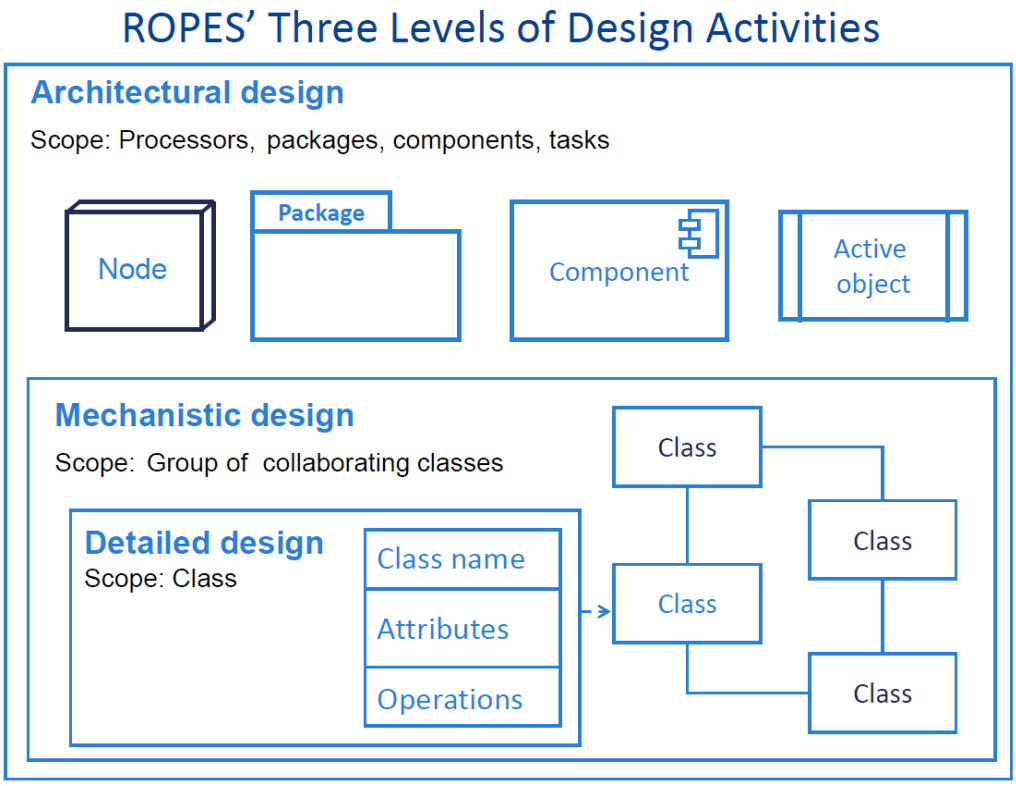
12. Software arkitektur

- Redegøre for begrebet softwarearkitektur.
- Hvordan er den typiske softwarearkitektur?
- Hvordan udarbejdes en software arkitektur?
- Hvorledes dokumenteres en software arkitektur?
- Hvorledes udarbejdes og dokumenteres en concurrency model?

Begrebet softwarearkitektur



- Softwarearkitektur er det højeste niveau hvor man prøver at bryde sin applikation ned i bidder
  - Beslutninger om systemet som er svære at ændre
  - Dokumentere hvorfor man har taget de beslutninger man har (tekst til figur)
- I arkitekturen tager vi os **ikke af**
  - Mechanistic design dvs. hvordan klasserne arbejder sammen. Typisk klassediagram med association, komposition, arv osv. eller sekvensdiagam.
  - Detaljeret design, dvs. klassernes navne, attirbutter og funktioner
- Vi er på et højere plan hvor man kan tage stilling til
- **Nodes**
  - Fysiske enheder som eksisterer: i systemet, computer, tablet, server, database osv. CPU.
  - Hvor mange, hvilke CPU/hardware kører de osv.
  - Hvordan snakker nodes sammen, hvilke teknologier.

