MOD -1

SOFTWARE ENGINEERING

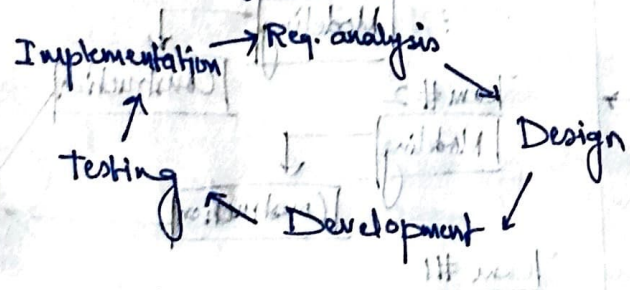
Software Engineering - branch of engineering that ~~defines~~ deals with the establishment and use of sound engineering principles to obtain sound economical softwares that is reliable and works effectively on machines.

4 Ps -
1. Product
2. Process
3. Project
4. People

System Engineering processes -

Req. Dev. → System design → System dev. → Sys. integration → Sys. installation → Sys. maintainance → Phasing out

Life cycle of Software Dev.



Software Development Process Models

1. Code - & - fix model
- Similar to trial & error
- Code the software & fix/modify errors
- Used for very small projects
- V. Short delivery time
- SE restricted to management of prog.
- No way to keep track of req., business logic & design

2. Waterfall Model
- Clear req.
- No ambiguous req.
- Well understood tech.
- Short project & low cost
- Min. risk

Feasibility study → Req. analysis → Design → Coding → Testing → Maintainance

Advantages:
- Simple to understand, easy to use & manage
- Small & low budgets, well understood req.
- Easy to arrange tasks
- Process & results well documented.
- Documents produced at each stage
- Identical & compliant to other engg. models.

Disadvantages:
- No working software produced until late
- High risk & uncertainity
- Difficult to measure progress stagewise
- Poor model for long ongoing projects
- All req. must be gathered at start
- Phase wise focus
- Restrictions end up process postponement
- Design is inflexible
- Postponement makes maintainance big phase

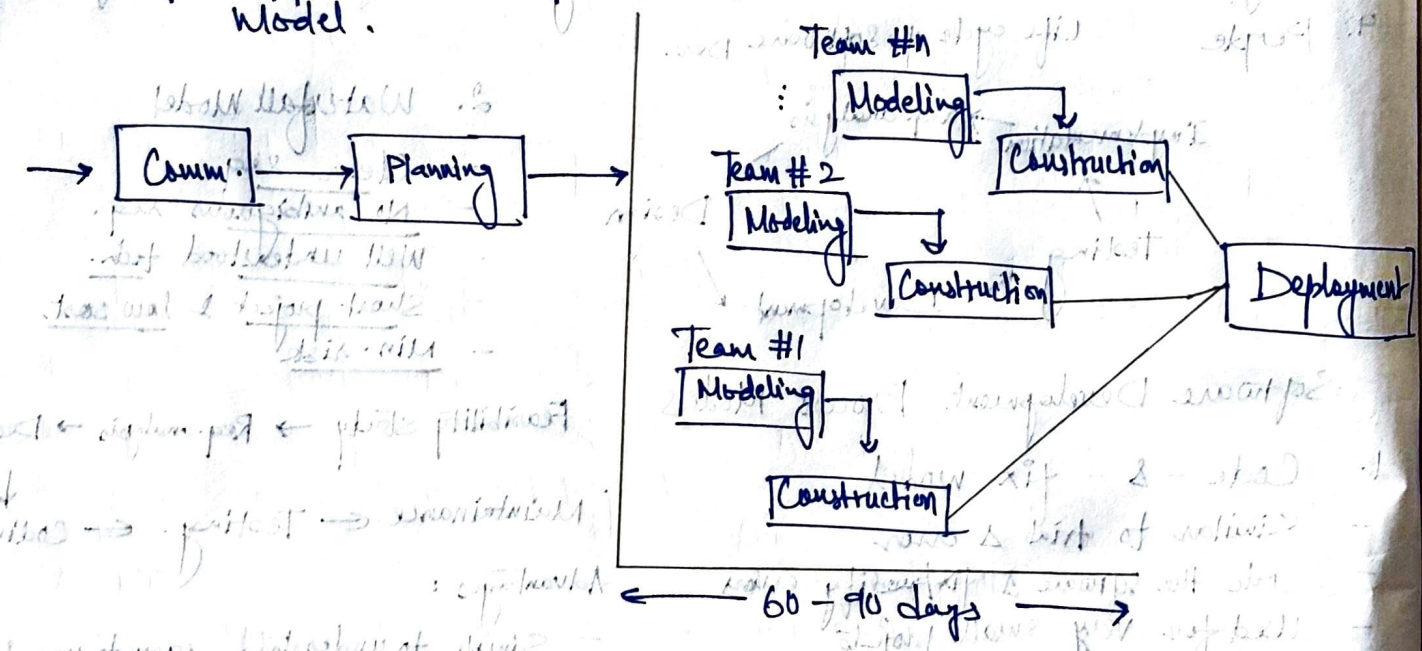
Types of Maintainance

Corrective - correcting errors not discovered during product dev. phases

Perfective - improve implementation, enhance functionalities of system.

Adaptive - ~~maintai~~ porting software to new environments.

3. Rapid Application Development (RAD) Model.



RAD

- Short duration delivery
- Multiple teams working parallelly on diff. depts like modeling, module/submodule construction.
- Custom. product for ~~one~~ 1-2 customers only

Features -

- Less time, cost to develop
- Flexibility to incorporate changes
- Short term plans, heavy usage of existing code.
- Split into indep. modules.

Minus points

- Inconsistent (maybe)
- Difficult integration

Methodology of RAD

1. Plans made for one increment at a time. Time planned for each it. is called time box.
2. Each it/inc. enhances implemented functionalit of the app.
3. ~~Ex~~ During each iteration,
   a. Quick-& dirty prototype software for some functionality is developed
   b. Customer feedback after evaluation
   c. Prototype refined.

Facilitating faster development

- Fast creation of working prototypes through use of specialized tools. Support features like -
  1. Visual dev. style
  2. Reusable components
  3. Standard APIs.

4. Iterative Model

- Software dev. processes repeated to improve performance, fix bugs
- Focus not on adding functionalities

Req. Outline → Split into features → Design → Develop increment → Validate → Integrate → Validate system → Final prod.

5. Incremental model (Iterative)

- Waterfall is single release, iterative can give multiple releases.
- First increment → core functionality
- Successive increments → add/fix functionalities
- Final increment → complete product.

Each iteration is like a short mini project with separate lifecycle.

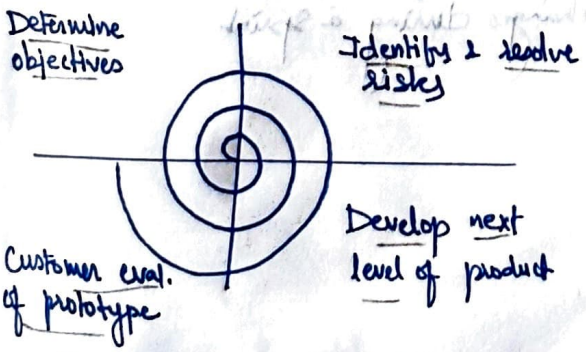
6. Evolutionary models

Prototyping

Communication → Quick plan → Modeling quick design → Prototype construction → Deployment delivery & feedback → Communication

- Evolutionary prototype model could be used for understanding requirements, functioning & understanding new tech.
- Code/product may not be final prod.

Boehm's spiral model



- Documents & prod. available after each iteration
- Improve & add functionalities after each cycle
- Requires all types of human resources at regular frequency

7. Agile Model

- Agile - Easily movable, light, nimble, active software processes.
- Primarily designed to help projects to adapt to changing requests.
- In agile model, requirements decomposed into many small incremental parts that can be developed over 1-4 wks each.

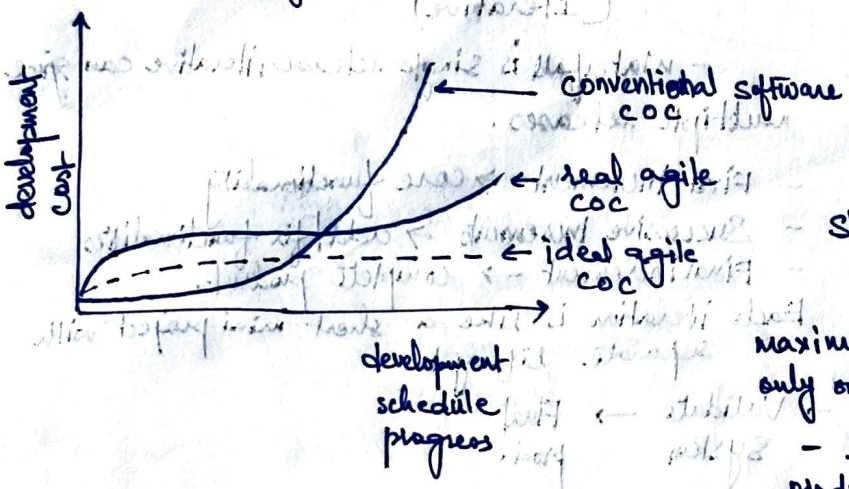
Features

- Effective response to change
- Effective communication from all stakeholders
- Involving customer into the team
- Working as a team
- Flexible planning keeping uncertainity in mind
- Organizing team in control of work performed
- Emphasis on incremental delivery to get working software delivered to customer quickly.
- Rapid delivery & incremental
- Development guidelines stress delivery over analysis & design, active & continuous communication between customers & developers.

Benefits

- Relatively easy to accomodate change
- Minimal costs of work done
- Well designed agile process may flatten cost of change curve by coupling incremental delivery with agile practices.

Cost of Change in Agile Approach



Agile Process

- Customer description – what is required, Scenarios, assumptions
- Plans are short-lived, req. will persist, priorities will change etc.
- Develops software iteratively with heavy emphasis on construction activities
- Design & construction interleaved
- Design models are proven as created
- Analysis, design, construction & testing not predictable.
- Adaptive
- Delivers multiple software increments (operational prototype / part of OS) for customer feedback.

Agile (Gate Smashers)
- Widely used in many companies now
- Agile → move quickly

Large projects

Small chunks (Iterations) → Release → feedback → enhance → re-release

Adv: Freq. delivery, face to face comm. with client, changes done quickly, less time, flexible teamwork, no hierarchy

Disadv: Less documentation, maintainance problem.

