SWE

Lec 3

- Core design concern of user-centric SW, is to
  Design usable sys (to cater to the needs & expectations of "layman" us
- => we need a sys. approach,
  so we use SDLC

SDLC: Software Development Life Cycle

- To comprehensively capture & represent the design & development activities
- Build SW in stages.
  (or)
- SDLC is nothing but a stage-wise development process of SW.

Eg: Calendar app.

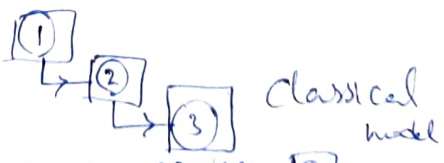
① ~~—~~ possible designs. (no unique-solution, there are many possible designs).

② — how, choose the right one.
for this, we need a sys. approach.
It is SDLCs. Use SDLCs to decide which one's better.

- There are many models in SDLCs.
  - Waterfall model
  - Spiral model
  - Evolutionary model
  - .....

→ most well-known.

Waterfall model:



- Seven major stages.

① Feasibility Study — availability of resources

② Requirement gathering, analysis & specification

→ not in terms of resources, but in terms of user-requirements, identify & gather them. Then analyse them to see if possible.

③ System design — both interface design, as well as the design of system & code.

④ Coding (actual implementation) and Unit testing
↳ identifying bugs, taking correct measures

⑤ Integration and Systesting

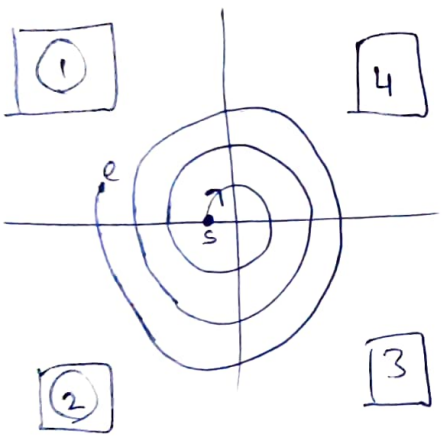
⑥ Deployment

⑦ Maintainance.

- We can even, iterate few of the ~~sgs~~ steps in the model, then it's called iterative waterfall model.

Spiral model:

- A meta model - encompasses other SDLCs.



① Identify Objectives
② Risk assessment & Mitigation.
③ Develop~~ed~~ (Prototype)
④ Customer evaluation & planning.

these can be said as iterations,
one circle - one ite (each ite has 4 phases/quadrant)

① → Objectives of the iterative phase are identified.
→ also the risks associated with these obss.

② → identified risks are analysed in detail.
→ Steps taken to reduce the risks.
eg: if there's a risk of inappropriate requirement specifica
then a prototype system may be developed to ~~help~~
mitigate this risk.

③ → Develop product (prototype) after resolving identified risks and evaluate.

④ → review the results so far, with customer and plan next ite, if required.

- Spiral model is not a fundamental development model, it captures higher-level concers or activities to implement lower-level or fundamental level SDLC.
- Through the ites, progressively more complete version of the SW gets built.

Iterative System & SDLC:

- Difficult to express with traditional SDLCs (eg: waterfall)

Interactive SDLC:

Waterfall model & UCD

waterfall → good for efficient development
not for UCD.

in UCD →

- requirements → usability requirements → has nothing to do with platform (or) programming lang
- feasibility →

usability vs feasibility ?

design & imp specifications.

- Gathering of usability requirements
  - → non-functional requirements.

Usability requirements → one type of NFR

NFR: ① Perf. related
② Operating constraints
③ Eco. Economic considerations
④ Life-Cycle requirements
⑤ Interface issues < HW / SW .

FR → easy to convert to code.

FR : Functional Requirements.

- Specifies a system in terms of "functions".
  - including input & output (also purpose)

  FR → input, output & purpose

- main idea = Abstraction
  i.e, identify the black box func, to be supported by system
  → with clearly defined input & output
  → no concerned about how it works, only about input & output

- Decomposition & Hierarchy : to manage ~~it better~~ the functional description better
  into sub-funcs.

- While identifying the functions :
  - Avoid exploratory style : from implementation to func (not the other way round)
  - Abstraction
  - Hierarchy.

FR ! not created based on end-user studies, some domain knowledge & some discussion with clients can be a good start then ~~work~~ subsequently usability required (some use-case) can be converted into FR & can be included later.

- End product in the function requirement Specification stage : functional Hierarchy.

SRS (Software Requirement Specification).

- these requirements in textual form, is SRS document
- (or) The "formal" representation/specification of requirements
- Hierarchy structure (R1, R1.1 etc)
  - each R (R1, R1.1, etc) → a function with clearly defined IP & OP and some description (optional).
  - R can be sub-divided again.
  - ~~4~~ more decomposition => better specification } esp for complex systems, simple systems no need much for multi-hierarchy
    less levels => not very specific

- Representation:
  name, IP, OP, description, Level.
  - Abract: nature of func: what we need only spec, NOT how to get
  - hierarchgl organisation
  - Naming convention: should be consistent in all funcs, levels.
- We should not be too specific about the IP & OP, we should keep it open for the later on implementation, at later stages of DLC. Should be kept flexible.