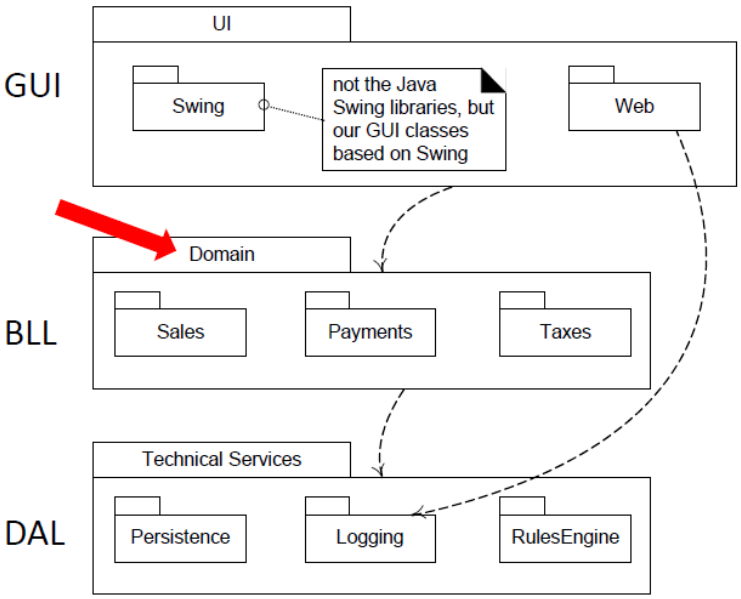
- **Packages**
  - Samling af UML elementer, typisk klasser eller use cases. Opdeler logiske og fysiske delsystemer. Hvilke nodes de arbejder på.
- **Component**
  - Beskriver en softwareenhed som eksisterer run-time (exe filer, programbiblioteker (dll), tabel i en database, filer)
  - Man kan explicit vise på deployment diagram hvilke nodes der indeholder run-time elementer
- **Active object**
  - Objekt som køres i en tråd eller klasse.
  - Concurrency modellen og tråd-kommunikationen mellem de aktive enheder.

- Tager man stilling til de ting, er man godt på vej i sin softwarearkitektur.
- Men der findes typiske softwarearkitekturer man kan bentte og bygge videre på

Hvordan er den typiske softwarearkitektur?

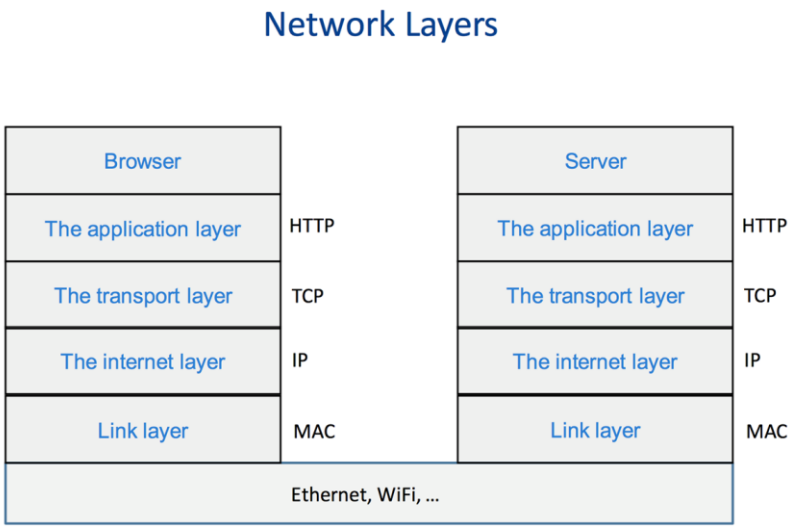
- Ligge vægt på layered architecture
- Et layer er en logisk separation af en del af systemet
- Separationen kan foregå på samme node i modsætning til fx et 3-tier hvor man vil dele systemet ud på flere node, fx flere computeren, klient, server, database.
- Tiered architecture er også typisk, hvis der er tid kan der ses på det.

