Introduction to UML

Dr.S.Neelavathy Pari

Assistant Professor (Sr. Grade)

Department of Computer Technology

Anna University, MIT Campus

What is UML?

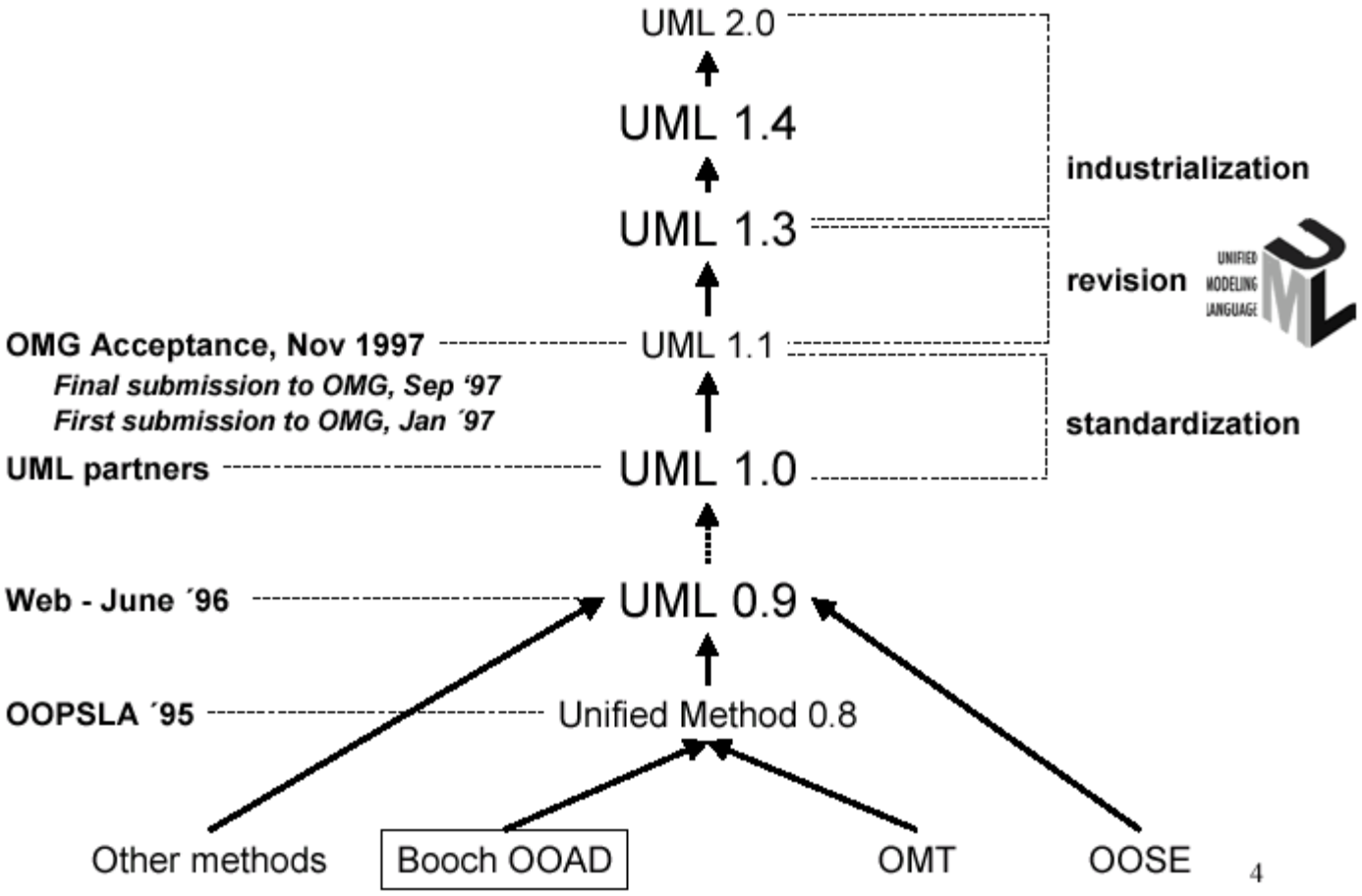
- Unified Modeling Language
  - OMG Standard, Object Management Group
  - Based on work from Booch, Rumbaugh, Jacobson

- UML is a modeling language to express and design documents, software
  - Particularly useful for OO design
  - Not a process, but some have been proposed using UML
  - Independent of implementation language

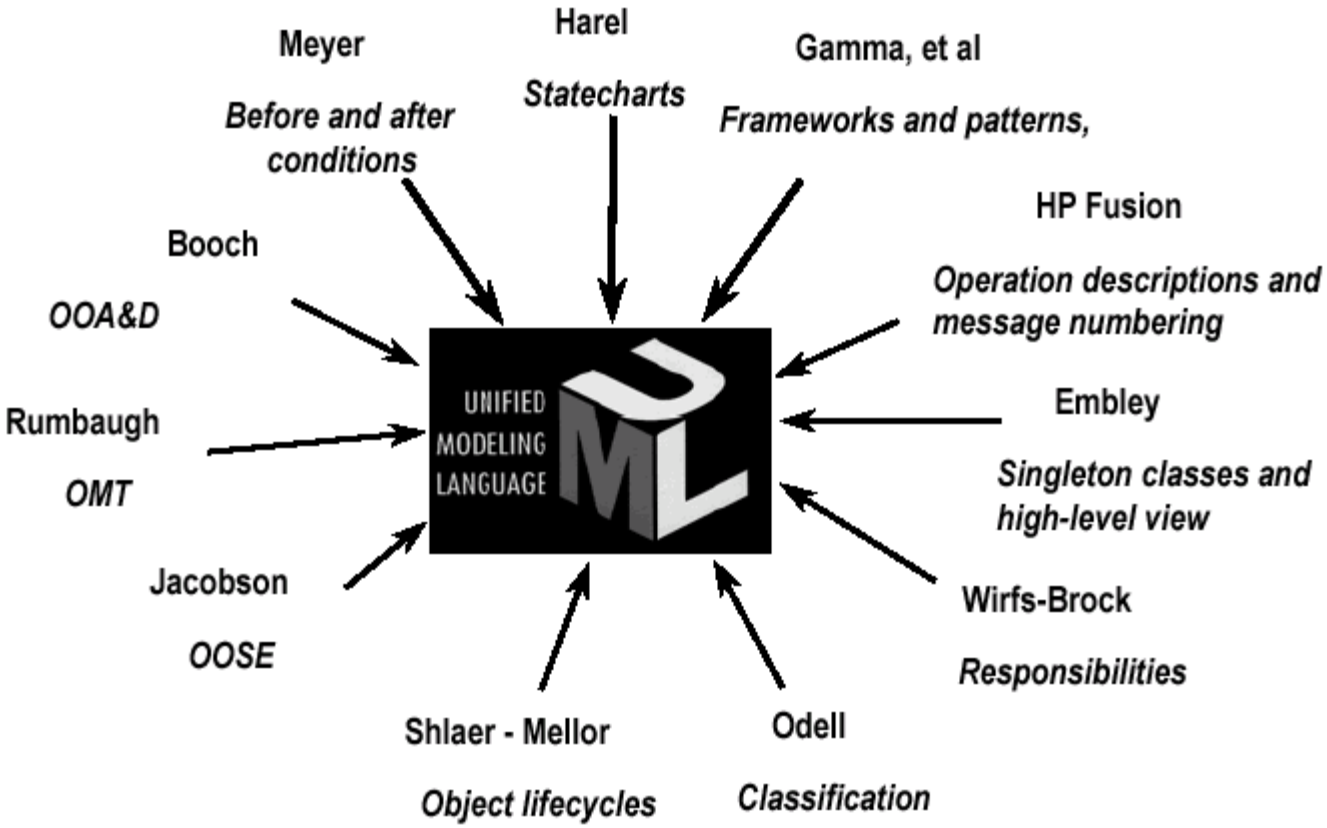
Why use UML ?

- Open Standard, Graphical notation for
  - Specifying, visualizing, constructing, and documenting software systems
- Language can be used from general initial design to very specific detailed design across the entire software development lifecycle
- Increase understanding/communication of product to customers and developers
- Support for diverse application areas
- Support for UML in many software packages today (e.g. Rational, plugins for popular IDE's like NetBeans, Eclipse)
- Based upon experience and needs of the user community

History of UML



Contributions to UML



UML

- OMG is continuously making efforts to create a truly industry standard.
  - UML stands for **Unified Modeling Language**.
  - UML is different from the other common programming languages such as C++, Java, COBOL, etc.
  - UML is a pictorial language used to make software blueprints.
  - UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
  - Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.
- UML is not a programming language but tools can be used to generate code in various languages using UML diagrams.
- UML has a direct relation with object oriented analysis and design.
- After some standardization, UML has become an OMG standard.

UML

- For now, understand that UML is a language – it is used to communicate information
- We will use UML to describe the problem domain, describe the activities that occur, and eventually describe the software classes
- Since it is a language, UML has specific rules, and we will see these later in the course
- You need to be able to read UML diagrams, as well as create them Here are some examples (we will learn more about how to create these diagrams later …)

UML

- Conceptual Perspective – defining the problem domain: Raw class diagrams, maybe mention some attributes (Domain Model)
- Specification Perspective – defining the software classes: Design Class diagram, which shows the actual software classes and their methods, attributes

A Conceptual Model of UML

- A conceptual model can be defined as a model which is made of concepts and their relationships.
    - A conceptual model is the first step before drawing a UML diagram.
    - It helps to understand the entities in the real world and how they interact with each other.
- As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −
  - UML building blocks
  - Rules to connect the building blocks
  - Common mechanisms of UML

Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system.
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other

Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Object-Oriented Concepts

- UML can be described as the successor of object-oriented (OO) analysis and design.
- An object contains both data and methods that control the data.
  - The data represents the state of the object.
  - A class describes an object and they also form a hierarchy to model the real-world system.
  - The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.
- Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

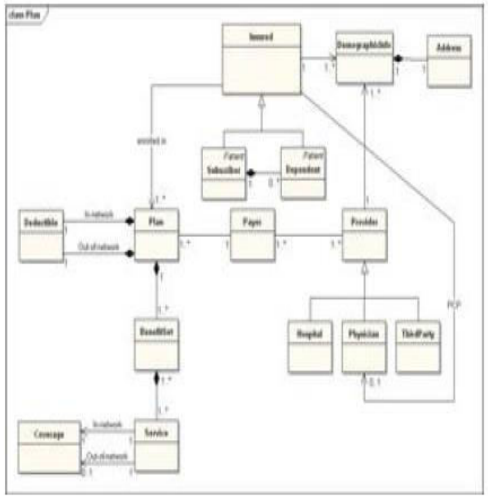
Fundamental concepts of the object-oriented world

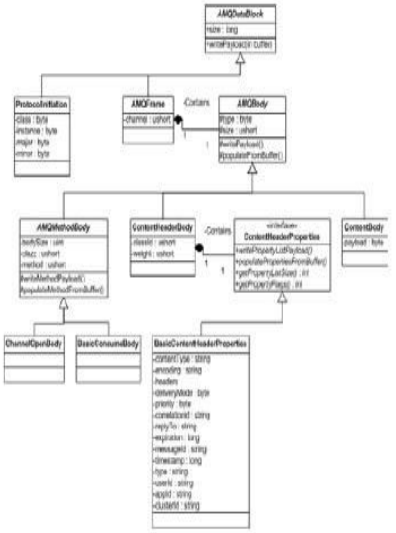
- **Objects** – Objects represent an entity and the basic building block.
- **Class** – Class is the blue print of an object.
- **Abstraction** – Abstraction represents the behavior of an real world entity.
- **Encapsulation** – Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** – Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** – It defines the mechanism to exists in different forms.

- The purpose of OO analysis and design can described as
  - Identifying the objects of a system.
  - Identifying their relationships.
  - Making a design, which can be converted to executables using OO languages.

OO Analysis → OO Design → OO implementation using OO languages

UML - Example



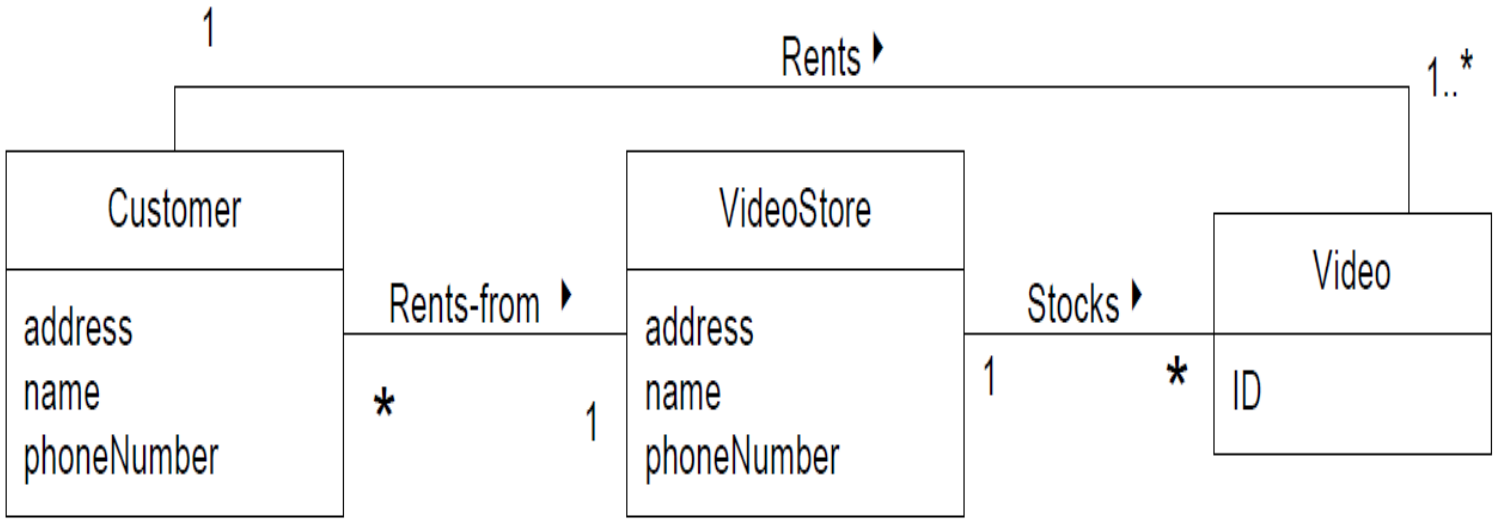


Analyze the system

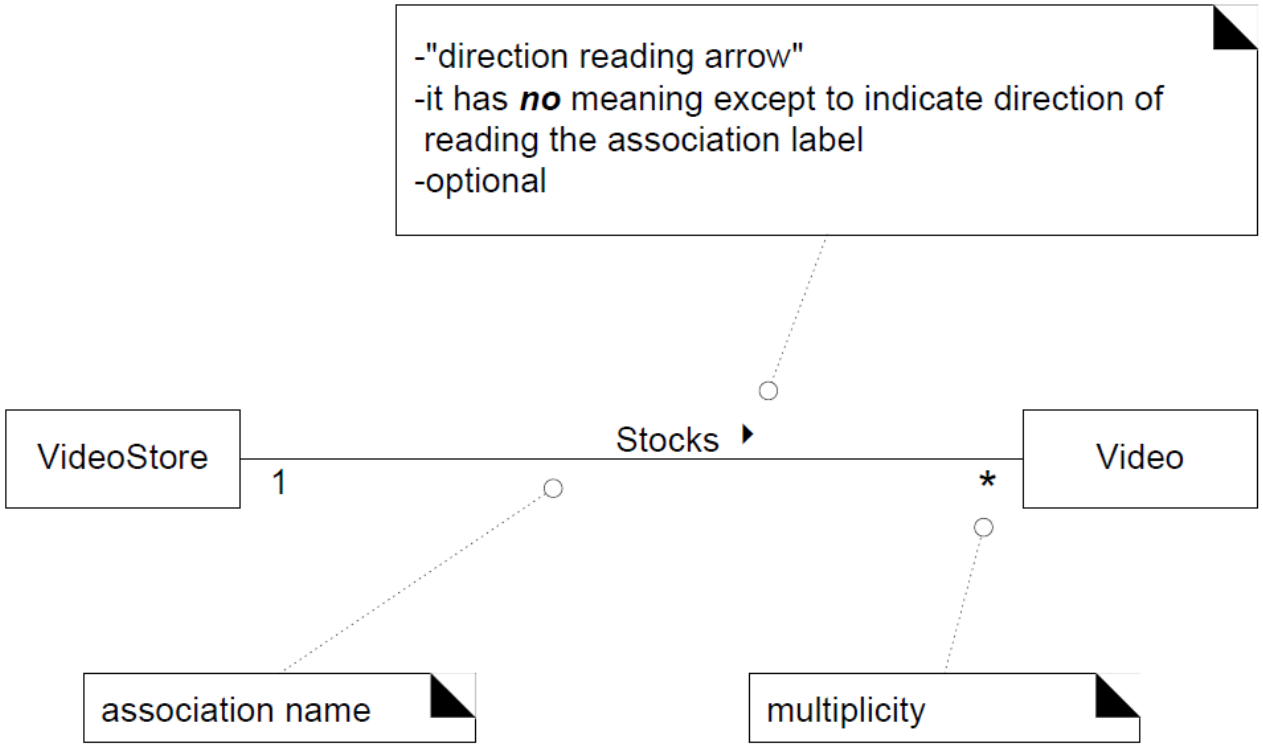
Model the system

Design the software

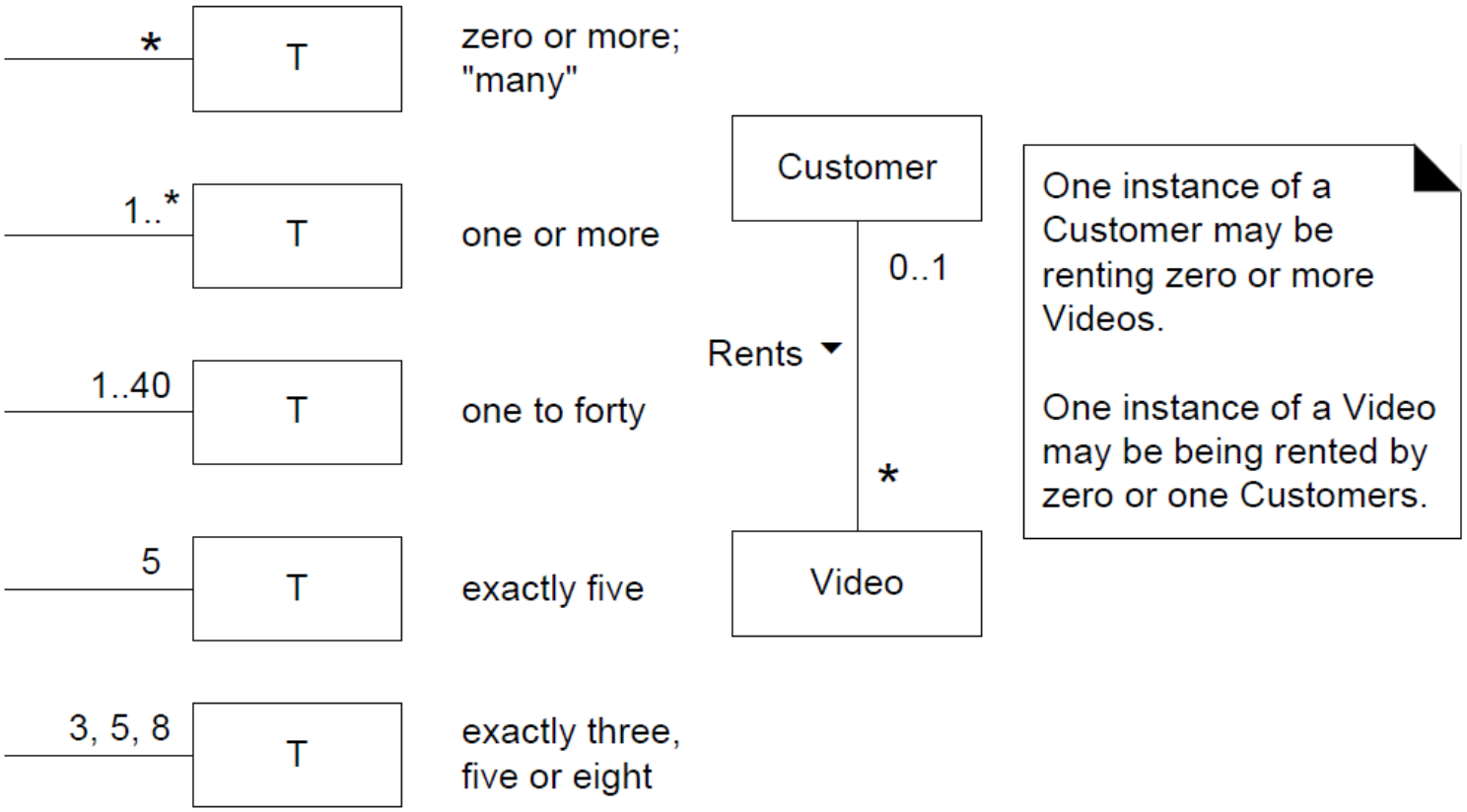
UML - Example


1
Rents
1..*
Customer
address
name
phoneNumber
Rents-from
*
1
VideoStore
address
name
phoneNumber
Stocks
1
*
Video
ID

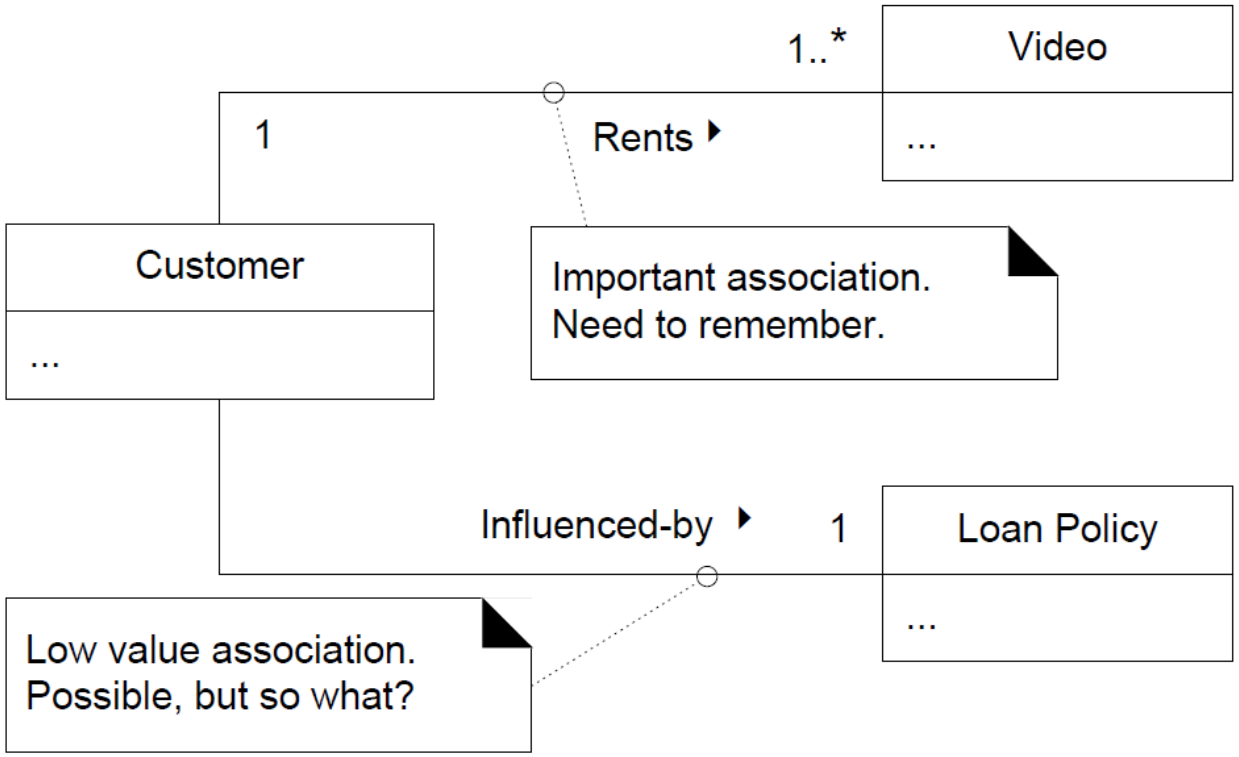
UML - Example


-"direction reading arrow"
-it has **no** meaning except to indicate direction of reading the association label
-optional
VideoStore
Stocks
Video
1
*
association name
multiplicity

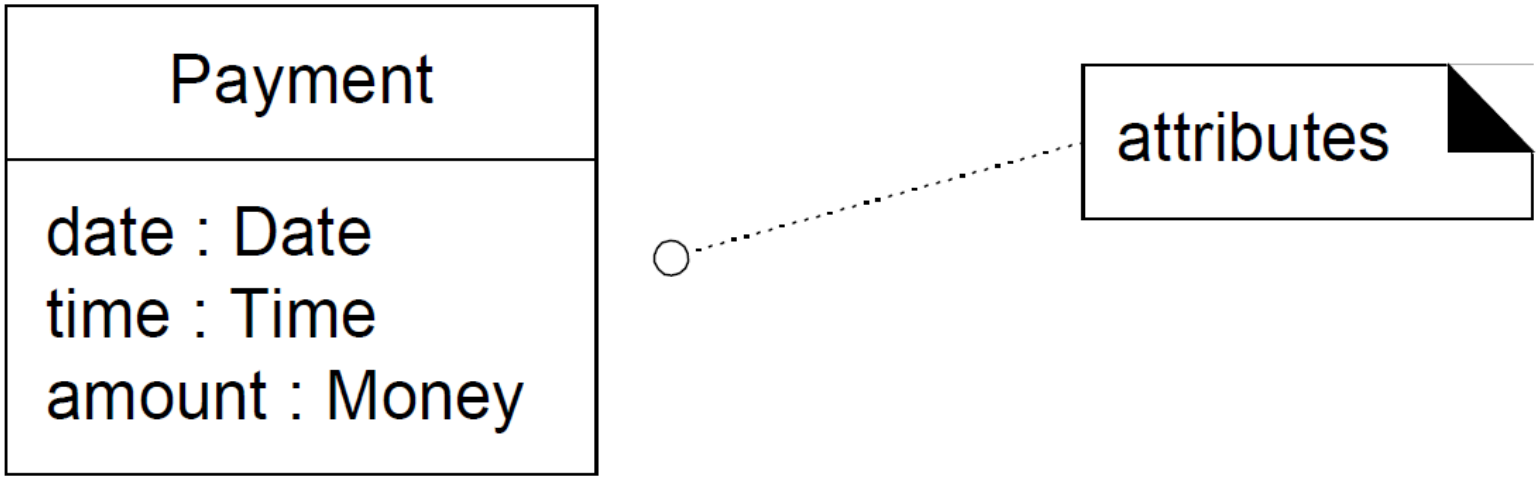
UML - Example


*
T
zero or more;
"many"
1..*
T
one or more
1..40
T
one to forty
5
T
exactly five
3, 5, 8
T
exactly three,
five or eight
Customer
0..1
Rents
*
Video
One instance of a Customer may be renting zero or more Videos.
One instance of a Video may be being rented by zero or one Customers.

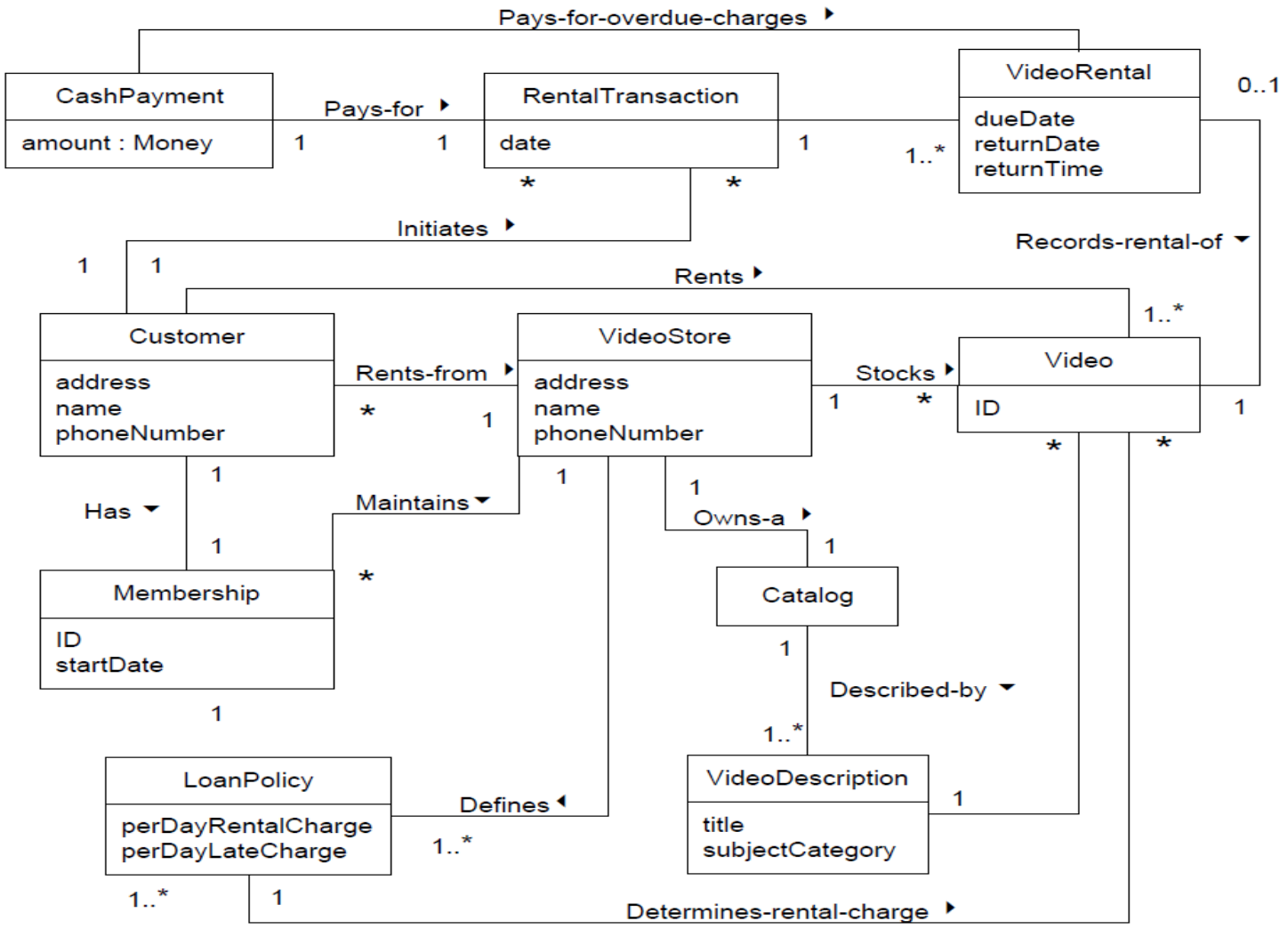
UML - Example


1..*
Video
...
1
Rents
Customer
...
Important association.
Need to remember.
Influenced-by
1
Loan Policy
...
Low value association.
Possible, but so what?

UML - Example



UML - Example


Pays-for-overdue-charges
CashPayment
amount : Money
Pays-for
1
1
RentalTransaction
date
1
1..*
VideoRental
dueDate
returnDate
returnTime
0..1
*
*
Initiates
1
1
Records-rental-of
Rents
1..*
Customer
address
name
phoneNumber
Rents-from
*
1
VideoStore
address
name
phoneNumber
Stocks
1
*
Video
ID
1
*
*
1
Has
1
Maintains
1
*
1
Owns-a
1
Membership
ID
startDate
1
Catalog
1
Described-by
1..*
LoanPolicy
perDayRentalCharge
perDayLateCharge
Defines
1..*
VideoDescription
title
subjectCategory
1
1..*
1
Determines-rental-charge

- The conceptual model of UML can be mastered by learning the following three major elements

1. UML building blocks
2. Rules to connect the building blocks
3. Common mechanisms of UML

1. UML building blocks

The building blocks of UML can be defined as

- Things
- Relationships
- Diagrams

1. UML building blocks - Things

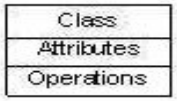
Things are the most important building blocks of UML. Things can be

- Structural
- Behavioral
- Grouping
- Annotational

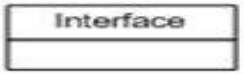
Structural Things

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

Class – Class represents a set of objects having similar responsibilities.

