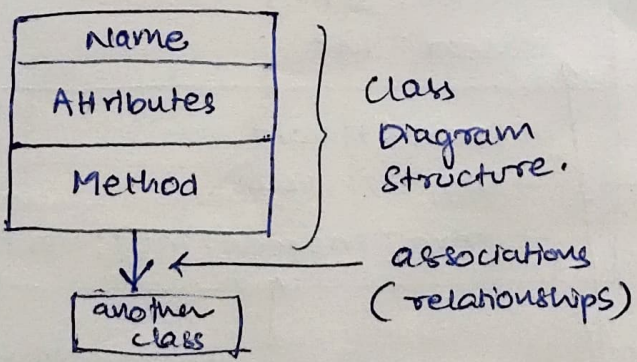
OOAD3: UML Class Diagrams, UML Interaction Diagrams, UML Activity Diagram and Modelling, Mapping Design to code, UML State Machine Diagram and Modelling, Test Driven Development and Agile concepts, Documenting Architecture, Case Studies.

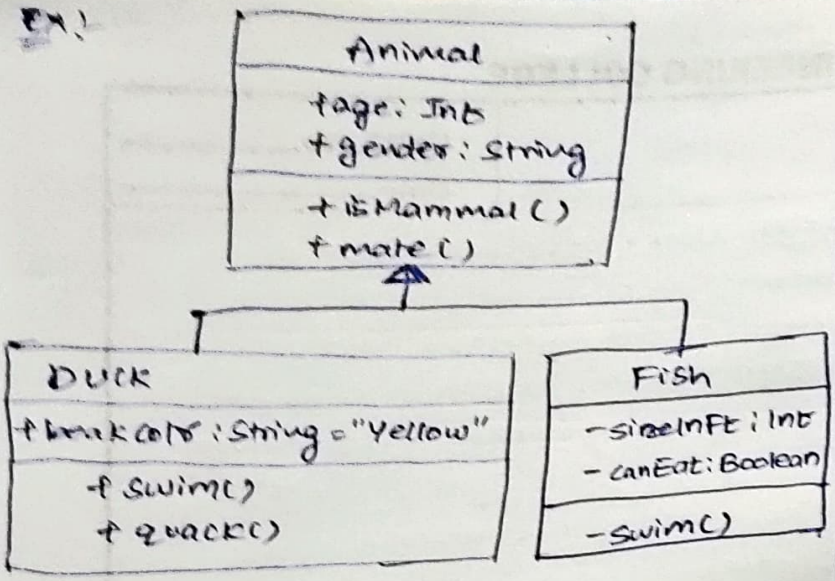
UML class diagram:- It is a visual tool that represents the structure of a ~~Symbol~~ system by showing its classes, attributes, methods, and the relationships between them.

* It helps everyone involved in a project - like developers and designers - understand how the system is organized and how it's components interact.
* Used in Software engineering to visually represent the structure and relationships of classes within a system
* i.e., used to construct and visualize object-oriented systems.



* provide's a high-level overview of a system's design, helping to communicate and document the structure of the software.

Eg:

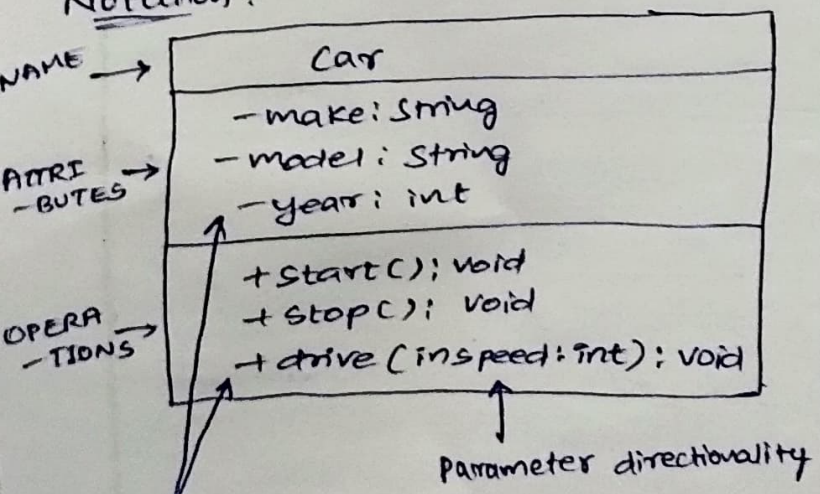


Class:- A class is a blueprint or template for creating objects. Objects are instances of classes,

⇒ Each defines a set of attributes (data members) and methods (functions) that the ~~projects~~ objects created from the class will possess.