Scrum Model

- Comes under Agile, very popular
- Lightweight, iterative & incremental framework.
- Breaks down development phases into stages / cycles called 'sprints'.
- Dev. time for each sprint is maximized & dedicated, thereby managing only one sprint at a time.
- Scrum Team → scrum master & product owner with constant comm on daily basis.
- Backlog: place where basic req., designs, etc. are mentioned
- Daily Scrum: daily short meeting with developers & stakeholders to discuss about features, work progress & changes
- Scrum master: person who is monitoring the scrum, comm with stakeholders
- Product owner: client

Adv:
- Freedom, adaptation
- High quality, low cost product
- Reduce dev. time upto 40%.
- Customer satisfaction
- Reviewing current sprint before moving to next one.
- Flexibility

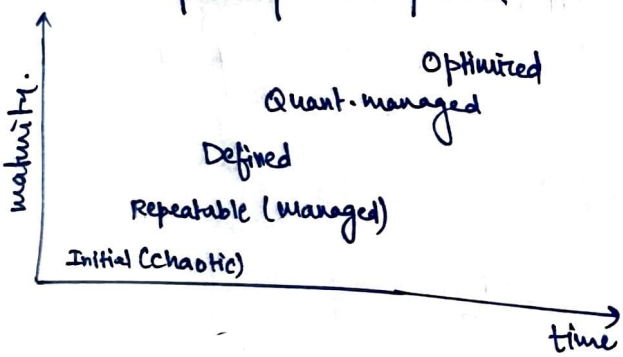
Disadv:
- More efficient for small size
- No changes during a sprint

Comparison of various SDLC Models

Classical Waterfall	Iterative waterfall	RAD Model	Spiral model	Agile model
Basic, Rigid, Inflexible, not suitable for real proj., sequential, (no parallelism) req. lock	Basic, well-understood problem & tech, req. lock	Time & cost constraint, user interaction at all levels, Code reusability	Risk evaluation, Not for small proj., No early lock on req., less experience dev. can work!	Flexible, advanced, parallelism, process divided into sprints.

2016

CMM – Capability Maturity Model



Agile Methodologies

- Scrum
- XP
- Crystal
- Adaptive Software Dev. (ASD)
- Dynamic Systems Dev. Method (DSDM)
- Feature Driven Dev (FDD)

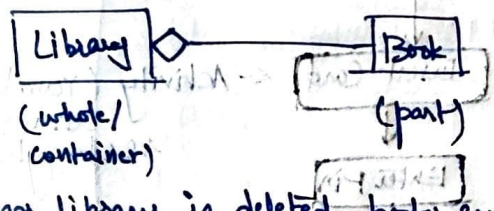
UML

Software Engineering

Aggregation

- A "whole-part" relationship where the part can exist independently of the whole
- Signifies a weaker relationship compared to composition.

Ex:



- If class library is deleted, books can still exist independently

Composition

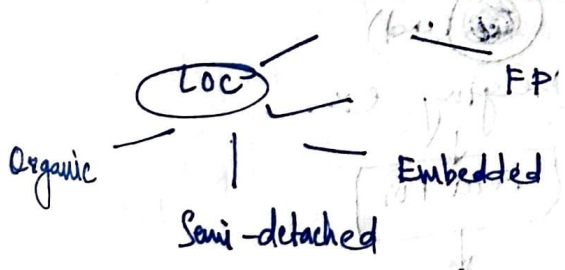
- A strong "whole-part" relationship where the part cannot exist independently of the whole.
- The lifecycle of the part is tied to the lifecycle of the whole.

Ex:

- If a house is destroyed, the room also would be destroyed.

Project Estimation – COCOMO – MOD-1

(Constructive Cost Estimation Model)



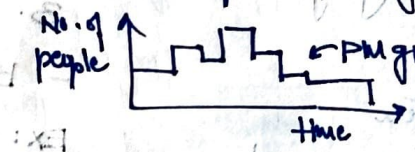
Organic:
- 2 - 50 KLOC
- Small size, Little innovation, lenient deadline, familiar & inhouse

Semi-detached:
- 50 - 300 KLOC
- Medium size, medium innovation, medium deadlines, medium dev. env.

Embedded:
- > 300 KLOC, large project
- Significant innov., tight deadlines, cust interface/hardware

Effort estimation = person months (PM)

1 PM = effort put by a 1 person in 1 month



1. Basic COCOMO Model

$$\text{Effort (PM)} = a \times (KLOC)^b$$

$$\text{Duration (months)} = c \times (\text{Effort})^d$$

$$\text{Person req. (nearest whole no.) / Avg staff size} = \frac{\text{Effort}}{\text{Duration}}$$

	Org.	Semi	Emb
a	2.4	3.0	3.6
b	1.05	1.12	1.2
c	2.5	2.5	2.5
d	0.38	0.35	0.32

Organic (2-50 KLOC): $E = 2.4 \times (KLOC)^{1.05}$, $T = 2.5 \times (E)^{0.38}$

Semi detached (50-300 KLOC): $E = 3.0 \times (KLOC)^{1.12}$, $T = 2.5 \times (E)^{0.35}$

Embedded (> 300 KLOC): $E = 3.6 \times (KLOC)^{1.20}$, $T = 2.5 \times (E)^{0.32}$

$$\text{Productivity} = \frac{KLOC}{E} \quad (KLOC/PM)$$

2. Intermediate COCOMO Model / COCOMO II

$$\text{Effort} = 2.94 \times EAF \times (KLOC)^b$$

$$\text{where } b = 0.91 + \frac{\text{(sum of scale factors)}}{100}$$

$$\text{Duration} = c \times (\text{Effort})^d \rightarrow c, d \text{ from basic COCOMO}$$

$$\text{Person req} = \frac{\text{Effort}}{\text{Duration}}$$

EAF: Early adj. factors – multiplication of seven factors on scale of very low, low, normal, high, very high and extra.

(select val from very low, low, etc. per ~~parameter~~ factor & multiply all).

1. Reliability & complexity
2. Requirement reusability
3. Platform
4. Personnel Attributes
5. Prev. Exp.
6. Team support
7. Req. dev. schedule

Scale Factors:

1. Precedentness / Prev. Exp
2. Dev. Flexibility
3. Risk Resolution
4. Team Cohesion
5. Process Maturity

Add all for Scale factor

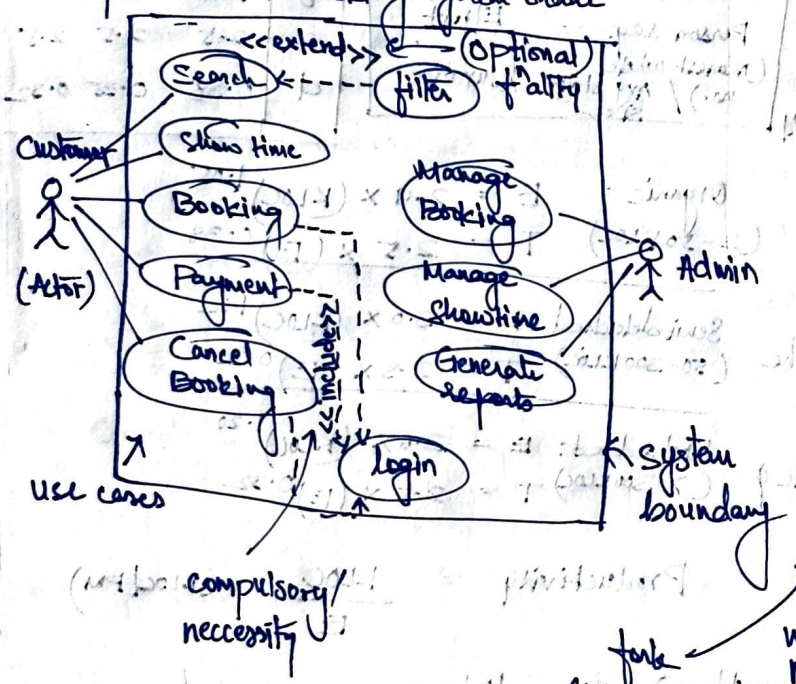
Software Engineering (SE)

Unified Modeling Language (UML)

1. Use Case Diagram

- Behaviour of system
- Illustrate functional requirements, interaction with external agents
- High level view of system without going in ~~functional~~ implementation details.

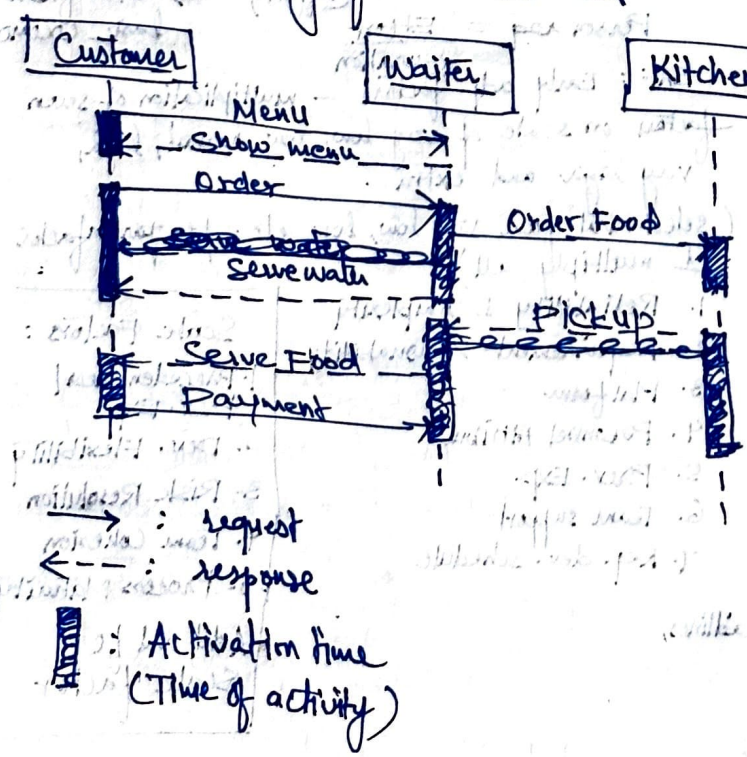
Example: Ticket booking system online



2. Sequence Diagram in UML

- Behavioural, dynamic aspects of system
- Multiple objects, timely order (chronological)

