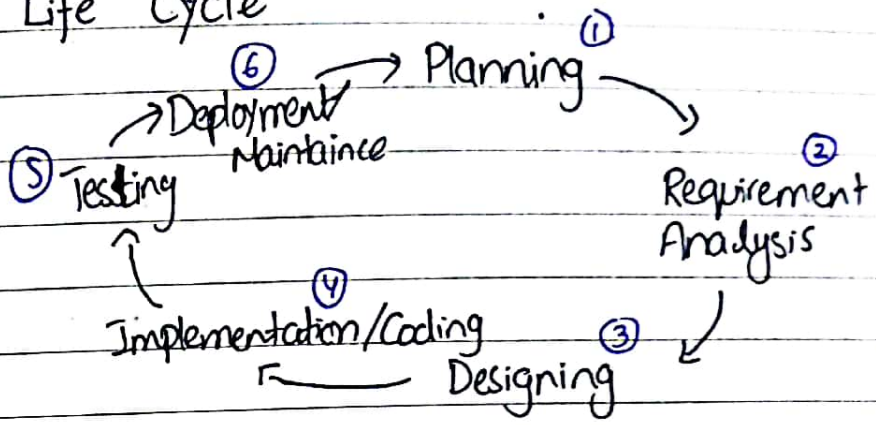
Date

SOFTWARE DESIGN AND ANALYSIS

→ Software Development Life Cycle



→ Software Development Model

- Classical Waterfall Model
- Iterative Waterfall Model
- V-Shaped Model
- Proto type Model → Making a model and then move to design phase
- Spiral Model → ~~Combines Iterative with waterfall~~ Risk management
- Scrum Model
- Incremental Model → adding large modules to smaller modules one by one
- Agile Model → Divide large projects in chuncks

→ Classical Waterfall Model

1. Planning / Feasibility of Study
2. Requirement Analysis (RSS)
3. Designing
4. Coding / Implementation / Unit testing (testing each Module side by side)
5. System testing / Integration testing (testing whole project)
6. Maintainance

* cannot go back to the previous step

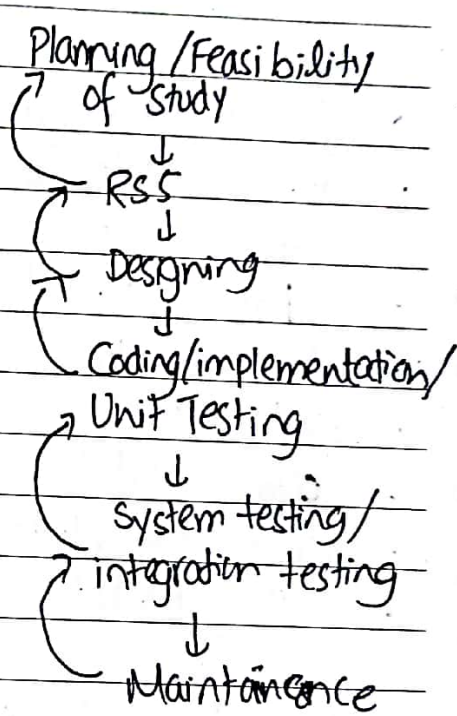
Date

* 60% effort goes to maintance as there is no feedback

Advantages
- good for small projects
- 1st model (1965-85 98% failed as no model existed) → acted as a base model

Disadvantages
- no feedback
- no parallelism (incase of one project others will be free)
- not suitable for large projects
- Maintance has alot of work to do - handle errors etc.
- no intermediate delivery

→ Iterative Waterfall Model
* allows feedback so saves time



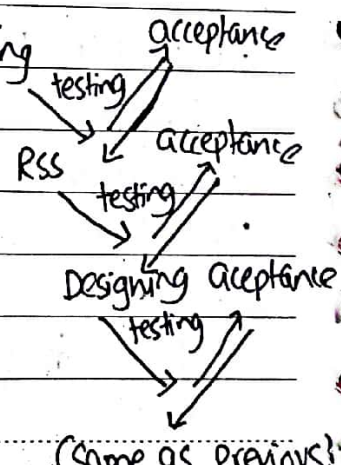
Disadvantages
- no intermediate delivery (no customer interaction)
- does not allow change
- no parallelism

* advantages are same as classical

→ V-Shaped Model
- every phase will be tested and in case it is accepted only then it will move to the next phase

Advantages
- time saving
- less chance of error
- tracking each phase side by side

Disadvantages
- no changes
- no feedback
- small scope (for small projects)



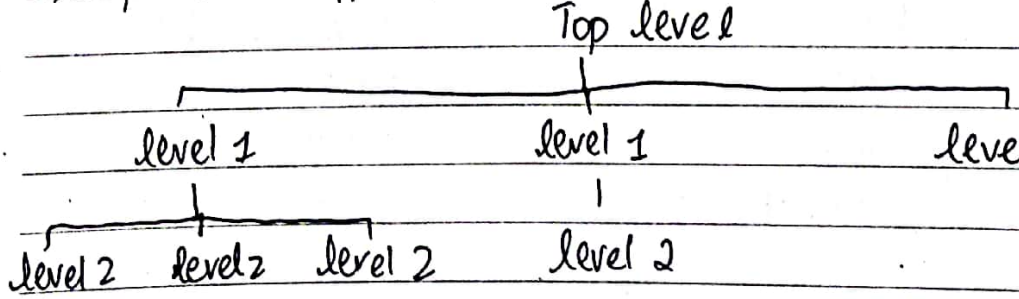
Date

→ Software Design Methods

- Factors to consider
  - ↳ single / distributed (check type of sofware project), standalone (usually offline) / network, small / large
    - ↳ system is running on multiple computers eg Bank systems or Stock exchange or web browsers has servers etc
  - ↳ Scope of Development - no. of modules/components / Complexity of coding / problems in implementation / either changes in modules or start from begining
  - ↳ Resources - man power / salary / time

- Methods
  - ↳ Structural Method: involves functions. for every task there is seperate function
  - ↳ Object Oriented. has real world example. Finds attributes and functions and make classes. Developent of an objects
  - ↳ Data Oriented: Deciding which data to bring to which class
  - ↳ Component based: Dividing projects in different components
  - ↳ Foreign Method: mathematical based. Used in Machine learning. for Safety and privacy. Expensive. Not easy to implement

- Structural Method/Functional Approach — uses top down approach
  - ↳ Top Down Approach ↓ seperates data from procedures

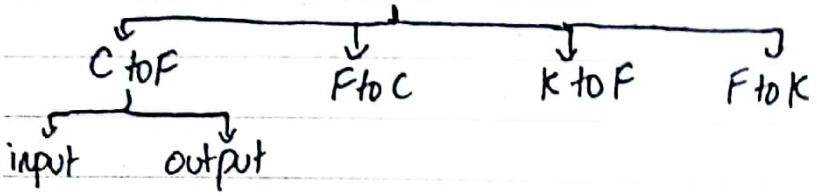


Date

- Object-Oriented Approach — Bottom up approach

↳ Top-Down Approach
- temperature conversion
- Calculator

temperature Conversion
↓
Selection Menu



- Drawback
↳ no dependency between functions
↳ might need to rewrite code in functions

- Object Oriented Approach
↳ every task is related to objects
↳ Object/ classes/inheritance/polymorphism/aggregation
↳ Bottom-up approch — good for reuseability
↳ encapsulate data and procedures
↳ good for making real world examples

