Introduction

Requirement is either

1. Something user needs to achieve obj.
2. Something system needs to meet contract
3. Something documented

} Not all stakeholders share same reqs.

Requirements engineering:

1. What should be solved
2. Why it should be solved
3. Who should be involved

⇒ Many software failures stem from these reqs.

System-as-is → (requirements) → System-to-be

This is the 1st part of software eng.

General process:

1. Domain understanding & elicitation: understand & compile reqs.
2. Evaluation & agreement: meet w/ stakeholders about reqs.
3. Specification & documentation: write it down (SRS = spec. req. specification)
4. Validation & verification: write & validate code w/ reqs.

Requirements

- Functional
  - Smth. sys. should do
    - I/O
    - Storage
    - Comp.
    - Timing
    - Other
- Non-functional
  - Anything not func.
  - "Quality reqs"

ELICITATION

Defn: get info about problem, reqs. & environment of system

Stakeholder: someone/smth. affected/can influence outcome of sys.

↳ Customer: gets direct/indirect benefit from process

Business analyst is usually point person for stakeholder mgmt.

Process:

① Decide on session scope

② Prepare questions

③ Perform session

④ Organize & share session output

⑤ Document open issues

Elicitation techniques:

① Interview

② Workshop

③ Focus group: take rep. sample of users

④ Observations of users

⑤ Questionnaires: especially for large groups, but needs good q's

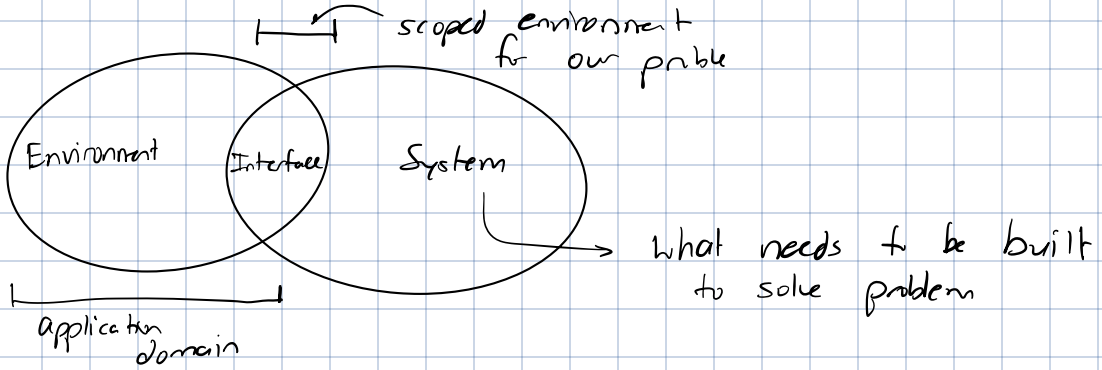
⑥ System interface analysis: look @ system each. of data

⑦ User interface analysis: look @ existing systems

⑧ Document analysis

Done elicitation if no new requirements are generated w/in scope

REQUIREMENTS ENGINEERING REFERENCE MODEL



Requirements defn: desired properties expressed in terms of environment

Specifications defn: description of sys. & what is should do (NOT how)

Scoping environment ⇒ much easier to define reqs & specs

Ex:// Create reqs & specs for turnstile

Reqs:

① Collect fee from user

② Only paid people can use turnstile

③ Prevent non-paid from entering

Specs:

① Sense fee paying

② Unlock barrier to allow paid entry

③ Lock barrier after allowing people through

Domain knowledge: properties of env. assumed true

Fundamental law of requirements:

Domain knowledge, specifications ⊨ (imply) requirements

How to derive specs:

① Determine how system will monitor & control env.

② Determine which domain assumptions necessary for constraining env.

③ Check dom, specs ⊨ reqs.

Ex:// Determine if dom, specs ⊨ reqs.

R: allow car traffic to cross intersection w/out collisions

D: drivers behave legally & cars function well

S: traffic light guarantees that perp. dir. do not show green & yellow @ same time

Make argument that D, S ⊨ req

DOMAIN MODEL

Defn: model of environment of system

Represented by UML diagram, but only representing data entities w/ no ops.

Conceptual objects

Defn: set of instances of domain manip. by system

Properties:

① Distinctly identifiable

② Can be enumerated

③ Share similar features

④ States can differ b/w each instance

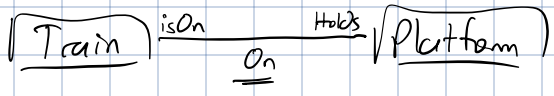
Types:

① Agent: active & autonomous
↳ UML class, eg:// TrainController, TrainDriver

② Entity: passive & autonomous
↳ UML class, eg:// Book, Train

③ Event: instantaneous object
↳ Only has 1 state
↳ UML class

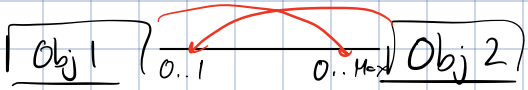
④ Association: object dependent on another object
↳ Conceptual links b/w ①, ②, ③
↳ UML association
↳ Reflexive: can appear under diff. roles