Layered architecture



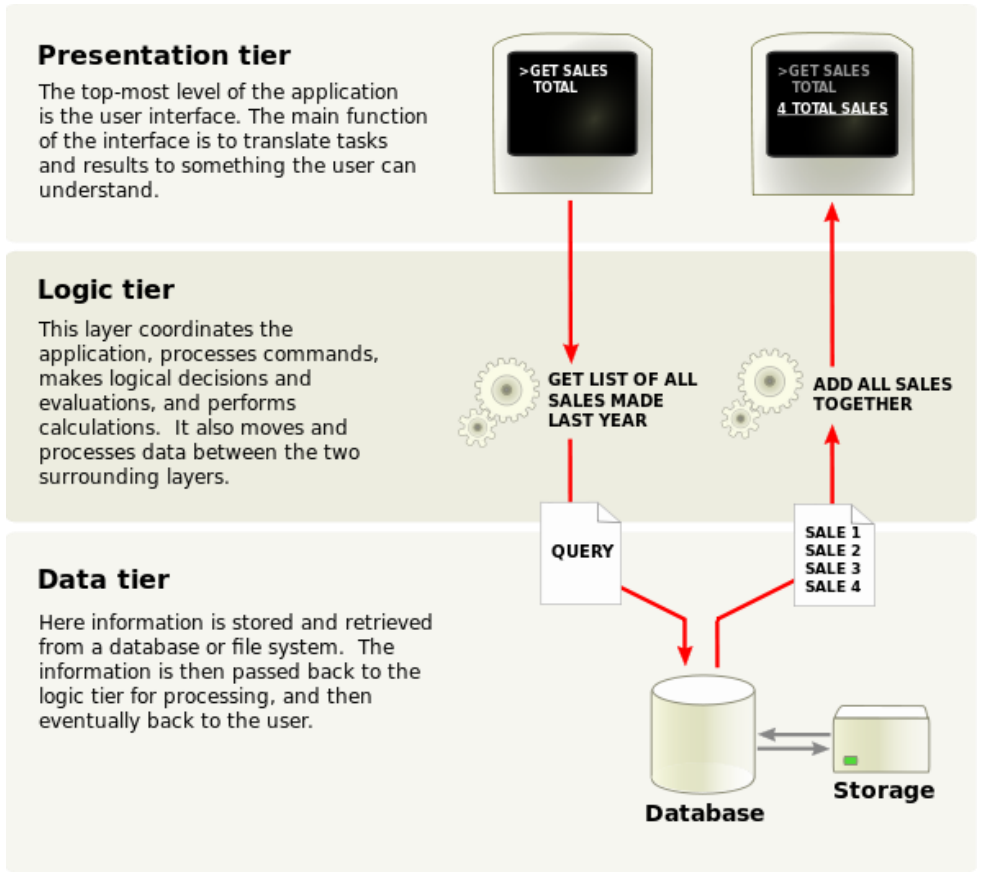
Overvejelser ved layers arkitekturen

- Layer er en logisk separation af ansvar
- Separation of concerns er i fokus
- Reducerer kobling (færre afhængigheder)
    - Højere lag kan kalde ned i lavere lag
    - ikke umiddelbart omvendt
    - Selvom det kan implementeres vha. subject/observer pattern
- Fordel at lag kan genbruges da de er lavt koblede
- Pille et lag ud eller indsætte et nyt
- Kan kodes parallelt



Eksempel fra IKN faget

3-tier klient server:



PC, Tablet, Smartphone
Håndterer alt GUI. Og præsentationen for brugeren.
Snakker sammen med server.

Server
Alt businesslogikken. Logiske algoritmer. Data-validering.
Snakker sammen med præsentationslag og data.

Database
Data gemmes sikkert. Kan hentes.
Snakker sammen med logic tier.

- Man opdeler sit system i tre tier: Præsentation, applikationslogik og datahåndtering.
- Dvs. minimum 3 nodes i sin arkitektur.

Overvejelser ved 3-tier klient server

- Præsentations-tier (web-side, pc-app) bliver "lettere" da alt applikationslogik er flyttet over på server
  - Til gengæld er der en del data der sendes frem og tilbage over netværket
- Nemt at skalere da databasen ikke behøver at have en connection for hver bruger
- Det mellemste tier applikationslogik kan håndtere data integritet
- Mere sikkert at brugeren ikke har direkte adgang til DB
- Ændringer i business logikken kan ændres ét sted, og skal ikke distribueres ud til alle klienter

Hvordan udarbejdes en software arkitektur

- Software arkitekturen består i at nedbryde systemets delsystemer
- Og få defineret de overordnede pakker(grupper af klasser) og tasks(tråde) inden for hvert delsystem
- Og hvilke nodes der er i systemet, og hvordan de snakker sammen og hvor koden skal køre

- Når der skal udarbejdes en softwarearkitektur gøres det ud fra kravspecifikationen og use-cases
- De skal sætte de mål som vores arkitektur skal kunne håndtere og understøtte

- Typisk vil en eller flere use-cases i kravspecifikationen stå som absolut nødvendig for systemets success
    - Software arkitekturen man udarbejder skal kunne understøtte denne
    - og man kan lave et **udkast** af denne software arkitekturen på et white-board

- Men der vil være andre vigtige scenarier som man må tage højde for i arkitekturen
- Man sørger for at ens arkitektur udkast også kan håndtere de andre use-cases, ITERATIV proces.