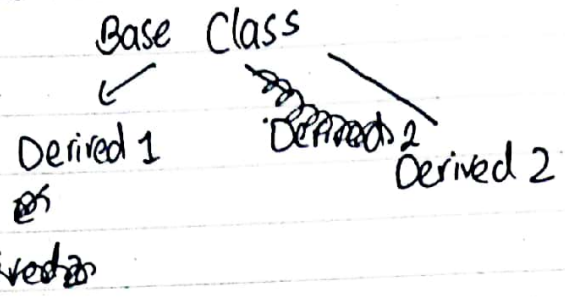
Types of Inheritance
1. Single Inheritance

Base
↓
Derived

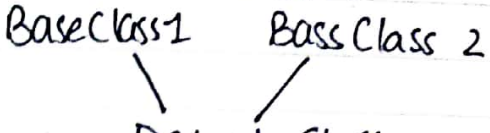
```
class Base{
    int a;
public:
    getdata(); };

class Derived::public Base{
    int x;
    int y;
```

Date

2. Hirearical Inheritance

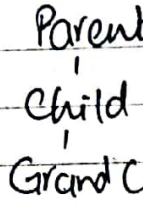


3. Multiple Inheritance

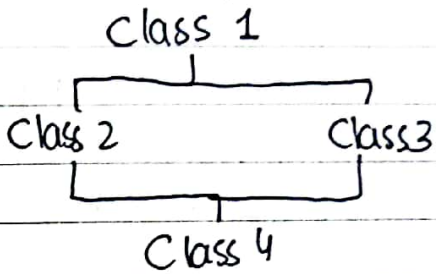


Class Derived :: public Base1 :: Base2{}

4. Multilevel inheritance



5. Hybrid (combination of multilevel and Hirearical)



* check Diamond problem (how class 4 accesses class 1 fuctions)

Polymorphism

compile time/ early binding
↳ compiler will know which functions to call through operator overloading or function overloading

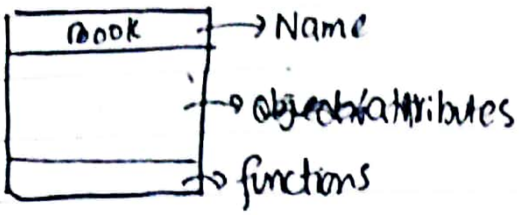
runtime/dynamic Bidading
↳ compiler does not know which function has to be called (parent / child). achieved through virtual functions

① UML (unified modelling language)
- requirements shown in form of diagram

② Use Case Diagram - not linked with OOP but good for showing requirements

Date

Class Diagram

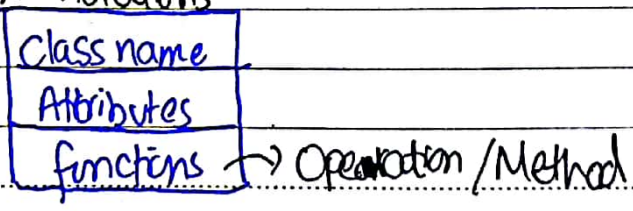


Types of UML Diagrams

- Structural Diagram
  - Composite Structure Diagram
  - Deployment Diagram
    - Object diagram
  - Class Diagram
    - package diagram
    - Component diagram
    - Profile diagram
- Procedural Behavioral Diagram
  - activity diagram
  - Used Case diagram
  - State machine diagram
  - Interaction diagram
    - Sequence diagram
    - timing diagram
    - Communication diagram
    - Interaction Overview diagram

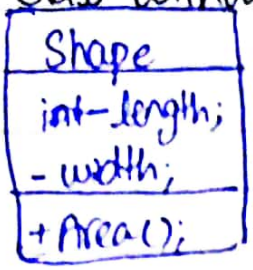
→ Class Diagram

- Components of CD
  - ↳ Classes (should be a noun/name of class)
  - ↳ Attributes
  - ↳ Functions / operation / Methods
  - ↳ Relationship between them
- Class Diagram notations

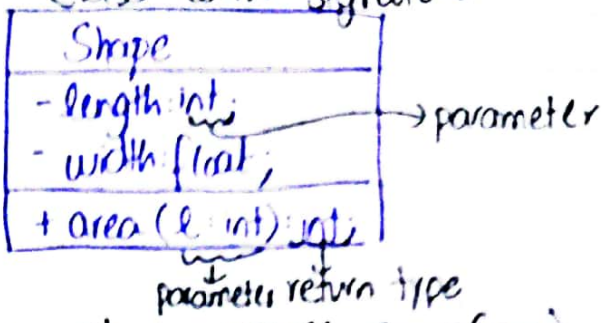


Date

eg. Class without Signature



Class with Signature



- Class Visibility (Symbols used to show access specifiers)
  - ↳ - private
  - ↳ + public
  - ↳ # protected

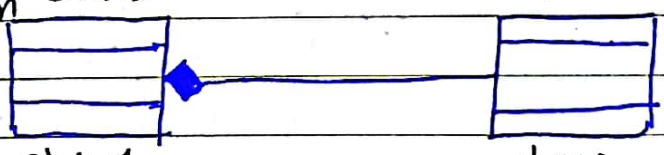
- Parameter passing in Class without Signature
  - ↳ area (in l: int): void; - only takes input nothing returned
  - ↳ area (out l: int): int; - returns integer
  - ↳ area (in out l: int): int; - the input taken is by reference and returns int

- Types of class diagrams
  - ↳ Conceptual: only making functions - no details of return type for functions etc
  - ↳ Specification: functions with data and return types - focus on interface or abstract data types
  - ↳ Implementation: has inheritance etc.

- Relationship between class Diagrams
  - ↳ Association

Both classes have no affect on each other. One class uses another

  - ↳ Composition

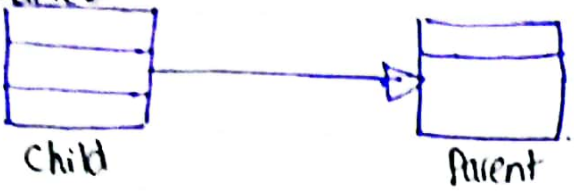


if class 1 is destroyed then class 2 will not exist

↳ Aggregation

If class 1 is destroyed, there will be no effect on class 2

↳ Inheritance



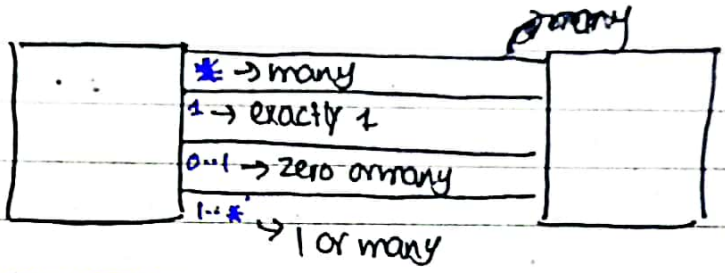
: Cardinalities

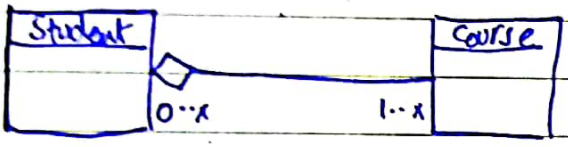
↳ One-to-One

↳ One-to-many

↳ Many-to-Many

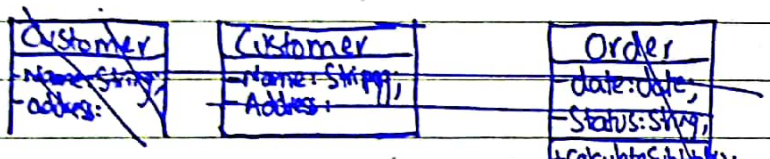
eg.

