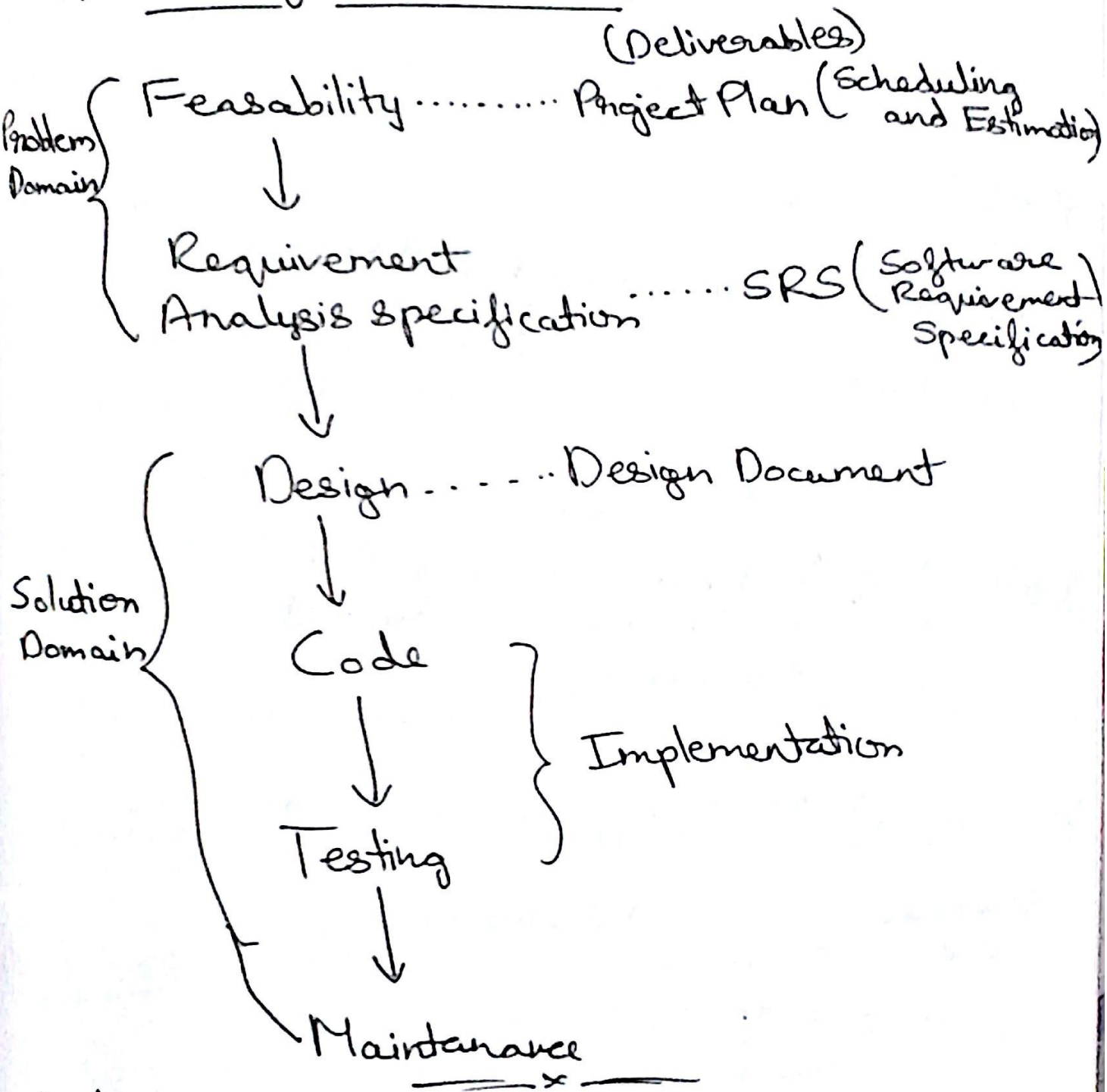
(Software Development Life Cycle)

1) Waterfall Method:



Note:

⇒ Test first development / test driven development is when testing happens before code.