⇒ The attributes represent the characteristics or properties of the object, while the methods define the behaviors / actions that object can perform.

Notation :-



Visibility notation.
(access level) ⇒ + for public
⇒ - for private
⇒ #t for protected
⇒ ~ for package (default).

Parameter Directionality:

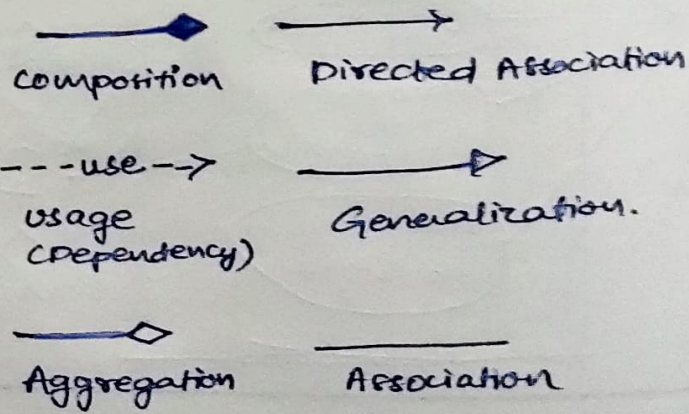
* It refers to the indication of the flow of information between classes through method parameters.
* It helps to specify whether a parameter is an input, an output, or both.
* This information is crucial for understanding how data is passed between objects during method calls.

Three main parameters:

* Input ("→")
  From ~~clas~~ calling object (client) → passed → To object (server)
* Output ("←").
  To ~~clas~~ calling object (client) ← passed back. From object (server)
* InOut (Input and output):
  ⇒ serves as both ↑ . "↔"

Relationships between classes

* They describe how classes are connected or interact with each other within a system.
* Each relationship serves a specific purpose.

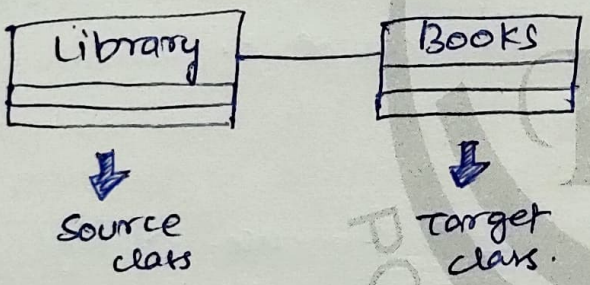


dependency -----> Realization ------▷

1. Association : (——)

* A bi-directional relationship btw two classes.
* It indicates that instances of one class are connected to instances of another class. (Arrows are optional).

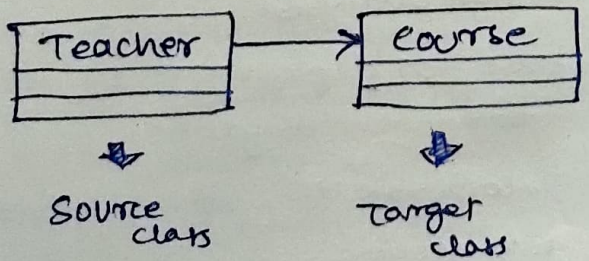
Ex:-



2. Directed Association; (——>)

* Association has a direction, indicating that one class is associated with another in a specific way.
* Used when the association has a specific flow or directionality

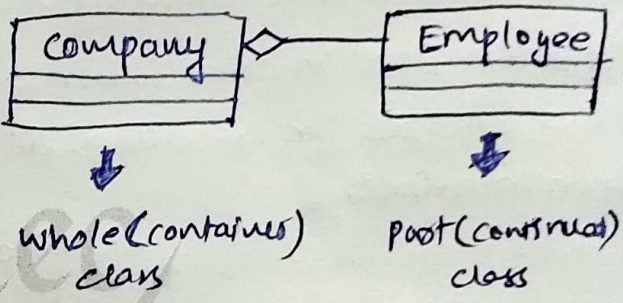
Ex:-



3. Aggregation : (——◇)

* Specialized form of association.
* It represents a "whole-part" relationship.
* It denotes a stronger relationship where one class (the whole) contains/composed of another class.
* The child class can exist independently of its parent.