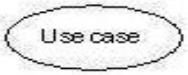


Interface – Interface defines a set of operations, which specify the responsibility of a class.

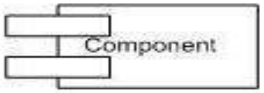


Collaboration –Collaboration defines an interaction between elements.

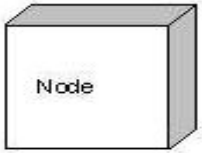
Use case –Use case represents a set of actions performed by a system for a specific goal.



Component –Component describes the physical part of a system.



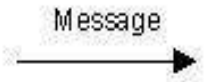
Node – A node can be defined as a physical element that exists at run time.



Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things −

Interaction − Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change

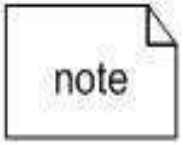
Grouping Things

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

Package − Package is the only one grouping thing available for gathering structural and behavioral things.

Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



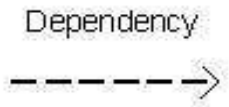
2. UML building blocks - Relationship

- **Relationship**. It shows how the elements are associated with each other and this association describes the functionality of an application.
- There are four kinds of relationships available.
  - **Dependency**
  - **Association**
  - **Generalization**
  - **Realization**

2. UML building blocks - Relationship

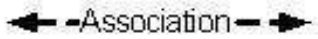
Dependency

Dependency is a relationship between two things in which change in one element also affects the other.



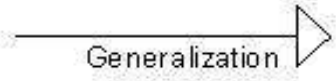
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



3. UML building blocks - **UML Diagrams**

- UML includes the following nine diagrams
  - Class diagram
  - Object diagram
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Activity diagram
  - State chart diagram
  - Deployment diagram
  - Component diagram

UML - Architecture

- UML plays an important role in defining different perspectives of a system namely
  - **Design** of a system consists of classes, interfaces, and collaboration. UML provides class diagram, object diagram to support this.
  - **Implementation** defines the components assembled together to make a complete physical system. UML component diagram is used to support the implementation perspective.
  - **Process** defines the flow of the system. Hence, the same elements as used in Design are also used to support this perspective.
  - **Deployment** represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

UML - Modeling Types

Structural Modeling

Structural modeling captures the static features of a system. They consist of the following –

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.

The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Behavioral Modeling

Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system. They consist of the following –

- Activity diagrams
- Interaction diagrams
- Use case diagrams

All the above show the dynamic sequence of flow in a system.

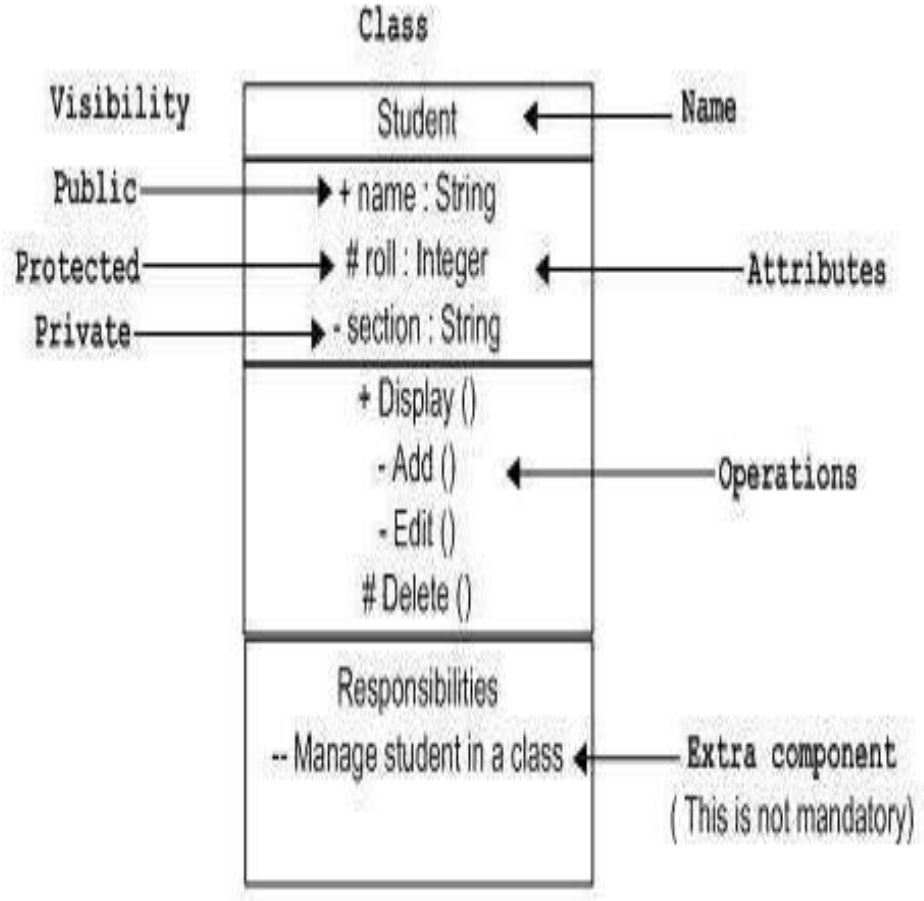
Architectural Modeling

Architectural model represents the overall framework of the system. It contains both structural and behavioral elements of the system. Architectural model can be defined as the blueprint of the entire system. Package diagram comes under architectural modeling.

Class Notation

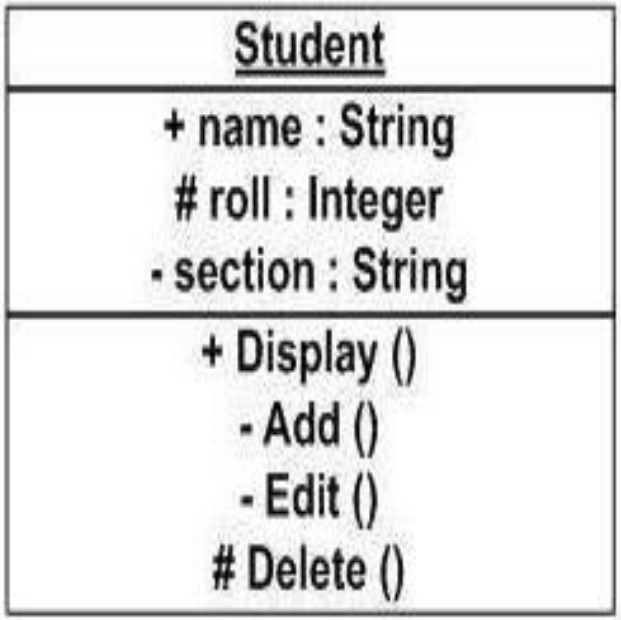
***class* is represented by the following figure**

- The top section is used to name the class.
- The second one is used to show the attributes of the class.
- The third section is used to describe the operations performed by the class.
- The fourth section is optiona to show any additional components.



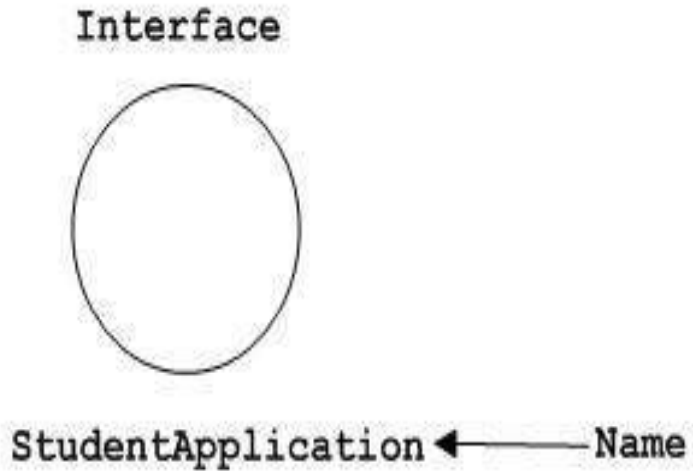
Object Notation

- As the object is an actual implementation of a class, which is known as the instance of a class



Interface Notation

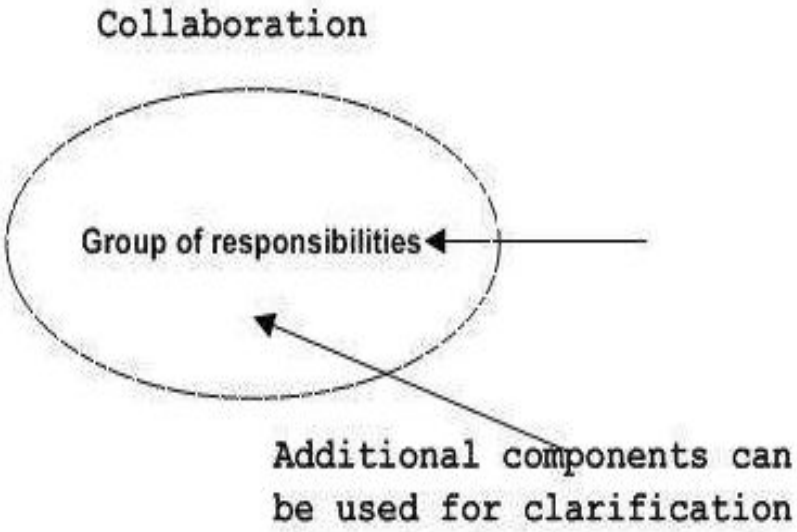
Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.



Interface is used to describe the functionality without implementation. Interface is just like a template where you define different functions, not the implementation. When a class implements the interface, it also implements the functionality as per requirement.

Collaboration Notation

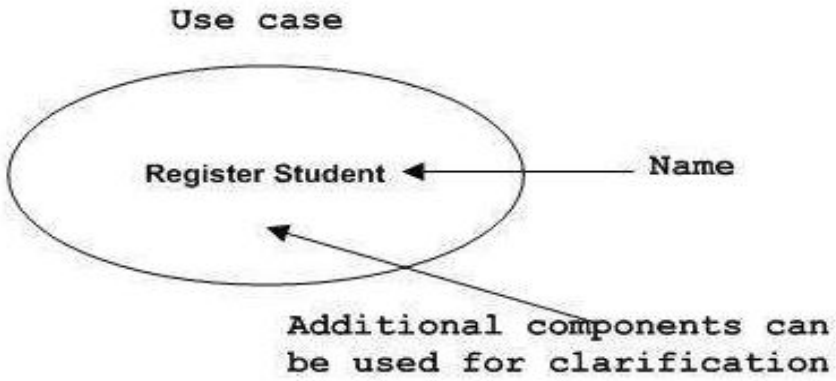
Collaboration is represented by a dotted eclipse as shown in the following figure. It has a name written inside the eclipse.



Collaboration represents responsibilities. Generally, responsibilities are in a group.

Use Case Notation

Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.



Use case is used to capture high level functionalities of a system.

Actor Notation

An actor can be defined as some internal or external entity that interacts with the system.

Initial State Notation

Initial state is defined to show the start of a process. This notation is used in almost all diagrams.

The usage of Initial State Notation is to show the starting point of a process.

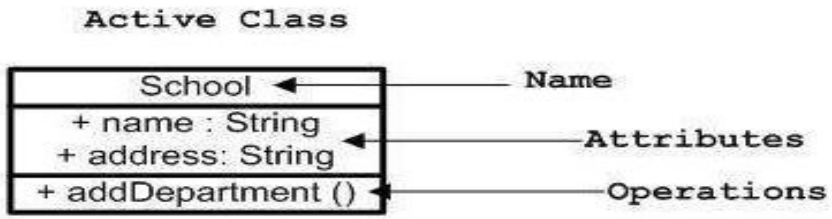
Final State Notation

Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.

The usage of Final State Notation is to show the termination point of a process.

Active Class Notation

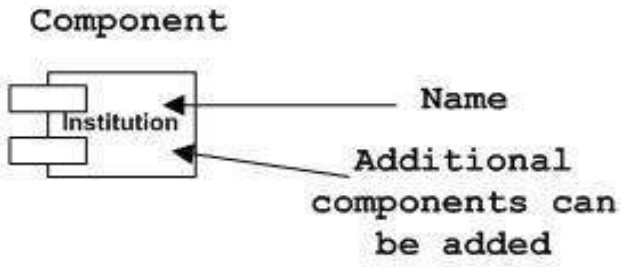
Active class looks similar to a class with a solid border. Active class is generally used to describe the concurrent behavior of a system.



Active class is used to represent the concurrency in a system.

Component Notation

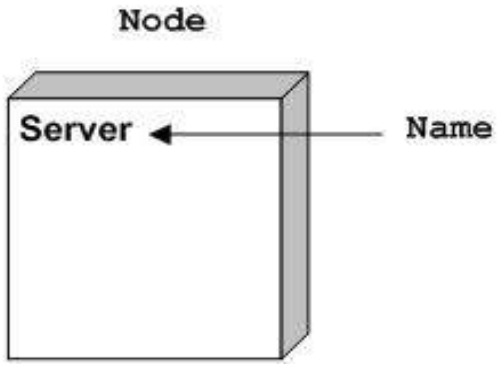
A component in UML is shown in the following figure with a name inside. Additional elements can be added wherever required.



Component is used to represent any part of a system for which UML diagrams are made.

Node Notation

A node in UML is represented by a square box as shown in the following figure with a name. A node represents the physical component of the system.



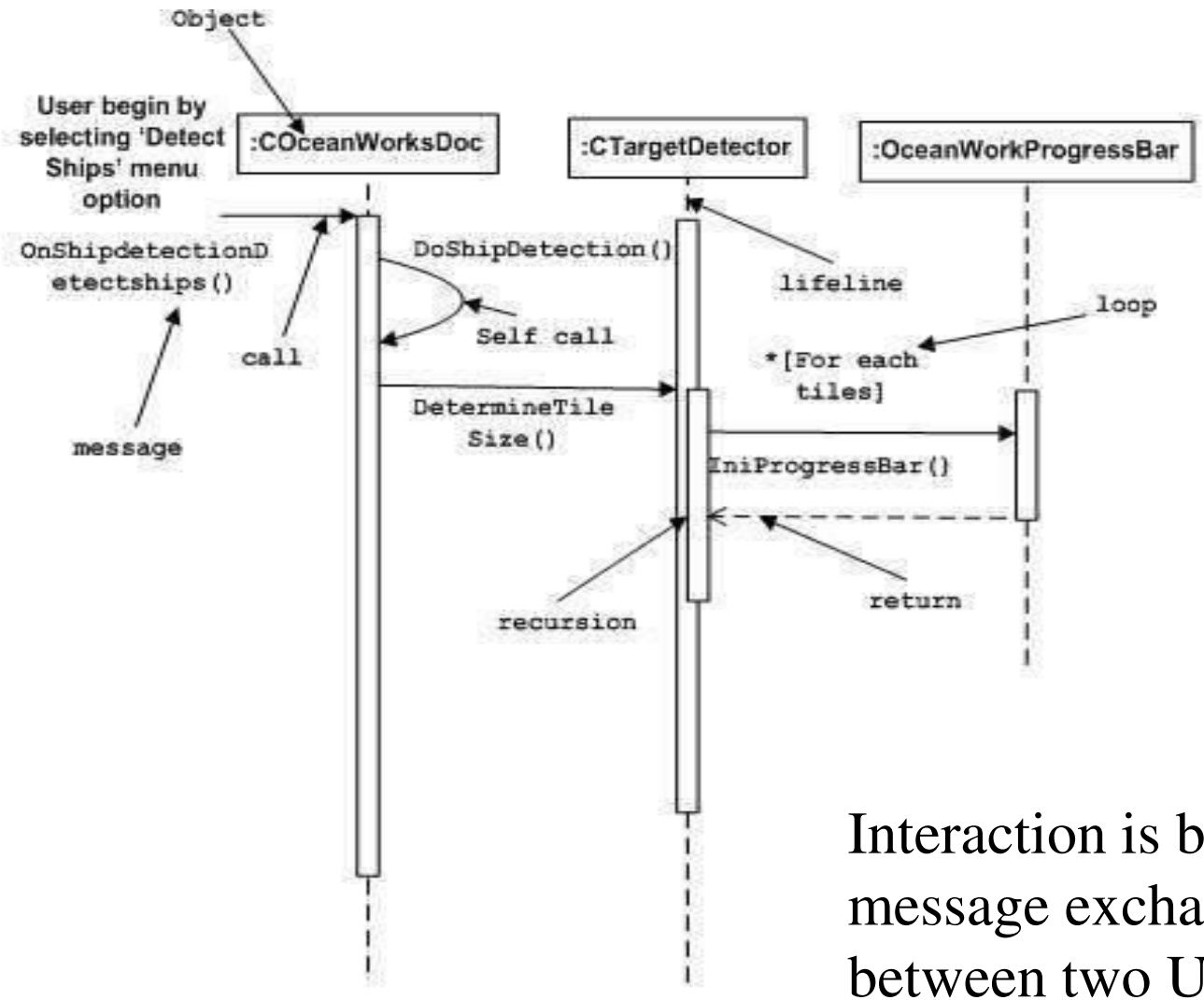
Behavioral Things

- Dynamic parts are one of the most important elements in UML.
- UML has a set of powerful features to represent the dynamic part of software and non-software systems.
- These features include *interactions* and *state machines*.

Interactions can be of two types −

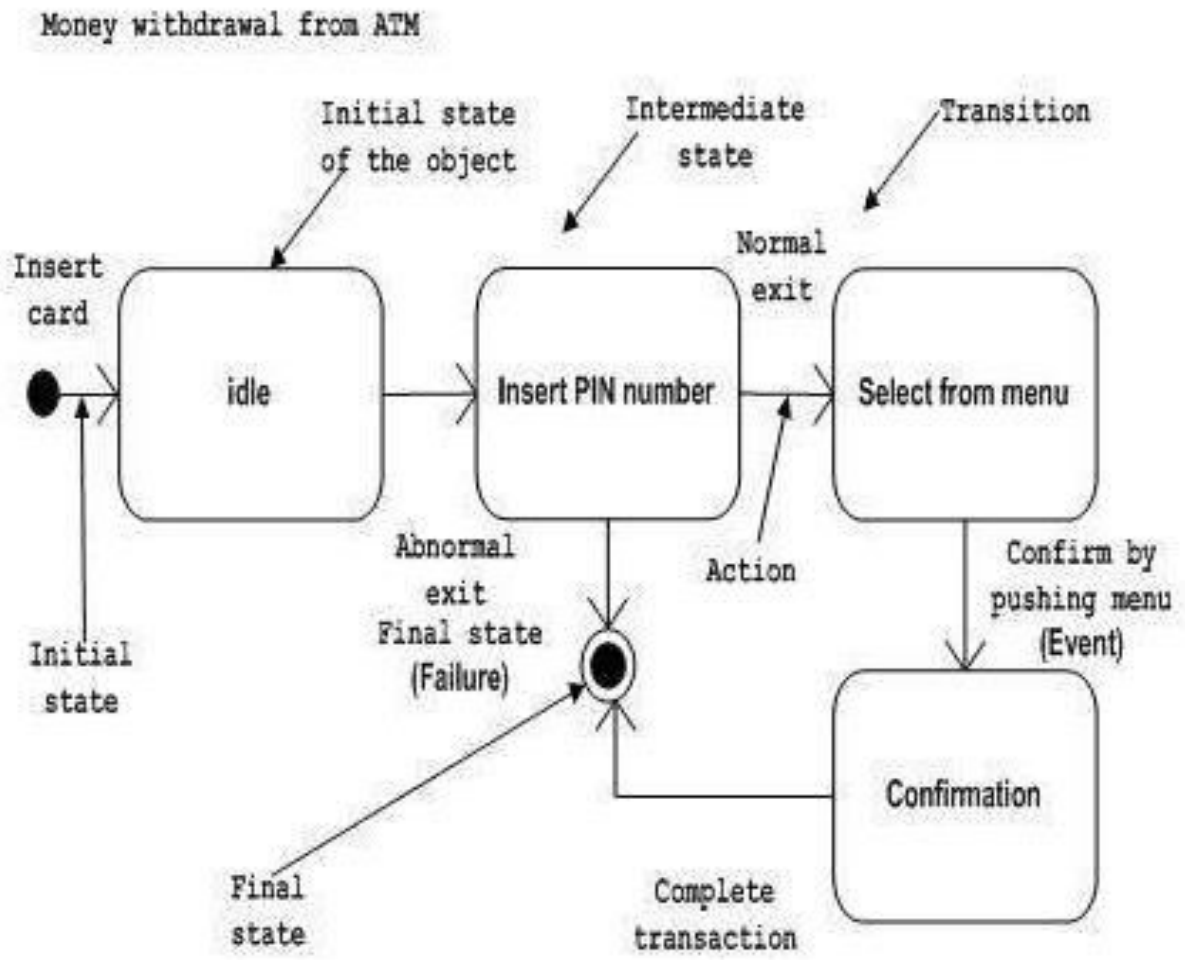
- Sequential (Represented by sequence diagram)
- Collaborative (Represented by collaboration diagram)

Interaction Notation



Interaction is basically a message exchange between two UML components

State Machine Notation



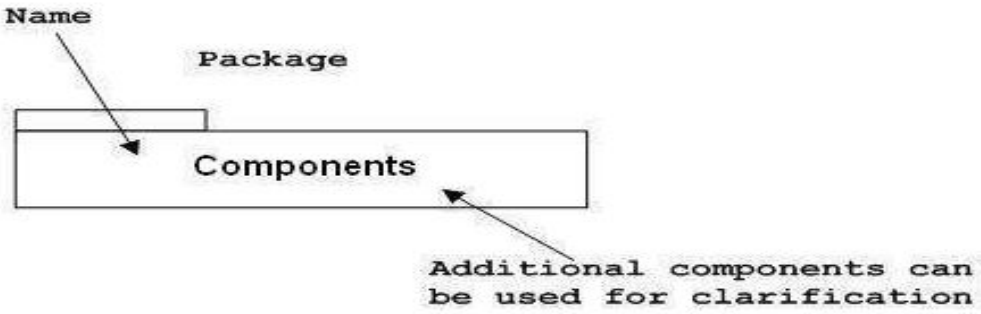
Describe different states of system component (active , idle)

Grouping Things

Organizing the UML models is one of the most important aspects of the design. In UML, there is only one element available for grouping and that is package.

Package Notation

Package notation is shown in the following figure and is used to wrap the components of a system.

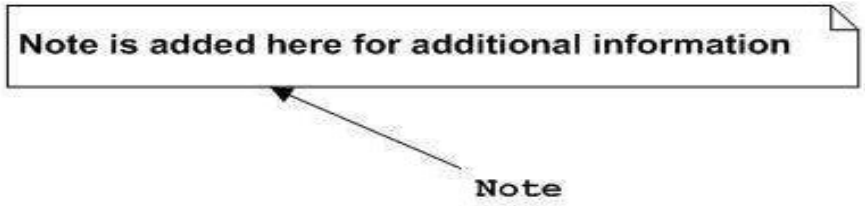


Annotational Things

In any diagram, explanation of different elements and their functionalities are very important. Hence, UML has *notes* notation to support this requirement.

Note Notation

This notation is shown in the following figure. These notations are used to provide necessary information of a system.



Relationships

- A model is not complete unless the relationships between elements are described properly.
- The *Relationship* gives a proper meaning to a UML model.

Following are the different types of relationships available in UML

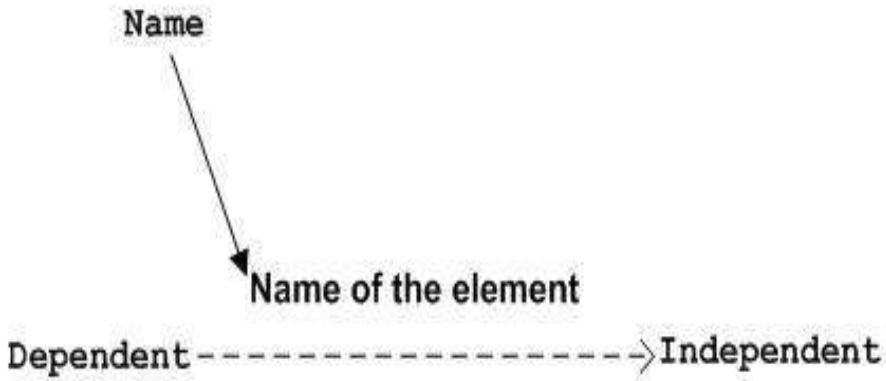
- Dependency
- Association
- Generalization
- Extensibility

Relationships

Dependency Notation

Dependency is an important aspect in UML elements. It describes the dependent elements and the direction of dependency.

Dependency is represented by a dotted arrow as shown in the following figure. The arrow head represents the independent element and the other end represents the dependent element.



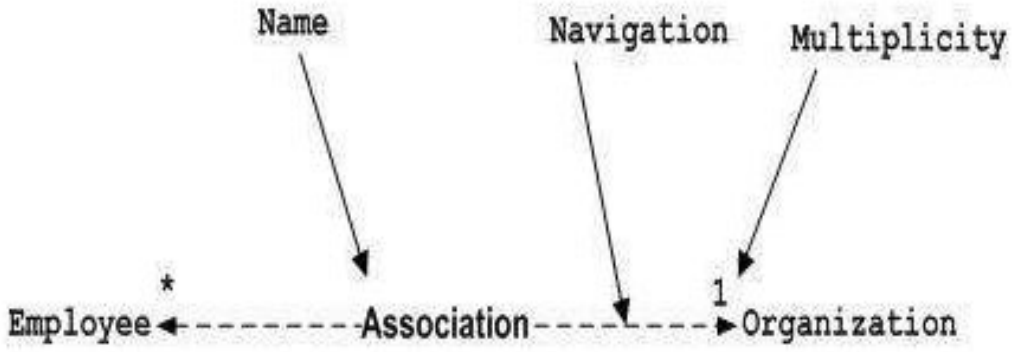
Dependency is used to represent the dependency between two elements of a system

Relationships

Association Notation

Association describes how the elements in a UML diagram are associated. In simple words, it describes how many elements are taking part in an interaction.

Association is represented by a dotted line with (without) arrows on both sides. The two ends represent two associated elements as shown in the following figure. The multiplicity is also mentioned at the ends (1, *, etc.) to show how many objects are associated.



Association is used to represent the relationship between two elements of a system.

Relationships

Generalization Notation

Generalization describes the inheritance relationship of the object-oriented world. It is a parent and child relationship.

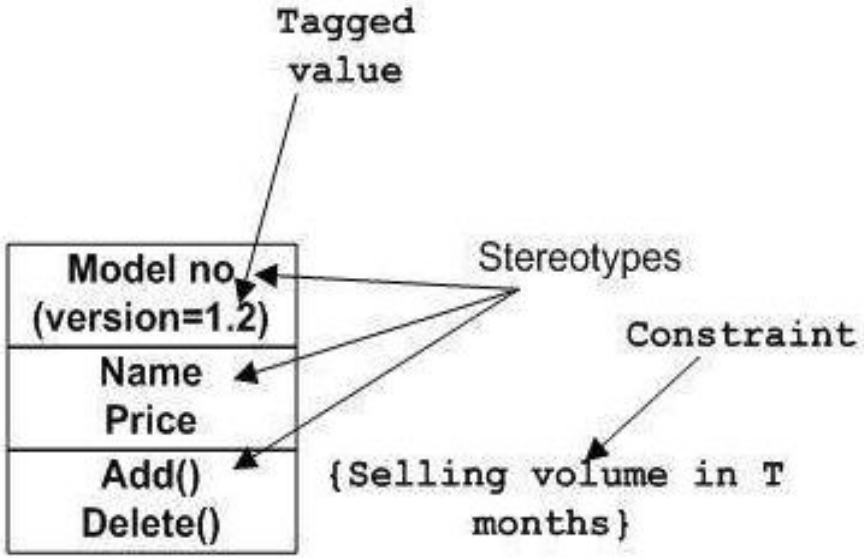
Generalization is represented by an arrow with a hollow arrow head as shown in the following figure. One end represents the parent element and the other end represents the child element.

Generalization is used to describe parent-child relationship of two elements of a system.

Extensibility Notation

All the languages (programming or modeling) have some mechanism to extend its capabilities such as syntax, semantics, etc. UML also has the following mechanisms to provide extensibility features.

- Stereotypes (Represents new elements)
- Tagged values (Represents new attributes)
- Constraints (Represents the boundaries)



Extensibility notations are used to enhance the power of the language. It is basically additional elements used to represent some extra behavior of the system. These extra behaviors are not covered by the standard available notations.

CLASS DIAGRAM

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

The following points should be remembered while drawing a class diagram −

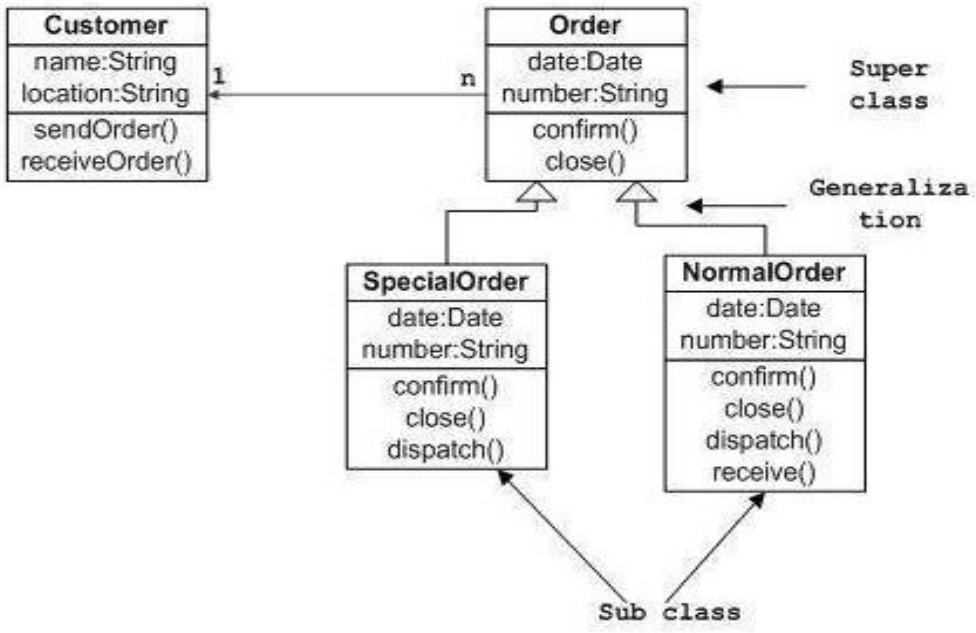
- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
- Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.

Sample Class Diagram



Where to Use Class Diagrams?

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system but they are also used to construct the executable code for forward and reverse engineering of any system.

Generally, UML diagrams are not directly mapped with any object-oriented programming languages but the class diagram is an exception.

Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

In a nutshell it can be said, class diagrams are used for −

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

OBJECT DIAGRAM

Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram.

Object diagrams are used to render a set of objects and their relationships as an instance.

Purpose of Object Diagrams

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.

The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.

It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.

The purpose of the object diagram can be summarized as −

- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective

How to Draw an Object Diagram?

We have already discussed that an object diagram is an instance of a class diagram. It implies that an object diagram consists of instances of things used in a class diagram.

So both diagrams are made of same basic elements but in different form. In class diagram elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

To capture a particular system, numbers of class diagrams are limited. However, if we consider object diagrams then we can have unlimited number of instances, which are unique in nature. Only those instances are considered, which have an impact on the system.

From the above discussion, it is clear that a single object diagram cannot capture all the necessary instances or rather cannot specify all the objects of a system. Hence, the solution is –

- First, analyze the system and decide which instances have important data and association.
- Second, consider only those instances, which will cover the functionality.
- Third, make some optimization as the number of instances are unlimited.