⇒ In waterfall we don't go back to problem domain from solution

* Make Software such that:

1) Specifications are met
2) Within Budget (Cost)
3) Time (Quickly)

} Process.

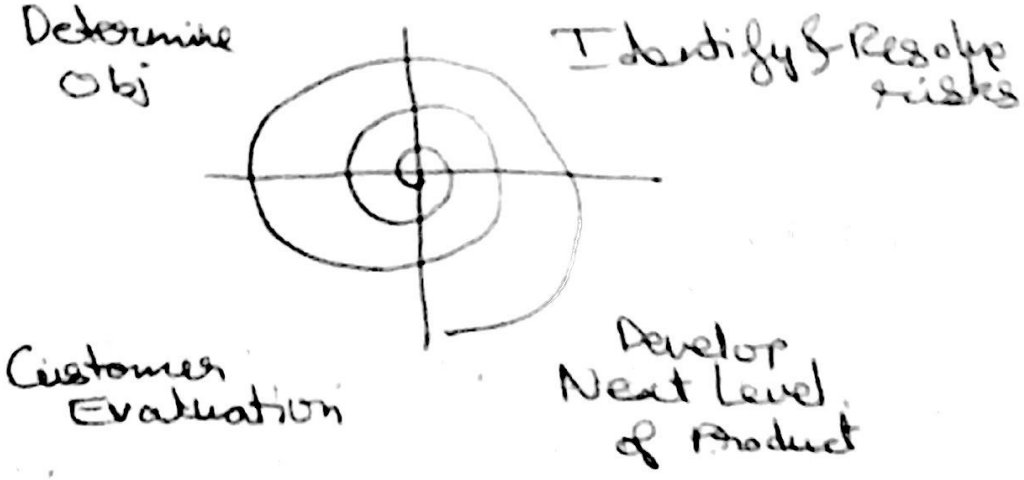
** Processes

a) Plan Driven → Waterfall Model
b) Agile → Incremental Iterative Development.

Note: PDCA = Plan Do Check Act
PDSA = Plan Do Study Act
(Agile Processes)

---

* Spiral Model (For Risk Projects)



Project Management;

1) Estimation
2) Scheduling
3) Tracking

Note; 1 week is 40 man hours

Note; Cone of Estimation (Ends at about +25%)

---

* Function points; Unit of Behaviour } Project cost depends on these.

---

*Scheduling;

1) Work Breakdown Structure (WBS)
2) Size estimation techniques
3) Dependency graph & network diagram
4) The actual schedule.

Note; (Types of WBS);

WBS → Process → Activity Oriented
WBS → Product → Entity Oriented
Process, Product } Hybrid

---

*Scheduling Tasks;

1) Gantt Chart } Not a lot of dependancies can be shown.
2) Network Diagrams } Dependancies are well defined
   ↓ (Pert Chart)

** Critical path: Path which takes max time
(This path should not become slower)

---

*Critical Paths; (In Pert Charts)

- The sum of hours in this path is your project time.

⇒ For other ~~paths~~ activities, we can set Earliest start, finish & Latest start, finish times.

——x——

Note; Slack time;

- Maximum allowable time delay for a non-critical activity.

——x——

eg;

} Pert Chart.

——x——

*Requirement Gathering & Analysis;

1) Functional → Requirements from stakeholders (function points)
2) Non-Functional
3) Business

} Types of requirements.

Quality requirements.

——x——

Note; Function points are nothing but features.

——x——

*Note; Requirements should be

a) Consistant (No conflicts b/w requirements)
b) Correct
c) Unambiguous.

Q) The product shall switch b/w displaying and hiding non printing characters instantaniously.

✓ ⇒ instantaniously → ambiguous

✗ ⇒ ~~Incorrect~~ → displaying non-printing characters
Incorrect

Q) The product shall provide status messages at regular intervals not less than every 60 seconds.

Rewrite) The app shall provide status messages to our phone ~~~~ every ~~~~ minute.