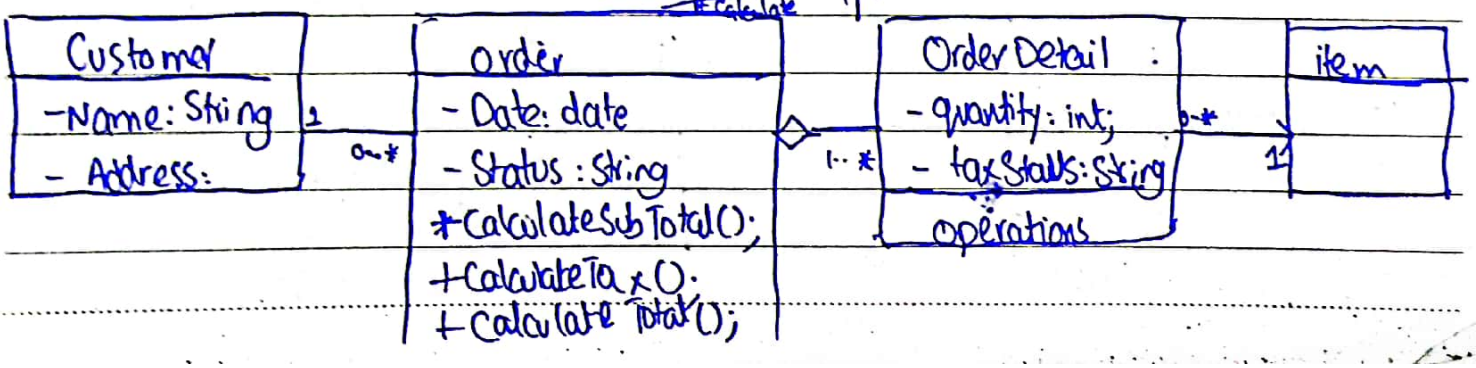




* generalization is Inheritance

Example - Order System



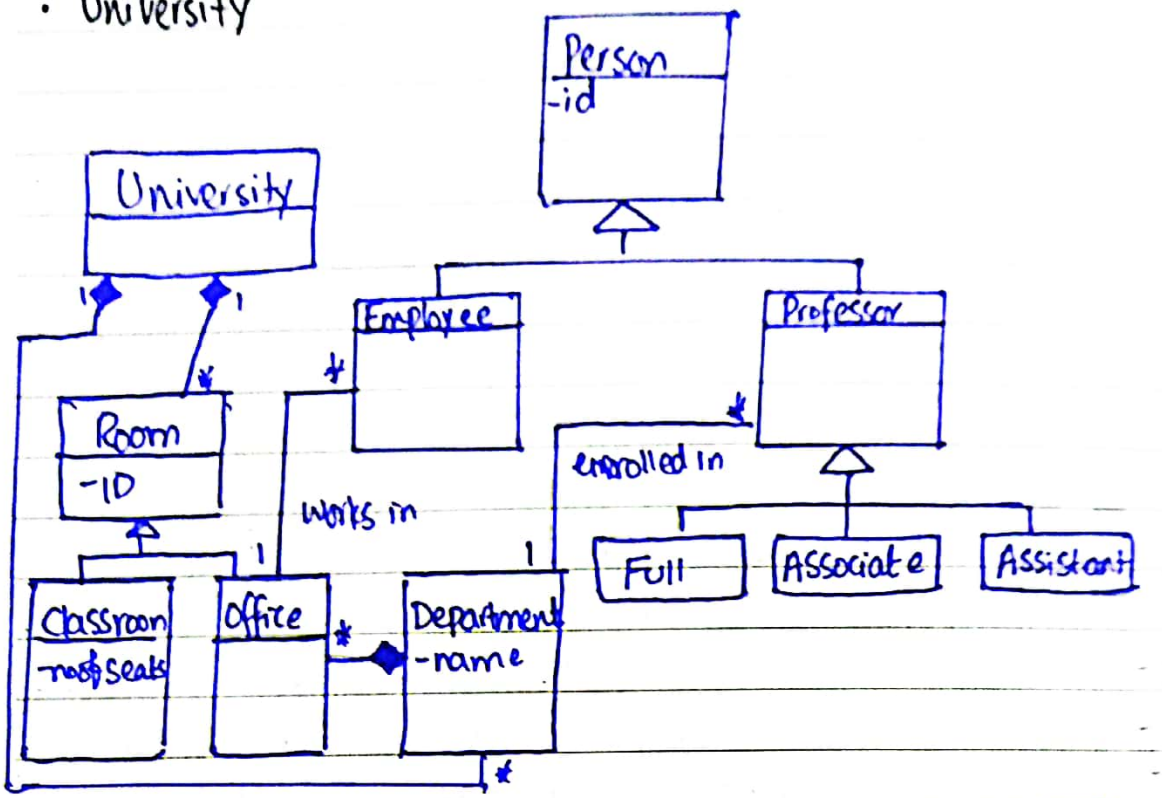


Date

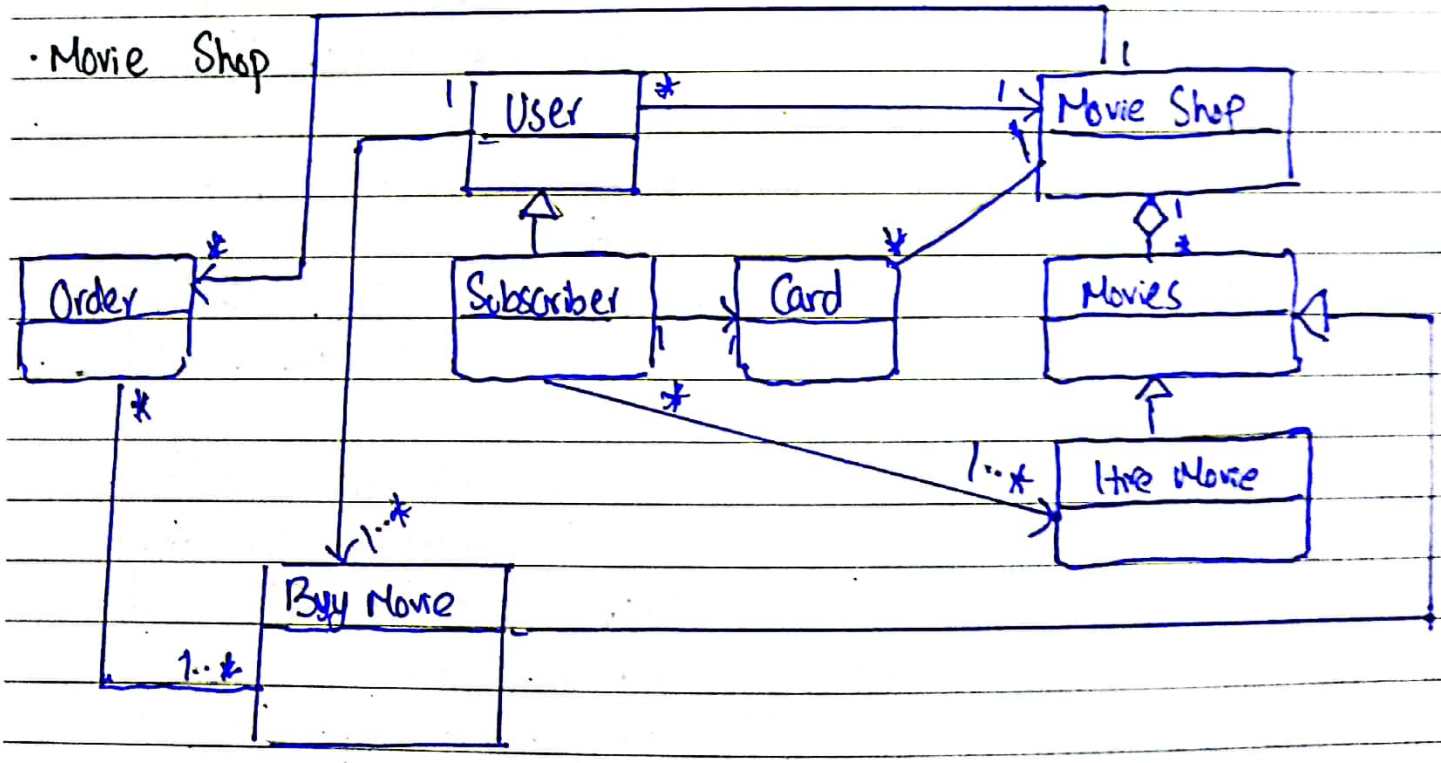
* Straight line means association from both sides

Q. → Class Diagram Practice Questions

· University



· Movie Shop



Date

→ Use Case Diagram
- how user can view/use
- when we represent the use case and their interaction with actor/users through the UML this kind of modelling is a use case diagram
- interaction between the ~~user~~ system and its actors

↳ Purpose.
- typically developed in the earliest stage of development. Developer often apply modelling ~~bec~~ to
  - ↳ Specify the context of the system
  - ↳ capture the requirement of a system
  - ↳ validate a system architecture
  - ↳ drive implementation and generate test cases.