↳ Often tuple of instances eg:// On(trl, plt 2)
↳ Arity can be def. by UML

UML for Specs & Reqs

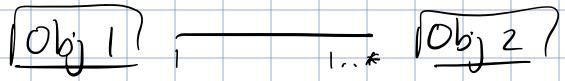
① Association multiplicities

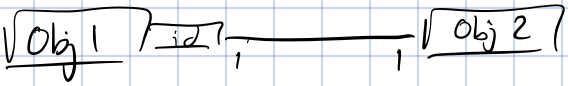


If min = 0 ⇒ optional assoc.
" = 1 ⇒ mandatory "
max = 1 ⇒ uniqueness

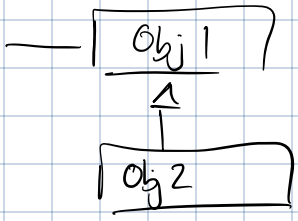
② Association qualifiers

Defn: unique assoc. to distinguish b/w set of linked instances

Before: 

After: 

③ Specialization

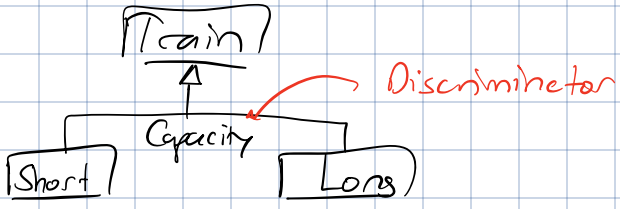


⇒ Obj 2 inherits associations & features, but specific features from Obj 2 overrides from Obj. 1
Obj 1 is generalization

Object can be specialized from multiple super-obj.

If multiple objects descending from an obj, can add a discriminator to indicate how they are diff.

Ex: //

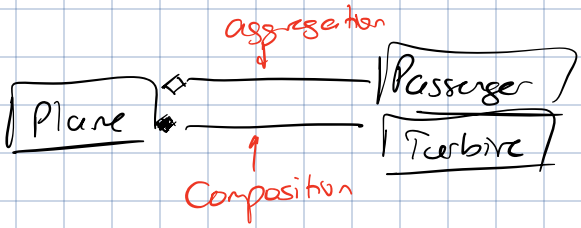


④ Aggregation / composition

Aggregation: object weakly belongs to several containers

Composition: " strongly " at most 1 "

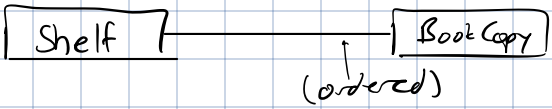
Ex: //



Whole should be left of part

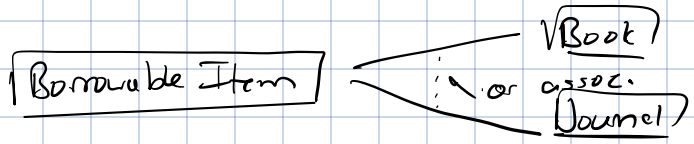
⑤ Ordered association

Multiple target instances from source instance are ordered



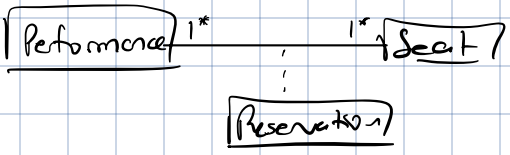
⑥ OR association

Some role can be played by multiple objects



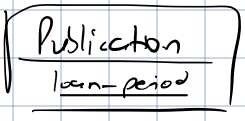
⑦ Association of associations

Object is an association



⑧ Class scope attribute

An attribute whose value is shared by all objects



Tips

Q: Should X be an object or an attribute?

Make X an attribute if:

① X is a function

② Instances of X do not need to be distinguished

③ No associations w/ X

Aim for parsimony

Include multiplicities as much as you can

We don't care about:

① Class methods

② Ctors / dtors

③ Attribute values

④ Method visibility (public, private, ...)

Use Cases & Scenarios

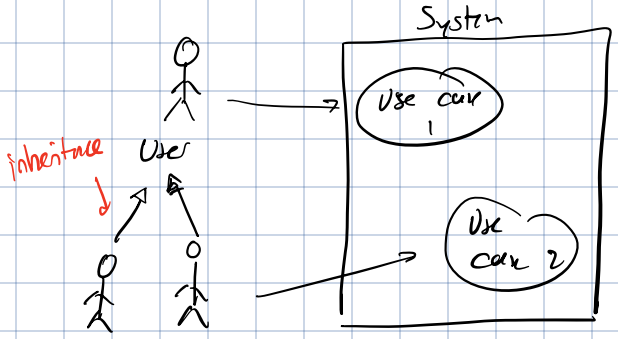
Use cases

Use case defn: a particular way to use the system by users/actors

↳ This is not the same as a scenario. Scenario is a sequence of interaction steps b/w user & system in a single use case

