Date: M T W T F S S

Topics that we have studied so far:

UML
Static vs. Dynamic Design
System, Models, views
Basic Modeling Steps
UML Baseline
- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
  - Sequence
  - Collaboration
- Activity Diagram
- State Transition Diagram
- Deployment Diagram

Class Diagrams
Instances
Association (Binary & Unary)
Aggregation
Composition
Inheritance
Multiplicities
Abstract class (written in italics)
Generalization
Specification
Realization

Object Diagrams
Objects and Links
Interaction Diagrams
State Invesient
Recursion
Syndronous
Asyndronous
Reply

Date: M T W T F S S

→ Static:
Describe code structure and object relations
Class relations
Objects at design time
Does not change

Dynamic:
Shows communication between objects
Similarity to class relations
Can follow sequences of events
May change depending upon execution scenario
Called Object diagrams

→ System: Aircraft
Models: Flight Simulator, scale model
Views: All blue prints, electrical wiring, fuel system
What is model?
An abstraction describing a (subsystem) subset of a system.
What is View?
It depicts aspects of model.
What is notation?
Set of graphical and textual rules for depicting views.

Date: M T W T F S S

→ What is Class Diagram?

- Gives an overview of a system by showing its classes and relationships among them.
- C. Diagrams are static.
- They display interacts but not what happens when they do interact.
- Shows attributes, operations of each class

→ Instances

- Represents a phenomenon.
- Name of an instance is underlined and can contain class of the instance.
- attributes are represented with their values

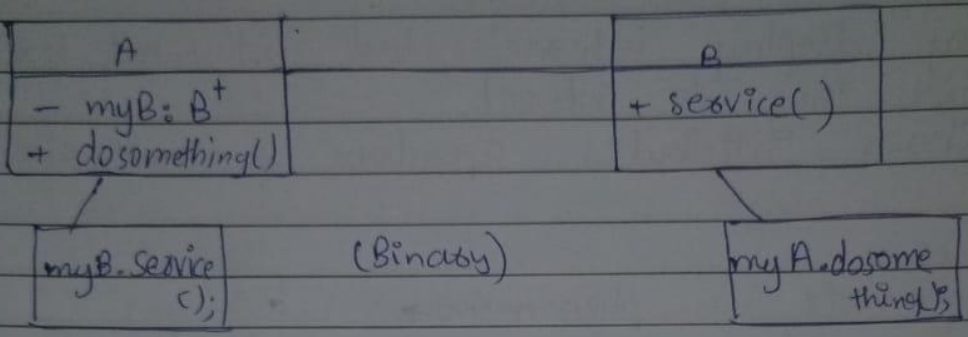
| tarif 1974 : Tariff schedule |
| --- |
| zone2 prices = {<br>{'1', .20},<br>{'2', .40},<br>{'3', .60}<br>} |

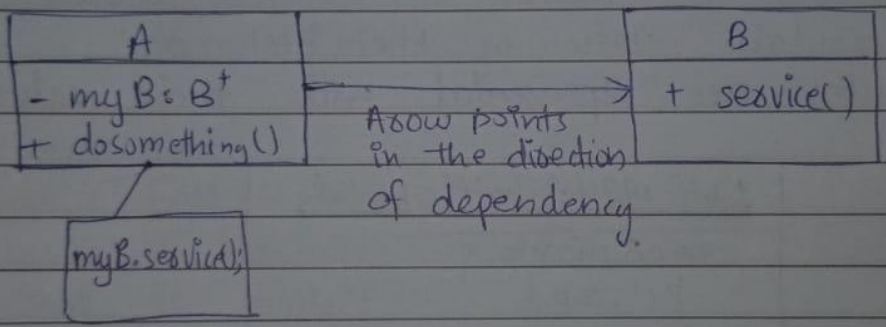
→ What is Association?

Represents a relationship between two or more objects in which the objects have their own lifetime and there is no owner.

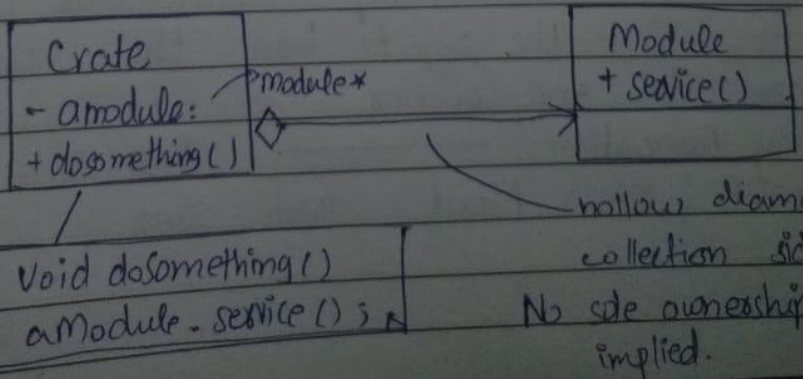
Binary Association: ——— Both entities A and B 'Know About' each other

Unary Association: ⟶
A knows about B but B know nothing about A.




→ What is Aggregation?
An association with a "collection-member"/ "part-whole" relationship.



Date: M T W T F S S

→ What is Multiplicities

Indication of how many objects may participate in the given relationship or the allowable number of instances of the element.

| | |
|---|---|
| 0..1 | zero or 1 instance. |
| 0..* | no limit |
| 1 | exactly 1 instance |
| 1..* | atleast one instance |

→ What is abstract class?

Represent by writing in italics.

An abstract class permits you to make functionality that subclasses can implement or override whereas interface only permits you to state functionality but not implement it.

A class can extend only one abstract class while a class can implement multiple interfaces.

→ What is generalization?

A relationship that implements inheritance occurs between two entities such that one entity is parent and other one is child

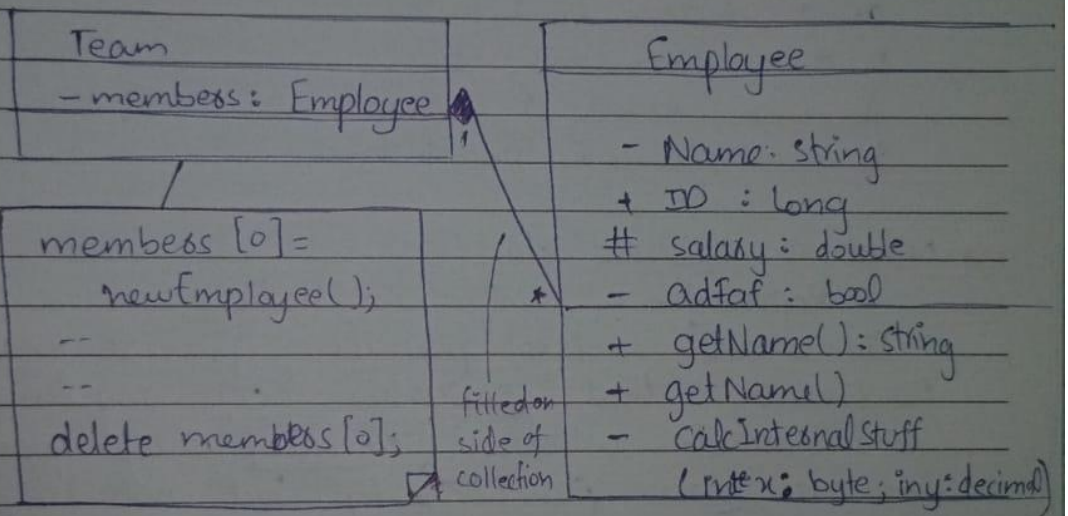
KING'S NOTES

Date:

M T W T F S S

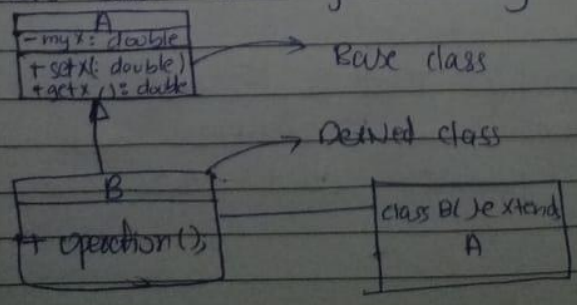
→ What is Composition?

- It is aggregation with lifetime control (owner control construction, destruction)
- Part object may belong to only one whole object




→ What is Inheritance?

The ability of one class (childclass) to inherit the identical functionality of another class (Super class) and then add new functionality of its own.




KING'S NOTES

Date: M T W T F S S

→ What is Object diagram?

Object diagram is snapshot of the objects in a system.

Shows instances of class Diagrams and links among them.

→ How class will communicate?

Synchronous: ——▶

Synchronous message requires a response before the interaction can continue.

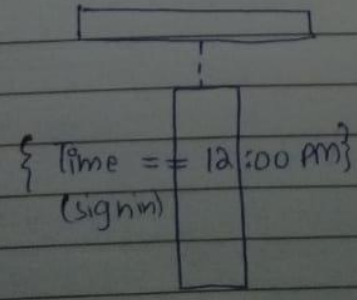
Asynchronous: ——→ and ——→

Don't need a reply for interaction to continue.

Reply: <------

→ State Invariant:

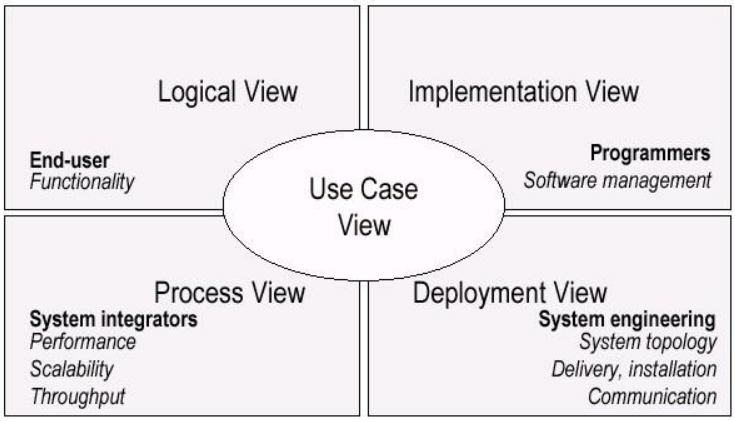
It is a condition applied to a lifeline, which must be fulfilled for the life line to exist.




KING'S NOTES

# 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




# 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

Date: M T W T F S S

# UML (Unified Modeling Language)

- Visual language

(Non-programming language) To define a standard way to visualize the way your system has been designed.

- Need?

To save time.

Explanation: Situation: You are creating a system many teams are working (some programmars and some non programmars.) A programmar presents a code of particular language before non programmar that now this system will perform, work and flow of system, due to it, non-programmars will take time to understand and work will be delay might be possible cross the deadline. Time is very important element in designing system. UML save you.

**who developed it?**

Grady Booch, Jim Rumbaugh and Ivar Jacobson developed UML in mid-1990s with much feedback from software development community.

Date:

M T W T F S S

# Class Diagrams

**Class diagram**

To model classes, including their **attributes**, **operations** and their **relationships** and **association** with other classes, UML provides class diagram

**Static Dynamic**

Provides static or structural view.
Don't show dynamic nature of communications

**Main elements**

main elements are boxes, which are icons used to represent classes & interfaces.

Divided into three horizontal parts.

| name of class |
|---|
| attributes |
| operations/ behaviours |

fig (a)

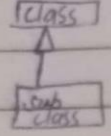
| Throughbred |
|---|
| - father: Throughbred |
| - mother: Throughbred |
| - birthyear: int |
| + get father ( ): |
| + get mother ( ): |
| + get current age (current Year Date): |

fig(a) presents a simple eg of 'Thoroughbred' class that models thoroughbred (ঘোড়া) horses. It has three attributes displayed - mother, father and birth year. The diagram also shows three operations: getCurrentAge( ); getFather( ); and getMother( ); Each attribute can have a name, type and level of visibility. - (private), + (public), ~ (package) # (protected).

KING'S NOTES

Date:

**Generalization**




The arrow points from subclass to the superclass, such a relationship is called a generalization.

**Realization:**

(An arrow with a dashed line from the arrow shaft indicates implementation of interface, such a relationship is called realization.)

**Association**

—— or → or ←

**Types**

Represented by solid lines.

Can have one or on both ends show navigability

At one end shows one-way navigability

جس کی طرف arrow ہے اس کے بارے میں پہلی کلاس کے پاس آسانی سے access کرنے کا اختیار ہے مگر پہلی کلاس کو دوسری کلاس access نہیں کر سکتی۔

When there is no arrow it means two-way communication.

دونوں ایک دوسرے کو access کر سکتے ہیں۔

**Composition** (filled diamond)

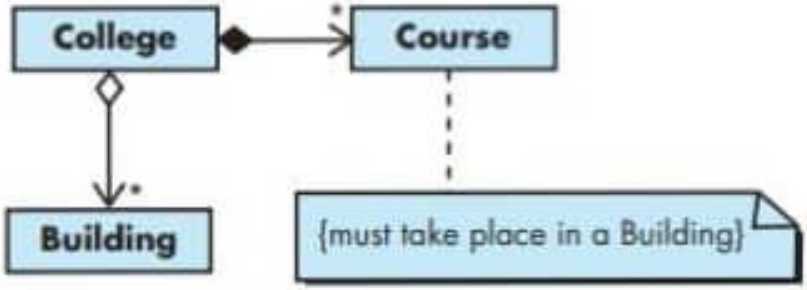
A composition is an aggregation indicating strong ownership of the parts. (Part/whole) parts live and die with owner because they have no role in the Software system independently of the owner.

**Aggregation**

Special kind of association, presented by hollow diamond. (whole/part)

KING'S NOTES

Another common element of a class diagram is a *note*, which is represented by a box with a dog-eared corner and is connected to other icons by a dashed line. It can have arbitrary content (text and graphics) and is similar to a programming language comment. It might contain information about the role of a class or constraints that all objects of that class must satisfy. If the contents are a constraint, braces surround the contents. Note the constraint attached to the **Course** class in Figure A1.3.




The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a non-negative integer or by a range of integers. A multiplicity specified by "0..1" means



that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a **Person** class and a **SocialSecurityNumber** class in a class diagram. A multiplicity specified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*" means zero or more. An * was used as the multiplicity on the **OwnedObject** end of the association with class **Person** in Figure A1.2 because a **Person** can own zero or more objects.