Before drawing an object diagram, the following things should be remembered and understood clearly –

- Object diagrams consist of objects.
- The link in object diagram is used to connect objects.
- Objects and links are the two elements used to construct an object diagram.

After this, the following things are to be decided before starting the construction of the diagram –

- The object diagram should have a meaningful name to indicate its purpose.
- The most important elements are to be identified.
- The association among objects should be clarified.
- Values of different elements need to be captured to include in the object diagram.
- Add proper notes at points where more clarity is required.

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

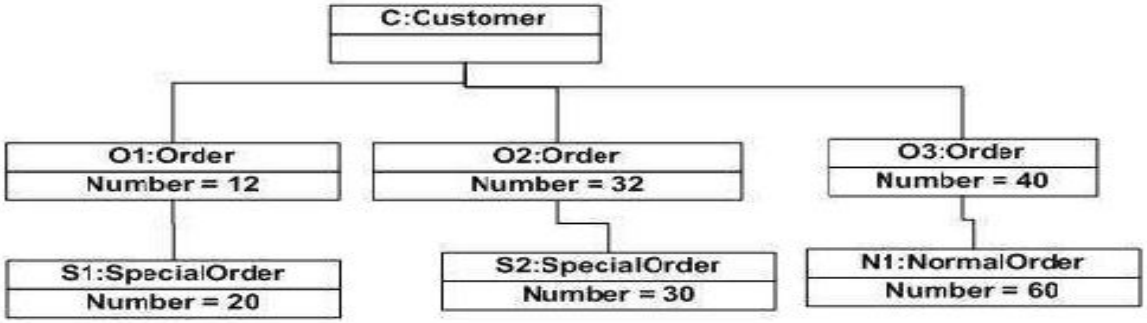
The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

For orders, the values are 12, 32, and 40 which implies that the objects have these values for a particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured

The same is true for special order and normal order objects which have number of orders as 20, 30, and 60. If a different time of purchase is considered, then these values will change accordingly.

The following object diagram has been drawn considering all the points mentioned above

Object diagram of an order management system



Where to Use Object Diagrams?

Object diagrams can be imagined as the snapshot of a running system at a particular moment. Let us consider an example of a running train

Now, if you take a snap of the running train then you will find a static picture of it having the following –

- A particular state which is running.
- A particular number of passengers. which will change if the snap is taken in a different time

Here, we can imagine the snap of the running train is an object having the above values. And this is true for any real-life simple or complex system.

In a nutshell, it can be said that object diagrams are used for −

- Making the prototype of a system.
- Reverse engineering.
- Modeling complex data structures.
- Understanding the system from practical perspective.

Component diagrams

- Component diagrams are different in terms of nature and behavior.
- Component diagrams are used to model the physical aspects of a system.
- Now the question is, what are these physical aspects?
  - Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.
- Component diagrams are used to visualize the organization and relationships among components in a system.
- These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to Draw a Component Diagram?

- Component diagrams are used to describe the physical artifacts of a system.
  - This artifact includes files, executables, libraries, etc

- The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

How to Draw a Component Diagram?

Before drawing a component diagram, the following artifacts are to be identified clearly –

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools.
- Use notes for clarifying important points.

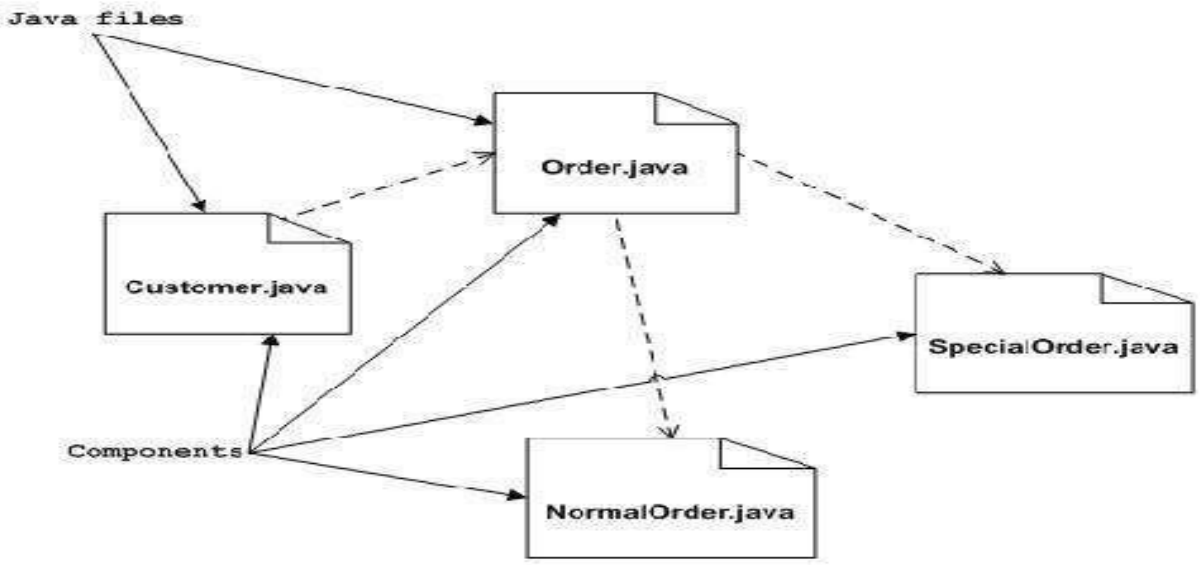
How to Draw a Component Diagram?

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.

Component diagram of an order management system



Where to Use Component Diagrams?

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to –

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.

UML - Deployment Diagrams

- Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.
- Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose of Deployment Diagrams

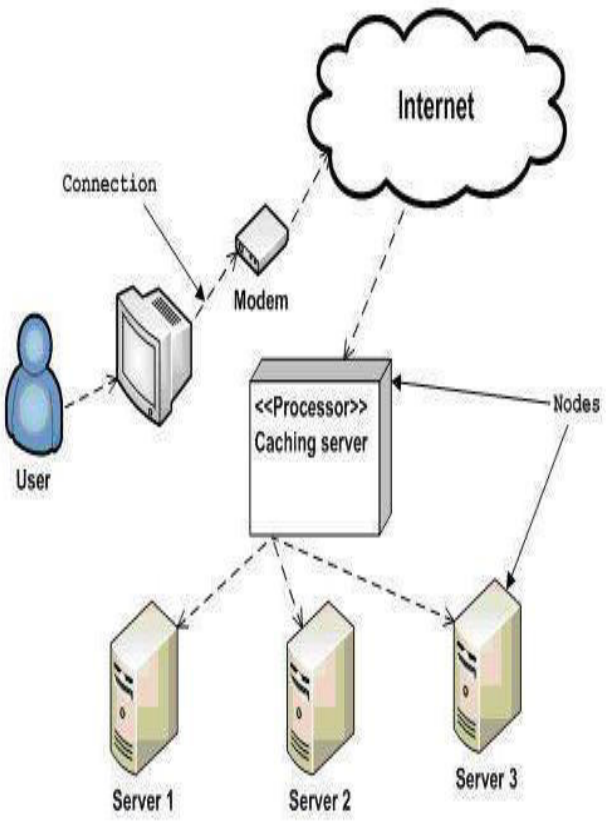
- Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.
- special diagrams used to focus on software and hardware components.
- The purpose of deployment diagrams can be described as
  - Visualize the hardware topology of a system.
  - Describe the hardware components used to deploy software components.
  - Describe the runtime processing nodes.

How to Draw a Deployment Diagram?

- Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −
  - Performance
  - Scalability
  - Maintainability
  - Portability
- Before drawing a deployment diagram, the following artifacts should be identified
  - Nodes
  - Relationships among nodes
- Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −
  - Monitor
  - Modem
  - Caching server
  - Server

How to Draw a Deployment Diagram?

- The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3.
- The user connects to the application using the Internet.
- The control flows from the caching server to the clustered environment.



Where to Use Deployment Diagrams?

- Deployment diagrams are mainly used by system engineers.
- These diagrams are used to describe the physical components (hardware), their distribution, and association

Deployment diagrams can be used −

- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

UML - Use Case Diagrams

- To model a system, the most important aspect is to capture the dynamic behavior.
- Dynamic behavior means the behavior of the system when it is running/operating
- These internal and external agents are known as actors.
- Use case diagrams consists of actors, use cases and their relationships.
- The diagram is used to model the system/subsystem of an application.
- A single use case diagram captures a particular functionality of a system..

Purpose of Use Case Diagrams

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

How to Draw a Use Case Diagram?

- Use case diagrams are considered for high level requirement analysis of a system.
- When the requirements of a system are analyzed, the functionalities are captured in use cases.
- Actors can be defined as something that interacts with the system. Actors can be a human user, some internal applications, or may be some external applications.

When we are planning to draw a use case diagram, we should have the following items identified.

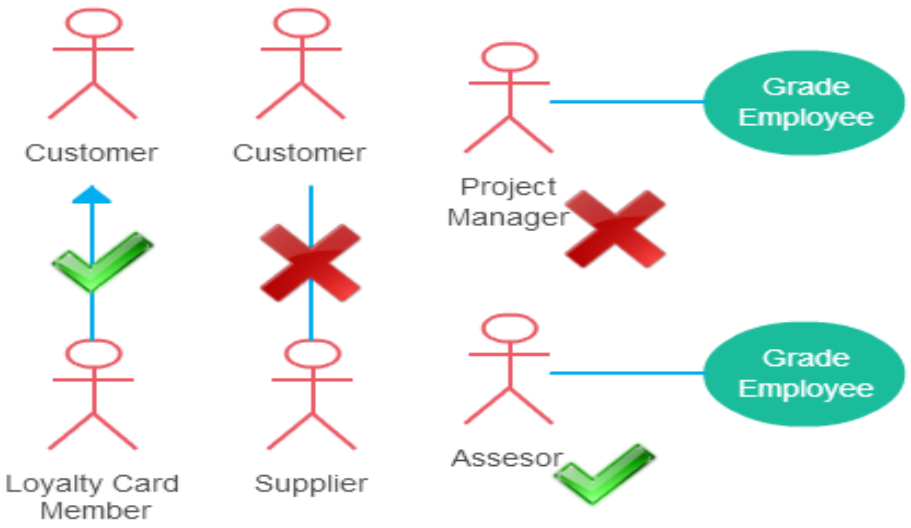
  - Functionalities to be represented as use case
  - Actors
  - Relationships among the use cases and actors.
  - System Boundary (System of interest in relation to the world around it)

How to Draw a Use Case Diagram?

- Use case diagrams are drawn to capture the functional requirements of a system.
- After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram
  - The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
  - Give a suitable name for actors.
  - Show relationships and dependencies clearly in the diagram.
  - Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
  - Use notes whenever required to clarify some important points.

Actors

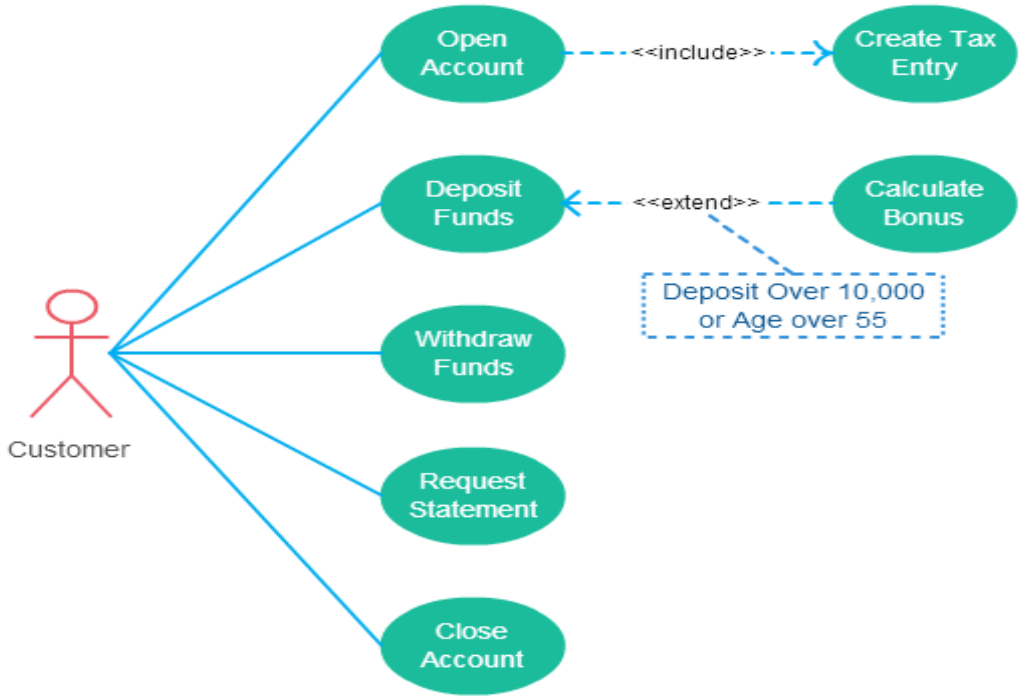
- **Give meaningful business relevant names for actors** – For example, if your use case interacts with an outside organization it's much better to name it with the function rather than the organization name. (Eg: Airline Company is better than PanAir)
- **Primary actors should be to the left side of the diagram** – This enables you to quickly highlight the important roles in the system.
- **Actors model roles (not positions)** – In a hotel both the front office executive and shift manager can make reservations. So something like "Reservation Agent" should be used for actor name to highlight the role.
- **External systems are actors** – If your use case is send-email and if interacts with the email management software then the software is an actor to that particular use case.
- **Actors don't interact with other actors** – In case actors interact within a system you need to create a new use case diagram with the system in the previous diagram represented as an actor.
- **Place inheriting actors below the parent actor** – This is to make it more readable and to quickly highlight the use cases specific for that actor.



Naming

Use Cases

- **Names begin with a verb** – A use case models an action so the name should begin with a verb.
- **Make the name descriptive** – This is to give more information for others who are looking at the diagram. For example "Print Invoice" is better than "Print".
- **Highlight the logical order** – For example, if you're analyzing a bank customer typical use cases include open account, deposit and withdraw. Showing them in the logical order makes more sense.
- **Place included use cases to the right of the invoking use case** – This is done to improve readability and add clarity.
- **Place inheriting use case below parent use case** – Again this is done to improve the readability of the diagram.



UC :Objects

Use Case Diagram objects

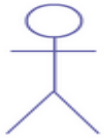
Use case diagrams consist of 4 objects.

- Actor
- Use case
- System
- Package

The objects are further explained below.

Actor

Actor in a use case diagram is **any entity that performs a role** in one given system. This could be a person, organization or an external system and usually drawn like skeleton shown below.



Actor

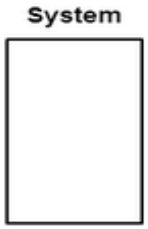
Use Case

A use case **represents a function or an action within the system**. It's drawn as an oval and named with the function.

UC : Objects

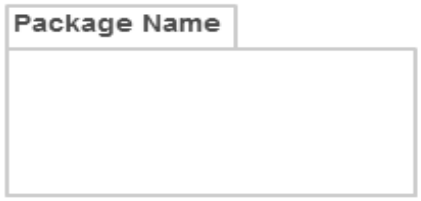
System

The system is used to **define the scope of the use case** and drawn as a rectangle. This an optional element but useful when you're visualizing large systems. For example, you can create all the use cases and then use the system object to define the scope covered by your project. Or you can even use it to show the different areas covered in different releases.



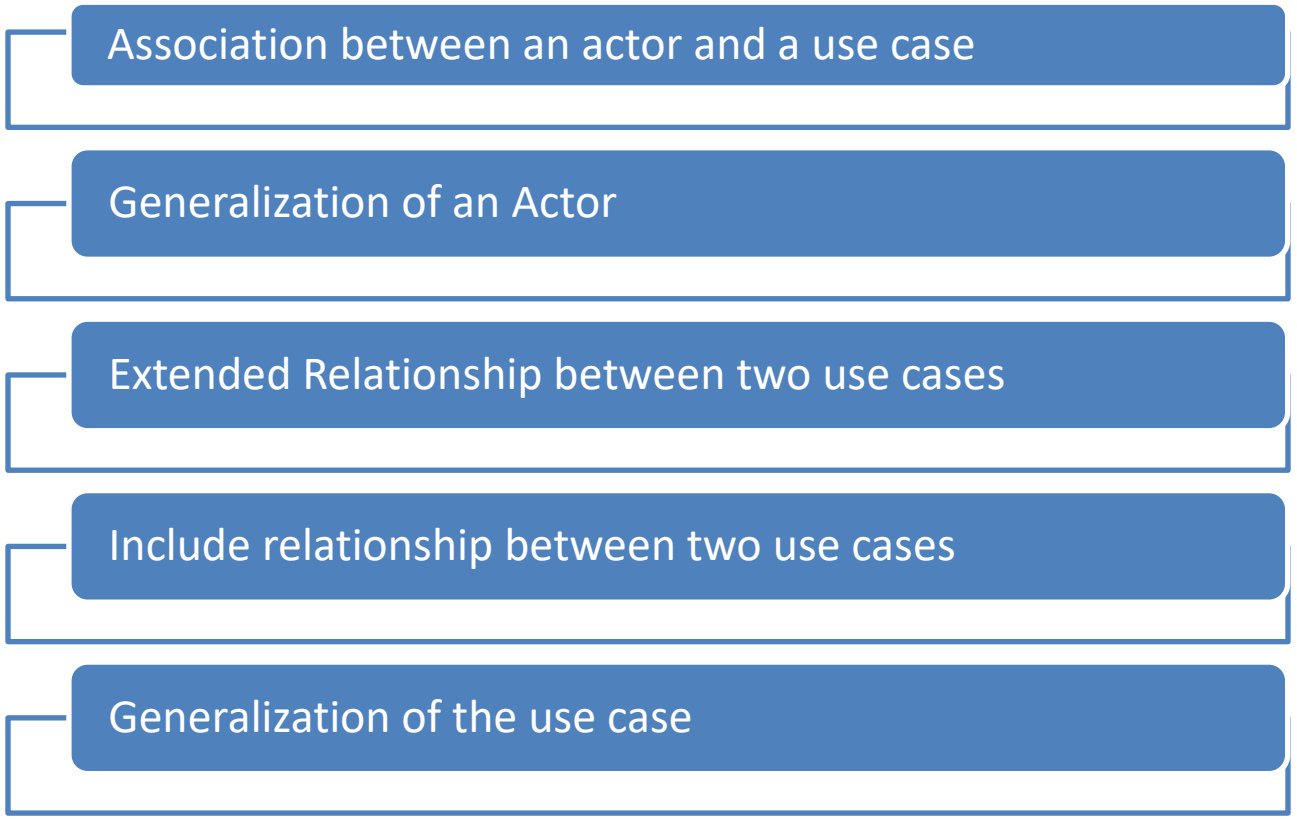
Package

The package is another optional element that is extremely useful in complex diagrams. Similar to class diagrams, packages are **used to group together use cases**. They are drawn like the image shown below.



Relationships in Use Case Diagrams

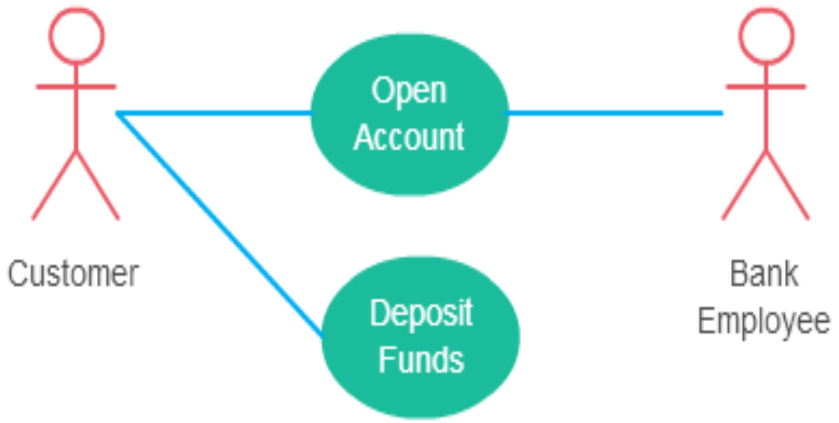
- There are five types of relationships in a use case diagram.



Association Between Actor and Use Case

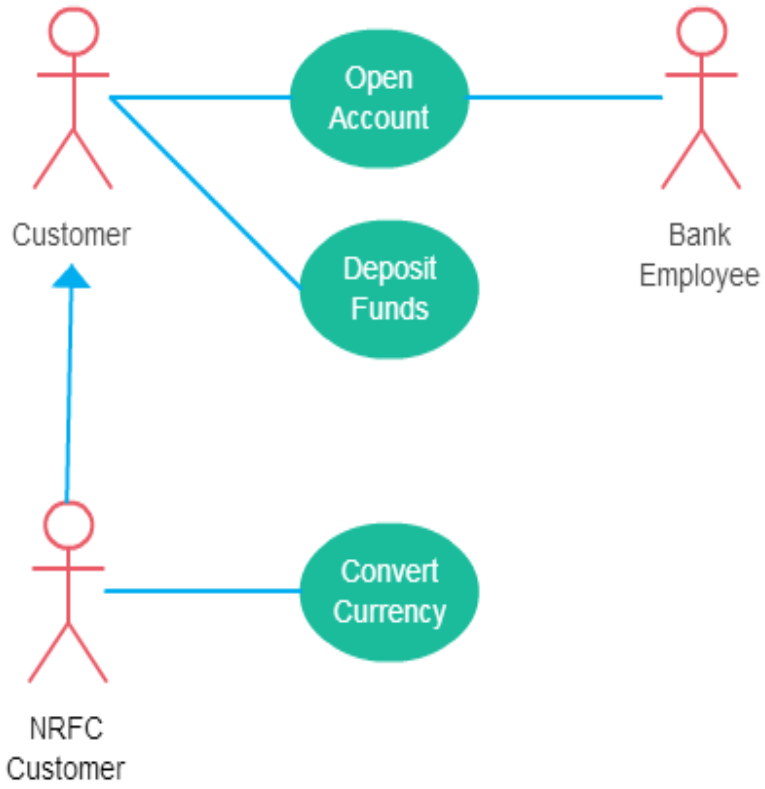
This one is straightforward and present in every use case diagram. Few things to note.

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.



Generalization of an Actor

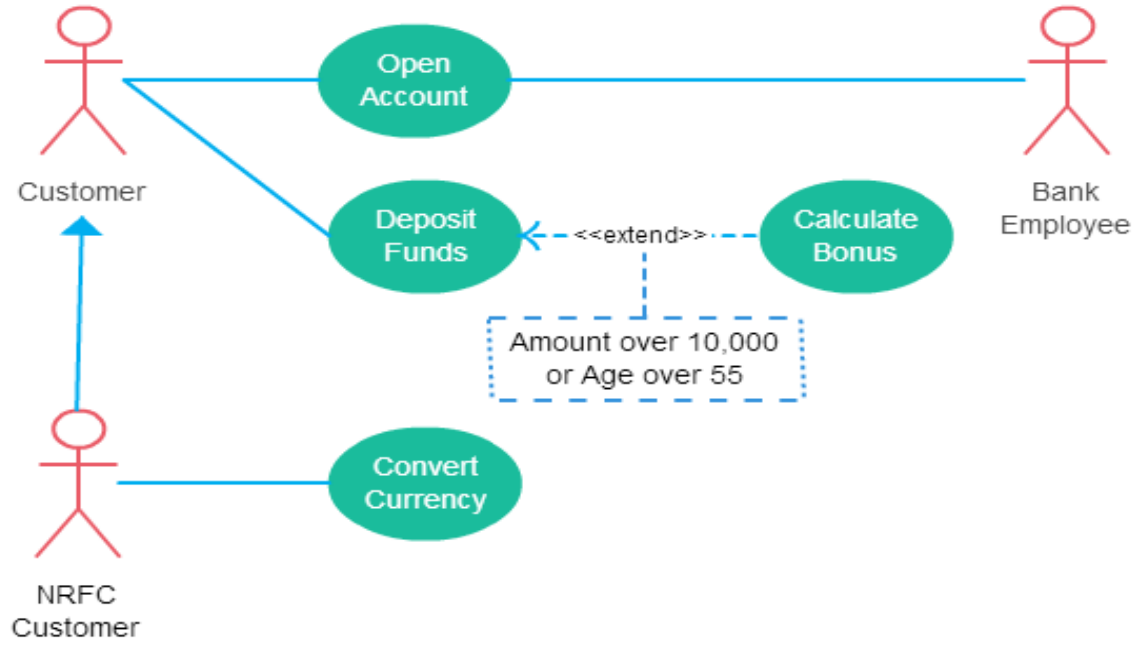
Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role. Let's expand the previous use case diagram to show the generalization of an actor.



Extend Relationship Between Two Use Cases

- **The extending use case is dependent on the extended (base) use case**. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- **The extending use case is usually optional** and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- **The extended (base) use case must be meaningful on its own**. This means it should be independent and must not rely on the behavior of the extending use case.

Lets expand our current example to show the <<extend>> relationship.

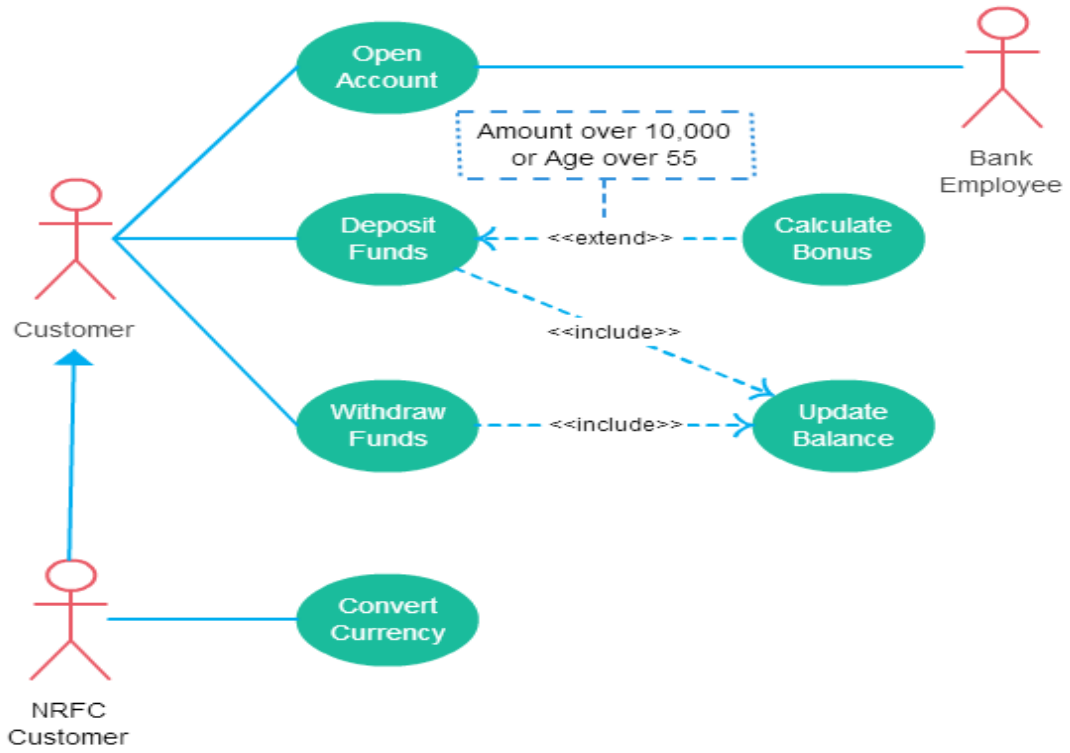


Include Relationship Between Two Use Cases

Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

Lest expand our banking system use case diagram to show include relationships as well.



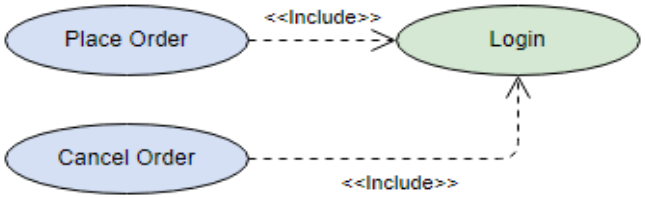
Generalization of a Use Case

- This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.
- For example, in the previous banking example, there might be a use case called “Pay Bills”. This can be generalized to “Pay by Credit Card”, “Pay by Bank Balance” etc

Structuring Use Cases

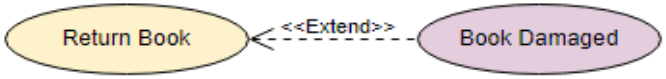
<<include>> Use Case

The time to use the <<include>> relationship is after you have completed the first cut description of all your main Use Cases. You can now look at the Use Cases and identify common sequences of user-system interaction.



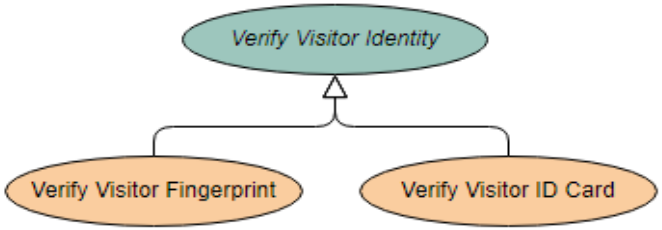
<<extend>> Use Case

An extending use case is, effectively, an alternate course of the base use case. The <<extend>> use case accomplishes this by conceptually inserting additional action sequences into the base use-case sequence.



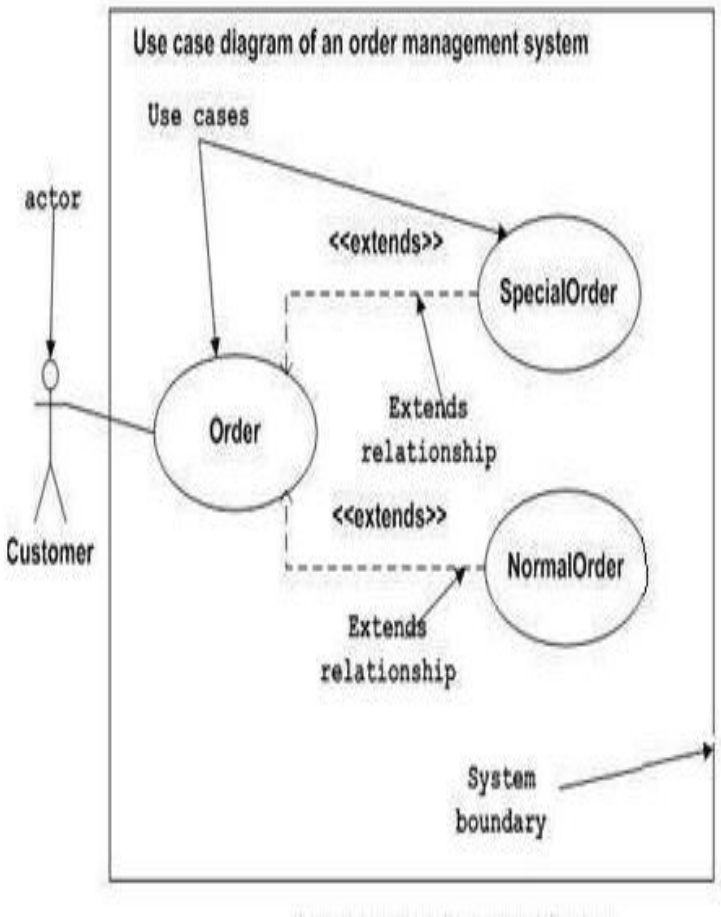
Abstract and generalized Use Case

The general use case is abstract. It can not be instantiated, as it contains incomplete information. The title of an abstract use case is shown in italics.



How to Draw a Use Case Diagram?