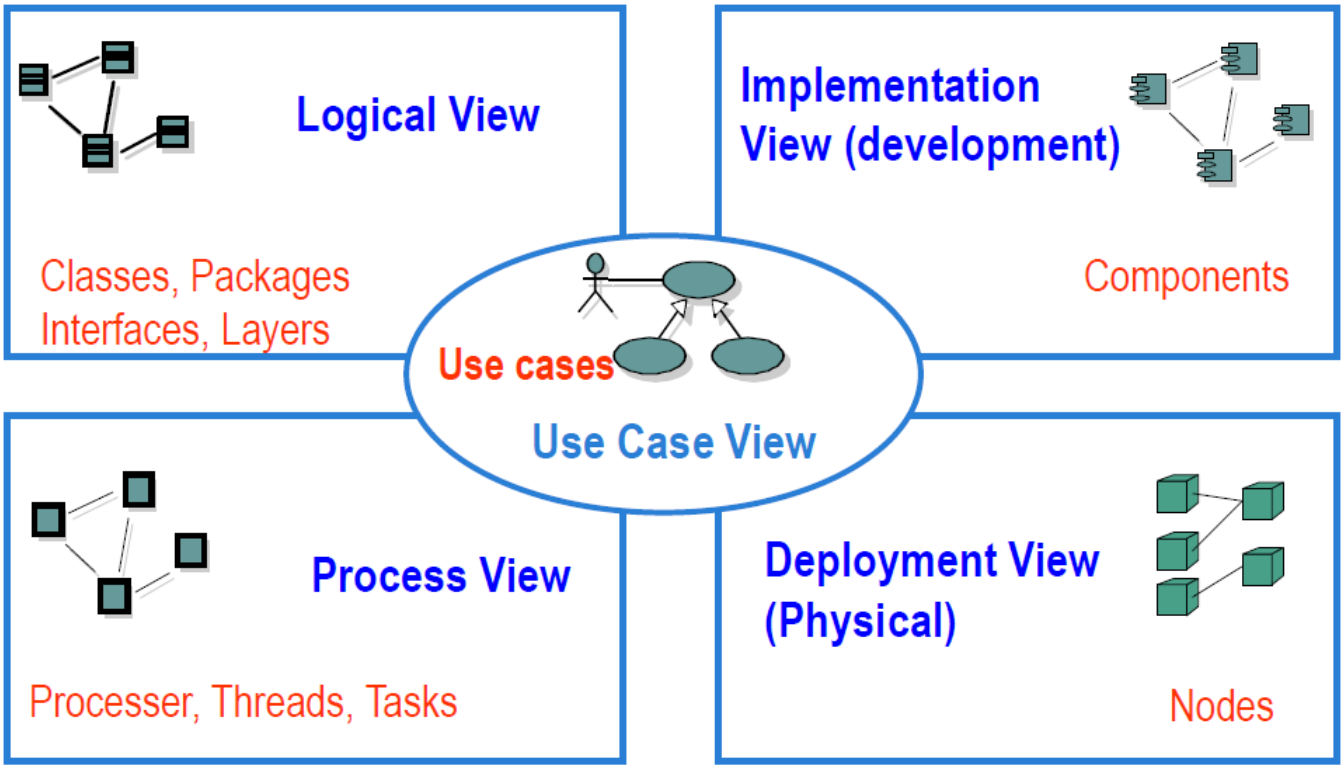
Mange overvejelser:

- Man skal have defineret applikationstypen
- Hvilket deployment architecture man vil bruge fx 2-tier, 3-tier evt. andre architectural styles såsom layered
- Og hvilke teknologier man vil benytte, kodesprog, platform, database platform osv.
- Få brugere/mange brugere?
- Sikkerhedsaspekt?
- Pålidelighed/uptime
- Fokus på brugervenlighed? (eller er det ekspert brugere)