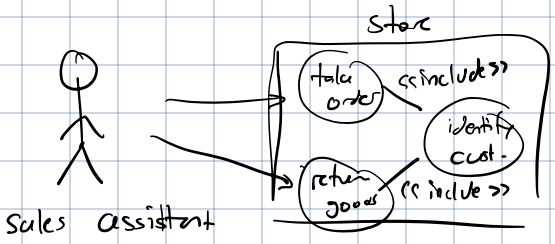
Remember that actors act on system, but stakeholder is just interested. Actors ⊆ stake

Diagram:



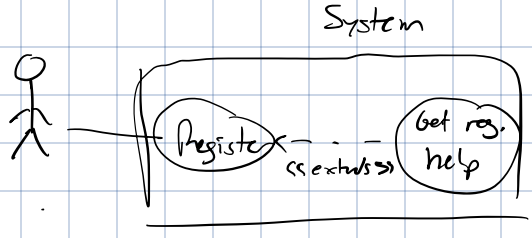
If a use case is part of multiple use cases, extend & <<include>>

↳ Ex:



If a use case extends another use case, use <<extend>>

↳ Ex:



Include & extend dependencies: include/extend imply that all associated use cases will be triggered. If that is not true, inherit rather than include/extend

Traps to avoid:

① Too many use cases

② Highly complex use cases

③ Including sys. design

④ Incl. data defn.

⑤ Incl. use cases users do not understand

Scenarios

Defn: one full execution path through a use case listing only observable actions of system & actors
↳ 1 use case → many scenarios

Structure:

Name:	ID:
Authors:	
Goal:	
Trigger:	
Preconditions:	
Notes:	
Actor 1 ...	Actor n
1. Action 2. Action 3. Action 4. Action ⋮	
Alternative 1: <descr.> Steps: a-b a. . . . ⋮ b. . . .	
Exception 1: <descr.> Steps: c c. . . .	

Main scenario

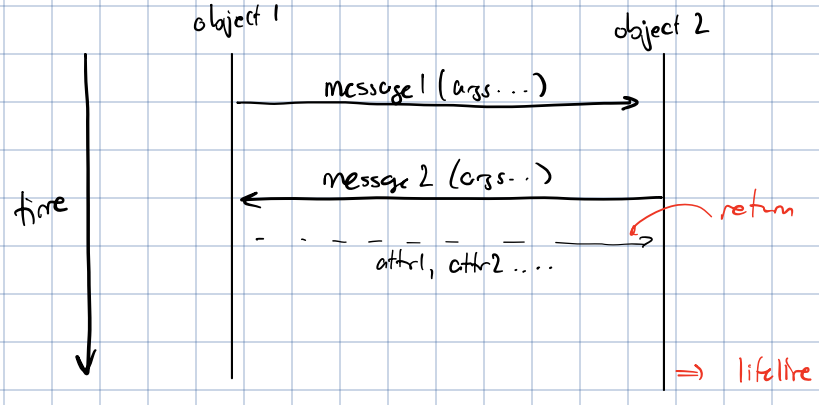
Alt. scenario: achieves goal of use case but through diff. steps/actions

Exception: special case

SEQUENCE DIAGRAMS

Defn: step-by-step explanation of a scenario. 1 scenario → multiple sequence diagrams

Systems are considered black boxes, mostly focused on events generated by actors in sys.



Specific:

① : object : emphasizes that "object" is an instance

② box around any messages encloses iteration area

- loop: iteration (must include guard stmt.)
- opt: execute if guard == true
- alt: cascading if stmt, execute 1st body w/ guard == true
- par: parallel execution

} Don't use too often.

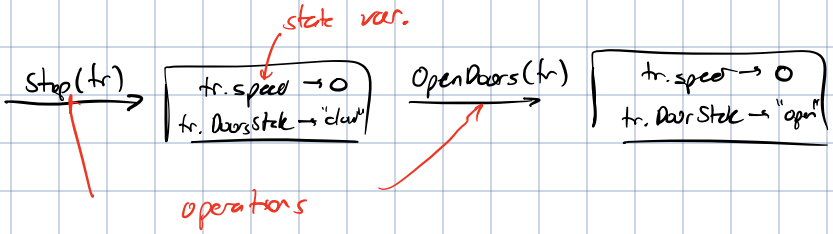
Can include subdiagrams & refer to it via ref / sd

FUNCTIONAL MODELLING

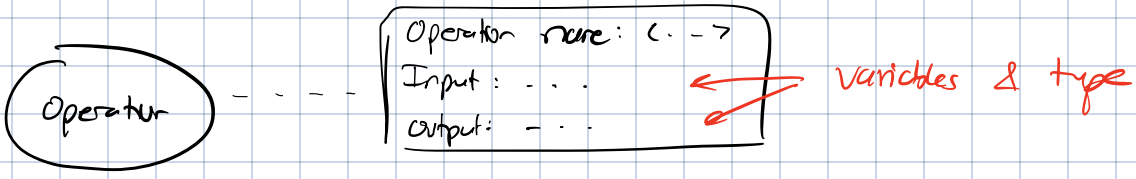
Operation: set of input-output pairs
- Input: state affects application
- Output: state changed by application

} Generally deterministic, atomic (instantaneous) & can be concurrent w/ other ops.

