# Welcome to
# Software Engineering (IT314)

# Lab 7
# UML Diagrams

Jash Rathi, Mahir Shah

Course Instructor: Prof. Jayprakash Lalchandani

Dhirubhai Ambani Institute of Information and Communication Technology

DA-IICT

# OUTLINE

- Types of UML diagram
- Class diagram
- Component diagram
- Deployment diagram
- Package diagram
- Object diagram
- Profile diagram
- Composite Structure diagram

# UML Diagrams

- UML stands for Unified Modeling Language
- It’s a rich language to model software solutions, application structures, system behavior and business processes
- These diagrams are further classified as:
  - Structure Diagrams
  - Behavioral Diagrams

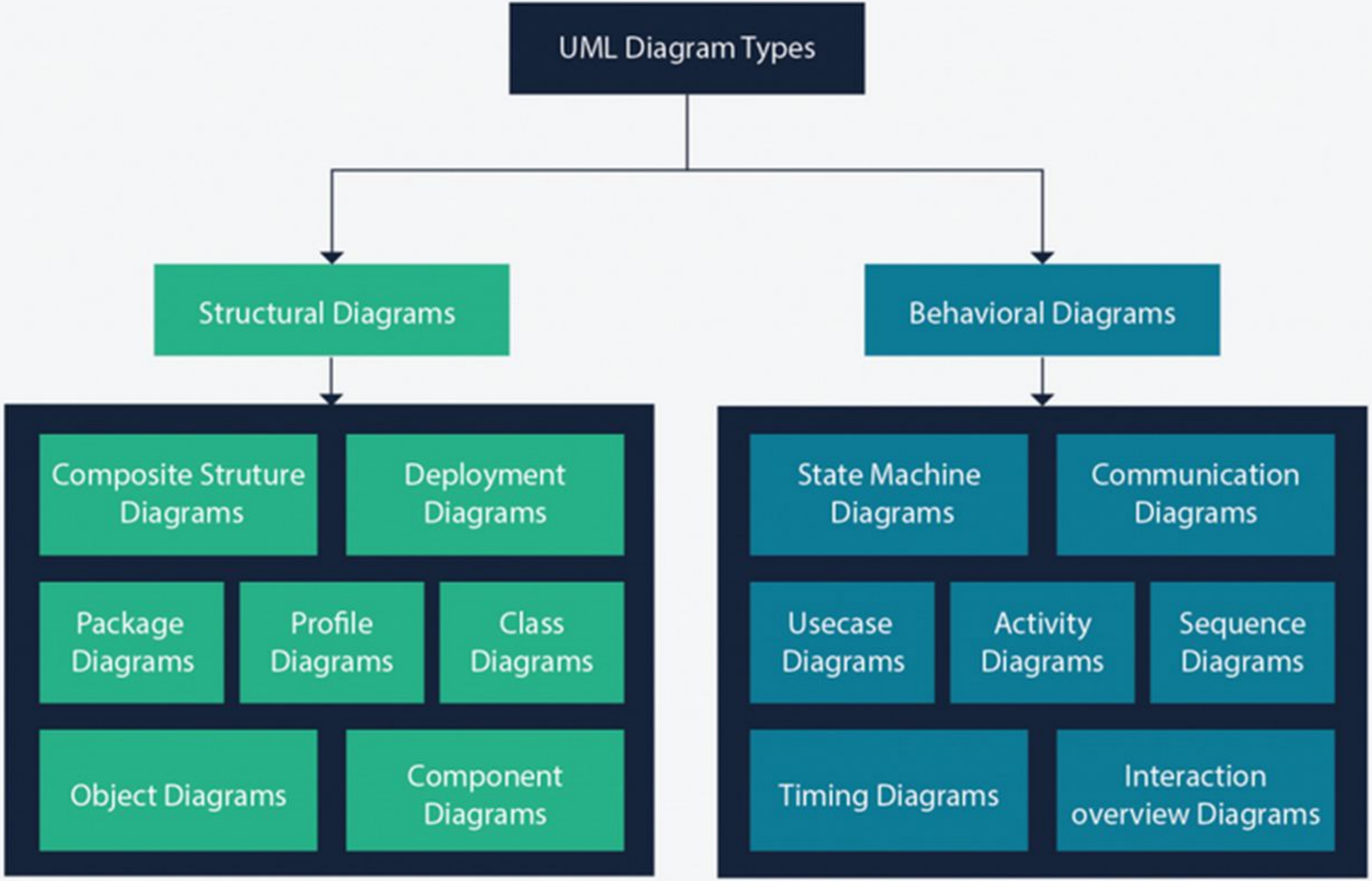
# Types of UML Diagrams I

## Structure Diagrams

- Structure diagrams show the things in the modeled system
- In a more technical term, it shows different objects in a system

## Behavioral Diagrams

- Behavioral diagrams show what should happen in a system
- It describe how the objects interact with each other to create a functioning system



UML Diagram Types

Structural Diagrams

Composite Struture Diagrams

Deployment Diagrams

Package Diagrams

Profile Diagrams

Class Diagrams

Object Diagrams

Component Diagrams

Behavioral Diagrams

State Machine Diagrams

Communication Diagrams

Usecase Diagrams

Activity Diagrams

Sequence Diagrams

Timing Diagrams

Interaction overview Diagrams

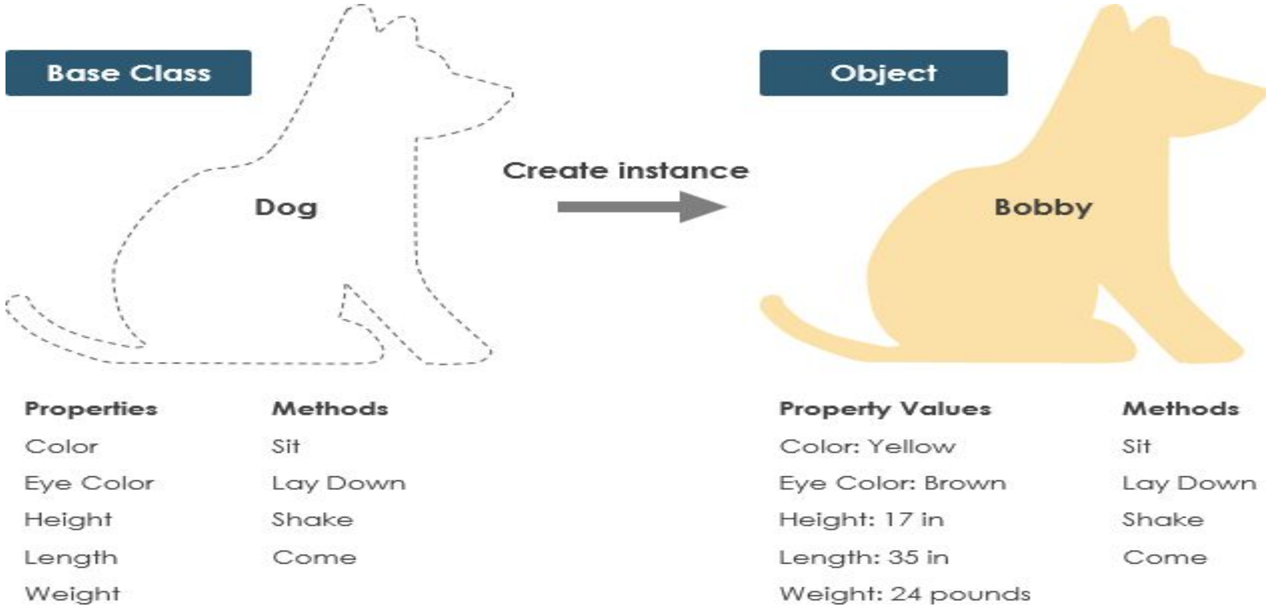
# Class Diagram

# What is a Class?

- A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without  talking about the other.
- The entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects.
- So a class describes what an object will be, but it isn't the object itself.
- In fact, classes describe the type of objects, while objects are usable instances of classes.
- Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods).
- The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

# Example

A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

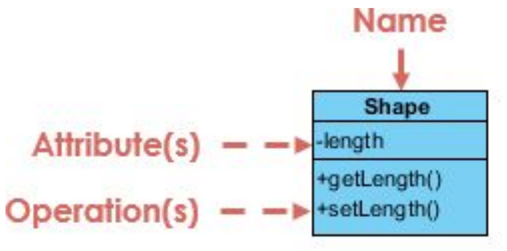


| Properties | Methods |
|---|---|
| Color | Sit |
| Eye Color | Lay Down |
| Height | Shake |
| Length | Come |
| Weight | |

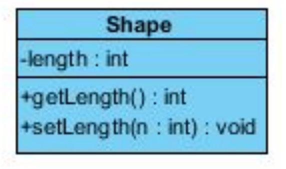
| Property Values | Methods |
|---|---|
| Color: Yellow | Sit |
| Eye Color: Brown | Lay Down |
| Height: 17 in | Shake |
| Length: 35 in | Come |
| Weight: 24 pounds | |

# UML Class Notation

- A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory information***.



Class without signature



Class with signature

# Class components

**Class Name:**

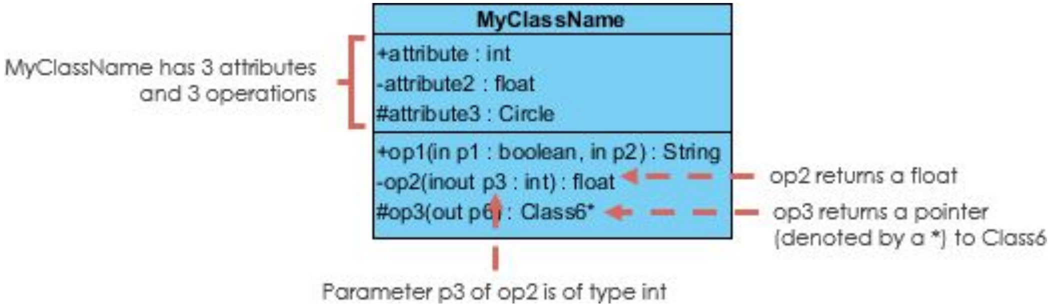
- The name of the class appears in the first partition.

