Date: __/__/20__

M T W T F S

Array maped on FIFO becomes queue. i.e. printer.

Final 2nd Semester.

St (20/05/22)

UML Diagram:

Final

30 → UML

10 → Testing.

Visual Lanuage:

(Non Programming langrage) 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 programmers and some non-programmers.). A programmer presents a code of particular language before non programmer that how this system will perform, work and flow of system, due to it, non-programmer will take time to understand and work will be delay might be possible cross the deadline. Time is very important element

• OMG → Object Management Group.
• UML is a Modelling language to express and design documents software.

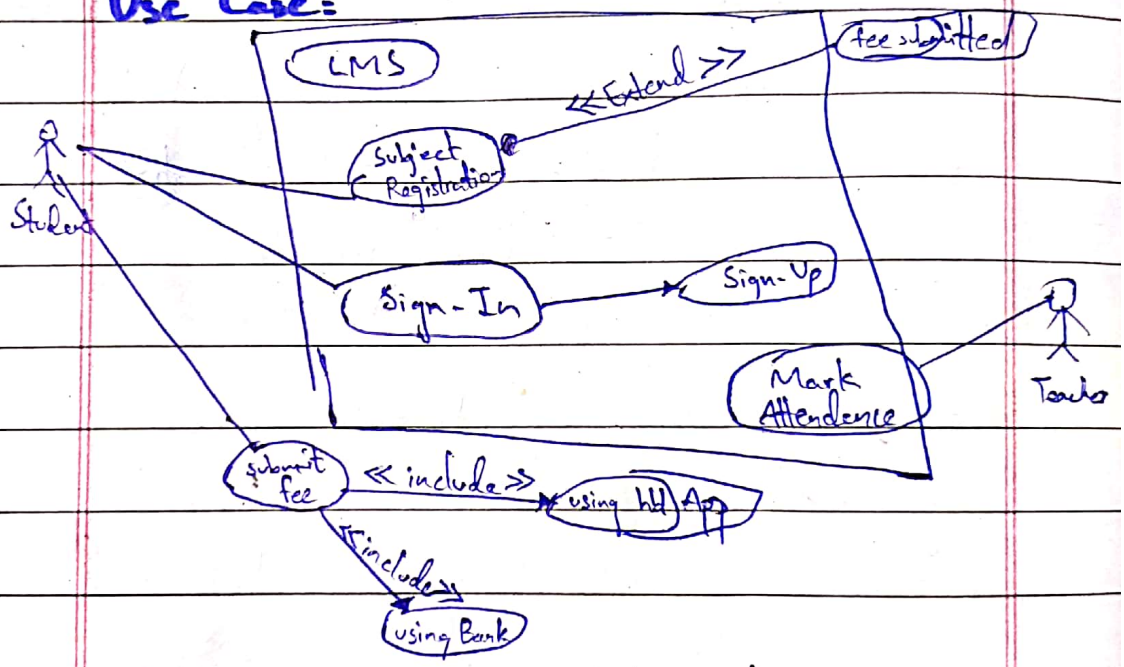
Date: / /20 M T W T F S

in designing system. UML save you.

- **Who developed it?**

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

Use Case:



<<Extend>> → When you describe exceptions in the use case.

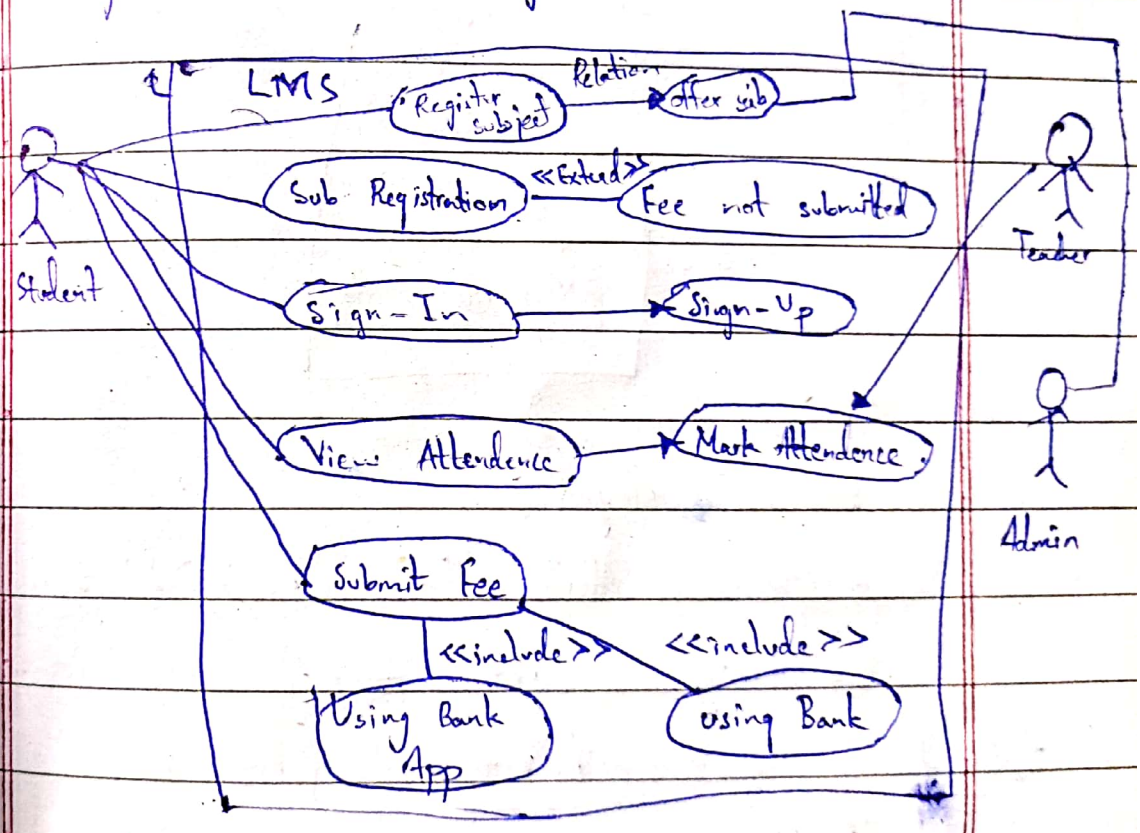
- Actor
- Use case
- Relationship

UML Diagrams:

- **Static:** A view in which system state do not change.
  (1 diagram will represent whole system).
  ↳ When message is not running

- **Dynamic View:**
  ↳ When message is running. (It shows communication with objects.

- In static view, 1 diagram will represent whole system.



Use cases are useful to:

1. Determining Requirements.
2. Communicating with clients.
3. Generating test cases.

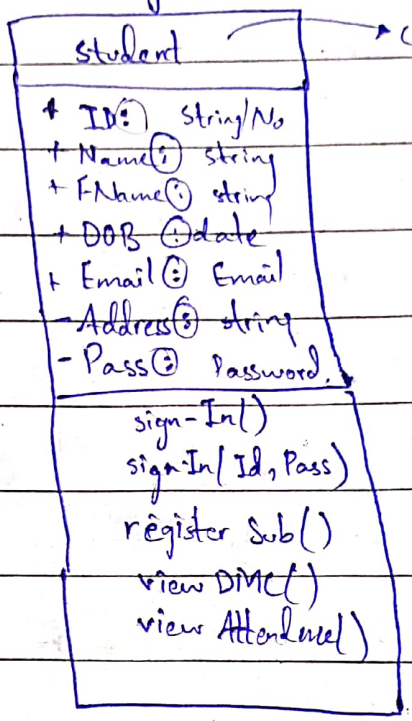
Date: __/__/20

M T W T F S

SE (23/05/22)

Class Diagram:

- It is a static view diagram.
- 4 ways to define a class in class-diagram.



1 - Name only.

2 - Name + Attribute/Parameter + Datatype.

3. Name + Operations ~~only~~.

(Complete) 4- Name + Attribute + Operations.

Public → +

Private → -

Protected → #

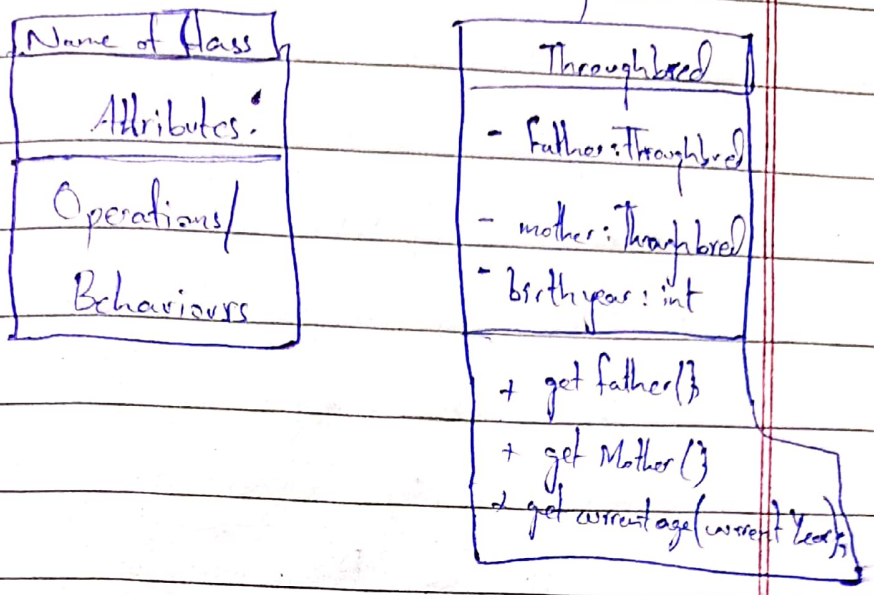
Static → ___

Date: __/__/20

M T W T F S

Main Elements: Main elements are the boxes, which are icons used to represent classes and interfaces.