- version, date, prepared by, place, supervised by
- Contents (Sections & Sub-sections)
- Revision history (of versions, changes, dates)
- The Contents

eg: Introduction:
1. Purpose
1.2 doc convention (maybe like Short-forms)
1.3 project scope

Sections

2. Overall description:
2.1 product perspective
2.2 product functions
2.3 user classes & characteristics
2.4 Operating environment
2.5 Design and implementation constraints
2.6 Assumptions and Dependencies.

380
320
700
800
1500
500
2000
1000

3. Ext. Interface Requirement
3.1 User Interface
3.2 Hardware Interfaces
3.3 Software Interfaces
3.4 Communication Interfaces

No action words in OUTPUT, INPUT.

RAT not required at every sub-func, 1/2 places - enough wherever required.

Func, Sub-fn can have IDs

most imp
4. Functional Requirements
4.1 User class 1: Instructor
1. Start class session
1.1 Established Connection - Instructor.
1.1.1 Realtime Individual Student Attnt. Statistics
1.1.2 Realtime Overall Class Attnt. Stats
1.1.3 End class session
1.2 Connection Error Notification

4.2 User class 2: Student

imp
5. Other NFR. (Usability Requirements)
5.1.
5.2
5.3

no IP, OP, Desp format like FR, just points.

Usability Requirement Gathering:

- it's a NFR
- Contextual Inquiry - ~~approach~~ CI

CI:

five stage process, Performed for URG.

1. ~~Plan - plan the process~~
   - observation based,
   → Primary objective is to observe the user, Contextual use
   → observer can take occational interviews

1. Plan - plan the process
   Preferably in form of a script, note down step-by-step sequence
2. Initiate - Start the process
   not like starting the Inquiry, but in form of communication,
   explain the users the purpose & ask for their time ....
3. Execute - perform the process
   two ways
   1) active mode: observer physically present
   2) Passive mode: observer is not present, some of the observe recorded, the recordings are used for URG.
4. Close - end the process
   send thank you notes, formally notify them that it's done, ...
5. Reflect - Analyse data
   analyse data, to get the info required.
   - we can follow diff methods, one of them is affinity diagram

→ affinity diagram method.

Affinity Diagram Method:

Five steps

1. Generate idea – whatever observed, take notes, generate ideas.
2. Display ideas – any way eg: on a sticky note, using text doc...
3. Sort ideas into groups.
4. Create "group headers".
5. Draw finished diagram – it contains ideas sorted into groups with some headers and arranged in some sequence.

Design: Two types

① Design of the interface & interaction. – User's POV

② Design of the system – system's POV

- In design–Prototype–evaluation cycle – we refer to ①
  this cycle stops when the design stabilises ⇒ no further problems found, after the evaluation of the prototypes.
- ② may not require Prototyping & quick evaluation.
  – there are other ways (in Code testing stage)

- In requirement gathering stage, the key requirement in the POV of interactive system) is usability of the system

  ⇒ usability & requirement – applicable for I & I design

  not necessarily a requirement for system design.

- Converting usability requirements into FR, to ~~integrate~~ integrate it into the ... ↓ we get them from CI

- The final outcome of affinity diagram is a set of headers with some observations in each of them.
These are used in development of the system.
- Two ways to use this info
  1. Come up with a set of "design guidlines / recommendations/principles," to guide the overall design of the interface
     → design of IRI, not the system.
  2. "Creation" of FRs

① → guidelines: guidelines say what to do / a function that has to be performed.
basically, guidelines are used to add functionality to the final product.
(where are used frequently by the specified group of users, in the specified work content).
→ This helps / leads to more efficiency, effectiveness and satisfaction of the end product, as per the std. def of usability.

Creation of FR:

Add new FRs based on the observations.

Eg: ~~IR5~~

- usability — NFRs.
not mandatory to convert into FRs.
but, some of them can be converted.
This helps in improving usability.
  - no worries even if there are few usability reqs, which can't be converted
  - we can specify those in NFRs, seperately

- design - prototype - evaluate cycle
  - ↳ refers to design of user interfaces also the code (or) program

- Code testing:
  test the code for it's ability to execute properly along expected lines

Emperical Study:
Test the usability of the end product by involving end-users in the evaluation of the whole product of the SW.

Design Stage:

- There are broadly 2 issues that we need to address in our design activities.

① Where to start?
Otherwise we might end-up with faulty/in-complete design.

② How to represent (design lang)?
So that we can communicate the idea of design clearly to the other members of the team, later on. like which lang to use?
→ generally, diff stages are handled by diff. teams, since the output of one stage is input to the next one, it should be clear & precise.

①
- creative thinking.
  → as a designer, we should think creatively & come up with a design.
  → implicitly, we rely on our intution (or) past experience (or) both.
  ↳ domain knowledge

- in UCD.
  → design is not a single idea, it encompasses 2 distinct concepts
  ~~1.) Interface design~~

1.) Interface design / I & I design.
→ also includes design of interaction
i.e, I & I design

2.) System design / Code design
↳ we don't require the UPE cycle for this.

→ User POV
→ System POV

Interface Design:

- we are refering to user interface (i.e, the sys: interface through which the user gets to interact with the sys)
  → ~~interaction~~ implicitly contains int interface implicitly contains interaction
  → without interaction, interface is not of much use.
  ⇒ refering to interface means refering to both I & I.
- In interactive - system, key element is user - system interface and interaction b/w them. bcz, through these, user gets to view the sys, perceive the sys & achieve goals with the system.
- Usability primarily depends on them.