**Class Attributes:**

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

# Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



MyClassName has 3 attributes and 3 operations

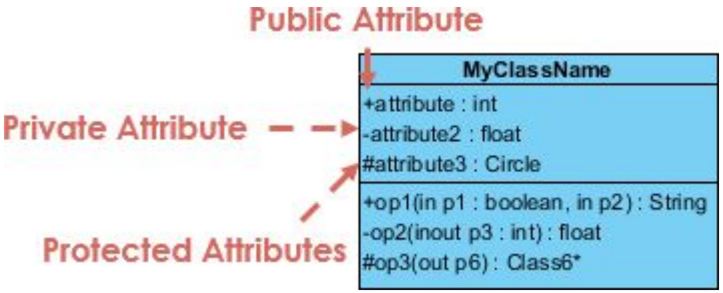
op2 returns a float

op3 returns a pointer (denoted by a *) to Class6

Parameter p3 of op2 is of type int

# Class Visibility

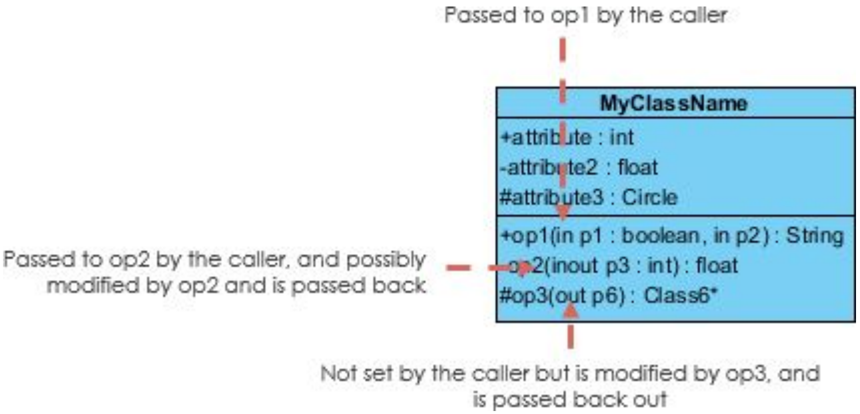
The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

# Parameter Directionality

- Each parameter in an operation (method) may be denoted as in, **out** or **inout** which specifies its direction with respect to the caller. This directionality is shown before the parameter name.



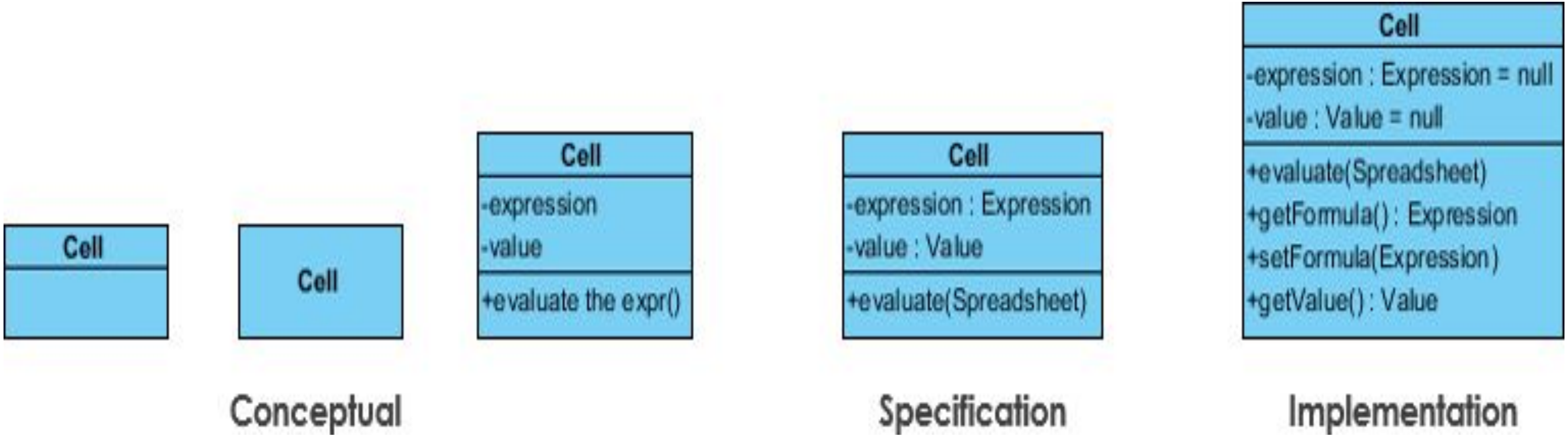
Passed to op1 by the caller

Passed to op2 by the caller, and possibly modified by op2 and is passed back

Not set by the caller but is modified by op3, and is passed back out

# Perspectives of Class Diagram

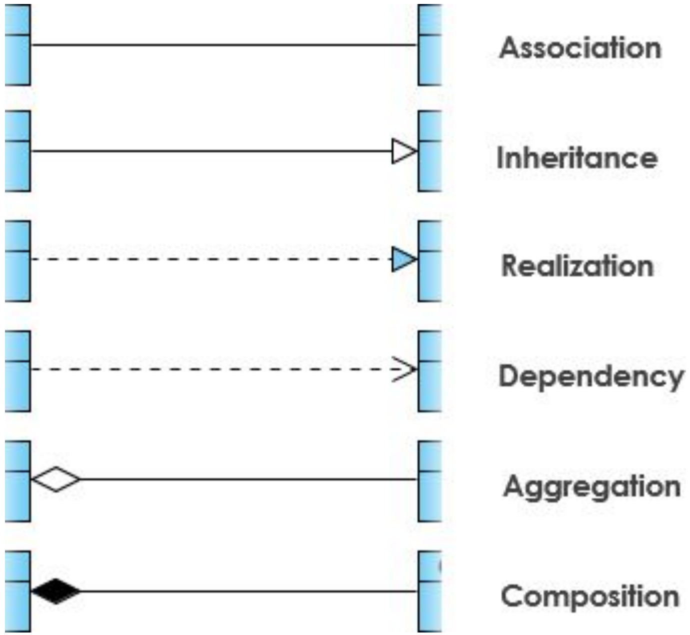
- **Conceptual**: represents the concepts in the domain
- **Specification**: focus is on the interfaces of Abstract Data Type (ADTs) in the software
- **Implementation**: describes how classes will implement their interfaces



# Relationships between classes

- UML is not just about pretty pictures.

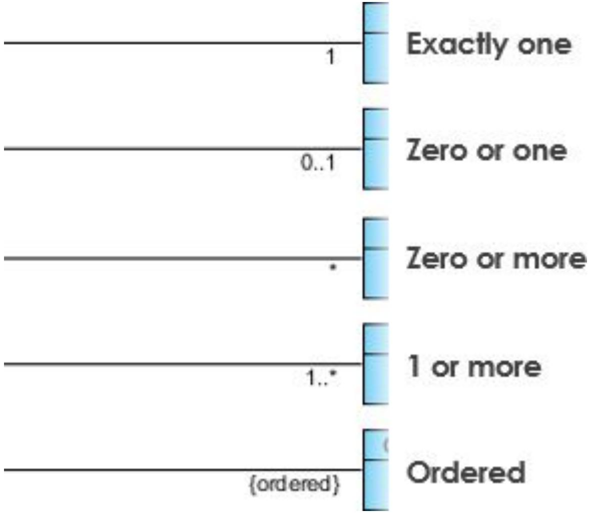
  If used correctly, UML precisely conveys how code should be implemented from diagrams.
- If precisely interpreted, the implemented code will correctly reflect the intent of the designer.
- Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?



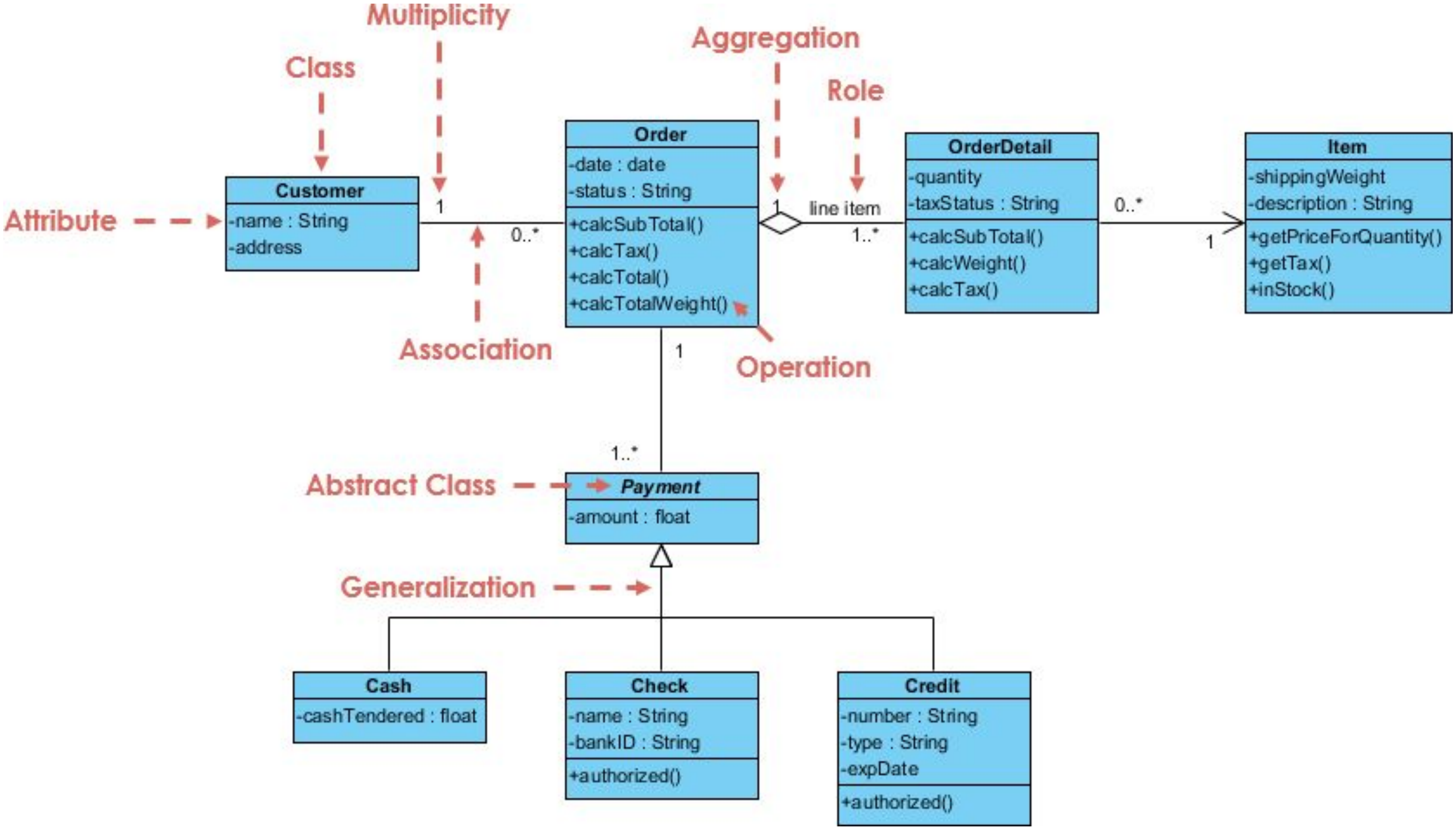
# Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many



# Example of Class diagram


Multiplicity
Class
Aggregation
Role
Attribute
Customer
-name : String
-address
1
0..*
Order
-date : date
-status : String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
1
line item
1..*
OrderDetail
-quantity
-taxStatus : String
+calcSubTotal()
+calcWeight()
+calcTax()
0..*
1
Item
-shippingWeight
-description : String
+getPriceForQuantity()
+getTax()
+inStock()
Association
Operation
1
1..*
Abstract Class
Payment
-amount : float
Generalization
Cash
-cashTendered : float
Check
-name : String
-bankID : String
+authorized()
Credit
-number : String
-type : String
-expDate
+authorized()

# Component Diagram

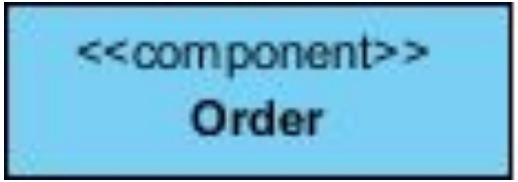
# Component Diagram

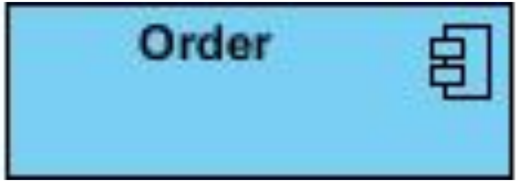
- Shows the various components in a system and their dependencies
- A component represents a modlar, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- UML Component diagrams are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- Component diagrams are essentially class diagrams that focus on a system's components that often used to model the static implementation view of a system.