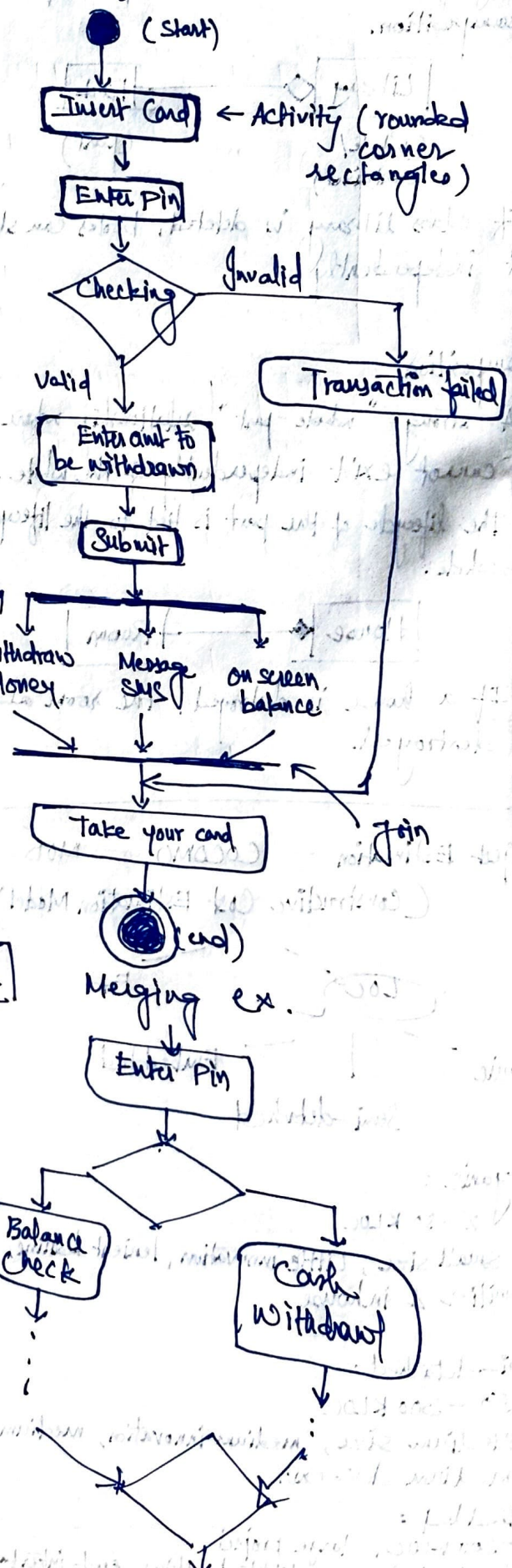
Example: Ordering food in restaurant



3. Activity Diagram in UML

- Behavioural, flow of activities

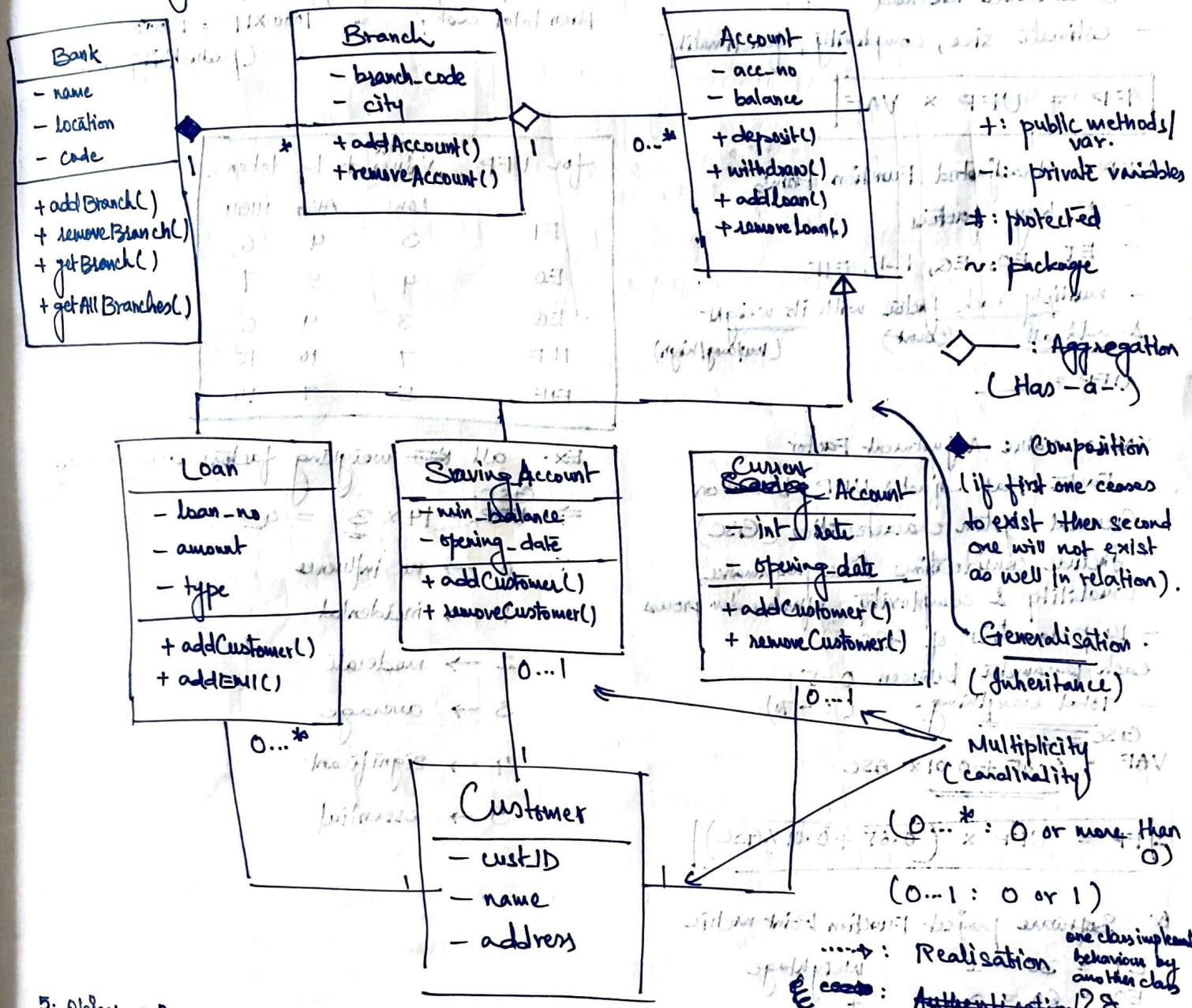
EX: Withdrawing cash from ATM. Kiosk



4. Class diagram

- Structured diagram, OOPS concepts
- Static view

Ex. Banking system

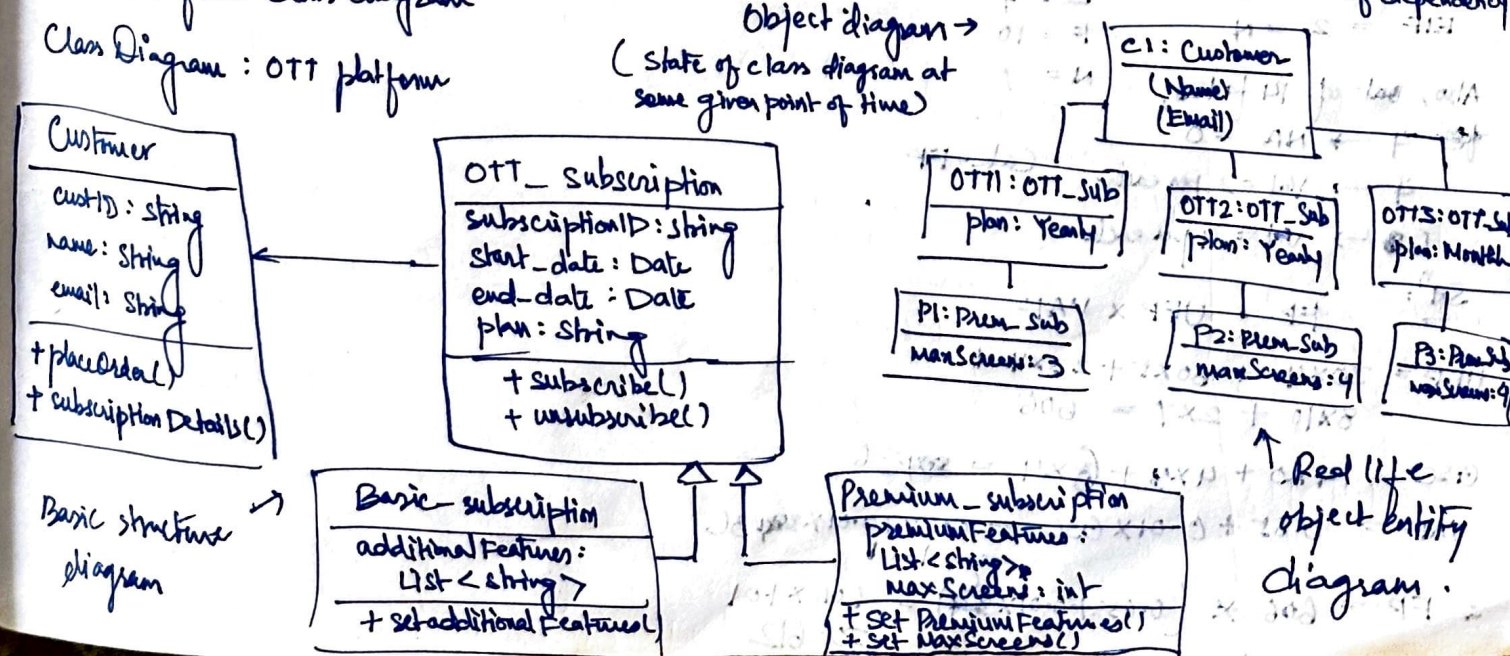


5. Object Diagram

- Derived from class diagram

Class Diagram : OTT platform

Object diagram → (state of class diagram at some given point of time)



Basic structure diagram

Real life object entity diagram.

Module - 1

Function Point Analysis (FPA)

- standardized method
- estimate size, complexity, functionality

$$\boxed{FP = UFP \times VAF}$$

UFP : Unadjusted Function Points
- 5 key factors
- EI, EO, EQ, ILF, EIF
- multiply each factor with its weight (low/avg/high) & add all. (count)

UFP = ___

VAF : Value Adjustment Factor
- Factor that adjusts UFP based on General system characteristics (GSC)
- Include considerations like performance, usability & complexity, influence dev. process
- 14 parameters of GSC
each parameter between 0-5
- Total everything. (0-70)

GSC = ___

$VAF = 0.65 + 0.01 \times GSC$

$$\therefore \boxed{FP = UFP \times (0.65 + 0.01 \times GSC)}$$

Q. Software project Function Point metric

EI = 30 = I
EO = 60 = O
EQ = 23 = E
ILF = 8 = F
EIF = 2 = N

weightage
I = 4
O = 5
E = 4
F = 10
N = 7

Also, out of 14 factors,
4 → NA = 0
4 → Val = 3 per each
6 → Val = 4 per each

Calc. FP

Sol^n:

$FP = UFP \times VAF$

$UFP = 30 \times 4 + 60 \times 5 + 23 \times 4 + 8 \times 10 + 2 \times 7 = 606$

$GSC = 4 \times 0 + 4 \times 3 + 6 \times 4 = 36$

$VAF = 0.65 + 0.01 \times GSC = 0.65 + 0.01 \times 36 = 1.01$

$\therefore FP = 606 \times (0.65 + 0.01 \times 36) = 606 \times 1.01 \approx 612$

Lets suppose efforts = 36 PM

what is productivity? $= \frac{FP}{PM} = \frac{612}{36} = 17$

Lets suppose proj. cost per month = 1000

then total cost? = $1000 \times 17 = 17000$ (productivity cost)

for UFP weight Values to be taken

	LOW	AVG	HIGH
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

Ex. all weighing factors are average

GSC

$\Rightarrow 14 \times 3 = 42$

0 → no influence
1 → incidental
2 → moderate
3 → average
4 → significant
5 → essential