Ex:



Structure:



- DomPre: cond. on inputs for operation (eg. DomPre of OpenDoor: tr.DoorState = "closed")
- DomPost: " outputs " (eg. DomPost of OpenDoor: tr.DoorState = "open")

Agents: perform op. w/ all input & output of op. controlled/monitored by agent
- Op. can only be performed by uniquely 1 agent

Can include exceptions

OCL: OBJECT CONSTRAINT LANGUAGE

Expresses UML constraints but not part of UML notations

Structure:

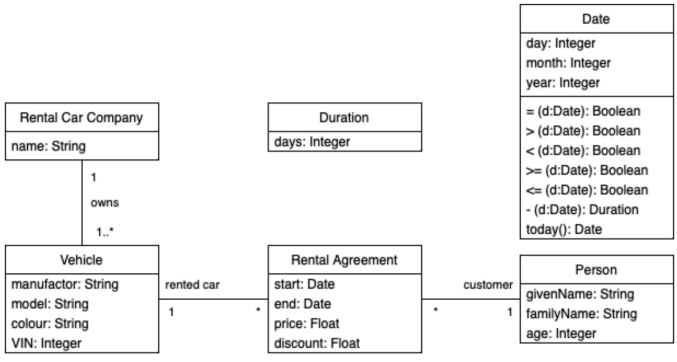
context <UML object> inv:
self. ... → ... : attr (eg. size()) <cond>
↳ variables/assoc.

Filtering state:
- self. ... → select (<attr.> = "...") => selects state when attr. = ...
- " reject (- - -) => " !=...

Exists:
- Ensure that one value of state exists (eg: ... → exists (color = "black")

for All
- Same as logical ∀: ... → forAll (r: ... | <cond on r>)
  (r: ... ← type/object)

Ex:



Write OCL for: People aged above 65 have a 10% discount on rental agreements created in / after 2024

context Person inv:
self.age >= 65 implies (self.RentalAgreement → select (start.year >= 2024) → forAll (discount = 10))

context RentalAgreement inv:
self.customer.age >= 65 implies ((self.start.year >= 2024) implies (self.discount = 10))

Nothing in OCL about recovery from violated invariants

BEHAVIOURAL MODELLING

Goal: model system behaviour as a whole as it appears to outside actors.

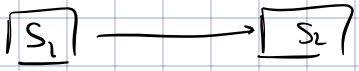
State-driven behaviour: object's behaviour can be divided into disjoint sets

UML state diagrams

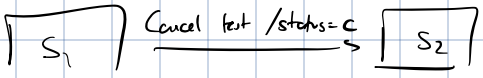
Shows state & behaviour of 1 object throughout lifetime via finite state machine
↳ descr. of data in obj. In FSM, only include state that obj. see & influence

Transition: Event [cond.] / action ← behaviour executed (optional)
↓ what triggers transition
↳ precond. bool.

- Should be deterministic & clear (should be clear which transition to take on event)
- If transition happens automatically



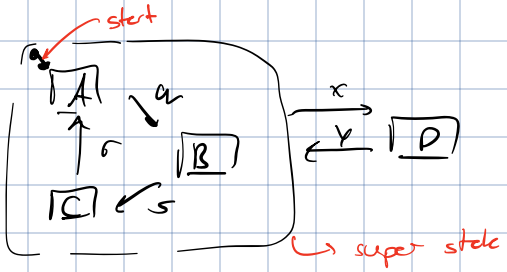
- Ex:



Internal activities: actions object does to itself
- entry / < boolean cond. > : events that ensure object stays in state
- exit / < " > : " exits state

Composite state: combine state & transitions that work toward common goal

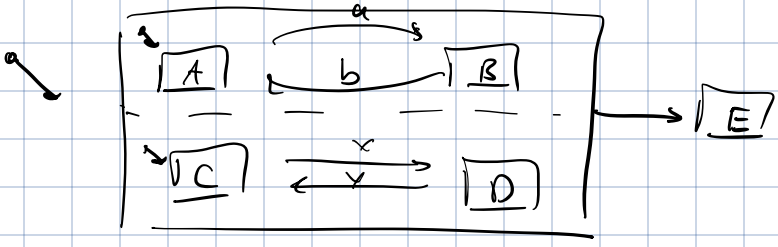
① Hierarchical state



Clusters similar states & behaviours

Transitions from superstate is from ALL descendants to start descendant (eg: A)

② Concurrent state

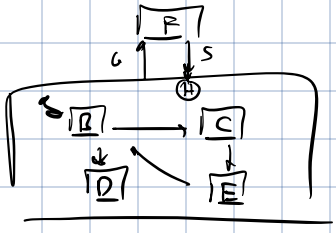


Orthogonal behaviours & states that can run @ same time

May need synchronization

History mechanism: enter most recently visited state

Ex:



⇒ If event 5 occurs:
① Go to most recent state of {B, C, D, E}
② Else, go to B

Usually applies to closed hierarchy. To go to lowest state in hierarchy, do H*

Time events: occurrence of specific date/time (notation: ... at (<time>))

Change event: condition is false → cond. is true:

All paths in scenario should be in state machines

Quality Requirements

Quality attributes distinguish products that perform functions & delightful products

Examples: performance, reliability, robustness, adaptability, security, usability, scalability, efficiency, accuracy

This often leads to constraints & requirements based off tradeoffs

Fit criteria: quantifies extent to which quality criterion is met

- Ex:// Entire network can fail no more than 5 min/yr
- Can enhance:

	Tier A	...	Tier D
Metric 1			
⋮			
Metric n			

If you cannot quantify req., it is not a req.

External quality attr.:

- Availability: $\frac{\text{uptime}}{\text{uptime} + \text{downtime}}$
- Installability: how easy it is to perform install ops.
- Integrity: prevent info loss & preserve data correctness
- Interop.: how easy system can exchange data w/ other sys. & integrate w/ hardware
- Perf.: response time, throughput, data capacity, dynamic capacity...
- Reliability: P(software executing correctly in time period)
  - ↳ Can use Monte Carlo approach to determine # of bugs
    1. Plant bugs
    2. $\frac{\#\text{ of detected errors}}{\#\text{ of planted errors}} \sim \frac{\#\text{ of detected errs}}{\#\text{ of errs in prog.}}$

} Not all errors are equal!

- Robustness: how well sys. performs if invalid cond. happen
- Safety: how well sys. respects hard boundaries
- Security: self expl.
- Usability: "

Internal quality attrs.: efficiency, modifiability, portability, reusability, scalability, verifiability

Constraints: restrictions you must adhere to → req.

REQUIREMENTS NEGOTIATION & CONFLICT MANAGEMENT

Conflicts happen when stakeholder needs & wishes contradict each other

If left unresolved, can lead to stakeholder trust ↓ & system development failure

Resolve via involving stakeholders & jointly talking about conflict scenarios
↳ Share diff. resolution scenarios

Types of conflicts:

① Data conflict: info. problem

② Interest conflict: stakeholder interests contradict

③ Value conflict: stakeholders evaluate reqs. differently

Negotiation techniques:

① Win-win scenarios

② Interaction matrix: 1 cell = pair of reqs.
- Value: $\begin{cases} 0 & \text{if } R_1 \text{ indep. of } R_2 \\ 1 & \text{conflict} \\ 1000 & \text{overlap} \end{cases}$
- Sum each col:
  ↳ if 0, non-problematic
  ↳ # of overlaps = sum /1000
  ↳ # of conflicts = sum % 1000

PRIORITIZING REQS.

Succesful prioritization requires understanding:

① Customer needs.
② Req. importance to customers
③ Timing delivery expectations
④ Req. relationships & dependencies
⑤ Which reqs. must be impl. as a group
⑥ Cost to satisfy req.

Techniques

① In or out

Simply get group of stakeholders & decide iteratively to keep/scrap req.

② Three-level scale

Eisenhove matrix:

	Imp.	Not imp.
Urgent	High pri.	scrap
Not urgt	Med pri	low pri

Can do this iteratively by breaking req down & repeating

③ MoSCoW

M: "must" do req to make sdn. succes
S: "should", essential but not mandatory req.
C: "could", nice-to-have
W: "won't", not going to be impl.

④ Cost-value approach

1. Review req. & check if they are complete
2. Compute value of each req.
   a) Take pair of req. (x, y) & score:

$$\begin{cases} 1 \rightarrow x = y \\ 3 \rightarrow x > y \\ 5 \rightarrow x \gg y \\ 7 \rightarrow x \ggg y \\ 9 \rightarrow x \gggg y \end{cases} \Rightarrow \{2, 4, 6, 8\} \text{ only if compromise needed.}$$

if $(x, y) = n$, $(y, x) = 1/n$

b) Normalize colums by dividing each entry by sum of respective col.

c) Sum each row

d) Normalize row sum

e) Report normalized values as relative importance

3. Check consistency

a) Multiply 2a matrix by 2d vector

b) Divide each elem by corr. elem in priority vector (2d)

c) Take avg.

d) $CI := \frac{avg - n}{n-1}$, n is # of req.

e) $CR = \frac{CI}{CI\ random}$ → from table, function of n

If $CR < 0.1$ → good!

4. Get relative cost using step 2

5. Plot & discuss

COST ESTIMATION

What do we need to estimate?

① Development time

② Cost

③ # of developers / month

Why is it hard to estimate? We don't know much about the project

Why even estimate?