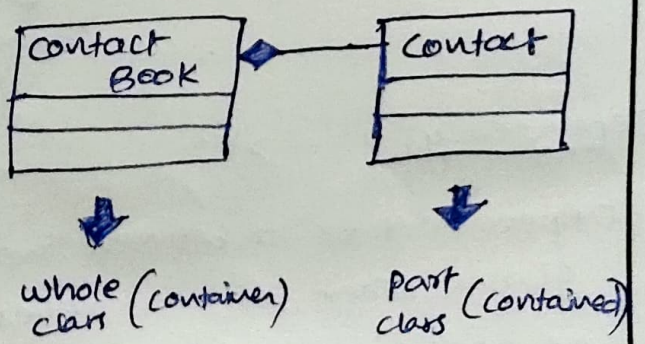
Ex:-



4. Composition; (——◆)

* Stronger form of Aggregation.
* Indicating a more significant ownership or dependency rship.
* The class part cannot exist independently of the whole class.

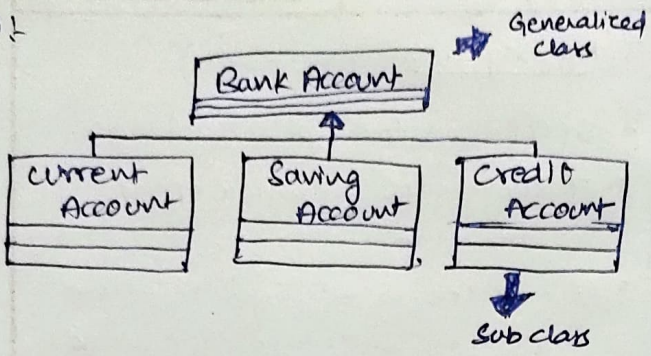
Ex:-



5. Generalization (Inheritance) (——▷):

* Inheritance "is-a" relationship.
* inherits the properties & behaviors.

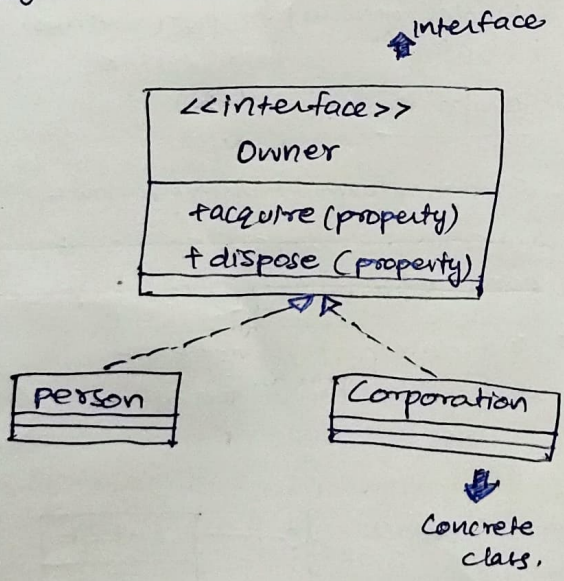
Ex:-



6. Realization (interface Implementation) (- - - - ▷):

* Indicates that a class implements the features of an interface.
* It is often used in cases where a class realizes the operations defined by an interface.

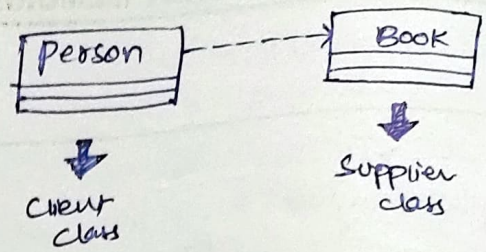
Ex:-



7. Dependency (- - - - >):

* Dependency exists between two classes when one class relies on another, but the relationship is not as strong as association/ inheritance.
* It represents a more loosely coupled connection between classes.

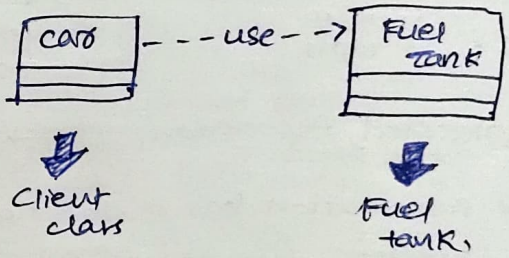
Ex:-



8. Usage (Dependency) (- - use - - >):

* It indicates that one class (client) utilizes/depends on another class (supplies) to perform certain tasks/access certain functionality.
* The client relies on the services provided by the supplier class but does not own (OO) create instances of it.

Ex:-



Purpose of class diagram:

* Depicts various aspects of OOPs concept appropriately.
* faster and efficient proper design analysis.
* base for deployment & component diagram.
* Incorporate forward and reverse Engineering.

How to draw class Diagram:

1. Identify classes.
2. List Attributes and Methods.
3. Identify Relationships.
4. Create Class Boxes.
5. Add Attributes and methods.
6. Draw Relationships.
7. Label Relationships.
8. Review and Refine.