Old LSE – SE

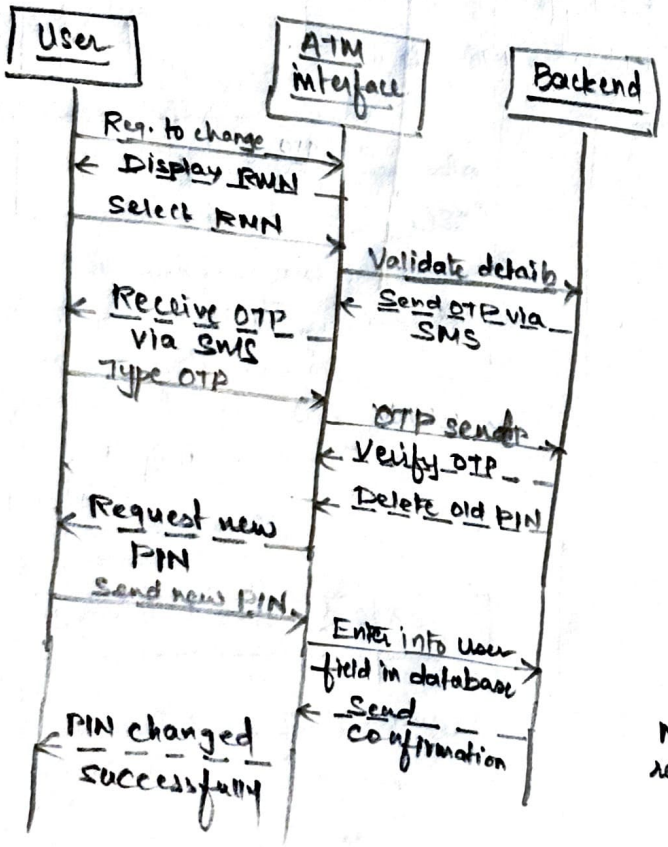
2020

Q.2

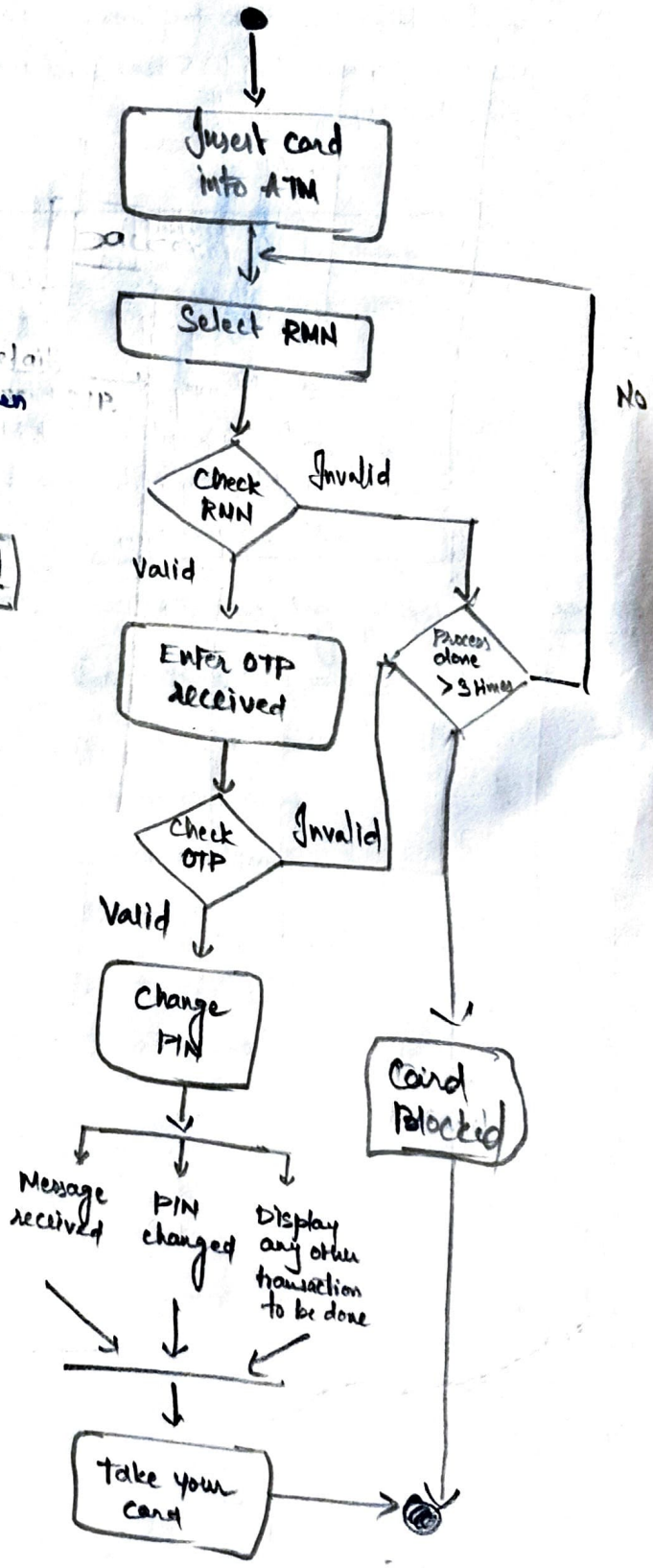
Sequence diagram

Assumptions :

1. The user inserts card into machine and selects option of changing PIN.
2. The user has entered the correct OTP.
3. Transaction defined to be completed when new PIN is stored.



Activity diagram



Mod - 2

Object oriented software design methodologies

Booch — Rumbaugh — Jacobson

Booch

- Micro dev processes: identify classes, semantics, relationships, interfaces & implementation between class & objects
- Macro dev processes: basic needs of software (conceptualization)

Rumbaugh

- Analysis: Object model, dynamic model, functional model
- System design: high level
- Object design: objects in detail
- Implementation: coding

Jacobson

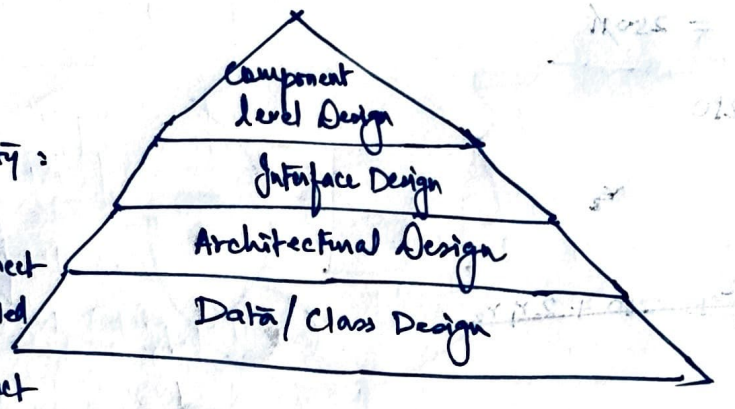
- Covers entire life cycle
- Req. model, analysis model, design model, implementation model, test model.

OOSE Activities

- Requirement Elicitation
- Analysis
- System Design
- Object Design
- Implementation
- Testing

Mod - 3 SE

Design Model



Steps for Quality:
- Arch. → represented
- Interface that connect the software → modelled
- Components → construct
- All views conform → basic design concepts → guide.
- Assessed to determine
  → errors / inconsistencies / dep.
  → better alternatives exist?
  → implemented within constraints/cost/etc)

Design
- Iterative, req. transfered to blueprint
- identify & prioritize qualities
- Explicit, Implicit, Latent Req.
- Readable & understandable guide for coding
- complete picture

Design Quality guidelines
- Architecture
  → using std arch. patterns
  → components of good design char.
  → implemented in evolutionary fashion
- Modular
  - elements
  - subsystems
- Distinct rep.
- Lead to appropriate data structures
- Components that exhibit ind. char. (f'al)
- Interfaces that reduce complexity of connections
- Repeatable method, driven by req. analysis info
- Notion that effectively communicates its meaning.

Design Processes
- Not suffer from tunnel vision
- Tracable to analysis model
- Not reinvent the wheel
- minimize intellectual gap b/w software & the problem at hand
- Uniformity & integration
- Structured to accomodate change
- Structured to degrade gently
- Design is not coding, coding is not design
- Assessed for quality as it is being created
- Reviewed to minimize conceptual errors.

Design Activities
- Architectural design
  - Info of app
  - Specific req.
  - flow plan / overview
  - Styles & patterns
- Interface Design
  - UI
  - Ext int
  - Internal int
- Component level design - fully describes internal detail of each component, defines DSA, rep. in UML
- Data Structure design - DSA defined & designed
- Algo. Design
- Deployment level design - how f'ality & subsystems will be allocated within the physical computing env.

Design Concepts

- Abstraction
  - i. Modular sol^n, many levels
  - ii. High level → broad
    low level → procedural
  - iii. Each step is refinement
  - iv. Procedural abstraction - seq. of instru., specific & limited f^n.
  - v. Data abstraction - collection of data, describes data object.
  - vi. Control abstraction - program controlling mech.

- Refinement
  - i. Elaboration, Top-down
  - ii. Successfully refining levels of procedural detail
  - iii. Elaborate & provide more detail on orig. statement as each refinement occurs.
  - iv. Abstraction & refinement are complementary

- Modularity
  - i. Divided → separate components (modules)
  - ii. Integrating modules → satisfy req.
  - iii. Allows program to be intellectually managable
  - iv. Divide & conquer.
  - v. 5 criteria - decomposability, composability, understandability, continuity, protection.

- Software Architecture
  - i. Hierarchial structure of modules, manner in which they interact
  - ii. Framework for detailed designing
  - iii. ~~5 criteria~~ Properties - Structural, Extra f^{n}al, Families of related systems.
  - iv. Arch. models - Structural, Framework, Dynamic, Process, Functional

- Patterns
  - i. Structure to solve prob within design context
  - ii. Intent determines -
    - → Pattern applicable to current
    - → Resuability
    - → Serve as guide for similar patterns.

- Information Hiding -
  - i. Hide data structures & procedural processing behind module interface
  - ii. Knowledge need not be known by users.
  - iii. Indep. module comm. only that info necessary
  - iv. greatest benefits when modifications req. because data & procedural detail hidden so ~~too~~ errors less likely to propagate.

- Functional Independence
  - i. Each module addresses specific subset of req., simple interface with others.
  - ii. Indep. modules (effective modularity) easier to develop.
  - iii. Indep. modules -
    - easier to maintain & test
    - effects limited
    - error propag. reduced
    - reusuable modules
    - key to good design.
  - iv. Qualitative criteria - Cohesion & Coupling

Cohesion - degree to which one module performs one & only one function. High cohesion advised

Coupling - degree to which a module is connected to other modules. Low coupling → interconnection, interface complexity.

- Refactoring
  - i. Changing software system such that it does not alter external behaviour yet improves internal structure
  - ii. Design examined for redundancy
  - iii. Unused design elements
  - iv. ineff./unnecessary algos
  - v. any other design failure to be rectified.

Pattern Based Software Design
- Ability to see patterns characterizing prob. & patterns to see sol^n.
- Set of design forces (non functional req)
- Pattern design indicate attributes to be adjust
- Apply existing design processes.