↳ Actor: Someone who interacts with the use cases. (Noun) - plays role in the business model. Similar to a user but user can perform multiple tasks at a time but actor will perform only one task. Actor has responsibilities towards the system.

Actor gives input and ~~it~~ system gives output

↳ Usecase: The functionalites performed (Verb + Noun)

usecase

Each actor must be linked to a usecase otherwise actor will not be required but a usecase might not have an actor

↳ Communication link: the participation of an actor in a usecase is shown by connecting solid link. Actor may be connected to use cases by association indicating that the actor and the use case with one another using messages

↳ Boundary System: ~~the~~ & potentially the entire System as defined in the required document. For ~~eac~~ large

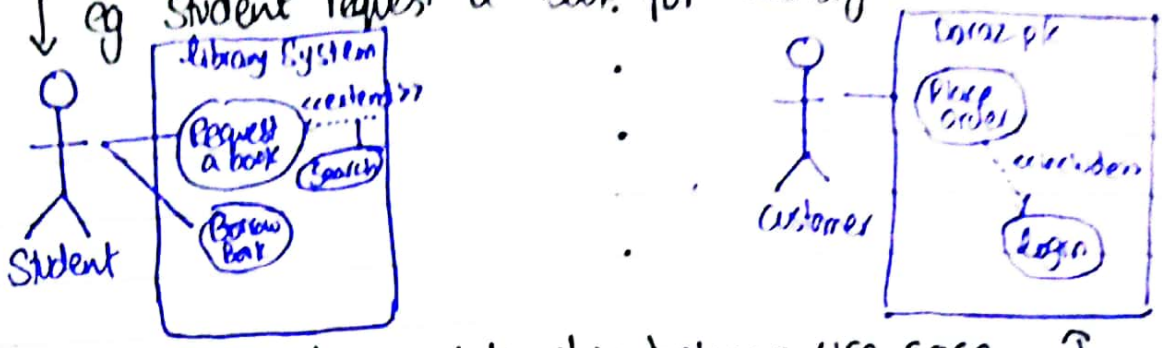
Date

and a complex system, each module are a subsystem maybe a boundary to the system

↳ Relationships

① Extends: optional relationship amongst the use cases

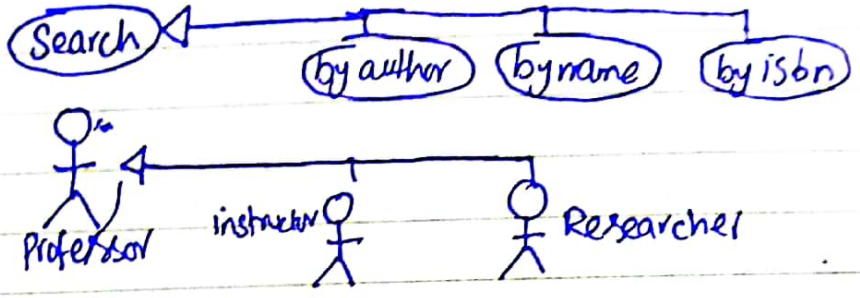
↓ eg student request a book for library



② Include: compulsory relationship between use case →

③ Generalization / Inheritance: Some functions / actors can be inherited

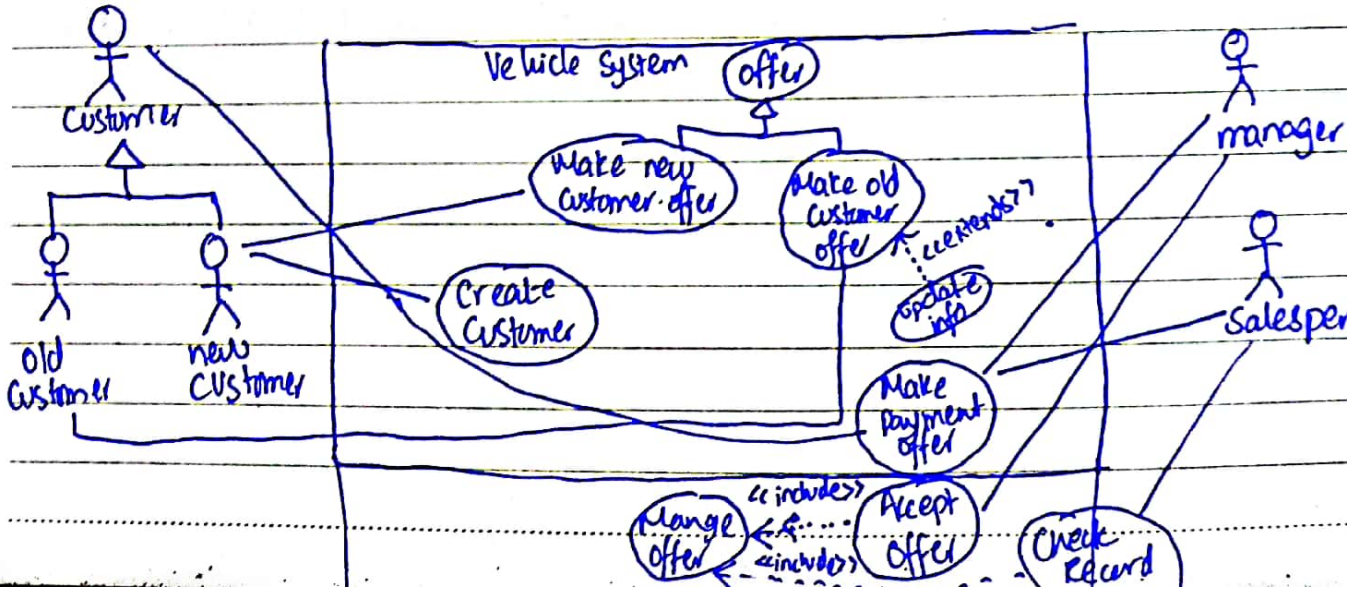
eg



· Vehicle System

↳ Actors: Salesperson, manager, customer

new old



Date

Usecase ID:	UseCaseName:	Description
Description:	Explain:	
Actors	User/System	
Goals		
Stakeholder		
Precondition		
Post Condition		
Basic flow		
Alternate path/ flow		

Who is asking to make it.

→ Step involved before

→ Steps involved after execution of use case (errors etc)

eg for quiz instant feedback

Description: Student will login. Go on quiz page, Slove quizes, Submit, get instant feedback. if no quiz rn feedback of previous

Actors: Student, Administration

Goals: Take a quiz / Review a quiz

Stakeholder: Students / administration

Precondition: Student must be logged in / Should be able to see quiz

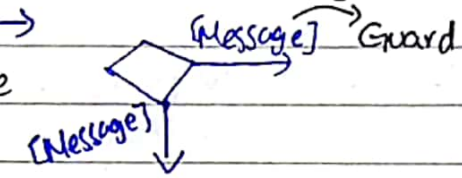
Post Condition: get instant feed after submission.