Divided into 3 horizontal parts.



fig(a) presents a simple eg of throughbred class that models throughbred (जिउ) horses. It has 3 attributes displayed - mother, father & 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).

Date: __/__/20

M T W T F S

/*

Use Case Diagram Summary:

- Use case diagram represent external behaviours.
- Use case diagram is useful as an index into the use cases.
- Use case diagram provides.
  Meet of Model.
  not the use case diagrams.
- All the use cases needs to be described, for the model to be useful.

Use Case Diagram.

- Actor
  - Name
  - Description
- use case.
  - Name.
  - Participating Actor.
  - Entry Condition.
  - Exit Condition.
  - Event flow.
  - Special Requirements / Exceptional Cases.
- use case model
  - Set of all use cases

Relationship (connection) ⟷
- << extend >> Relationship
- << include >> Relationship.

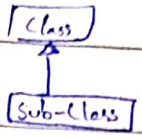
*/

Date: __/__/20

M T W T F S

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:

——— (simple line) ⇒ 2-way Associated (Undirected).

or

——→ ⇒ 1-way Associated (Directed).

←——

Represented by: Solid line

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 کا Access نہیں کر سکتی۔

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

Date: __/__/20

M T W T F S

→ Both can access each other.

Composition

(Part/whole)

A composition is an aggregation indicating strong ownership of the parts.

↳ Parts live and die with owner because they have no role in the software system independently of the owner.

Aggregation

(part/whole)

Aggregation is special kind of association, presented by a hollow diamond).

Abstract Class vs Interface

Abstract Class : A class which is used to implement certain or common properties in more than one class.

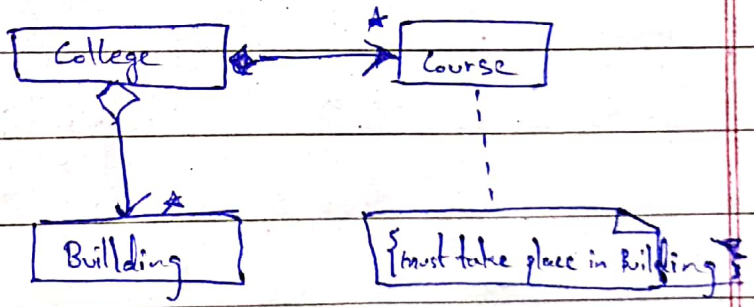
Syntax: Abstract Class Name { };

- All student inherit/use the Abstract class.
  - ↳ Protected + Private
  (Not define the body of each class).
- Interface is the complete Abstraction.
  - ↳ Only Public.
- Abstract Class name will be written in Italic.

Date: __/__/20 M T W T F S

Note:

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 (& 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.



Multiplicity:

The multiplicity at one end of an association means the number of objects of that class associated with the other class.

↳ Multiplicity is specified by a non-negative integer

Date: __/__/20

M T W T F S

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 Social Security Number 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 Owned Object end of the association with class Person in figure, because a Person can own zero or more objects.

Date: __/__/20

M T W T F S

Use Case Diagram:

Use cases (Chapter 8 and 9) and the UML use-case Diagrams 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 diagram 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 applications 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.

Date: __/__/20 M T W T F S

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

A use case describes how a user interacts with the system by defining the steps required to accomplish a specified 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 the 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.

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 fill the machine.

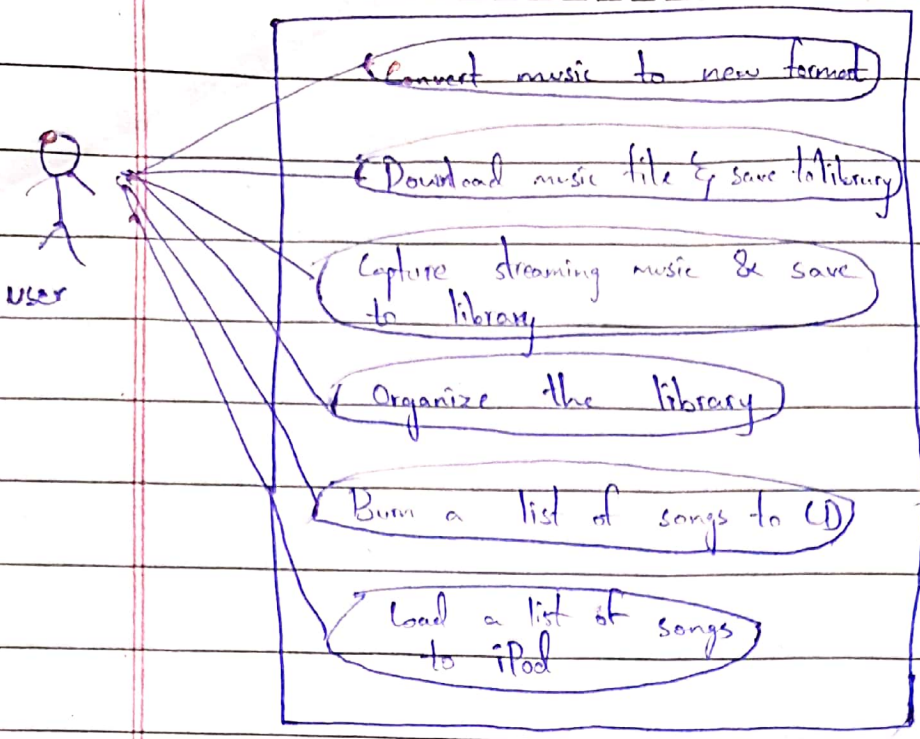
Date: __/__/20

M T W T F S

In the use case diagram, the use case 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 the rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and 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 use cases include this new use case as one of their steps.

Date: __/__/20__ M T W T F S



Such inclusion is indicated in use-case diagrams by means of a dashed arrow labeled <<include>> connecting the use case with an included use case.

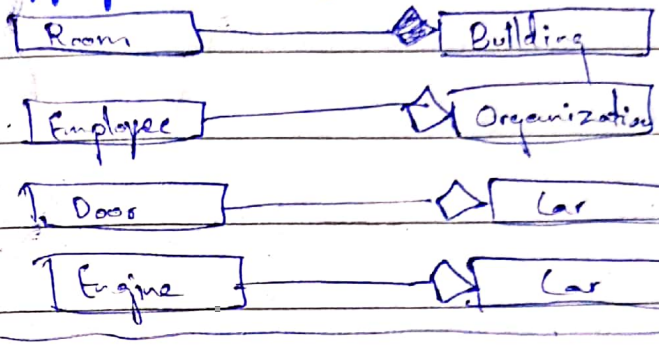
A use-case diagram, because it displays all the use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In our digital music organizer, we would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most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

Date: __/__/20

M T W T F S

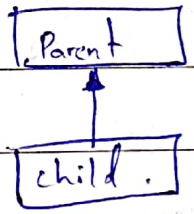
use-case diagram: It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

Composition/Aggregation Examples.



Inheritance:

↳ Child class can inherit both operations and variable.



Memory is the advantage of inheritance

- Public & protected will be inherited
- Parent class cannot access any attribute of child class but child have all access.

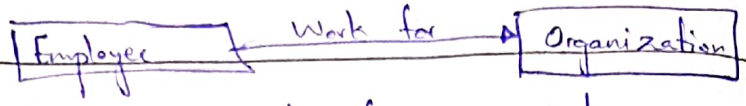
Name of Relationships:

"has" → but not written

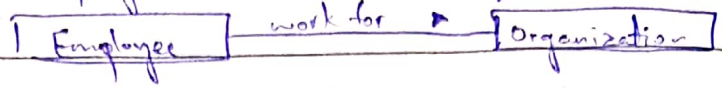
Student has Dep and Dep has Student

Date: __/__/20

M T W T F S



Employee work for organization.



Employee work for organization &
organization has employee.