## Use-Case Diagrams

Use cases (Chapters 8 and 9) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we'll create some for a software application for managing an online digital music store. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

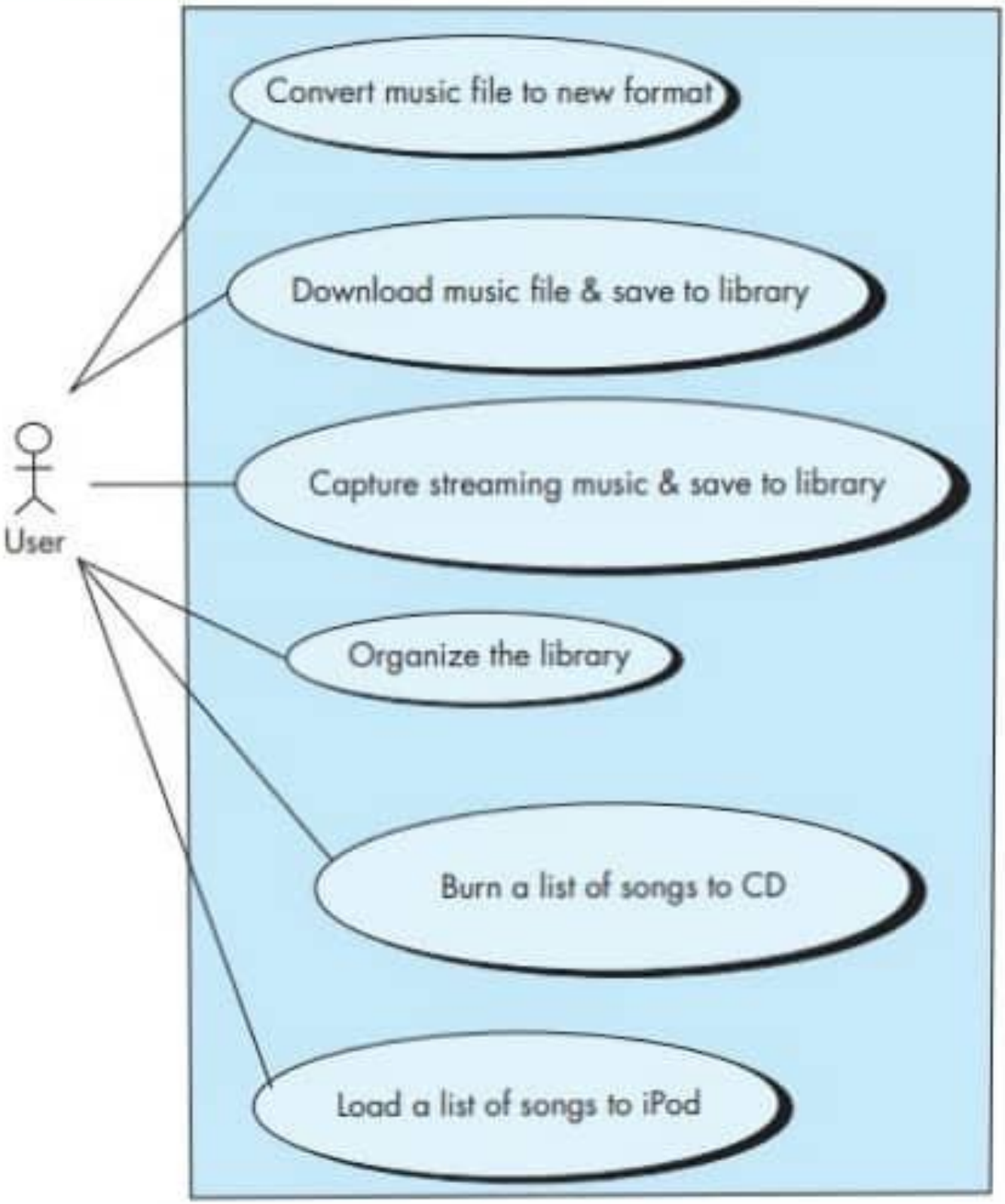
This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD). Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

In this diagram, the stick figure represents an *actor* (Chapter 8) that is associated with one category of user (or other interaction element). Complex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the


Convert music file to new format
Download music file & save to library
Capture streaming music & save to library
Organize the library
Burn a list of songs to CD
Load a list of songs to iPod
User


Such inclusion is indicated in use-case diagrams, as in Figure A1.6, by means of a dashed arrow labeled «include» connecting a use case with an included use case.

A use-case diagram, because it displays all use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In our digital music organizer, we would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most valuable contribution of use cases to the software development process is the textual description of each use case, not the overall use-case diagram [Fow04]. It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod or smartphone. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps.

## Use-Case Diagrams

Use cases (Chapters 8 and 9) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we'll create some for a software application for managing an online digital music store. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD). Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

In this diagram, the stick figure represents an *actor* (Chapter 8) that is associated with one category of user (or other interaction element). Complex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the actors are outside the system.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod or smartphone. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps.

# 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:*

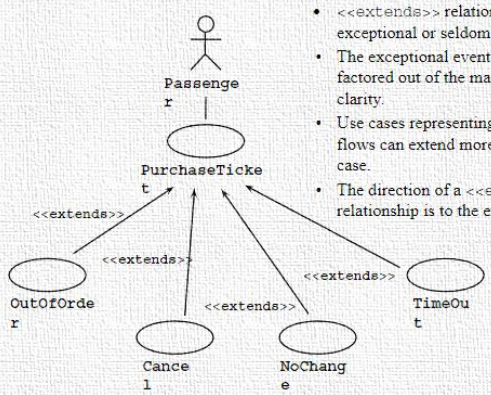
- Passenger has ticket.

*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.

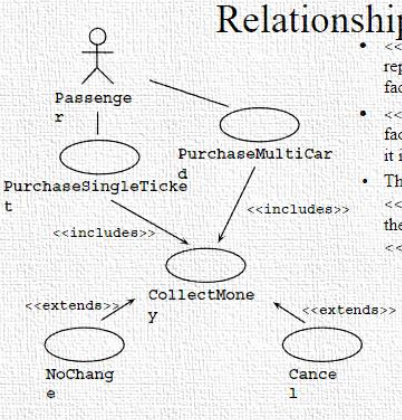
Anything missing? Exceptional cases!

## The <<extends>> Relationship



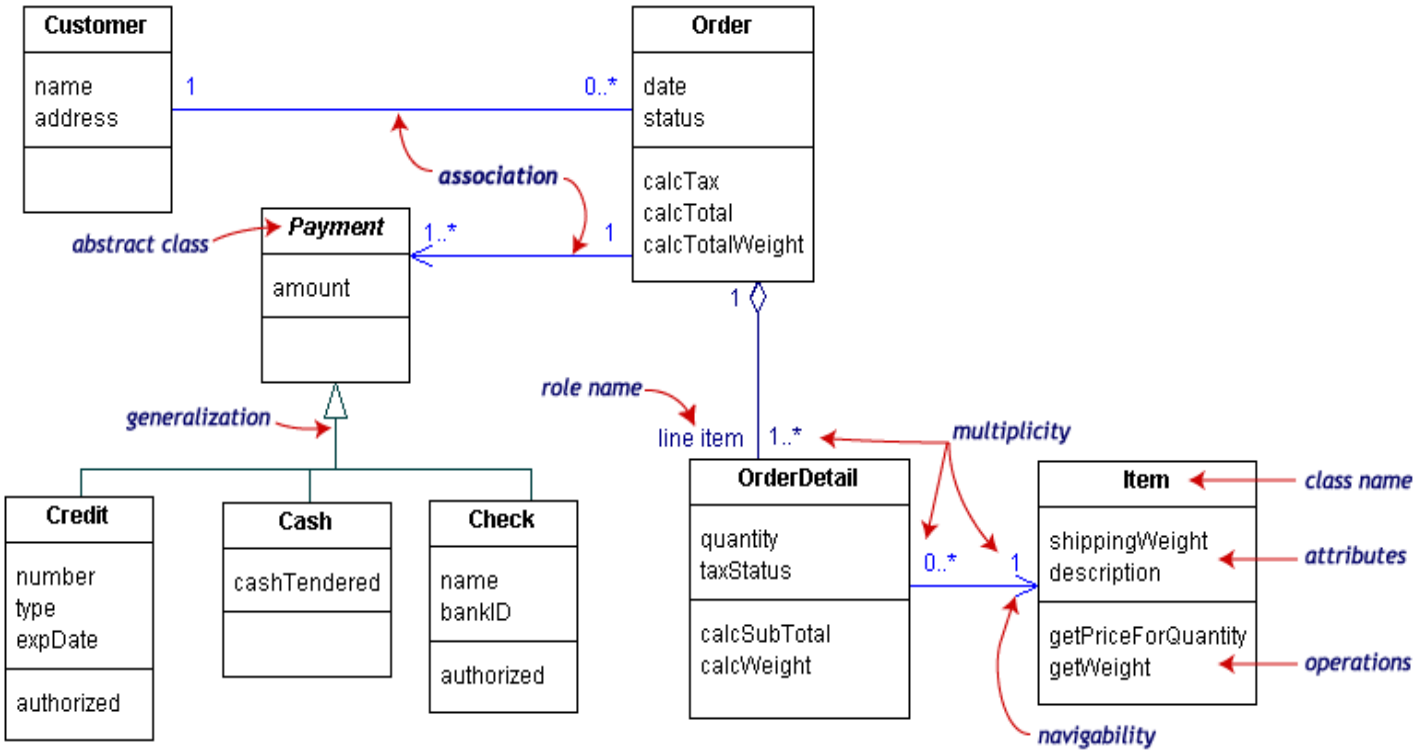

- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended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

## The <<includes>> Relationship

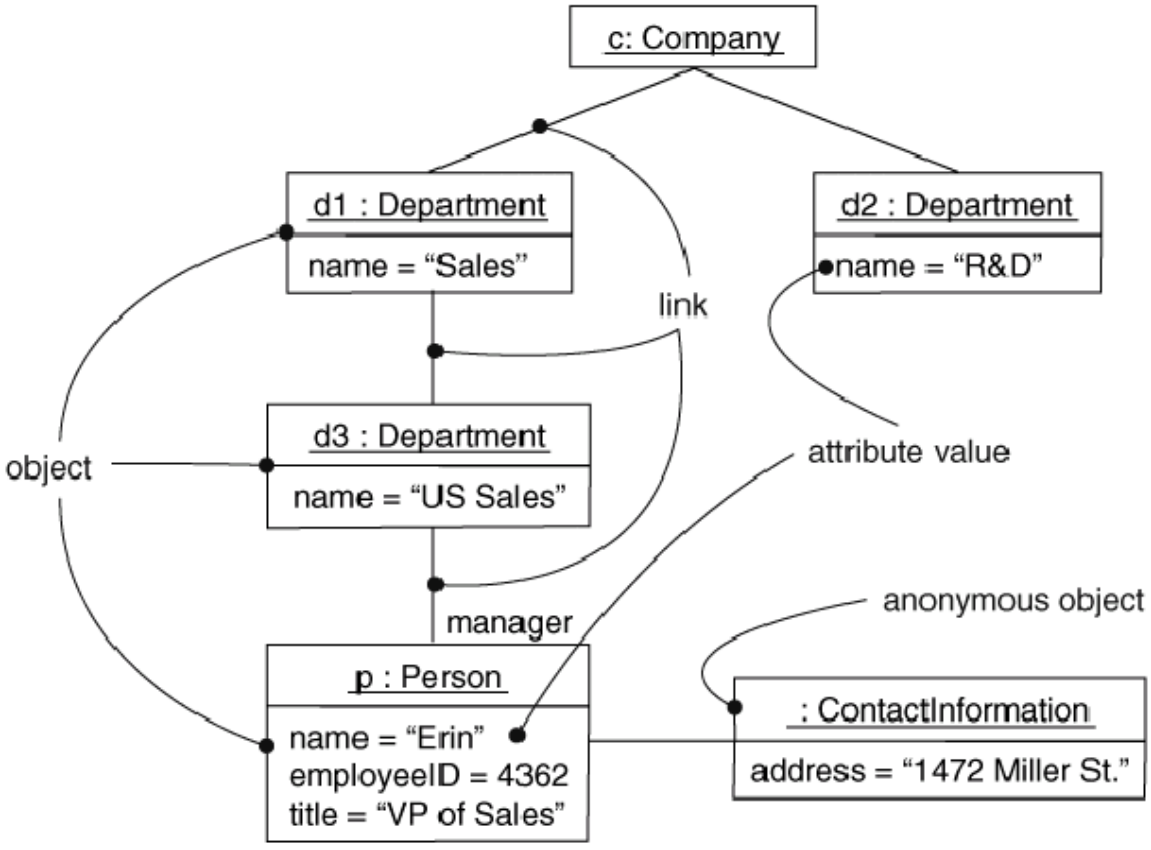



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

# UML Class Example




# Objects and Links



An **anonymous object** is essentially a value that has no name. Because they have no name, there's no way to refer to them beyond the point where they are created. Consequently, they have "expression scope", meaning they are created, evaluated, and destroyed all within a single expression.
Here is the add() function rewritten using an anonymous object:

```cpp
#include <iostream>

void printValue(int value)
{
    std::cout << value;
}

int main()
{
    printValue(5 + 3);

    return 0;
}
```

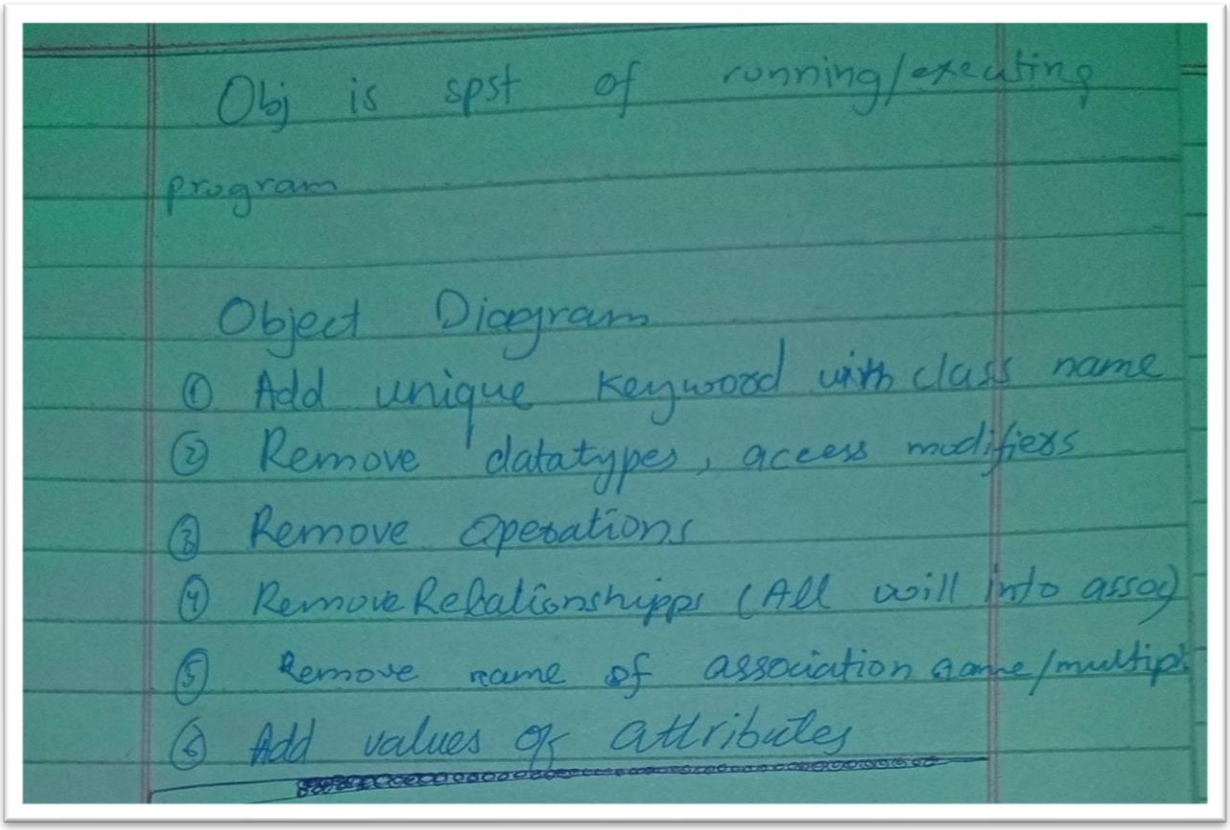
In this case, the expression 5 + 3 is evaluated to produce the result 8, which is placed in an anonymous object. A copy of this anonymous object is then passed to the printValue() function, (which prints the value 8) and then is destroyed.