- Man snakker om **Crosscutting concerns**
    - **(bekymringer som rammer alle tier ellers layers i din arkitektur):**
- Authentication and Authorization - hvordan håndteres sikkerheden i systemet, hvordan gemmes brugerdata, er der nogle områder som er admin beskyttet, bruger beskyttet osv.
- Communication - hvilke teknologier skal vores nodes bruge til at snakke sammen
- Exception management - hvordan skal fejl håndteres (samme på alle layer eller tier)
- Logging - skal fejl logges og hvor, hvordan?
- Validering - skal der være et validerende lag inden det gemmes ned i DB?
- Performance aspekter og skalerbarhed

Hvorledes dokumenteres en software arkitektur?

- Dokmenteringen af softwarearkitekturen skal kommunikeres ud til alle stake-holders, og de vil hver især være fokuseret på noget forskelligt
- Bl.a. derfor har vi hvad vi kalder 4+1 view af vores arkitektur
- Viser hver især arkitekturen for systemet, men fra forskellige "vinkler"



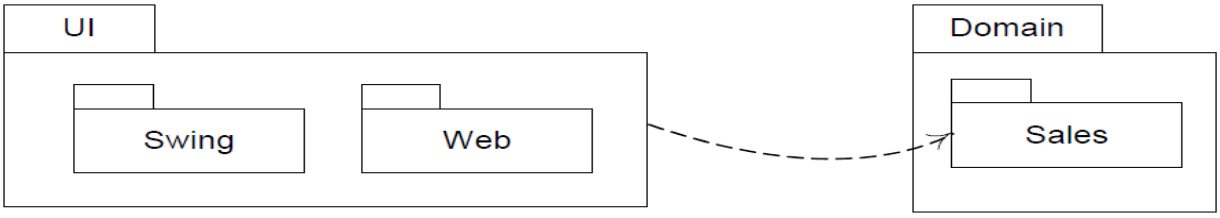
Ref.: Philippe Kruchten, "The 4+1 View of Architecture," IEEE Software, 12(6) Nov. 1995

Før UML (mærkelig notation)

Hvis tiden er knap så lad være med at beskrive views! (Vi kan evt. snakke om views hvis der er tid eller i kan spørge ind)

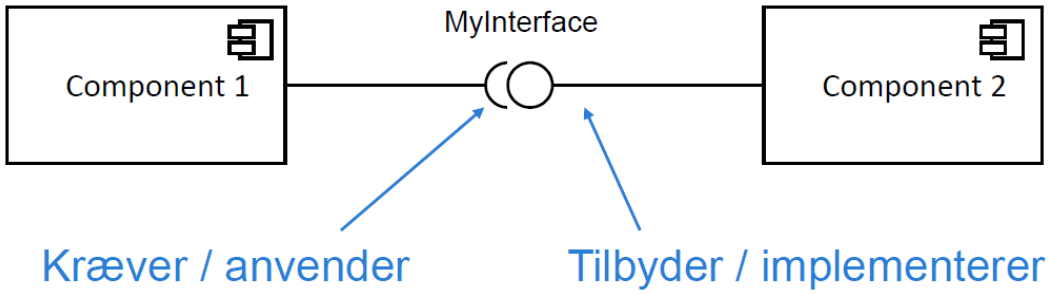
Logical view:

- Til det logiske view i forbindelse med software arkitekturen er package-diagram
    - I software-design så: klasse-diagrammer, sekvensdiagrammer
- Package-diagram beskriver pakker af softwareklasser eller use-cases som logisk er sat sammen
- Man kan sætte afhængigheder på men kun "bruger" afhængigheder