Multiplicity:

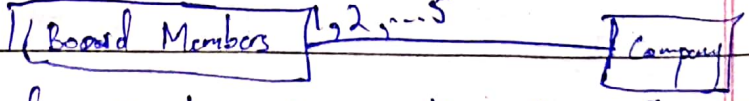
n (minimum 1)	1-n
* (minimum 0)	0-∞
1-1	
1-n	
1-*	

Range:

2 ---- 5 *-n

0 ---- 1 - 1

- Multiplicity can be defined using both numbers & range -

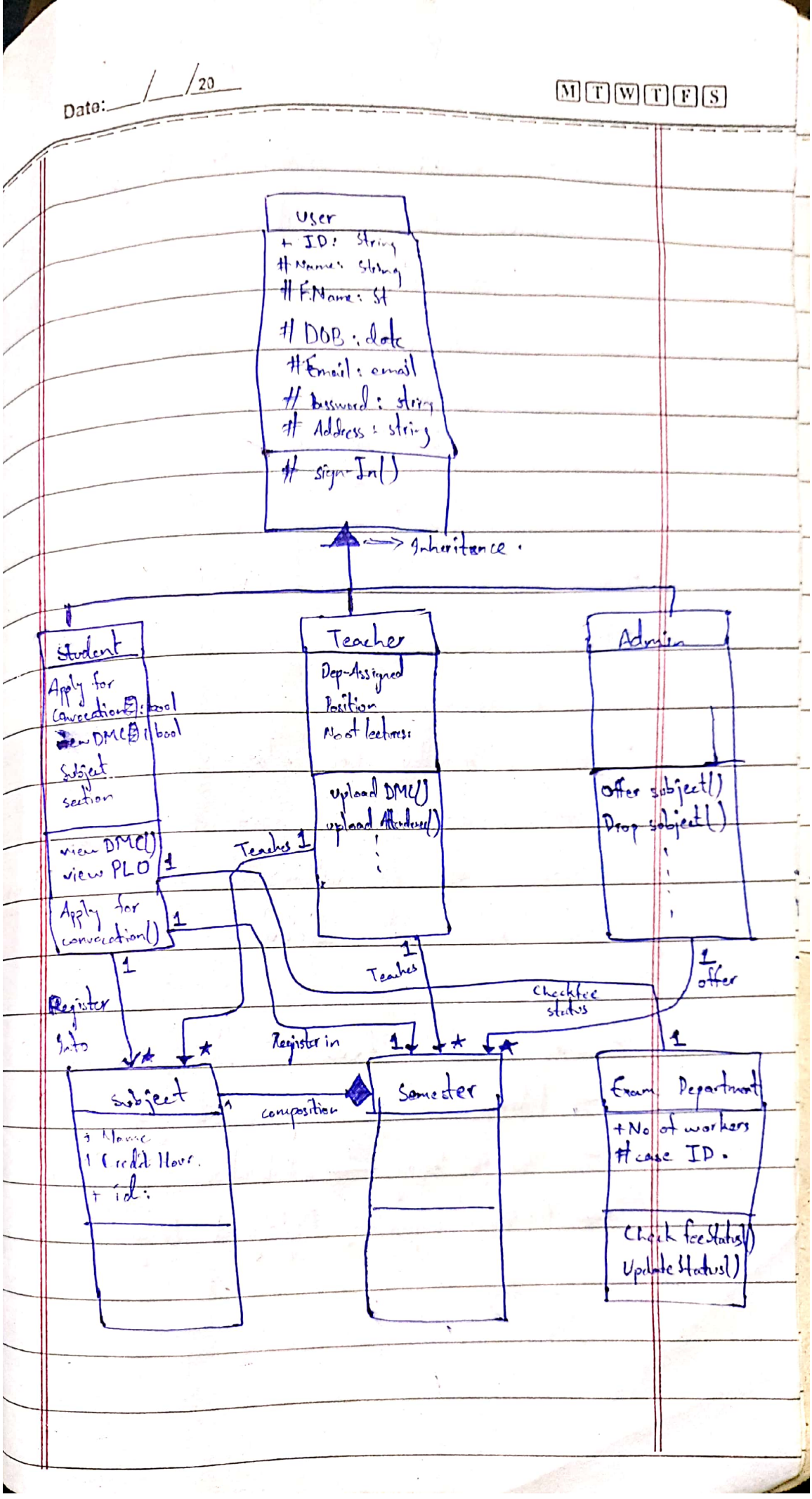


Board Members has either 1 or 2 to 5 company.

- Inheritance.
- Part/whole. → Composition (Dependent Part)
  → Aggregation (Independent Part else association)
- Association. → Directed
  → 2-way
- Multiplicity.

Date: / /20

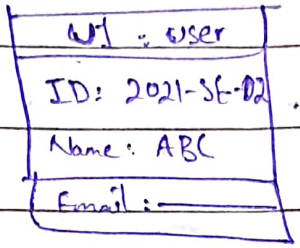
M T W T F S



Date: __/__/20__ M T W T F S

Convert Class Diagram to Object Diagram.

Object Diagram is a snapshot of executing or running program in which each attribute has some value.



- Add unique keyword with name.
- Remove data types, Access modifiers.
- Remove Operations.
- // All Relationships (All relations will convert into two way association).
- // Names of Associations & Multiplicity (Remove)
- Add value of attributes.

↳ with selected focus.

⇒ Interaction — Sequence Diagram.

⇒ Message Passing — Collaboration Diagram.

⇒ Operation — Deployment Diagram.

Date: __/__/20

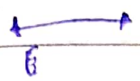
M T W T F S

SE (27/05/22)

Sequence Diagram:

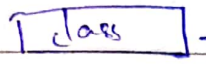
Static X / Dynamic ✓

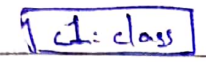
It is Dynamic Diagram (Messages are shown).

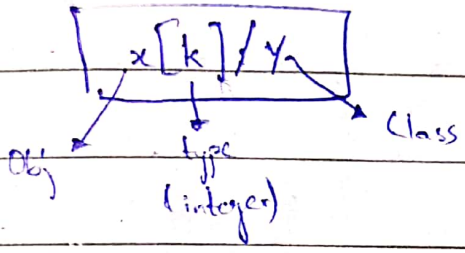


Equals to ——

but ⟷ it is wrong.







⇒ x object of class y of type k.

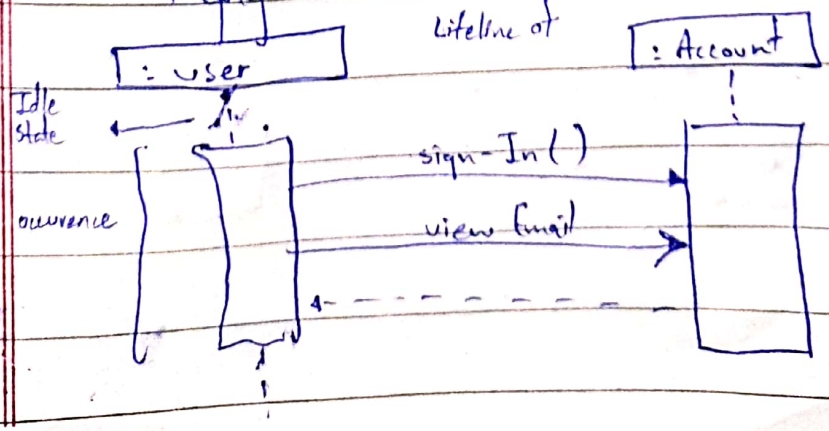
Messages:

How a class will communicate with other.