---

*Use Case; (Description) (Also known as *User Story)

- Stakeholders interacting with system are called 'Actors'. (Provide inputs)
- Use Case is where the product can be applied/used in real life.
- Features map to use cases.

Sequence of actions a system can perform interacting with actors of the system.

---

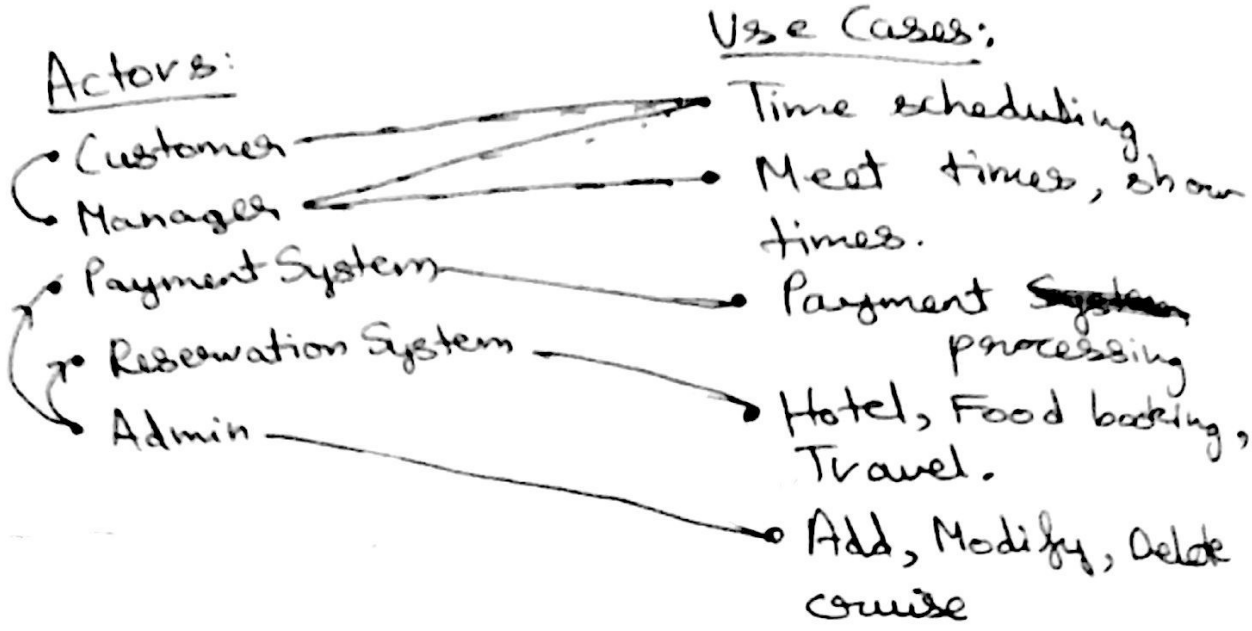
*Use Case Modelling; (UML)

- Primary actors on left side (), supporting on right side.
- Use cases : (use case)
- System/subsystem : 

Note; UML — Unified Modelling Language
OMG — Object Management Group.

—x—

ex: Use case diagram for ~~Caribbean~~ Caribbean fashion tours. (Look at Slides)



—x—

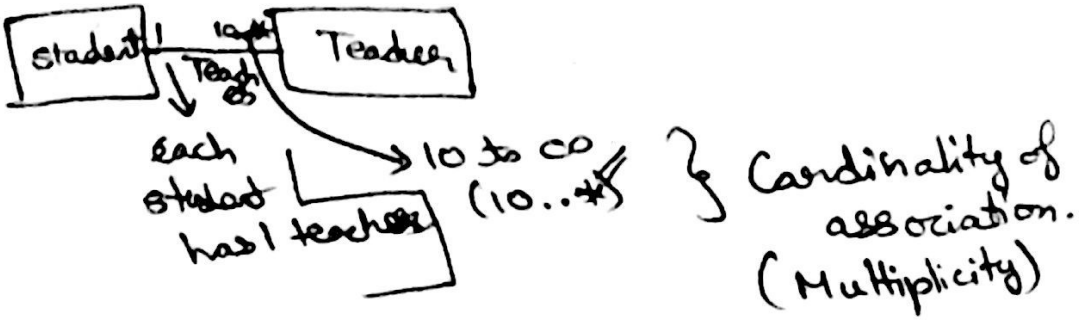
*Note: Coupling & Cohesion in classes-
(Class should not do everything).