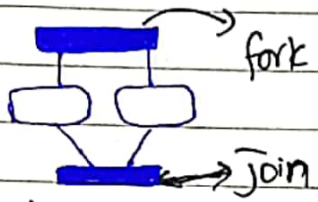
Basic flow: describe the entire procedure here and also mention alternative paths

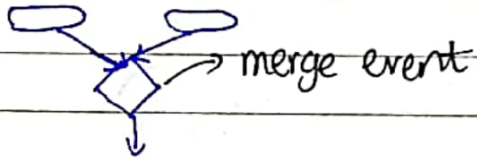
Date

→ Activity Diagram
↳ Behavioural Diagram

· Notation
↳ Initial State ● (filled circle)
↳ Action / Activity State ▭ (round corner rectangle
↳ Action Flow / control flow →
↳ Decision ~~Node~~ / Branching / node
↳ Guards
↳ fork
↳ Join
↳ Merge event
↳ Time event ⧖
↳ End ◉ or final State

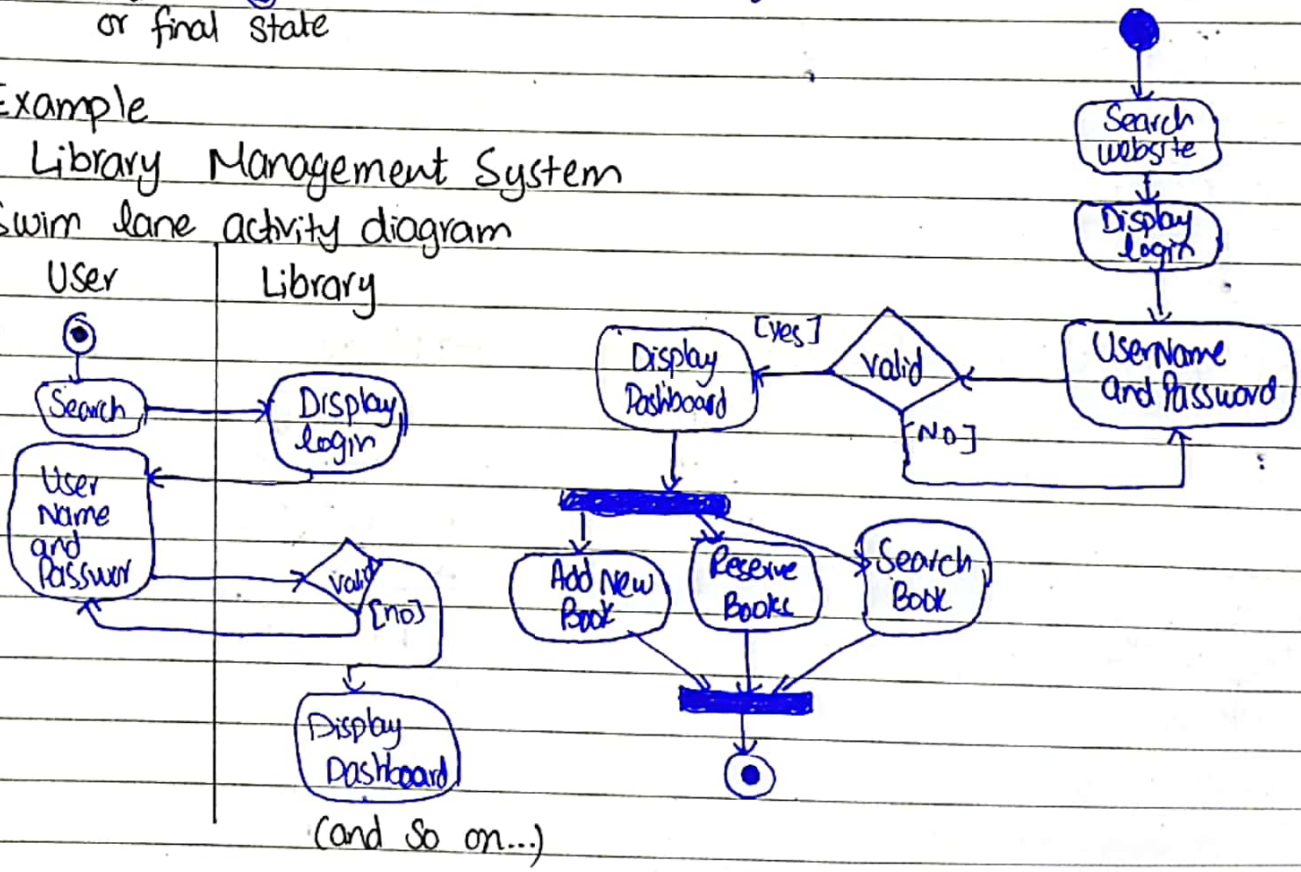






· Example
Library Management System

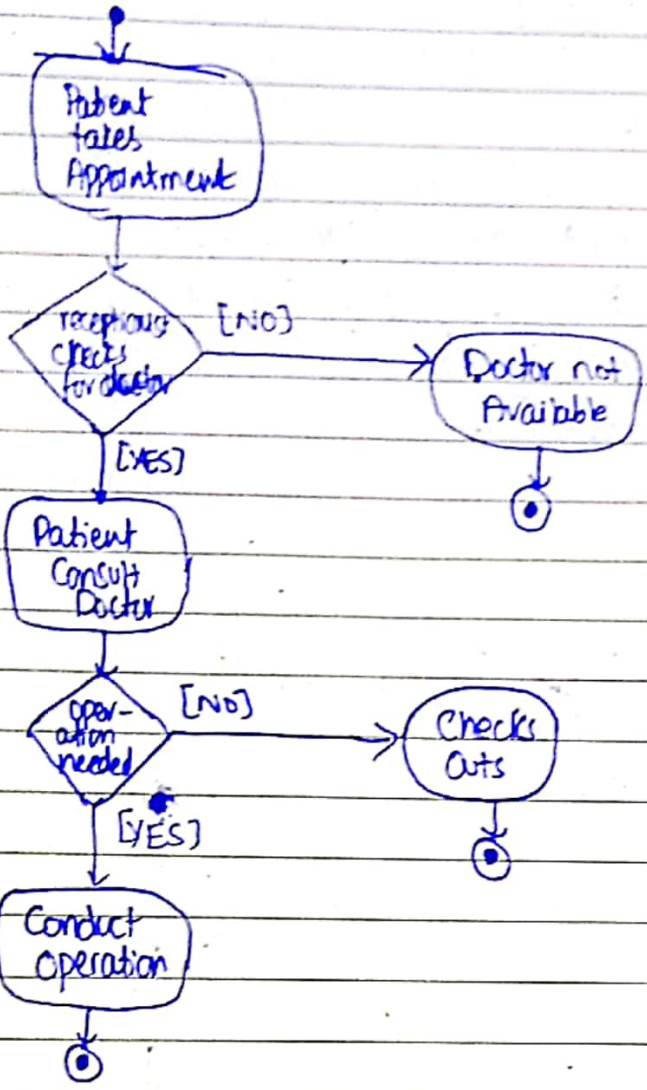
· Swim lane activity diagram



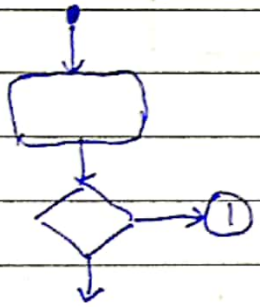
(and so on...)