⇒ We should come up with a design that incs usability

(1) cont.
- Creative thinking depends on experience, intuition. more exp ⇒ better/easier to create a design.
- We can also make use of some guidelines/thumb-rules/ Checklists/heuristics as starting point (to aid our creative thinking)
- So, we can start with either of the both, which will aid our thinking process, to create good design, can lead to improved usability/enhanced usability.

Design guidelines:

Design Guidelines:

- Some very generic and consequently small in size - covers broad aspects of interactive systems, at a rather high level.

  eg: "Eight golden rules" (Shneiderman, 1986)

  "Seven Principles" (Norman, 1988)

→ These two originated during the proliferation of GUI (Graphical User Interfaces). GUI contains windows, icons, menus and pointers or WIMP's interfaces.

→ these guidelines are formulated based on empirical data collected over a long period of time, analysis of that data, some behavioural models are found, out of these models guidelines emerged.

i.e, they are rooted in behavioural analysis of human users.

- others are more detailed and specific and ∴ large in (set) size - intended to cover minute aspects of the design, often for specific products.

  eg: "human interface guidelines" for the apple systems.

Code Design:

- the DPE cycle keeps caters to I&I design. once I&I design stabilized and no further ite required, we focus on code design.
- In I&I design, our primary focus is to make it usable, using guidelines based on user's behavioural models.

  But, in code design those are not useful,
- In code design, primary focus is on efficiency of the implementation. and make the code "managable".
- done based on SRS.

- Two phases involved.
  1.) Preliminary (high-level) design
  2.) Detailed design (also called module-specification document)

1.) High-Level Design:
- Identification of modules (units) → together constitute the overall sys
- Control relationships b/w the modules.
  define relationships b/w the modules, control relatn b/w the modules,
  how the modules are linked to each other.
- Definition of interfaces b/w modules.

2.) Detailed Design:
In this Phase, we design individual modules, Particularly
- Identification of DS&As for different modules.

Design Language:
- Refers to specification of design (lang used for specification of design)
- diff "langs" for I&I and code design specification.

Interface Design Lang:
- to express I&I design ideas, we use "prototypes".
  Prototypes — scaled modules models of the design.
  Creating prototype ⇒ expressing the design.
  → Can be considered as lang for I&I design.
  → not really a lang/formal lang.

Code Design Lang:
- various ways are possible.
  → Natural lang. (although it'll lead to ambiguity)
  → Semi-formal langs.
  → Formal lang. (clearly defined syntax & semantics, mathematically defined)

lang - can cause ambiguity.

Can be technically used, but are not suitable or preferable to express design lang. They also change from person to person (subjective nature)

- Formal lang - best way to express

but, very rigorous and require mathematical background, good skills in understanding the particular conventions & notations and to understand the logical deductions that follow from the specifications.

- Semi-formal lang — middle ground b/w both.

is natural lang.

Partly graphical lang & rather informal way of expressing things and easier to understand but also reduces ambiguity.

→ Two such langs are

1.) DFD - data flow diagrams.

Graphical lang to express design in the form of images.

2.) UML - Unified Modeling Lang.

Partly graphical lang, express in form of some schematic diagrams.

Shneiderman's golden guidelines:

Applicable for design of GUI

- 8 rules
- originated for GUIs
- partly (or fully) applicable for other types of interfaces as well.

→ coz, it gives a broad picture, and not detailed guidelines. Can be used generic.

rule ①: Strive for Consistency

- Internal · con
- Ext · consis

Int · con => whatever symbols, metaphors, icons, texts, or tasks that we are using for one part should be consistently

used for other parts of the sys

- Ext. con => we may use some standards in our design.

  Whatever we see/experience in day-to-day life should be consistent with whatever we are defining for our sys.

  eg: Traffic signals

- For an interface to be usable, it should follow both intl & ext cons.

rule (2): Design for universal usability.

Our design should cater to diff types of users

→ doesn't mean it should be used by everyone
  it violates definition of usability.

→ Basically, it means diff user class, sees diff interfaces acc to their preferences & not the same for all.

- For novice, intermittant, expert. (novice → first time users)

  eg: Menu & hot keys for saving a file.

rule (3): Offer informative feedback.

whenever the user performs some activity with the interface

eg: progress bar (progress of the task)

eg: Color change of the floppy (storage metaphor) after the saving operation is done

rule (4): Design dialogues to yield closure.

apart from feedback there should be some dialogues b/w user & interface, to take the user wherever towards the closure of operation /to guide the us

- related to prev. rule
- organise activities into groups – beginning, middle, end.
- some feedback at the end of each group.

eg: Online shopping (lots of subtasks, grouping helps).

rule (5): Offer error prevention and simple error handling.

Objective is to keep error rates low.

- design to keep error rates low
  (eg: Close & Start operations shouldn't be kept closer to each other)
- Don't show complicated msgs
  - preferably natural lang / easier to understand msgs.

rule (6): Permit easy reversal of actions

Kinda continuation to rule 5.

- when some mistake happens, should be easier to reverse the action
- undo & redo.

rule (7): Keep users in control.

- Let the user feel that they are in control.

for this • user should perceive their interaction & change in the sys state

eg: Drag & drop a file into a folder.

this type of visualisation, helps feel user, that they're in control.

rule (8): Reduce short-term mem. load:

for the interaction to be smooth. the knowledge to keep for short-term should be less.