Note how much cleaner this keeps our code -- we don't have to litter the code with temporary variables that are only used once.

**Anonymous class objects**

Although our prior examples have been with built-in data types, it is possible to construct anonymous objects of our own class types as well. This is done by creating objects like normal, but omitting the variable name.

```cpp
Cents cents{ 5 }; // normal variable
Cents{ 7 }; // anonymous object
```



Date: M T W T F S S

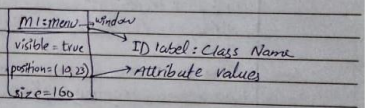
## Object diagrams

**Object diagram**

Snapshot of the objects in a system.
- at a point in time.
- with selected focus:
  - ⇒ Interactions – Sequence diagram
  - ⇒ Message passing – Collaboration diagram
  - ⇒ Operation – Deployment diagram

**Format**

Format :
- Instance name : Class name
- Attributes and values




**Symbols**

Some symbols:

| | |
|---|---|
| :class | class |
| C1:class | object |
| x[K] : Y | complex |

obj type class

read as :
(x) object of class(y) of type(K)

KING'S NOTES

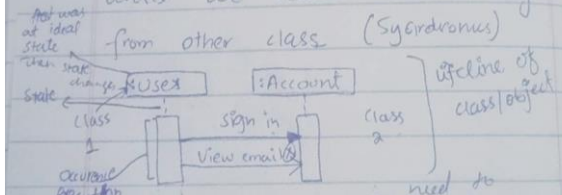
How a class will communicate with other.

Syndronous ——►

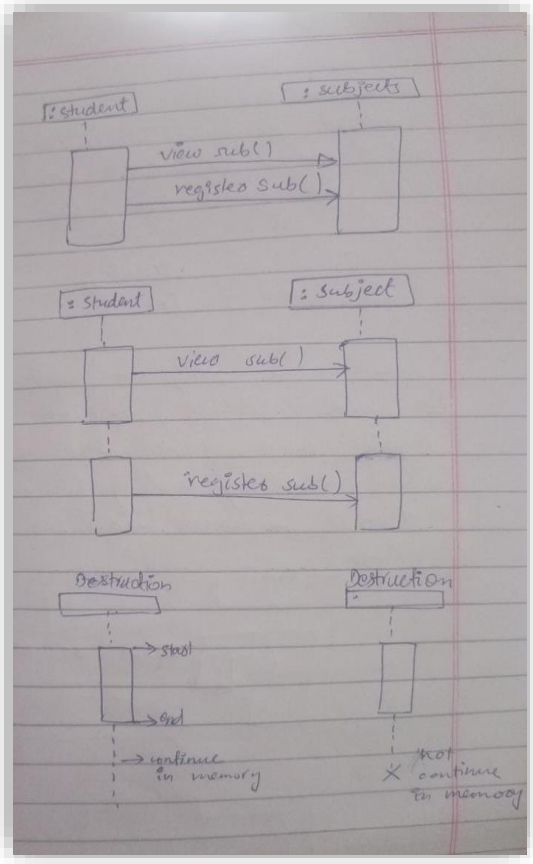
Asy- ——→

reply <- - - -

Gmail → email → (sign in) where we can't perform a task until we have received msg from other class. (Syndronous)




where we don't need to receive msg and performing task is asynchronous.

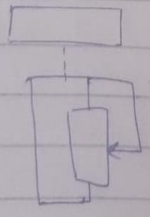



Recursion

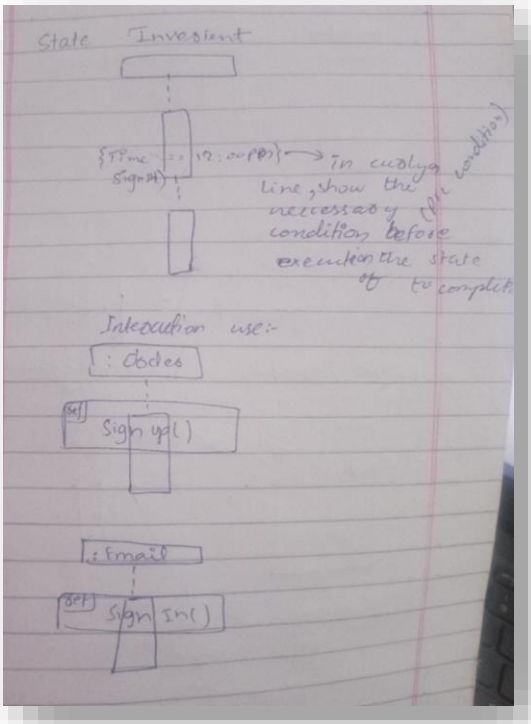
When program is calling itself or a class calls another class and that class this one.

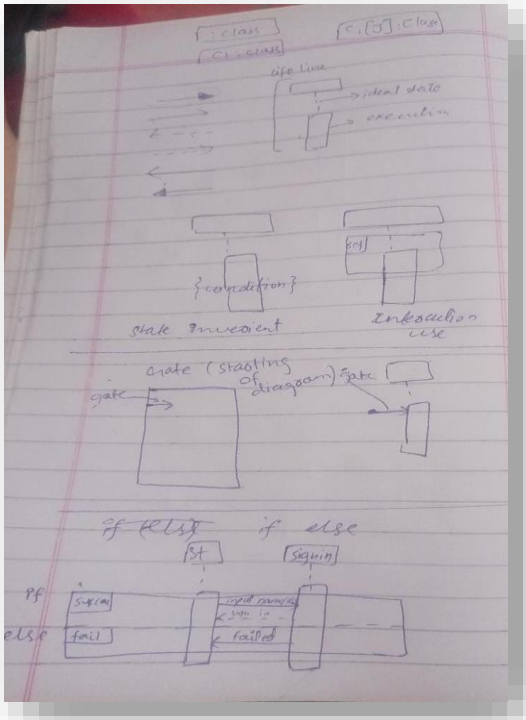
Ending and starting are defined for it.

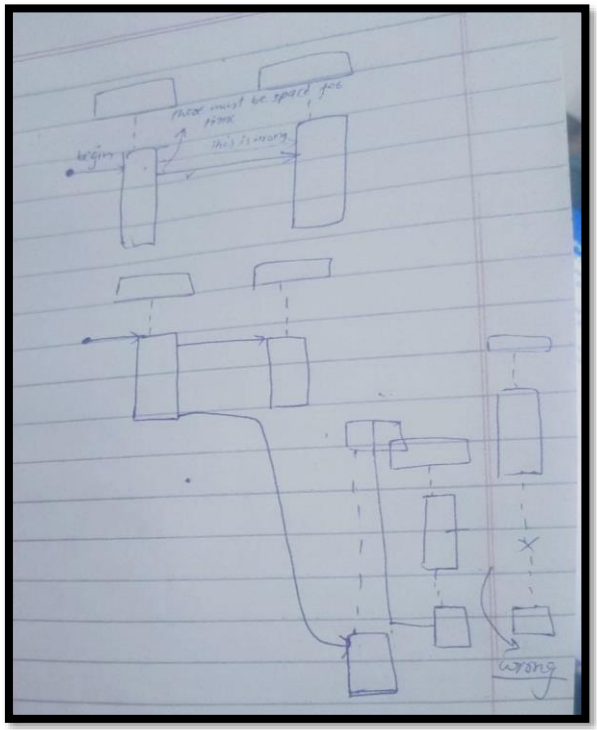
When a program continues to class itself, excution starts and never ends up when ~~calling~~ ending is not defined → basic virus.



shows recursion


State Invesient
Interaction use:-
: Codes
Sign up()
: Email
Sign In()

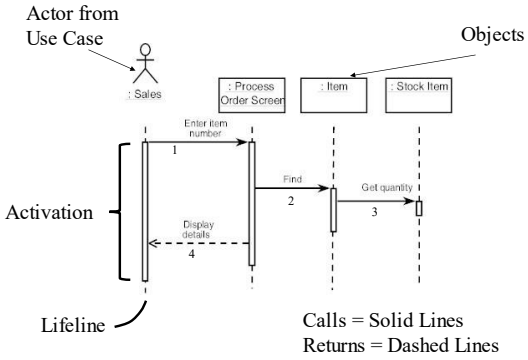

life line
ideal state
execution
{condition}
State Invesient
Interaction use
gate (starting of diagram)
if else
Sign in
success
fail
failed


begin
wrong

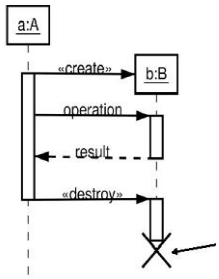
# Interaction Diagrams

- 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

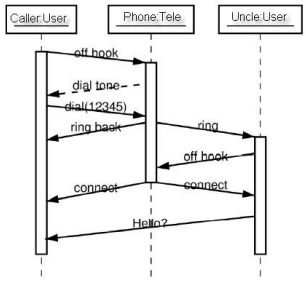



# Sequence Diagram : Destruction




# Sequence Diagram : Timing

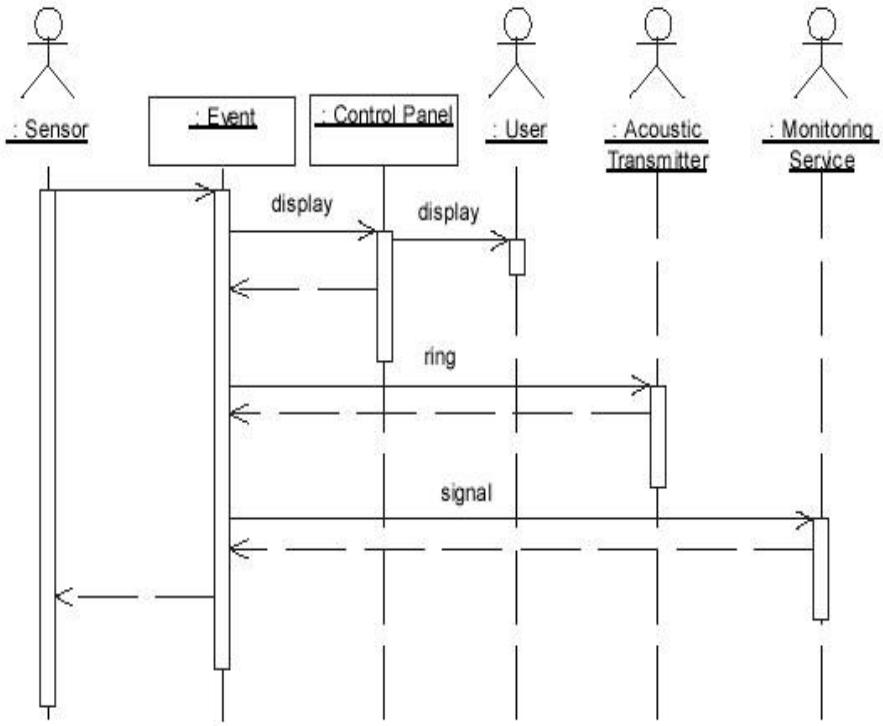
Slanted Lines show propagation delay of messages
Good for modeling real-time systems