1. Pattern Based Design in Context
- Methods & modelling tools for arch. component, interface level design

2. Thinking in patterns
- Patterns, context (big picture)
- Work inward from context
- Iterative, refinement.

3. Design Tasks
- Examine req., develop prob. hierarchy
- Reliable pattern lang.?
- Architectural patterns? (from broad level downwards)
- Using collaborations, examine subsystems & find patterns to address
- Repeat till broad probs. addressed.
- Search for interface design patterns repo.
- Compare various patterns against existing ones for refinement
- Refine the design

4. Pattern Organizing Table
- Spreadsheet, organization, categorization

CN →

Software Arch.

Architecture enables –
- Analyze effectiveness of design, enabler for stakeholder comm
- Consider alternatives
- Early design decisions that will have profound impact on dev.
- Reduce risks.

Styles – encompasses –
- Set of components
- Set of connectors
- Constraints
- Semantic models.

SOFTWARE ENGINEERING MODULE - 4

SYSTEM IMPLEMENTATION AND CONFIGURATION MANAGEMENT.

Version Control

- combines procedures & tools
- manage diff versions
- config objects created during process.

Version Control capabilities

- Database (stores config obj)
- Version Management (stores all versions)
- Make facility (construct specific version)
- Version & change control (issues/bug tracking)

Elements of change management

- Access Control (who has access auth. & can modify config. obj.)
- Synchronization control (ensure ||el changes do not overwrite)
- Informal change control (developer changes)
- Project level CC (approval from PM to make changes)
- Formal CC (instituted when product released)

(# Go thru change flow diag. Fig 11, Mod 4)

Change set - collection of all changes req. to create specific version

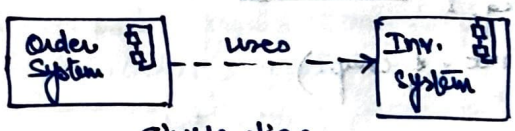
- construct version by specifying & identifying change sets by name.

Modeling VC systems

- Template for building version
- Construction rules
- Verification rules.

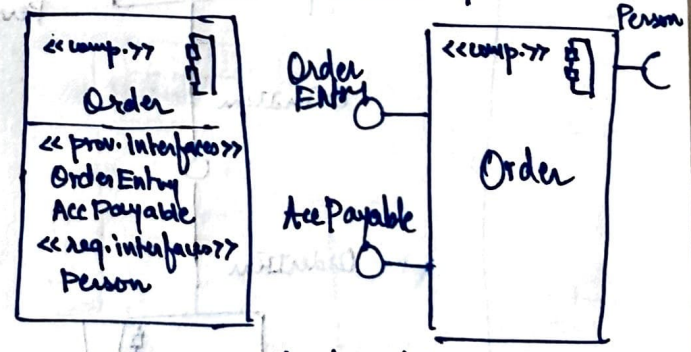
Component diagram

EX:



simple diag.

Diff. ways to show components



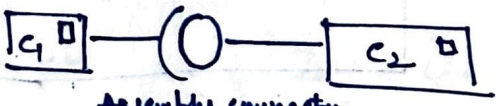
Diff. ways to show interfaces

Change control

- Large projects → uncontrolled rapid change leads to problems.
- Combines human procedures + automation
- mechanism for control change.

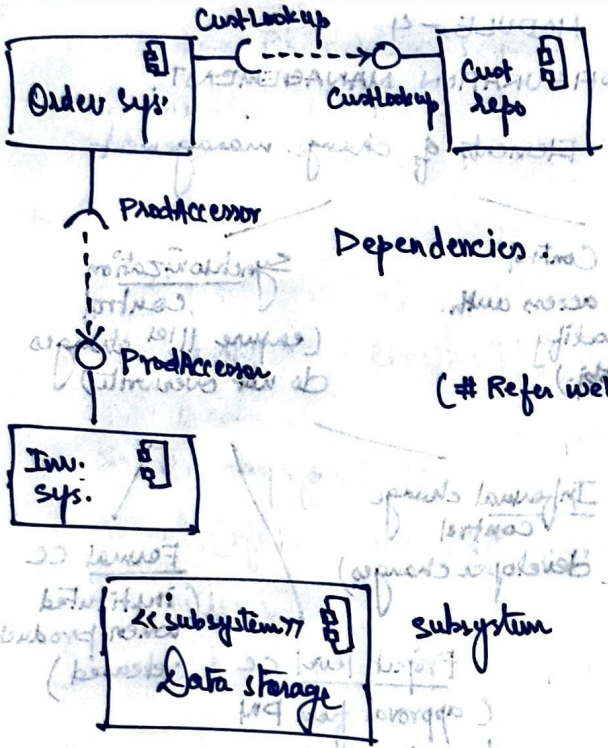
Engineering change order (ECO) - document

- new prod. design / details or proposed changes to existing.
- List of components, assemblies, other docs affected.
- Change control board (CCB) responsible to determine impact & whether ECO can be implemented on time.
- Authority to approve or reject.



Assembly connector

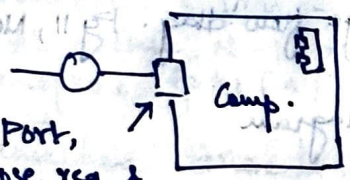
C2 provides interfaces that C1 needs.



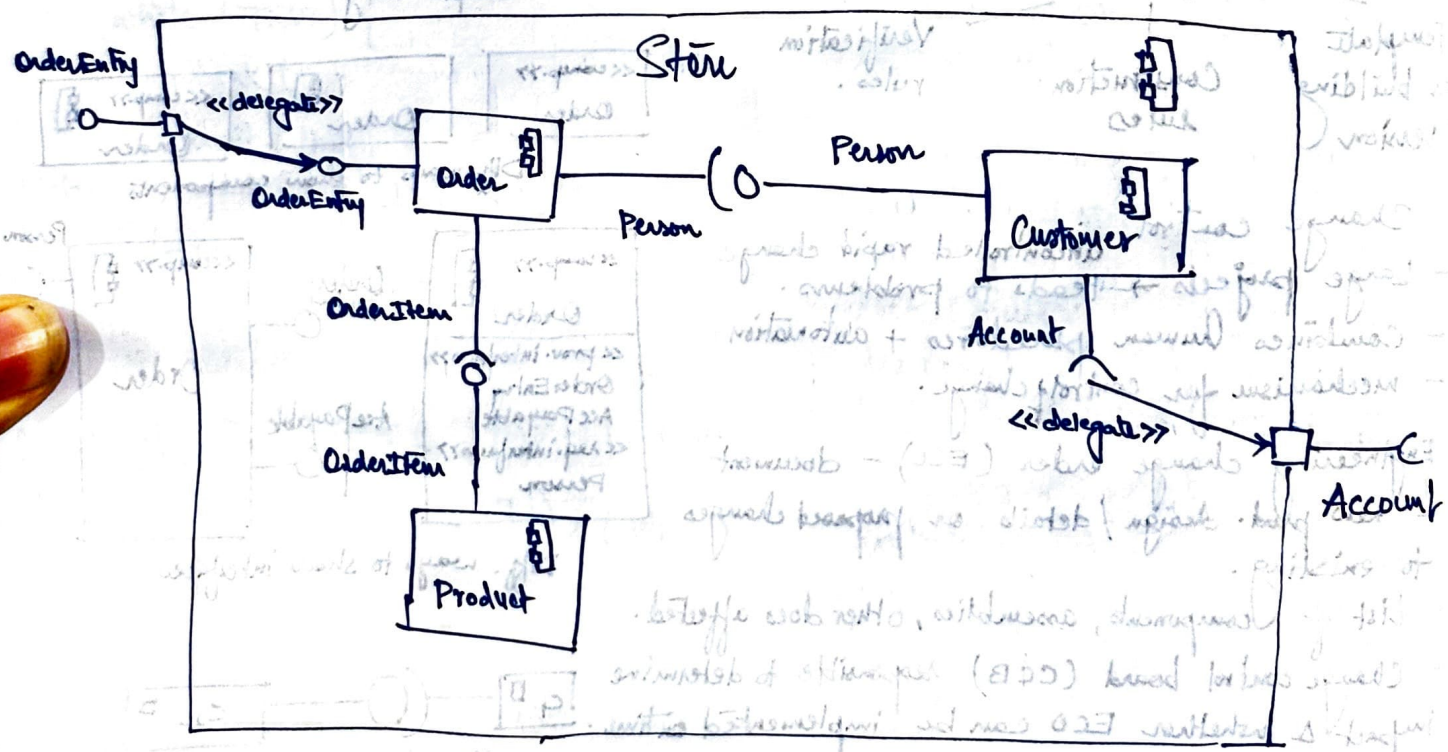
Dependencies

Component diagram
- structural diag. / static impl. view
- components, interfaces (prov. + req.), ports, relationships b/w them
- describes components used, not functionalities.
- structural relationships btw components (main)

(# Refer web store diag. Pg 24 Mod 4)



Port,
(expose req. & provided int. of comp.)



Components inside component.

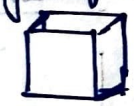
Deployment diagram

- Allocation of artifacts to nodes in physical design.
- Graph of nodes connected by associations
- Phy. config of software + hardware.

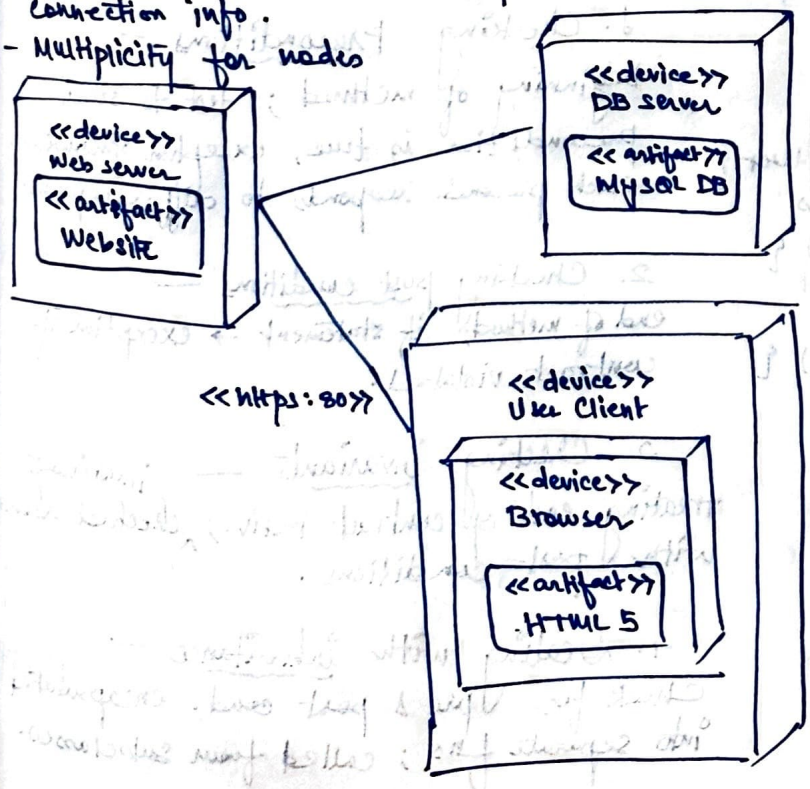
DD Elements — Artifacts, Connections, Nodes

Artifacts –
- Physical item
- Implements portion of design
- source file / doc / any other code related
- <<artifact>>

Nodes –
- computational resource
- memory & processing on which
- artifacts deployed for exe.
- 3D cube
- Only hardware entities



Connections –
- Comm. thru msgs & signals thru paths (solid lines)
- Usually bidirectional (for uni show arrow) →
- Optional label e.x. <<TCP/IP>> provides connection info.
- Multiplicity for nodes



Mapping Models to Code

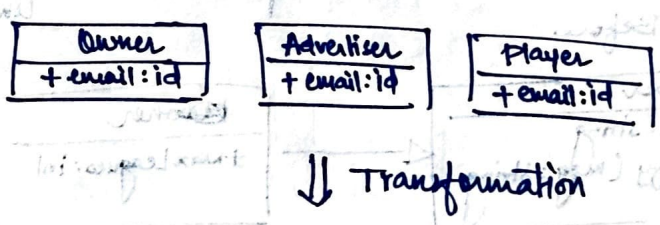
Transformation – improve one aspect while preserving all other properties of the model.