—x—

* Class Diagrams (UML); (Move to solution stage from question stage)

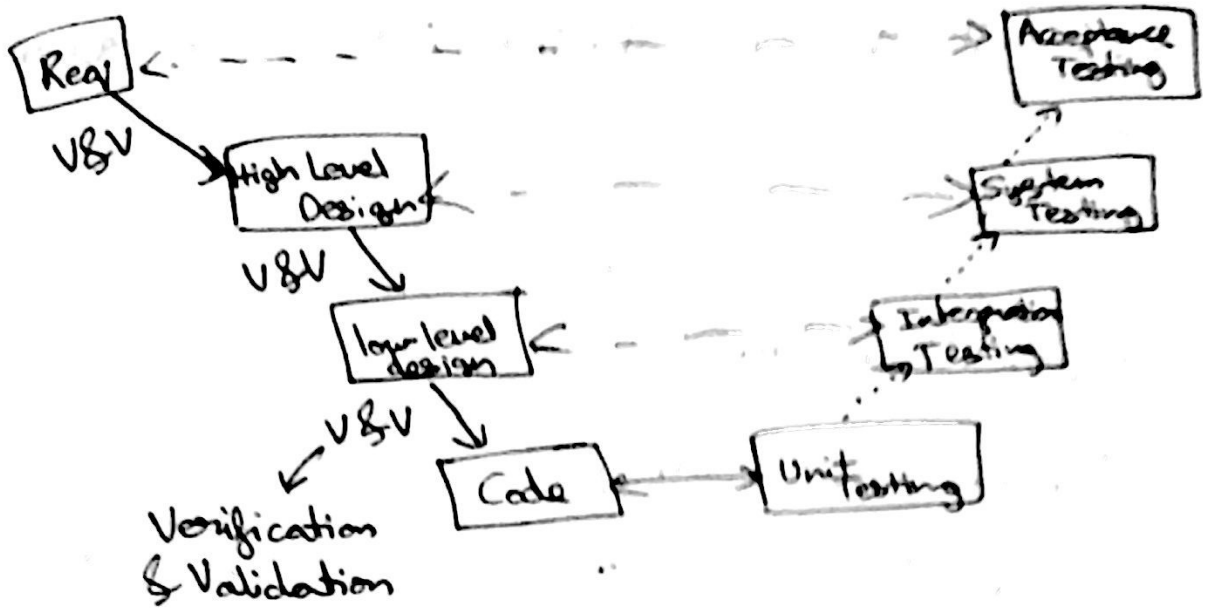
- Association b/w 2 classes is just a reference of one class from another.

ex:



—x—

Testing: (V-Model)