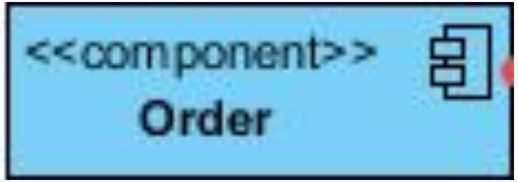
# Basic Concepts of Component Diagram

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as:

1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon





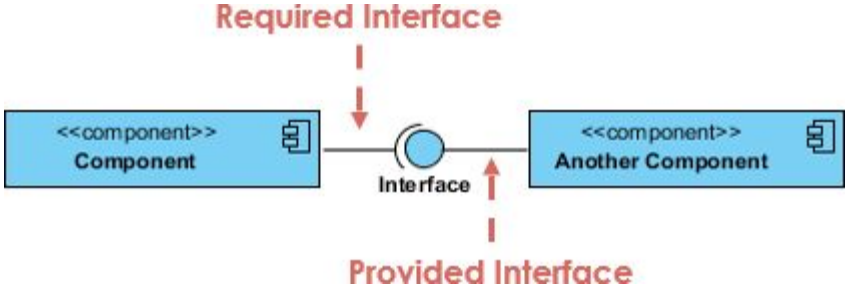


# Interface

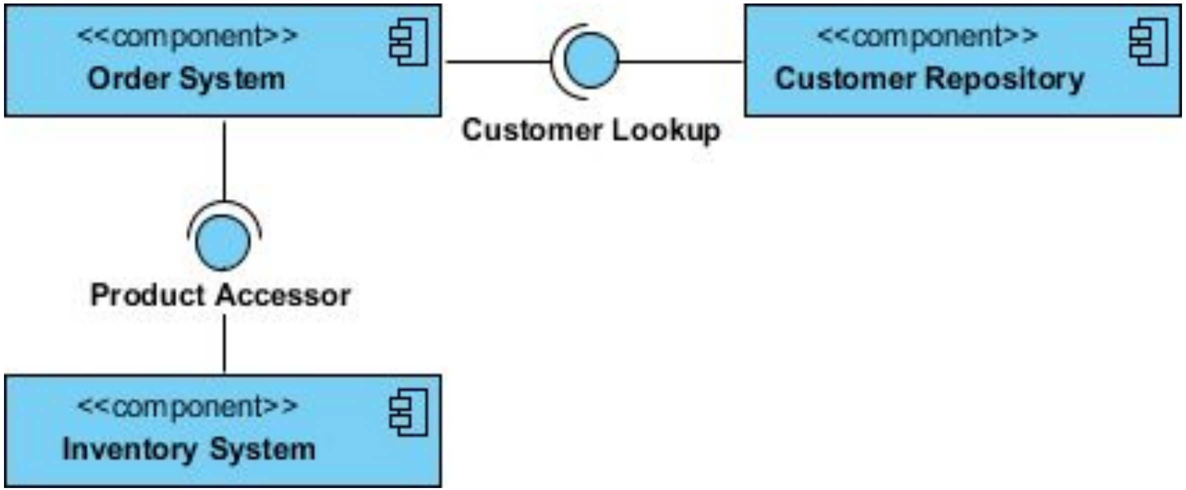
In the example below shows two type of component interfaces:

**Provided interface** symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.

**Required Interface** symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).

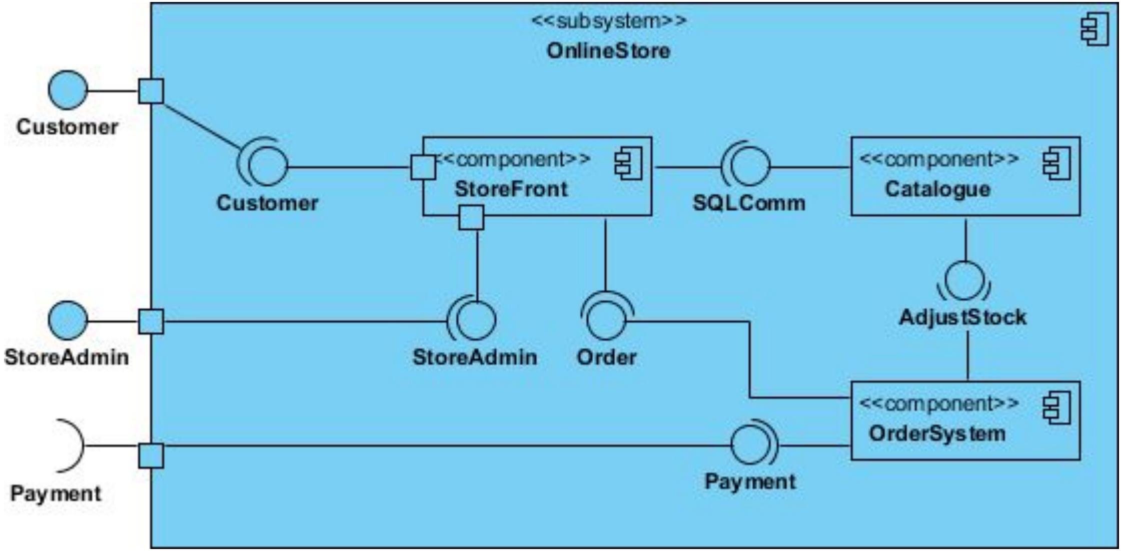


# Component Diagram Example - Using Interface (Order System)



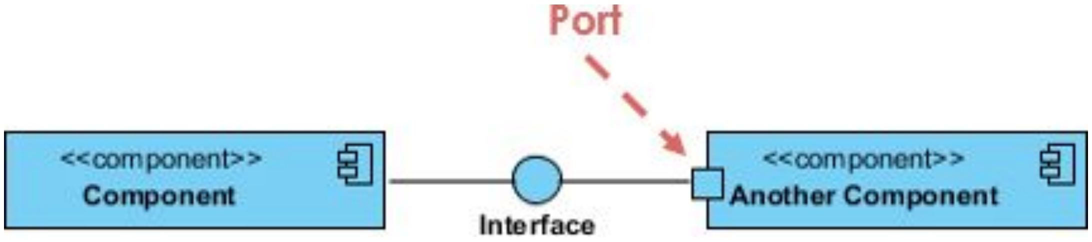
# Subsystems

The subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem instead of component.



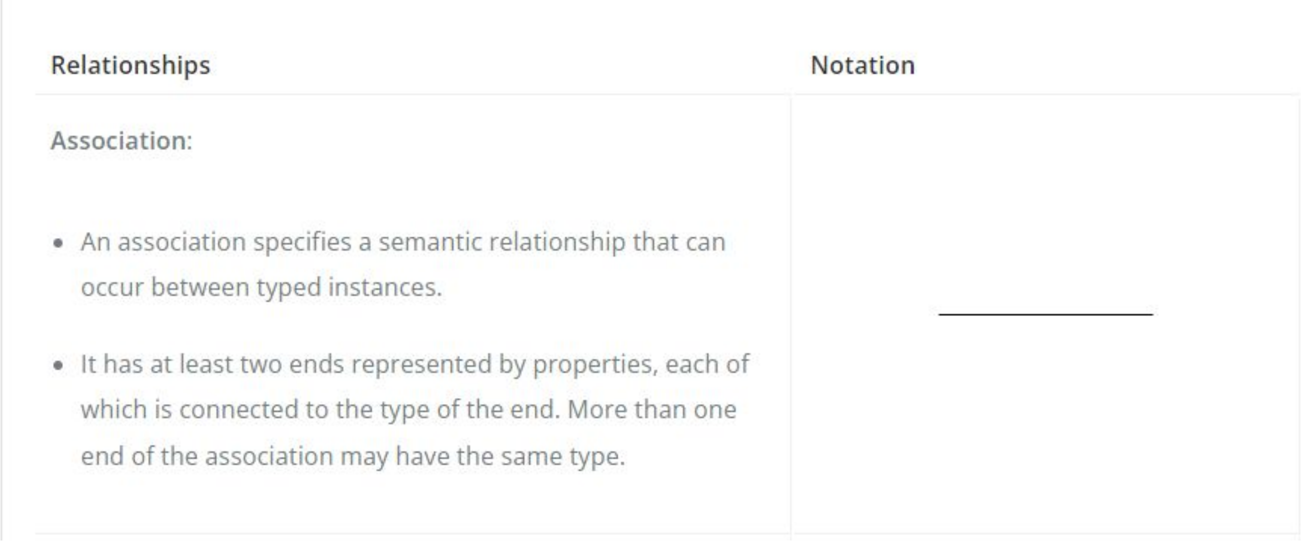
# Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.

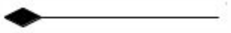
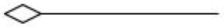
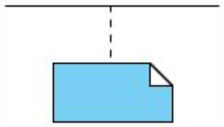


# Relationships

Graphically, a component diagram is a collection of vertices and arcs and commonly contain components, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.

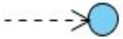


| Relationships | Notation |
|---|---|
| **Association:**<br>• An association specifies a semantic relationship that can occur between typed instances.<br>• It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type. | ——— |

| | |
|---|---|
| **Composition:**<br>• Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time.<br>• If a composite is deleted, all of its parts are normally deleted with it. |  |
| **Aggregation**<br>• A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation. |  |
| **Constraint**<br>• A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. |  |

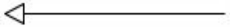
**Dependency**

- A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.
- This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).