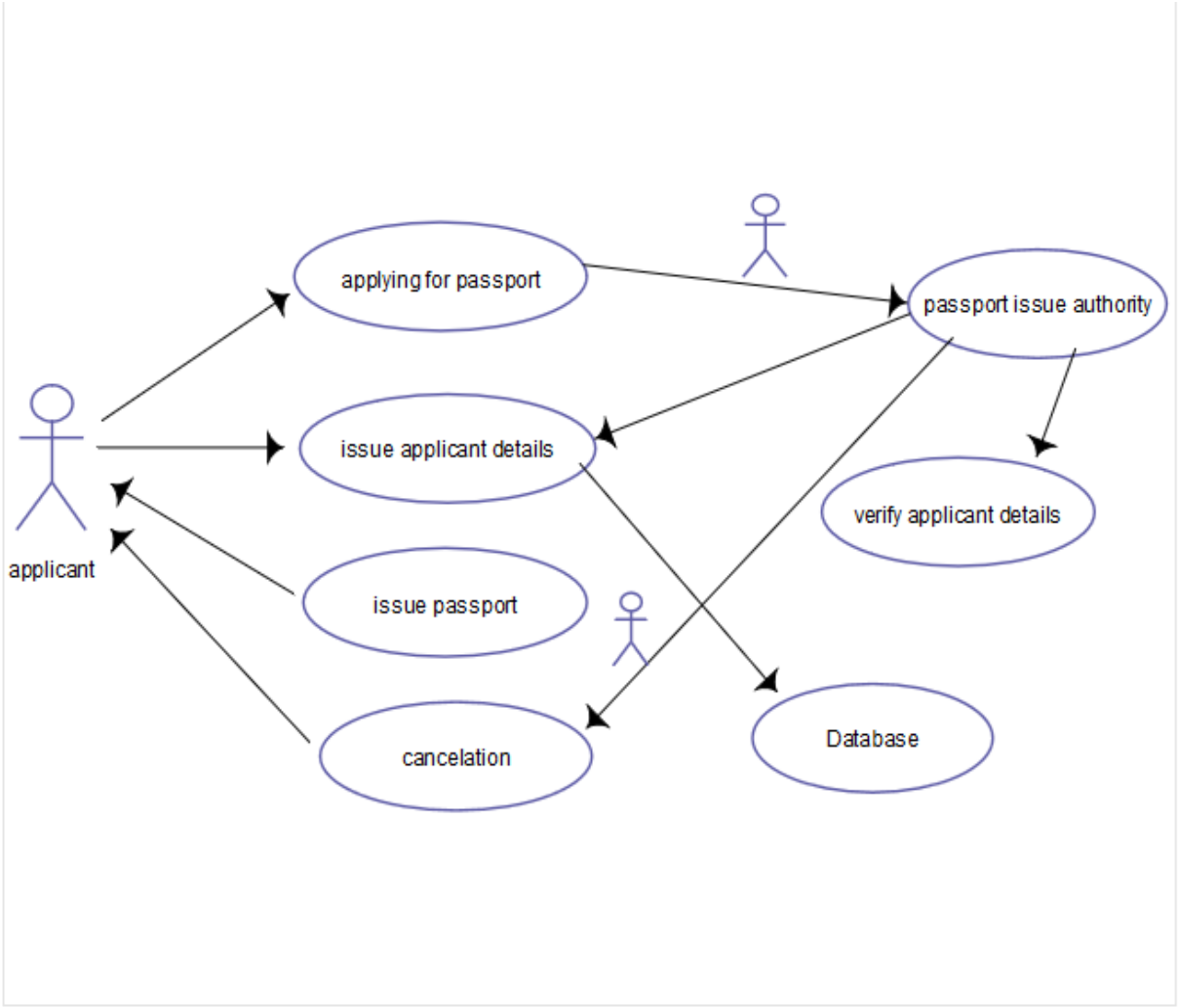
- Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.



Where to Use a Use Case Diagram?

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

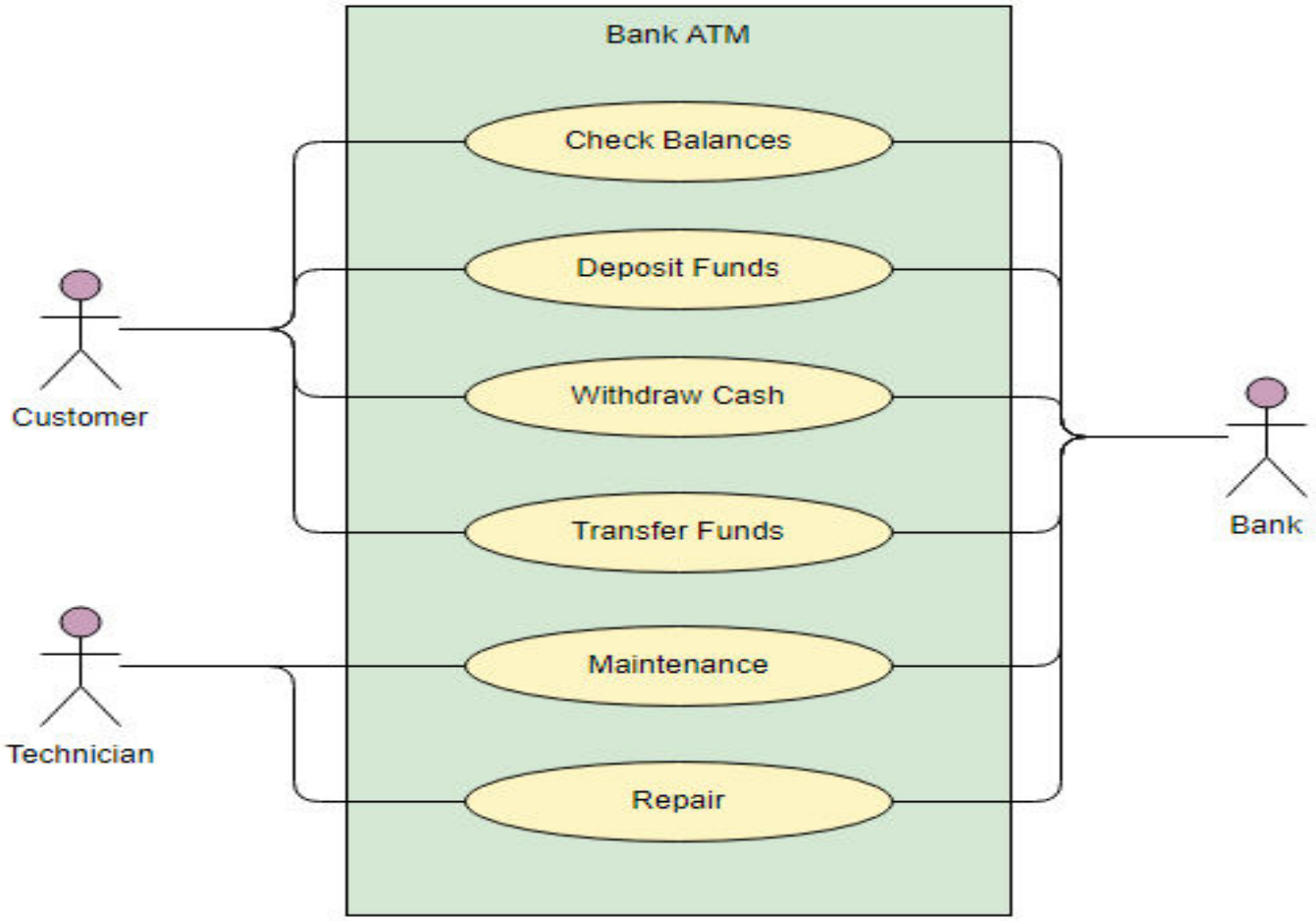
Online passport system



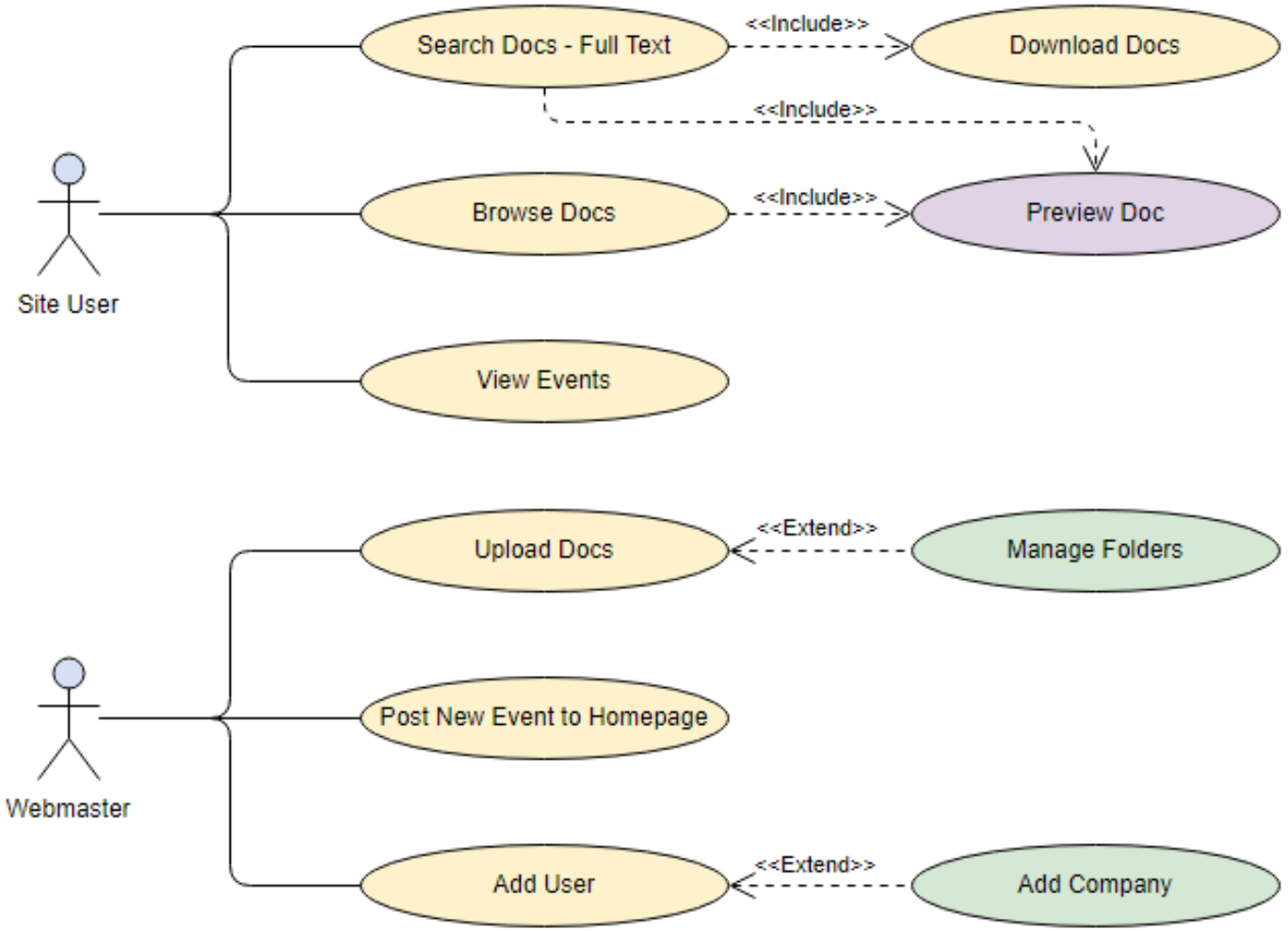
Business USE CASE

- A business use case is described in **technology-free terminology** which treats the business process as a black box and describes the business process that is used by its business actors, while an ordinary use case is normally described at the **system functionality level** and specifies the function or the service that the system provides for the user.
- Business use case represents how the work to be done manually in the currently situation and it is not necessarily done by the system or intend to be automated in the scope of target system.

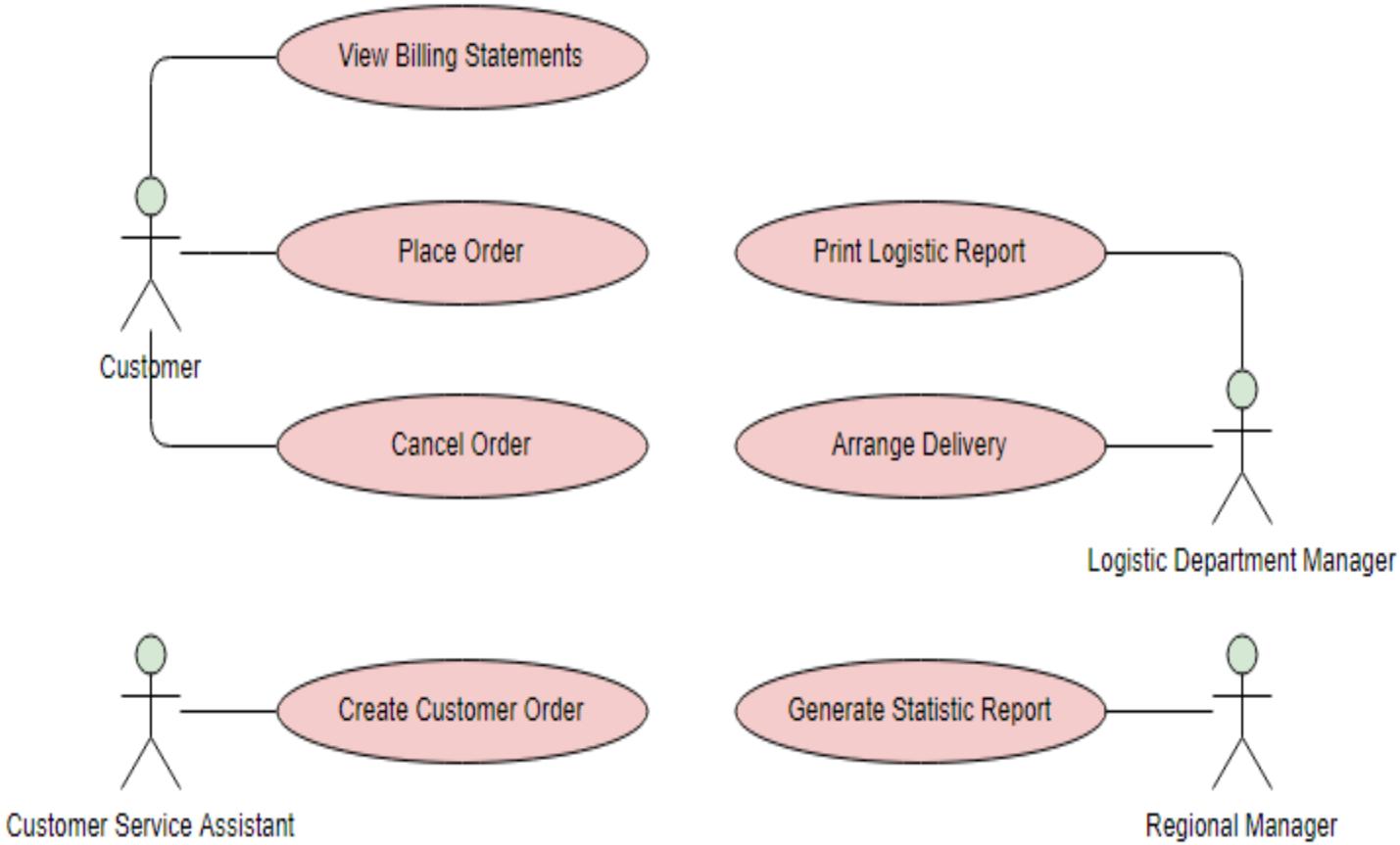
Examples : **ATM** use case diagram



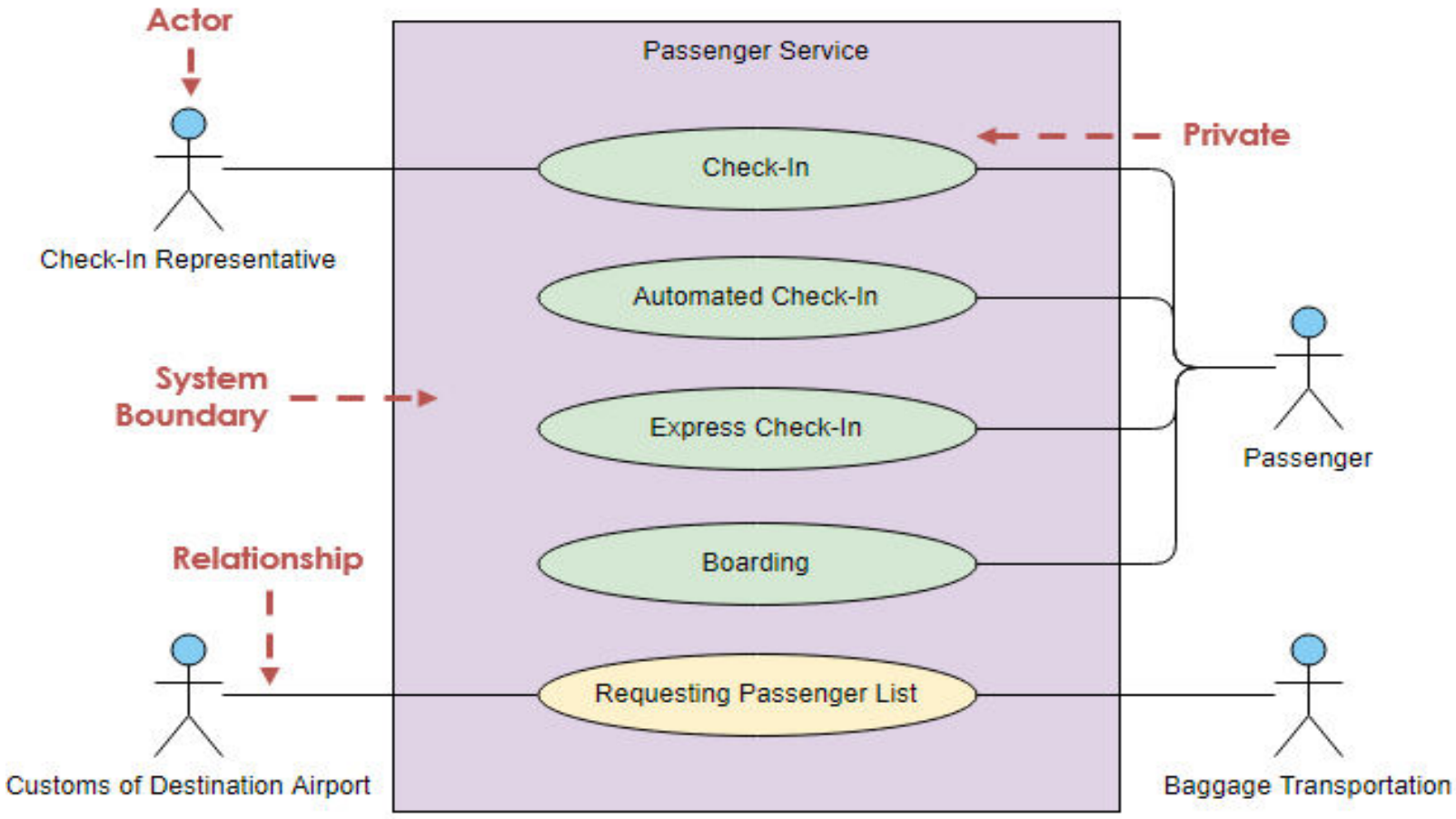
Document Management System (DMS)



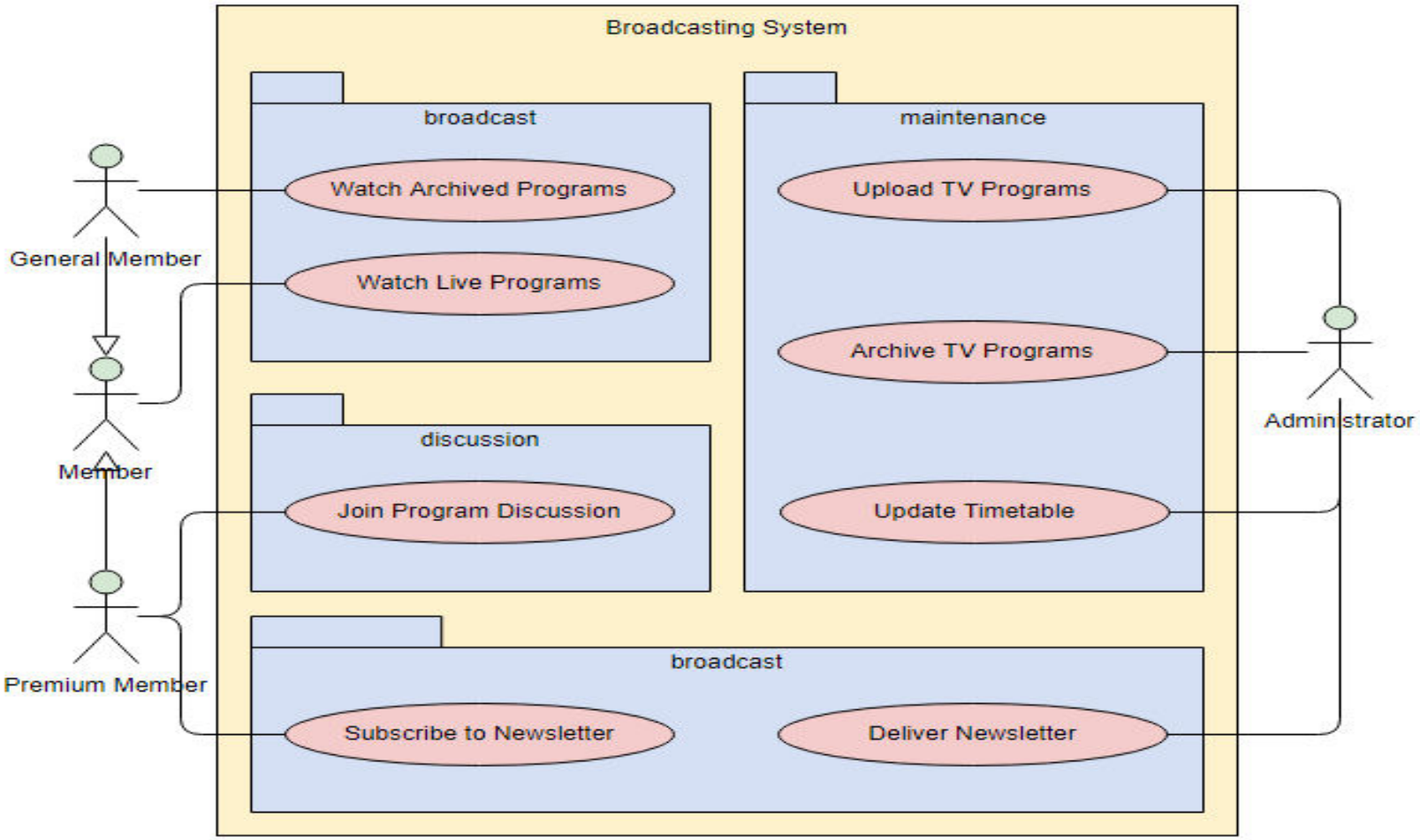
Order System



Passenger service - airport



Draw packages for logical categorization of use cases into related subsystems.



UML - Interaction Diagrams

- This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**.
- Sequence diagram emphasizes on time **sequence of messages** and collaboration diagram emphasizes on the **structural organization of the objects** that send and receive messages.

UML - Interaction Diagrams

- The purpose of interaction diagram is −
  - To capture the dynamic behavior of a system.
  - To describe the message flow in the system.
  - To describe the structural organization of the objects.
  - To describe the interaction among objects.
- Following things are to be identified clearly before drawing the interaction diagram
  - Objects taking part in the interaction.
  - Message flows among the objects.
  - The sequence in which the messages are flowing.
  - Object organization.

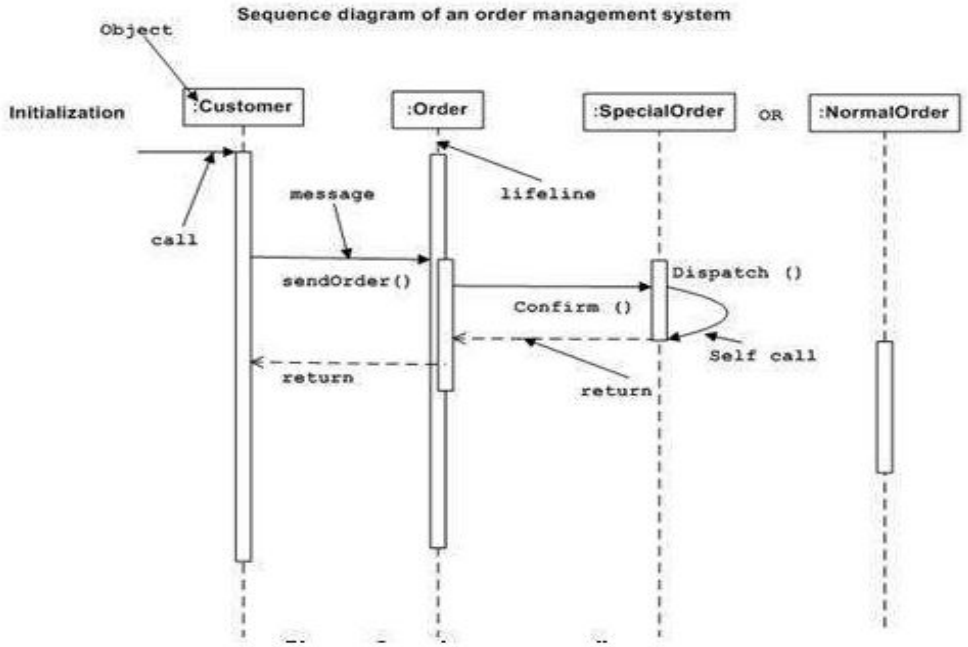
UML - Interaction Diagrams

The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



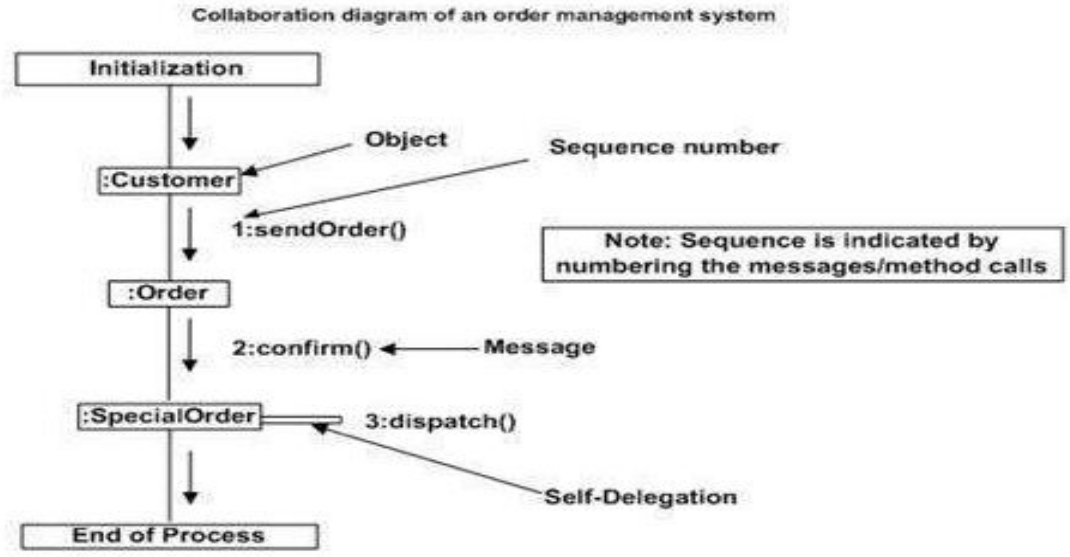
UML - Interaction Diagrams

The Collaboration Diagram

The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



Interaction diagrams can be used –

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

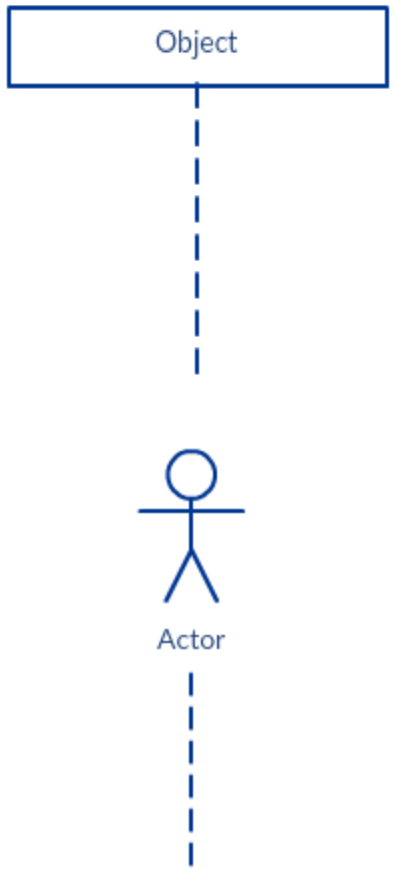
Sequence Diagram Notations

- A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions

. Each object has a column and the messages exchanged between them are represented by arrows.

Sequence Diagram :Lifeline Notation

- A sequence diagram is made up of several of these lifeline notations that should be arranged horizontally across the top of the diagram.
- No two lifeline notations should overlap each other.
- They represent the different objects or parts that interact with each other in the system during the sequence.
- A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.



Sequence Diagram :Lifeline Notation

A lifeline with an entity element represents system data. For example, in a customer service application, the Customer entity would manage all data related to a customer.

Sequence Diagram :Lifeline Notation

A lifeline with a boundary element indicates a system boundary/ software element in a system; for example, user interface screens, database gateways or menus that users interact with, are boundaries.

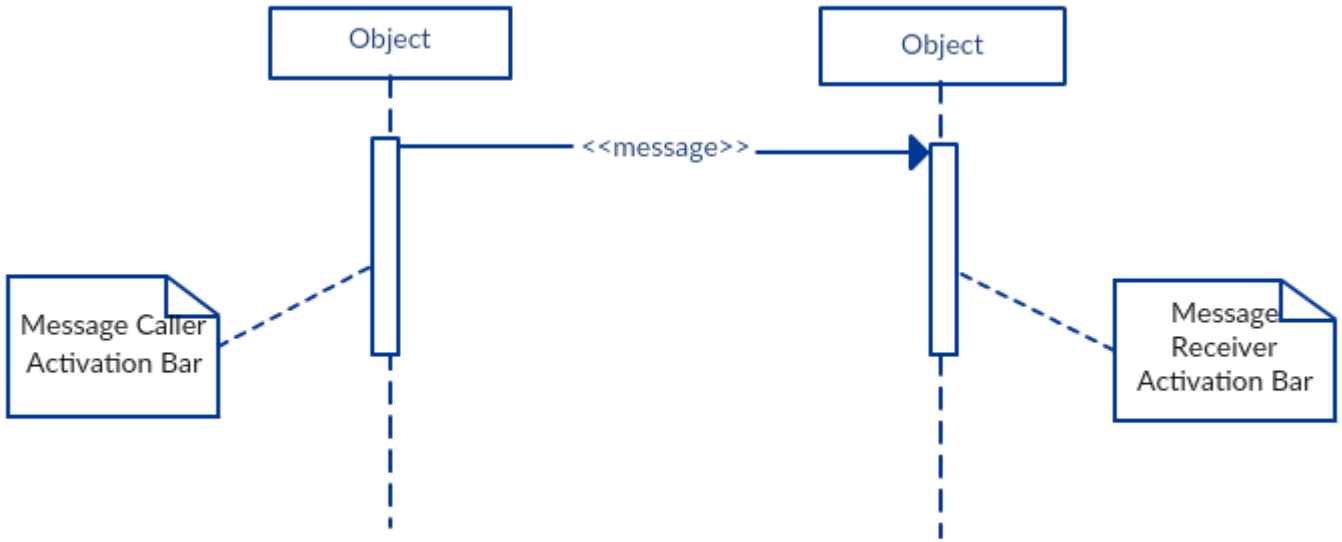
Sequence Diagram :Lifeline Notation

And a lifeline with a control element indicates a controlling entity or manager. It organizes and schedules the interactions between the boundaries and entities and serves as the mediator between them.