UML Interaction Diagrams:

* The diagram, which can picture a control flow with nodes that can contain interaction diagrams which show how a set of fragments might be initiated in various scenario's.
* Interaction overview diagrams focus on the overview of the flow of control.

  nodes ⟶ interactions (sd)
  (or) use (ref).

Purpose of Interaction Diagrams:

* To visualize the interaction/ interactive behavior of the system.
* The solution is to use different types of models to capture the different aspects of the interaction:
  1. Sequence Diagram.
  2. Collaboration Diagrams.

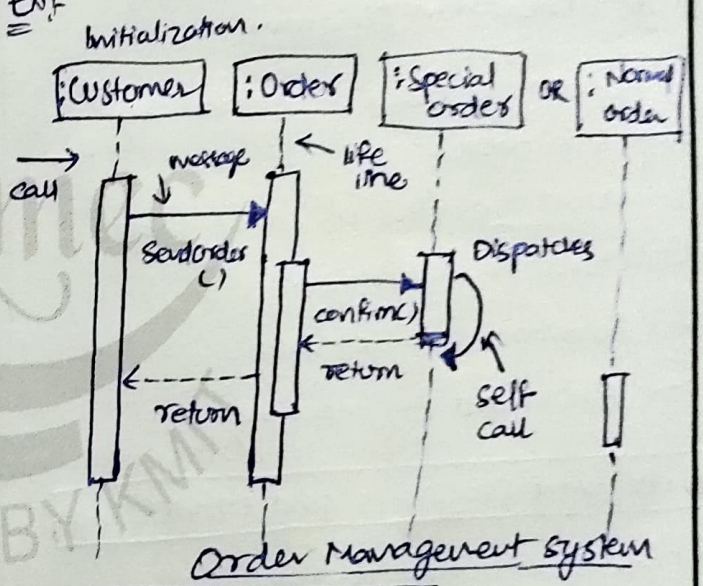
Sequence Diagram:

* The sequence diagram has four objects,

Ex:- Customer, Order, special order and Normal order

* It is important to understand the time sequence of message flows,
* The message is nothing but a method call of an object.

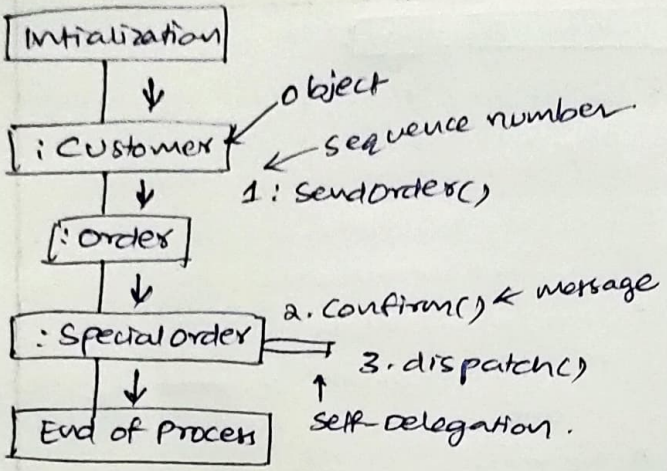
Ex:-



Collaboration Diagram:

* It shows the object organisation as seen in the following diagram.
* In CD, the method call sequence is indicated by some numbering technique.
* The number indicates how the methods are called one after another.
* Method calls are similar to sequence diagram. However, difference being the sequence

diagram does not describe the object organization, where as the collaboration diagram shows the object organization.



Order Management System

Uses of Interation Diagrams:

* To model the flow of control by time sequence.
* To model the flow of control by structural organization.
* For Forward Engineering.
* For Backward Engineering

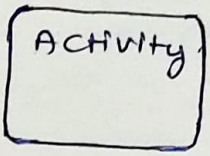
UML Activity Diagram and Modelling:

* It is to describe the dynamic aspects of the system.
* Activity diagram is basically a flow chart to represent the flow from one activity to another activity.
* The activity can be described as an operation of the system.
* The control flow is drawn from one operation to another.
* This flow can be sequential, branched, or concurrent.
* Activity diagrams deal with all type of flow control by using different elements such as fork, join etc...

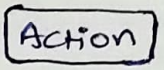
Purpose:

* Draw the activity flow of a system.
* Describe the sequence from one activity to another.
* Describe the parallel, branched and concurrent flow of the system.

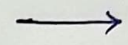
Notation:-



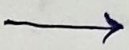
→ Activity is used to represent a set of actions.