Date

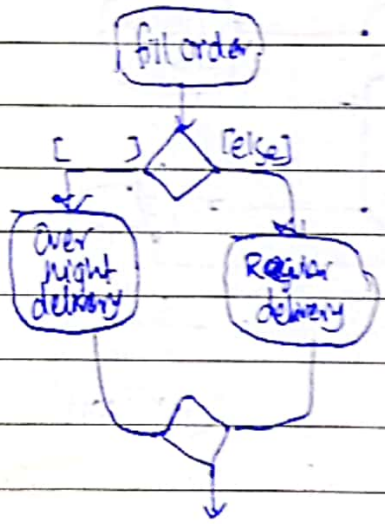
→ Hospital Management System (Activity Diagram)



→ Making chunks in diagram.

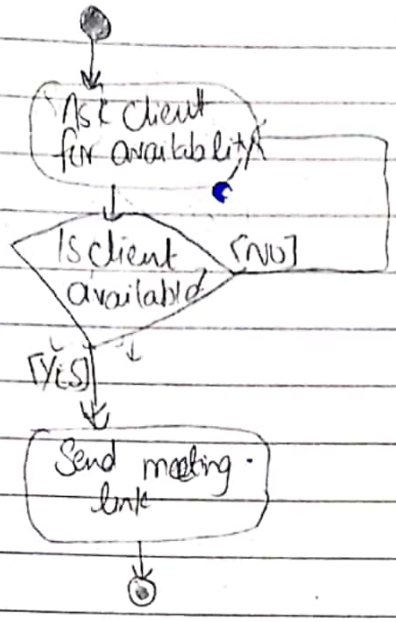


→ Merge



Date

Meeting with a new client - Activity diagram



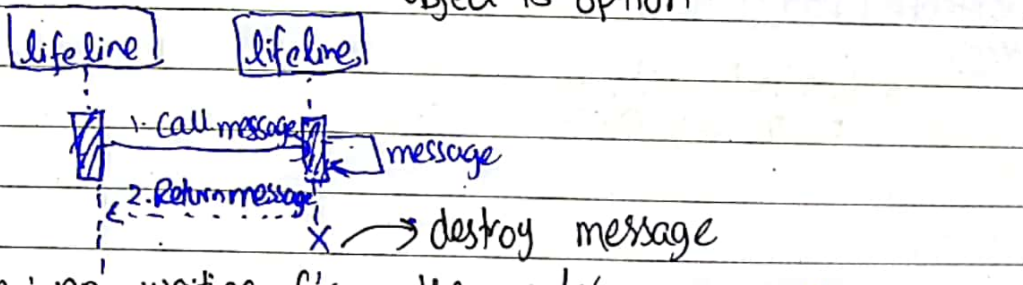
→ Sequence diagram
↳ concerned with time

· Purpose
↳ model high level active objects in a system
↳ model interaction between use case
↳ operations of models.

↳ object dimension - horizontal ↳ time dimension - vertical

· Notations

① Actor



② lifeline (object)

Object : class

* class name is necessary, object is option

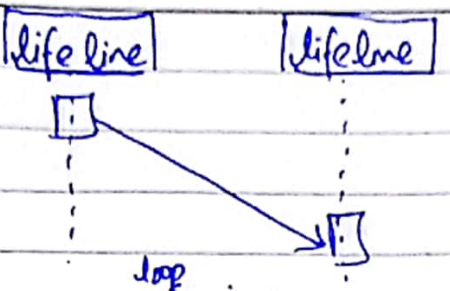
:class

③ Activations
④ Call message
⑤ Self message
⑥ Asynchronous / Sync: no waiting for another actor →
Synchronous →

X → destroy message

Date

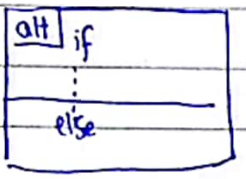
⑦ Wait / Duration



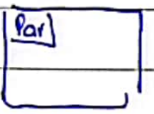
⑧ Note

⑨ Fragmentation - process in loop

⑩ alt - alternative



⑪ Par - parallel process



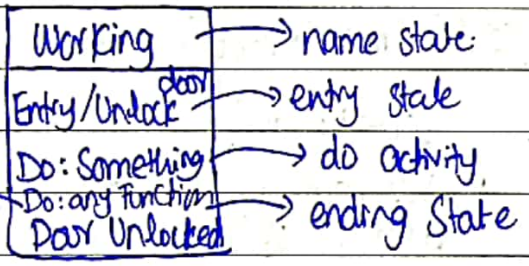
→ State Diagram

· State: is a situation in life of an object eg person can go through listening, sleeping etc. Functions / events

· This diagram shows how an object responds to various states (events) by changing its states

· This diagram is used to model the dynamic behaviour of a system.

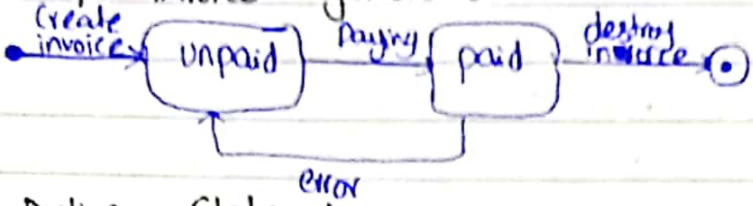
· State



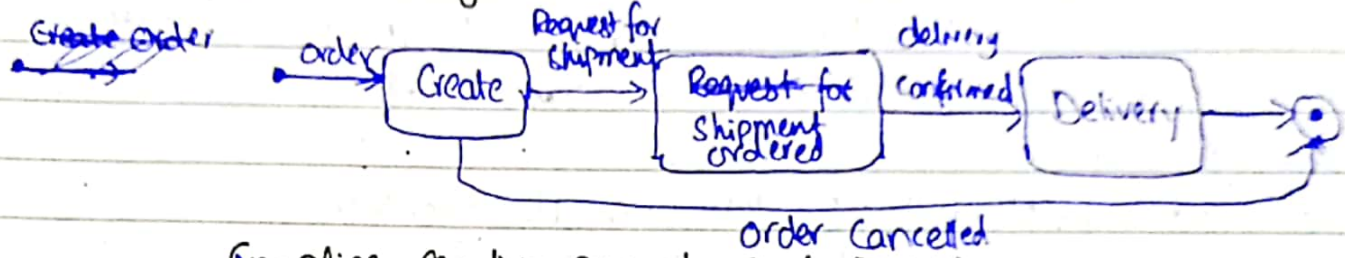
· An object can respond differently to an event

Date

- Notations
  - Starting state
  - State
  - Transition / Input
  - ending state
- Example for invoice generation



- Order Delivery State Diagram



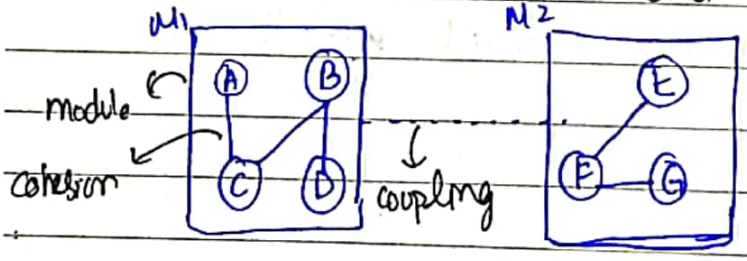
Coupling equation principle and Design

→ Cohesion and Coupling

- Module has functions. there should be no dependency between them
  - ↳ eg class. May have one or more functions. When data is coming from another module - Cohesion
    - → one function should be doing one task

Cohesion should be high and coupling should be low

- Principle concept used here are ~~modular~~ divide and conquer
- Cohesion within a module and coupling within different modules



Date

→ Types of cohesion

↳ logical equation: all elements of a module perform similar operation
eg update, delete, add etc all functions should be in same module

↳ procedural equation: if modules are implemented in a sequence (has steps which each step performed by different function) can be added in same module.