Activation Bars

- Activation bar is the box placed on the lifeline. It is used to indicate that an object is active (or instantiated) during an interaction between two objects.
- The length of the rectangle indicates the duration of the objects staying active.
- In a sequence diagram, an interaction between two objects occurs when one object sends a message to another.
- The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message

Activation Bars



Message Arrows

- An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.
- A message can flow in any direction; from left to right, right to left or back to the Message Caller itself.
- The message arrow comes with a description, which is known as a message signature, on it.

attribute = message_name (arguments): return_type

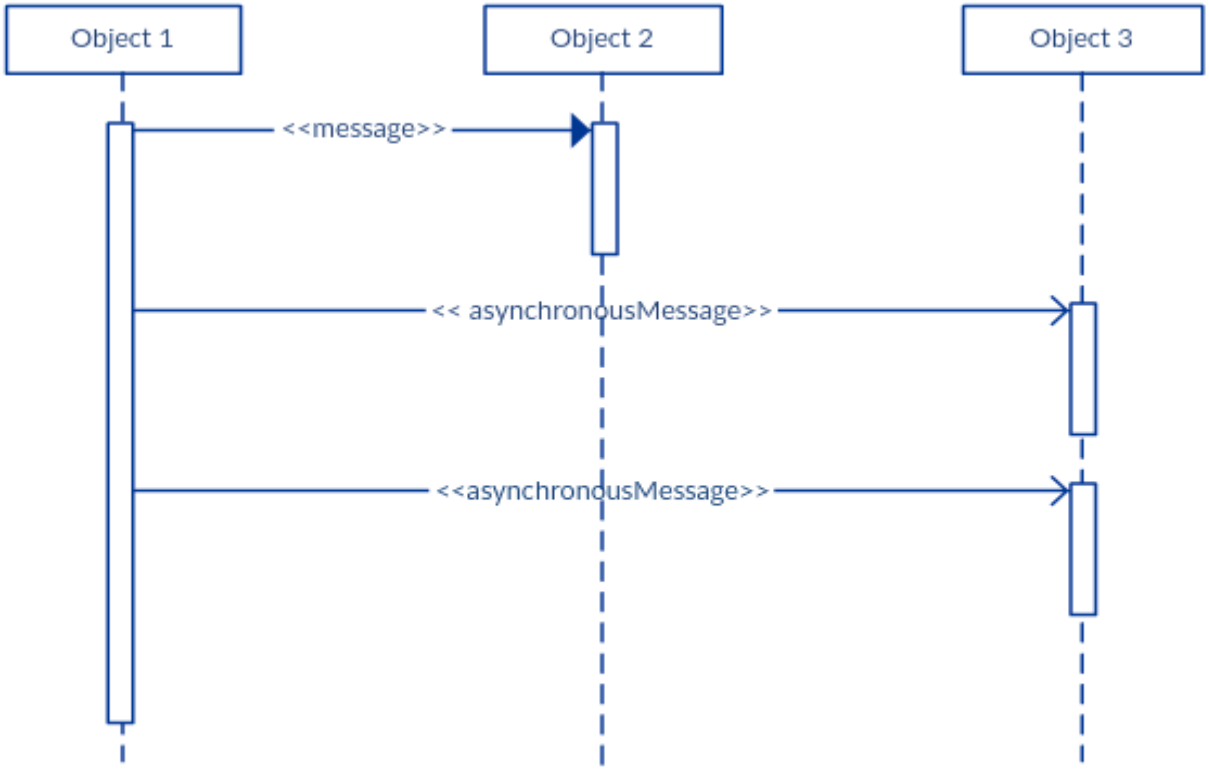
Note : All parts except the message_name are optional.

Message Arrows

- ***Synchronous message*** is used when the sender waits for the receiver to process the message and return before carrying on with another message.
- The arrowhead used to indicate this type of message is a solid one, like the one below.

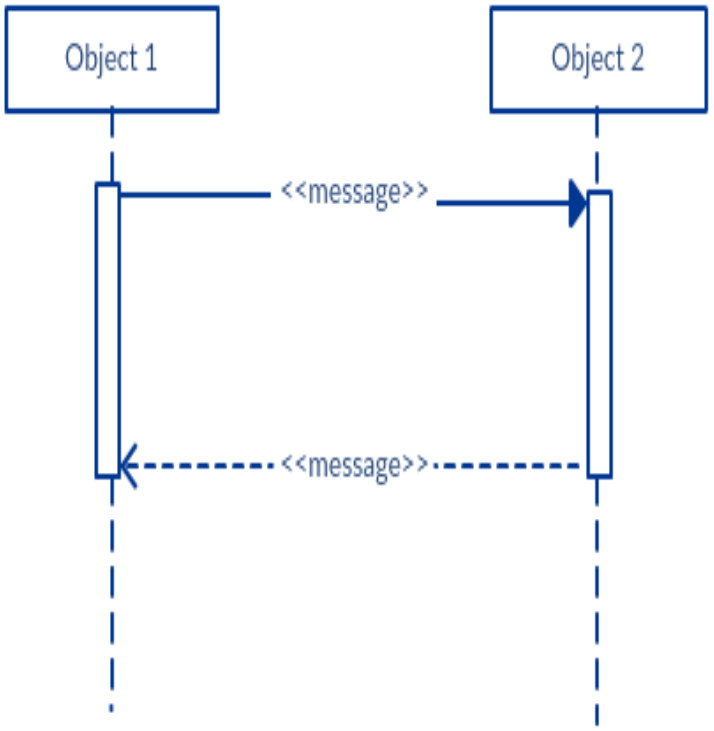
- ***Asynchronous message*** is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system

Message Arrows



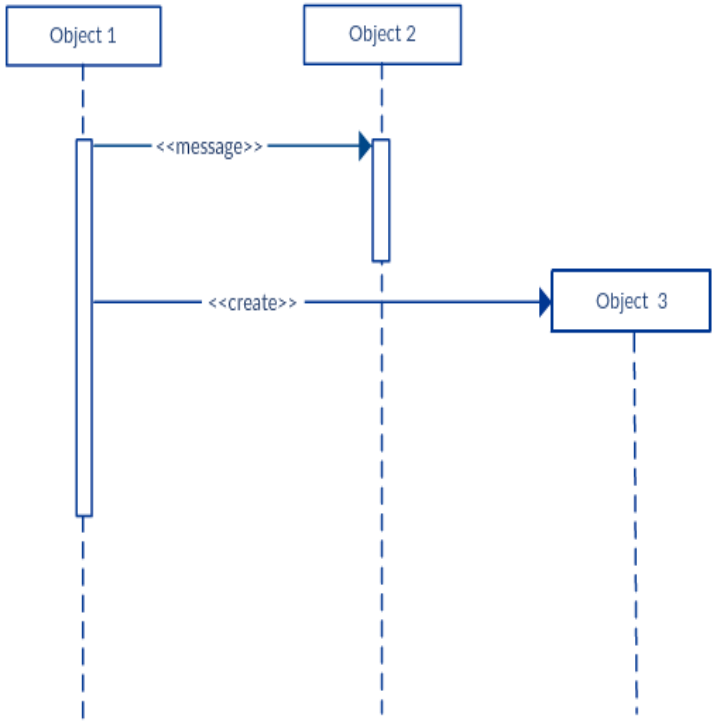
Message Arrows

- ***Return message*** is used to indicate that the message receiver is done processing the message and is returning control over to the message caller.
- Return messages are optional



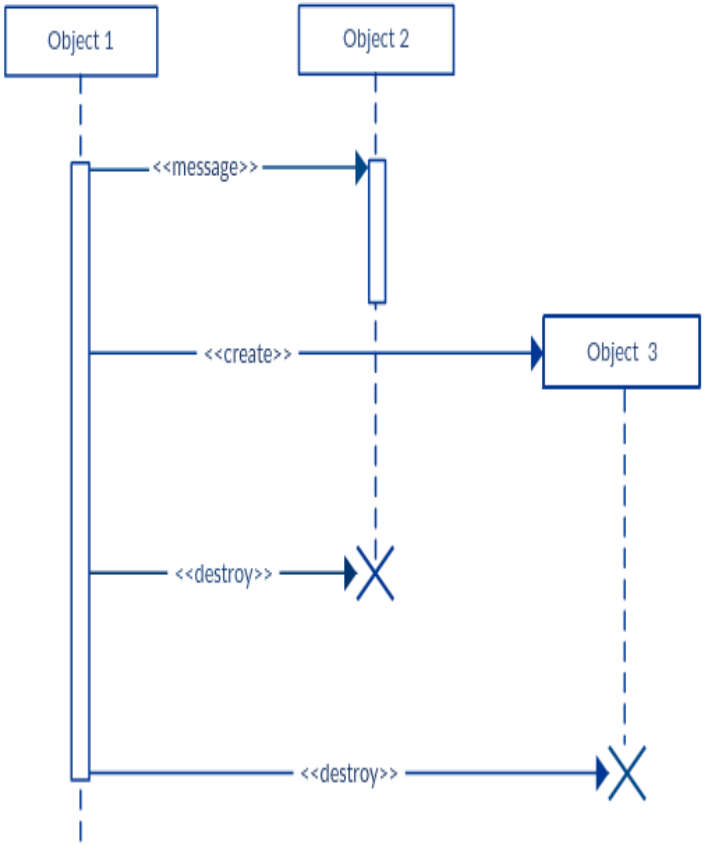
Participant creation message

- Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.
- The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent



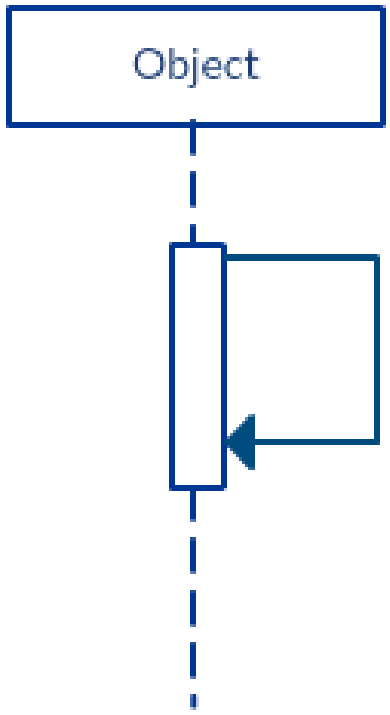
Participant destruction message

- participants when no longer needed can also be deleted from a sequence diagram.
- This is done by adding an ‘X’ at the end of the lifeline of the said participant.



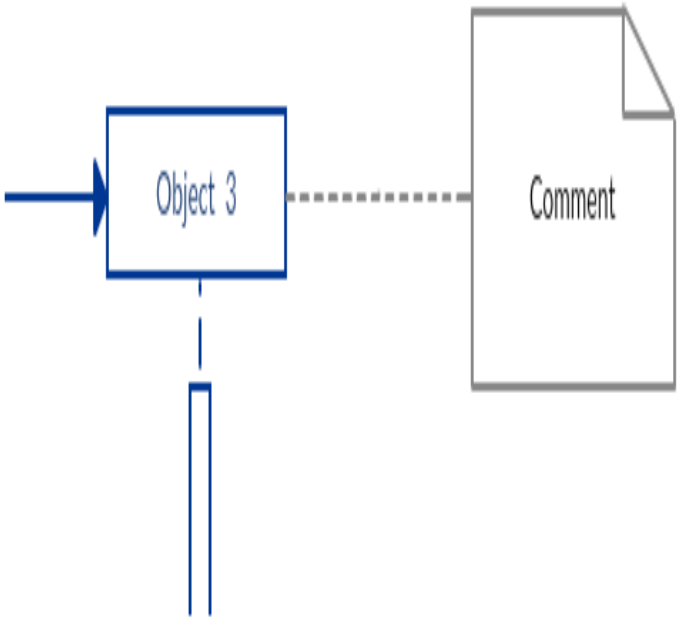
Reflexive message

- When an object sends a message to itself, it is called a reflexive message.
- It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example below.

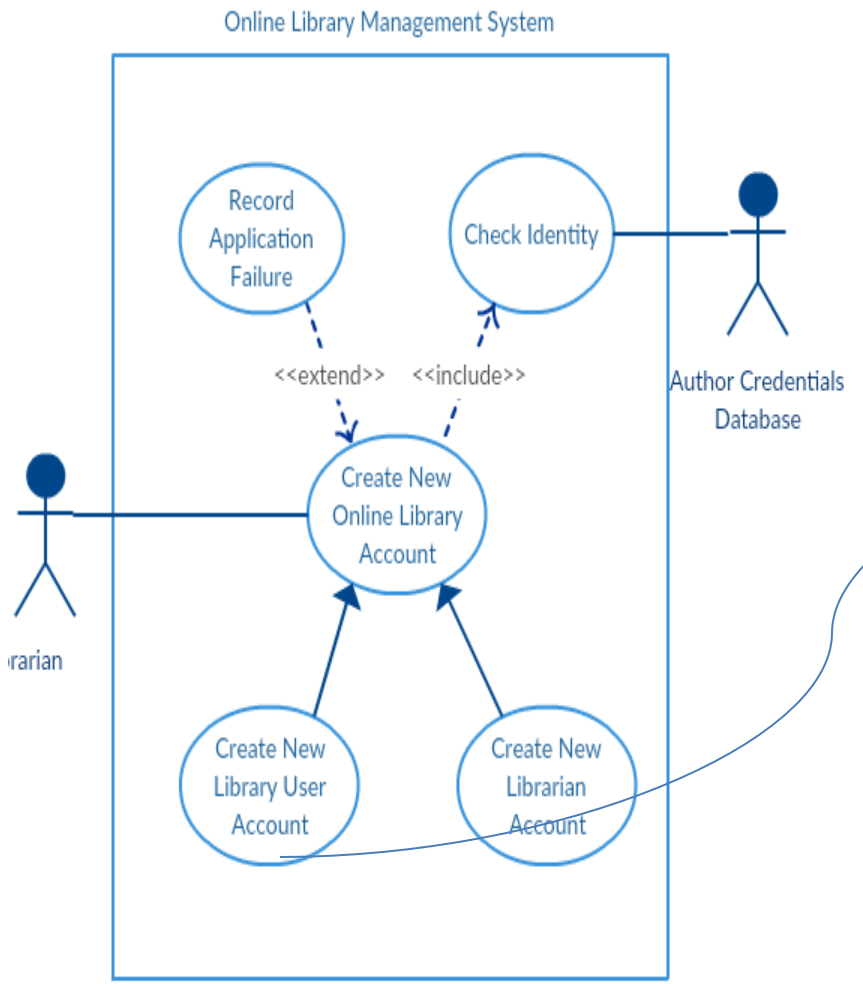


Comment

- UML diagrams generally permit the annotation of comments in all UML diagram types.
- The comment object is a rectangle with a folded-over corner as shown below.
- The comment can be linked to the related object with a dashed line.



How to Draw a Sequence Diagram



- use case diagram example of ‘Create New Online Library Account’,
- we will focus on the use case named ‘Create New Library User Account’ to draw our sequence diagram

Sequence Diagram – New Library user account

- Identify the objects or actors
  - Librarian
  - Online Library Management system
  - User credentials database
  - Email system

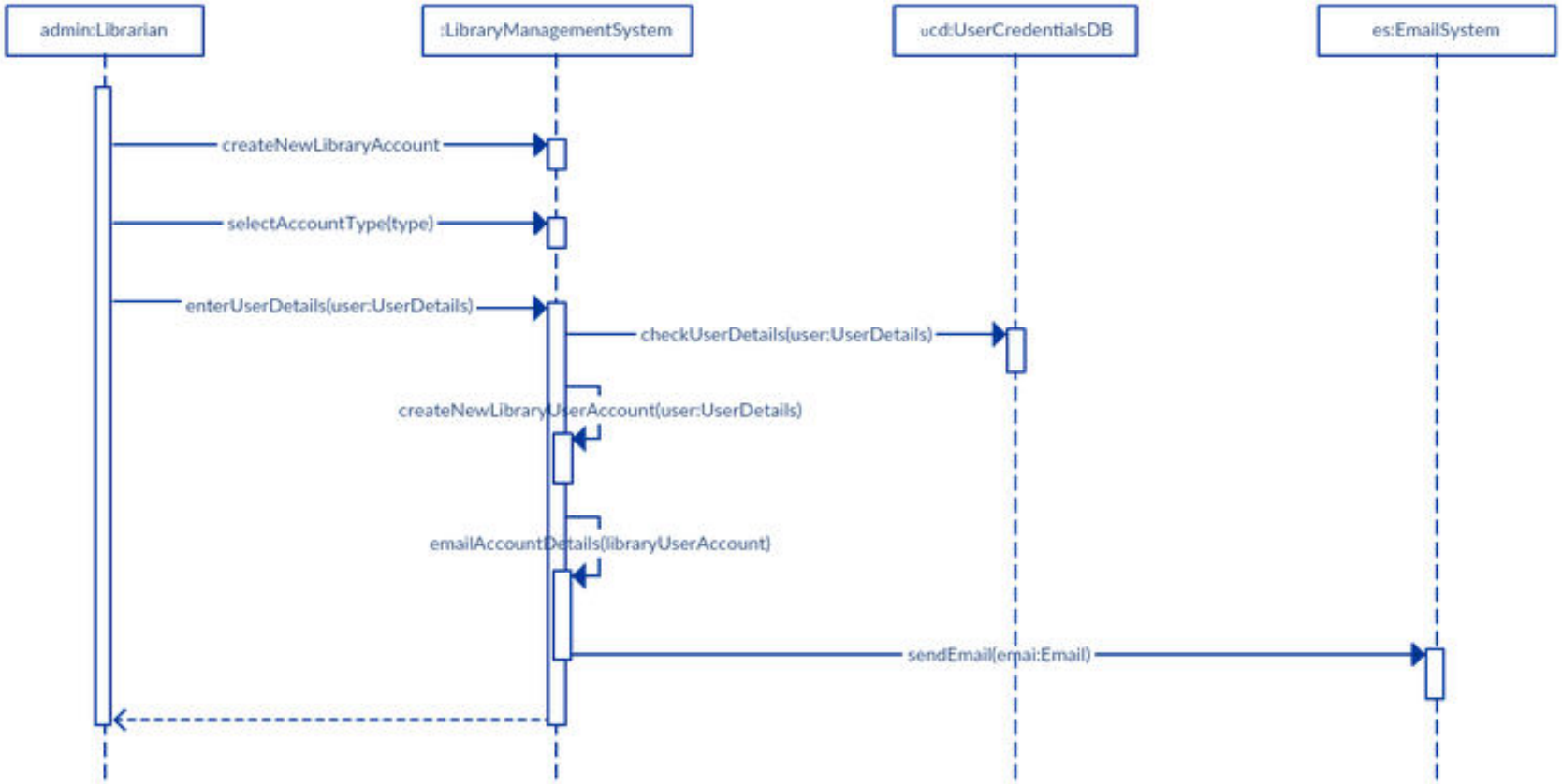
- Write a detailed description on what the use case does.

Sequence Diagram – New Library user account

- Here are the steps that occur in the use case named ‘Create New Library User Account’.
  - The librarian request the system to create a new online library account
  - The librarian then selects the library user account type
  - The librarian enters the user’s details
  - The user’s details are checked using the user Credentials Database
  - The new library user account is created
  - A summary of the of the new account’s details are then emailed to the user

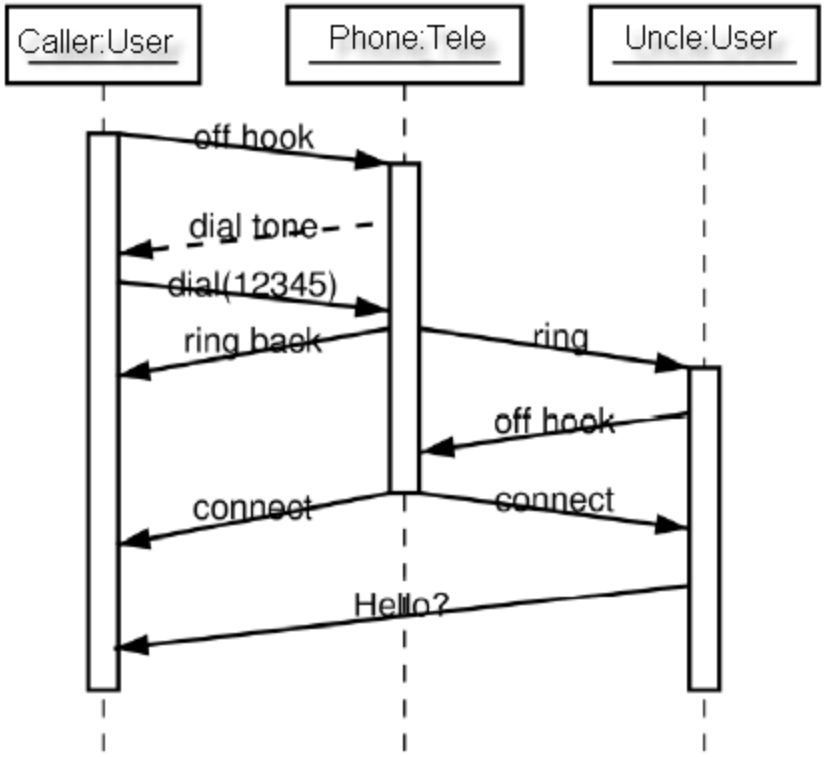
Sequence Diagram – New Library user account

The sequence diagram below shows how the objects in the online library management system interact with each other to perform the function 'Create New Library User Account'.



Sequence Diagram : Timing

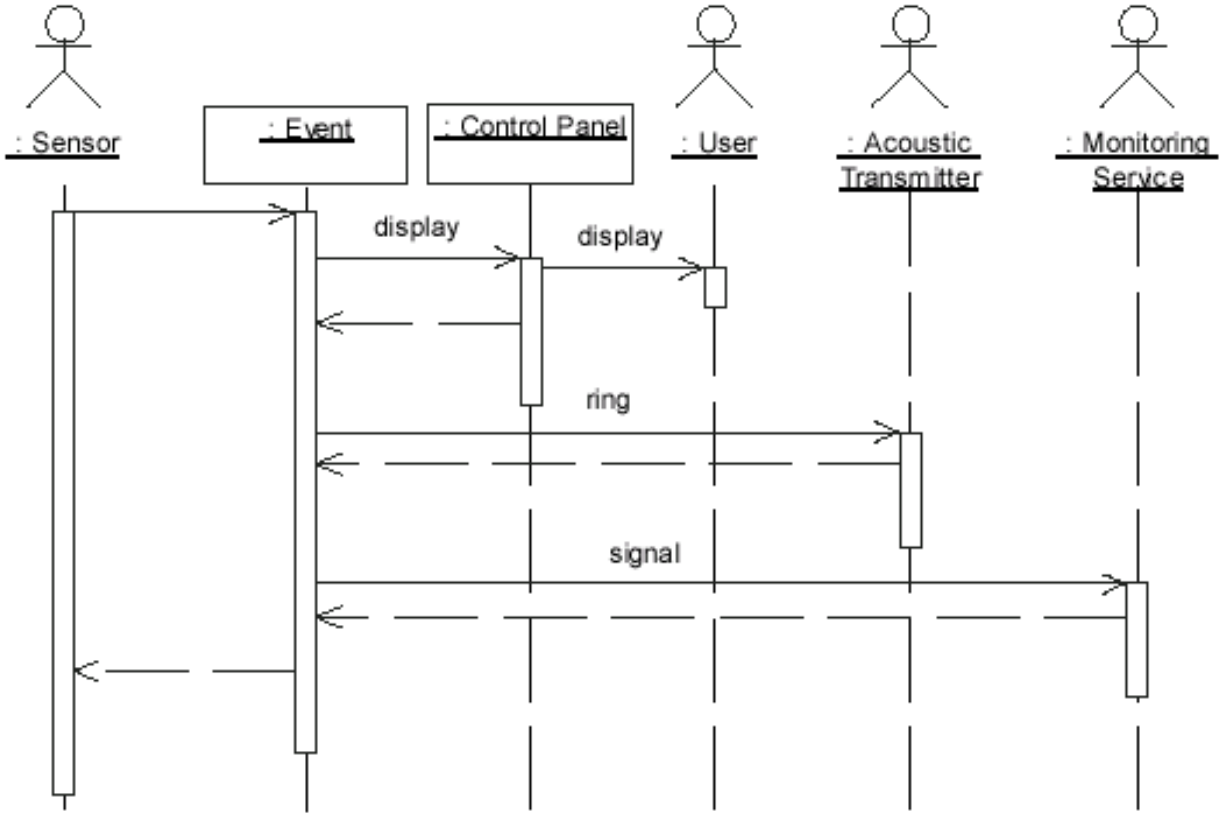
Slanted Lines show propagation delay of messages
Good for modeling real-time systems



If messages cross this is usually problematic – race conditions

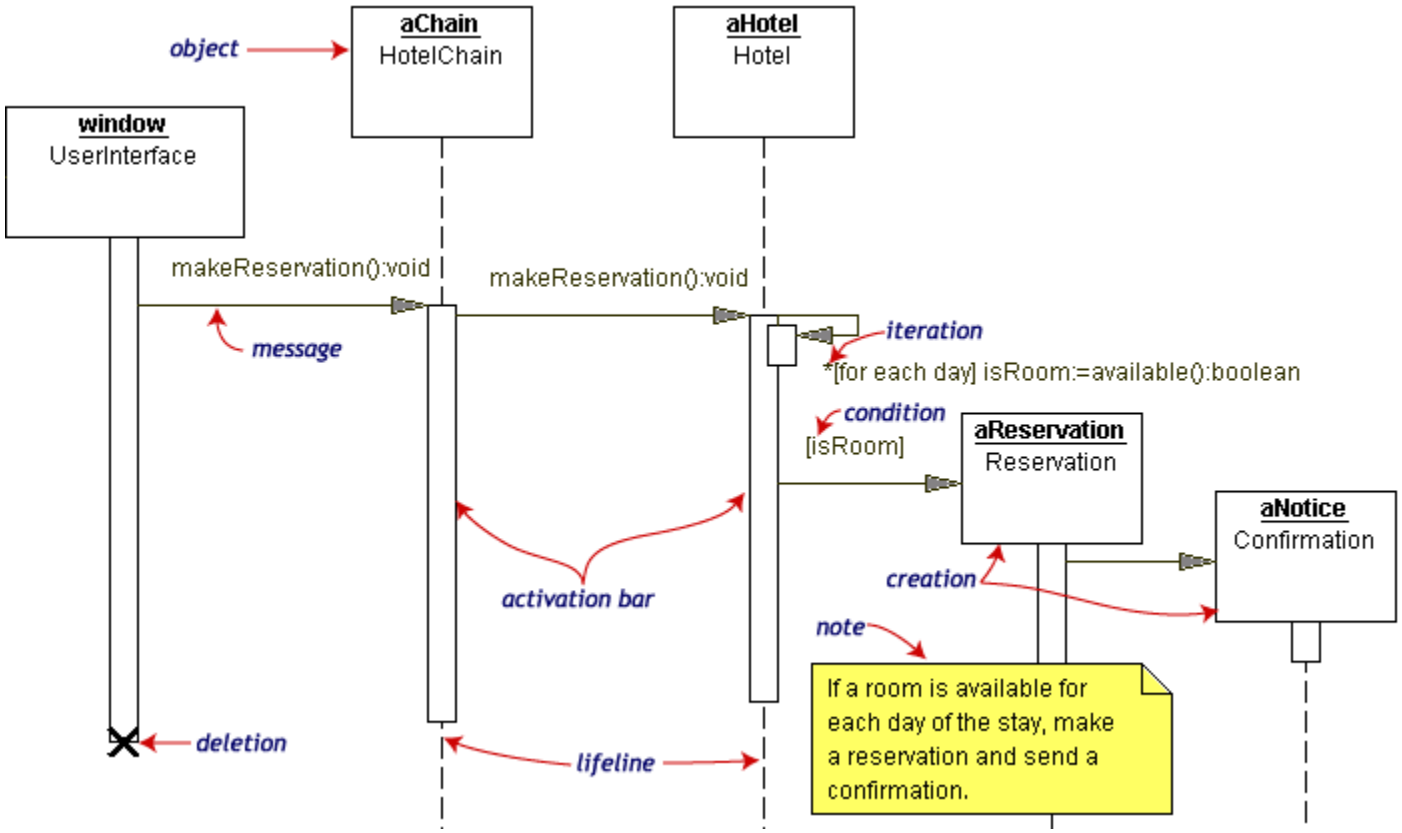
Sequence Example: Alarm System

- When the alarm goes off, it rings the alarm, puts a message on the display, notifies the monitoring service



Sequence Diagram Example

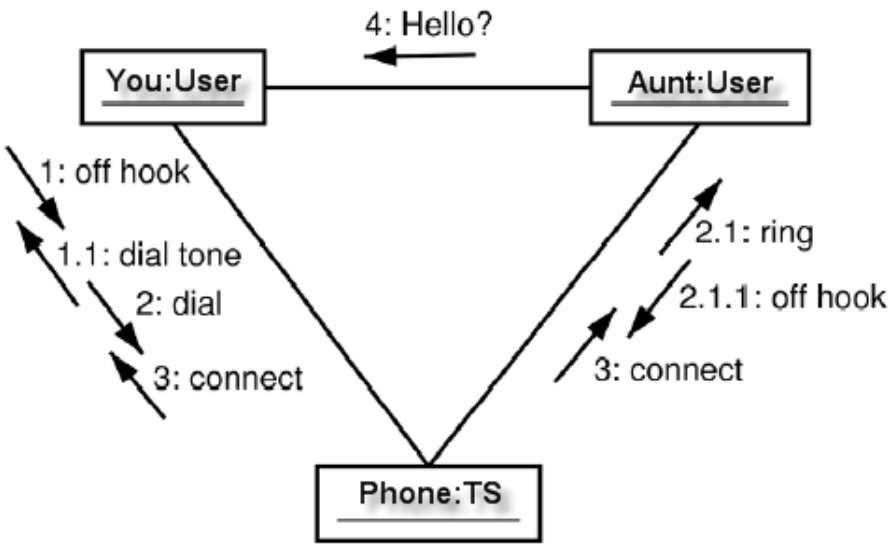
Hotel Reservation

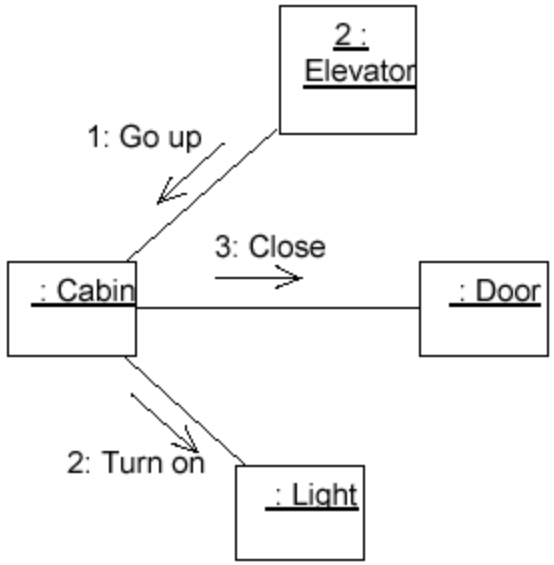


Collaboration Diagram

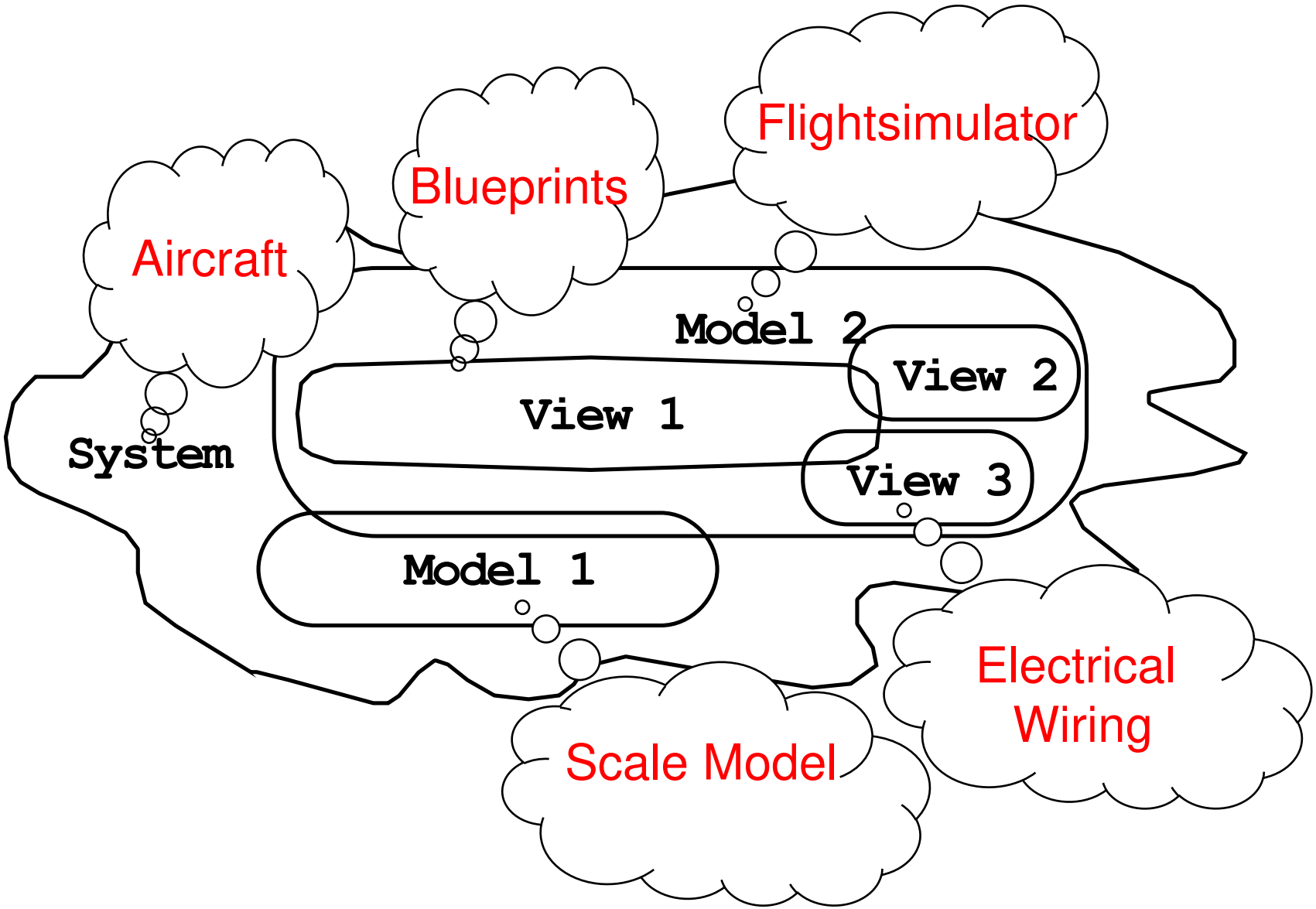
- Collaboration Diagrams show similar information to sequence diagrams, except that the vertical sequence is missing. In its place are:
  - Object Links - solid lines between the objects that interact
  - On the links are Messages - arrows with one or more message name that show the direction and names of the messages sent between objects
- Emphasis on static links as opposed to sequence in the sequence diagram