- Gorge A Miller (1956) — "7 ± 2" rule

It says that our short-term mem can remember b/w 5 to 9 pieces of info.

- design should not force users to remember too many things.

- We should start with our intution, and check if these rules violates, if they do, modify them ....

- Predictive interface

Norman's Principles :

- Based on a descriptive model of human-computer interaction.
- this model is ,

Norman's Model of Interaction : (1988)

- to represent interaction with GUI
- represent the behaviour of user of interactive systems in terms of a series of "actions".
- These activities represent the cognitive and sensory-motor behaviour.
  - ↳ behaviour that goes in the mind
  - ↳ using sensory organs, motor organs (which can move)
- there are 7 such actions.
- interaction has two stages (any interaction happens as a cycle : Execution-Evaluation cycle : if evaluation matches the goal, the cycle stops, otherwise it continues till achieved the goal.)
  1.) Execution Stage
  2.) Evaluation stage.

Stage	Action
1) Execution	1.) establish goal
	2.) Formulate intention
	3.) Specify action at the interface
	4.) Execute the action
2.) Evaluation	5.) Percieve sys state
	6.) Interpret sys state
	7.) Evaluate sys w.r.t goal

3.) We translate our intention to sequence of real-word tasks.

eg : tasks required to select a button on screen are :

grab the mouse, drag the pointer to the button, Press the mouse button.

- to the user, interface & system are synonymous.

5.) look at the screen, after selecting the button.

6.) we try to make sense of the sensory input.

- This model helped to develop two powerful concepts.

① The gulf of execution:

actions we specify to translate intentions maynot be supported by the interface leading to gap (gulf) b/w the first three and the last actions in the execution stage.

② The gulf of evaluation:

our interpretation of the interface state, based on our perception, maynot match with the actual state of interface (pointing to a gap or gulf b/w the first two and the last actions in the feedback stage.)

①, ② are used in analysing GUI and other interface designs and identifying design flaws.

→ particularly in the content of errors during interaction.

(errors can be explained in terms of "slips and mistakes")

Human Error: Slips and Mistakes

- Slip

Slip occurs when the user

→ understand sys and goal

→ Correct formulation of action

→ Incorrect action

- Mistake

happens when the

- user unable to establish the goal itself.

Fixing them:

- Slip – better interface design.
- Mistake – better understanding of the design.

(make user understand the design better)

Norman's 7 principles:

These are the outcome of the model of interaction.

(1) user should "use both knowledge in the world and knowledge in the head" and the sys should support this

(2) any task performed by the user should have a simple structure.
"Simplify the structure of tasks".

(3) "Make things visible: bridge the gulfs of execution & evaluation"
whatever the interface does should be visible to the user

(4) "Get the mappings right".
mapping from intention to actual physical action.
design such a way that it's easier for user to map.

(5) "Exploit the power of constraints, both natural and artificial".
designer should be aware of the limit of interaction and can impose constraints in the interaction and interface elements to make it easier for the user to understand, comprehend and use the sy

(6) "Design for error"
error handling.

(7) "When all else fails, standardize".
If it's not behaving the way it is expected, we can standardize all the interactions & interface elements. So that it is enforced on everybody to use the way it is recommended.
So, flexibility is minized by standardization.

- For testing if our intuition is right or not, we need not fully implement it, instead we can do a light-weight implement. This is called a prototype for the design idea
- i.e, Prototyping can be used to get feedback on our idea, if it is going to work in practice or not.
- In UCD, Prototyping - very imp
- these are build typically at a very early stage of devdopment / Le cycle.
- Once its purpose is served, it can be discarded.
- But, it can be incrementally refined (and tested) as we Progress in the development till the end, at which point it becomes the fully-implemented product in itself.
  This is the best case.

Prototype Categories:

① **Horizontal**: entire interface is depicted at the surface level without any functionality.

→ Interactions are not prototyped.

→ No real work can be done.

→ Suitable to discuss, brain-storm or elicit feedback on the interface look and feel primarily.

② **Vertical**: designed to represent interaction.

→ Few selected features implemented in-depth, starting from the first screen to the screen after the last action is performed (fully implem them.)

→ Suitable for analysis of interactions and features.

How to create prototypes:

Three types of prototypes, basing on how they're created.

(1) Low fidelity:

- No technological intervention - made with paper, reeds, clay etc.
- eg: paper mock-ups for interface feel, look and even functionality
- Quick and cheap to make and modify
- Good for horizontal prototypes and used for brain-storming on alternative designs and get user response on idea.

Low fidelity prototype: Interface Sketches

- Drawings depicting major components of an interface.
- Provide way to envision appearance of the interface
  → Can be done using computer, or draw on paper...
- Sketches itself are not sufficient to prototype interaction
- From the eg sketch, it is not possible to know the sequence of actions and the corresponding changes on the interface to execute a task (a "stroller" purchase" in this case).
- Sketches usually represent horizontal prototypes
- For vertical prototype, instead of a single sketch, use a series of sketches, often called "Storyboarding" to represent interaction.
  Each sketch — a "key frame".
  — "snapshots" of the interaction at a particular time step

(2) Medium fidelity:

Similiar to low fidelity, except that these are created using computer, instead of pen&paper.