If messages cross this is usually problematic – race conditions

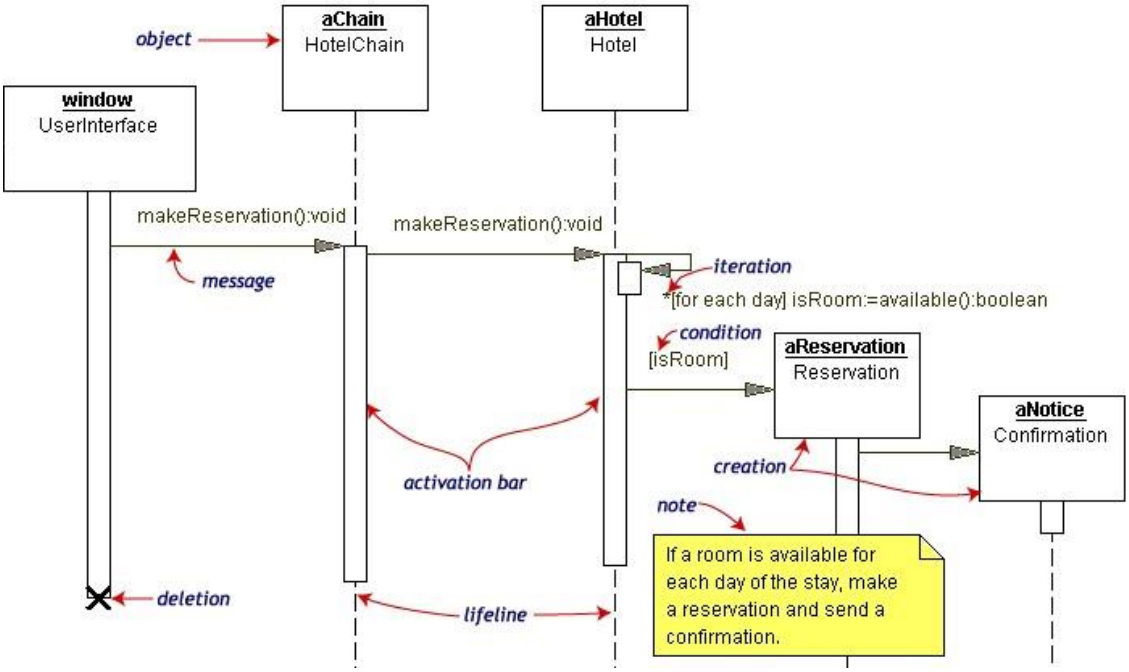
# Sequence Example: Alarm System

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

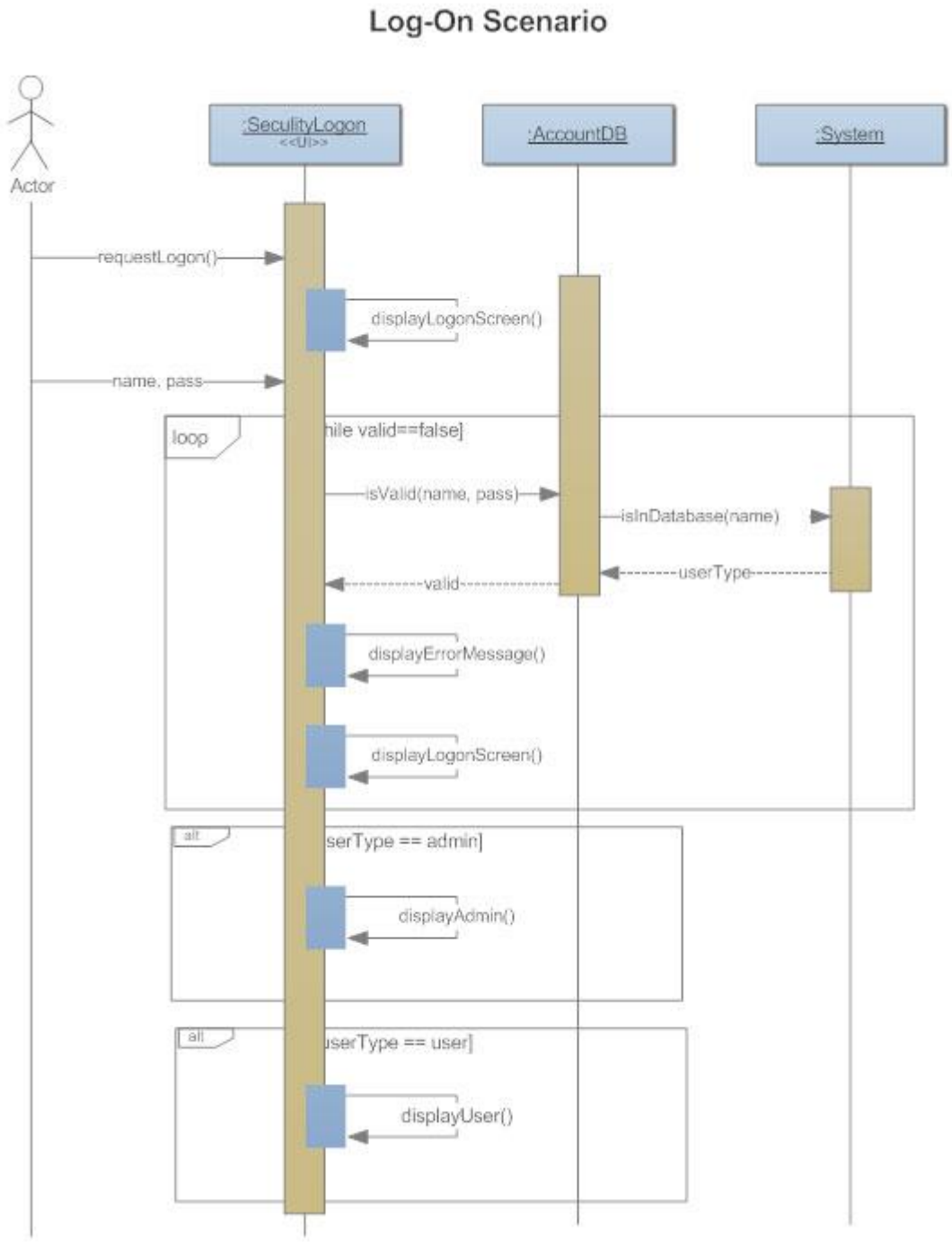


# Sequence Diagram Example

Hotel Reservation



# Sequence Diagram(copy-paste from website url is given below with title sequence diagram)




## What is a Sequence Diagram?

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time. They're also called event diagrams. A sequence diagram is a good way to visualize and validate various runtime scenarios. These can help to predict how a system will behave and to discover responsibilities a class may need to have in the process of modeling a new system.

# Sequence Diagram Tutorial

Start with one of SmartDraw's included sequence diagram templates. You'll notice that all the notations and symbols you need are docked to the left of your drawing area. Simply stamp them to your page and connect the symbols.

- Model and document how your system will behave in various scenarios
- Validate the logic of complex operations and functions

Learn how to draw UML diagrams of all kinds with SmartDraw.

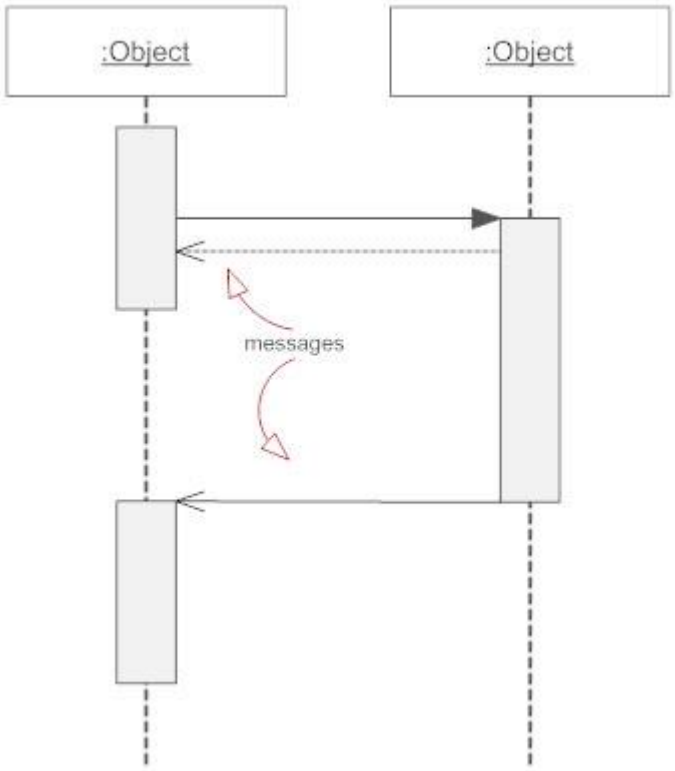


## Basic Sequence Diagram Notations

**Class Roles or Participants**
Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

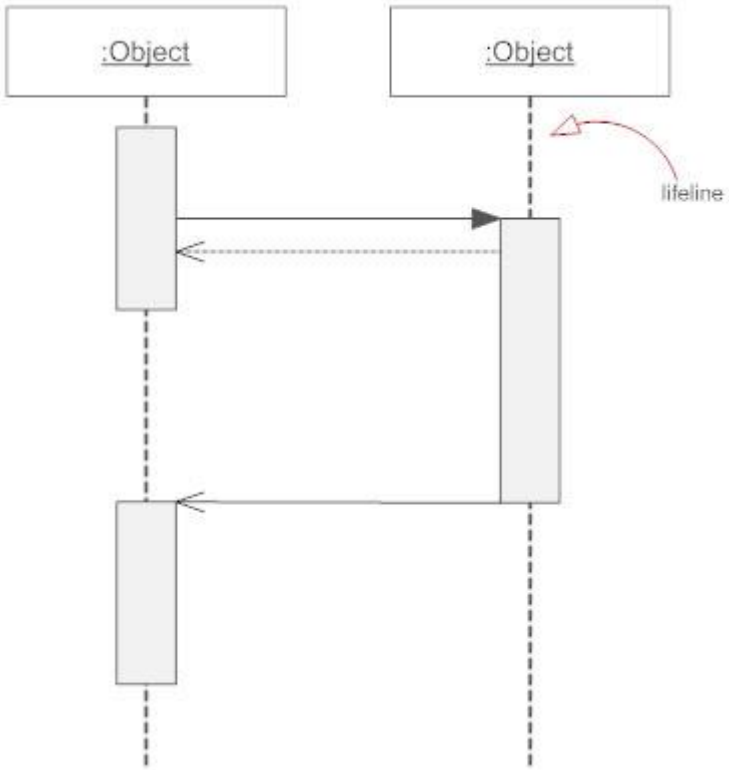

**Activation or Execution Occurrence**
Activation boxes represent the time an object needs to complete a task. When an object is busy executing a process or waiting for a reply message, use a thin gray rectangle placed vertically on its lifeline.


**Messages**
Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks. For message types, see below.




**Lifelines**

Lifelines are vertical dashed lines that indicate the object's presence over time.




**Destroying Objects**

Objects can be terminated early using an arrow labeled "<< destroy >>" that points to an X. This object is removed from memory. When that object's lifeline ends, you can place an X at the end of its lifeline to denote a destruction occurrence.

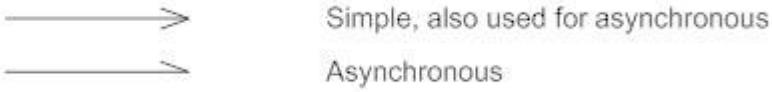
**Loops**

A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].

## Types of Messages in Sequence Diagrams

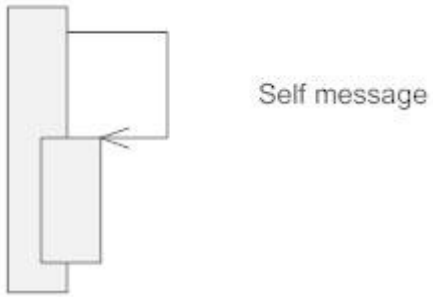
**Synchronous Message**
A synchronous message requires a response before the interaction can continue. It's usually drawn using a line with a solid arrowhead pointing from one object to another.


**Asynchronous Message**
Asynchronous messages don't need a reply for interaction to continue. Like synchronous messages, they are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted.

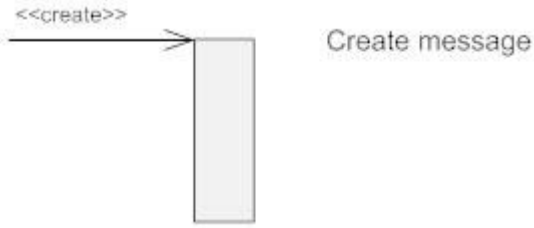



**Reply or Return Message**
A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline.


**Self Message**
A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.

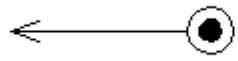



**Create Message**
This is a message that creates a new object. Similar to a return message, it's depicted with a dashed line and an open arrowhead that points to the rectangle representing the object created.

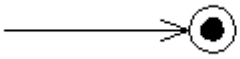



**Delete Message**
This is a message that destroys an object. It can be shown by an arrow with an x at the end.


**Found Message**
A message sent from an unknown recipient, shown by an arrow from an endpoint to a lifeline.




**Lost Message**
A message sent to an unknown recipient. It's shown by an arrow going from a lifeline to an endpoint, a filled circle or an x.




## Sequence Diagram Examples

The best way to understand sequence diagrams is to look at some examples of sequence diagrams.

Click on any of these sequence diagrams included in SmartDraw and edit them:

Sequence Diagram - Log On Scenario

Sequence Diagram - Shopping Cart

Class diagram
https://www.ictdemy.com/software-design/uml/uml-class-diagram
Abstract class and interface
https://www.guru99.com/interface-vs-abstract-class-java.html
http://cse.csusb.edu/dick/cs202/abstraction.html
Abstraction and uml
https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-abstraction-relationships
Links of youtube to clear concepts of the usecase and uml
https://youtu.be/r1ex_V6WsOs
https://youtu.be/hKrB34O6E0M
https://youtu.be/bXsqoeY0B1w
https://youtu.be/IPxLHKysEws

link of website for usecase diagrams
https://www.edrawsoft.com/article/use-case-diagram-examples.html

Sequence Diagram
https://www.smartdraw.com/sequence-diagram/

Lots of tools exist to help(from slides)

- 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

## Collaboration diagram

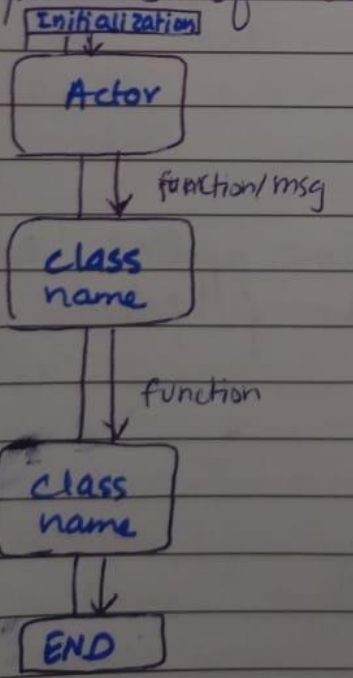
Also known as interaction diagram, or communication diagram.

These diagrams can be used to portray the dynamic behavior of particular use case and define the role of each object.

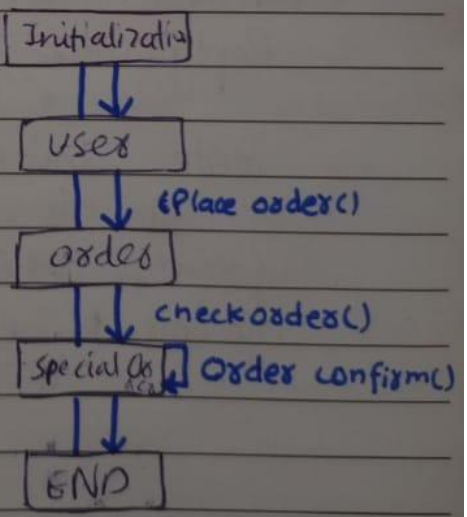
### What are 3 primary elements of C.Diagram?