Synchronous ⟶

Asynchronous ⟶

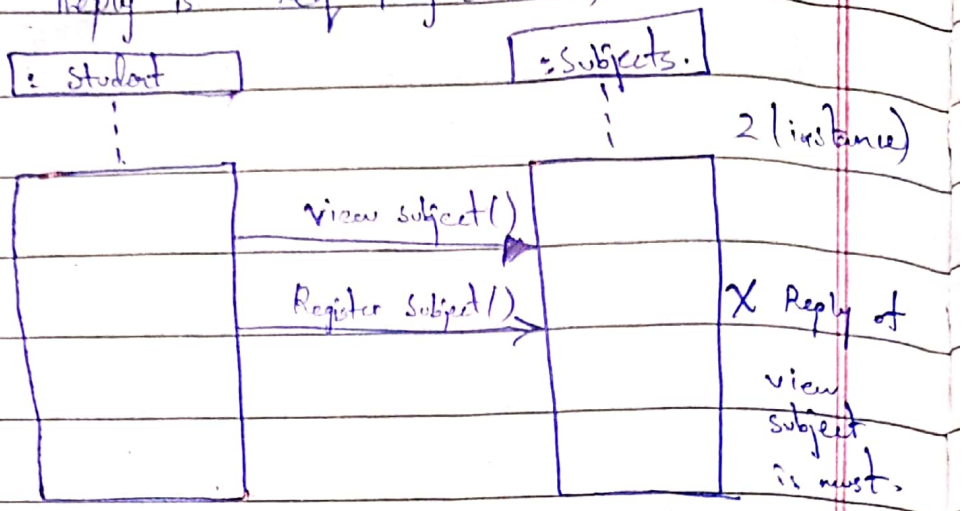
Reply ⟵ - - - -



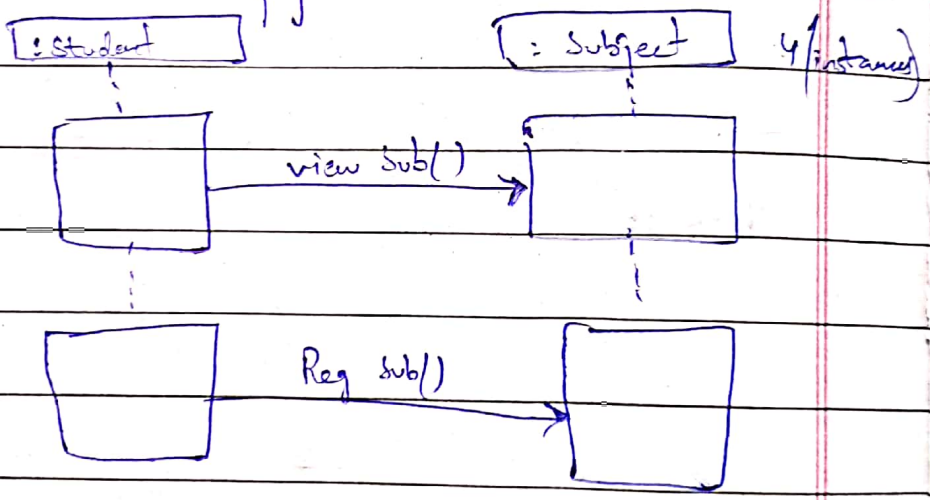
Date: __/__/20

M T W T F S

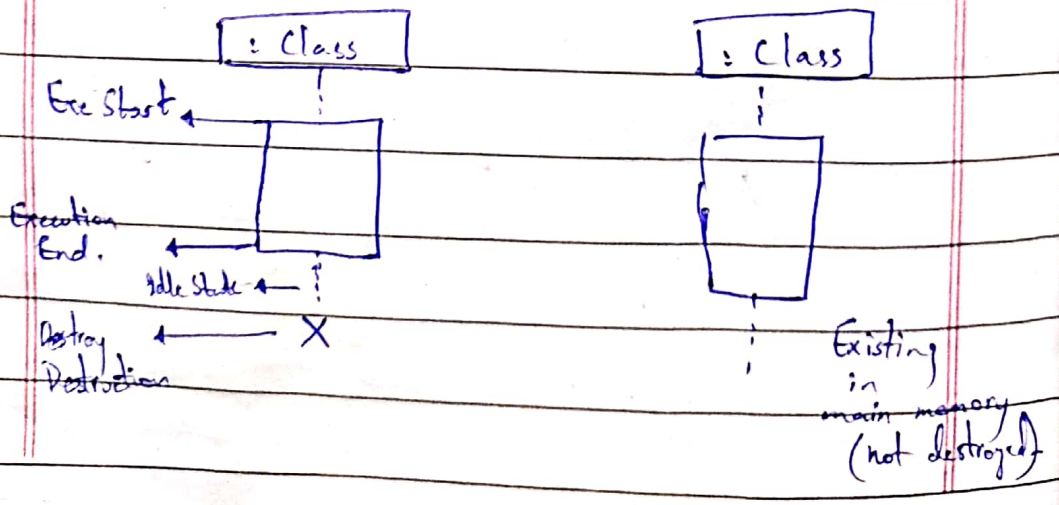
Reply is Req (Synchronous).



- Asynchronous message can or cannot have a reply but synchronous message must have a reply.

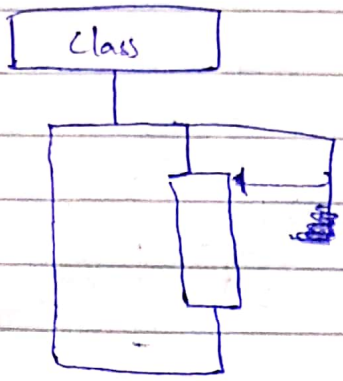


Destruction:

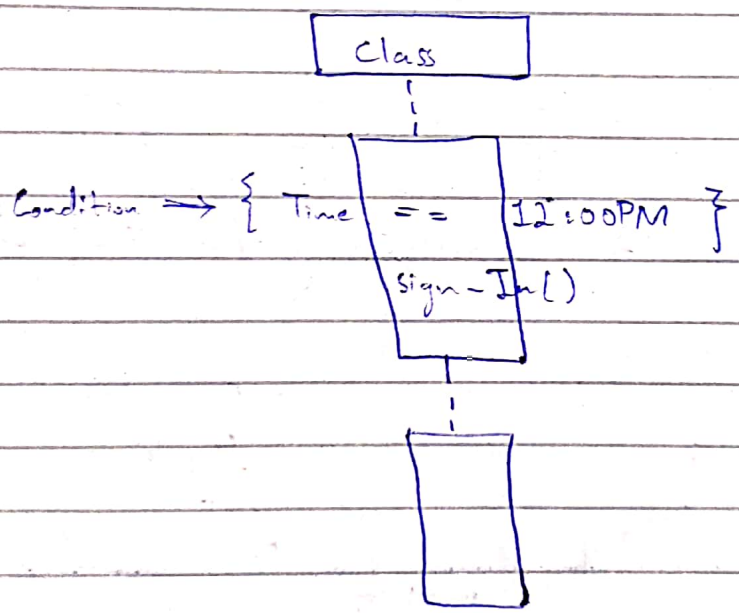


Recursion:

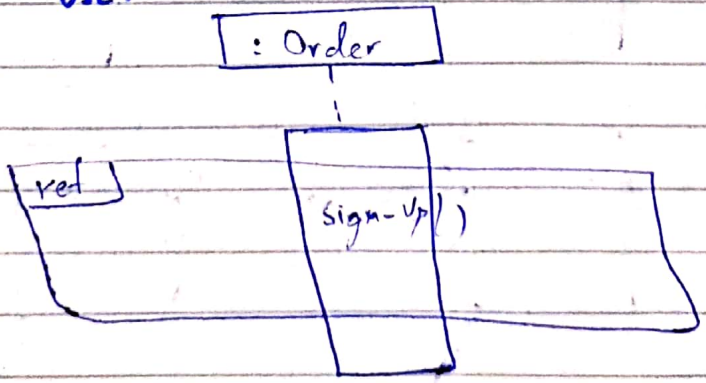
A class calls itself or a class calls another class and that class calls this one.



State Invarian (Condition).



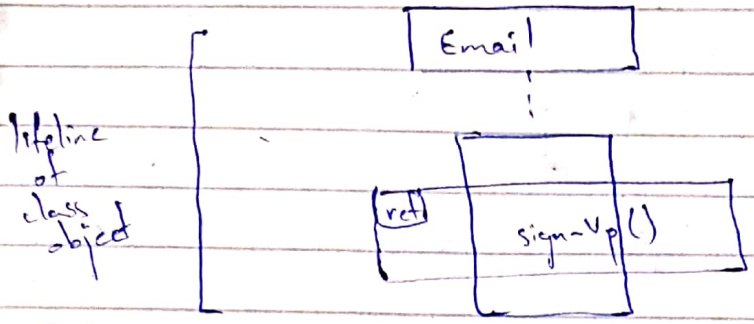
Interactio Use:



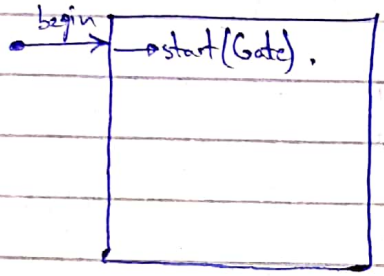
Sign-Up is referred.

⇒ You cannot order until you sign-Up to the account.

In Interactio use, we refer another diagram in sequence diagram

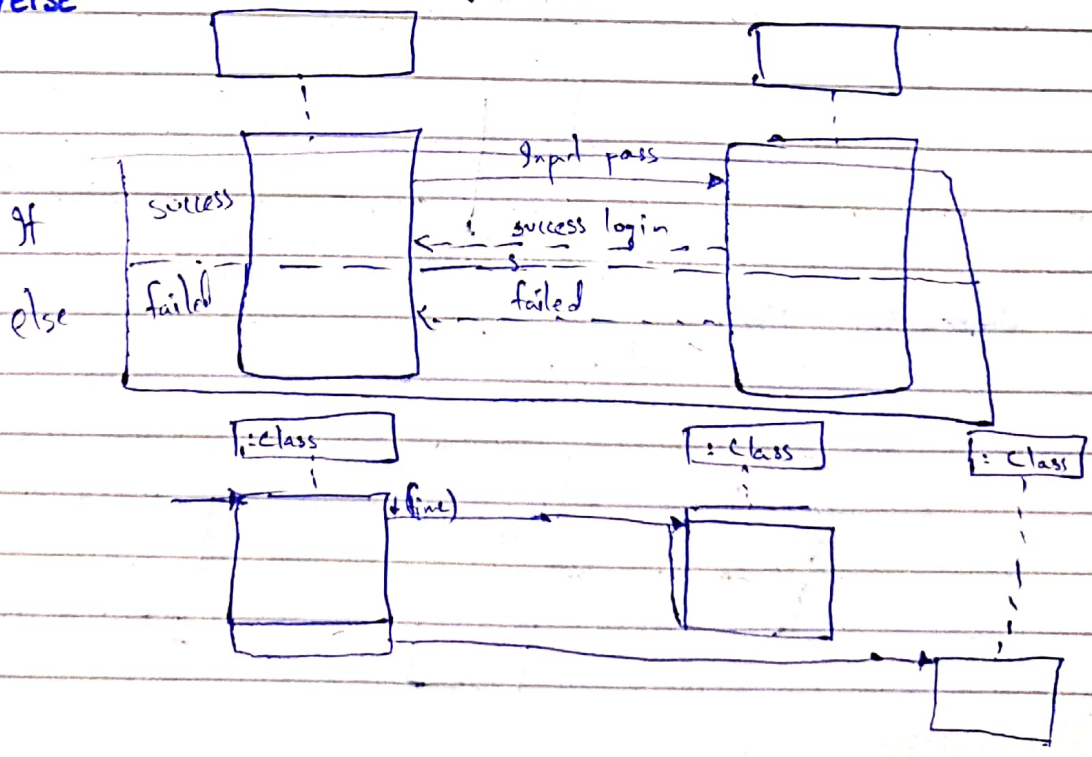


Gate:



The starting of diagram/begin/Initialization/start.