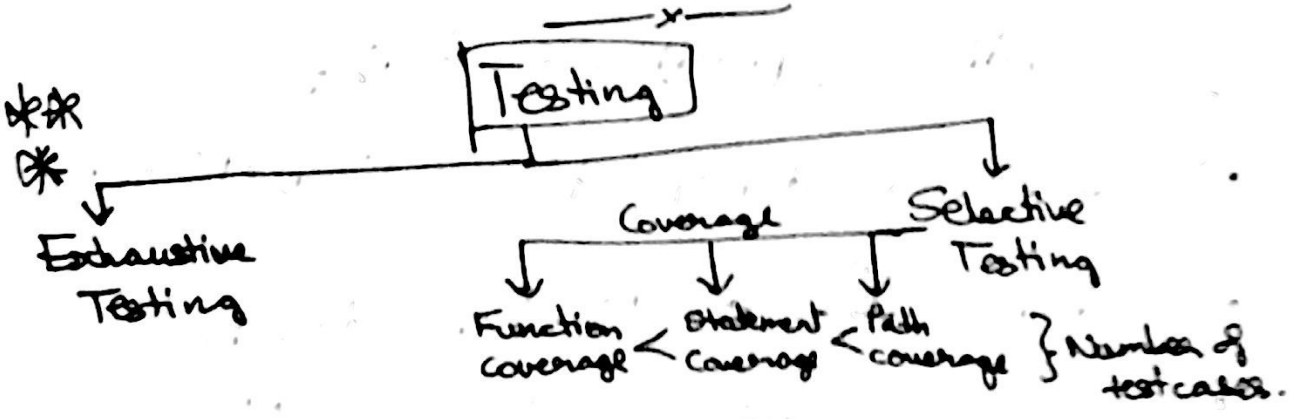


---

* **Test Case;**
  - Triplet consisting of input, steps to be performed, and the output (expected outcome).

  <input, seq of steps, expected outcome>

⇒ **Note;** Exhaustive Testing is not feasable



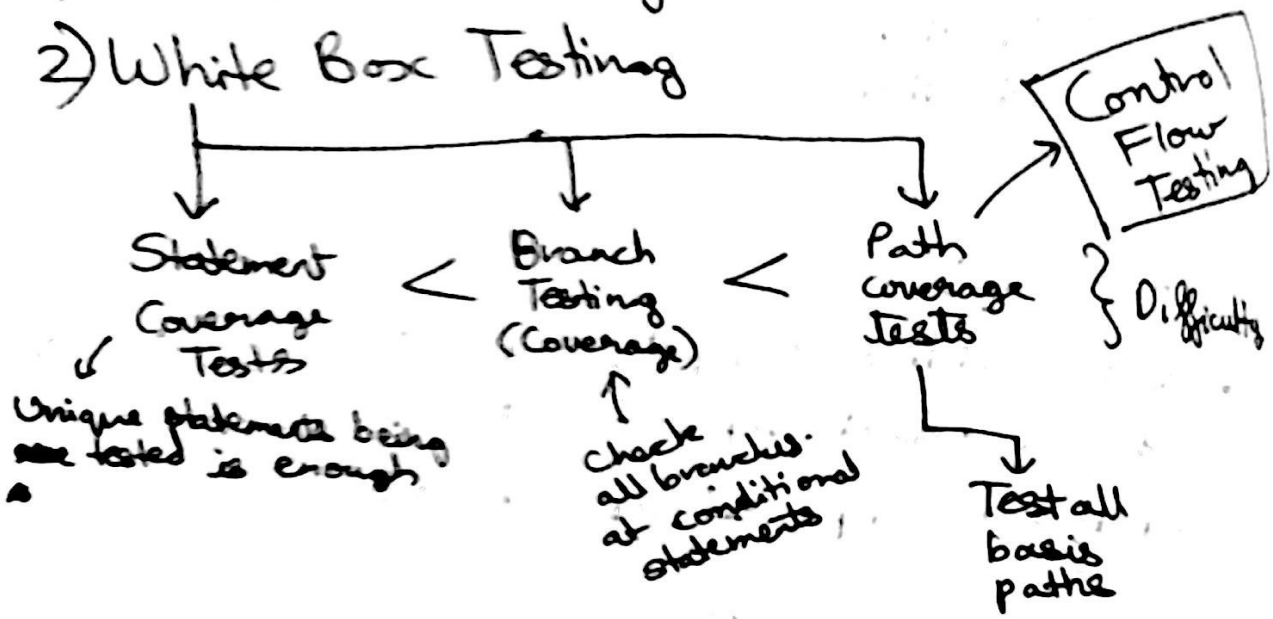
*Equivalence Classes;

- Divide input into groups which you think are treated equivalently by most algorithms. (These are equivalence classes)

—— x ——

Types of Testing;