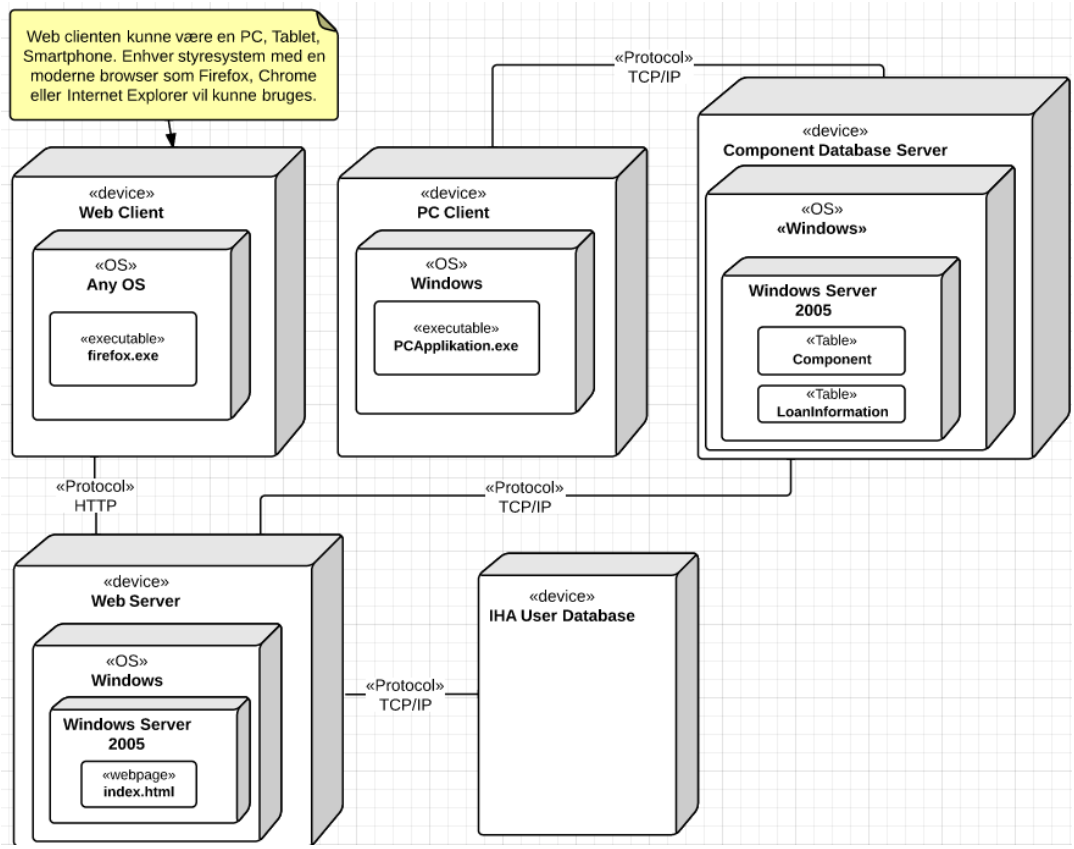


Implementation view:

- Viser de components systemet indeholder
    - Components eksisterer run-time (exe filer, programbiblioteker (dll), tabel i en database, filer) ol.
- Typisk meget store systemer, ikke noget jeg har arbejdet med
- Kan beskrive interfacet mellem fx to programmer



Deployment view:

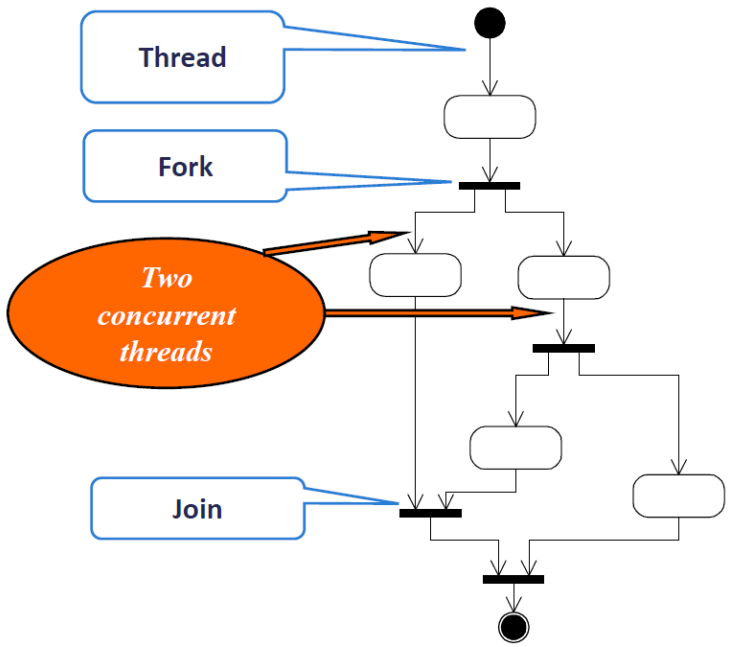


- Tager sig af at vise de fysiske enheder der er i systemet
    - Kan evt. vise hvilket software der kører på hver node fx PCApplikation, Table Component
- Viser hvordan nodes snakker sammen
- **Vigtigt at skrive noget tekst til figuren. Figuren beskriver hvordan, men hvorfor?**
- Evt. design alternativer.