If/else



Collaboration Diagram:

↳ It is an interaction diagram.

{Sequence and collaboration | 2 interaction Diagrams}

↳ It is easy to draw after sequence diagram.

Interaction Diagram:

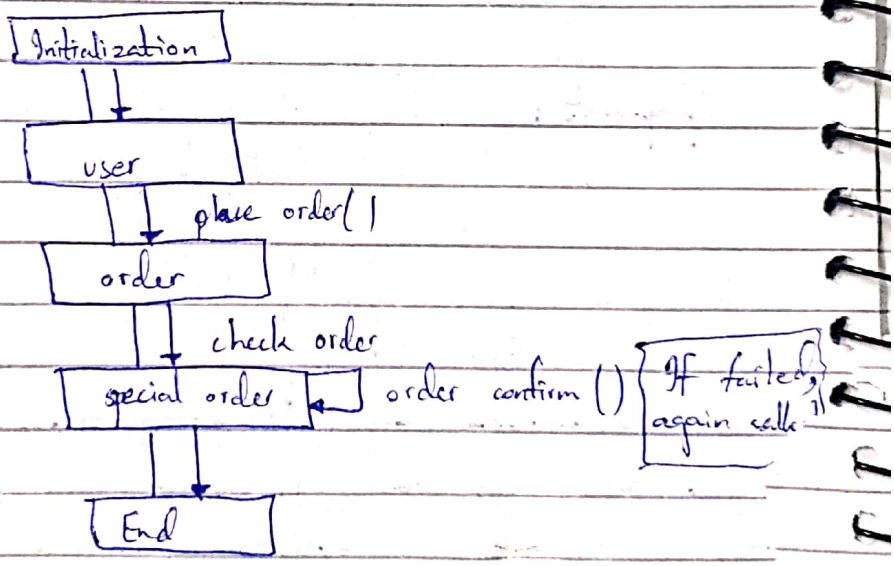
In which class/object interact with each other

1- Communication 2- Time ~~3- Structure~~.

- Collaboration Diagram is vertical, Sequence Diagram is horizontal.

- Time is important in Sequence Diagram, Structure is important in collaboration Diagram.

Structure:

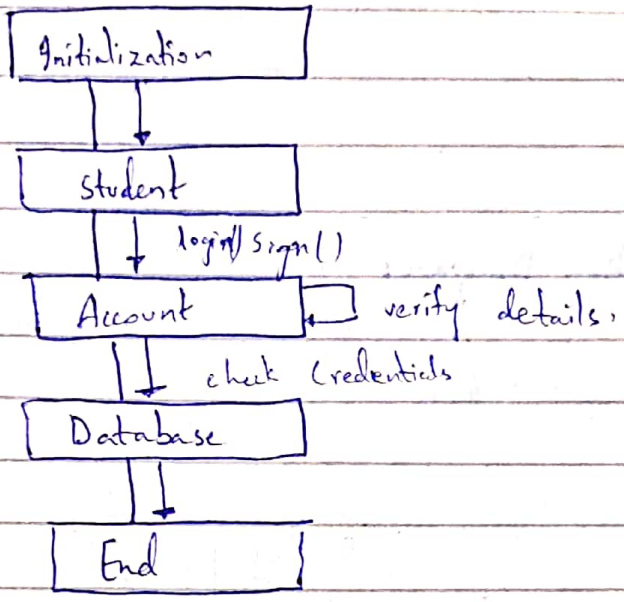


↳ No if/else.

↳ No interactio use.

↳ Imp to show structure.

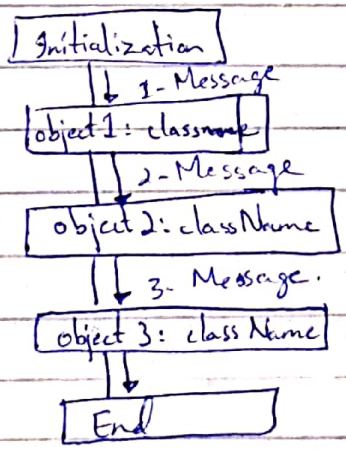
- Draw collaboration Diagram for login on LMS.



↳ Relationships and interactions among software objects.

↳ Dynamical behaviour of a particular use case.

Components :



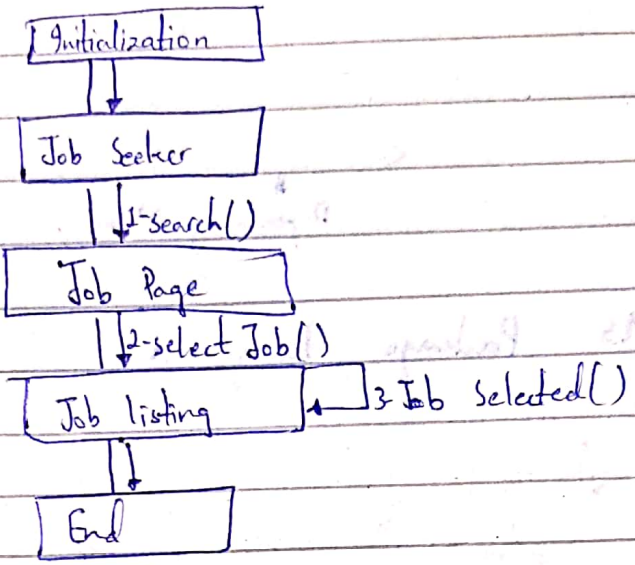
↳ Collaboration diagrams have no concept of the asynchronous messages, since its focus is not on the message ordering.

↳ Collaboration diagram has message numbering.

↳ In rectangular ~~thing~~ boxes member functions will not be written.

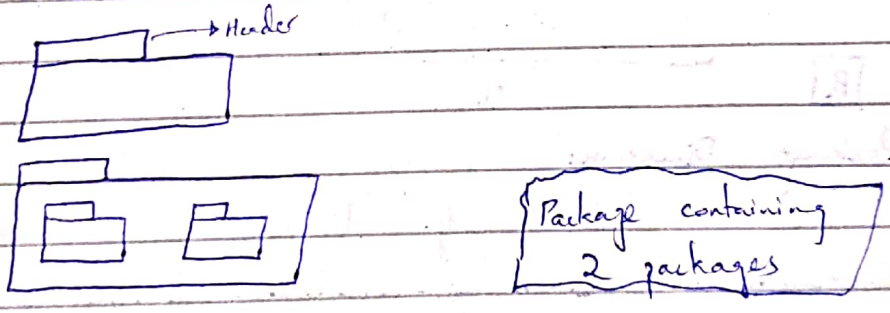
↳ There is ending and starting in collaboration diagram.

Example:



Package Diagram: (Static Diagram)

Package: Combination of related classes.



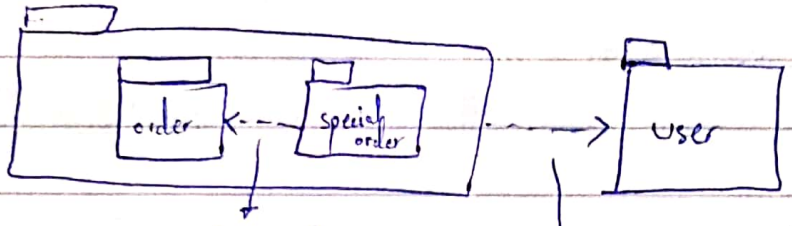
Uses:

↳ To increase readability of the program.

→ Show Dependencies in diagrams.

Dependency:
If there is change in one class there must be change in second class

→ Combined the classes with same functions.



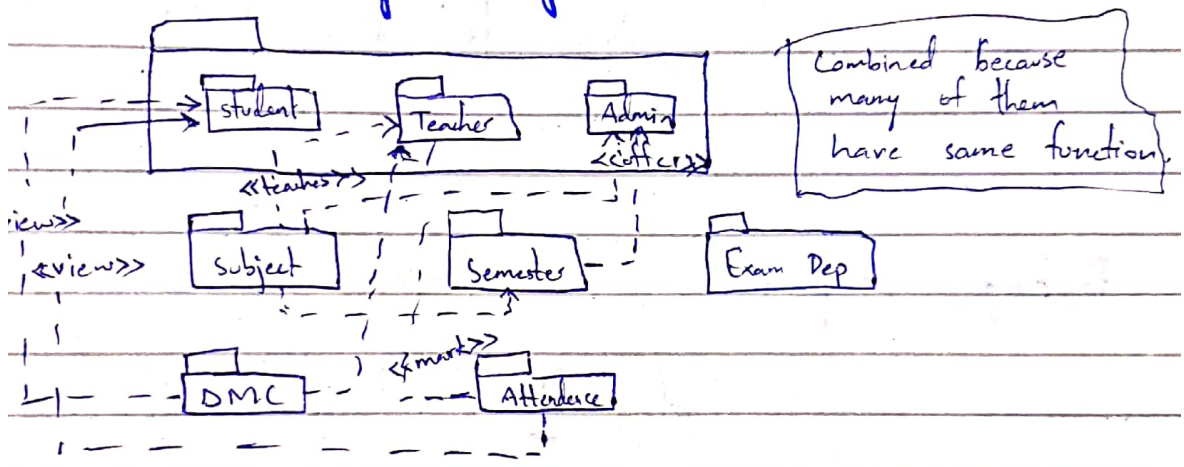
special order depend on order

order & special order depend on user.