→ A task to be performed.



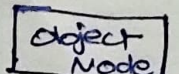
→ Control Flow shows the sequence of execution.



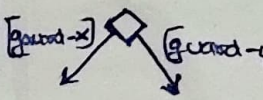
→ Object Flow shows the flow of an object from one activity / action to another.

→ Initial node portrays the beginning of a set of actions or activities.

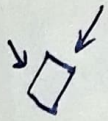
→ Activity Final Node stop all control flows and object flows in an activity / action.



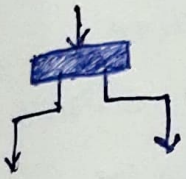
→ Object Node represent an object that is connected to a set of object flows.



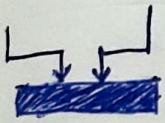
→ Decision node represent a test condition to ensure that the control flow / object flow only goes down one path.



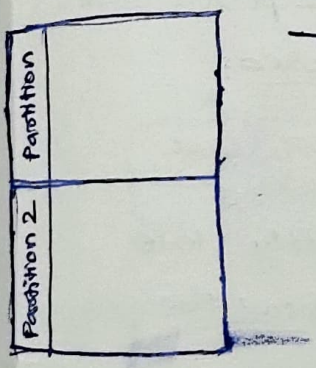
→ Merge node bring back together different decision paths that were created.



→ Fork node split behavior into a set of parallel/concurrent flow of activities/ actions.



→ Join Node bring back together.

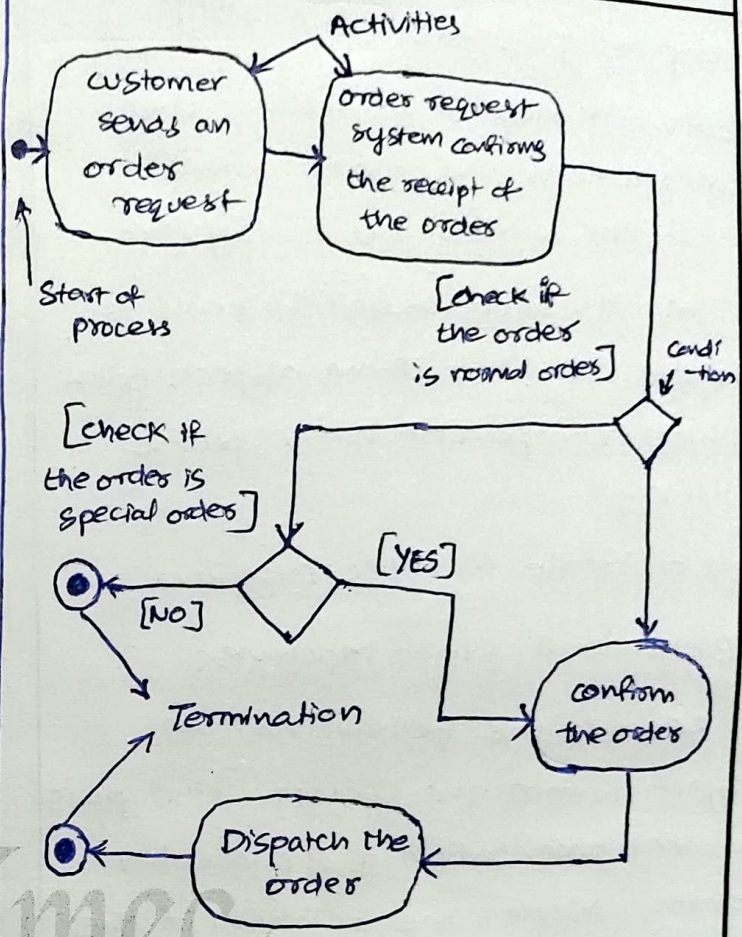


→ Swimlane & partition: A way to group activities performed by the same actor on an activity diagram (or) to group activity in a single thread.

* Before drawing an activity diagram, we should identify the elements;
  - ⇒ Activities ⇒ Association
  - ⇒ Conditions ⇒ constraints.

* Activity Diagram (case study):-

For the ordermanagement system the activity diagram is drawn with four main activities
- → Send order by the customers.
- → Receipt of the order.
- → confirm the orders.
- → Dispatch the order.



Order Management System

Uses of Activity Diagram:

* Modeling work flow by using activities.
* Modeling business requirements.
* High level understanding of the System's functionalities.
* Investigating business requirements at a latter stage.

UML State Machine Diagram:

* The state machine diagram is also called the state chart (or) state Transition diagram, which shows the orders of states undergone by an object within the system.