**Links:**

- A generalization is a taxonomic relationship between a more general classifier and a more specific classifier.
- Each instance of the specific classifier is also an indirect instance of the general classifier.
- Thus, the specific classifier inherits the features of the more general classifier.



# Deployment Diagram

# Deployment Diagram

- The deployment diagram describes the physical deployment of information generated by the software program on hardware components.
- The information that the software generates is called an artifact.
- Deployment diagrams are made up of several UML shapes. The three-dimensional boxes, known as nodes, represent the basic software or hardware elements, or nodes, in the system.
- Lines from node to node indicate relationships, and the smaller shapes contained within the boxes represent the software artifacts that are deployed.

# When to Use Deployment Diagram

- What existing systems will the newly added system need to interact or integrate with?
- How robust does system need to be (e.g., redundant hardware in case of a system failure)?
- What and who will connect to or interact with system, and how will they do it
- What middleware, including the operating system and communications approaches and protocols, will system use?

# When to Use Deployment Diagram

- What hardware and software will users directly interact with (PCs, network computers, browsers, etc.)?
- How will you monitor the system once deployed?
- How secure does the system needs to be (needs a firewall, physically secure hardware, etc.)?

# Purpose of Deployment Diagrams

- They show the structure of the run-time system
- They capture the hardware that will be used to implement the system and the links between different items of hardware.
- They model physical hardware elements and the communication paths between them
- They can be used to plan the architecture of a system.
- They are also useful for Document the deployment of software components or nodes

# Deployment Diagram at a Glance

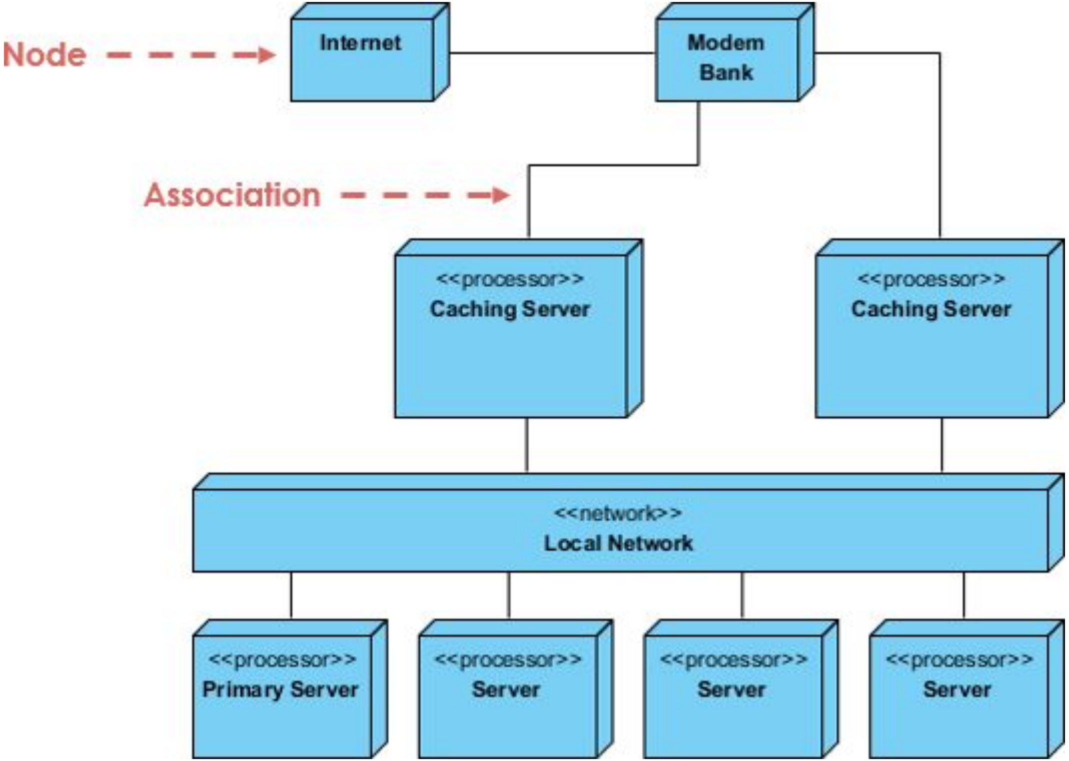
A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes

Nodes

- 3-D box represents a node, either software or hardware
- HW node can be signified with <<stereotype>>
- Connections between nodes are represented with a line, with optional <<stereotype>>
- Nodes can reside within a node

# Other Notations

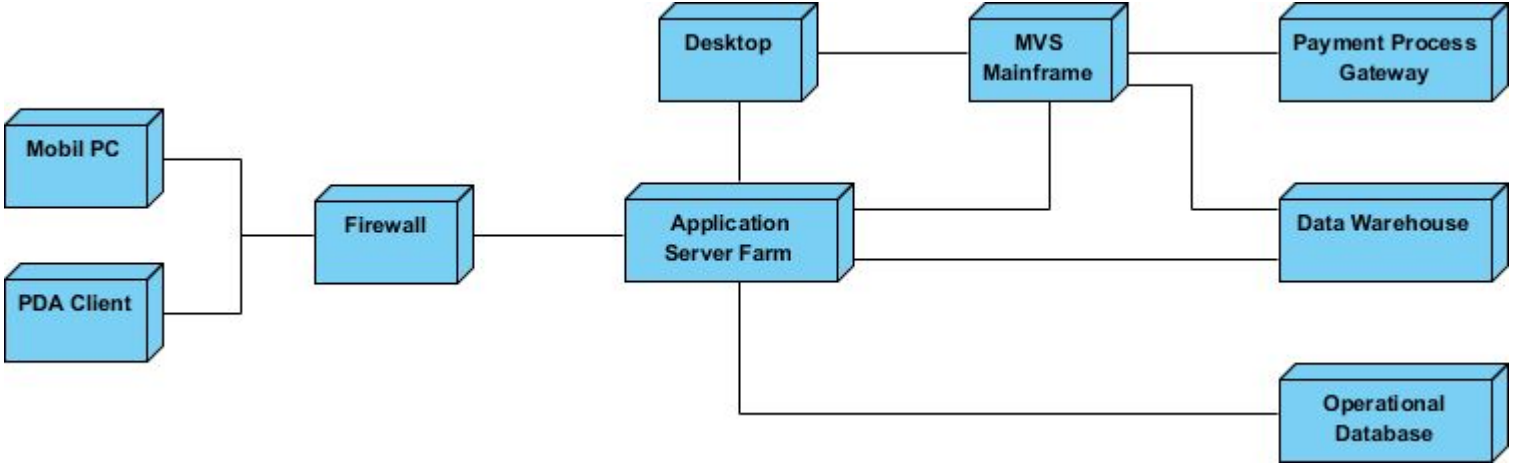
- Dependency
- Association relationships.
- contain notes
- constraints.


Node
Internet
Modem Bank
Association
<<processor>> Caching Server
<<processor>> Caching Server
<<network>> Local Network
<<processor>> Primary Server
<<processor>> Server
<<processor>> Server
<<processor>> Server

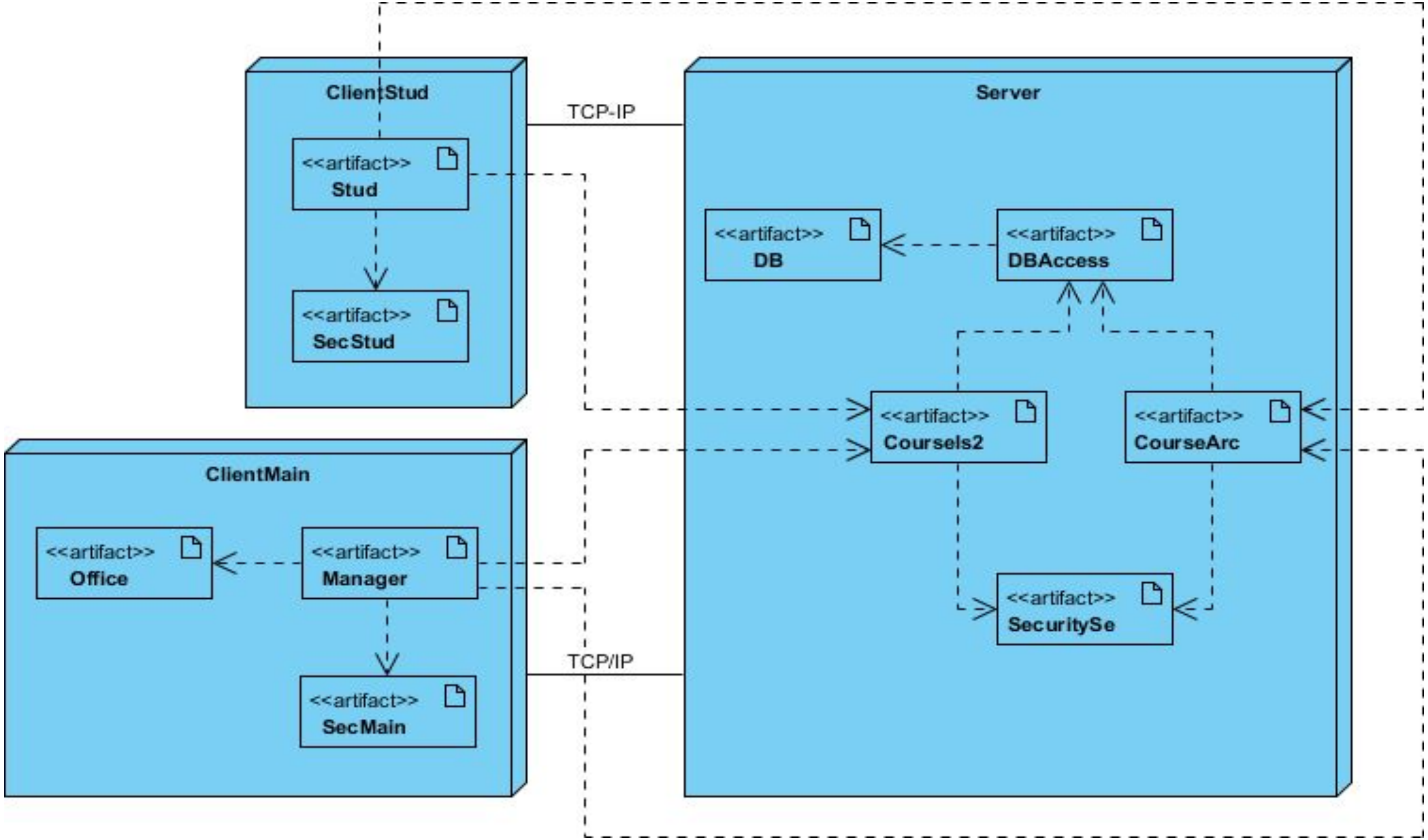
# Modeling an Embedded System

1. Identify the devices and nodes that are unique to your system.
2. Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
3. Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
4. As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

# Corporate Distributed System



# TCP/IP Client / Server Example



# Package Diagram

# Package Diagram

- Package diagrams are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages

- Each element is nested within the package, which is depicted as a file folder within the diagram, then arranged hierarchically within the diagram.

- Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.

# Purpose of Package Diagrams

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

# Basic Concepts of Package Diagram

Package diagram follows hierarchical structure of nested packages. Atomic module for nested package are usually class diagrams. There are few constraints while using package diagrams, they are as follows.

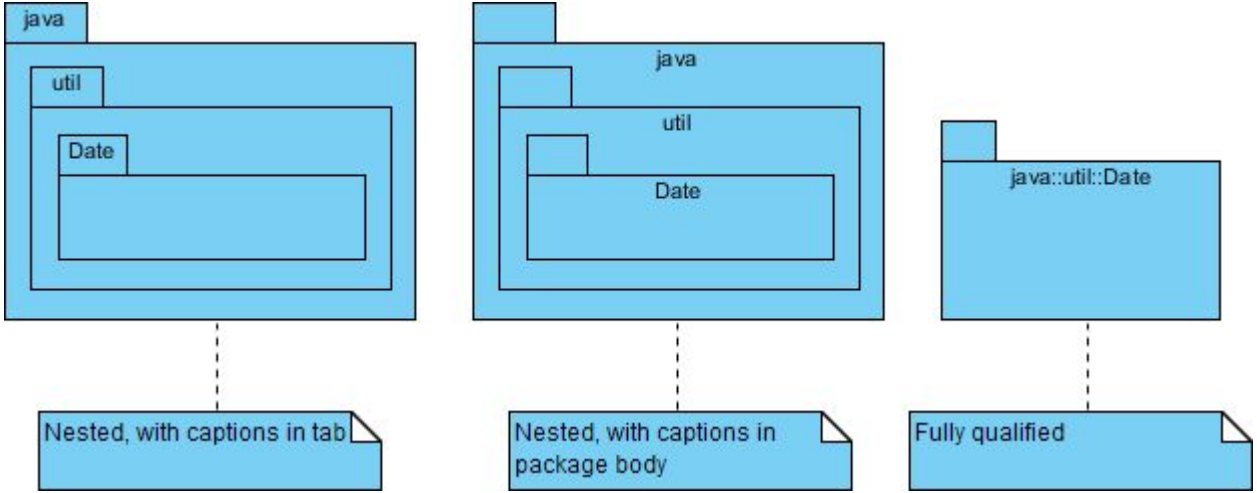
- Package name should not be the same for a system, however classes inside different packages could have the same name.
- Packages can include whole diagrams, name of components alone or no components at all.
- Fully qualified name of a package has the following syntax.

# Package Diagram Example

Packages can be represented by the notations with some examples shown below:

Name owing the package : : Name of the package

java : : util : : Date



Nested, with captions in tab

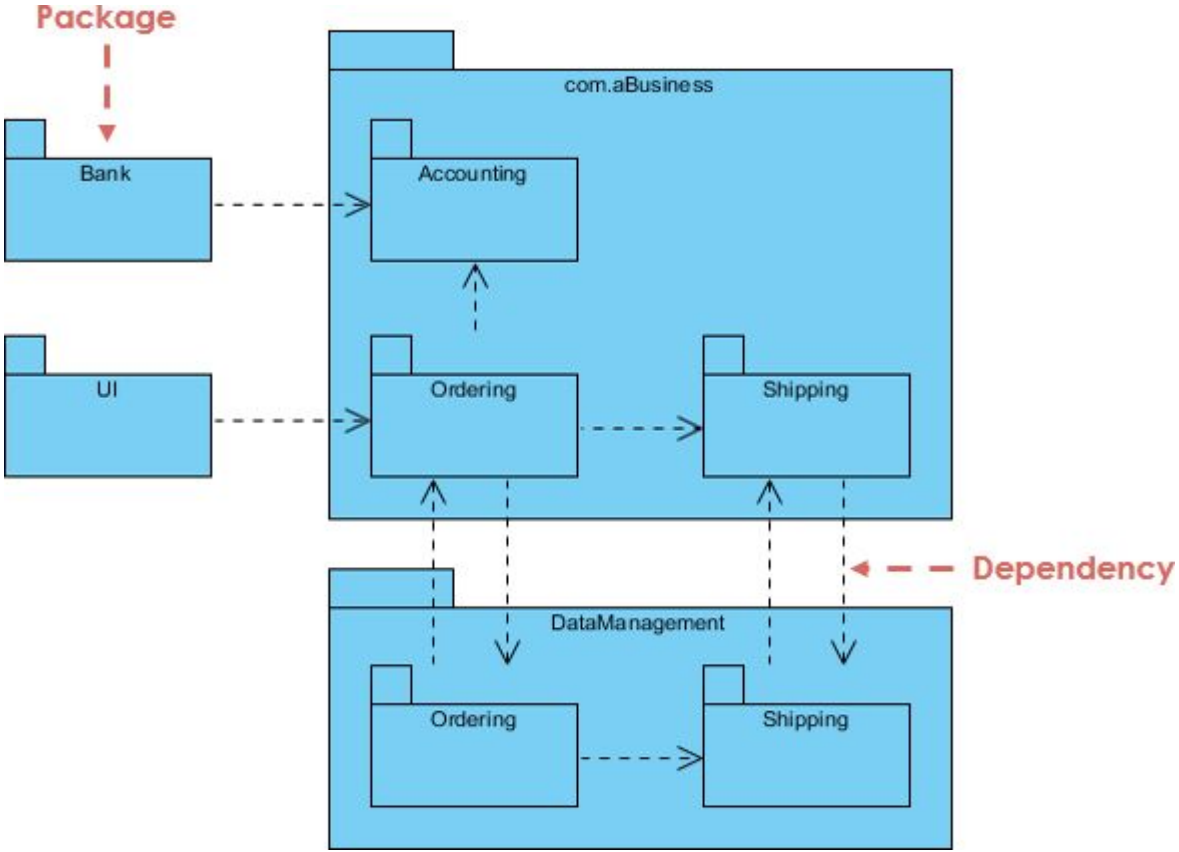
Nested, with captions in package body

Fully qualified

# Package Diagram at a Glance

- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.

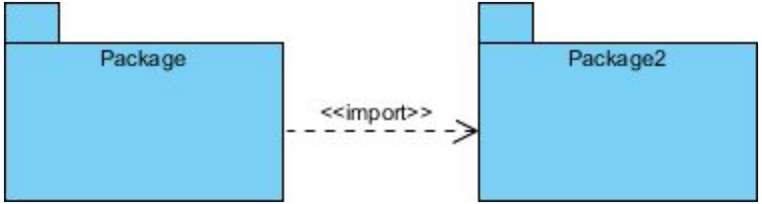
# Package Diagram Example



# Package Diagram - Dependency Notation

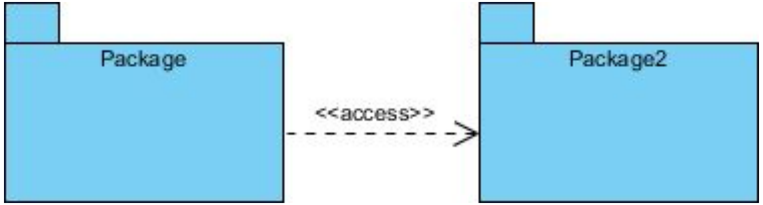
Package Diagram Example - Import

**<<import>>** - one package imports the functionality of other package

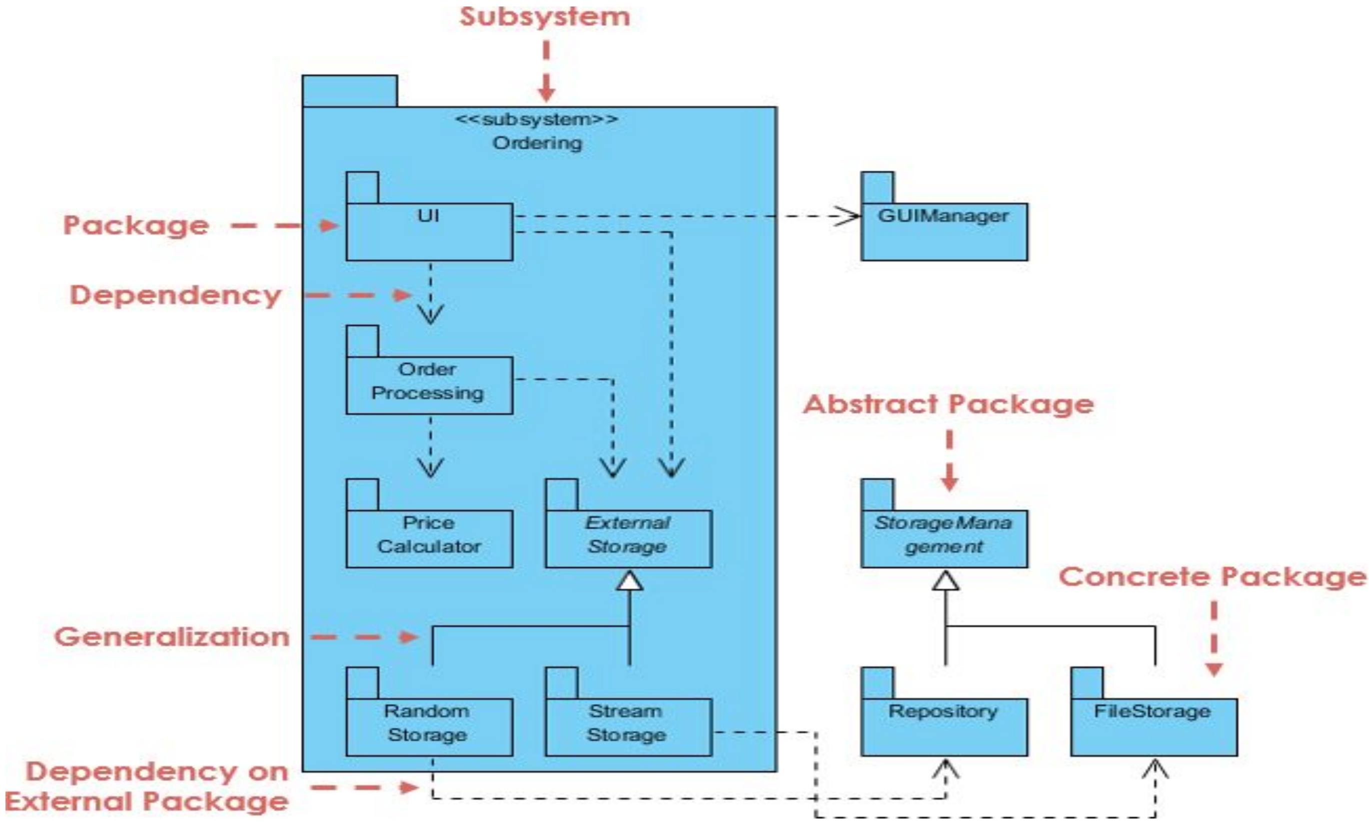


Package Diagram Example - Access

<<access>> - one package requires help from functions of other package.