<< ——— >>

Dependency Name.

LMS Package Diagram:



R — SE

Package Diagram:

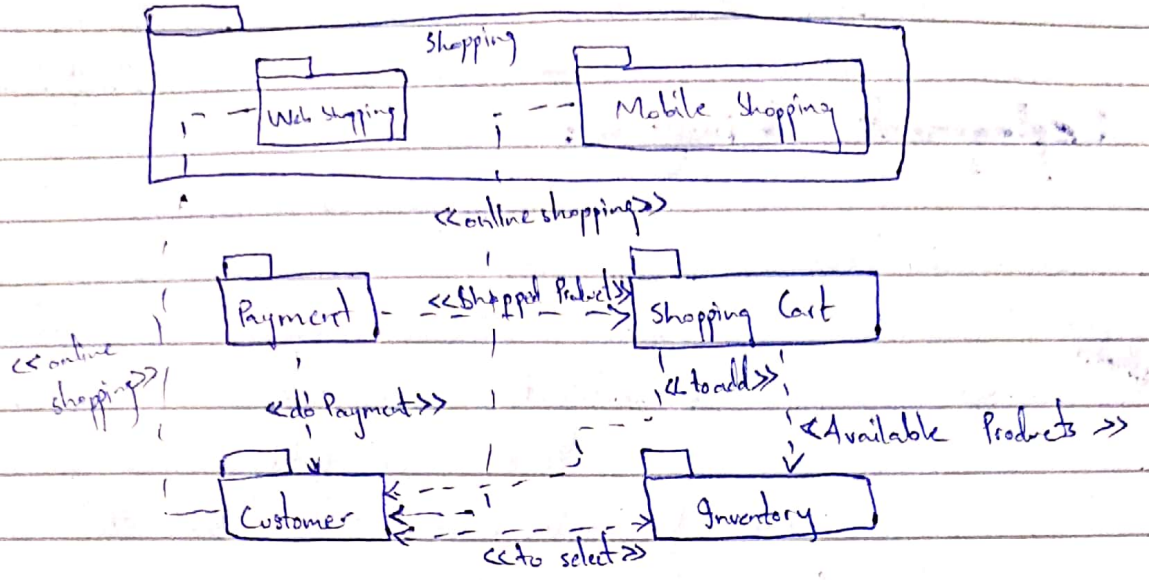
↳ Package Diagrams are structural diagrams used to show the organization and arrangement of various model elements in the form of packages.

↳ It depicts the dependencies between packages that makeup a model.

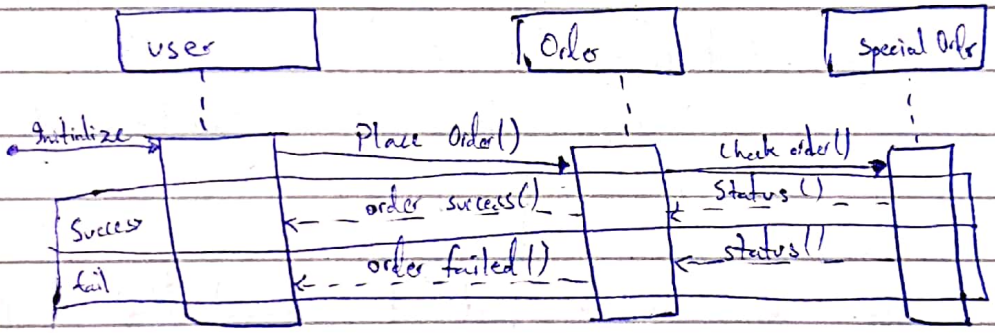
↳ Package diagram is created to increase readability of the program

↳ Classes names will be written in packages.

Example:



Sequence Diagram Example (Lec)



SE LAB (24/03/22)

Use Case:

- Login / Sign-up -
- Add preview / Display.
- Communicate with seller — PIN forget.
- Card expire.
- Balance not available.
- Insert card.
- Enter PIN.
- Card out of order.

Aggregation/composition are types of Association.

Test case ⟹ Use case

Deployment Diagram:

user — modem — Internet — website / server / Database

↳ How can we deploy a system in real world.

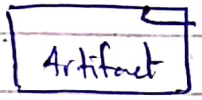
↳ How user accesses it after system is posted.

Methods:

1) Simple

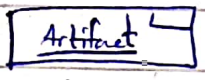
2) Artifact

↳ (Real world entity; source file, exe file, DB table, output file, DLL etc)



2) Artifact Instance.

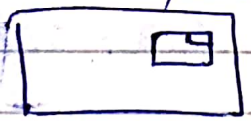
When name is underlined, it is an instance.



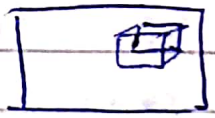
3- Node on which artifact executes/host

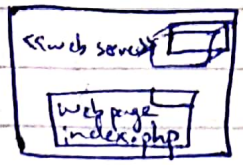
↳ host/server

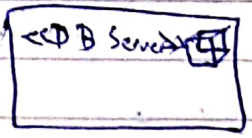
Either



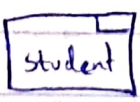
or

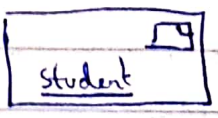






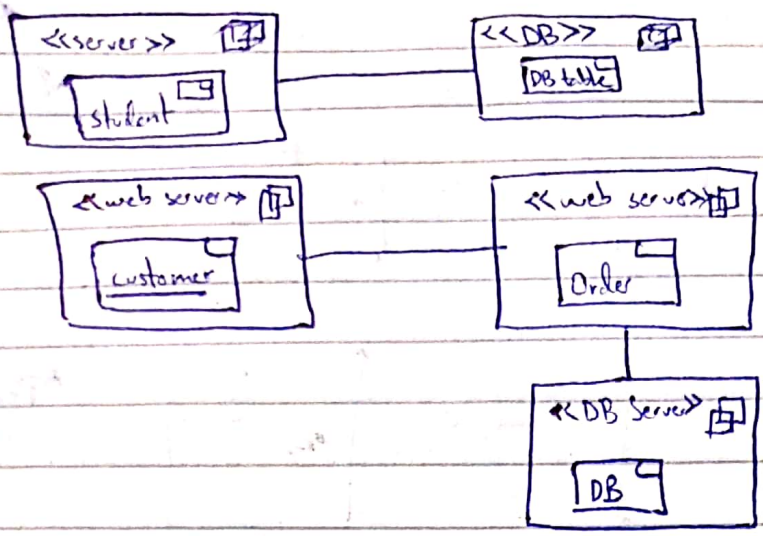
To make student Artifact.





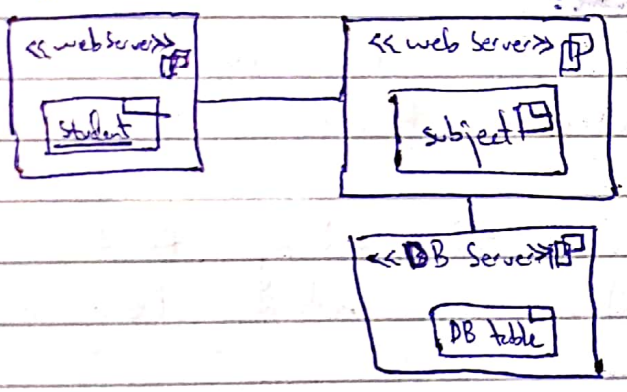
↳ Can never exists on its own

↳ Needs to be showed on machine/server.



Customer is accessing order underlying on webserver.

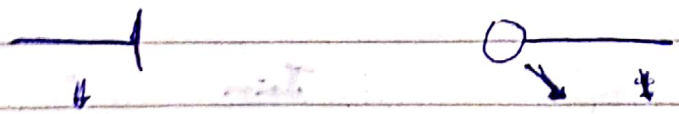
↳ Only shows hardware.



Component Diagram:

X Hardware Devices

- Only showing artifacts (no nodes, etc)
- Not hosted on server.
- Otherwise same as Peployment Diagram.
- To show the interaction, communication of the artifacts, we use component diagram.

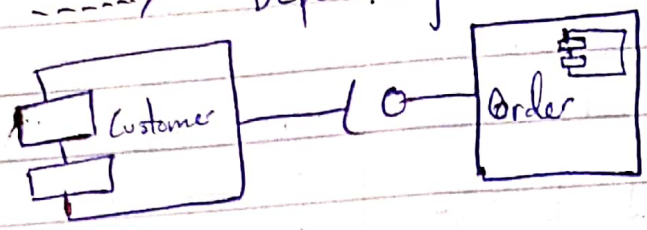


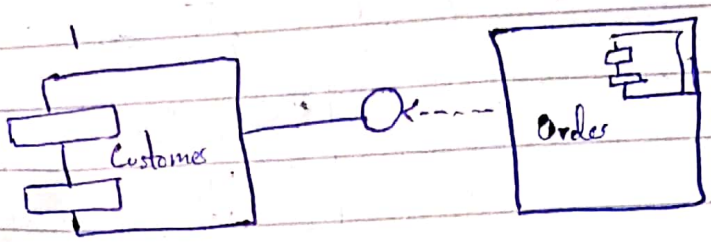
Component with this requires the interface/functionality

Component with this, has the interface/functionality.