* It captures the software system's behavior.
* It models the behavior of a class, a subsystem, a package, and a complete system.
* Efficient way of modelling the interactions and collaborations in external entities and the system.
* It models event-based systems to handle the state of an object. Each object / component has a specific state.

Types of State Machine Diagram :-

1. Behavioral State Machine :

Records the behavior of an object within the system. It depicts an implementation of a particular entity. Models the behavior of the system.

2. Protocal state Machine :

It captures the behavior of the protocal. The protocal state machine depicts the change in the state of the protocal and parallel changes within the system.

Purpose of the SMD :-

* Visualizing the behavior of a system.
* Identifying potential issues.
* Documenting system behavior.
* Designing and testing system.

Notations :

- ● → initial state.
- State 1 → state box.
- ◇ → decision box.
- ◉ → Final state.
- ⊗ → Exit point.

Transition : change of control from one state to another state.

State box : Depicts the conditions or circumstances of a particular object of a class at a specific point of time.

Types of States :-

* Simple State : It does not constitute any substructure.
* composition state : It consists of nested states. It can be nested to any level.
* Sub machine state : The submachine state is semantically identical to the composite state, but it can be reused.

SMD case study :

Calender availability state diagram example shows the process by which

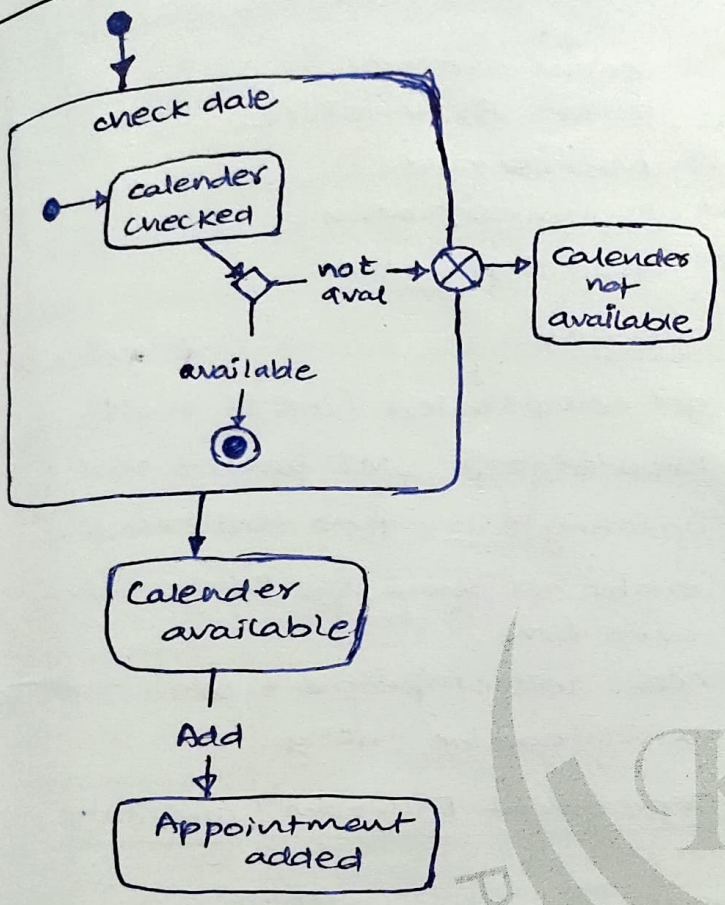
→ a person sets an appointment on their calender

→ In the "check date" composition state.

→ The system checks the calender for availability in a few different substates.

→ If the time is not available on the calender, the process will be escaped.

→ If the calender shows availability, however, the appointment will be added to the calender.



Uses of SMD :-

* Depicting event-driven objects in a reactive system.
* Showing the overall behavior of a state machine or the behavior of a related set of state machines.
* Illustrating use case scenarios in a business context.

UML Deployment Diagram :

* Deployment Diagram are used to visualize the topology of the physical component of a system, where the software components are deployed.
* used to describe the ~~state~~ Static deployment view of a system.
* It consists of nodes and their relationships.

purpose :