1) Objects
2) links
3) Messages

### Components of collaboration diagram




### Example




KING'S NOTES

Explanation of elements

1. Objects- Objects are shown as rectangles with naming labels inside. The naming label follows the convention of object name: class name. If an object has a property or state that specifically influences the collaboration, this should also be noted.
2. Actors- Actors are instances that invoke the interaction in the diagram. Each actor has a name and a role, with one actor initiating the entire use case.
3. Links- Links connect objects with actors and are depicted using a solid line between two elements. Each link is an instance where messages can be sent.
4. messages- Messages between objects are shown as a labeled arrow placed near a link. These messages are communications between objects that convey information about the activity and can include the sequence number.

https://cdn.ttgtmedia.com/rms/onlineimages/whatis-collaboration_diagram.png

Date: M T W T F S S

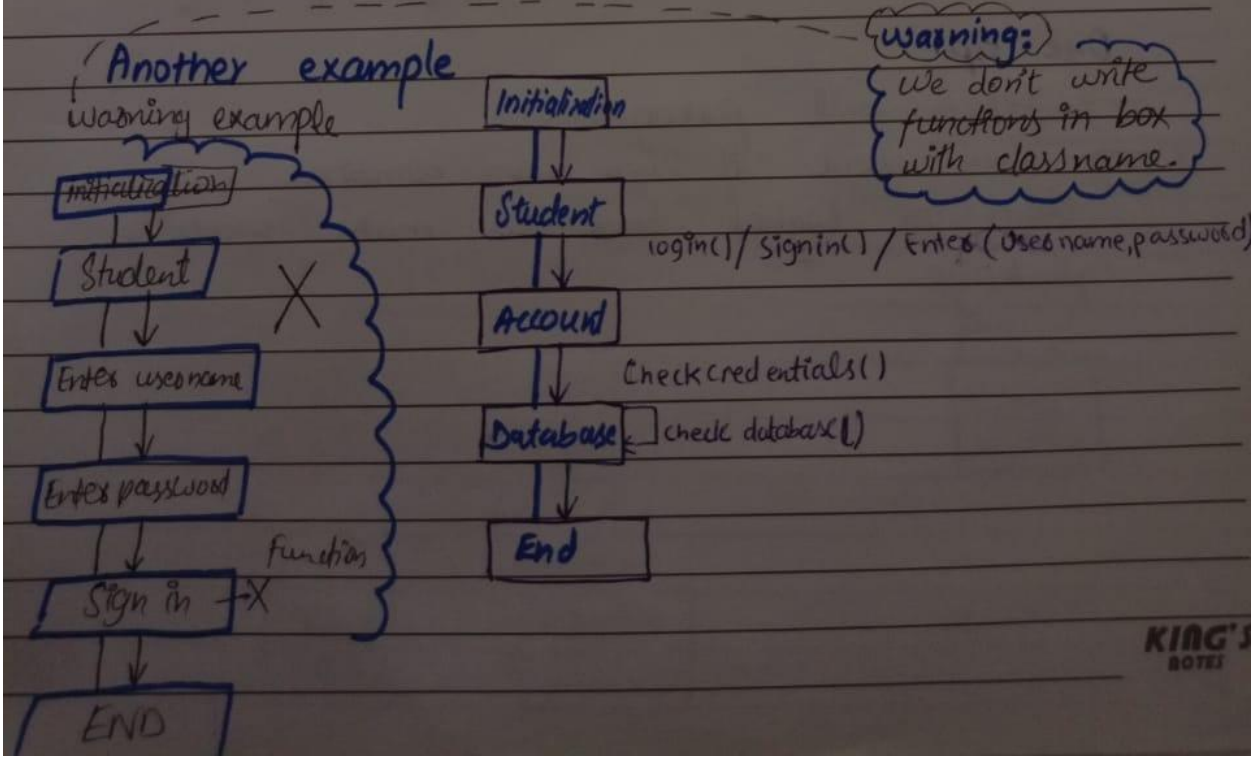
## Collaboration vs Sequence diagram

| | |
|---|---|
| → Vertical flow of info | → Horizontal flow of info |
| → starting and ending is shown. | → Here starting (Gate) is shown but not ending. |
| → focus on order of messages that flow between objects. | → Timing is important element of sequence diagram. |

**Why we use collaboration diagram?**

To visualize the structural organization of objects and their interactions.
To increase the readability.

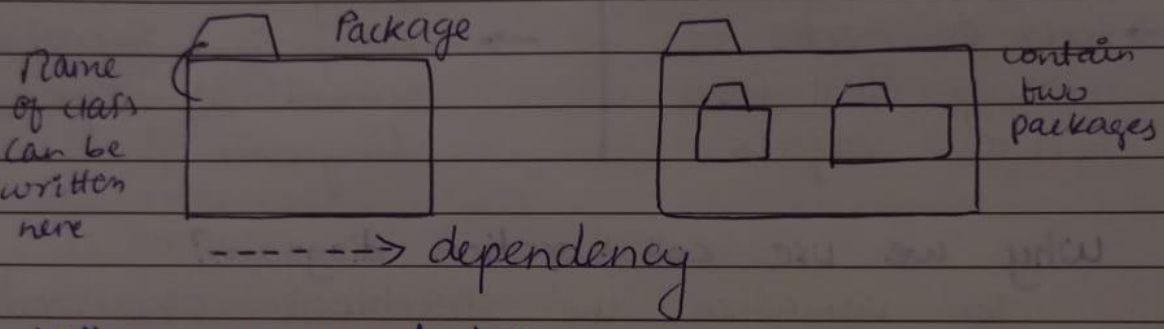
**Another example**



Date: M T W T F S S

# Package Diagram

Combination of related classes, collection of logically related UML elements such as (diagrams documents, classes or even other packages.
Static diagram that's why only one diagram is required.




## Why we need it?

To increase readability of program

**One tip:**

→ Draw all packages
→ Then check if you can combine any of them on logical base to make combine package.

Notation for Package diagram

- 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

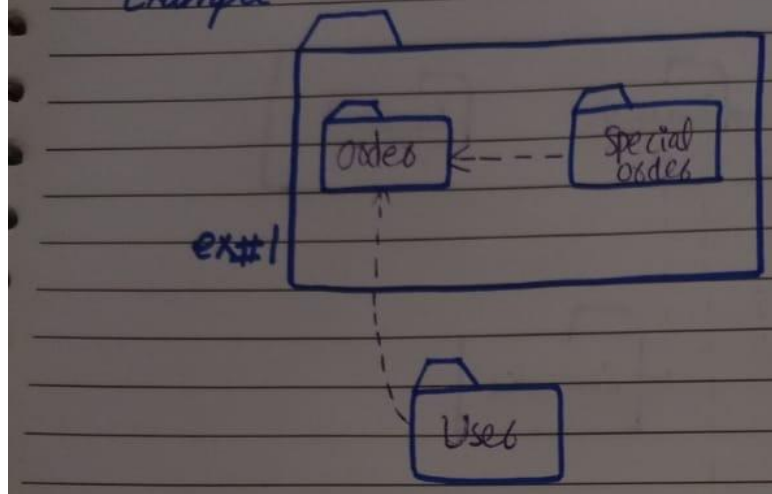
## Benefits of a package diagram

A well-designed package diagram provides numerous benefits to those looking to create a visualization of their UML system or project.

- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.

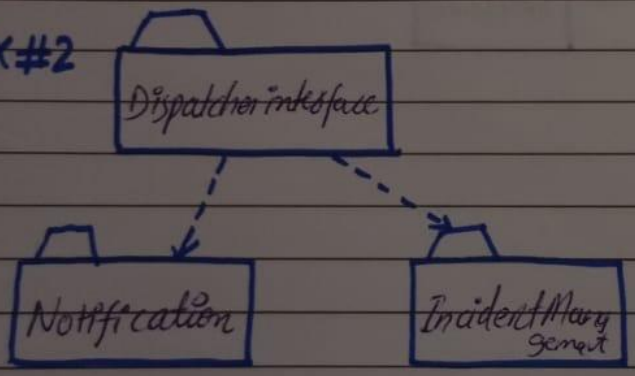
Date:

Example

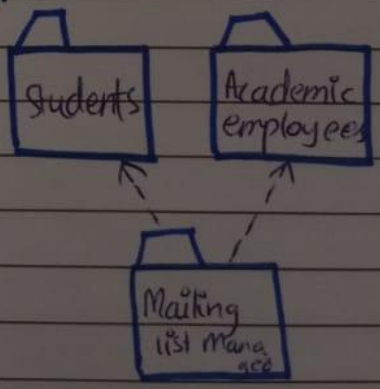



→ orders depends upon users
→ orders depends upon special orders.

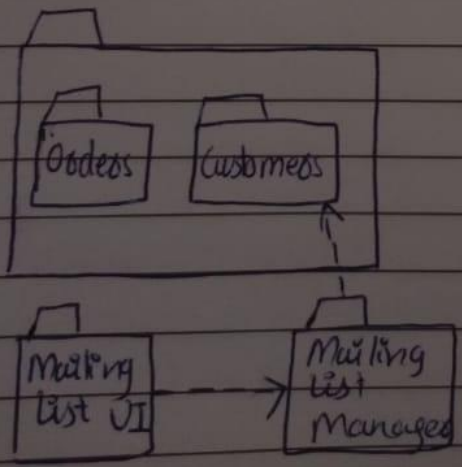
ex#2

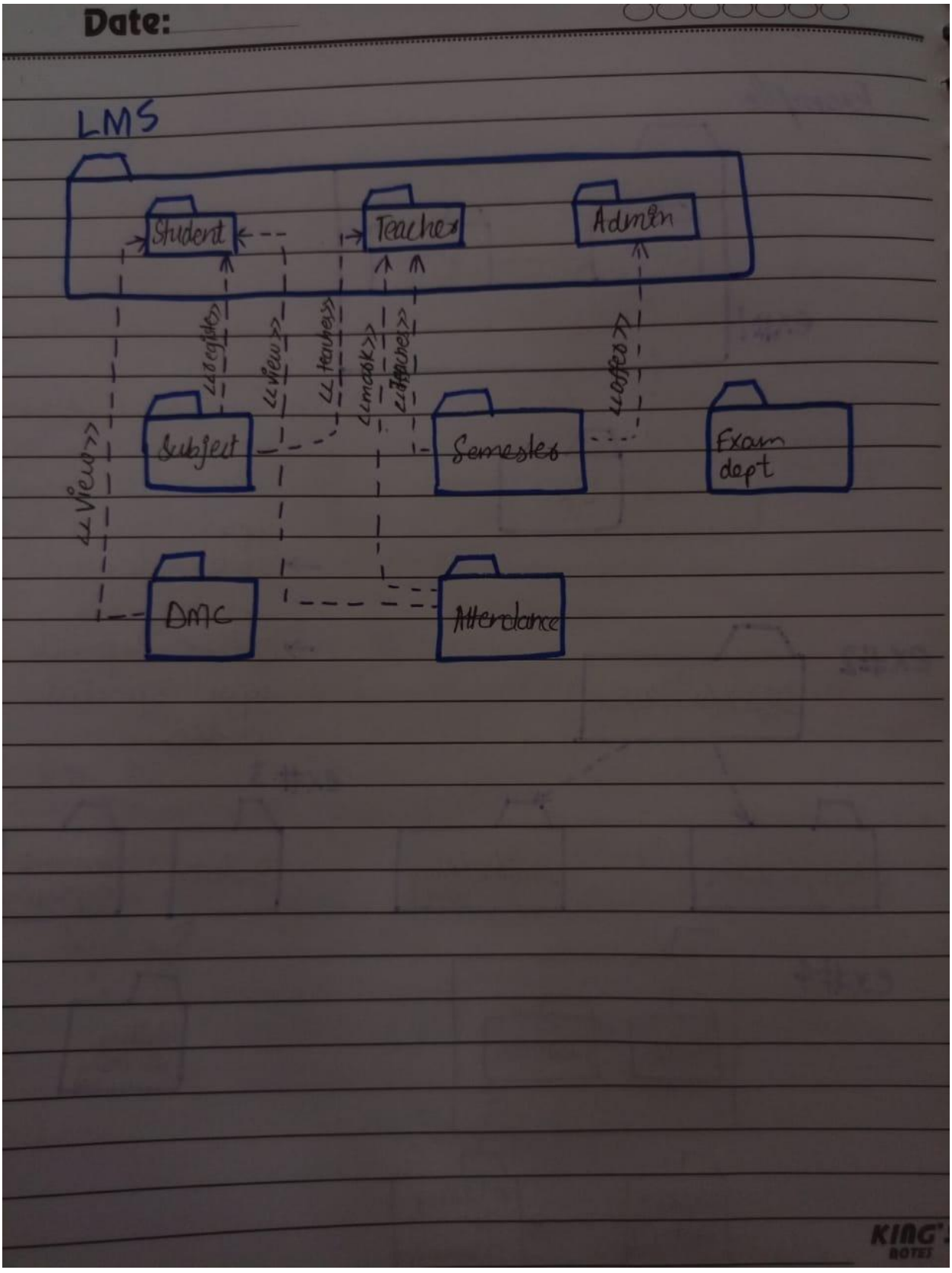



ex#3




ex#4




Date:
LMS
Student
Teacher
Admin
<<login>>
<<view>>
<<teaches>>
<<mark>>
<<assign>>
<<offers>>
<<view>>
Subject
Semester
Exam dept
DMC
Attendance
KING'S NOTES

S

# Deployment Diagram

## Deployment Diagram

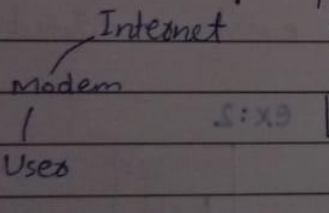
Deployment Diagram is a type of diagram that specifies the physical hardware on which the software system will execute.

It determines:

How software is deployed on hardware

maps (hardware) software pieces of a system to device that are going to execute it.

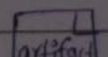
maps software architecture created in design to the physical system architecture.




These are shown by icons in deployment diagram

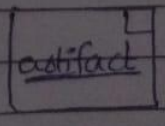
### Important Elements:

Artifact: Any real world entity.




(source file, exe file, DB file, output file, dll file)

Artifact instance: object of artifact name is underlined.

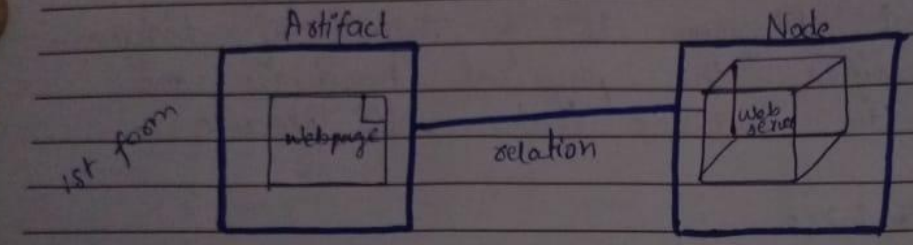



KING'S NOTES

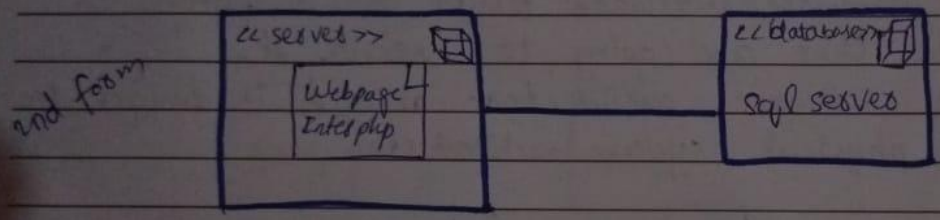
Date: M T W T F S S

Node: On which artifact execute / exists / hosts.