# Package Diagram Example



# Object Diagram

# Object Diagram

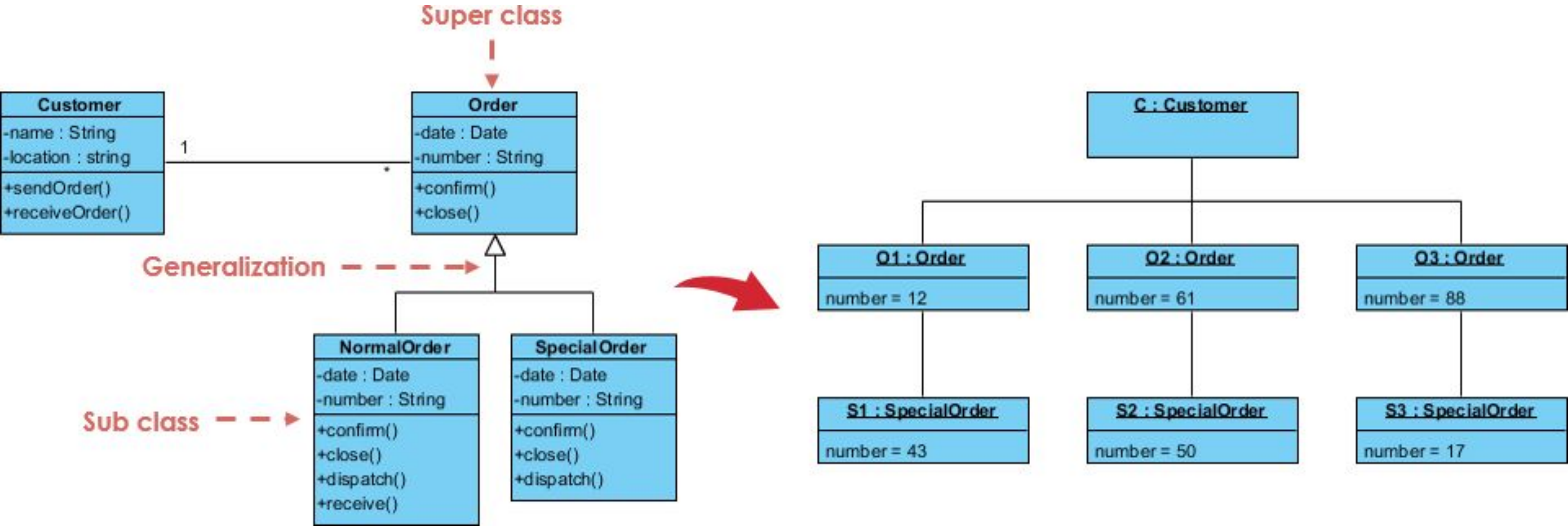
- Object is an instance of a class in a particular moment in runtime that can have its own state and data values. Likewise a static UML object diagram is an instance of a class diagram
- It shows a snapshot of the detailed state of a system at a point in time, thus an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram or a communication diagram.

# Purpose of Object Diagram

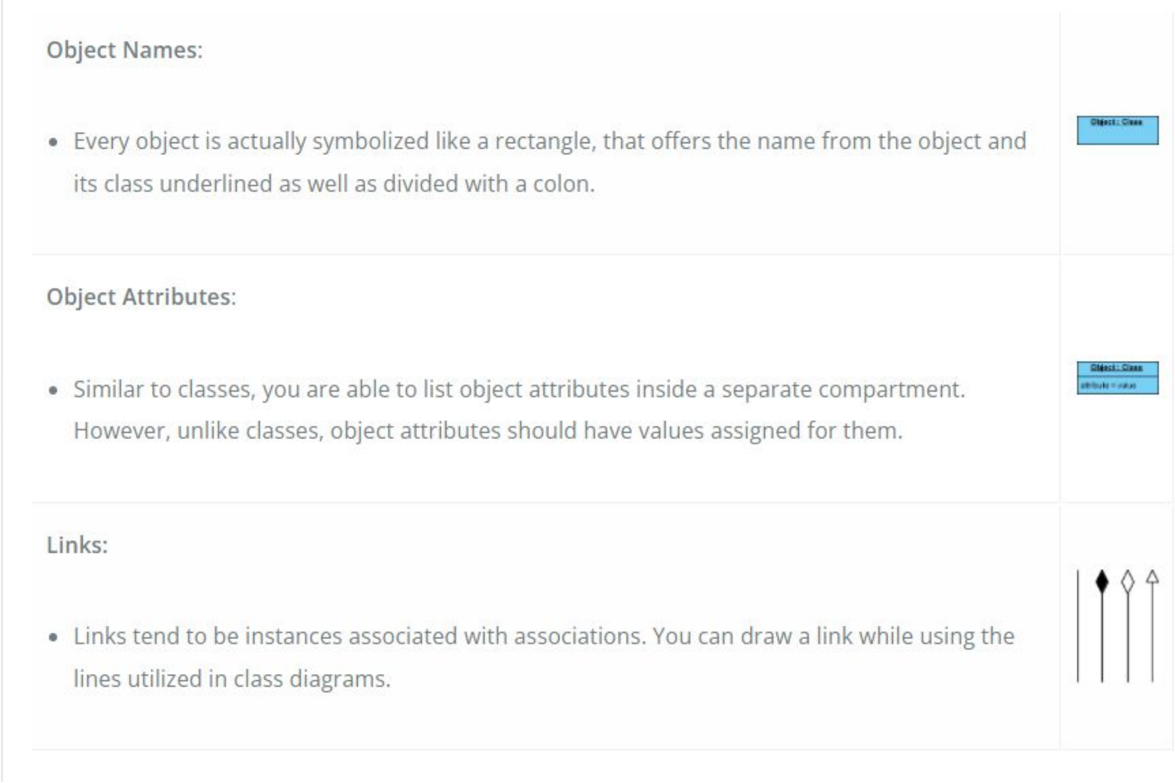
The use of object diagrams is fairly limited, mainly to show examples of data structures.

- During the analysis phase of a project, you might create a class diagram to describe the structure of a system and then create a set of object diagrams as test cases to verify the accuracy and completeness of the class diagram.
- Before you create a class diagram, you might create an object diagram to discover facts about specific model elements and their links, or to illustrate specific examples of the classifiers that are required.

# Class to Object Diagram Example



# Basic Object Diagram Symbols and Notations



| | |
|---|---|
| **Object Names:**<br>• Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon. | Object : Class |
| **Object Attributes:**<br>• Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them. | Object : Class<br>attribute = value |
| **Links:**<br>• Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams. | |

# Class Diagram vs. Object Diagram

In UML, object diagrams provide a snapshot of the instances in a system and the relationships between the instances. By instantiating the model elements in a class diagram, you can explore the behavior of a system at a point in time.

An object diagram is a UML structural diagram that shows the instances of the classifiers in models.

- Object diagrams use notation that is similar to that used in class diagrams.
- Class diagrams show the actual classifiers and their relationships in a system
- Object diagrams show specific instances of those classifiers and the links between those instances at a point in time.
- You can create object diagrams by instantiating the classifiers in class, deployment, component, and use-case diagrams.

# Profile Diagram

# Profile Diagram

- Profile diagram, a kind of structural diagram in the Unified Modeling Language (UML), provides a generic extension mechanism for customizing UML models for particular domains and platforms.
- Extension mechanisms allow refining standard semantics in strictly additive manner, preventing them from contradicting standard semantics.
- Profiles are defined using **stereotypes**, **tagged value definitions**, and **constraints** which are applied to specific model elements, like Classes, Attributes, Operations, and Activities.
- A Profile is a collection of such extensions that collectively customize UML for a particular domain (e.g., aerospace, healthcare, financial) or platform (J2EE, .NET).

# Basic Concepts of Profile Diagram

Profile diagram is basically an extensibility mechanism that allows you to extend and customize UML by adding new building blocks, creating new properties and specifying new semantics in order to make the language suitable to your specific problem domain.

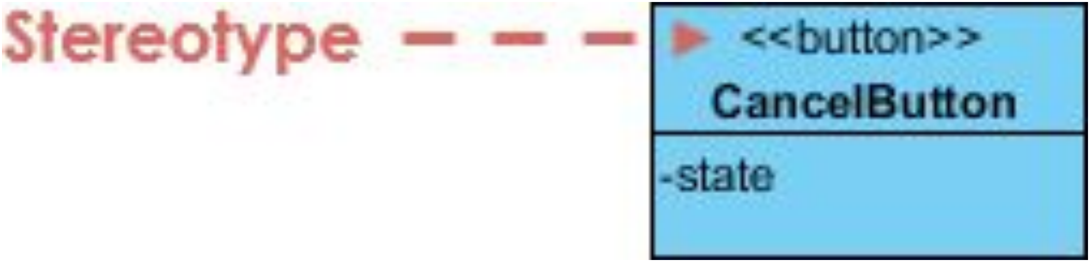
Profile diagram has three types of extensibility mechanisms:

- Stereotypes
- Tagged Values
- Constraints

# Stereotypes

Stereotypes allow you to increase vocabulary of UML. You can add, create new model elements, derived from existing ones but that have specific properties that are suitable to your problem domain. Stereotypes are used to introduce new building blocks that speak the language of your domain and look primitive. It allows you to introduce new graphical symbols.

**For example:** When modeling a network you might need to have symbols for <<router>>, <<switches>>, <<hub>> etc. A stereotype allows you to make these things appear as primitive.



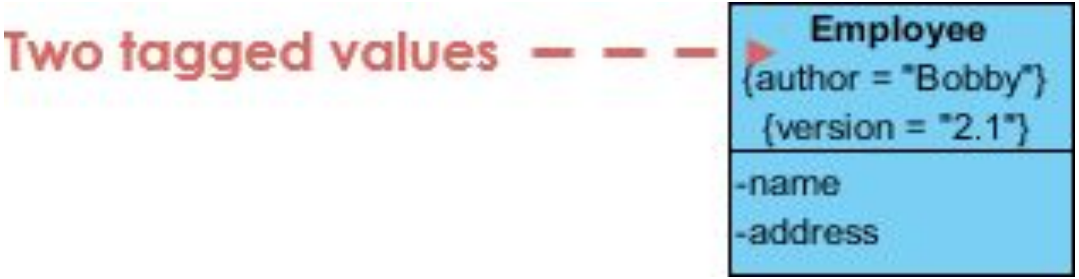
# Tagged Values

Tagged values are used to extend the properties of UML so that you can add additional information in the specification of a model element. It allows you to specify keyword value pairs of a model where keywords are the attributes. Tagged values are graphically rendered as string enclose in brackets.