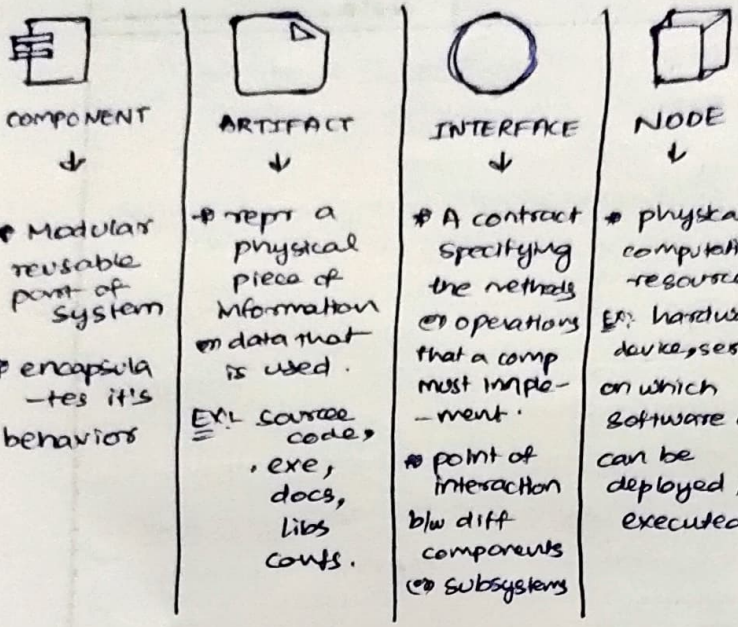
* visualize the hardware topology of a system.
* Describe the hardware components used to deploy software components.
* Describe the runtime processing nodes.
* Shows how the Software design turns into the actual physical system where software will run.
* It shows software components, hardware devices and how they connect with each other.

Key Elements :

* Nodes : physical hardware entities where software components are ~~developed~~ deployed.
  Ex : serves, routers etc ...
* Components : software modules / artefacts that are deployed onto nodes.
  Ex : .exe, .lib, .dll, .conf etc...
* Artifacts : physical files that are placed on node (Actual implementation of software components).
  Ex : .exe, scripts, .db.
* Dependencies : These show the relationships / connection b/w

Nodes & components.

* Associations : relationships b/w nodes & components.

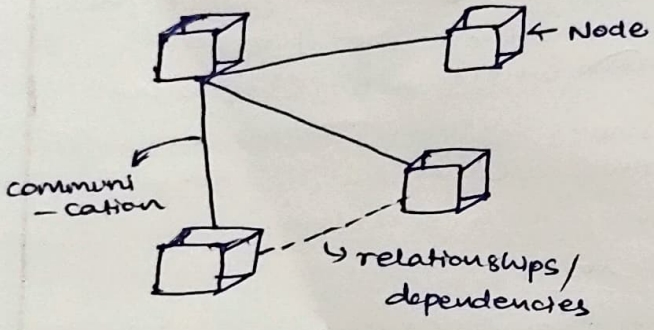


COMPONENT
* Modular reusable part of system
* encapsula -tes it's behavior

ARTIFACT
* repr a physical piece of information
* data that is used.
Ex: source code, .exe, docs, libs confs.

INTERFACE
* A contract specifying the methods & operations that a comp must imple- -ment.
* point of interaction b/w diff components & subsystems

NODE
* physical / computational resource
Ex: hardware device, servers, on which software comp can be deployed / executed.



Usecases:

* help set up on different devices.
* design the hardware supports software.
* make sure enough resources.
* shows dependencies.
* easier for teams.

Steps for creating:

1. Identify components.
2. Understand Relationships.
3. Gather Requirements.
4. Draw nodes & components.
5. Connect nodes & components.
6. Add details.
7. Documentation.

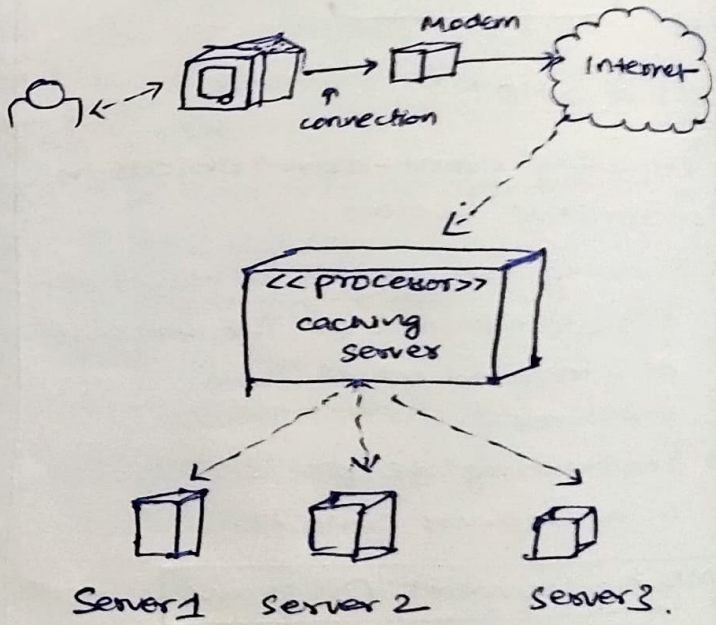
Deployment patterns

* Standardized methods for efficiently installing software on hardware.
* offer guidance for organizing and deploying software components.
  - => client - server.
  - => 3-Tier Architecture.
  - => Microservices.
  - => Containerization.
  - => cloud Deployment.

challenges

* get complicated (lots of parts)
* knowledge of UML symbols needed.
* updating takes time and effort.
* might not show how things change over time.
* needs lots of people to work together, which can be tricky.