① Basis for agreeing to job

② Make commitments

③ Tracking progress

Estimation techniques

① Delphi method

1. Expert sends secret estimate
2. Avg estimate sent to all experts
3. Expert revises secret estimate

} Repeat until no revision

② Function point analysis

1. Estimate # of function points from reqs.

$$FPs = a_1 EI + a_2 EO + a_3 EQ + a_4 EIF + a_5 ILF$$

- EI: external inputs: # of user inputs
- EO: external outputs: # of user outputs
- EQ: external inquiries: # of queries (no state change)
- EIF: external interface file: # of ext. interfaces
- ILF: internal logical files: # of internal files

$a_1, \ldots, a_5$: weights from a table

2. Estimate code size from FPs

$FPs \xrightarrow{\text{lookup table / log.}}$ code size

3. Estimate cost from code size

③ CoCoMo

Constructive cost model: predict cost from LoC

$$E = a \times KLOC^{b} \times X$$

- E: effort in man-months
- a: empirical weights, f(code size)
- $KLOC$: estimated project size
- b: empirical weights, f(code size)
- X: Project attribute multiplier: change effort estimate based on constraints

Corollaries:

1. Development time (months) $= cE^{d}$
2. People required: E/d

} $c, d = f(LoC)$, empirical weights

Cost of adding more devs is increased communication

Consider an estimate good if cost w/in 20%, time w/in 70%

Even though estimation may not be good, it gives basis for planning. We get better over time

Cross-check estimate w/ other methods to see if estimate makes sense

Validate estimate by making sure assumptions are correct

Cost estimation challenges:

① Historical data access
② Data validity
③ Limited time to estimate
④ Resources

Risk Analysis

Risk defn: uncertain factor that can negatively impact obj.

↳ Product-related risk: product cannot deliver required services

↳ Process-related risk: risk w/ process of building product

Early risk management during req. phase will save us a lot of problems

Risk types:

① Software requirement risk
② Software cost risk
③ Software scheduling risk
④ Software quality risk

Risk identification

① Risk checklists

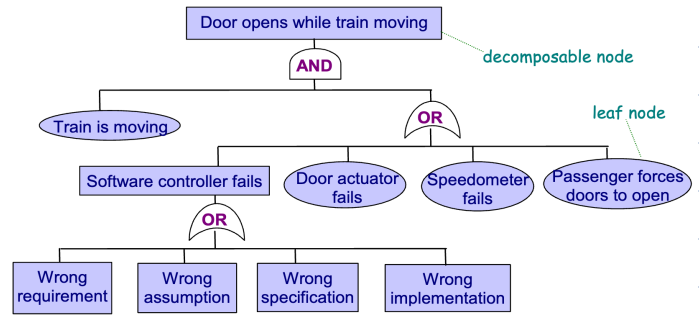
Go through risk categories → customize to proj. → work through list

② Component inspection

Go through each component & determine how it can fail

③ Risk trees

Leaf nodes are failures → connect to tree via logical operators.



Cut set: minimal AND combos of leaf nodes that can cause root node fail

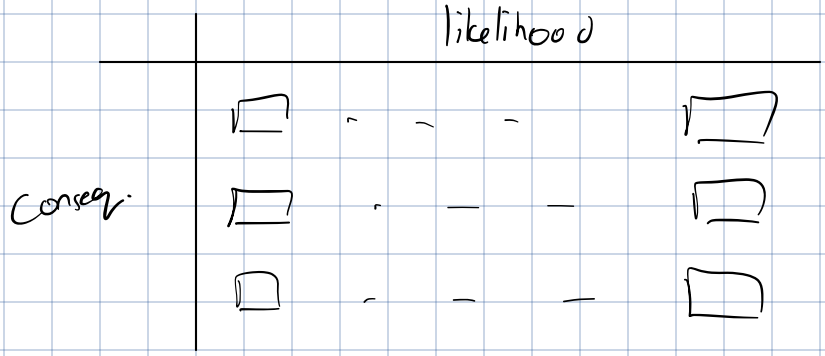
↳ Process:

1. Start w root node
2. If AND: group into 1 child node in cut set
3. If OR: seperate each child node int cut s/ child node.

Risk assessment

Mostly qualitative assessment

For each risk:



Risk control

Deploy countermeasures to reduce risk

Reduction tactics:

A: Reduce likelihood

B: Avoid risk

C: Reduce conseq. likelihood

D: Avoid conseq.

E: Mitigate conseq.

To select best countermeasure, use risk-reduction leverage:

$$RRL = \frac{e^r - e^{rlcm}}{cost(cm)}$$

e^r: exposure to risk; e^{rlcm}: exposure to risk given countermeasure; $cost(cm)$: cost of countermeasure

Make sure to document risks

Defect Detection & Prevention (DDP)

Steps:

1. Identify most critical requirements
2. Identify potential risks
3. Estimate impact of each risk on each requirement
4. Identify countermeasures
5. Identify most effective countermeasures

We use a risk consequence table for ①-③

req.	weight	risk 1	...	risk n
likelihood				
⋮	⋮	a	b	c ...
		d	e	f ...

⇒ Matrix values are put on 0-1 scale.

0 if risk happens & no impact on req.