Activities
- Optimizations (address performance req.)
- Mapping class models to storage schema (persistent storage strat.; mapping to schema ex. RDBMS)
- Realizing asso. (map asso. to source code constructs)
- Mapping contracts to exceptions (describe behaviour of ops when contract broken; raise exceptions to violations)

Four types of transformations
- Model transform
- Refactoring
- Forward Engg.
- Reverse Engg.

1. Model transformation –
- applied to obj. model, results in another obj. model
- purpose → closer compliance with req.
- add / remove / rename classes / ops / asso. / attr.

Ex:



(3 refactorings)

2. Refactoring –
- transform. of source code
- improves readability / modifiability
- behaviour not changed of system.
- Improving design
- Small, incremental steps
- interleaved with tests.

(earlier example) –

3 refactorings (algo + code)

↳ Pull Up field – email from sub to superclass

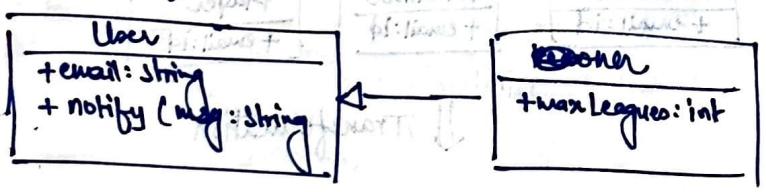
↳ Pull Up Constructor Body – constructor (init.) code from sub to super

↳ Pull Up Method – methods ~~[illegible]~~ concerning email from sub to super.

- Compile & test for each refactoring
- effectively transforms source code in same way as object transformation.

3. Forward Engineering —
- applied to set of model elements
- results in set of corresp. source code statements
- class dec. / Java expression / DB schema etc.
- purpose → maintain strong correspondance
  ↳ design & code models
  ↳ reduce impl. errors.

Ex: Before



⇓ Transformation

(source code)

```
public class User {
  ~~private~~ String email
  String getEmail() {
  }
  void notify() {
  }
}
```

```
class Owner extends User {
  int maxLeagues
  int getmax () {
  }
  void setmax () {
  }
}
```

4. Reverse Engineering —
- applied to source code elements
- results in set of model ele.
- purpose → recreate model for sys.
  why?
  ↳ model was lost / never created
  ↳ out of sync with code.
- inverse of forward engg.

Transformation principles
- Address a single criteria
- Local (few methods / classes only, at once)
- Isolation to other changes
- Validation step

Mapping Asso. to Collections

One to one
Uni
Bidirectional
Many to many
One to many

Mapping Contracts to Exceptions

1. Checking Preconditions – beginning of method; test to see if precondition is true, exception otherwise; each precond. responds to diff. exception.

2. Checking post conditions — end of method, if statement → exception if contract violated.

3. Checking invariants — treating each op contract indiv.; invariants checked along with post conditions.

4. Dealing with inheritance — check for pre & post cond. encapsulated into separate fns; called from subclasses.

5. Coding efforts —
code for checking pre & post may be longer & complex than code for real task.

6. Increased opportunities for defects —
checking code can have errors; inc. testing effort

7. Performances drawback —
systematic all contracts checking signif. slows code; response time & throughput goals not met even if correctness ✓.

(* Example of exception handling try - catch in Java).

Risk Management

Risk - chance of exposure to adverse effects/ consequences of future ~~of~~ events
(exaggerate & explain)
- uncertain event that can cause negative or positive effect on project objectives.
- ~~state~~ future possible problems
- Cause & effect
ex: developer leaves → delay task.

Sources / Natures of risk
- People (resource mismatch, ~~skill~~ skillset, less ppl)
- Tech (new tech)
- Structure (organization; Projectized/ Strong/weak matrix / open)
- Tasks (done for first time)

Risk resolving approaches
- Reactive (respond when risk occurs, fix on failure, crisis management)
- Proactive (Formal Risk Analy., TQM & SQA (stat.), developing skills)

Boehm's Risk Engineering

Risk Engg.
- Risk Analy.
  - Risk Identi
  - Risk Est.
  - Risk Eval.
- Risk Mgmt
  - Risk plan.
  - Risk Control
  - Risk monitor.
  - Risk directing
  - Risk staffing

Boehm's Top 8 Risks
(# Pg 7,8 Mod 4.5)

1. Personnel shortfalls
2. Unrealistic time & cost est.
3. ~~Unrealistic~~ Wrong software fn.
4. Wrong user int.
5. Gold Plating
6. Late changes to req.
7. Shortfalls in ext. supplied components
8. Shortfalls in ~~ext.~~ supplied ~~components~~ tasks.

Design to cost - 2,5,8
Incremental dev. - 2,6
Quality - 7,8
Formal specification - 3,7
Prototyping - 3,4,5,8

Risk Planning & Control
- Identification (what risks?)
- Assessment (which are most serious?)
- Planning (what we can do?)
- Monitoring (current status?)

RE = PD × P
- RE: Risk exposure
- PD: Money value
- P: Probability ($0 \le P \le 1$)

Risk Analysis & Prioritization

- Reduce risk exposure by
- reducing likelihood or impact
- Draw contingency plans to deal
- Risk assessment confidence
- Compound risk, no. of risks, cost of action

$$\text{Risk Reduction Leverage (RRL)} = \frac{RE_{before} - RE_{after}}{\text{Risk red. cost}}$$

RRL > 1 → gain of risk red. plan impl.

Dealing with Risks

1. General Risks - broad challenges/uncert. like unclear goals
2. Application Factor - type of nature of app; complex then extra testing
3. Staffing factor - staff satisfaction; turnover rate (rate of leaving team)
4. Project factor - clear obj. & goals; quality & formal plans
5. Project Methods - well defined & structured methods (Agile / waterfall); minimize confusion & risks.
6. Hardware & software factors - new tools/tech unfamiliarity, unexpected issues.

Risks can be dealt by

- Acceptance
- Avoidance
- Reduction
- Transfer
- Mitigation/ Contingency.

PERT - Program Evaluation & Review Technique.

- Proj. mgmt tool
- Plan, schedule, estimate time req. to complete tasks
- useful for uncertainity

a : Optimistic time / shortest time

m : most likely time / realistic estimate

b : Pessimistic time / longest time if delays

t_e : expected time

S : Activity std. dev. / range of uncertainity

$$t_e = \frac{a + 4m + b}{6}$$

$$S = \frac{b - a}{6}$$

Ex: User acc. mgmt in library mgmt system

a = 3 days

m = 5 days

b = 9 days

$$t_e = \frac{a+4m+b}{6} = \frac{3+4\times5+9}{6} = 5.33 \text{ days}$$

$$S = \frac{b-a}{6} = \frac{9-3}{6} = 1 \text{ day}$$

TESTING & MAINTAINANCE

Testing

- process to analyze a system / component
- detect diff.
- observed (existing) and specified (req.) behaviour
- set of activities planned in adv. & conducted systematically.

Principles of Testing

1. All tests should be tracable to software requirements
   - presence of errors; doesnt verify no bugs; most severe defects cause prog to fail req.
2. Exhaustive testing is not possible
   - Always spot tests; efforts need to be managed.
3. Tests should be planned long before it begins
4. Test early and regularly
   - early testing helps detect errors at early stage; helps correction process.
5. Accumulation of errors
   - no equal dist; prob. high at place where error already found
   - flexibility & responsiveness of testing
6. Fading effectiveness
   - effectiveness fades over time; repeat test cases doesn't give new errors
   - test cases altered & reworked time to time.
7. Testing depends on context
   - No 2 sys are same; cannot be tested same.
8. Pareto principle applies to software testing
9. No errors means usable system - False conclu.
   - error det. & fix. doesn't guarantee user expect. usable sys.
   - early user integration & rapid prototyping.
10. Testing should begin 'in the small' and progress towards testing 'in the large'.
11. Testing conducted by indep. third party to be most effective.

Goals of Testing

- Validation
- Verification / Defect testing

Validation Testing - To demonstrate to developer & customer
- software meets req.
- working expectedly given test cases set.

Verification testing - Discover situations
- behaviour is incorrect / undesirable
- does not conform to specification
- Defect testing - TC design to expose defects
- ensure software correctly implements f^n / prog. → activities.

Testing Concepts

1. Test component - part of sys. isolated for testing
2. Fault / bug / defect - design / coding mistake that causes abnormal behaviour
3. Erraneous state - indication of fault during execution.
4. Failure - deviation b/w specification and actual behaviour. Triggered by the erraneous states

Requirements of Testing

1. Testability
- how easily software can be tested
- metrics of testability

2. Operability
- Better it works, more efficiently tested
- few bugs

3. Observability
- What you see is what you test
- Distinct outputs; sys. states & variables visible;
past sys. states & variables;
Factors affecting output;
Incorrect o/p identified easily;
Internal errors automatically detected & reported; Accessible source code.

4. Controllability
- Better we can control, more testing automated & optimized.
- All possible o/p generation
- All code executable
- SW & HW states directly controlled by test engineer
- I/P O/P consistent & structured
- Tests conv. automated, specified & reproduced.

5. Decomposibility
- By controlling scope of testing, quickly isolate problems, smarter testing.
- software built from indep. modules; modules can be tested indep.

6. Simplicity
- Less to test, more quickly tested.
↳ Functional simplicity
(feature set min. necc. to meet req.)
↳ Structural simplicity
(modularized arch. to limit fault propag.)
↳ Code simplicity
(coding standard for ease of insp. & maint.)

7. Stability
- Fewer the changes, fewer the disruptions
- changes to software infrequent, controlled, do not invalidate existing, recovers well from failures.

8. Understandability
- More info, smarter testing
- Design, dependencies well understood
↳ internal
↳ external } components
↳ shared
- Technical documentation is
↳ instantly accessible
↳ well organized
↳ specific & detailed
↳ accurate.
- changes to design well communicated.