- Sketches, storyboards created using computer tools: eg.
- If no functionality - horizontal, if some — Vertical. both can be done using media-flow.

- Can include videos
- eg: an animation video (created with Adobe Flash tool, for eg) to prototype the food selection app (swiggy food app).
  → In the video, simple controls can be provided to simulate the interaction.
- Interaction can be simulated with as simple a tool as,
  - PPT Slideshow.
  → Key frames – Slides
  → Simple controls (eg: timers or key press), slide transition takes place depicting the interaction.
  → In this way storyboard can be converted to a vertical prototype
- MF is just implementing the same idea as LF, but with computer tools & usage.

(3) Hi-Fidelity :

Prototypes created with computer programs (eg: actual SW development)

- more sophisticated, require much more effort (including expertise in programming) than the MF & LF.
- There are many interface builders; "toolkits" and "wizards" to ease the programming efforts.

eg: Tcl/Tk toolkit, Visual Basic Programming Language, JAVA Swing Library .....

→ The toolkits and libs provide support for "widgets" (GUI elements)
- a programmer can directly use those (rather than creating his/her own) and build GUIs.

- The interaction is implemented through programming.

- mainly used to create Vertical prototypes
- "Wizard of O_2" ~~an~~ approach - a prototyping technique
  → In this technique, a human subject believes to interact with an autonomous computer
  → In reality, however, the computer is operated by an "unseen" human being (the "Wizard").
  ~~→ The term originated in~~.

How to use prototypes:

① Thrown away - LF — if not expensive
② Incremental approach, the sys is designed into units (modules)
  → each unit is seperately prototyped and tested.
  → afterwards, it is integrated into the system.
③ Evolutionary approach: the whole sys is prototyped and tested
  → based on testing, prototype is altered.
  → eventually, it becomes the final product.
  → most typical.

Prototype Evaluation:

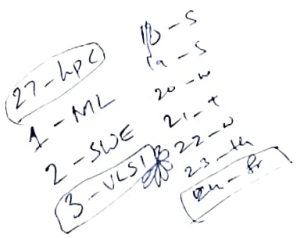
- how to evaluate?
  Ans: By evaluating usability.
- Usability is tested in empirical study also, at the end, by end users, but it's very costly and need more efforts, can't be done ~~often~~ more than once(or)twice. But, Prototype evaluation can be done as many times as we want. this is also quicker.
- expert evaluation:
  Used for quick & cheap evaluation (typically at early stages).

Expert Evaluation: (done by experts)

- Used for quick & cheap evaluation of prototypes for usability (typically at the early stages of design)
- We need 2 things (requirements)
  - → atleast a low-fidelity prototype
  - → An evaluation team
    - maybe the design team, may include other skilled designers
    - Optionally, few end users
    - atleast 3-5 members

How it is done:

- Each team member evaluates individually and produces a report.
  - report contains a list of usability issues
- All these reports combined to produce a final list.
- There are several ways to implement EE.
  two of them are:



① Cognitive Walkthrough:

- An usability Inspection method.
- requirements:
  - → atleast a LF prototype, with support for several tasks (vertical prototype) [actually more than one prototype needed]
  - → 3-5 members as team.

Additional requirements - Questions:

Purpose - identify problems that user likely to face.
Evaluators not expected to ans only "yes" or "no".
→ Should give a detailed report on what they felt about the I&I w.r.t each question. Reasoning should be given.

Feedbacks are analyzed to identify broader usability issues.

Heuristic Evaluation

- Cognitive walkthrough useful in the early stages of design
- Problems: method is scenario-based
  - → We evaluate wrt specific usage scenario
  - → Complex systems likely to be numerous usage scenarios
  - → Can't evaluate wrt all.

  reason: ① we may not have that much time
  more likely ② we may not even be able to enumerate all possible usage scenarios

- We work with representative use cases.

  even if there are large no. of use cases, in real-life it is not necessary that all the use cases are frequently performed. Most likely a very small subset are frequently performed, we have to identify that set.
  - → It is not easy to identify this set.

heuristic evaluation:

- We may go for comprehensive evaluation of the whole system
  - → We no longer need to create scenarios and ask evaluators to perform tasks.
  - → Instead, we can ask evaluators to tick on a checklist of features of the system as a whole.
- This is heuristic evaluation — evaluate a system with a checklist
  - → Items in the checklists are called heuristics

- There are different kinds of checklists
  - → Some are quite detailed and system specific
  - → Some are more focused on broader principles of usability.
    - Here, no. of heuristics are fewer/lesser.
- eg: 10 heuristics by Nielsen [Nielsen, 1994].

- Requirements:
  - → LF prototypes – horizontal prototype enough (102, execution sequence not required, we don't deal with tasks here)
  - → 3-5 members
- Only end-users, not good enough, expert feedback is needed
- Evaluation process slightly different
  - → Task scenario not required – each evaluator checks design w.r.t heuristics and report their findings.
  - → Reports are combined to determine heuristics that are violated.
  - reporting is not just yes/no. (like cog. walkthrough)

User Evaluation:

A way of evaluating prototypes rapidly.