**For Example:** Consider a release team responsible for assembling, testing and deployment of a system. In such case it is necessary to keep a track on version and test results of the main subsystem. Tagged values are used to add such info.

Tagged Value can be useful for adding properties to the model for some useful purposes:

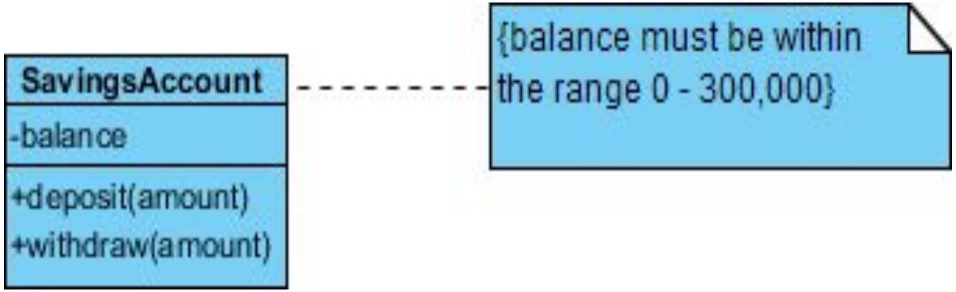
- Code generation
- Version control
- Configuration management
- Authorship

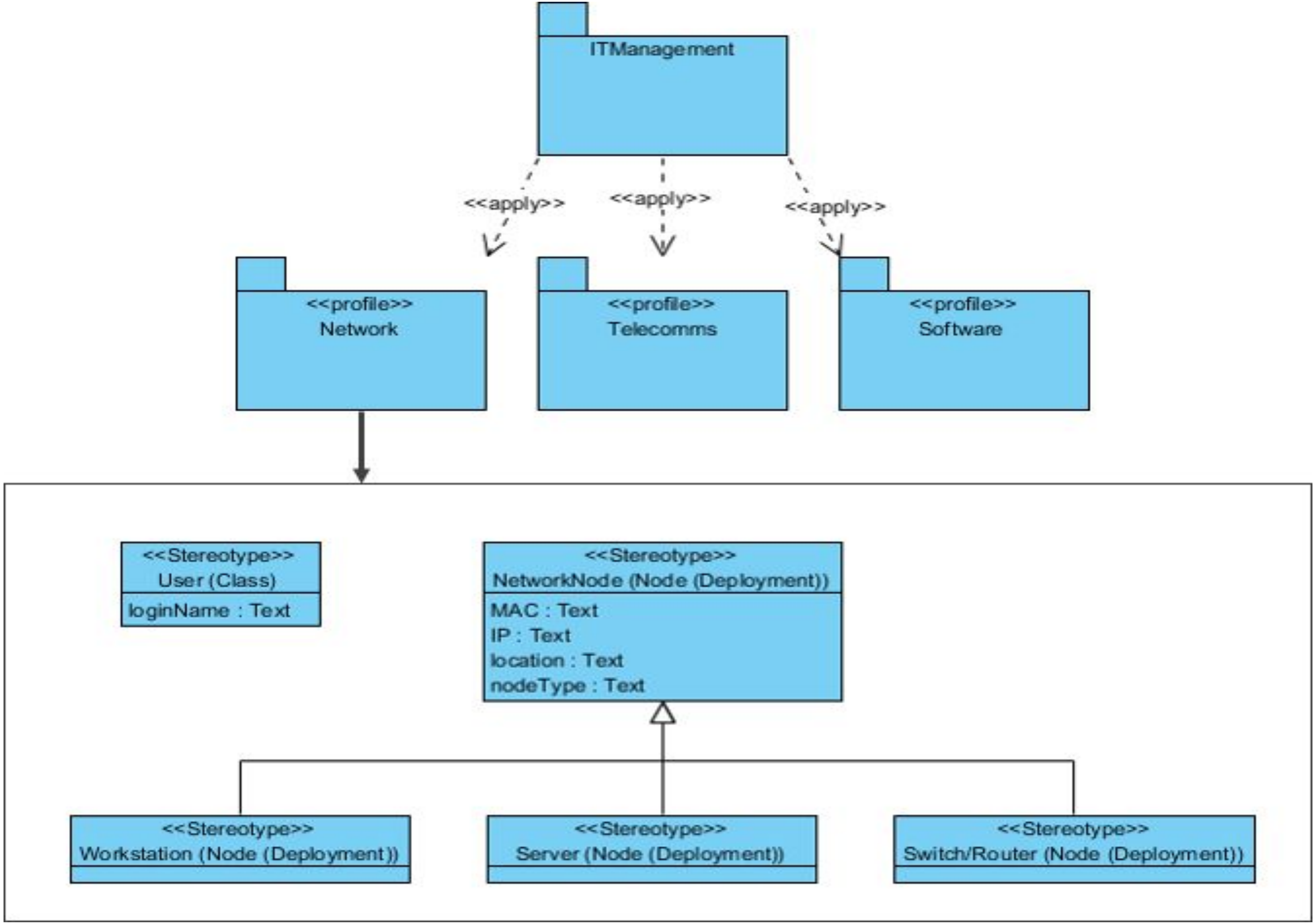


# Constraints

They are the properties for specifying semantics or conditions that must be held true at all the time. It allows you to extend the semantics of UML building block by adding new protocols. Graphically a constraint is rendered as string enclose in brackets placed near associated element.

**For example:** In development of a real time system it is necessary to adorn the model with some necessary information such as response time. A constraint defines a relationship between model elements that must be use *{subset}* or *{xor}*. Constraints can be on attributes, derived attributes and associations. It can be attached to one or more model elements shown as a note as well.




ITManagement
<<apply>>
<<apply>>
<<apply>>
<<profile>>
Network
<<profile>>
Telecomms
<<profile>>
Software
<<Stereotype>>
User (Class)
loginName : Text
<<Stereotype>>
NetworkNode (Node (Deployment))
MAC : Text
IP : Text
location : Text
nodeType : Text
<<Stereotype>>
Workstation (Node (Deployment))
<<Stereotype>>
Server (Node (Deployment))
<<Stereotype>>
Switch/Router (Node (Deployment))

# Composite Structure Diagram

# Composite Structure Diagram

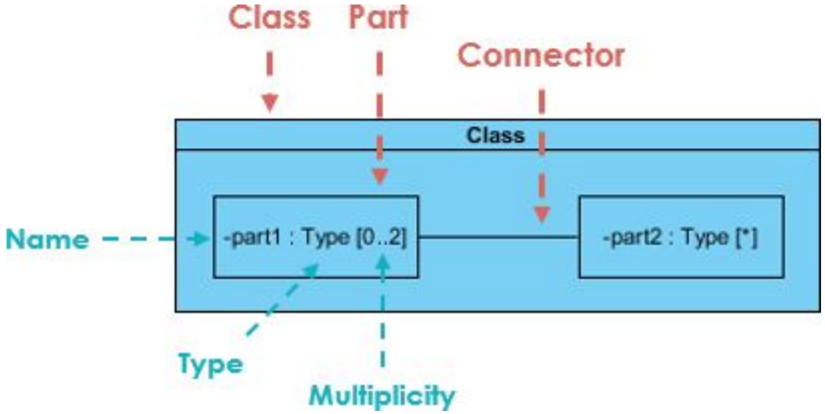
- Composite Structure Diagram is one of the new artifacts added to UML 2.0. A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system. It shows the internal structure (including parts and connectors) of a structured classifier or collaboration
- A composite structure diagram performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them. You can graphically represent inner classes and parts and show associations both between and within classes.

# Purpose of Composite Structure Diagram

- Composite Structure Diagrams allow the users to "Peek Inside" an object to see exactly what it is composed of.
- The internal actions of a class, including the relationships of nested classes, can be detailed.
- Objects are shown to be defined as a composition of other classified objects.

# Composite Structure Diagram at a Glance

- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class but parts are not necessarily classes, a part is any element that is used to make up the containing class.

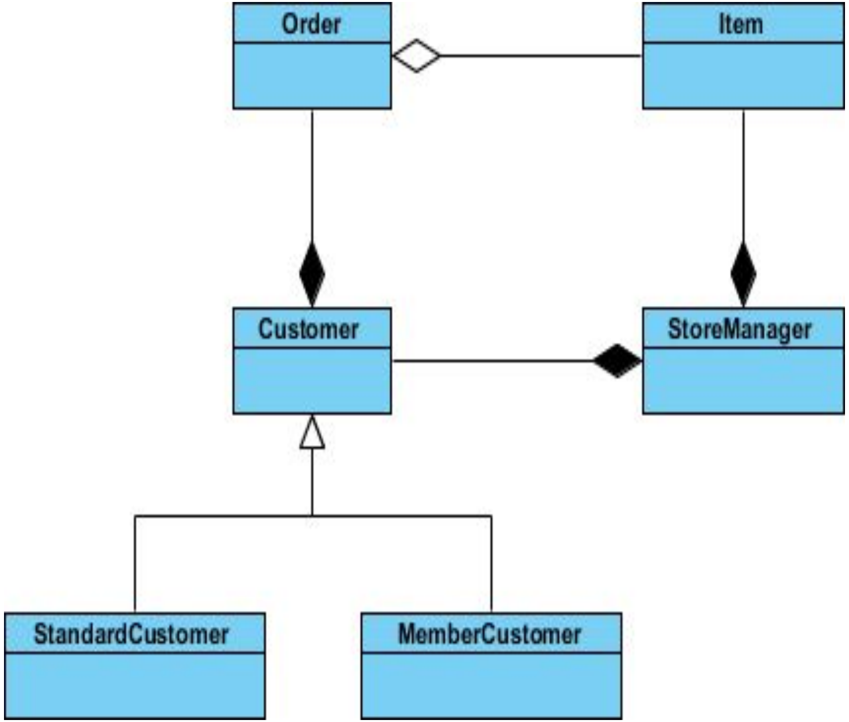


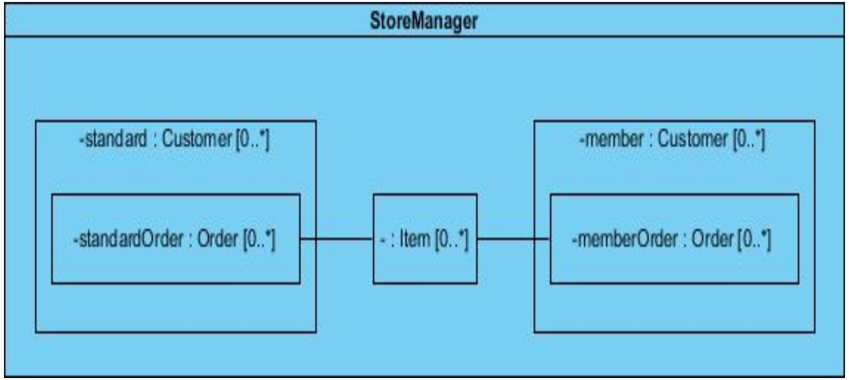
# Deriving Composite Structure Diagram from Class Diagram