Test Case
- set of conditions / variables
- tester determines whether system
- satisfies req. (works correctly) or fails.
- help find problems in req. or design.

Test case attributes
- Name
- Location (path/URL)
- Input commands/data
- Oracle (expected behaviour)
- Log
  - time-stamped correlations b/w exp. & observed o/p.

Software Testing Strategies

1. Testing begins at component level and works outward towards entire system integration.
2. Different testing techniques appropriate at diff. points of time.
3. Testing conducted by developer.
   For large projects, indiv. test group conducts testing
4. Testing & debugging are different, but debugging should be in ~~every~~ any test strategy.
5. Implement low level & high level tests
6. Provide guidance to ~~praction~~ practitioner & set of milestones to manager.

Strategic Issues

1. Specify product req. in quantifiable manner long before testing starts.
   - Objective → find errors
   - Assess quality charac.
   - Measurable req → unambiguous results.
2. State testing objectives explicitly
   - Sp. objectives → measurable terms, stated within test plan.
3. Understand the users of the software and develop profile for each category.
   Use cases → interaction scenario → reduce overall testing effort
   focus on actual use testing
4. Develop a plan that emphasizes 'Rapid cycle Testing'.
   - Mindset, Skillset → Quick, less exp., best result testing
   - Feedback → control quality levels → test strategies.
5. Build robust software designed to test itself
   - ~~Capabl~~ Capable to diagnose some errors
   - Accomodate automatic testing & regression testing
6. Use effective technical reviews as filter prior to testing
   - Technical reviews very effective
   - Reduce amount of testing effort req.
7. Develop a continuous improvement approach for testing process.
   - Measurable test strategy.
   - Metrics collected use as part of Statistical process control approach for testing

Verification & Validation (SQA)

Verification - set of activities
- ensure software correctly implements f^n / prog / algo.

Validation - diff. set of activities
- ensure software built / tracable acc. to customer req.

(# SQA activities Pg 11 Mod 5.2)

Formal Technical Review (FTR)

FTR → software quality control activity (meetings)

Objectives :
- Uncover errors (f^n/logic/impl.)
- Verify that software meets req.
- Ensure software rep. acc. to predefined stds.
- Achieve software dev. in uniform manner.
- Make projects more managable

FTR → training ground → observe diff. approaches

Walkthroughs and inspections ← Analysis, design, impl.

Properly planned, controlled, attended (successfully)

FTR
- The Review Meeting
- Review Reporting & Record Keeping
- Review Guidelines

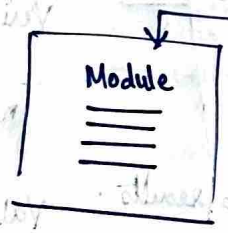
Levels of Testing
- Unit testing
- Integration testing
- Regression Testing
- Acceptance Testing
- Black ~~White~~ box
- White box

(i) The Review Meeting
- 3-5 people
- Adv. prep. (< 2hrs per each person)
- Duration < 2hrs
- Review leader + Producer + all reviewers
- Recorder – one of reviewers
- Producer – walk through
- At the end, resolutions/decisions:
  1. Accept without further modif.
  2. Accept with minor revisions
  3. Reject due to severe faults.

(ii) Review Reporting & Record Keeping
- Recorder records all issues raised.
- Review issues list produced
- What was reviewed?
  Who reviewed it?
  What were findings & outcomes/conclu.?

(iii) Review Guidelines
1. Review product, not producer
2. Set agenda & maintain. FTR on track & schedule.
3. Limit debate and rebuttal
   - when issue raised, may not be univ. agreement on impact.
4. Enunciate problem solving areas, but do not attempt to solve every problem. FTR is not prob. solv. session.
5. Take down written notes – good practice helpful for wording & priority assessment as info is recorded.
6. Limit no. of participants & insist on advance prep.
7. Develop checklist for each prod to be reviewed.
8. Allocate resources & schedule time for FTR.
9. Conduct meaningful training for all reviewers.

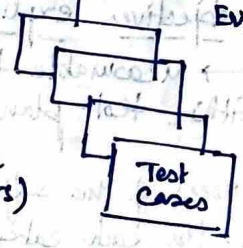
1. Unit testing
- Indiv. units/components tested
- Functional correctness of standalone modules
- Control paths tested to uncover errors within boundary
- Rel. complexity & uncovering errors limited by constrained scope for unit testing est.
- Internal processing logic & data structures
- ||el for multiple components



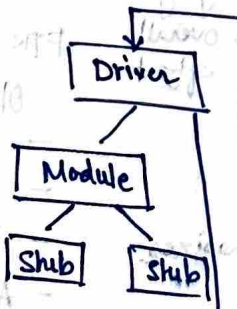
Interface
Local DS
Boundary cond.
Indep. paths
Error handling

Ensures
- Module interface → info properly flows in & out
- Indep paths → all statements executed.
- Boundary cond. → limit/restrict processing
- Error handling paths → tested
- Data flow → tested first

(i) Unit Test

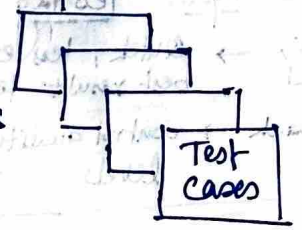


Boundary testing
- max or min value allow occur (errors)

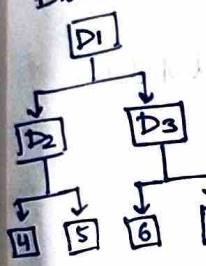


Int.
LSD
Bound. cond.
Indep paths
Error hand.

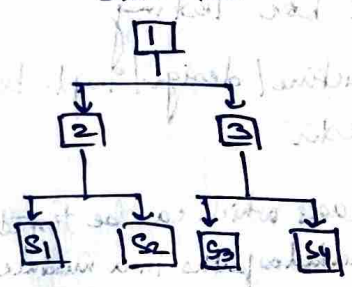
(ii) Driver – Stub testing



Driver Flow



Stub Flow



Drivers → simulate high level modules
Stubs → low level modules
Both are dummies → software that must be written but not delivered with final

2. Integration Testing

- Systematic technique
- constructing sw arch. +
- conducting tests at same time
- uncover errors associated with interfacing

why integration testing?
- Data can be lost across int.
- one component can have adverse effects
- combining subfunctions may not produce desired fn.
- global data structures can cause prob.

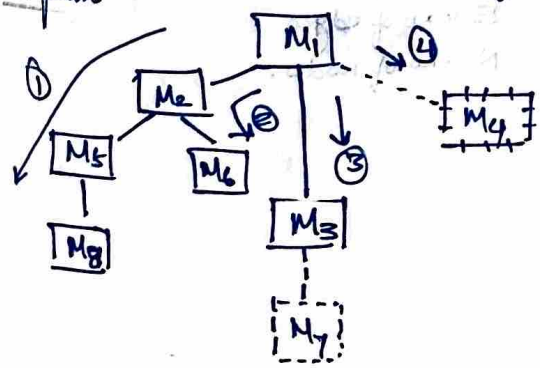
Incremental integration – small incremental construct & testing
- easier to isolate & locate errors.
- more likely to be tested completely.

Strategies
- Top down
  - Depth first
  - Breadth first
- Bottom up

Top down – top levels tested first; after that lower levels step by step

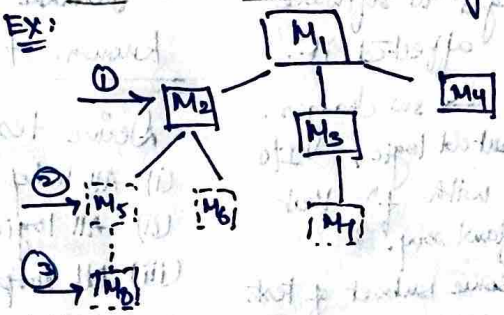
depth-first integrates all components of major control path

Ex:



breadth-first incorporates all components directly subordinate at each level moving horizontally

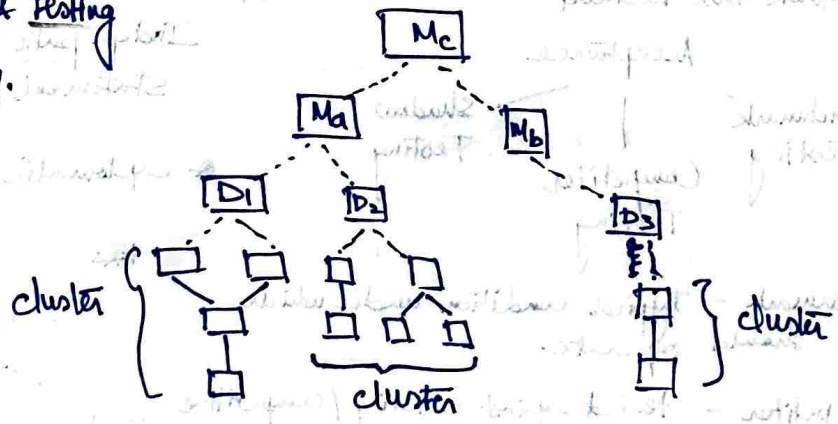
Ex:



Steps to ~~inte~~ integrate (top-down) –
- Main control module → test driver
  Stubs → substituted ∀ comp. directly subordinate
- Depending on approach, subordinate stubs replaced one at a time with actual components.
- Tests conducted as each comp integrated.
- After that another stub replaced
- Regression Testing → ensure no new errors.

Adv: ~~Subordinate~~ early verify major control/decision pts
Disadv: low levels req. to adequately test upper levels

Bottom-up – bottom levels tested first; after that upper step-by-step.
- need for stub elim.; whole pgm exists after last module int.



Steps –
- Low levels combined into clusters/builds → specific sw fn.
- 'Driver' written → coordinate TC i/p o/p
- Cluster tested
- drivers removed & clusters combined moving upward.

3. Regression Testing

- Ensure that changes to software has not adversely affected it.
- New module added → sw changes.
- New data flow, control logic, I/O
- May cause problem with f^ns that worked prev. flawlessly.
- Re-execution of some subset of tests already been conducted
- ensure changes haven't propagated unwanted side effects.
- Reg. test → manual
  → automated playback capture tools.