- involve end-users
- With expert evaluation, we rely on data provided by those who are not likely to be the users of the system
  - → this is allright in early phases
  - → We'll get more insights from data collected from the users themselves
    - but we should do this in a rapid way.
- One way is to conduct empirical studies
  - involves elaborate setup and expertise to collect and analyse data.
- Another method (less complex) – get feedback from users by asking them Qs directly.
  - this is called evaluation with self-reports.
  - data collected in this method sometimes called subjective or preference data.

Evaluation with self reports:

- Many ways to collect user feedback
  - → rating on a scale (eg: Likert scale)
  - → Can ask them to choose from a list of sys. attrs.
  - → ask direct, open-ended Qs (eg: What do you think about this?)
- problems with open-ended Qs
  - feedback may not be reliable
- Feedback on rating scales are generally considered more eff & reliable
  - → pre-requisite — identify items (commonly refered to as a questionnaire though it need not contain Qs).
  - → Challanging to identify suitable questionnaire to collect the feedback
  - → When we ask for feedback, we assume user is already exposed to the system.
    - He/She might be asked to use the system before feedback, like in a controlled study.
    - In such studies, users are typically asked to perform a set of tasks.
- We have two points for data collection.
  - → After every task.
  - → At the end of the session (after all the tasks)
  - → desirable to collect at both the points.
  - → corresponding questionnaire maynot be the same.

System Design

- The design - Prototype evaluate cycle primarily caters to interface design.
- Once interface design stabilized (no evaluation done) and no further ites are required, we focus on sys. design. i.e., two
  → Converting interface design into sysdesign
- Two issues
  → where to start
  → how to represent (design tools) ?
- In ID, main concern is how to make it more interactive & percept
  In SysD, " " is to make it managable, it's implementat
- Generally, Projects done in teams, it is divided into units, then the units are combined together to create the whole system.
  i.e, a modular design.
- When designing a system, it should be done in a modular way so that, when it is converted into code, modular development is possible
- Our main objective is to help in managing the code development well and help in making the code manageable.

Where to start :

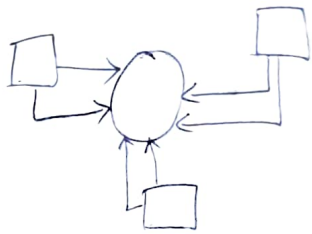
- Start with SRS
- Two phases in designing system
  (1) Preliminary → high level design (modular level)
  (2) Detailed design → minute details of the system modules.
  ↳ also known as "module - specification document".

(1) High-Level design:

- Identification of modules
- try to find out Control relationships between modules
- provide Definition of interfaces between modules
  (→ like modules need to interact with other, how they interact, what kind of interfaces should be there ..

How to represent: there are many lang/ways.

- an example: DFD: Data Flow Diagram.
- graphical notations – to express the design idea.
- DFD – graphical lang.



- ○ → represents the whole sys. (top level representation)
- □s → who all are interacting with the sys.
  eg: users
  administrator
  other users.
- → → represents what data flows, in what direction. also, what that particular user can do (while interacting).

- level 2 diagram
  the circle can be further divided into modules, □ are used to repr...
- we can even show/represent databases
- we can go more lower levels, represent a particular module seperately.
  → Conventions will be same tho.
- There are other ways too.

2) Detailed Design Phase:

- Identify DS & A required for implementing different modules.

Characteristics of Good Design:

After getting a design, we should evaluate it. So, we specify Characters of a good design.

1) Coverage: this characteristic tells that a good design
   - Should implement All functionalities in the SRS.
2) Correctness: Should correctly implement all functions in SRS.
3) Understandability: should be easily understandable (by other team members)
4) Efficiency: ~~Should be made in a way s.t, it can be implemented~~ Should be efficient
   (in terms of resources required to implement)
   it Shouldn't require things that are not available.
   → When we're going to design something, we should be aware of the resources available and accordingly we should design. So that those resources are used optimally.
5) Maintainability: Should be amenable to change.
   → ~~less~~ if we want to change something, we should be able to do that easily, without doing the design from scratch.
   → i.e, Changes can be performed at a local level without having to redo everything.

- If these characteristics are not present, it should be re-designed so that these five characteristics are maintained.

Cohesion & Coupling:

- Good software design requires
  - Clean decomposition of the Problem into modules. (modular design)
  - Neat arrangement of modules in a hierarchy.
- Modularization depends on Cohesion & Coupling.

Cohesion:

- property of a module (of one module).
- several things determine the cohesive-ness of a module, there are several types of cohesion. (eg: logical cohesion).
- ① Logical cohesion:
  If all the funcs performs similiar operations (eg: error handling)
- ② Temporal cohesion:
  if all the funcs of a module should be performed in the same time span (eg: initialisation module - whenever sys starts)
- ③ Procedural / Functional Cohesion:
  if all funcs of a module are part of the same procedure (algo). eg: decoding algo
- ④ Communication:
  if all funcs refer to or update same DS
  eg: a set of funcs operating on a LL.
- ⑤ Sequential:
  if output from one element is input to the next element of the module.
  eg: the sequence of funcs get-input, validate-input, sort-input.
- module is a collection of funcs, cohesion say about the behaviour of these funcs (like their execution, communication ....).

Coupling:

- A property b/w modules (2 or more)
- Types

① Data coupling:
If two modules communicate through a data item.
eg: passing an integer b/w two modules

② Control coupling:
If data from one module is used to control the flow of in... in the other module (eg: flag setting)

③ Content:
If two modules share code
eg: branch from one module to another.