↳ communication ~~func~~equation: if 2 modules are passed same data structure, they can be brought in the same class.

↳ sequential equation: eg in binary search, sort(), display() etc... should be in same module

↳ functional equation: if function is using alot of attributes from another module, add those attributes in the same module.

→ Types of Coupling

↳ Data coupling: M1 will get data when M2 executes.

↳ Stamp Coupling: Passing data structure (?)

↳ Control Coupling: flow. First M1 will execute and then M2. M2 will be waiting Avoid this.

↳ ~~ext~~ extra Coupling: waiting for external resources. eg another module might be accessing database

↳ Common coupling: different modules share same data. At the same time, these modules should not use same data.

↳ Content coupling: if M1 module will execute only then other will. M2 might never execute

↳ Message coupling: depends on message which module will execute.

Date

→ Solid Principles
- during changes, maintance and writing code these should be used to enhance the code.
- called Solid principle by some Robert
- design principles which manage most of the software design problems.
- instead of removing stuff only add on to enhance. no need to modify already written code.
- code should be understandable,.... etc

↳ Advantages
- helps to write a better code
  ↳ avoid duplication
- easy to maintain
- easy to understand.
- reduce complexity

S : Single responsibility principle (SRP)
O : Open and Close principle (OCP)
L : Liskov substitution principle (LSP)
I : Interface segregation principle (ISP)
D : Dependency inversion principle (DIP)

↳ SPP : a class or module should have only one reason to change

```
eg class Marker{
        String color;
        int price;
        int year;
    public void Marker(String color, int price, int year){
        this->color = color.;
        this->price = price;
        this->year = year; } };
```

Date

```
Class Invoice{
    private Marker obj;
     int quantity;
    // constructor
Change① → public calculationTotal(){ // this function can be changed according to requirement
     int Totalprice = obj.price * quantity;
     return Total price; }
Change② → public void printInvoice(){ ..... } // what change will be reflected here
Change③ → public vord saveonDb(){ ..... } }; as well
```

* to cater this add these functions in different classes.

↳ OCP: open for extension but no modification of already written code

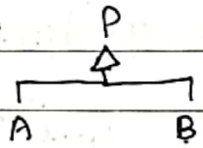
eg interface invoice{ public save(); }

Class save SQL extends invoice{ save(); }

Class SaveFile extends invoice { Save(); }

added later → Class Save MongoDB extends invoice { Save(); }
no need to change previous code

↳ LSP : if class B is subtype of class A then we should be able to replace A and B without any ~~change (msg)~~ execution error. Output can be different but it should execute. Used in inheritance.



↳ ISP: interface should be such that client should not implement such functions that it does not need.

eg ▷ employee : takeorder(), serve(), cook(), washdishes()
- cook : cook()
- helper : washdishes()
- waiter : takeorder(), serve()

} violates LSP

Date

↳ DIP : class should dependence on interface rather than concrete classes

Mouse → wired mouse, wireless mouse
Keyboard → wired keyboard, wireless keyboard

```
Class Macbook {
    Keyboard keyboard;
    Mouse mouse;
    public Macbook (Keyboard WiredKeyboard, Mouse WiredMouse) {.....}
}
```

Violates DIP – should be generic eg pass simple keyboard and mouse (not wired/ wireless)

→ Object Oriented Matrices
↓
check maintainability, reuse-ability, complexity

· rules to measure coupling, cohesion etc
· measure and quantitative techniques of a software
↳ code quality
↳ software quality – things used to improve code – make code better
· measuring the concept of
① CK matrix
Set of object oriented: (class based)
complexity, maintainability
↳ Weighted Method per class (WMC) : measure the complexity of class by counting the total methods of class
↳ Depth of Inheritance Tree : Count number of children, complexity will be accordingly. overall
↳ Number of children : count number of immediate children (reuseability and maintainence will depend on this)
↳ Coupling between objects : if class A uses / refence B C D the CDO = 3 (A is referenced by 3 classes)
↳ Response for class : the number of method that are executed in return for a message from a class – highly response. Other classes depend on this class

Date

eg 15 methods will execute then it will go to other class
↳ lack of equations: A variable can be shared between many class

② LK Matrix - class and operation ~~free~~ base
↳ Class size: total number of methods both inherited and local + total number of attributes encapsulated by a class. Inherited members have more weight.
large value of class size → too much responsibility / low reuseability
↳ number of operation over written by a sub class: account of methods in sub classes that have been redefined.
large → issue in design
↳ number of operation added by a sub class: Account of the new method appearing in sub classes. a large number of operation value can indicate design abstraction violation
↳ Specialization index: degree to which sub class are differentiated from super classes. Computed as (number of operation overwritten * level at which the class resides in the inheritance hierachy) / total number of methods of class
low specialization index → conformance to superclass index

Date

→ Design Patterns

- ~~neat~~ choose the design pattern according to the problem faced and this will help in
  ↳ maintability, Scalability, reusability or it increased
- Similar things are combined to increase efficiency
- "each pattern describe a problem which occur over and over again in your environment, software or application and then describe the core solution to that problem in such a way that you can use this solution a million time over without ever doing it the same way twice."
- "design pattern's thumb rules are different concepts using which you can solve the problem of modelling real world examples into object oriented design"

↳ essential elements of patterns

① pattern name: it is a handle which we can use to describe a design problem in a word or two. Naming a pattern immediately increases our design vocabulary. It makes it easier to think about designs and to communicate them and their trade off to others. Finding good names has been one of the hardest part of developing our catalogue. ~~The problem~~

② The problem describes when to apply the pattern. It explains the problem and it contexts. The problem will include a list of conditions that must be met before. It makes sense to apply the pattern.

③ Solution: The solution describes the elements that make up the design, their relationship, responsibilities and collaborations. The solution does not describe a particular concrete design or implementation because a pattern is like a template that can be applied in many different situations.

④ Consequences: The consequences are the results and trade offs applying the pattern. Consequences are often invoiced when we describe design decisions. They are all critical for evaluating design and for understanding the cost and benefits of applying the pattern.

↳ Classification of design patterns

① Creational design patterns
② Structural design patterns
③ Behavioral design patterns

} These are different from each other on the basis of their level of detail, complexity and scale of applicability to the entire system being designed

① Creational: As the name ~~suggests~~ it provides the object of classes mechanism that enhance the flexibility and reuseability of existing code. It reduces the depenency and controlling how the user interaction with our classes so we would not deal with complex constructions. Its types are as follows:

↳ abstract factory.
↳ Builder
↳ factory method
↳ prototype
↳ ~~Sig~~ Singleton

② Structural: They are mainly responsible for assembling objects and classes into a larger structure making sure that these structures should flexible and efficient. They are very ~~as~~ echancing readability and maintainability of the code. It also ensures that functionalities are properly seperated and encapsulated. It reduces the minimal interface between independent things. Structural design are used for structuring more than one classes or objects together. Moreover we will

Date

develop deep inheritence and ISP (Interface Segregation) etc (can be other solid principles as well). Types.