Reg. Test. Case Suite Classes

- Rep. sample (all sw f^ns) general
- Additional tests (sw likely to be affected)
- Focused (sw that has been changed)

4. Acceptance Testing

- Determine if sw met the req. or not
- Evaluate sys. business req., verify delivery
- Black box method

Acceptance

- Benchmark Testing
- Competitor Testing
- Shadow Testing

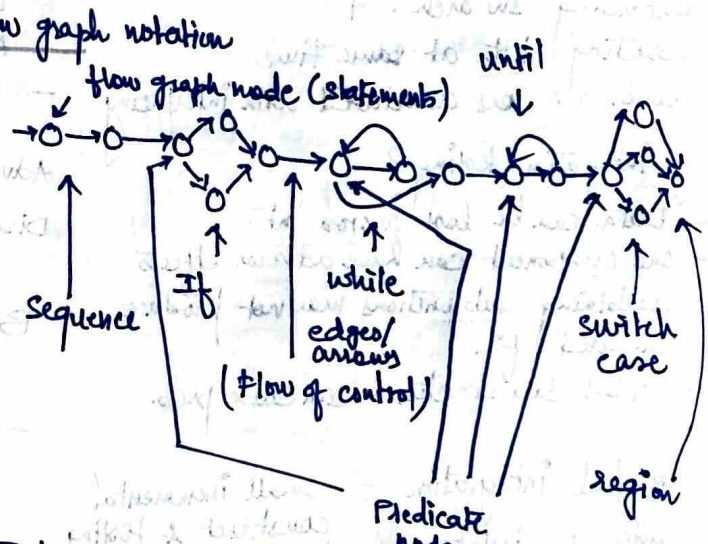
- Benchmark – typical conditions under which sys. should operate.
- Competitor – tested against existing / competitive product.
- Shadow – new & legacy run in parallel, output compared.
- After acceptance testing client → PM if req. not satisfied
- Dialog b/w client & developers
- Req. change → SDLC iteration
- sys. accepted if customer satisfied.

5. White box Testing

- Internal structure / design / impl. being tested is known to tester
- Derive test cases which can be tested:
  (i) All indep ~~modules~~ paths in a module
  (ii) All logical decisions (T/F)
  (iii) All loops at boundaries within bounds of operation
  (iv) Validity of internal data structures.

- Basis path method – complexities, exec. paths, data flow, indep. paths & procedural design
  TC guarantees to execute every statement

- Flow graph notation



Indep. path → atleast one new set of processing statements / condition.

- Cyclomatic complexity → no. of indep. paths further used in TC dev.

Cyclomatic complexity

- $V(G)$ = No. of regions
- $V(G) = P + 1$, P = no. of pred. nodes
- $V(G) = E - N + 2$, E = no of edges, N = no. of nodes.

Deriving TC in white box -

1. Using design/code, draw flow graph
2. Calculate cyclomatic complexity
3. Determine linearly indep. paths
   $V(G) \rightarrow$ upper bound
4. Prepare TC that force execution of each path.

- Graph matrix → assists in basis path testing

5.4 Black-box Testing

- Behavioral testing
- functional req. testing
- ~~errors~~ find errors in incorrect/missing fn, interface errors, errors in DS/ext DB access, behavioral/performance errors, initialization/termination errors.
- not white box alt., uncovers diff. class of errors but complementary

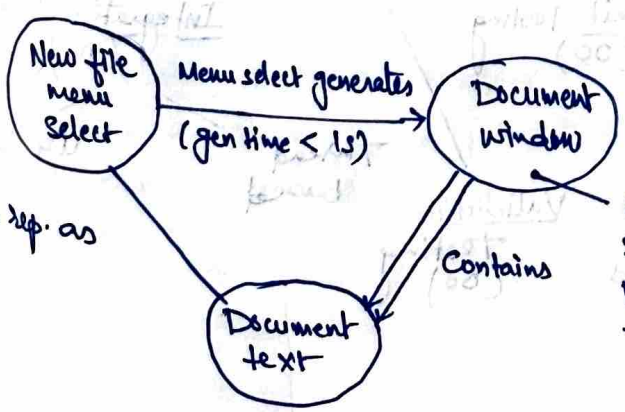
Criteria -

- Identifying classes/types of errors
- additional TC designed to achieve reasonable testing.

Graph-based Testing models

- Understand obj. modeled → relationships
- Create graph of imp obj & relation
- Series of tests → verify all obj. & relationships with each other
- tests exercised & errors uncovered.

Ex:



Attributes:
Start dimension
Background colour
Text colour

— : undirected
→ : directed (link wt)
= : Parallel links

Node weight (value)

Graph based behavioural testing methods

- Transaction flow modeling (how diff modules will be invoked; Data flow diagram)
- Finite state modeling (diff. observable states; State transition diagram)
- Data flow modeling (transformations of one data obj to other translation)
- Timing modeling (seq. conn.; specify req. exec. times)

Equivalence Partitioning

- divides I/P O/P of unit to partitions of data
- derived from req. specif. of input
- TC cover each partition atleast once.
- uncovers classes of errors.
- valid/invalid states for input cond.
- number or range of val. or boolean inp. cond.
- TC selected such that largest no. of attributes of equiv. class exercised at once.

Guidelines —

1. if (input cond = range) define 1 valid; 2 invalid equiv classes
2. if (input cond = specific value) define 1 valid; 2 invalid
3. if (input cond = member of set) define 1 valid; 1 invalid
4. if (input cond = boolean) define 1 valid; 1 invalid

Input condition

- Range → 1V, 2IV
- Sp. value
- Member of set → 1V, 1IV
- Boolean

Boundary Value Analysis (BVA)

- Selecting test cases exercising boundary values. (edge cases)
- no. of errors at bound. > no. of errors at center
- BVA derives TC from ~~output domain~~ input + output domains.

Guidelines —

1. $a \le$ inp. cond. $\le b$ (range) then design TC ~~just below a & just above b~~ just above and below a & b.
2. inp cond = no. of values then design TC for max & min values.

5.5 Review of OOA & OOD models

OOA – Object oriented Analysis (what sys. needs to do)

OOD – Object oriented Design (how it will fulfill req.)

Object oriented Software — Req. (Analysis) — Design

↑

Review useful → same semantics → various levels

Early review in Analysis –

- Unnecessary special subclasses for invalid attributes avoided
- Avoid misinterpretation of class def. → incorrect relationships.
- Behaviour of sys./classes improper characterization to accomodate extraneous attr. avoided.

Early review in design —

- Avoid improper alloc. of class to subsystem / tasks
- Avid unnecessary design work
- Avoid messaging model. incorrect

Later stages –

- provide substantial info
- structure & behaviour
- subj. to rigourous review prior to generating code.

Correctness —

- Syntactic correctness → proper symbol use.
- Each model reviewed → proper modeling convention
- Semantically correct if accurately reflects real world.

Consistency —

- judged by considering relationships among entities
- each class & its connection to other classes
- Class-responsibility-collaboration (CRC) model

Steps to evaluate –

1. Revisit CRC model & obj-rel. model seq.
2. Inspect description of each CRC index card to determine if delegated responsibility is part of collab. defn.
3. Invert connection to ensure each collaboration that is asked for service is receiving req. from reasonable source.
4. Determine if classes valid / resp. properly group among classes.
5. Determine if widely req. resp. might be combined.

Object oriented Testing Strategies

- Unit Testing (OO)
- Integration Testing (OO)
  - Thread based
  - Use based
- Validation Testing (OO)

1. Unit testing
- class, obj. → attributes & ops.
- smallest testable → class
- superclass → ops. def. → inherited by subclass
- op x() → varies indirectly
  ∴ need to test in context of each sub.
- cannot test standalone
- class testing → ops encap. by class & state behaviour

2. Integration Testing
- 2 parts – thread, use based

Thread based
- integrate set of classes req.
- respond to one i/p / event of sys.
- integrated & tested indiv.

Use based
- testing indep. classes → construct sys.
- dependent on indep. classes tested
- seq. continues till entire constructed
- no drivers & stubs used.

3. Validation Testing
- Focuses on User-visible actions & user ~~visible~~ recognizable outputs
- Draw upon use cases based on req model
- Scenario → high chance of uncovered errors
  ↓
  user interaction req.

---

3. Rev. engg -
- Disassembling competitor product
- Understand its design & manif. secrets
- Extract data, tools, arch., procedural designs from existing prog.

4. Code ~~restructing~~ restructuring

5. Data restructuring
- current arch. dissected, necessary data models identified.
- Obj & attr. identified, data structures reviewed

Software Transformation / Rejuvenation

- Redocumentation (creating / revising alternate rep. of software)
- Restructuring (transforming code without changing behaviour)
- Reverse Engg. (analy. to extract info about beh. & structure; Design recovery - design models recreated from code, docu, domain info)
- Re-engg. (reconstitute in another form, renovation, rebuilding)

Software Re-engg process model

Forward engg ⑥ → Inventory Analysis ①

Data restruct → Code restruct.

Document restructuring

Reverse Engg.

1. Inventory analy. –
- spreadsheet model
- info that gives detailed description of every active app.
- revisited freq. as status can change any time

2. Document restructuring
- Legacy sys. → weak docu.
- Creating docu → time consuming → static prog.
- Update only changed portion ∵ resources limited
- Business critical & redocu. req. then use intelligent approach to minimize docu.

6. Forward Engg
- Recovers design info from existing &
- ~~use~~ alter / reconstitute existing
- improve overall quality.

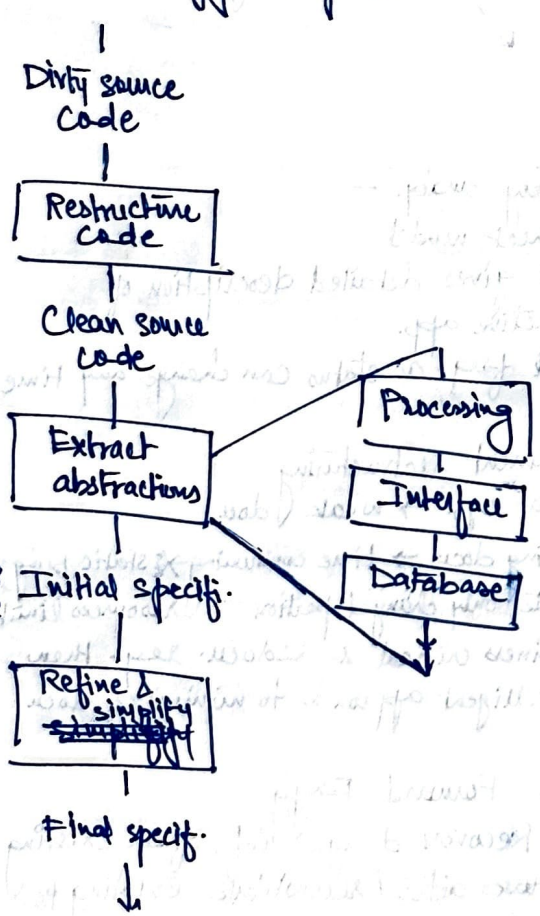
Reverse Engg.

- Recreating design by analy. final prod.

Terms:

1. Abstraction level – sophistication of design info extracted from code.
2. Completeness – level of detail provided at abstraction level.
3. Interactivity – degree to which human is integrated with automation for effective rev. engg. process.
4. Directionality
   - One way (maintainance)
   - Two way (restructuring)

Reverse Engg. diagram



Software Maintainance

- general process
- changing sys. after it has been delivered
- simple changes → coding errors fix
  extensive → design errors fix
  sig. enhancement → specif error fix / include new functionality

Types of Software Maintainance

- Fault repairs / Corrective maint. (coding errors → cheap, design → exp., Req. errors → most exp.)
- Functionality addition / Perfective maintainance (necessary when sys. is change, scale of change very high, perfecting software by adding new features)
- Environmental adaptation / Adaptive maintainance (modified to adapt to cope with sys. env. changes)