Collaboration Diagram



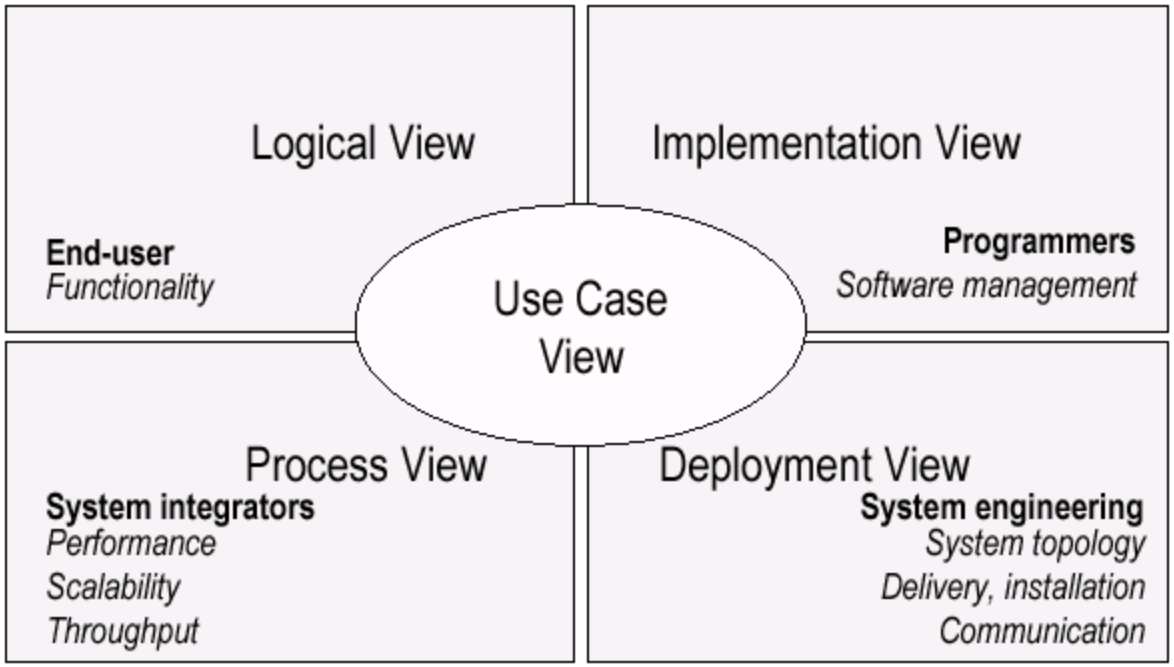


Systems, Models and Views

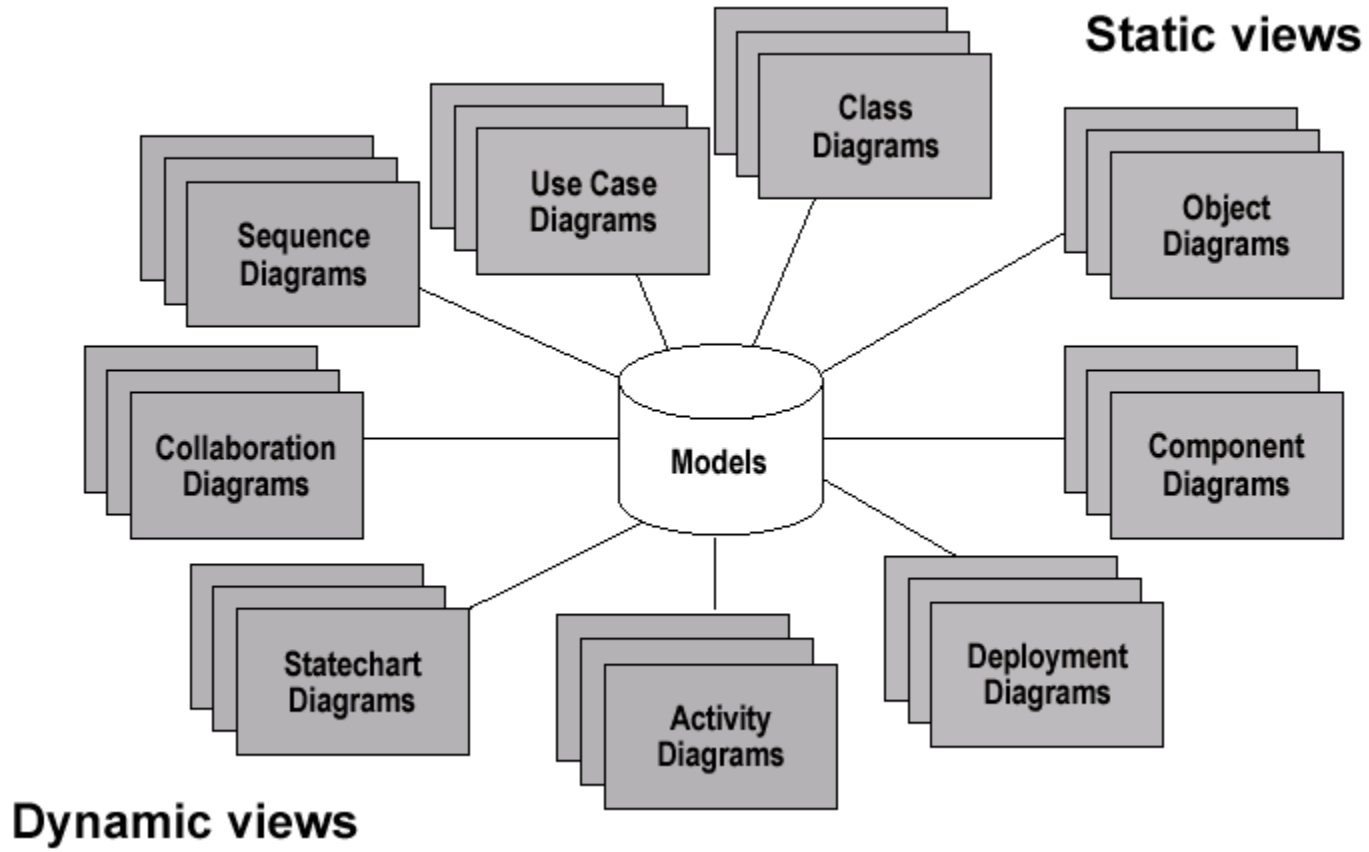


UML Models, Views, Diagrams

- UML is a multi-diagrammatic language
  - Each diagram is a view into a model
    - Diagram presented from the aspect of a particular stakeholder
    - Provides a partial representation of the system
    - Is semantically consistent with other views
  - Example views



Models, Views, Diagrams



How Many Views?

- Views should to fit the context
  - Not all systems require all views
  - Single processor: drop deployment view
  - Single process: drop process view
  - Very small program: drop implementation view
- A system might need additional views
  - Data view, security view, …

UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We only cover the 20% here

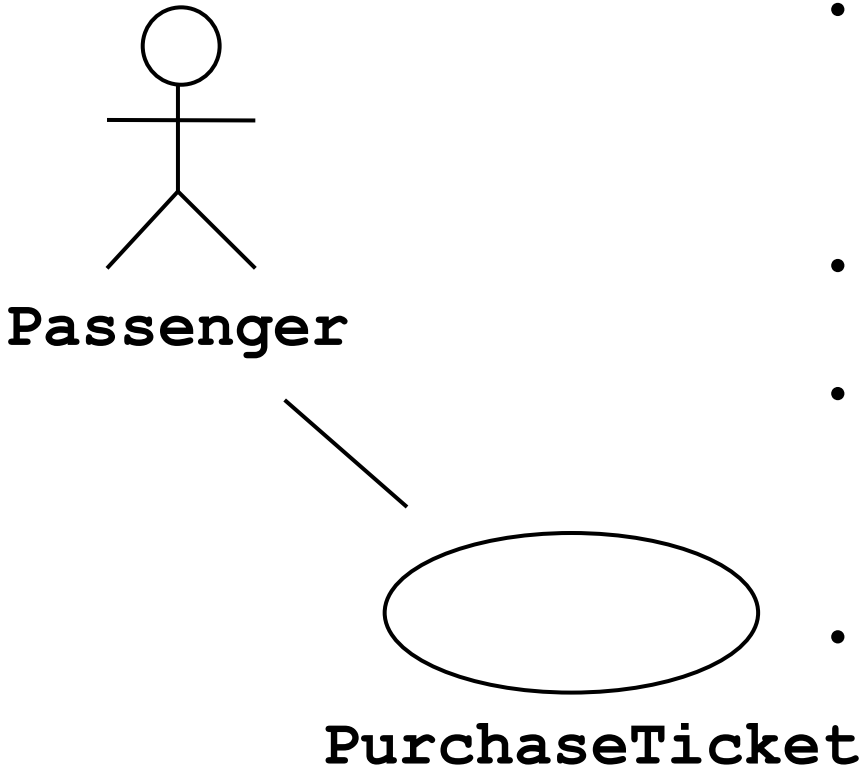
Basic Modeling Steps

- Use Cases
  - Capture requirements
- Domain Model
  - Capture process, key classes
- Design Model
  - Capture details and behaviors of use cases and domain objects
  - Add classes that do the work and define the architecture

UML Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

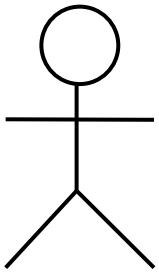
Use Case Diagrams



- Used during requirements elicitation to represent external behavior

- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality; summary of scenarios
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors

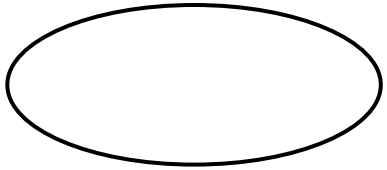


Passenger

- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

Use Case

A use case represents a class of functionality provided by the system as an event flow.



PurchaseTicket

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Diagram: Example

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

Exit condition:

- `Passenger` has ticket.

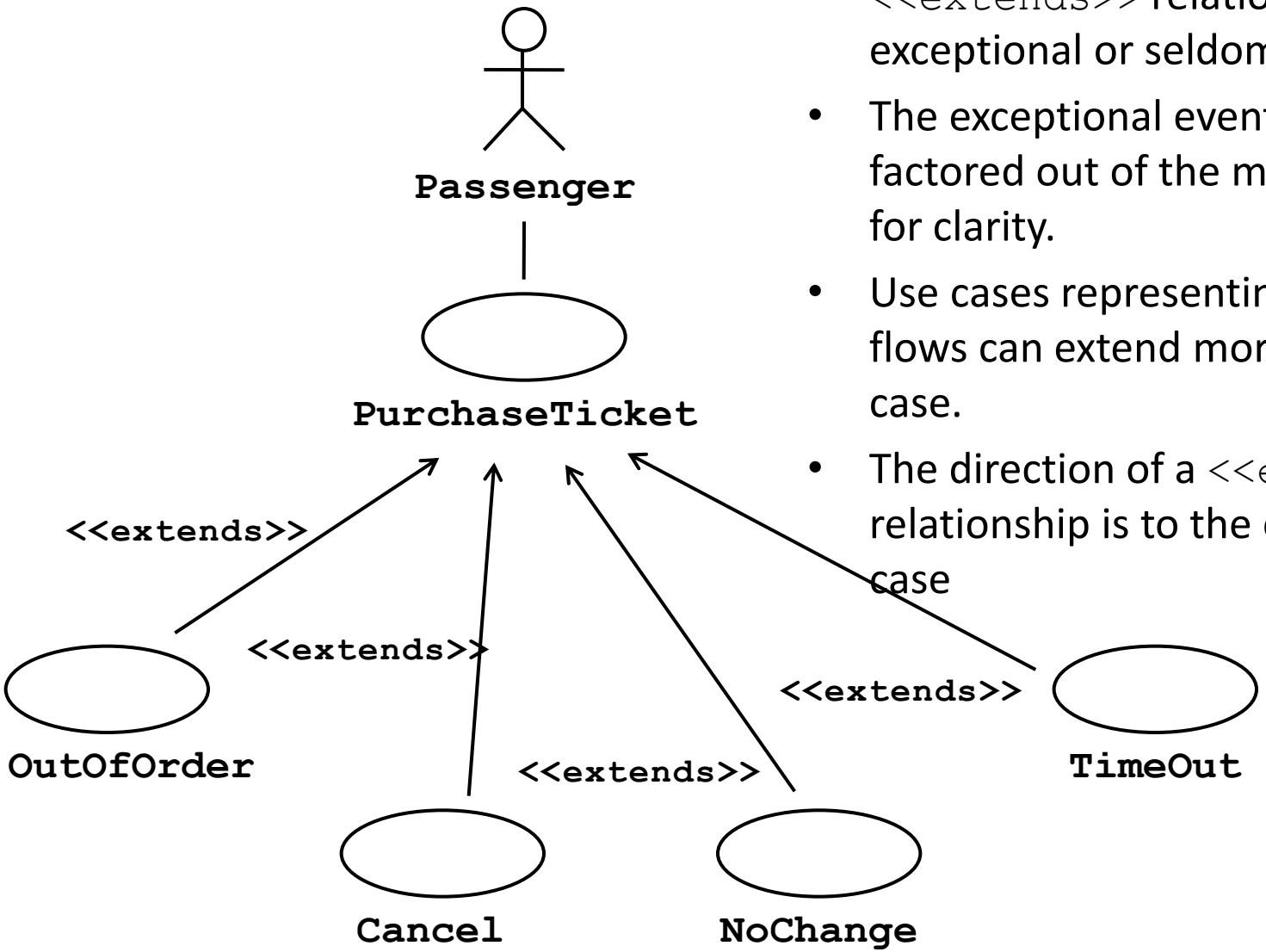
Event flow:

1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

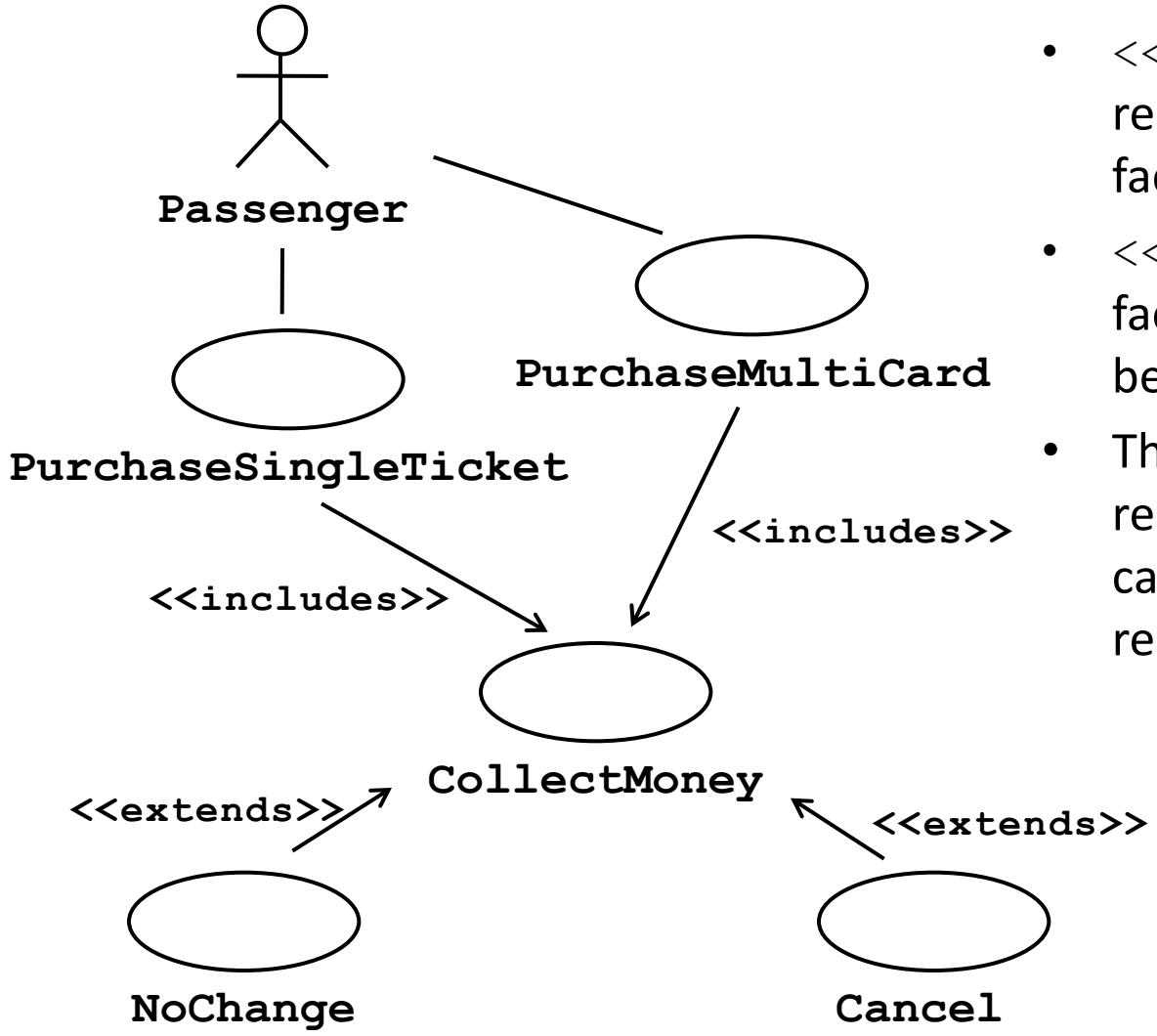
Exceptional cases!

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

The <<includes>> Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Cases are useful to…

- Determining requirements
  - New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients
  - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- Generating test cases
  - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Use Case Diagrams: Summary

- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

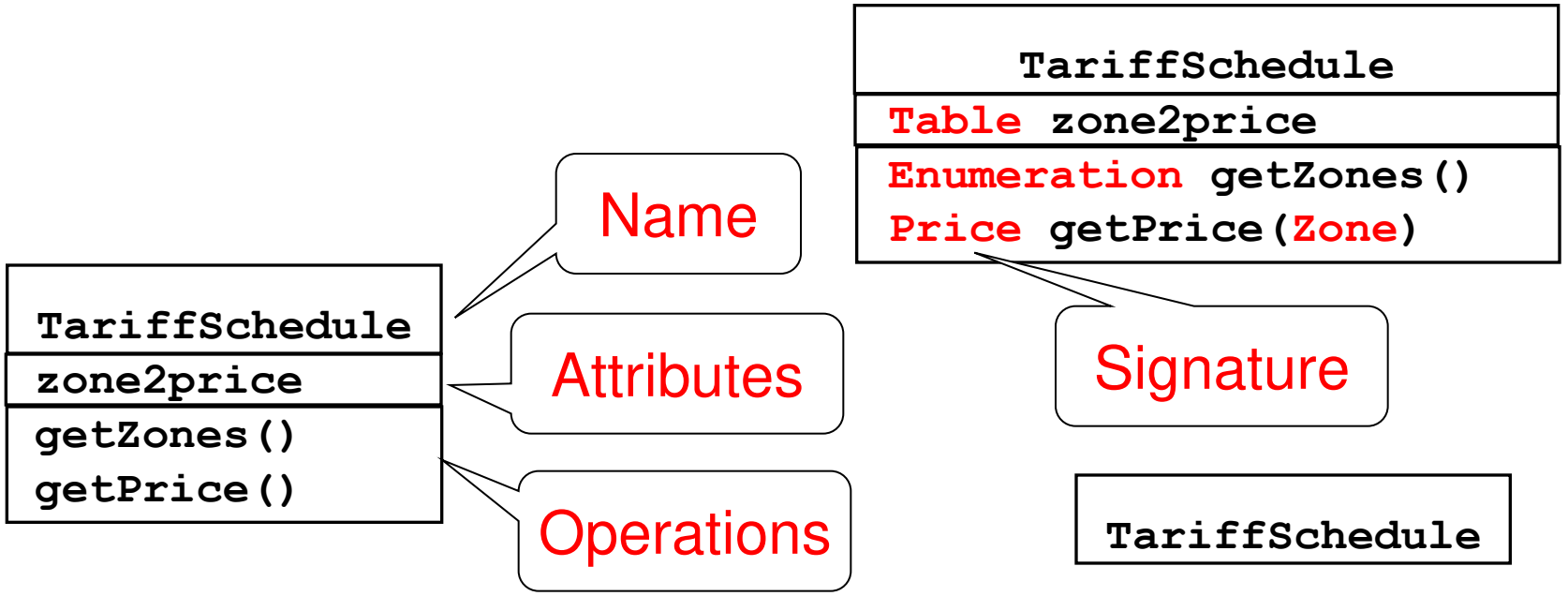
Class Diagrams

- Gives an overview of a system by showing its classes and the relationships among them.
  - Class diagrams are static
  - they display what interacts but not what happens when they do interact
- Also shows attributes and operations of each class
- Good way to describe the overall architecture of system components

Class Diagram Perspectives

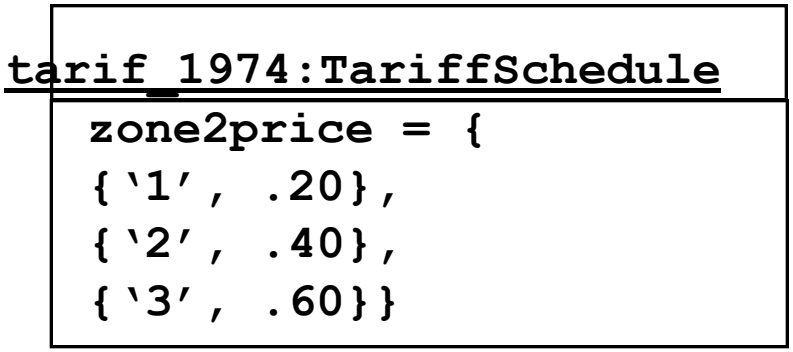
- We draw Class Diagrams under three perspectives
  - Conceptual
    - Software independent
    - Language independent
  - Specification
    - Focus on the interfaces of the software
  - Implementation
    - Focus on the implementation of the software

Classes – Not Just for Code



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

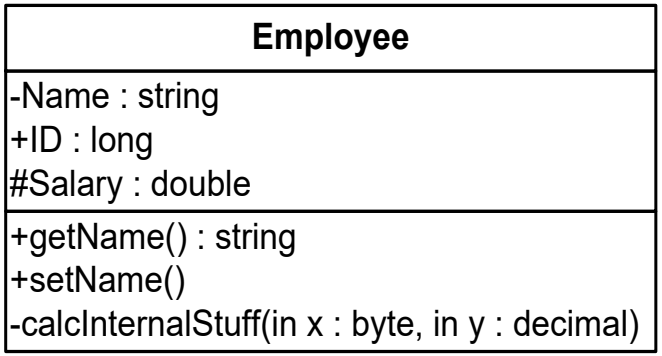
Instances



- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e. data members, variables)
  - Class operations (i.e. methods)
- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined (i.e. shared among all members of the class)
- Abstract class: Name in italics

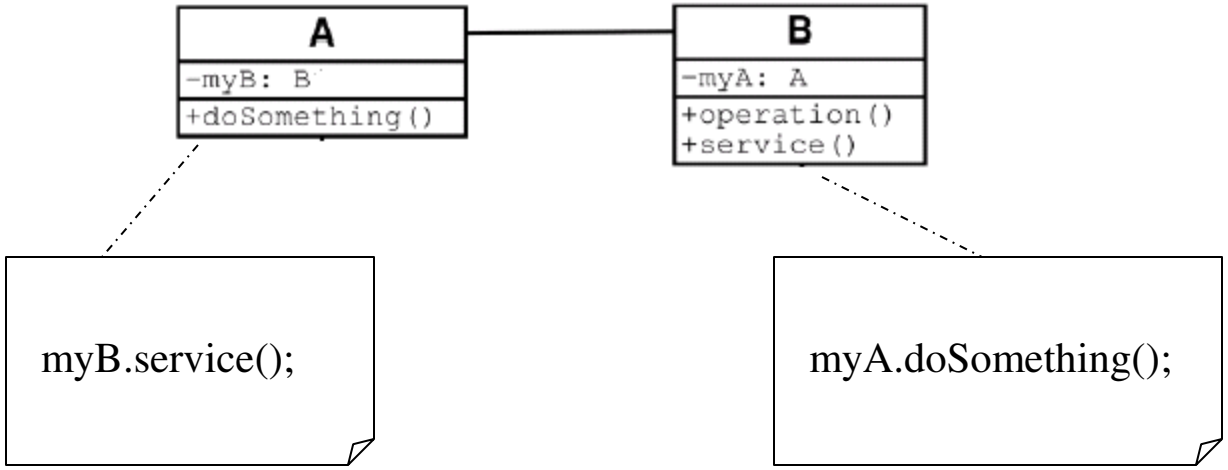


UML Class Notation

- Lines or arrows between classes indicate relationships
  - Association
    - A relationship between instances of two classes, where one class must know about the other to do its work, e.g. client communicates to server
    - indicated by a straight line or arrow
  - Aggregation
    - An association where one class belongs to a collection, e.g. instructor part of Faculty
    - Indicated by an empty diamond on the side of the collection
  - Composition
    - Strong form of Aggregation
    - Lifetime control; components cannot exist without the aggregate
    - Indicated by a solid diamond on the side of the collection
  - Inheritance
    - An inheritance link indicating one class a superclass relationship, e.g. bird is part of mammal
    - Indicated by triangle pointing to superclass

Binary Association

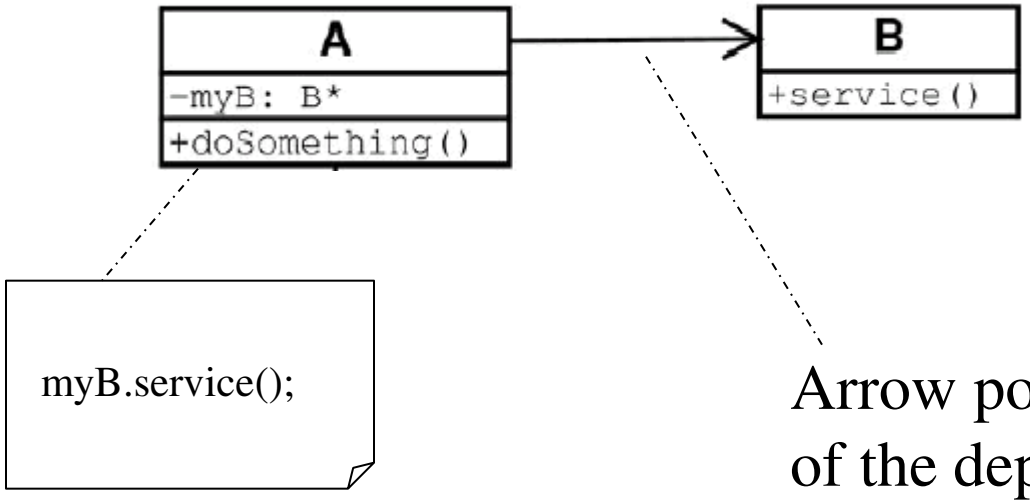
Binary Association: Both entities “Know About” each other



Optionally, may create an Associate Class

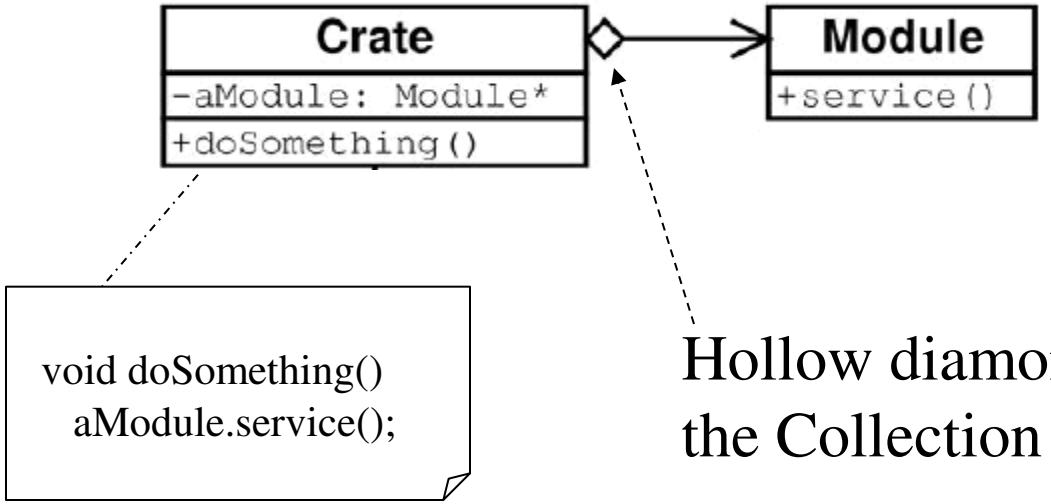
Unary Association

A knows about B, but B knows nothing about A



Aggregation

Aggregation is an association with a “collection-member” relationship



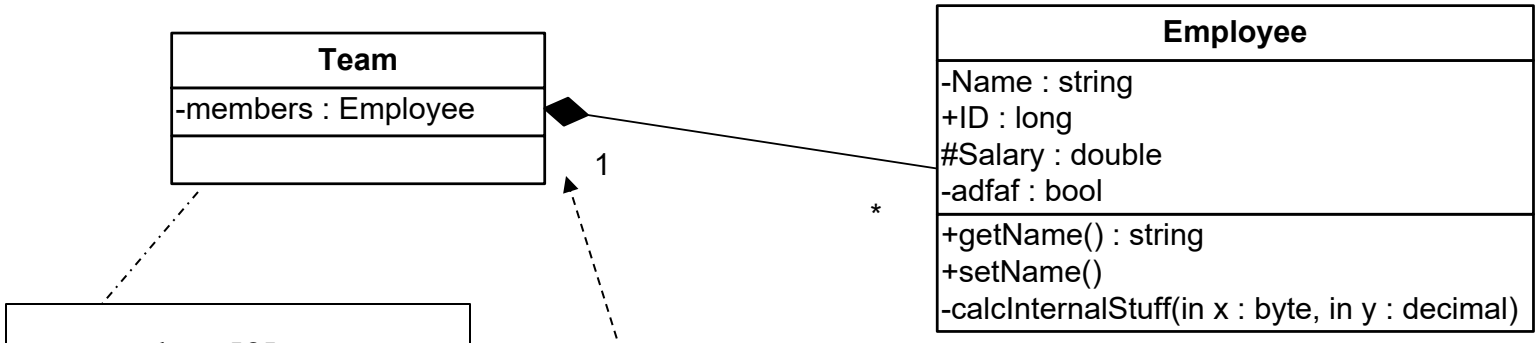
Hollow diamond on the Collection side

No sole ownership implied

Composition

Composition is Aggregation with:

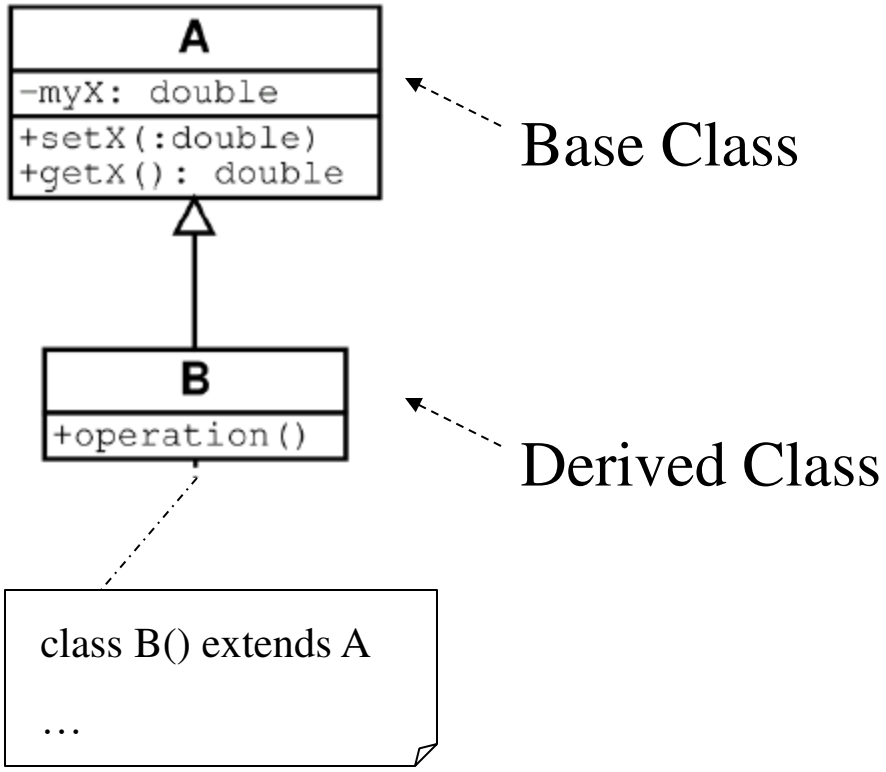
- Lifetime Control (owner controls construction, destruction)
- Part object may belong to only one whole object



Filled diamond on side of the Collection

Inheritance

Standard concept of inheritance

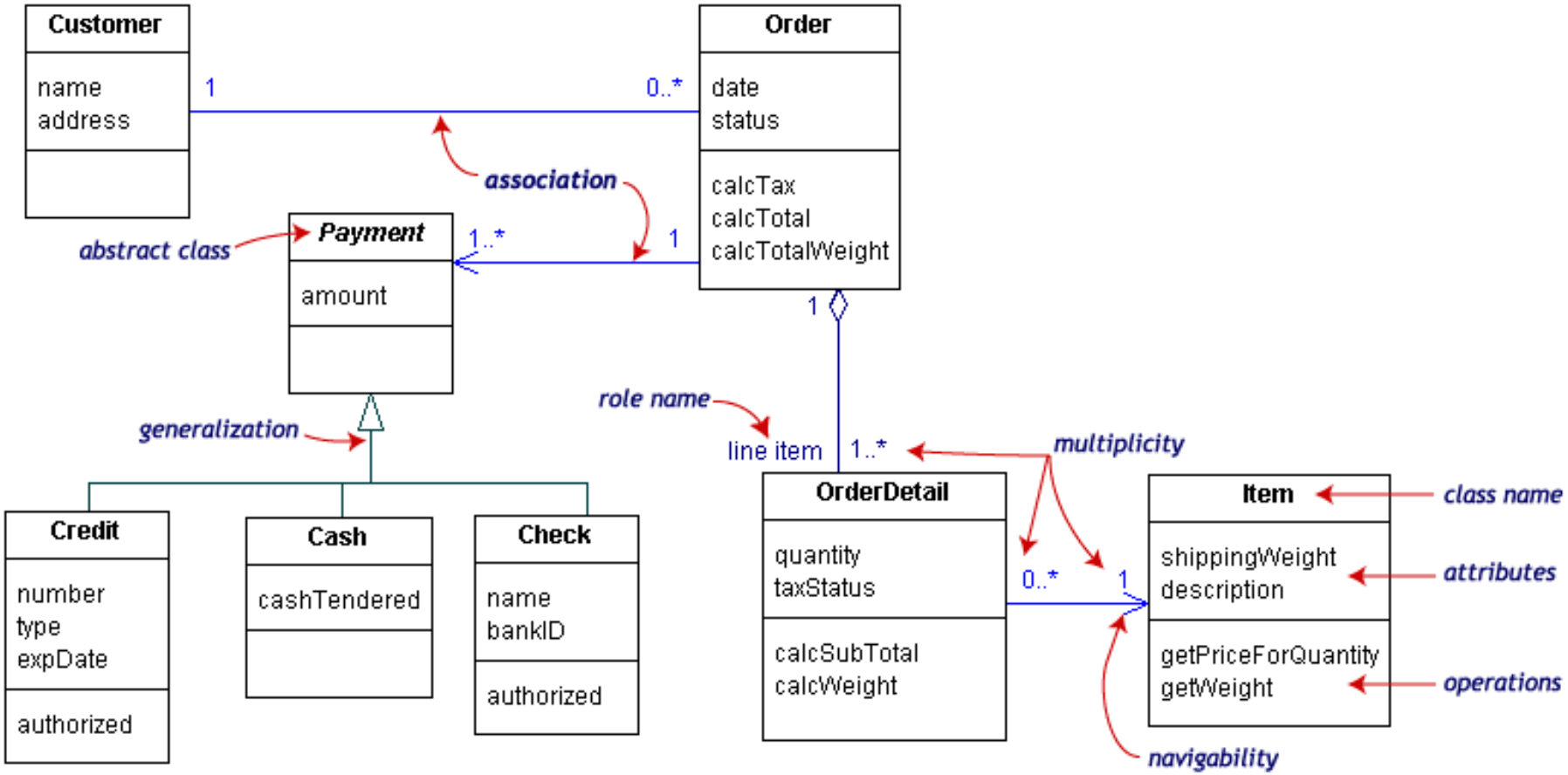


UML Multiplicities

Links on associations to specify more details about the relationship

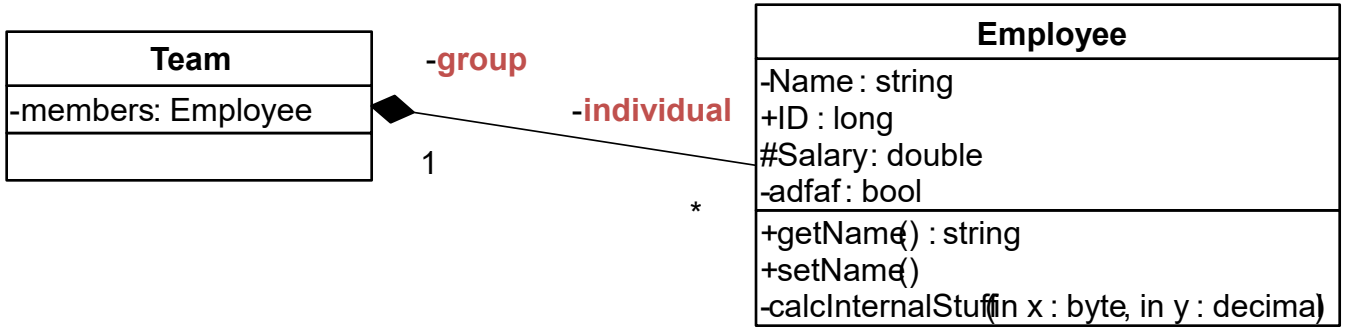
Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n*** **. .** ***m*** indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

UML Class Example



Association Details

- Can assign names to the ends of the association to give further information



Static vs. Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at design time
  - Doesn’t change
- Dynamic design shows communication between objects
  - Similarity to class relations
  - Can follow sequences of events
  - May change depending upon execution scenario
  - Called Object Diagrams

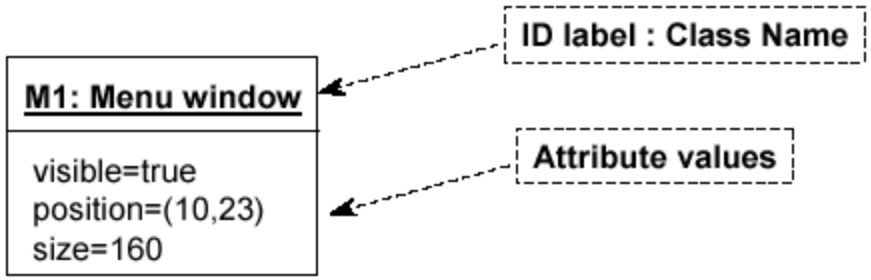
Object Diagrams

- Shows instances of Class Diagrams and links among them
  - An object diagram is a snapshot of the objects in a system
    - At a point in time
    - With a selected focus
      - Interactions – Sequence diagram
      - Message passing – Collaboration diagram
      - Operation – Deployment diagram

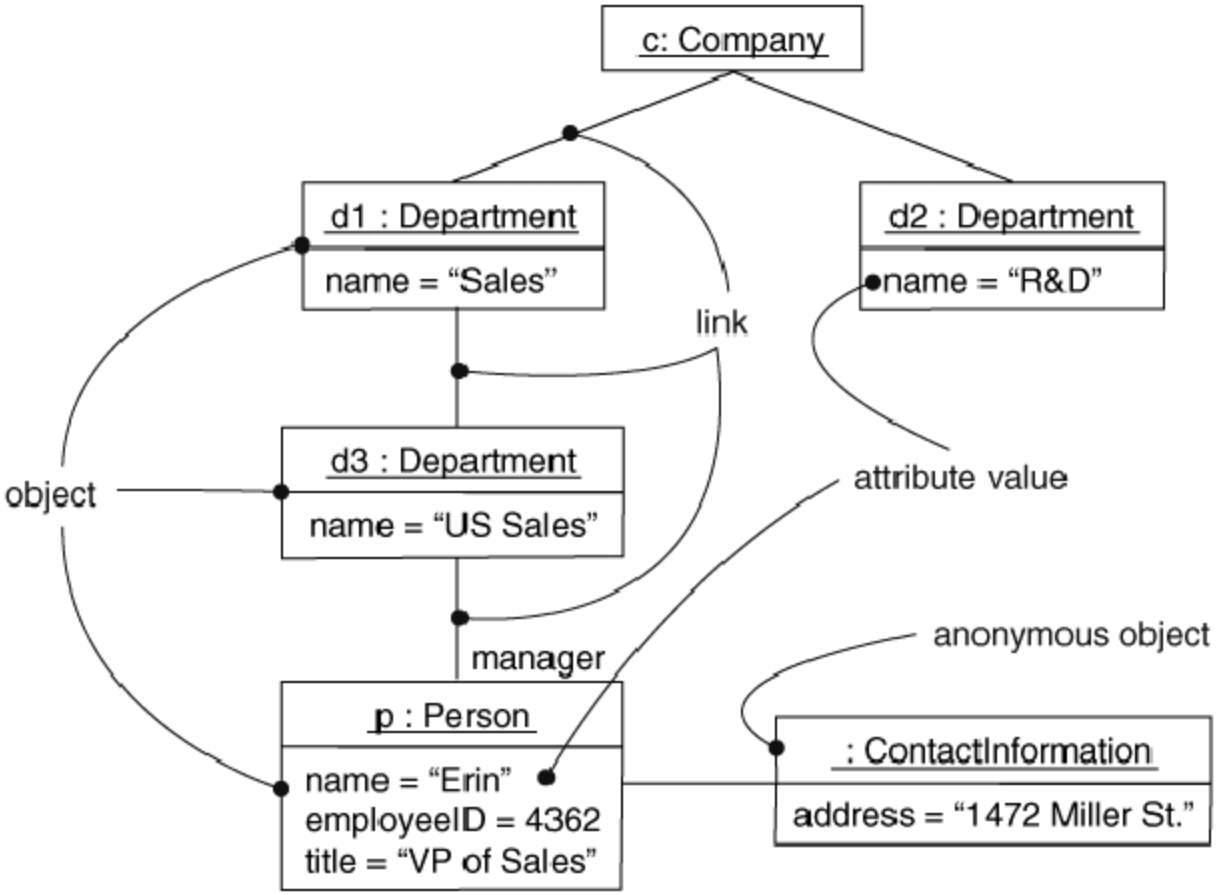
Object Diagrams

- Format is
  - Instance name : Class name
  - Attributes and Values

  - Example:



Objects and Links

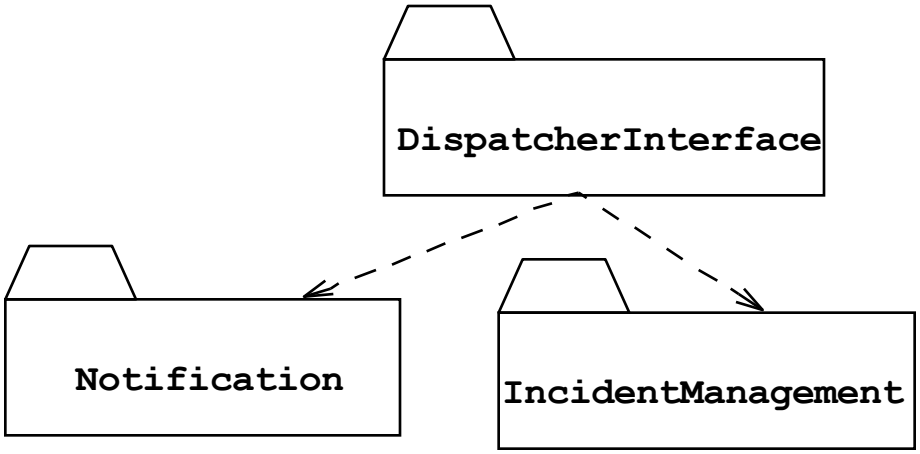


Can add association type and also message type

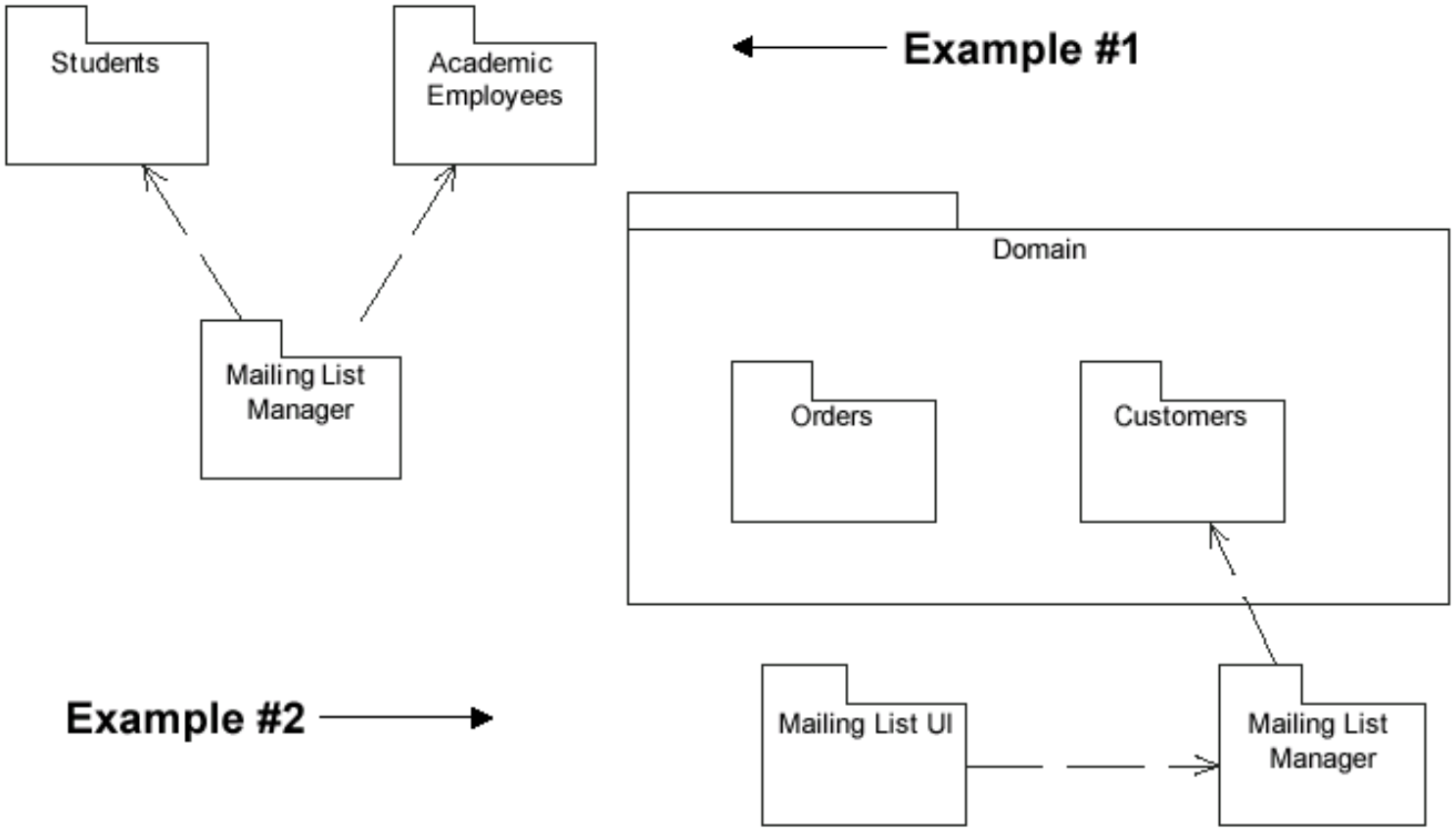
Package Diagrams

- To organize complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements
- Notation
  - 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.
  - Packages are the basic grouping construct with which you may organize UML models to increase their readability

Package Example



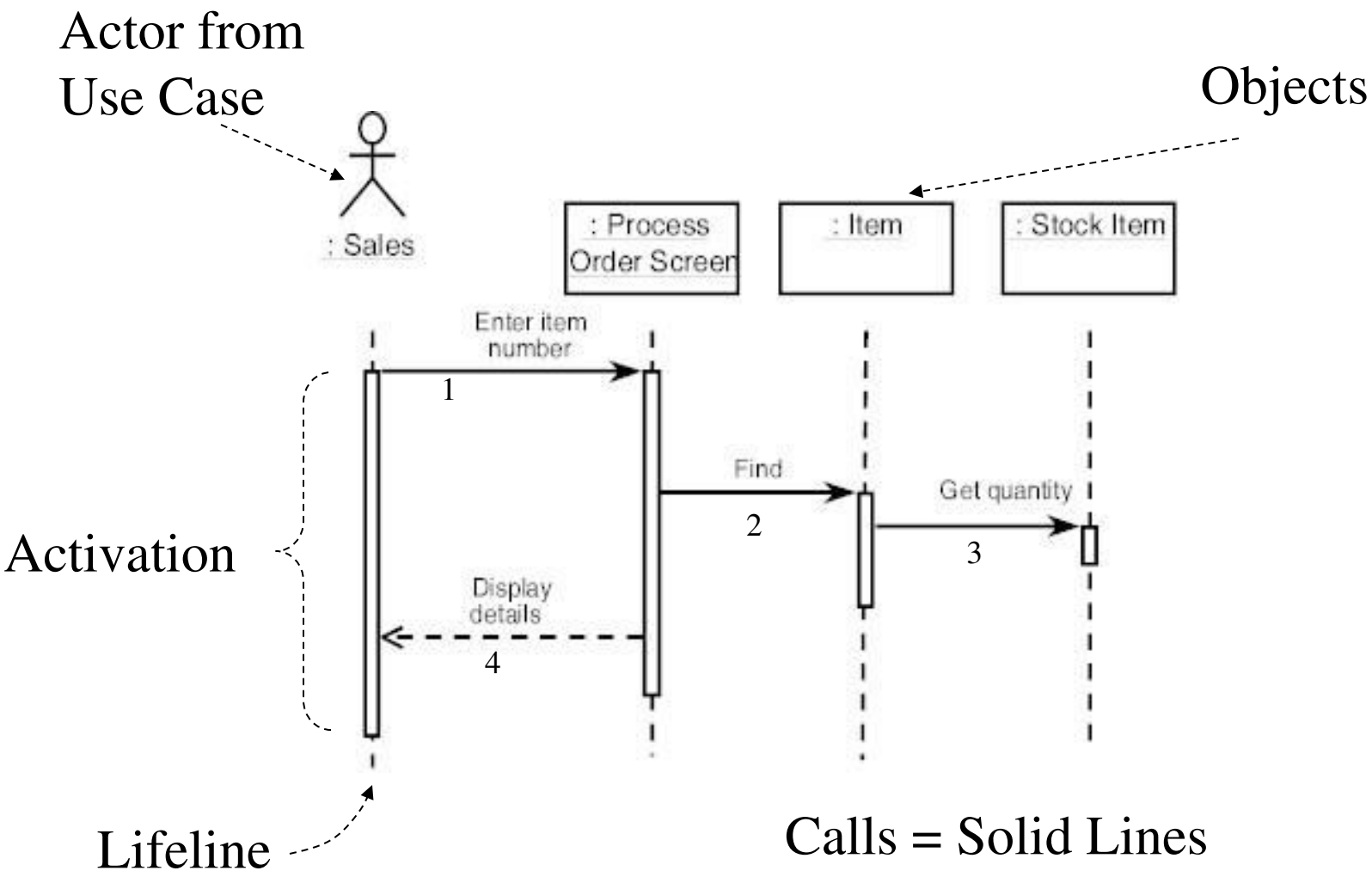
More Package Examples



Interaction Diagrams

- Interaction diagrams are dynamic -- they describe how objects collaborate.
- A Sequence Diagram:
  - Indicates what messages are sent and when
  - Time progresses from top to bottom
  - Objects involved are listed left to right
  - Messages are sent left to right between objects in sequence

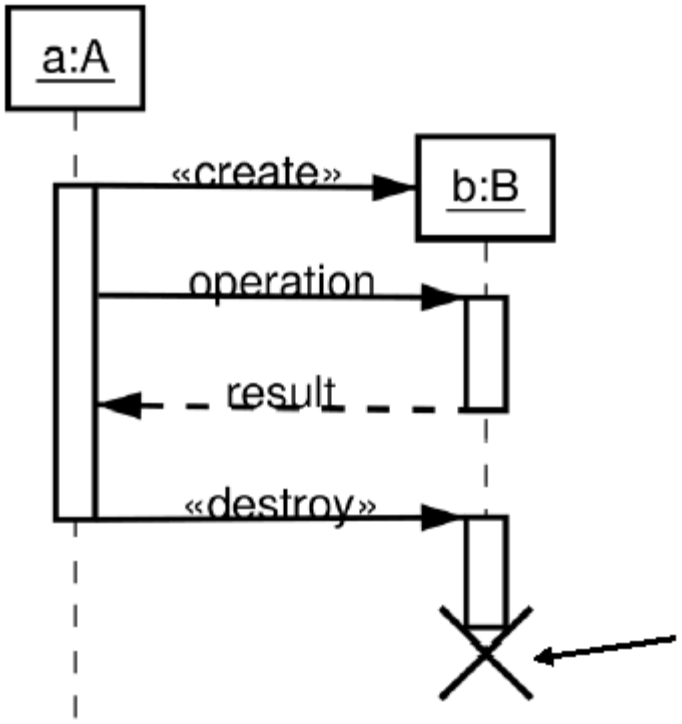
Sequence Diagram Format



Calls = Solid Lines

Returns = Dashed Lines

Sequence Diagram : Destruction



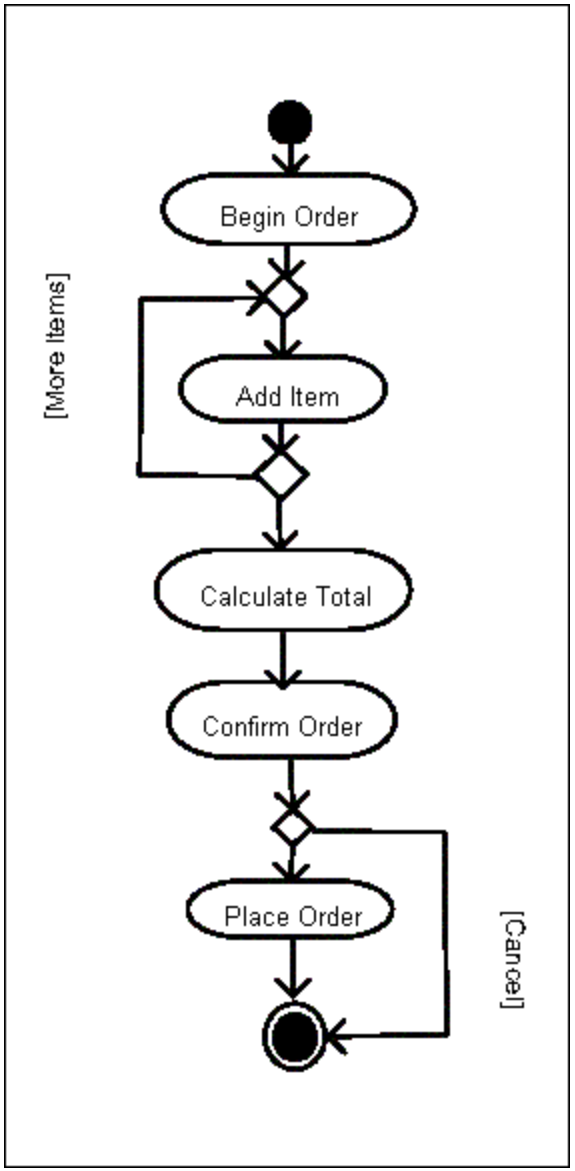
Shows Destruction of b
(and Construction)

Activity Diagrams

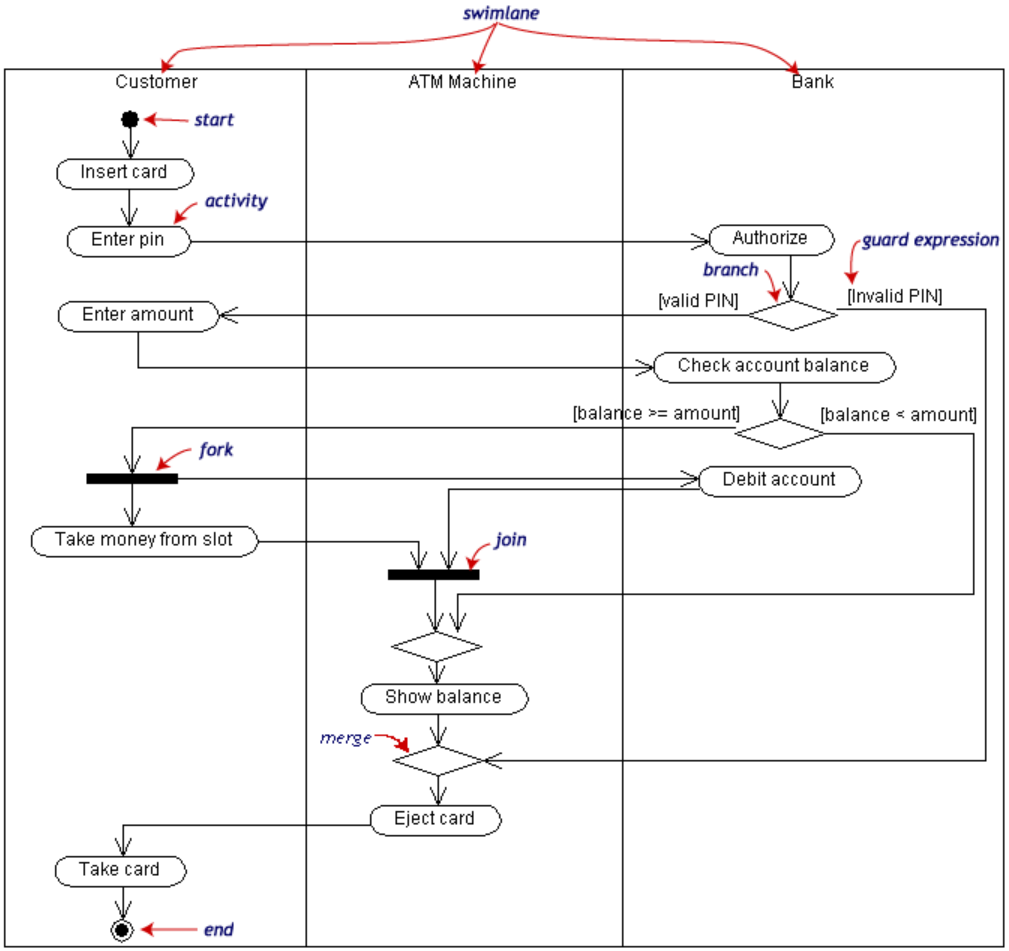
- Fancy flowchart
  - Displays the flow of activities involved in a single process
  - States
    - Describe what is being processed
    - Indicated by boxes with rounded corners
  - Swim lanes
    - Indicates which object is responsible for what activity
  - Branch
    - Transition that branch
    - Indicated by a diamond
  - Fork
    - Transition forking into parallel activities
    - Indicated by solid bars
  - Start and End