1 if ... & req. is impossible to fulfill

$$\text{Loss Of Objective}(req) = weight(req) \times \sum_{risk} impactScore(req, risk) \cdot likelihood(risk)$$

↳ Requirements w/ highest values are called risk-driving requirements

$$\text{Risk Criticality}(risk) = likelihood(risk) \times \sum_{req} impactScore(req, risk) \cdot weight(req)$$

↳ Tall poles: highest risk criticality

For ④:

1. Elicitation techniques
2. Use prev. countermeasures (e.g. Boehm's countermeasures)

3. Risk-reduction tactics

For (S): risk countermeasure table.

Countermeasures	risk 1	. . .	risk n
Criticality	. . .	. . .	. . .
	a	b	. . .
	c	d	. . .

⇒ Estimate risk reduction

0: CM has no effect

1: CM eliminates risk

Combined Risk Reduction (risk) = $1 - \prod_{cm} (1 - \text{reduction}(cm, risk))$

↳ Higher the value, easier it is to reduce risk.

CounterMeasure Effect (cm) = $\sum_{risk} (\text{reduction}(cm, risk) \times \text{criticality}(risk))$

↳ Low value → not effective. High value → effective.

Can apply cost analysis to choose best countermeasures

TEMPORAL ANALYSIS

We want to model time-based logic via predicate logic of relationships.

Ex:// Barrier is unlocked if coin is inserted

$$\forall t \in Time : coin(t) \longrightarrow \neg locked(t+1)$$

OR

$$\text{"} : coin(t_1) \rightarrow \exists t_2 \in Time : (t_1 < t_2 \wedge \neg locked(t_2))$$

Ex:// Always the case that # of part entries ≤ # of coins received.

$$\forall t \in Time : numEntries(t) \leq numCoins(t)$$

Sometimes we care about specific time intervals, but usually it's only relationships

Linear temporal logic

Express temporal order of events & variables while leaving the explicit

Properties:

① Time is ordered

② Time is past-bounded but future unbounded

③ Time is continuous

Connectives

$\Box$: henceforth $= \begin{cases} T & \text{if true in current \& all future states} \\ F & \text{o.w.} \end{cases}$

Ex:// Always the # of entries in fork $\leq$ # of coins recen.

$\models \Box$ (# of entry $\leq$ # of coins)

$\Diamond$: eventually $= \begin{cases} T & \text{if true in current or some future state} \\ F & \text{o.w.} \end{cases}$

Note: $\Box(\Diamond f) \Rightarrow f$ is happening infinitely often

$\Diamond(\Box f) \Rightarrow f$ is true forever

O : next $= \begin{cases} T & \text{in next future state} \\ F & \text{o.w.} \end{cases}$

Ex:// Turnstile is unlocked when coin is inserted

$\models \Box$ (coin $\rightarrow O \neg$ locked)

$\mathcal{U}$: until $\Rightarrow f \mathcal{U} g = \begin{cases} T & \text{if } g \text{ is eventually true \& } f \text{ is true until } g \text{ true} \\ F & \text{o.w.} \end{cases}$

Ex:// If barrier is pushed, then in next state, the barrier will be rotating until visitor has entered.

$\models \Box$ (push $\rightarrow O$ (rotating $\mathcal{U}$ enter))

$\mathcal{W}$: unless $\Rightarrow f \mathcal{W} g = \begin{cases} T & \text{if } f \text{ infinitely true or } f \text{ holds until } g \text{ true} \\ F & \text{o.w.} \end{cases}$

$f \mathcal{W} g \Leftrightarrow \Box f \vee f \mathcal{U} g$

Usually use to express some constant prop. temporarily

Ex:// If turnstile is locked, stay locked until coin entered

$\models \square$ (locked $\rightarrow$ (locked W coin))

LTL can be used in place of FSM

VALIDATION AND VERIFICATION

Validation vs. Verification

Validation: are we building right software?

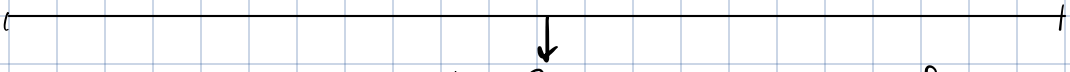
Verification: are we building software right?