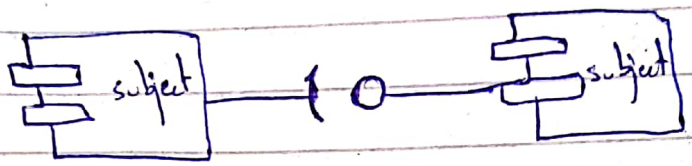
Independent ⟹ Provides Interface.

-----> Dependency.





Dependency



Activity Diagram:

↳ Dynamic

↳ All Dynamic Diagram have some starting point.

State Change ⟹ Activity Effect (e.g Input PIN)

State (By System) Activity (By user).

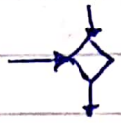
⟹ (Show) ⟹ Box with rounded corners.

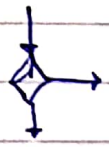
Start •

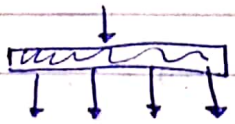
End ◉

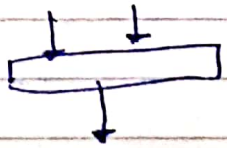
Activity ▭

Flow →

Merge 

Decision/ Branch (if/else) 

Fork 

Join 

One Input, multiple output (without decision).

Swim Lane

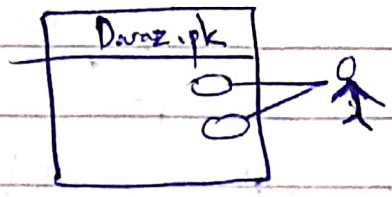
To show parallel.

SE (09/06/22)

Use Case:

1- Actor
2- use
3- Relation.

(Black hole of system).
↳ Ya don't write down each use case detail.

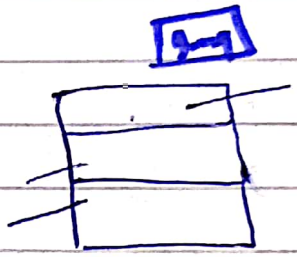


Static.

Both have Inverse signs.
- Inclusion << Include >>
- Exclusion << Extend >>.
- Dependency ----->

All the problems you face using use-case

Class / Object Diagram:



Name:
Attribute:
Function:

Association. ——
Directed Association. →
Aggregation ◇→
Composition ◆→
Dependency ----→
Inheritance ▷

1 —— * 0-∞
1 — n 1-∞
1 — 1
* — 1
n —— *

\+ Public
\- Private
\# Protected
underline Static.

underline ClassName → Object Diagram.

↳ (No operations / functions in Object Diagram).

↳ Identifier CL + Attributes.

↳ + Name : string

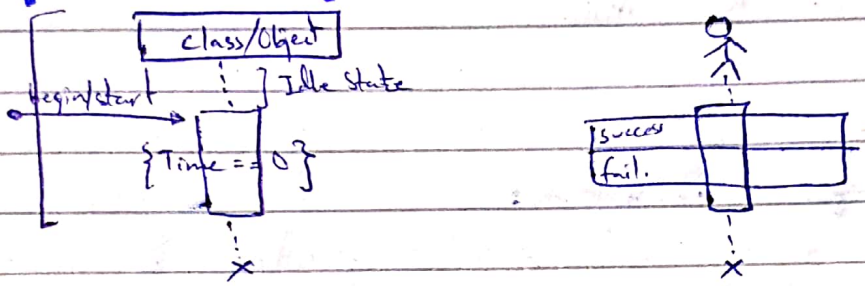
⇒ Name : "ABC".

↳ only Association (in object Diagram).

↳ (In class Diagram) ⇒ name each relation.

Sequence Diagram:

Dynamic.



→ Synchronous Message.

--> Reply.

→ Asynchronous Message.

Sequence Diagram. ⇒ Time

Collaboration Diagram:

Dynamic.

Deployment Diagram: Static

↳ Hardware Devices.

↳ Node: A machine on which a software/artifact executes itself.

Component Diagram: Static.

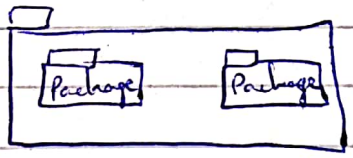
↳ Software Artifact. (real world entity).

↳ No Node.

Package Diagram:

↳ Package is a collection of related classes.

↳ contains class or sub classes.



Activity Dynamic

↳ User Point of view

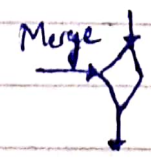
↳ Activity is performed by user, e.g. click button.

Start •

End ⊙

Branch

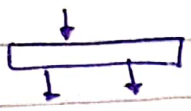
Merge

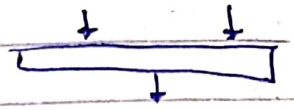


State. Dynamic

↳ System Point of view.

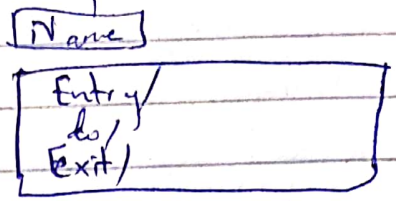
→ Flow

Fork 

Join 

↳ Activity has simple name

State can also have simple name.



SE (13/06/22)

Software Testing.

Q: What is the need of testing?

Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss and history is full of example: (book)

Software testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is defect free.

It involves execution of system components using manual or automated tools.

Purpose:

To identify errors gaps.

missing requirements in contrast to actual requirements.

Why software testing is important?

If there are any error, should be identified and removed before delivery of software product.

It ensures:

- Reliability.
- Security.
- High Performance.
  - ↳ that saves time.
  - ↳ cost effectiveness.
  - ↳ Customer Satisfaction.

Types:

Here are the software testing types:

- Functional testing.
- Non-functional testing / Performance testing.
- Maintenance.

Functional Testing:

- Unit T
- Integration T
- Smoke T
- UAT (User Acceptance Testing).
- Localization.
- Globalization.
- Interoperability
- So on.

- Non Functional testing /Performance testing.
  - Performance.
  - Endurance
  - Load.
  - Volume.
  - Scalability.
  - Usability.
  - So on.

Maintenance.

- Regression.
- Maintenance.

Testing Strategies in SE

Unit Testing:

↳ Basic approach followed by programmer to test the unit of program.

↳ Helps developers to know the individual unit of code is working properly or not.

Integration Testing.

↳ Focuses on construction and design of software.

↳ You need to see that integrated units are working without errors or not.

System Testing.

Software is compiled as whole and tested as whole.

This strategy check functionality, security, portability amongst others.

Error Terms:

Error: Human made mistakes, incorrect results produced errors.

Fault: State of software caused by errors.

Bug: Presence of error is bug.

Failure: Deviation of software from expected results.

Test Case

A test case is a specific procedure of testing a particular requirement.

It includes:

→ identification of specific requirement tested.
→ Test case success/failure criteria.
→ Specific steps to execute test.
→ Test Data.

Details of Strategies of SE

Unit Testing (White Box)

→ Individual components are tested.
→ It is a path test.
→ Focus on relatively small segment of code.
→ Aim to exercise a high percentage of the internal path.

White box testing:

When tester knows the code behind functionality and uses that knowledge for testing purposes.
Done by developers, code is visible for developers that's why called white box testing.
(Before giving to Test Engineer).

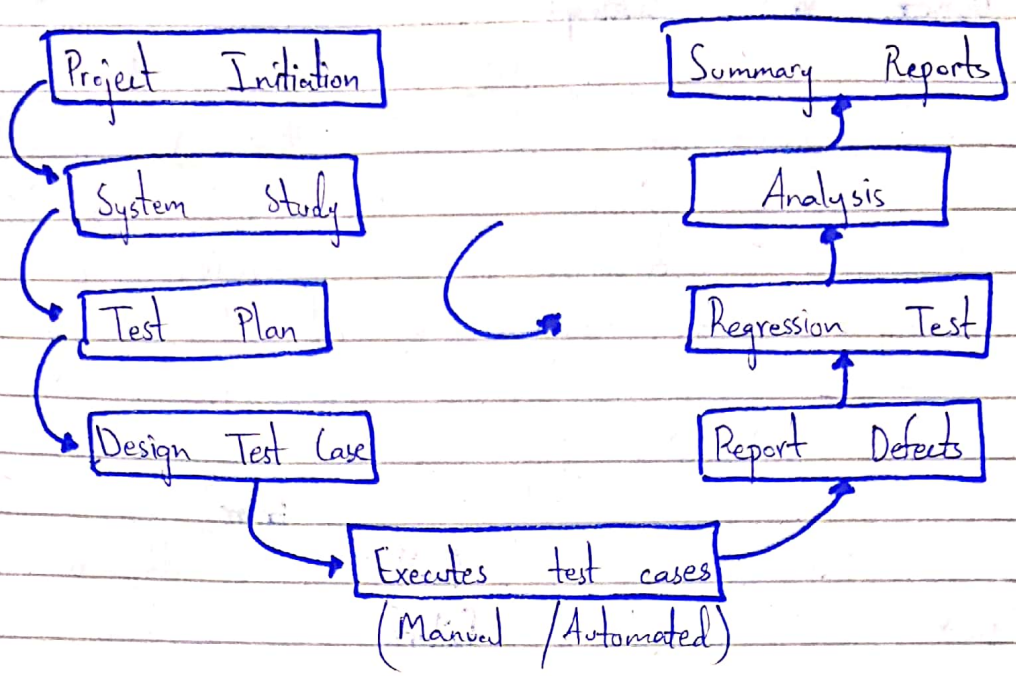
A person makes an **Error**.
That creates a **fault** in software that can cause a **failure** in operation.