Hvordan udarbejdes og dokumenteres en concurrency model?

- Til dokumenteringen kommer vores process view ind i billedet
- Der kan tegnes activity diagrams eller task diagram

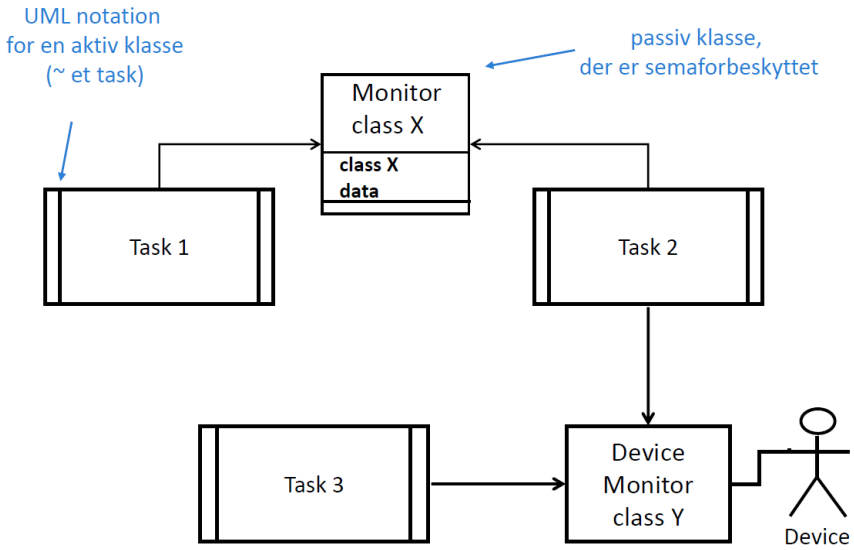
Activity diagram:



- Man kan holde styr på udviklingen af tråde
- Hvilke tråde der gør hvad (afrundede rektangler er aktiviteter)
- Holde styr på hvad der kan paralleliseres og hvad der ikke kan

Task diagram:

- Er et klasse-diagram men hvor vi viser aktive klasser og deres associationer
  - Vi kan også vise passive klasser som de aktive klasser kommunikerer med
- Der fokuseres på hvad der kører parallelt og evt. kommunikation mellem tråde (shared memory)



Udarbejdning af concurrency model:

Går i store træk ud på at identificere hvilke tasks man har eller ønsker i sit system

Man skal bruge task/tråd/process når:

- Man vil have en funktion til at køre asynkron (samtidig med andet)
- Når man har en tidsinitieret hændelse, en periodisk hændelse (hvert sekund skal Foo køres)
- Når en funktion skal køres som er koblet på et interrupt fx på noget I/O

Grupperingen af tasks

- Der findes mange måder at gruppere tasks på
- Disse danner også grundlag for at identificere om systemet har behov for flere tasks/tråde

Event source:

- Alle hændelser fra samme kilde håndteres af samme tråd/task, fx en ekstern kilde som laver interrupt på os ved forskellige hændelser.

Sekventiel processering:

- Gruppering af hændelser som alligevel skal processeres sekventielt

Relateret information:

- Hændelser/funktioner der skal arbejde på samme data kan samles i en task

Arrival pattern:

- Data der kommer med samme frekvens/hastiged kan håndteres af samme task

Computationally extensive processing

- Processor krævende beregninger (eller langsom pga. langsom datatransmission) kan grupperes i en tråd

Purpose

- Tråde der udfører et bestemt mål kan grupperes

Sikkerhedsmæssige hensyn

- Fx at lave en watchdog som holder øje med om Mars roveren er gået i stå, kan køres i en separat tråd

5+1

Man beskriver hvorfor man har taget de designbeslutninger man har

n+1 view?

Fx Data View fx ER diagrammer for databasen