1) Black Box Testing
2) White Box Testing



—— x ——

Cyclomatic complexity number;

- ~~Number of closed loops +~~ (V(G) = N - E + 2)
- Number of ~~nodes~~ regions — Number of edges + 2 = C.C.N
- ~~Number of conditions + 1 = C.C.N~~

—— x ——

* Cyclomatic Complexity Number;

- Number of edges − Number of nodes + 2 = C.C.N

$\Rightarrow V(G) = CCN = E - N + 2$

(Smaller complexity ⇒ Better programs)

— x —

* Testing (Whitebox)

** → Data Flow Testing

→ DEF-USE Coverage (Definition-Usage)
(DU pairs)

```
eg;1) if(a<0) {
      2)   return 0}
      3) x=0
      4) c=a
      5) while(c>0){
      6) x = x+a
      7) c = c-1 }
      8) return x
```

LHS has variable
def(x) = 3,6
def(c) = 4,7

use(x) = 6,8
use(c) = 5,7 → RHS has variable

- DU paths for c; → Independent of other values

{4,5}, {4,5,6,7}, {7,5}, {7,5,6,7} ← This can be ignored

- DU paths for x; (Start from definition)

~~{3,4,5,6}~~ {3,4,5,6}, {3,4,5,8},

{6,7,5,8}, ~~{6,7,5,6}~~

{6,7,5,6} ← Cannot be

* (Start from definition & end at usage)

— x —

Black-Box Testing;

- Top-Down or Bottom-Up integration.
  - Top-Down → Have driver (Need Stubs)
  - Bottom-Up → Have stubs (Need Driver)

⇒ Do not do Big Bang integration!
(Do incremental).

—x—

* Cohesion: A module should be able to accomplish its purpose by itself (Instead of depending on other classes → coupling).

—x—

Note: Structural → Class Diagrams
Behavioral → Sequence Diagrams

Design

1) Logical design
2) UI design
3) Protocol design
4) Algorithmic design

3 C's (For Design)
↓
Components
+
Connectors
+
Constraints

Note; Cosmetics vs Bare Bones

Note; Coupling ↓ & Cohesion ↑ } For Ideal design

---

*Architecture vs Design
↓
Requirements

Architecture
↓
Constraints
→ Logical
→ Implementational

Patterns; → In Models.

1) Layered Patterns:
2) MVC Pattern.

---

*UML Diagrams

→ User's View
→ Structural View
→ Behavioural View
→ Implementation View
→ Environmental View

(Structure)
* Static Models vs (Behaviour) Dynamic Models

- Class Diagrams used
- Describes structure of the system

- Describes behaviour of the system.
- Statechart, Sequence & Collaboration diagrams.

positive abstract methods

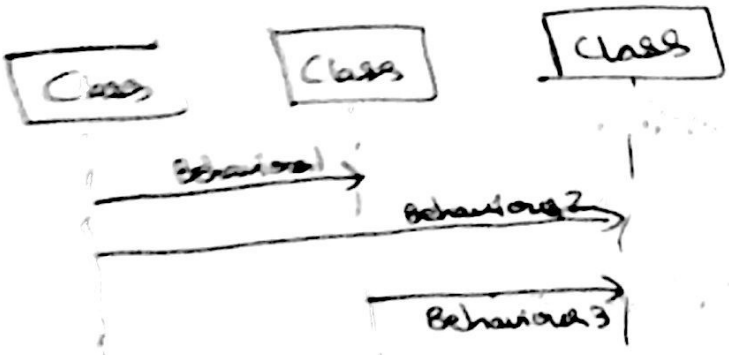
* Interface → <<interface>> → In UML Diagrams

- Interfaces for classes are just the function definitions that must be present in every subclass.