* Deployment Diagram (case study):



UML Component Diagrams:

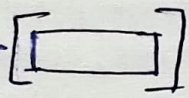
* Component Diagrams are different in terms of nature and behavior.
* Component diagrams are used to model the physical aspects of a system.

Propose of component Diagram:

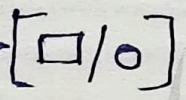
* visualize the components of a system.
* construct executables by using forward and reverse engineering.
* Describe the organization and relationships of the components.

Notations:

* Component :



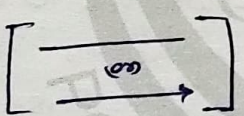
* port :



(Interaction btw a compone-nt and its environment).

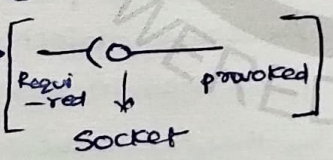
attached on the boundary of the component.

* Connector :



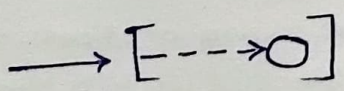
Connects two parts/components.

* Interface



Set of operations that a component can perform.

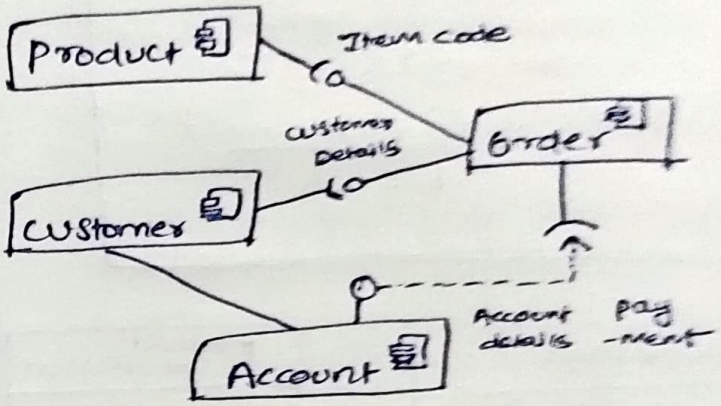
* Dependency



Uses :

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

Component diagram (Case study):



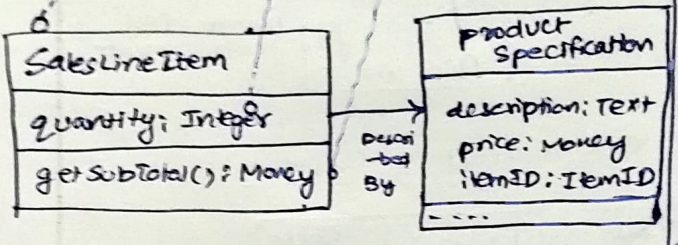
Online shopping system

Mapping Design to code:

* Mapping design to code involves taking the visual designs of a website or application and converting them into code that can be executed by a computer.
* Mapping design to code is a complex process that requires a combination of technical skills and creative thinking.
* It's important to work closely with designers and stakeholders throughout the process, and to constantly test and iterate on the code to ensure that it meets the needs.
* This process involves:
  - → understanding the design.
  - → creating a plan.
  - → writing a code.
  - → Testing and debugging.
  - → Deployment.

Defining a class with Methods and Attributes:

```
public class SalesLineItem {
    private int quantity;
    public SalesLineItem (ProductSpecification spec,
        int qty) { ---- }
    public Money getSubTotal()
        { --- }
}
```



SalesLineItem in Java

Mapping Concepts:

* Forward Engineering:
  - → It is a method of creating or making an application with the help of the given requirements.
  - → It is also renewation.
  - → It is requires high proficiency skills.
* Reverse Engineering:
  - → It is also k-a Backward Engg.
  - → It is process of forward Engg in reverse.
  - → The important info is collected from existing application.
* Refactoring:
  - → process of re-constructing the existing component code without changing its external behavior.