★★★ High Cohesion & Low Coupling → functionally independent modules.

- Our objective is to have clean, neat decomposition of sys design into modules & should be able to represent them in a hierarchical manner.
  → this is possible if we have functionally independent modules
  ⇒ High cohesion within modules & Low coupling b/w modules should be satisfied for a high-quality modular design

Basic Code Design Approaches:

- Broadly two approaches
  ① Function oriented app:
  - we use funcs as basic abstractions.
    i.e, whole sys is designed based on funcs.
  - use DFD to represent design

2. Object oriented:

Objects are the basic abstractions
↳ instantiation of class.

- use UML (unified modeling lang) to represent design.

DFD:

- used for functional design approach.
- it's a graphical lang.

Basics:

- represents "flow of data" through a process or a sys.
  i.e, focus is on data "movement" b/w external entities ① and processes ②, and between processes and Data Stores ③

  → ①, ②, ③ Concepts are used to express the design, through DFD.

- DFD provides overview of:
  → data processed by a sys. (diff types of data that a sys processes)
  → Transformations that are performed on this data. (by some processes)
  → ~~idea about~~ which data are stored (and which are not)
  → What results are produced, where they go.

- Graphical nature of DFD makes it a good communication tool between ① User and (system) designer
  ② Designer and developer.

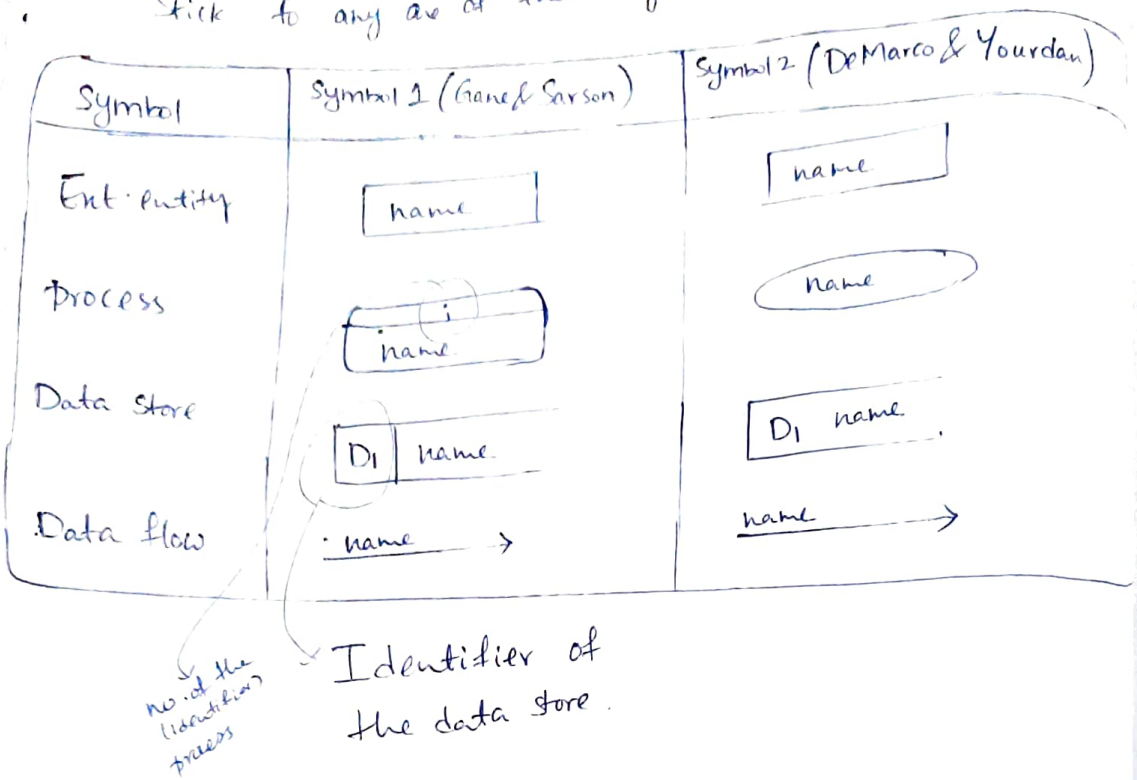
Components of DFD:

- Source/sinks (external entities)
- processes
- Data Stores

} primary components of any DFD.

★ Data flows is also considered as a component of DFD.

- there are two ways of notations.
- Stick to any one of them only. don't mix them up



- DFD naming guidelines
  - Process → verb phrase
  - Data Flow Label → noun (as the name of data.)
  - Data Store → noun
  - Ext. entity → noun

① Ext. entity:

- people or organizations that send data into or receive data from system.
- They either supply or receive data
  - Source → supplies data to sys.
  - Sink → receives data from sys.

② Process:

- A series of actions that transform data.
- Notation:

- notation:
  - St. line with incoming arrow => input data flow
  - " " outgoing " => output data flow
  - Labels are assigned to data flow.

(3) Data Store:

- represent permanent data used by system.

  Some data – transient in nature => destroyed after usage

  Some data – permanent – remains for use in subsequent iteration of the processes.

- notation:
  - data written into data stores => incoming arrow
  - " read " => outgoing arrow.

(4) Data Flow:

- depicts actual flow of data b/w elements (①, ②, ③).

  → connects ①, ②, & ③.