Testing in SDLC

- Requirements.
- Analysis.
- Design.
- Coding
- Testing.
- Implementation.
- Maintenance.

(Testing starts from Requirement phase.)

Testing Life Cycle



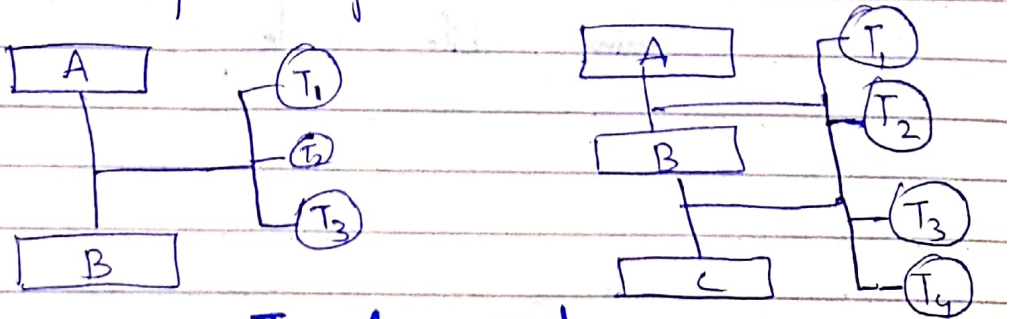
Disadvantage:

~~Code~~, cost, rapidly changing code, missed cases (If explanation is required, see the text under heading disadvantages of white box testing).

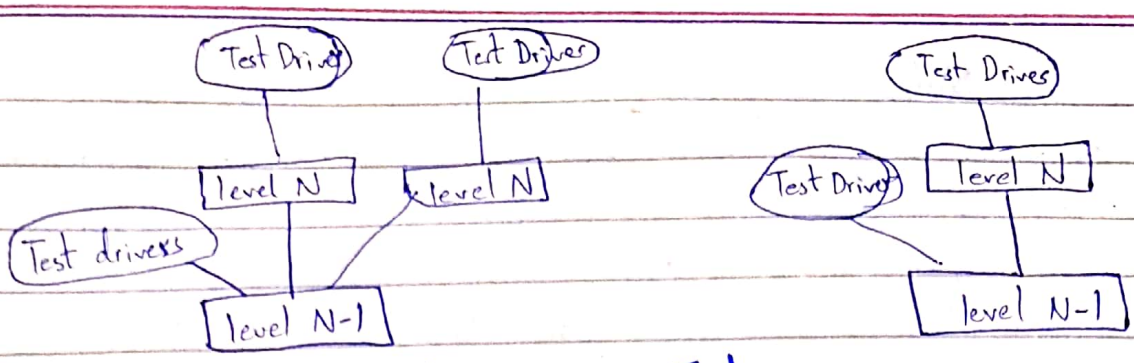
Integration: Testing:

After completing unit testing and dependent modular development, programmers connect the modules for Integration Testing.

- Top-down Integration Test
- Bottom-up Integration Test.



Top-down Testing.



Bottom Up Testing.

Top-down

- Control Program is tested first.
- Modules are integrated one at a time.
- Emphasize on interface testing.

Advantages

- No test drivers needed.
- Interface errors are discovered early, Modular features aid debugging.

Disadvantage

→ Test stubs are needed.
→ Errors in critical modules at low levels are found late

Bottom-up

- Allow early testing aimed at proving feasibility.
- Emphasize on module functionality and performance.

Advantages.

- No test stubs are needed.
- Errors in critical modules are found early.

Disadvantage

→ Test drivers are needed.

Stubs

A stub may simulate the behaviour of existing code.

جب ہمارے پاس ایک Module (sign-In) بن کر تیار ہو اور دوسرا Module (sign-Up) بننے والا ہو یعنی complete نہ ہو تو ہم Database میں dummy code لکھ کر Testing کرتے ہیں اس کو drivers & stubs کہتے ہیں۔

Stubs are basically known as "called programs" and are used in Top-down integration Testing.

Diivers are "calling program" and are used in bottom-up integration testing.

Black box testing.

- Black box testing is done by Test Engineer.
- Check functionality of application or the software according to customer's/client's needs
- code is not visible while performing test, that's why called black box testing.

Gray Box Testing.

- Combination of white and black box testing.
- Performed by the person who knew coding and testing.

SE (18/06/22)

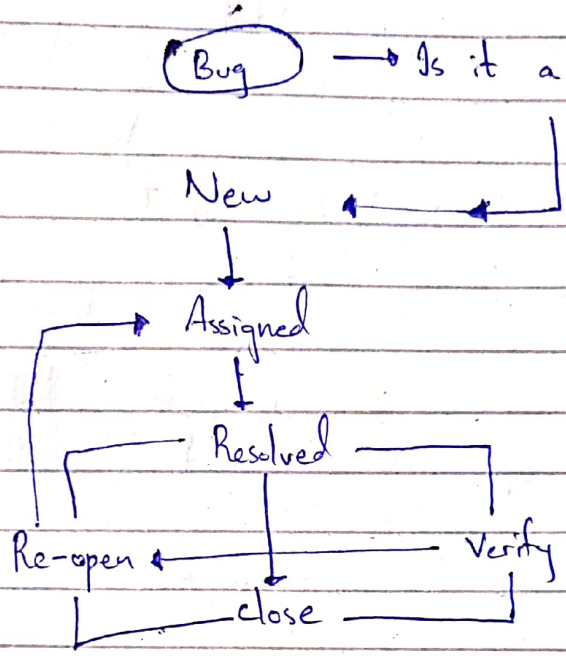
i) Unit Testing
ii) Integration Testing.
iii) System Testing.
iv) Usability | GUI (look n feel, speed etc).
v) Performance (loading, stress, data)
Breaking point
10,000 user
↳ How much data system can handle at one time.
vi) Security. (Unauthorized access, virus).
vii) Smoke // (small inputs/outputs) for
Not at
GUI
↳ Test main functions only.
viii) Alpha Testing
Both by Non-Dev
(Test on developers side, controlled condition)
↳ not developers but part of software house.
ix) Beta Testing.
(outside the software house, uncontrolled condition)
x) Acceptance Testing.
Testing of all user level requirements.
xi) Regression Testing.
(Test a function, if bug is identified, correct it, test Again)
↳ Combine functions then test again (whole sys).
xii) Monkey Testing.
↳ Testing function of a software randomly.

(During SDLC)
↳ Verification:
↳ Specifications, Guidelines, Inspection etc.
Is the Product right?

Validation: (SDLC / After development).

Is it the right product?
(meeting of user requirement).

Bug Lifecycle:



Functional Testing

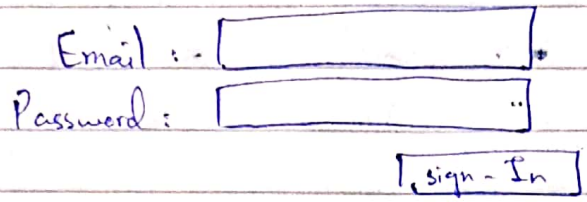
- Initialization.
- Usability.
- Smoke
- Regression.
- Monkey.
- Destructive. (till Crash)
- Recovery (After crashing recover)
- Acceptance.

Non-functional Testing.

- Performance
- Stress
- Security.
- Accessibility.
- Load Testing.
- Data Testing.

SE (20/06/2022)

Unit level Testing:



→ Email shall be correct
→ Password shall have 8 characters including Alphabet, Numbers, Special Character.

For Email:

Test Data	Expected Result	Actual Result	Result
abc@gmail.com	Accepted	Accepted	Pass
abc@	Rejected	Rejected	Pass
abc@gmail	Rejected	Rejected	Pass
abc@.com	Rejected	Rejected	Pass

For Password:

Test Data	Expected Result	Actual Result	Result
abc	Reject	Rejected	Pass
abc12	Reject	Rejected	Pass
abc@1	Reject	Rejected	Pass
abc@1234	Accept	Accepted	Pass
@8123456	Reject	Rejected	Pass

Integration Level:

Test Data	Expected Result	Actual Res	Result
① User Sign-In into the sys (after inputting credentials).	user should sign-In	user signed - In	Pass.
② user does not exist in the sys			
③ user entered wrong email			
④ user entered wrong password.			
⑤ user forget his password.			
⑥			