Two forms of deployment diagram



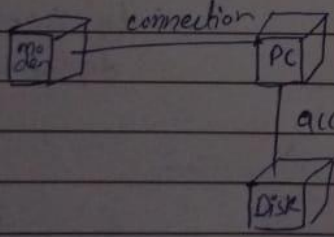

1st form



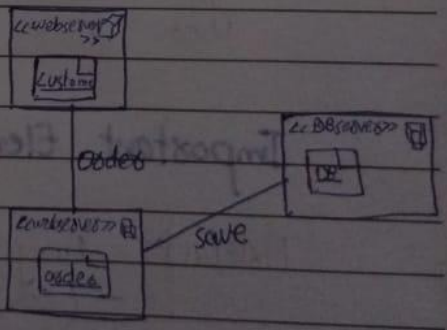

2nd form

Artifact can't exist without node

ex:1




ex:2



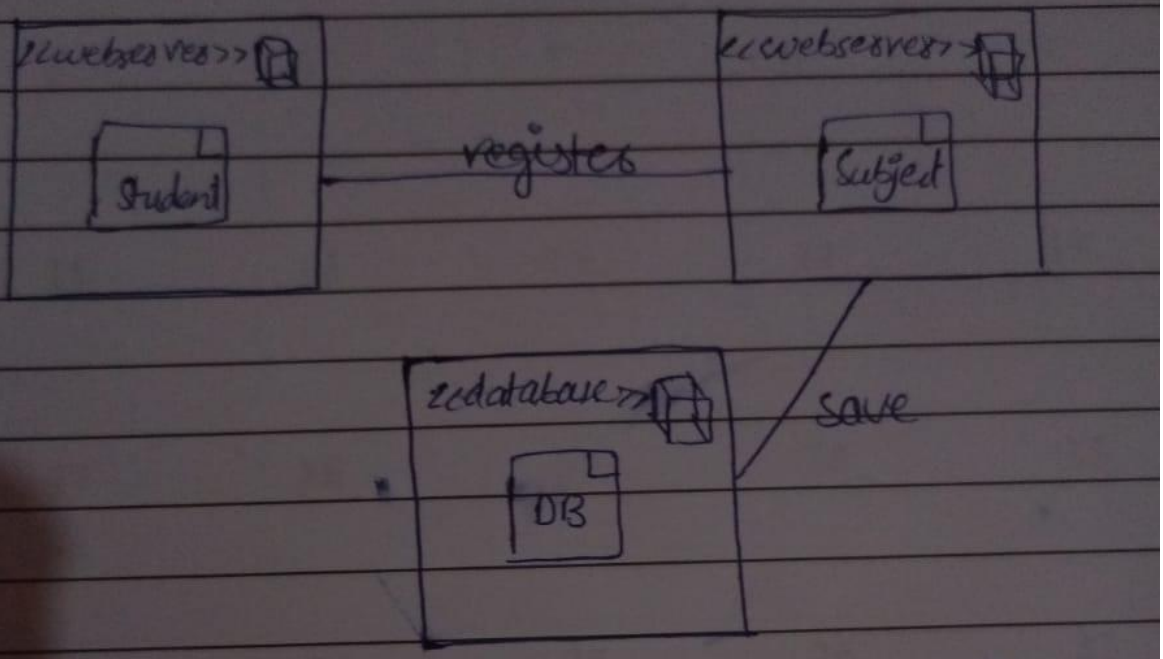

DB ← servers

Customers → modem → internet → website

KING'S NOTES

Date:

ex:3

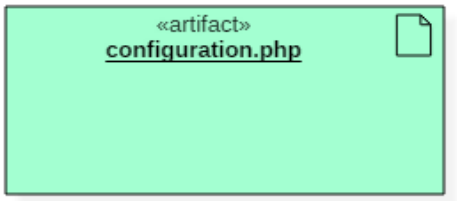

webserver
Student
registers
webserver
Subject
database
DB
save


Artifact

An artifact represents the specification of a concrete real-world entity related to software development. Artifacts are deployed on the nodes. The most common artifacts are as follows,

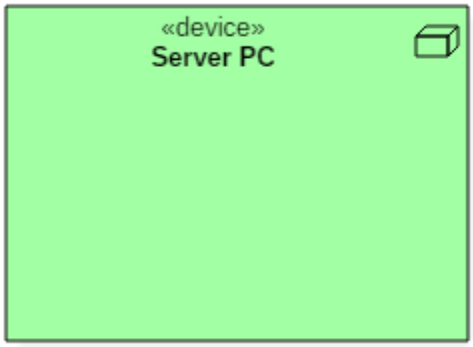
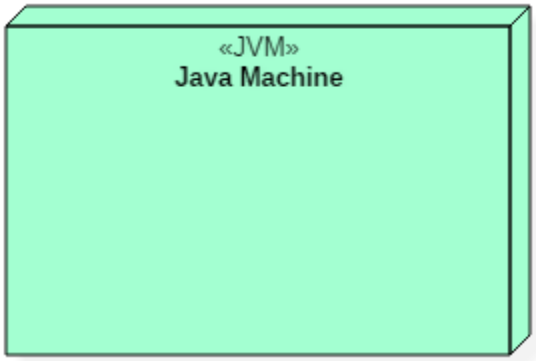
1. Source files
2. Executable files
3. Database tables
4. Scripts
5. DLL files
6. User manuals or documentation
7. Output files

Artifacts are deployed on the nodes. It can provide physical manifestation for any UML element. Generally, they manifest components. Artifacts are labeled with the stereotype <<artifact>>, and it may have an artifact icon on the top right corner.




Artifact instance

An artifact instance represents an instance of a particular artifact. An artifact instance is denoted with the same symbol as that of the artifact except that the name is underlined. UML diagram allows this to differentiate between the original artifact and the instance. Each physical copy or a file is an instance of a unique artifact.

| | |
|---|---|
|  <br>Node<br>It is a node that represents a physical machine capable of performing computations. A device can be a router or a server PC. It is represented using a node with stereotype <<device>>. |  <br>Execution Environment<br><< execution environment >><br>It is a node that represents an environment in which software is going to execute. For example, Java applications are executed in java virtual machine (JVM). JVM is considered as an execution environment for Java applications. We can nest an execution environment into a device node. You can net more than one execution environment in a single device node. |

# When to use a deployment diagram?

Deployment diagrams are mostly used by system administrators, network engineers, etc. These diagrams are used with the sole purpose of describing how software is deployed into the hardware system. It visualizes how software interacts with the hardware to execute the complete functionality.
To make the software work efficiently and at a faster rate, the hardware also must be of good quality. It must be designed efficiently to make software work properly and produce accurate results in a quick time.

Deployment diagrams can be used for,
1. Modeling the network topology of a system.
2. Modeling distributed systems and networks.
3. Forward and reverse engineering processes.

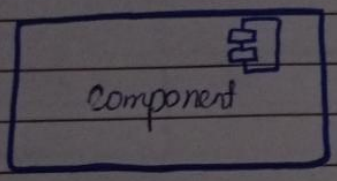
# Component Diagram

## Component Diagram

A component diagram depicts how components are wired together to form larges components or software system.

### Important Elements

**Artifact:** Any real world entity that is known as component.



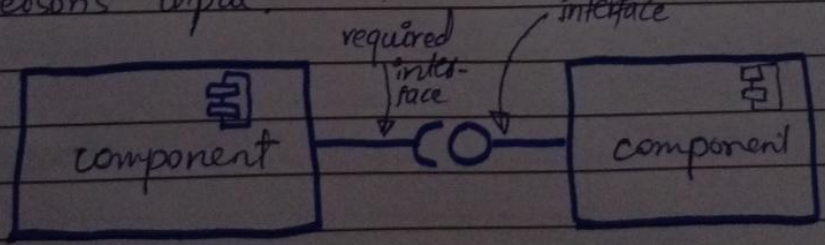

Component

**Interface:**

A interface (—○ or —⊂) describes a group of operations used (required) or created (provided) by components.

→ A full circle represents an interface created or provided by the component.

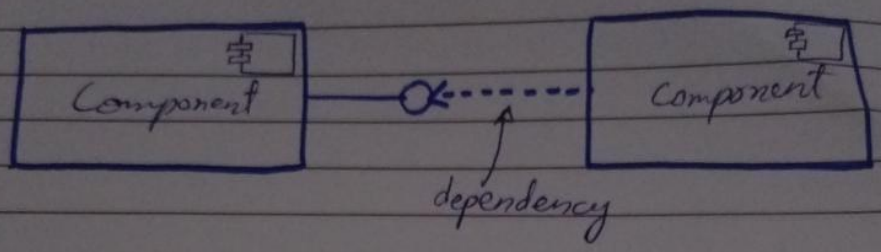
→ A semi circle represents a required interface, like a person's input.




KING'S NOTES

Date: M T W T F S S

## Dependencies

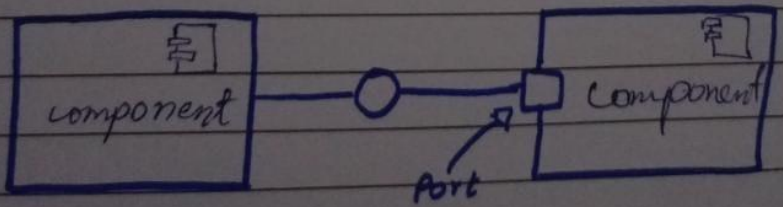



## What's difference b/w package and component diagram

Package diagram (elements) are always public while Component diagrams (elements) are private

## Port

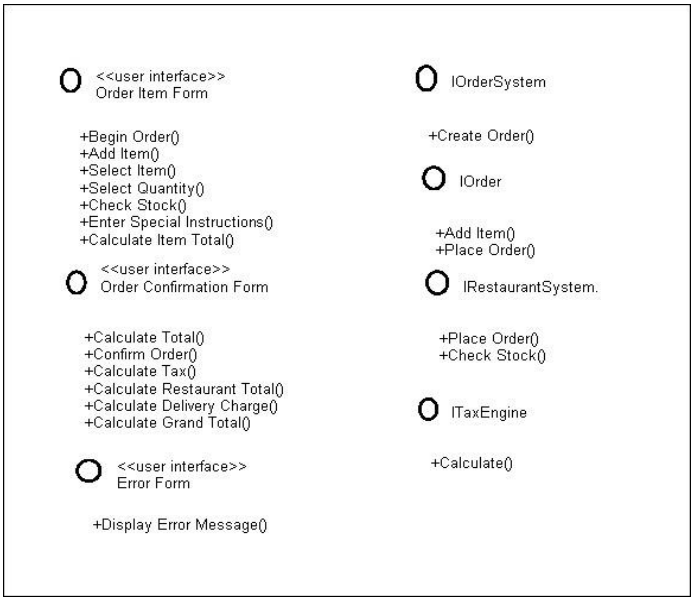
Ports are represented using a square along the edge of the system or a component. It is often used to help exposes required and provided interfaces of a component.