---

* Sequence Diagrams:

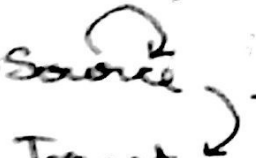
- Inter-object behaviours (Between Classes)



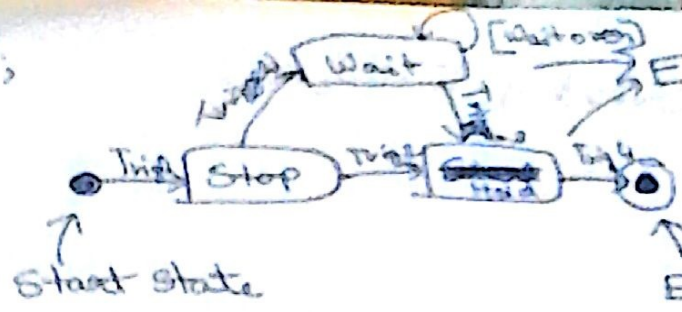
---

* State Diagram:

- Intra-object behaviours. (within Class)



ex:



⇒ [...] is a guard condition which allows us to get to the next states

---

Note; Cyber physical systems → systems which interact with people.

END of Mid 2

Note;

* Publish-Subscribe protocol (Subject & Observers)
(Observer pattern)

↑ Observe state changes to subject.

---

* Design Patterns
  - → Creational (ex: Singleton)
  - → Structural (ex: Adaptor) → Wrappers (allows different things together)
  - → Behavioural (ex: Observer, Command patterns) → Delay is downside
    - Push/Pull functions.

---

* • Pattern → problem, solution pair for problems in literature, its a generic solution.
* • Context tells us how & where we apply a certain pattern.

---

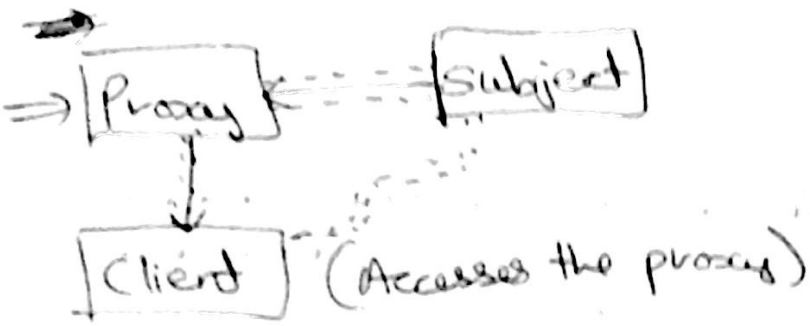
* Note; Adaptee & Target } Adaptor pattern.

---

* Adapter Pattern;
  - Class Adapter → Inheriting
  - Object Adapter → Links objects } (1)
  - Target object → (2)
  - Adaptee object → (3)

* Proxy pattern: (Subset of Interface)

- Cache Proxy
- Network proxy
- Remote proxy



⇒ Adapter has different interface from original whereas proxy is a copy of the interface

——x——

* Facade Pattern:

- Simplifying interaction b/w Clients & subsystems.

Note: Mediator pattern

——x——

* Note: Design should not be open for modification (Everytime we have a new subject) (within classes)

——x——

* Composite Pattern: (Similar to part-whole relations)

- (Leaf) Primitives & Composites

Belongs to

Note:

——x——

- Write Test cases → Refactor → Test

——x——

H.W: Library System: (Existing Design)

- Find potential problem areas. (Make Class Diagram after fixes)

———x———

Note: Refactoring → Changing structure to make it easier to understand

———x———

Good code ⇒

Classes → 6 to 8 methods, Method → 20 to 25 lines

———x———

* Maintanance;

- Perfective
- Corrective
- Adaptive
- Preventive

———x———

Note: Reengineering = Reverse Engineering + Forward Engineering

———x———

Anti-Patterns ; (Bad patterns)

1) BLOB → God Classes lead to blob patterns → low cohesion
2) Dead Code
3) Spaghetti-code → Cut & Pasting code can lead to this
4) Analysis-Paralysis pattern → Planned too much but couldn't get it done.

⇒ Refactor these so the anti-patterns are gone

Note: Kano Model → For new products

—x—