## Example - Online Store

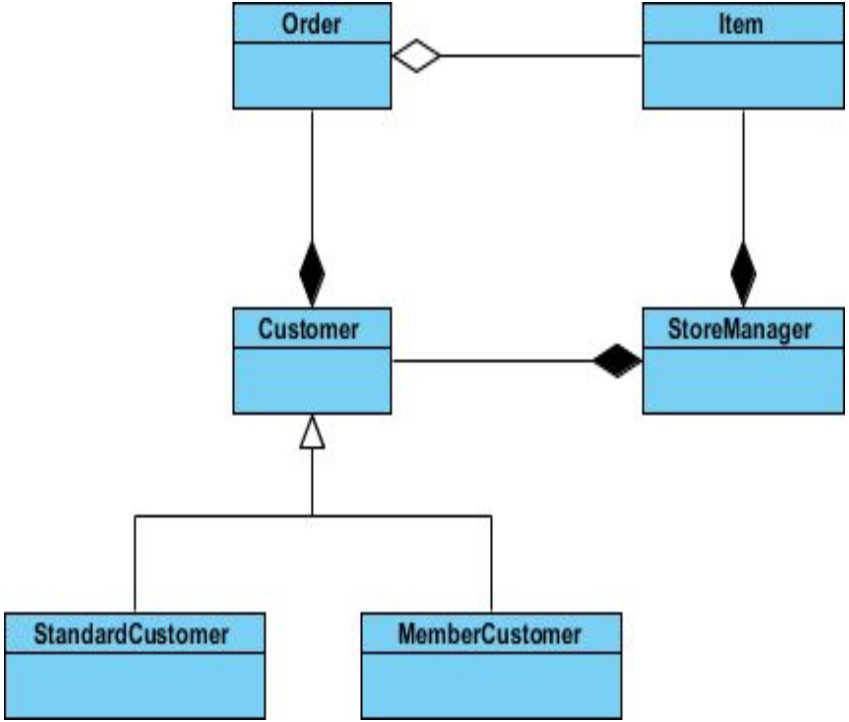
- Suppose we are modeling a system for an online store.
- The client has told us that customers may join a membership program which will provide them with special offers and discounted shipping.
- So we have extended the customer object to provide a member and standard option.

# Deriving Composite Structure Diagram from Class Diagram



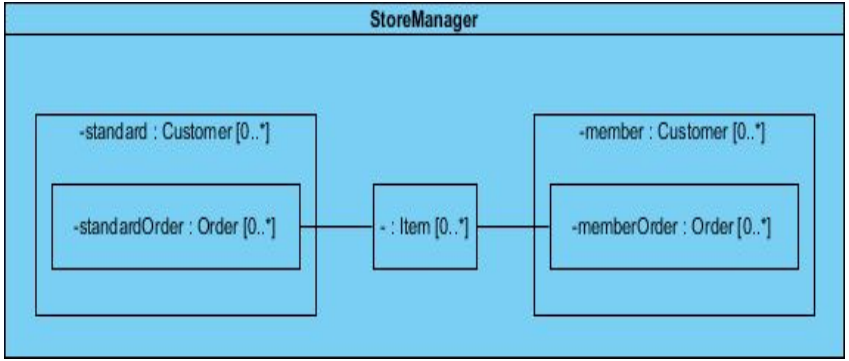

StoreManager
-standard : Customer [0..*]
-standardOrder : Order [0..*]
- : Item [0..*]
-member : Customer [0..*]
-memberOrder : Order [0..*]

# Deriving Composite Structure Diagram from Class Diagram



- We have a class for Item which may be aggregated by the Order class, which is composed by the Customer class which itself is composed by the StoreManager class.
- **We have a lot of objects that end up within other objects.**

# Deriving Composite Structure Diagram from Class Diagram


StoreManager
-standard : Customer [0..*]
-standardOrder : Order [0..*]
- : Item [0..*]
-member : Customer [0..*]
-memberOrder : Order [0..*]

In the example, we can see:

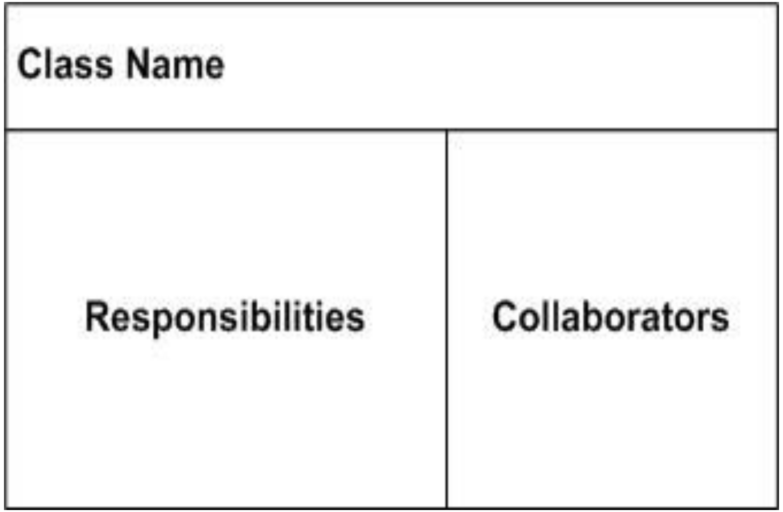
- StoreManager from its own perspective, instead of the system as a whole.
- StoreManager directly contains two types of objects (**Customer** and **Item**) as is indicated by the **two composition arrows on the class diagram.**

# Deriving Composite Structure Diagram from Class Diagram

- The composite structure diagram here shows more explicitly is the inclusion of the subtypes of Customer.
- Notice that the type of both of these parts is Customer, as the store sees both as Customer objects.
- We also see a connector which shows the relation between Item and Order.
- Order is not directly contained within the StoreManager class but we can show relations to parts nested within the objects it aggregates.

# Class-responsibility-collaborator cards (CRC cards)

Class-responsibility-collaborator cards (CRC cards) are not a part of the UML specification, but they are a useful tool for organising classes during analysis and design. A CRC card is **a physical card representing a single class**.

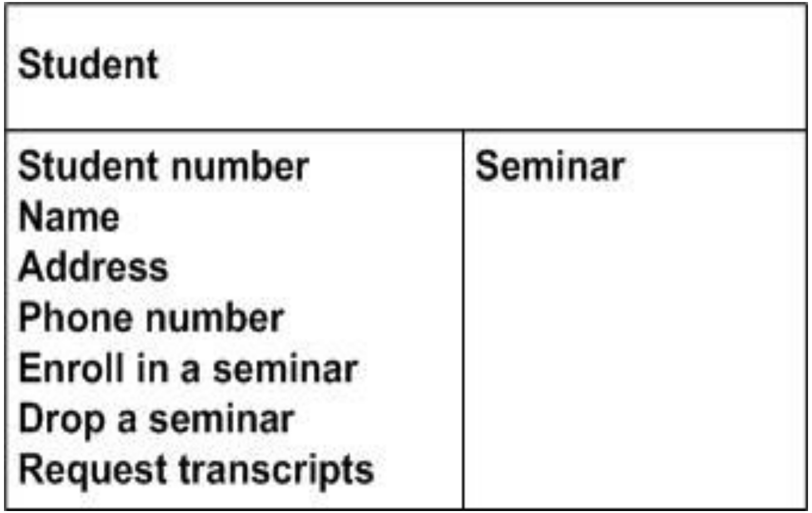


Class Name

Responsibilities

Collaborators

# Example



| Student | |
|---|---|
| Student number<br>Name<br>Address<br>Phone number<br>Enroll in a seminar<br>Drop a seminar<br>Request transcripts | Seminar |

# CRC

- A responsibility is anything that a class knows or does. For example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does. The things a class knows and does constitute its responsibilities. Important: A class is able to change the values of the things it knows, but it is unable to change the values of what other classes know.
- Sometimes a class has a responsibility to fulfill, but not have enough information to do it. For example, students enroll in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled *Seminar* to sign up for a seminar. Therefore, *Seminar* is included in the list of collaborators of *Student.*

# CRC

- Collaboration takes one of two forms: A request for information or a request to do something. For example, the card *Student* requests an indication from the card *Seminar* whether a space is available, a request for information. *Student* then requests to be added to the *Seminar*, a request to do something. Another way to perform this logic, however, would have been to have *Student* simply request *Seminar* to enroll himself into itself. Then have *Seminar* do the work of determining if a seat is available and, if so, then enrolling the student and, if not, then informing the student that he was not enrolled.

# Creating CRC Cards

So how do you create CRC models? Iteratively perform the following steps:

- **Find classes**. Finding classes is fundamentally an analysis task because it deals with identifying the building blocks for your application. A good rule of thumb is that you should look for the three-to-five main classes right away, such as *Student*, *Seminar*, and *Professor* in Figure 4. I will sometimes include UI classes such as *Transcript* and *Student Schedule*, both are reports, although others will stick to just entity classes. Also, I'll sometimes include cards representing actors when my stakeholders are struggling with the concept of a student in the real world (the actor) versus the student in the system (the entity).
- **Find responsibilities**. You should ask yourself what a class does as well as what information you wish to maintain about it. You will often identify a responsibility for a class to fulfill a collaboration with another class.

# Creating CRC Cards

- **Define collaborators**. A class often does not have sufficient information to fulfill its responsibilities. Therefore, it must collaborate (work) with other classes to get the job done. Collaboration will be in one of two forms: a request for information or a request to perform a task. To identify the collaborators of a class for each responsibility ask yourself "does the class have the ability to fulfill this responsibility?". If not then look for a class that either has the ability to fulfill the missing functionality or the class which should fulfill it. In doing so you'll often discover the need for new responsibilities in other classes and maybe even the need for a new class or two.
- **Move the cards around**. To improve everyone's understanding of the system, the cards should be placed on the table in an intelligent manner. Two cards that collaborate with one another should be placed close together on the table, whereas two cards that don't collaborate should be placed far apart. Furthermore, the more two cards collaborate, the closer they should be on the desk. By having cards that collaborate with one another close together, it's easier to understand the relationships between classes.