

Requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation

The process of finding out, analyzing, documenting & checking these services is called requirements engineering (RE)

- User requirements are broad statements, NOT plus diagrams
- System requirements are more detailed descriptions of services
- SRS document

User story

As a <type of user> I want <some goal> so that <some reason>

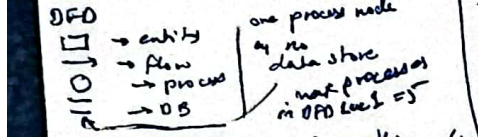
Acceptance Criteria - step by step

RE Tools

Inception, Elicitation, Elaboration, Negotiation, Validation, Management

Inception, 1st, 2nd, 3rd step

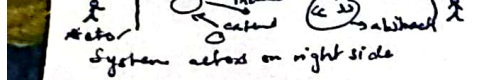
DFD Level 0 or context diagram



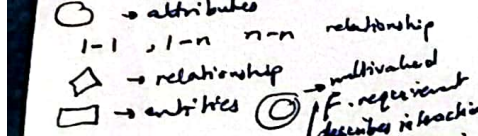
verb is process, noun is anything else

DFD Level 1 contains data stores

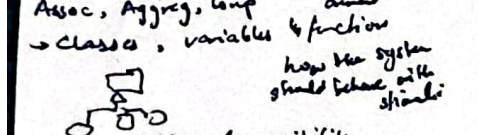
Use Case Diagram - remember SDA



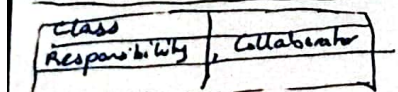
ER Diagram - DB



Class Diagram



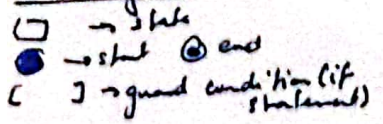
CRC (Collaborator Responsibility Collaborator)



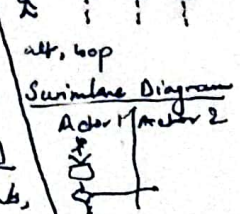
CRC Modelling

- no reviewer should have 2 cards that will collaborate
- leader reads use case - as leader times to a moving object, passes a token to a person holding the corresponding class card
- when token is passed, card person tells responsibilities on card
- If error then remove list or new classes

State Diagram - SDA



Sequence Diagram



Perlin's Nets



Decision Table

Condition	Condition Entries
Action	Action Entries

Requirements Prioritization

- essential, desirable, optional
- Characteristics of Requirements
 → correct, consistent, complete, unambiguous, feasible, relevant, testable, traceable

Functional / NF Requirements

Functional describes what product should do

NF describes how the product should do it

- F → system must do [req]
- NF → system shall be [req]
- NF → constraint, design, process

Testable Requirements

- unambiguous, quantitative, nouns
- Validation ensures we build right system
- Verification ensures we build system right

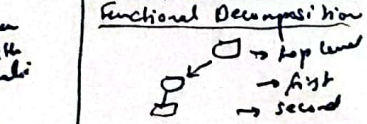
Measuring Req.

→ from 1 to 5, lower score is good

low means you understand requirement

- Cloning: borrow a design/code & make minor adjustments
- reference model: standard generic architecture

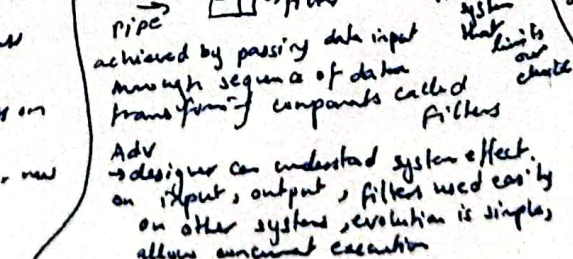
Decomposition & Views



- Feature oriented → features to modules
- Data → for data to modules
- Process → process to modules
- Event → events to modules
- Object → objects to modules

Architectural Styles

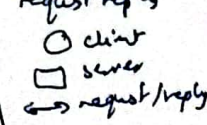
PIPE & Filter



- Dis
 → support fixed data, pass data, & compute hamper performance
- Not good for handling interactive application

UI Rules
 → make interface consistent, place the user in control, reduce user's memory load

Client Server request reply



Peer-to-Peer

each component executes as its own process is client & server to other components

→ has interface

- Adv
 → scale
 → address, system capabilities increase
 → Network failures tolerant
- Dis
 → add requests

Not to use

- file contents change
- speed has impact on file quality with all but required
- Publish/Subscribe

- Adv
 → strong support for evolution
 → component can be added without affecting others
- Dis
 → need shared repository
 → Difficult to test

Repository

- knowledge source
- blackboard
- read/write
- layering
- Adv
 → useful decomposition
- Dis
 not easy to structure layers
 → performance cost

Call & Return

Main Program/Subprogram

Quality Attributes

- maintainability, performance, security, reliability, robustness, usability, business goals
- Cost-benefit Analysis
 $ROE = \frac{Benefit}{Cost}$
 $\% \text{ avg ROE} = \frac{Benefit - Cost}{Cost} \times 100$
- Payback = $\frac{Cost}{Benefit}$
- Trade off Analysis
 - Data Abstraction Solution
 - Circular shift
 - Alphabetical shift
 - change & system flexibility not easy
- Implicit Invocation Solution
 - Data in AOT
 - Repository solution
 - Pipe & Filter

Implicit Invocation best - easy to change algo, good performance, efficient data representation

Analysis Module

- Scenario based, use case, stories
- Class = class diagram
- Behaviour (state, sequence, CDF, data models)
- Flow
- State flow
 → makes DFD level 1, 2, 3
 → Make boundary
 → Put into diagram
 → Do first level function & repeat

Trade-off

- Repository
 → breaks problem into four primary functions: input, circular shift, alphabetical & output
- concludes by a master program
 → data are categorized
 → Design is difficult to change modules access & manipulate the data directly
- some of the elements are reusable

Data Abstraction Solution

- data computed by each computational module
- stored in module
- circular shift
- contains index to words in text
- Alphabetical shift
- makes a sorted version of this
- easier to reuse
- changes to functionality not easy

Implicit

- Data in AOTS
- AOTS generic more reusable for data abstraction
- easily extended
- multiple compute methods may create by end, which order

Pipe & Filter

- filter modules
- easily extended
- operates in parallel
- space efficiency
- circular shift
- can no longer be top as indices of text
- parallel copies of text

Component

