not maintain information about Class B.

1.7: Class Diagrams

Generalization - Features

- Generalization:
  - Generalization indicates relationships between super-class and sub-class.
  - For example:

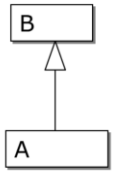


Relationships: Generalization:

Generalizations are denoted as “paths” from specific elements to generic elements, with a hollow triangle pointing to the more general elements.

1.7: Class Diagrams

What does Generalization Translate to in Code?



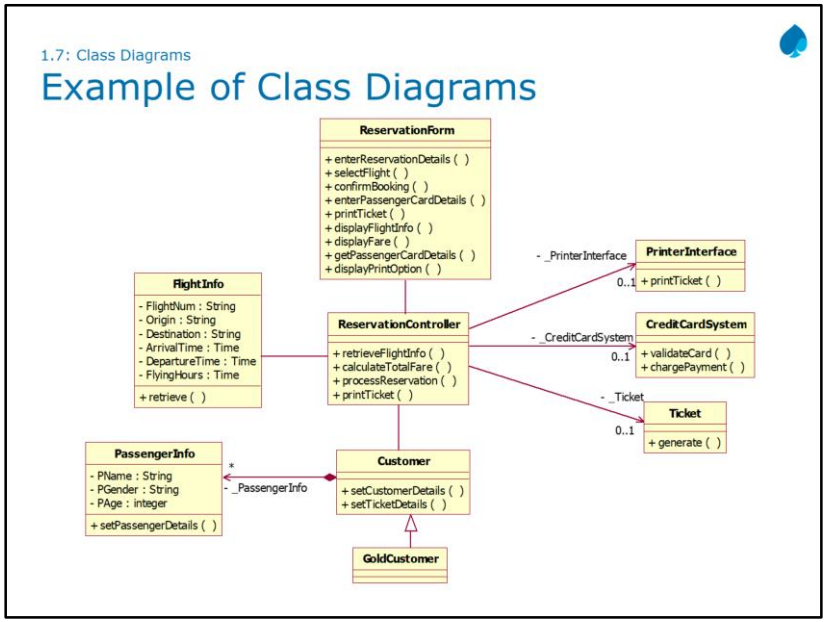

```
class B {
  ...
  ...
... };
```

```
class A extends B {
  ...
  ...
... };
```

- Generalization entails using the language constructs to implement inheritance relationship.

UML Relationships: What does Generalization translate to in code?

Object oriented languages provide language constructs for implementing inheritance relationship. This will be used for coding generalization.



How do you interpret this diagram?

Lab Book

- UML Lab 1,2,3

Summary

- In this lesson, you have learnt:
  - Class Diagrams, use case diagrams and sequence diagram

Answers to review questions.
Question 1: A,B

Question 2: Generalization, Association, Aggregation, Composition, Dependency

Review Question

- Question 1: A Class Diagram gives information about:
  - A. Attributed defined for a class
  - B. Operations defined for a class
  - C. Logic to be used for an operation of a class
- Question 2: Relationships that you may find on a Class Diagram are ___, ___, ___, ___ and ___.

Answers to review questions.

Question 1:
1- A, D,E
2- C
3-B,F

Review Question: Match the Following

1. Dynamic View Diagrams	A. Use Case Diagram
2. Static View Diagrams	B. Deployment Diagram
3. Physical View Diagrams	C. Class Diagram
	D. Activity Diagram
	E. Sequence Diagram
	F. Component Diagram

Answers to review questions.

Question 1: Unified Modeling Language.

Question 2 : True

Question 3: False

Review Question

- Question 1: UML Stands for ___.
- Question 2: UML offers an approach to capture different views of the system.
  - Option: True / False
- Question 3: UML describes a sequence in which diagrams must be drawn.
  - Option: True / False