KING'S NOTES

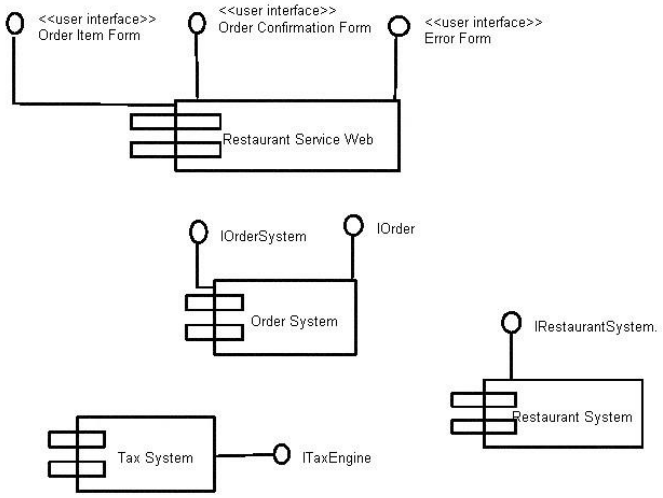
# Component Example - Interfaces

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



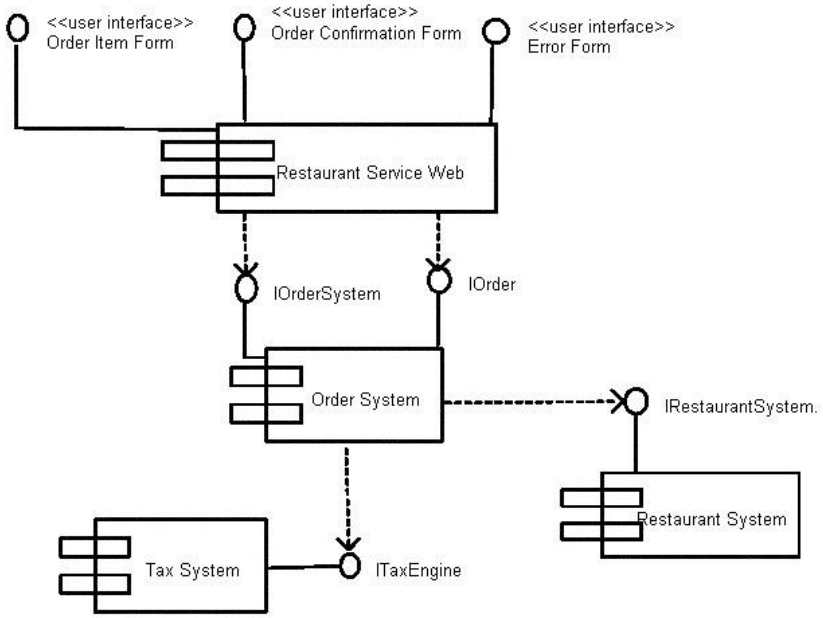

# Component Example - Components

- Graphical depiction of components

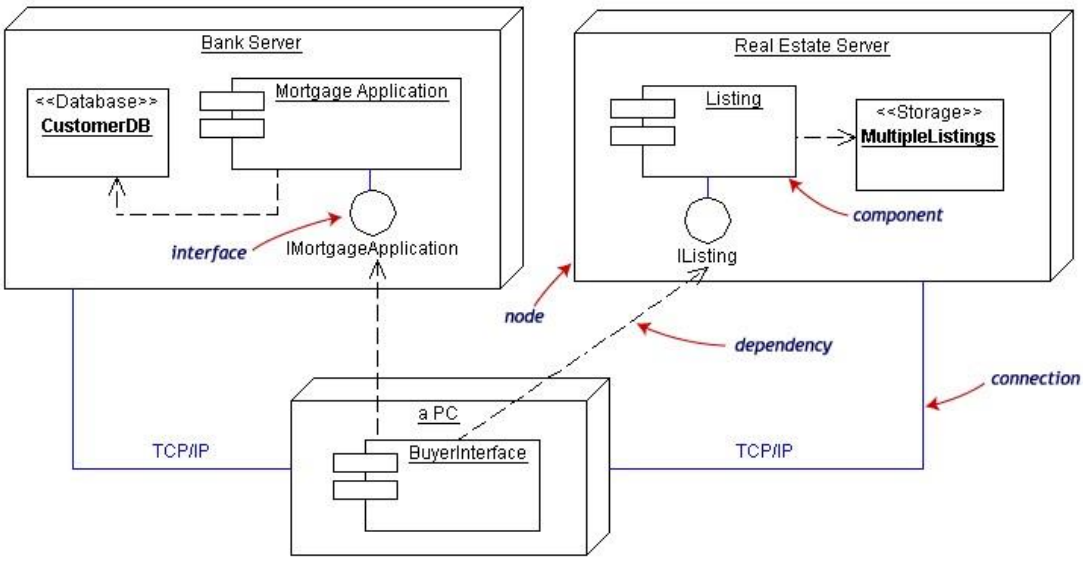


# Component Example - Linking

- Linking components with dependencies

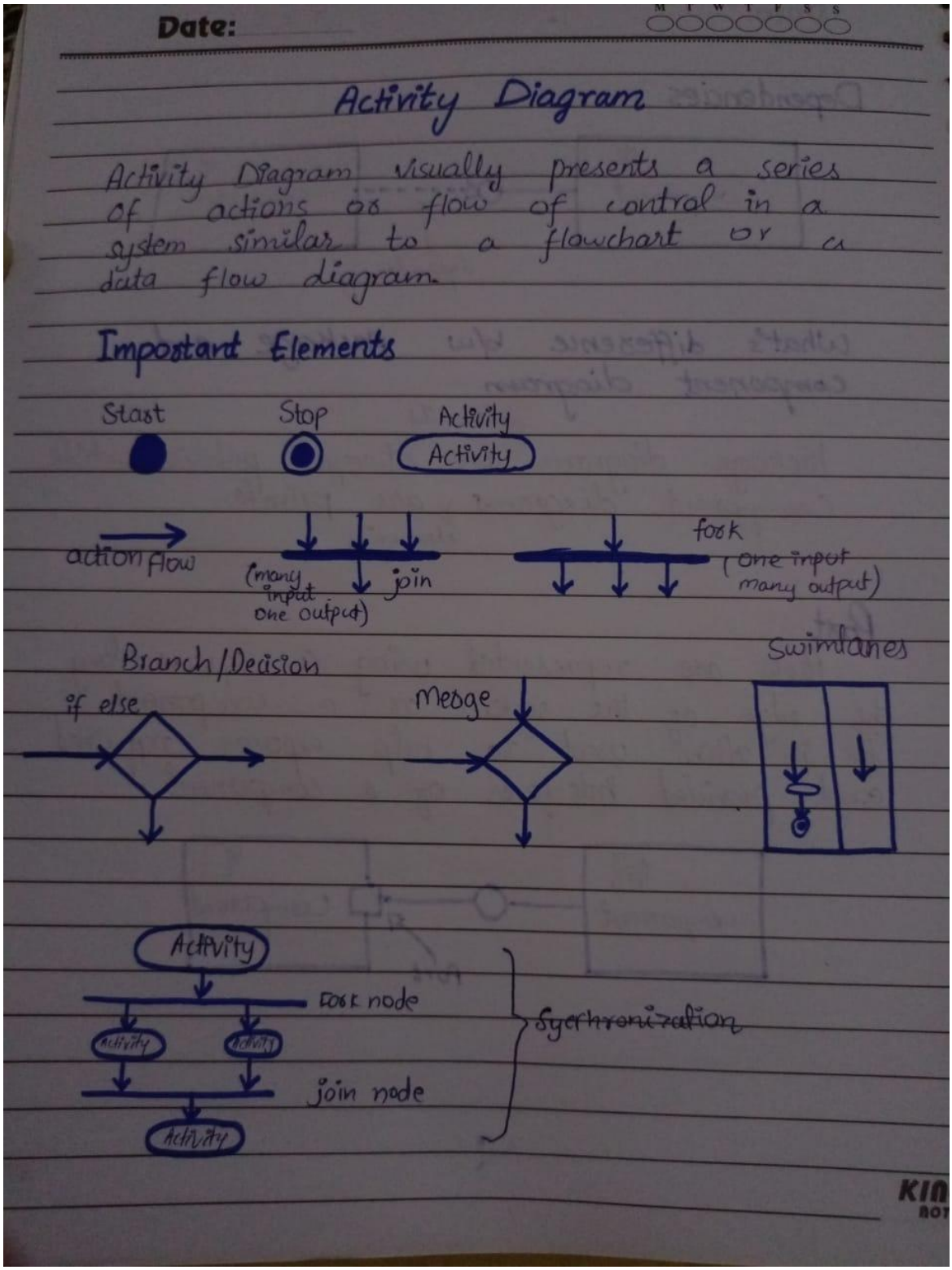


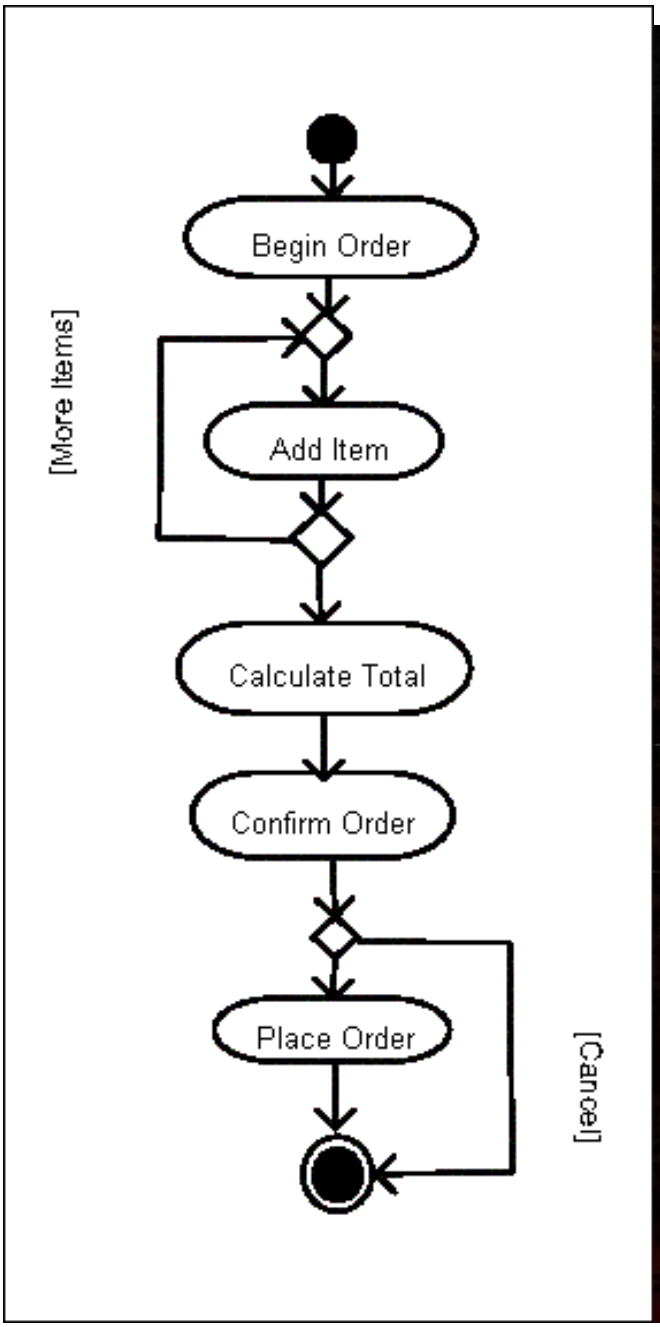
# Deployment Example

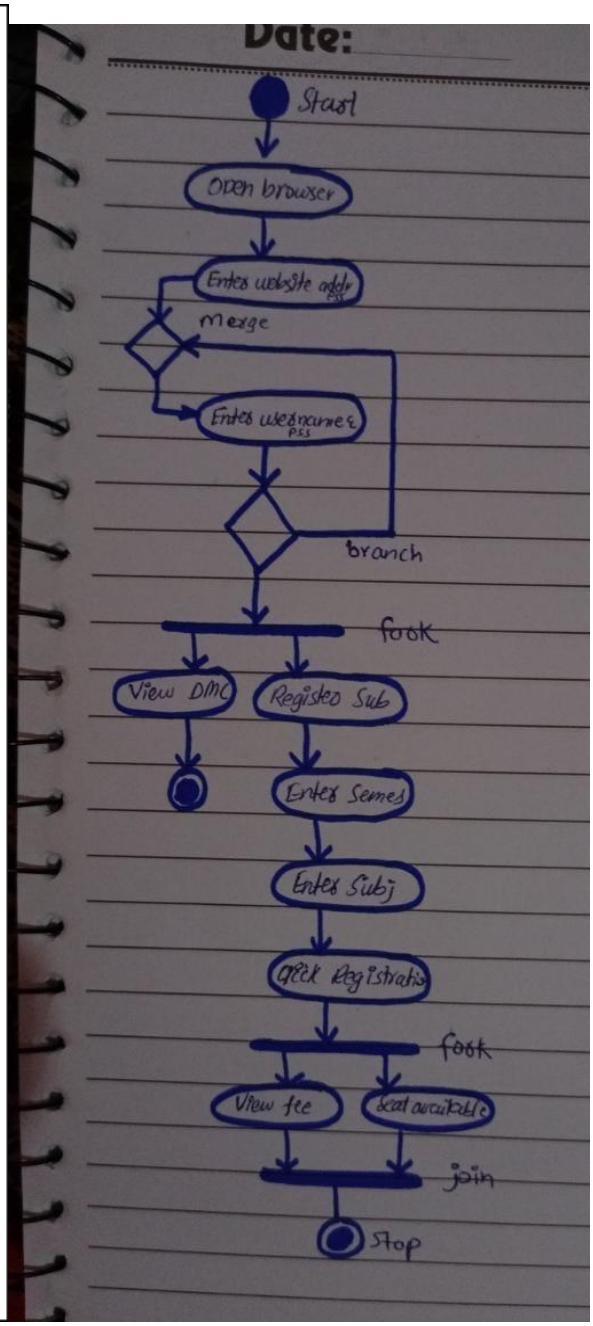


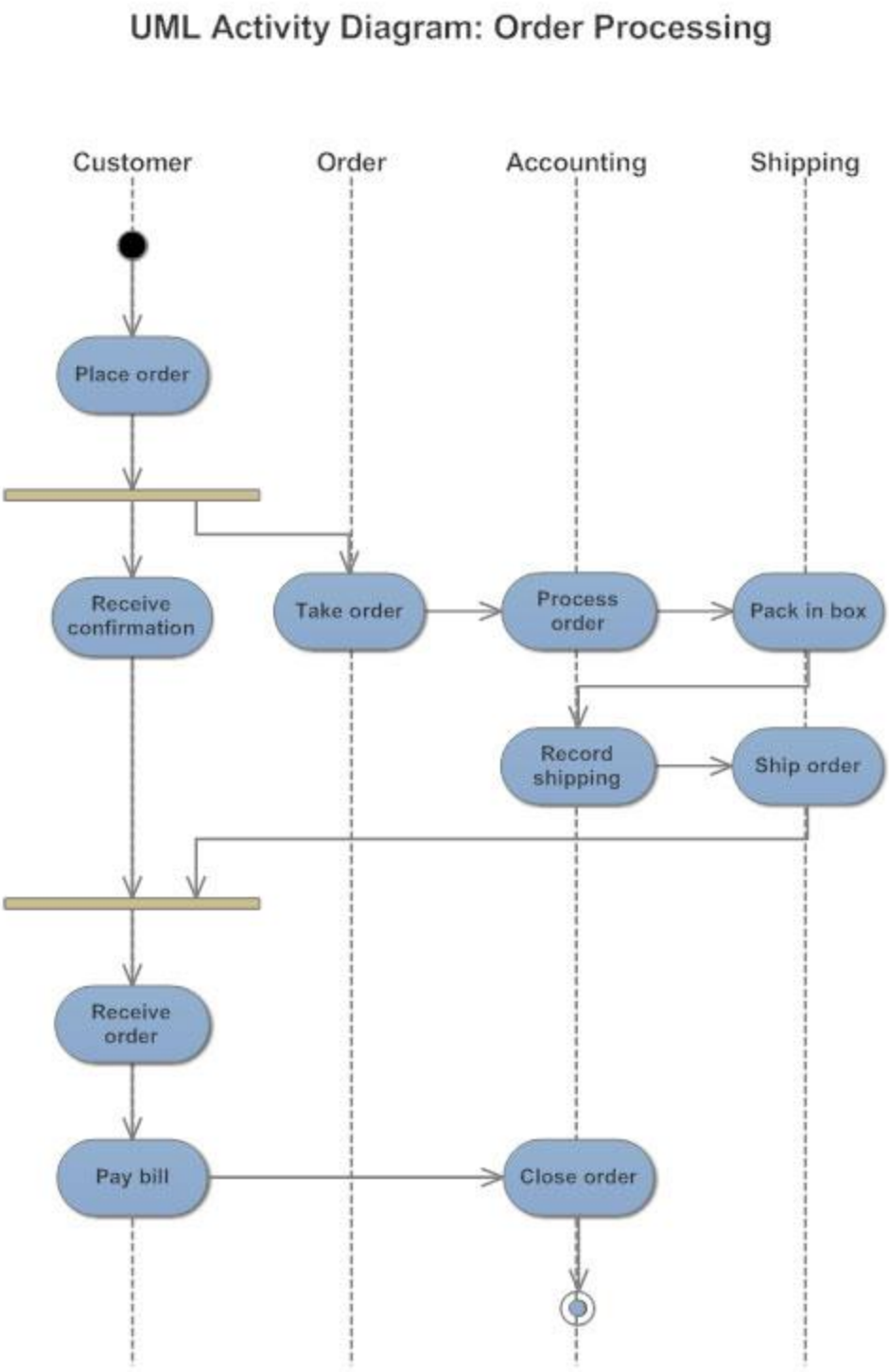

Often the Component Diagram is combined with the Deployment

# Activity Diagram

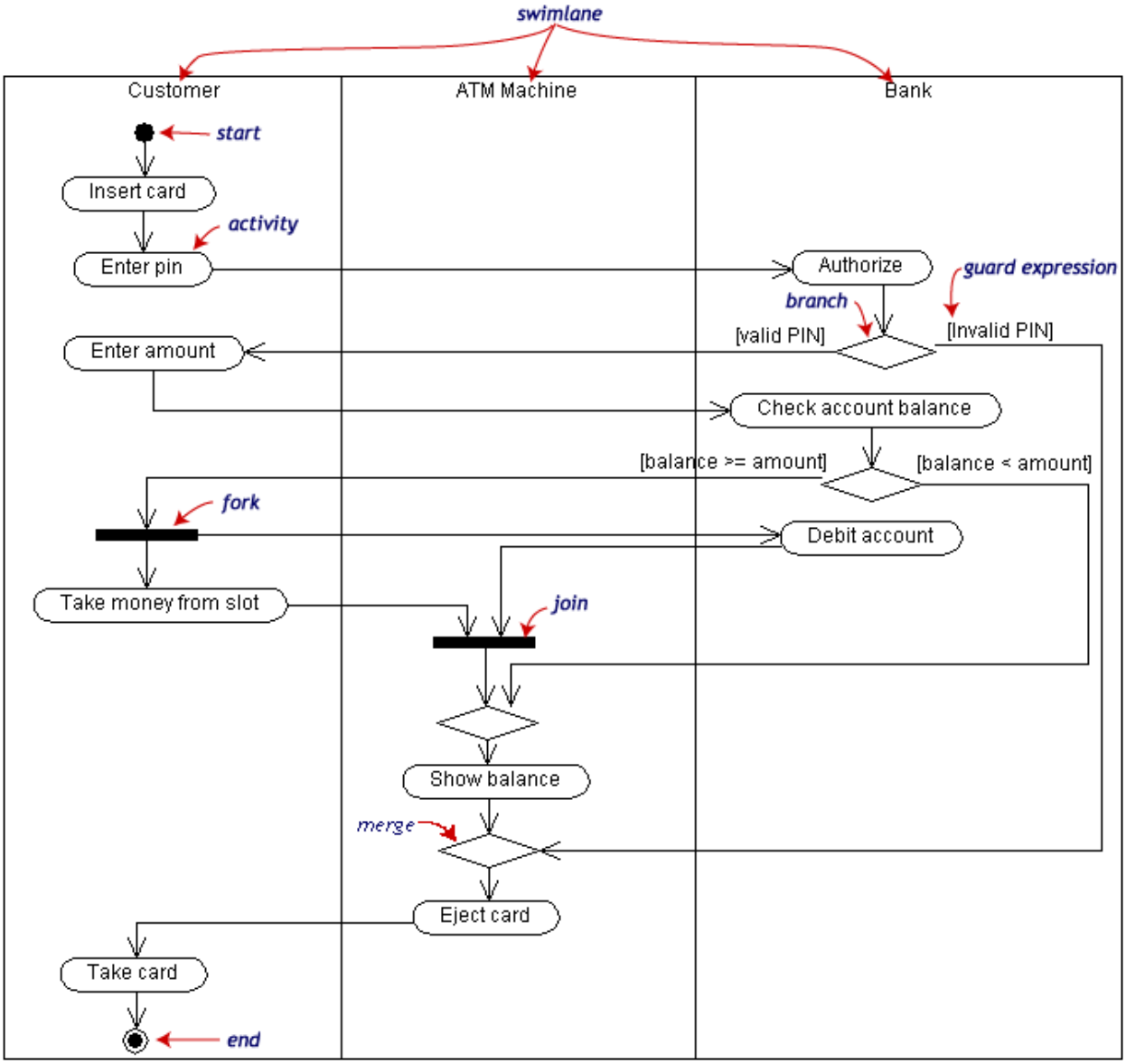



Begin Order
Add Item
Calculate Total
Confirm Order
Place Order
[More Items]
[Cancel]


Date:
Start
Open browser
Enter website addr
merge
Enter username & pss
branch
fork
View DMC
Register Sub
Enter Semes
Enter Subj
check Registratio
fork
View fee
Seat availabili
join
Stop


UML Activity Diagram: Order Processing
Customer
Order
Accounting
Shipping
Place order
Receive confirmation
Take order
Process order
Pack in box
Record shipping
Ship order
Receive order
Pay bill
Close order

Swim lanes


swimlane
Customer
ATM Machine
Bank
start
Insert card
activity
Enter pin
Authorize
guard expression
branch
[valid PIN]
[Invalid PIN]
Enter amount
Check account balance
[balance >= amount]
[balance < amount]
fork
Debit account
Take money from slot
join
Show balance
merge
Eject card
Take card
end

# State Diagram

Date:

## State Diagram

State diagram mainly depicts states and transitions.
A state diagram sometimes known as a state machine is a type of behavioral diagram in UML that shows transitions between various objects.

## Important elements

Same as activity diagram elements.