Sample Activity Diagram

- Ordering System
- May need multiple diagrams from other points of view



Activity Diagram Example



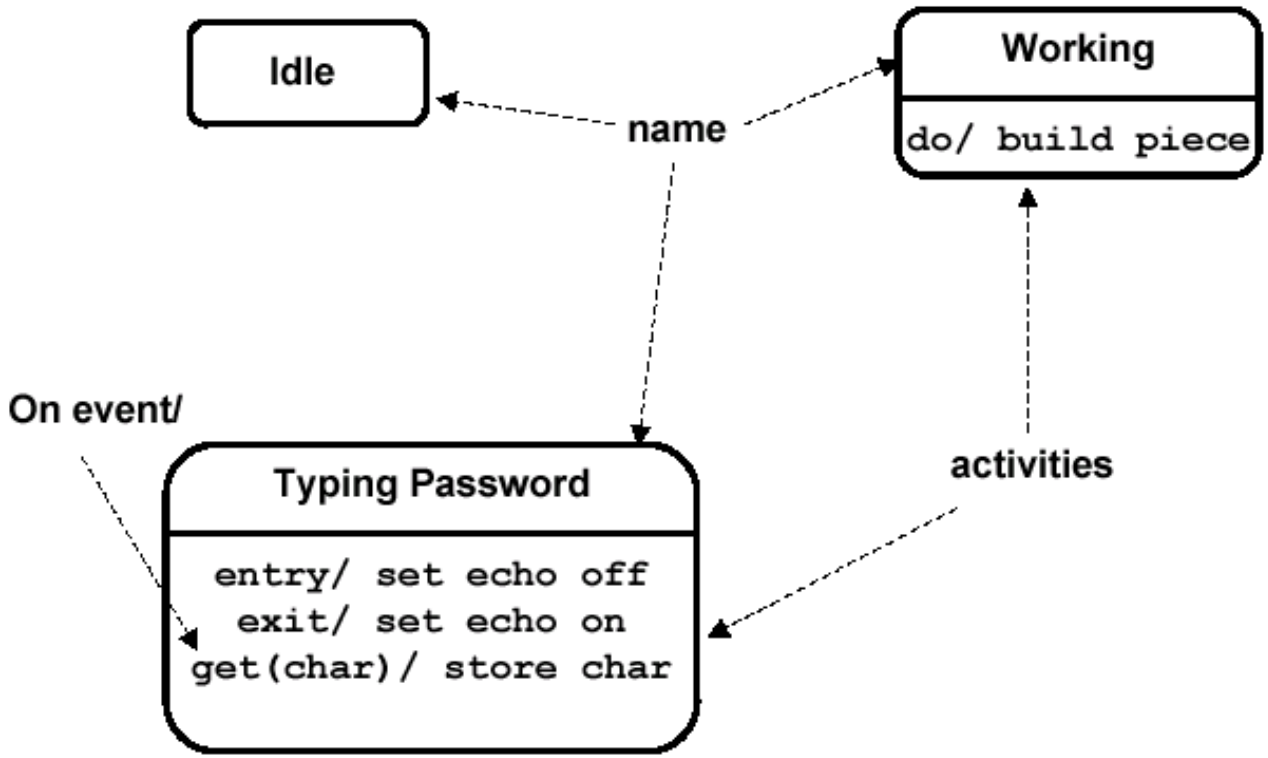
State Transition Diagrams

- Fancy version of a DFA
- Shows the possible states of the object and the transitions that cause a change in state
  - i.e. how incoming calls change the state
- Notation
  - States are rounded rectangles
  - Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows.
  - Initial and Final States indicated by circles as in the Activity Diagram
    - Final state terminates the action; may have multiple final states

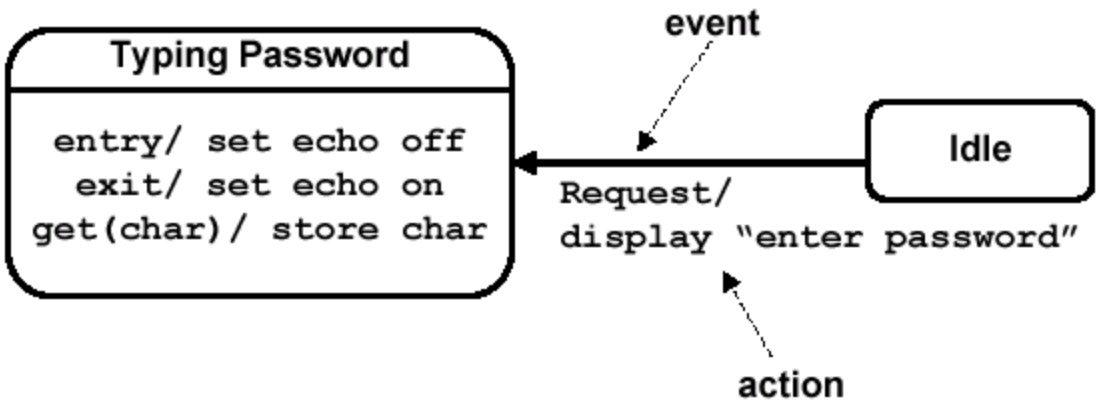
State Representation

- The set of properties and values describing the object in a well defined instant are characterized by
  - Name
  - Activities (executed inside the state)
    - Do/ activity
  - Actions (executed at state entry or exit)
    - Entry/ action
    - Exit/ action
  - Actions executed due to an event
    - Event [Condition] / Action ^Send Event

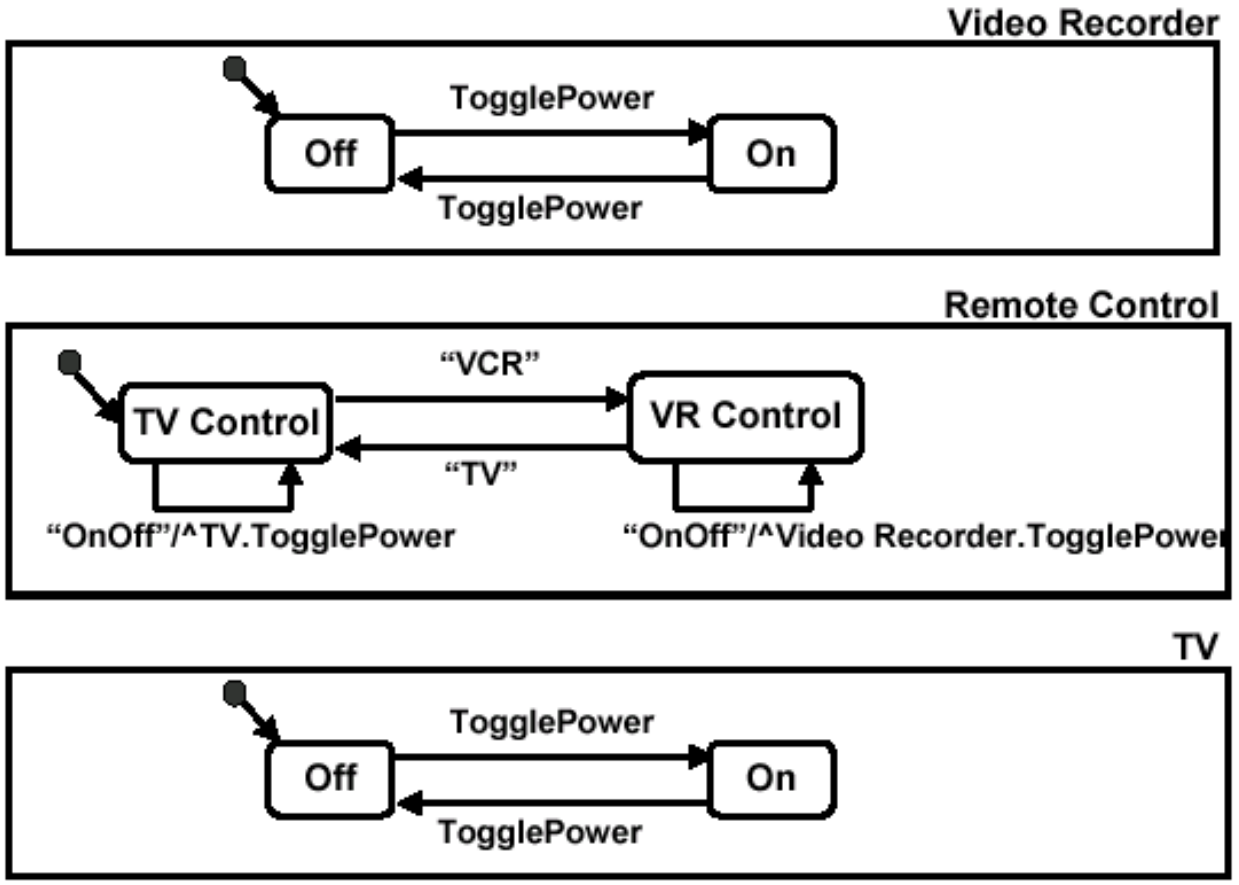
Notation for States



Simple Transition Example

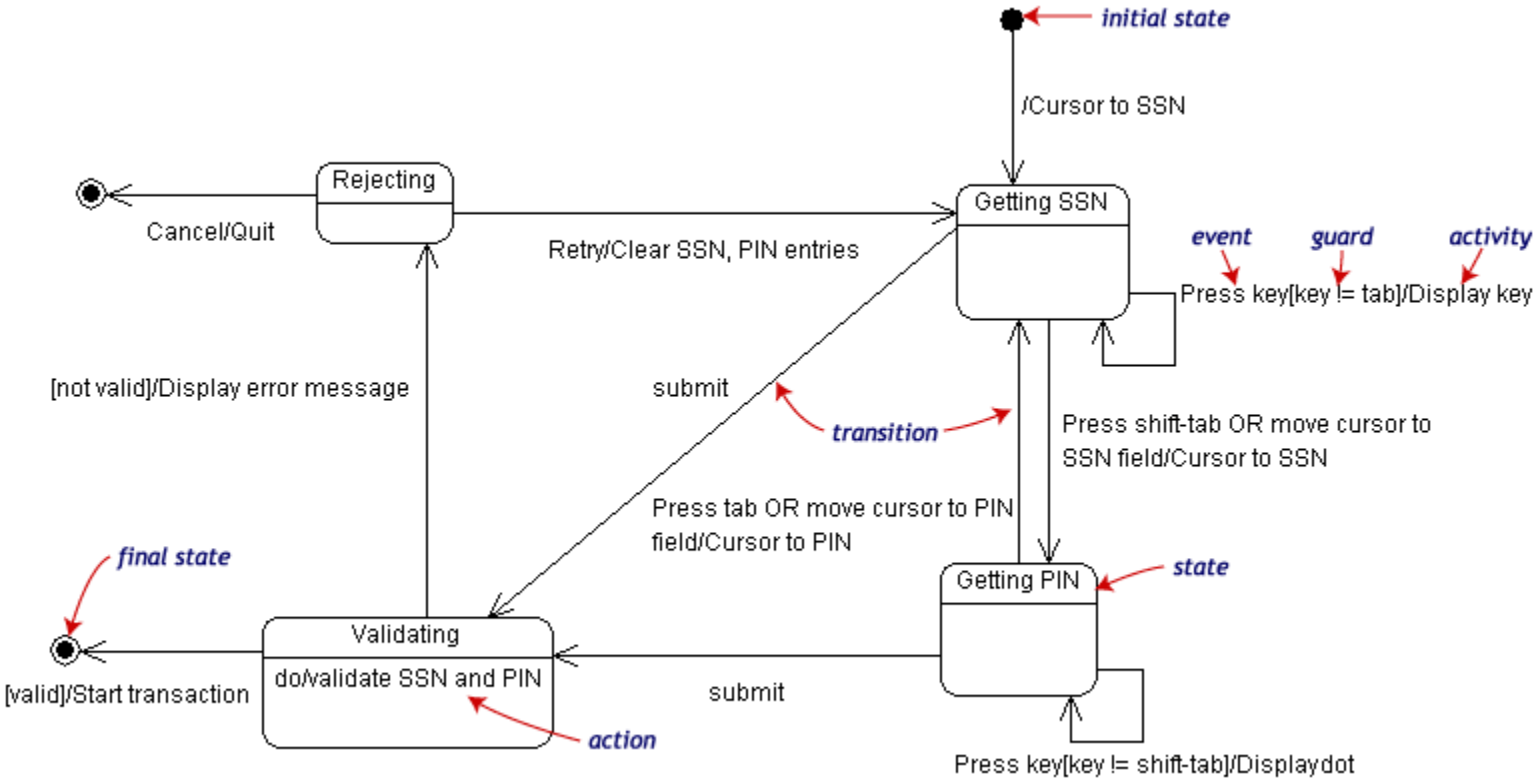


More Simple State Examples



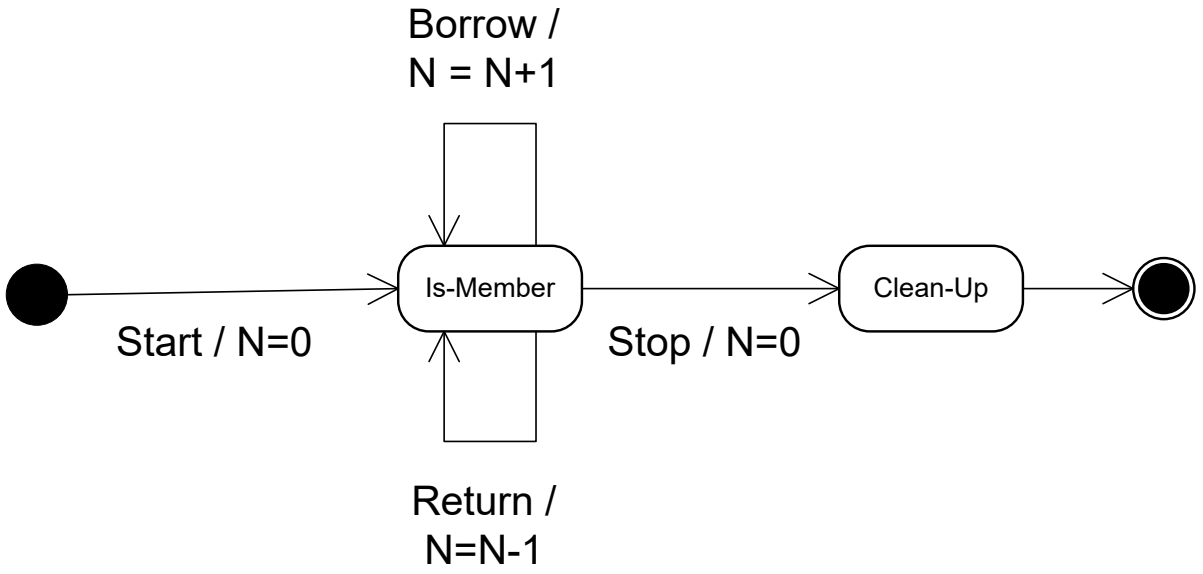
State Transition Example

Validating PIN/SSN



State Charts – Local Variables

- State Diagrams can also store their own local variables, do processing on them
- Library example counting books checked out and returned

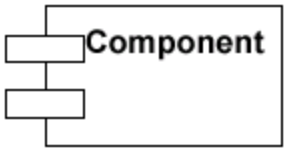


Component Diagrams

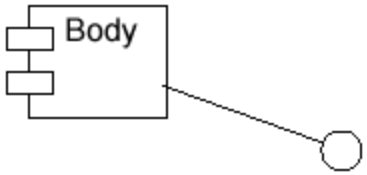
- Shows various components in a system and their dependencies, interfaces
- Explains the structure of a system
- Usually a physical collection of classes
  - Similar to a Package Diagram in that both are used to group elements into logical structures
  - With Component Diagrams all of the model elements are private with a public interface whereas Package diagrams only display public items.

Component Diagram Notation

- Components are shown as rectangles with two tabs at the upper left

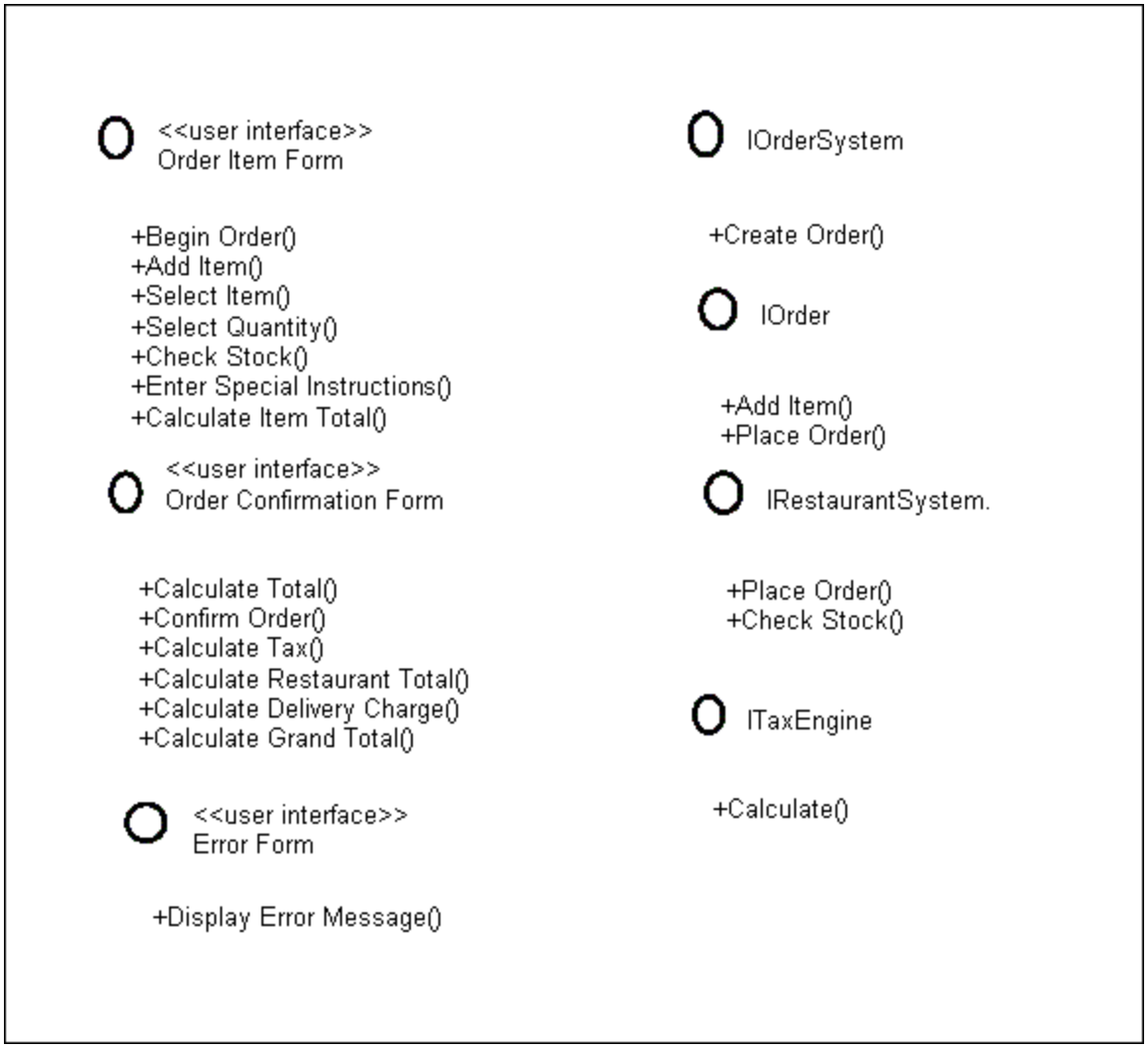


- Dashed arrows indicate dependencies
- Circle and solid line indicates an interface to the component



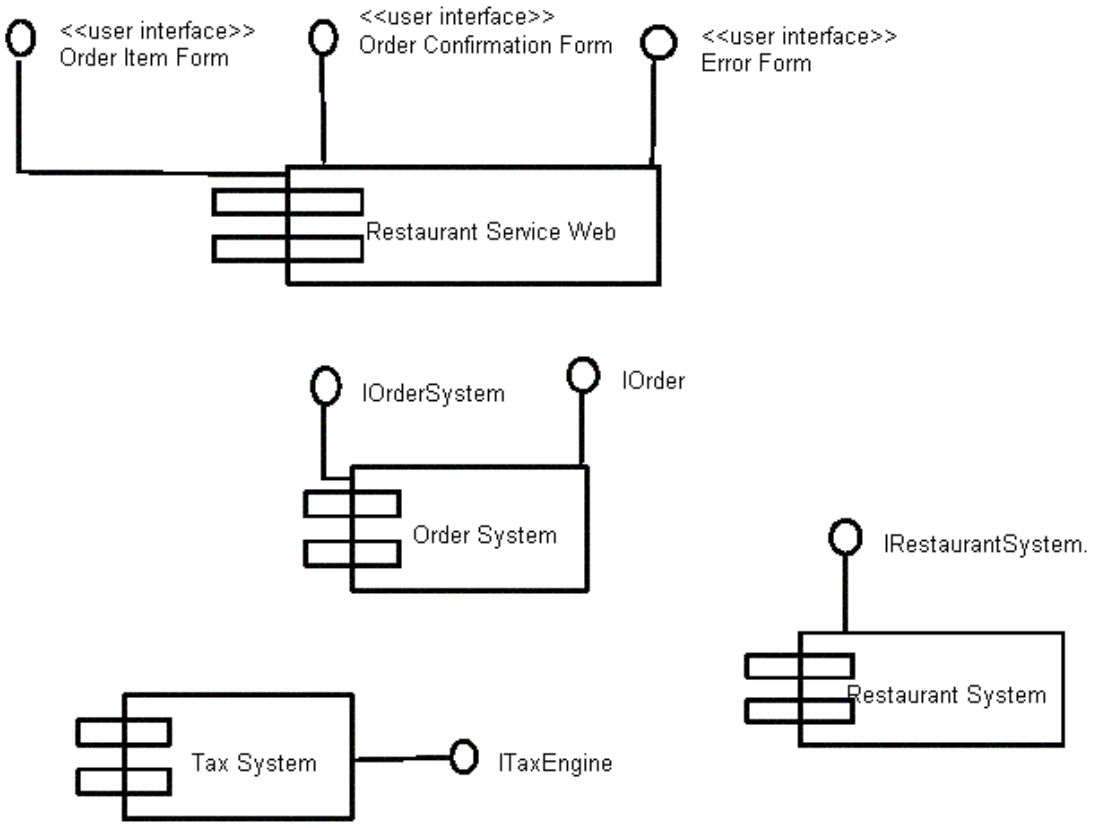
Component Example - Interfaces

- Restaurant ordering system
- Define interfaces first – comes from Class Diagrams



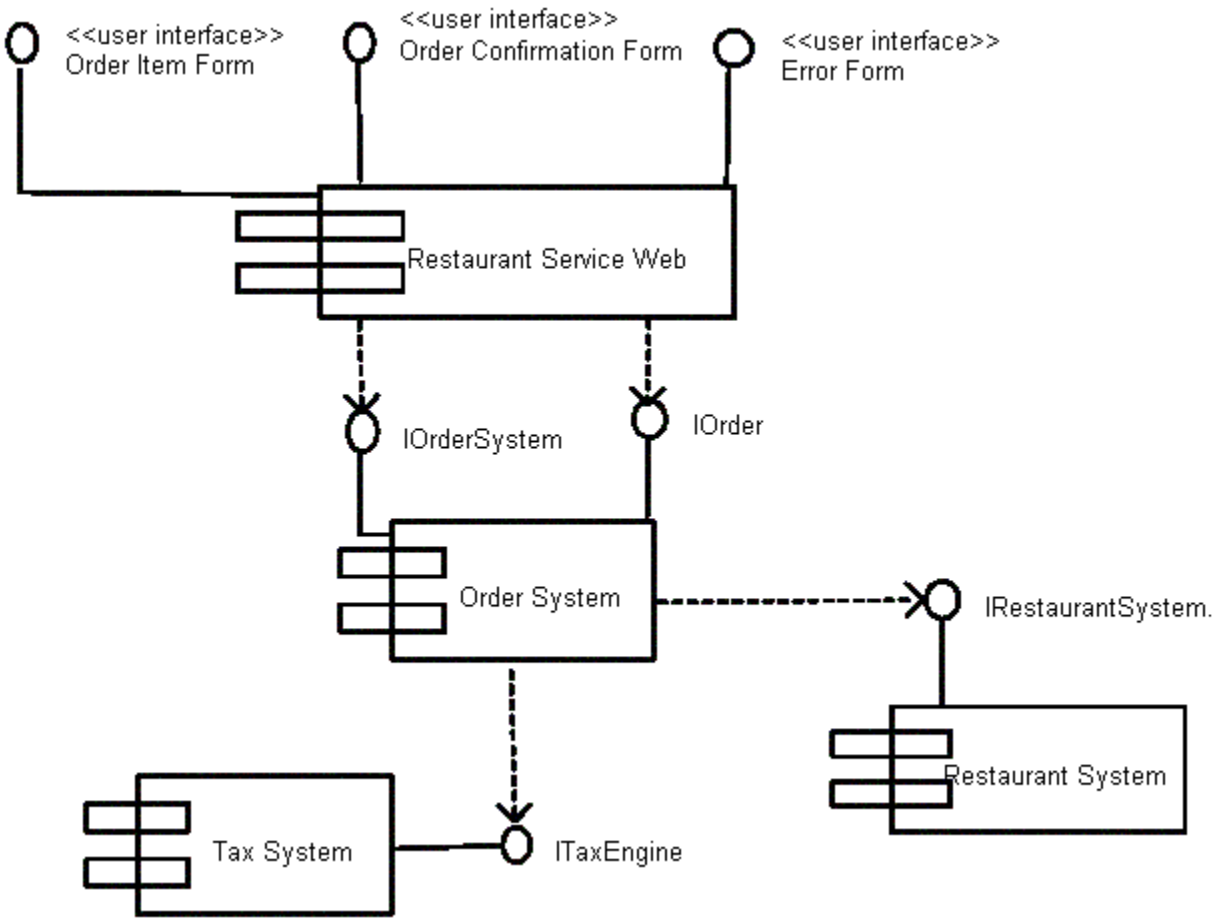
Component Example - Components

- Graphical depiction of components



Component Example - Linking

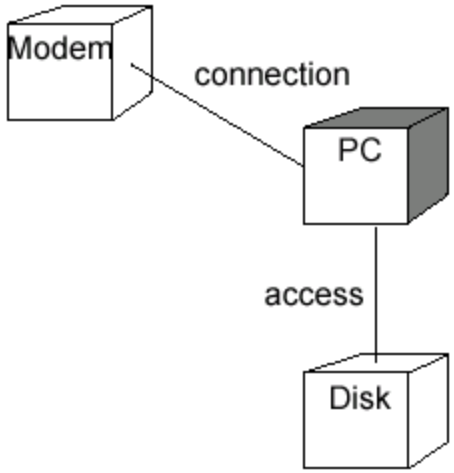
- Linking components with dependencies

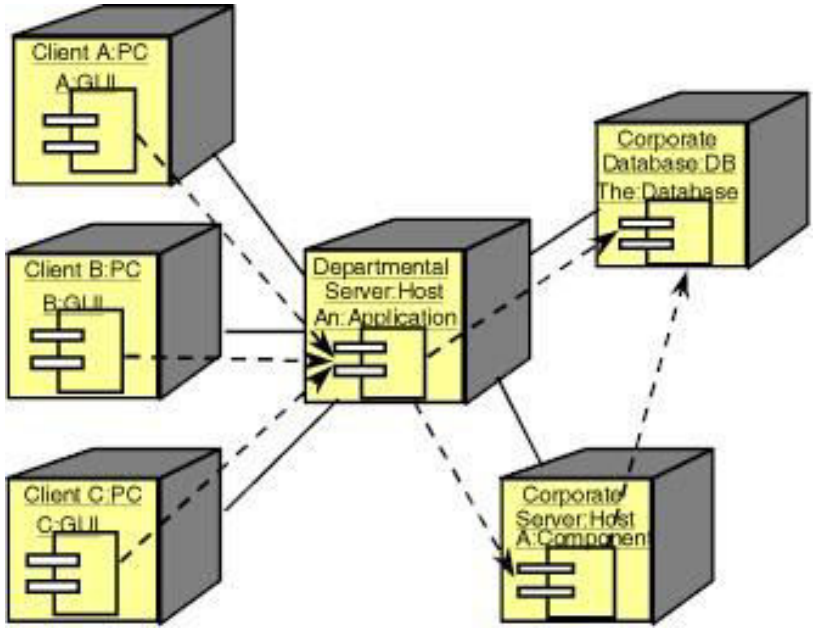


Deployment Diagrams

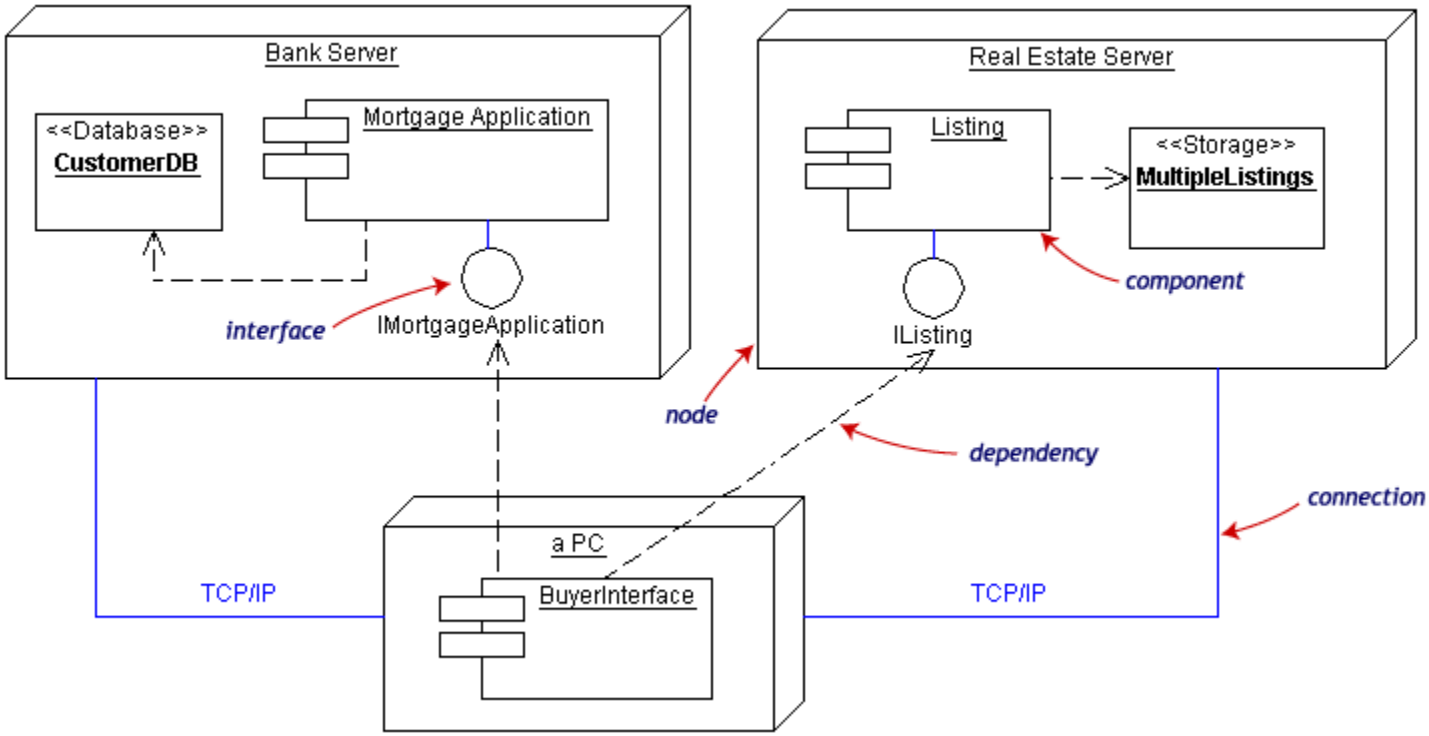
- Shows the physical architecture of the hardware and software of the deployed system
- Nodes
  - Typically contain components or packages
  - Usually some kind of computational unit; e.g. machine or device (physical or logical)
- Physical relationships among software and hardware in a delivered systems
  - Explains how a system interacts with the external environment

Some Deployment Examples





Deployment Example



Often the Component Diagram is combined with the Deployment

Summary and Tools

- UML is a modeling language that can be used independent of development
- Adopted by OMG and notation of choice for visual modeling
  - http://www.omg.org/uml/
- Creating and modifying UML diagrams can be labor and time intensive.
- Lots of tools exist to help
  - Tools help keep diagrams, code in sync
  - Repository for a complete software development project
  - Examples here created with TogetherSoft ControlCenter, Microsoft Visio, Tablet UML
  - Other tools:
    - Rational, Cetus, Embarcadero
    - See http://plg.uwaterloo.ca/~migod/uml.html for a list of tools, some free