- notation: an arrow with label.

  generally, arrow – unidirectional.

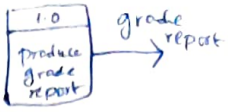
  but, it can be bi-directional (double-headed arrow)

  if data flows in both directions.

Rules for creating DFD:

- Data can flow from
  - → entient to process
  - → process to entient
  - → process to store & back.
  - → process to process.
- Data can never flow from
  - → entient to entient
  - → entient to store
  - → store to entient
  - → store to store.

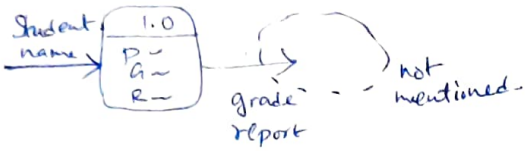
- 3 incorrect flows:



① Miracle : no input given, but gives an output

② Black hole : input given, output is not produced

③ Gray hole : input given & output produced, but not mentioned where the output data goes.



(Sink is not specified)

These are common ones, there are other incorrect ways too.

Representing Complex Designs:

- Real-life systems can be very big
  - involving large no. of processes and data stores
- Representing such systems with DFD difficult.
  - very complex diagram.
  - difficult to understand.
  - difficult to modify. (Large portions need to be modified.)
- use decomposition
  - Create hierarchy of "levels".
- Atleast 3 levels needed.
  - level 0 → Content diagram
  - level 1 → Overview diagram
  - level 2 → detailed diagram.

Level 0 (Content Diagram):

- Contains only one process
  - represents entire system
- Data arrows show input & output
  - (from/to ext. entities)
- Data stores not shown - implicitly contained within system
- Diagram basically shows the sys and the ext. entities along with the data flows.

  One process → system

  and no data stores, other processes.

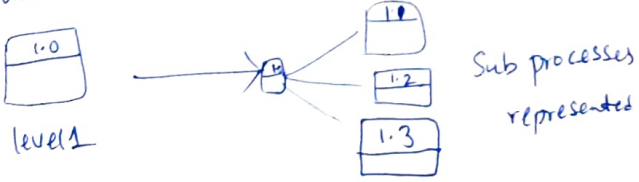
Level 1 (Overview diagram):

- utilizes all four elements
  → data stores first shown at this level.
- data flow b/w the processes, ext entities and data stores.
- all four components can be used.

Level 2 (Detailed diagram):

- Detailed representation of processes that are depicted in level 1.
- like level 1, utilizes all four elements.
- all level 1 processes are further decomposed & shown in level 2
- detailed design of individual processes (or) modules.
- if there are four processes, ⇒ 4 diagrams

eg:



Creating good DFD:

- use meaningful names for dataflows, processes & data stores
- use top-down development starting from context diagram and successively levelling DFD.
  - → don't use bottom-up.
  - → first understand all processes then, try to create hierarchy
- Stop decomposition, if levels become trivial (no significant change)
- remember – only previously stored data can be read
- remember – data stores can't create new data.

- DFD contains data stores.
  but, doesn't tell us, how to represent data.
  - → the notation doesn't reveal int. structure of the data
  - → but, it is useful to know the int. struc of data.
  - → for this we use ER diagram.
    - = Entity–Relationship diagram.
- Labels don't reveal "structure" and "organisation" of data stores.
- there maybe relationships b/w various data elements, not revealed by simple labels.

→ One way to express rich int. struc, organisation and relationships in data stores is to use ERD.

ERD:

- includes entities and relationships.
- most common representation for relational databases.

→ Can be considered as another graphical lang to represent data just like how DFD → design of sys.

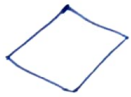
Basic components:

- Entity: an identifiable object/concept of significance.

  notation: 

- Attribute: property of an entity or relationship.

  not: 

- Relationship: an association b/w entities. 

- Data store (database) can be modeled as:
  - A collection of entities.
  - Relationship among entities.

Entity:

- An object that exists and is distinguishable from other objects

  eg: person, company, student, ...

- have attributes

  eg: person has name, age ..

- Entity set: Set of same type entities that share same properties..

  Eg: Set of all persons.

- each entity must be uniquly identifiable.

Attributes :

- Descriptive properties of entity.
- A particular instance of an att is a value.
- Domain – set of permitted values of an attributes
- There are two types
  1. Simple :– Contains only atomic values.
  2. Multi-Valued :– Contains several atomic values

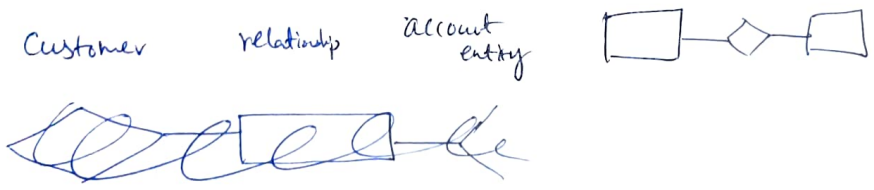
entity : Student
att 1 : id : 2001010048
att 2 : ph.no ← ph.no 1, ph.no 2 } multi-value.

- null att ⇒ when an entity doesn't have value for an att
- derived att ⇒ derived from other atts or entities.
  eg: age from DOB (an att)

Relationship :

- an association b/w entities.

eg: Sam depositor E-100



- degree of relationship : no. of entities in a relationship