## Diff b/w state and activity

Activity is action performed by **user** and in return action performed by **system** is called state.

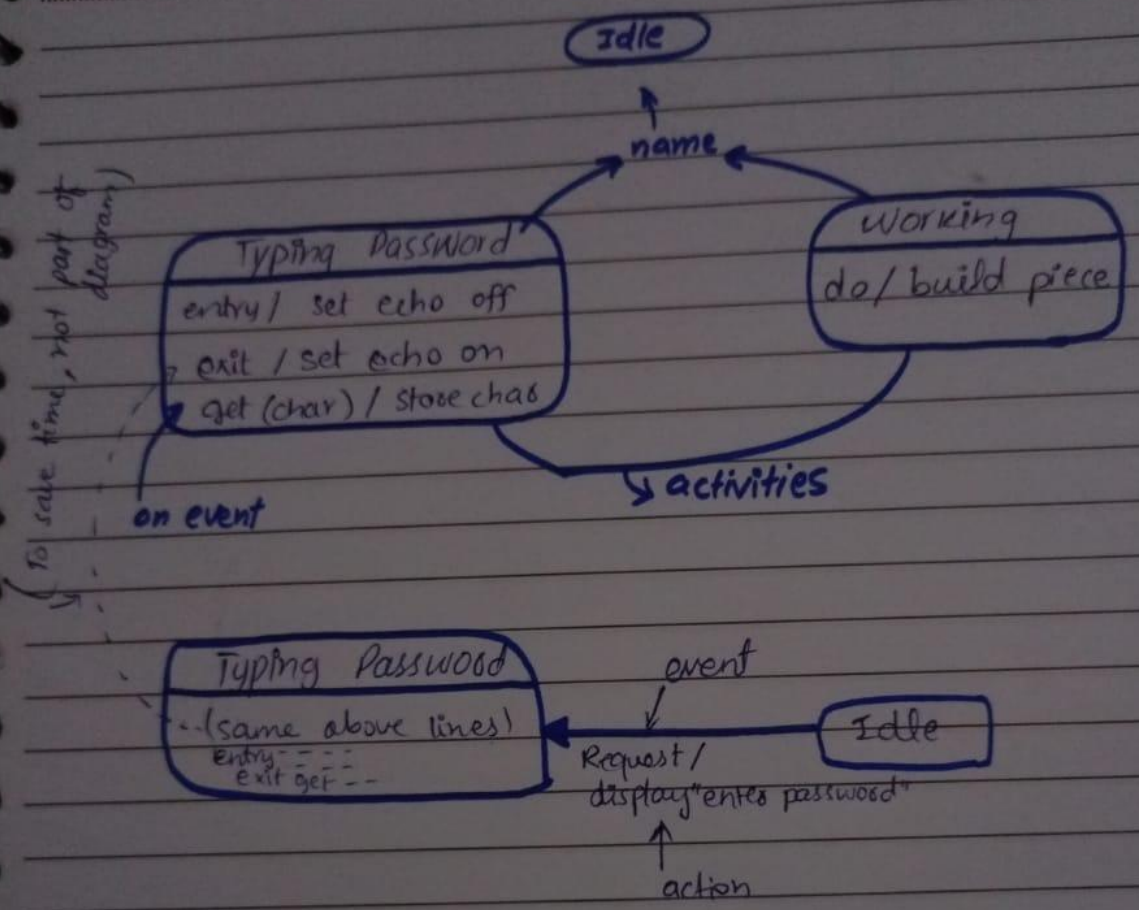
## Detailed States can be described as:

Do , Entry , Exist

| |
|---|
| Entry / login page is shown |
| Do / (Username & Pss are entered) |
| Exit / credentials are forwood to DB |

KING NOTES

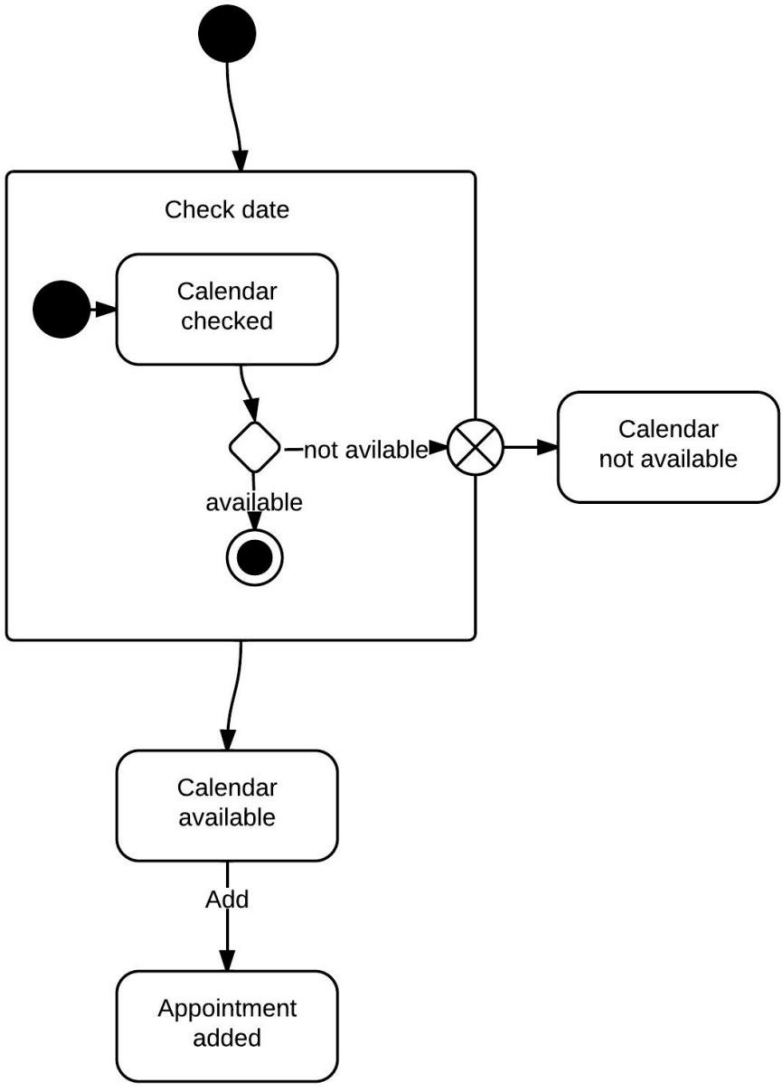
Date:


Idle
name
Typing Password
entry / set echo off
exit / set echo on
get (char) / store char
Working
do / build piece
activities
on event
(to save time, not part of diagram)
Typing Password
(same above lines)
entry - - -
exit get - -
event
Request /
display "enter password"
Idle
action

# State diagram examples

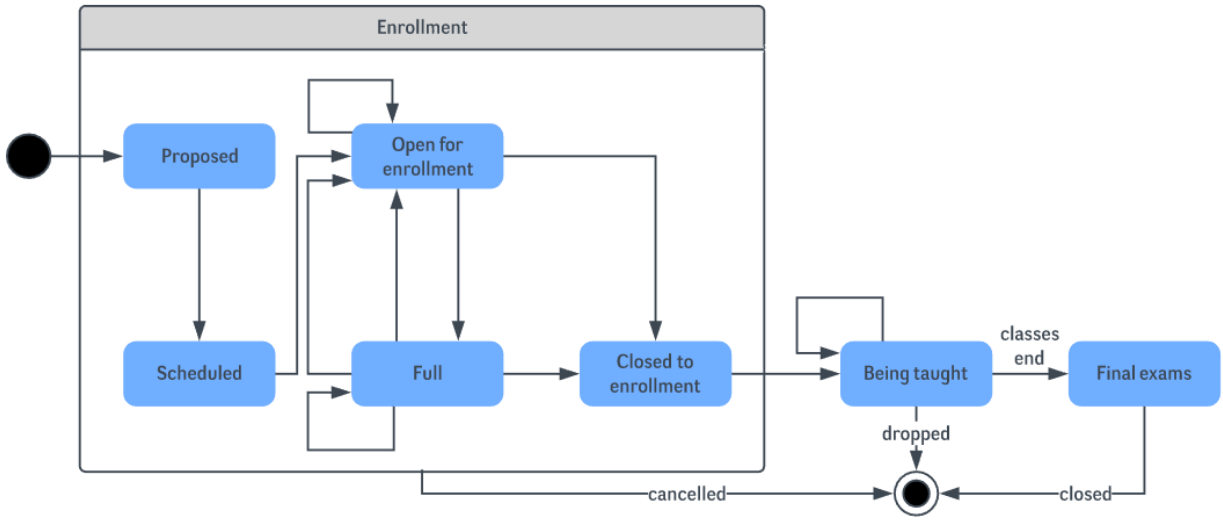
### Calendar availability state diagram example

This state machine diagram example shows the process by which a person sets an appointment on their calendar. In the “Check date” composite state, the system checks the calendar for availability in a few different substates. If the time is not available on the calendar, the process will be escaped. If the calendar shows availability, however, the appointment will be added to the calendar.




**University state diagram example**

This state diagram shows the process of enrollment and classes at a university. The composite state “Enrollment” is made up of various substates that will lead students through the enrollment process. Once the student has enrolled, they will proceed to “Being taught” and finally to “Final exams.”




**Airport check-in state diagram example**

The following example simplifies the steps required to check in at an airport. For airlines, a state diagram can help to streamline processes and eliminate unnecessary steps.