↳ Adapter
↳ Composite
↳ Bridge
↳ Decorater
↳ facad
↳ flyweait
↳ proxy

③ Behavioral : They are responsible for how one class communicates with others. It is also used for identifying and setting up common communication pattern among objects. They deal with class and object interaction and distribute their responsibility

↳ Chain of responsibility
↳ Command
↳ interpreter
↳ iterator
↳ mediated
↳ memnto
↳ observer
↳ State
↳ Strategy
↳ template method
↳ visitor

↳ Benefits :

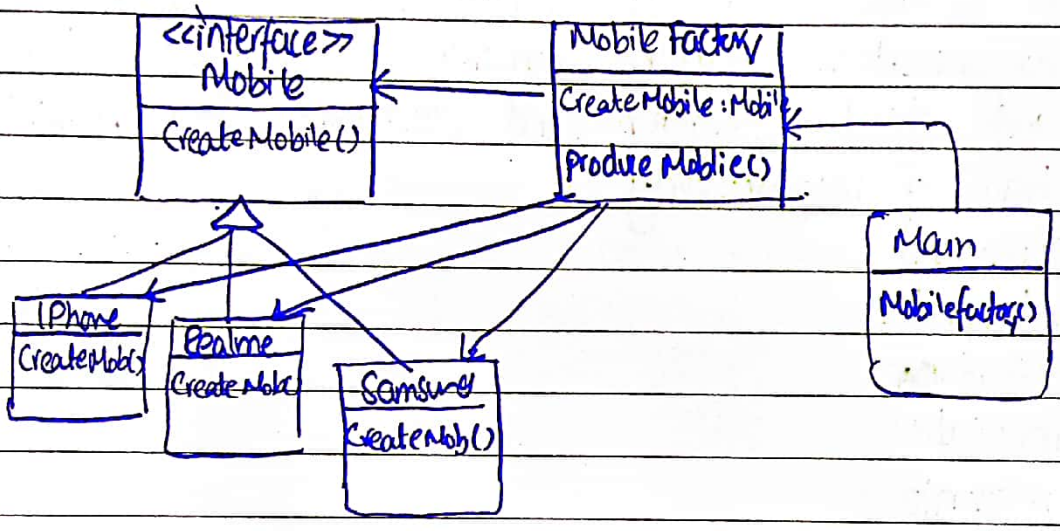
- improves the performance of the system
- Solve the bottleneck of the problem
- best design for the system is possible
- improve the code for writing in a more object oriented way like inheritance and encapsulation

Date

- development process is speed up with well designed principles
- ~~Sch Such as~~ clear seperation of modules and loosely coupled system
- reuse things across all applications or modules

→ Factory Method

- Creational design pattern that provides an interface for creating object in a super class but all subclasses to alter the type of object that will be created.



↳ Advantages

- you avoid the tight coupling between the creater and concrete classes ~~(or concrete product~~
- Single responsibility Principle and Ocp not violated
- you can move the product creation code into 1 place in the program making the code easier to support
- you can make new types of products in the program without breaking existing client code

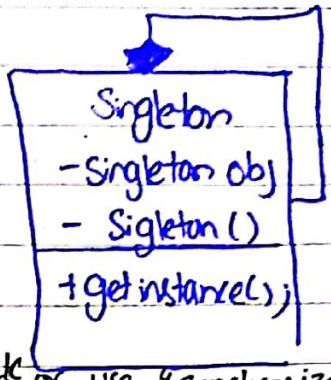
↳ Disadvantages

- the code may become complicated since you need to introduce alot of nuse of classes to implement the pattern.

Date

· the best senario is when you ~~are~~ can introduce the pattern into an existing hierarchy of creater class

→ Singleton Pattern



· making private constructor would mean that it cannot be called from public space (outside the class)

```
class Singleton{
    Singleton * obj;
  private:
    Singleton() { };
public:
  Singleton * getinstance(Singleton obj){
        if (obj == null){
            Singleton obj = new obj;}
        return obj;} };
```

to save multithreading issue or use mutex lock or use "synchronized"

```
Singleton *obj = new obj();
```

```
Singleton obj1;
obj1.getinstance();
Obj1.get instance();
```

will return same as obj has already made

· only one Singleton object can be made

· Can be problem in multithreading if if(obj==null) run on the same time in multiple threads. Singleton pattern ~~failed~~ violated as more than one objects will be formed.

↳ Advantages

· flexibility: Since the classes control the intenciation of ~~problem~~ process

· Save memory - no more that one object can be formed

· ~~when~~

Date

→ Structural Design Pattern

- Structural design pattern are ease to design by identifying a simple way to describe relationship between entities
- flexible interconnecting modules which can work in larger system

↳ Adapter design Pattern

- match interfaces of different classes, the adaptor design patterns allows in compatible classes to interact with each other by converting the interface of one class into an interface expected by the client
- It improves reuseability of older functionality.
- A class needs to be reused which does not have an interface.
- We need to work through a seperate adaptor that adapts the interface of the existing class without changing it.
- Client does not know whether the work with target class. directly or through another alternative that does not have the target interface
- Advantages

① Allows us to use code from 3rd parties relatively

② No any need to change existing classes to use new code

③ Loose coupling and SRP is enforced in this pattern.

④ A change in either the client interface or the adaptee only means a change to the adapter

⑤ It allows the reuseability of an existing code and functionality

⑥ We can isolate the interface from the data conversion code, thus supporting the SRP

Date

① We can introduce new variants for adaptors in our application without breaking the existing client code

· Disadvantage

① It's some time easier to just manually refactor the client interface, than it is to use the adaptor pattern.

② If you are using the class adaptor pattern you will want to avoid dimorad inheritance at all costs. This is unlikely to happen but if it does, you will be in trouble.

③ This design pattern increases the overall complexity of the code.

④ It is relatively ~~this~~ Simpler to change the service class the matches the rest of your code.

↳ Composite Pattern

· It lets you compose objects into tree structures and the work with these structures as if they were individual objects

· It is used where we need to treat a group of Object in similar way as a single object

· Tree structure represents part as well as whole hierarchy. A thing made up of several parts or elements

· It is a partitioning design pattern and it describes that the group of object are treated same way as a single of instance as the same type of the object.

⇒ Implementation guidelines

· represents part whole hierarchy of objects

· clients ignore the difference between the compositions of objects and individual objects.

Date

- Advantages
① Reduces code complexity by eliminating many loops over the homogeneous collection of object
② This intern increases the maintainability and testability of code with fewer chances to break existing running & tested code
③ The relationship is described in the composite design pattern isn't a sub class relationship, it a collection of relationship which means client or API user does not need to care about operations like translating, rotating, scaling and drawing whether it is a single object or a collection of objects
④ Simplified hierarical objects
⑤ flexible object structure
⑥ encapsulation
⑦ recurssive operations
⑧ reuseable code

- Disadvantages
① limited type checking
② The use of common interface for both leaf and composite objects limits the type checking capabilities at compile time. It may not be possible to distinguish between individual objects and a group of objects at compile time
③ Performance overhead espically when the hierarchical structure is larger and deeper
④ The recurssive nature of operation can result in multiple traversal of hierachy impacting the system performance.
⑤ Complexity in structure modification: modifying the structure of composite objects dynamically can be complex.

Date

Adding / removing objects from hierarchy may require careful management and updating of references.

(5) Lack of transperency: it hides the difference between individual objects and the groups of objects by providing a uniform interface. While this simplifies client code but it also reduces transperancy. Clients may not be aware of the internal structure of the hierarchy or the specific objects they are interacting with which can make debugging and troubleshoting more challenging.

(7) Difficulting in designing component interface: designing a component interface that caters to both individual and group of objects can be challenging. finding the right balance and defining operations that are relavent and meaningful for all components in the hierarchy requires careful consideration and design decisions

→ Behavioral Design Pattern

↳ Observer

- observes the system and notifies the change to everyone
- example: auction system, amazon → when product is available notify user, and youtube subscription

Date

↳ template Method

- Step wise algo converted into classes and then functions
- eg 1) MVC → frontend, backend, db
  2) Game development → make menu (have intialization etc)
     Display functions (play, top scorrer etc)
- inside fintionality is different but basic structure is same. in every game.