Requirements Eng. Ref. Model

① Requirement: cond. that must be achieved

② Specification: descr. of proposed system

③ Domain assumption: assumption of how world should be here



Validation: 1. Req. are correct & complete

2. Domain assum. "

Even if we have requirements, we need to validate requirements

1) Requirements accurately describe system & satisfy stakeholders

2) " are correctly derived

3) " complete, feasible & verifiable

4) " necessary & sufficient

5) " consistent w/ each other

6) " provide basis to go ahead w/ building

Capturing errors in validation stage helps us catch errors earlier

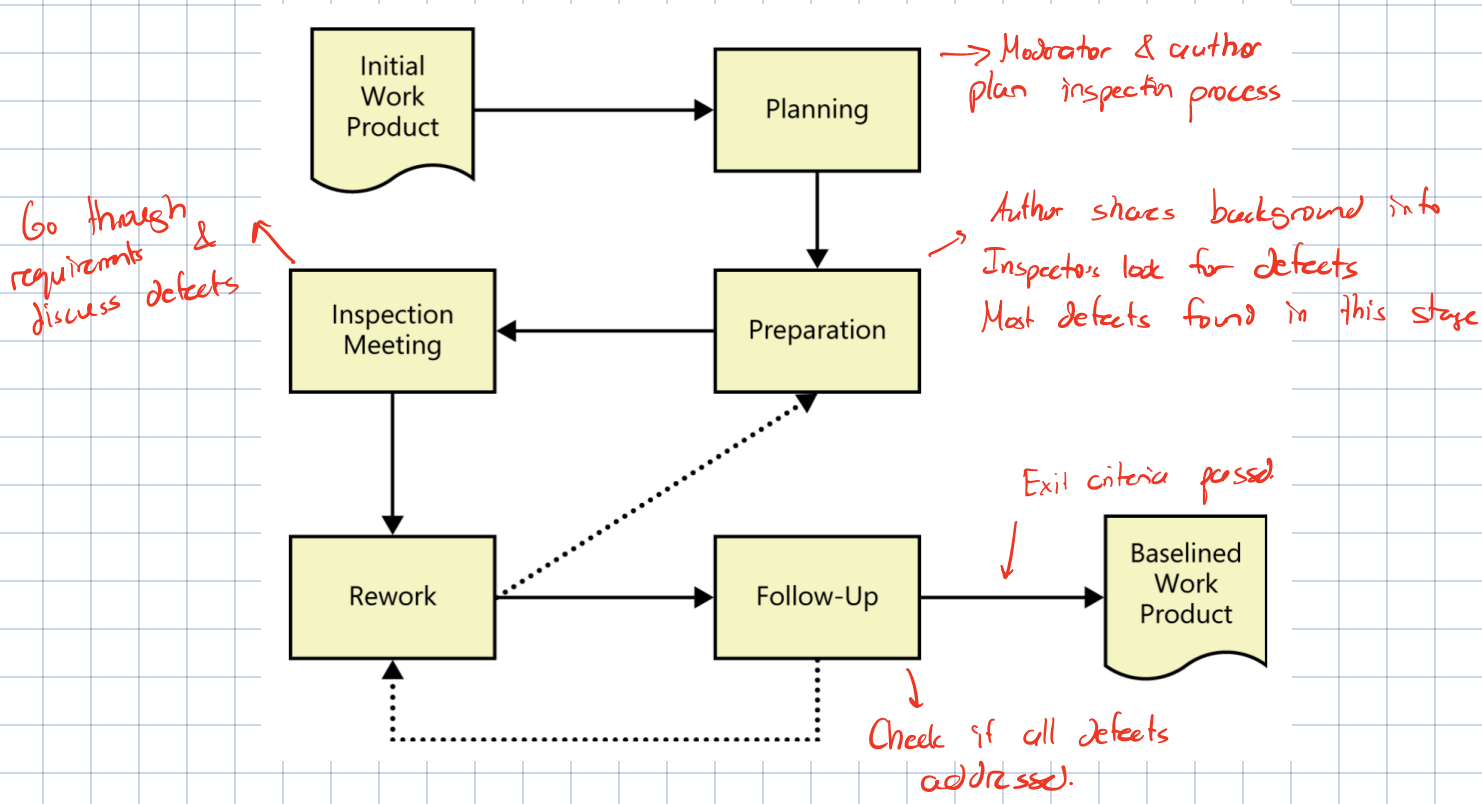
Reviewing Requirements

Inspection process: small team of participants check one part

① Participants: author of part & stakeholders

② Entry criteria: make sure work is in a reviewable state

③ Work though stages:



To help inspectors, create a defect checklist that lists common defects

Prototyping & Testing Requirements

Prototypes can help test requirements & identify blind spots

Writing functional tests on your envisioned system will also help

Acceptance Criteria

Work w end users to determine what is acceptable for both reqs. & soln.

DOCUMENTATION

SRS (software requirements specification): describes functions, capabilities & constraints

- Should it contain design / impl. details

Structure:

1. Introduction
    - 1.1. Purpose
    - 1.2. Scope
    - 1.3. Definitions
    - 1.4. References
    - 1.5. Overview
2. Description
    - 2.1 Product perspective
    - 2.2 Product functions
    - 2.3 User characteristics
    - 2.4 Constraints
    - 2.5 Assumptions
    - 2.6. Apportioning of Reqs. : reqs. that may be delayed
3. Specific requirements
    - 3.1. External interface req
        - 3.1.1. User interfaces
        - 3.1.2. H.W. "
        - 3.1.3. S.W. "
        - 3.1.4. Comms. "
    - 3.2. System features.
    - 3.3. Perf. reqs.
    - 3.4. Design constraints
    - 3.5. Software sys. attributes
        - Ls Non func. reqs. that are not perf. related
    - 3.6. Other reqs