Indhold

[Hvad er IT: 2](#_Toc494278068)

[Hvad er IT(informations teknologi): 3](#_Toc494278069)

[App-domain: 3](#_Toc494278070)

[Problem domain: 3](#_Toc494278071)

[Fire hovedaktiviteter: 4](#_Toc494278072)

[Komponent design: 4](#_Toc494278073)

[Applications domæne analyse: 4](#_Toc494278074)

[Problem domæne analyse: 4](#_Toc494278075)

[Komponent design: 4](#_Toc494278076)

[System valg: 5](#_Toc494278077)

[System definition: 5](#_Toc494278078)

[Valg af system: 6](#_Toc494278079)

[Beskriv situationen: 6](#_Toc494278080)

[’Rich Pictures’: 6](#_Toc494278081)

[At tegne rich pictures: 7](#_Toc494278082)

[Processer: 7](#_Toc494278083)

[Strukturer: 8](#_Toc494278084)

[Problemer: 8](#_Toc494278085)

[Praktiske råd: 8](#_Toc494278086)

[Få idéer: 8](#_Toc494278087)

[Eksempler 8](#_Toc494278088)

[Metaforer 9](#_Toc494278089)

[Eksperimenter 9](#_Toc494278090)

[Definer systemer: 9](#_Toc494278091)

[Systemer er holistiske syn: 9](#_Toc494278092)

[Eksempler på definitioner, side 38. 9](#_Toc494278093)

[Praktiske råd: 9](#_Toc494278094)

[Factor standard: 10](#_Toc494278095)

[Evaluering og valg: 10](#_Toc494278096)

[Principper: 10](#_Toc494278097)

[Genbrug af mønstre: 10](#_Toc494278098)

[Prototype 10](#_Toc494278099)

[Forskelle: 10](#_Toc494278100)

[Typer af eksperimenter!!!: 10](#_Toc494278101)

[Quick and dirty: 10](#_Toc494278102)

[Throw away 11](#_Toc494278103)

[Design – drevet: 11](#_Toc494278104)

[Mock-up: 11](#_Toc494278105)

[Evolutionary: 11](#_Toc494278106)

[Explanatory: 11](#_Toc494278107)

[Evaluating: 11](#_Toc494278108)

[Hvad kan smides væk: 11](#_Toc494278109)

[Implementation af prototyper: 12](#_Toc494278110)

[Boehm’s specific technique/ constructive cost model/ cocomo 12](#_Toc494278111)

[Wizard of Oz prototype: 12](#_Toc494278112)

[Iterativ fremgangsmåde: 12](#_Toc494278113)

[Contingency teori: 12](#_Toc494278114)

[Davis 12](#_Toc494278115)

[Burns og Dennis: 13](#_Toc494278116)

[Mathiassen og Stage: 13](#_Toc494278117)

[Problemdomæne analyse: 13](#_Toc494278118)

[Klasser: 14](#_Toc494278119)

[Vælg klasser: 14](#_Toc494278120)

[Find events 14](#_Toc494278121)

[Systematisk evaluering 15](#_Toc494278122)

[Udviklingskriterier for klasser: 15](#_Toc494278123)

[Evalueringskriterier for events: 15](#_Toc494278124)

[Overblik over analyse (Hårsalon): 16](#_Toc494278125)

[Situation: 16](#_Toc494278126)

[Development task: 16](#_Toc494278127)

[Prototype eksperimenter: 16](#_Toc494278128)

[System definition: 16](#_Toc494278129)

[Problemdomæne analyse: 16](#_Toc494278130)

[Noter – forelæsning d. 25/09 16](#_Toc494278131)

# Hvad er IT:

Hvad er IT(informations teknologi):

Input/output, enhed der behandler data og information, hukommelse. Netværk/ kommunikation.

* *Brugen af computere til at gemme, udlede, kommunikere og manipulere data og information.*

ICT(c = kommunikation)

**IT system(computerised system):** Et software system i stand til at lave IT.

* *Et organiseret system til at gemme, udlede, kommunikere og manipulere data og information.*

**Andre mulige elementer (Ikke definitioner brugt i kurset):**

* Eksterne enheder
* Andre systemer, hvad kommunikerer systemet med
* Brugere og andre personer, hvem bruger systemet
* Procedure, hvordan bruges systemet
* Reglerfor hvordan man bruger systemet

## App-domain:

Hvor brugere er. Organisationen der administrerer, overvåger eller kontrollerer et problemdomæne

## Problem domain:

Delen af konteksten der er administreret, overvåget eller kontrolleret af et system. Når noget sker i problemdomænet, opdateres modellen og app domænet.

Problem domæne 🡪 Model 🡪 App-domæne

Opdateringen fra problemdomænet til modellen, og fra modellen til app domænet kan variere i vigtighed af hastigheden. Det er noget man bestemmer når man laver requirements.

Det sammen objekt kan være en del af begge domæner, men ikke altid. Det handler om hvad menneskers rolle er i handlingen.

Man kan have en model med en funktion der automatisk opdaterer modellen, der igen opdaterer problemdomænet.

**Eksempel:**

Hvis en ansat på et universitet tager et kursus, vil denne både være en del af problem og app domænet – både en del af systemet og administreret af systemet.

I problemdomænet vælger en underviser en bestemt karakter til brugeren. Nu opdateres modellen, og fortæller karakteren. Denne vises i app domæne. Hun går altså ikke gennem problemdomænet og beder om karakteren, men får den gennem modellen. Her er hastigheden af opdateringen ikke kritisk. Det kan den fx være i et system for at tilkalde ambulancer.

Lærer informerer om karakter 🡪 Systemet ændrer karakteren 🡪 Brugeren ser karakter

**Bruger:**

Er i app domæne, og får info om problem domænet gennem modellen.

Fire perspektiver:

**Informations perspektiv** - skal tilbyde en brugbar model til problemdomænet. Er vigtig i analysen, men også med design da dette skal gøre modellen mere effektiv og brugbar.

**Brugerens synspunkt:** systemet skal integreres i app domænet. Forstå de personer og andre systemer der skal benytte systemet. Skal kunne samarbejde med andre enheder og ændre sig i forhold til ændrede krav.

**Arkitekturelle perspektiv:** systemet skal køre på en bestemt platform. Hvordan skal systemet opbygges osv. Husk de fysiske komponenter i dette.

**Et hele:** det skal virke.

Emphasize the architecture:

Fokuser på systemet fra starten. Brugeren og andre systemer ser kun interfaces, under det ligger alle funktionerne, og som fundament af det, modellen.

Arkitekturen er en generel struktur af det der bliver udviklet videre

Systemet er den samling af komponenter der implementerer

# Fire hovedaktiviteter:

## Komponent design:

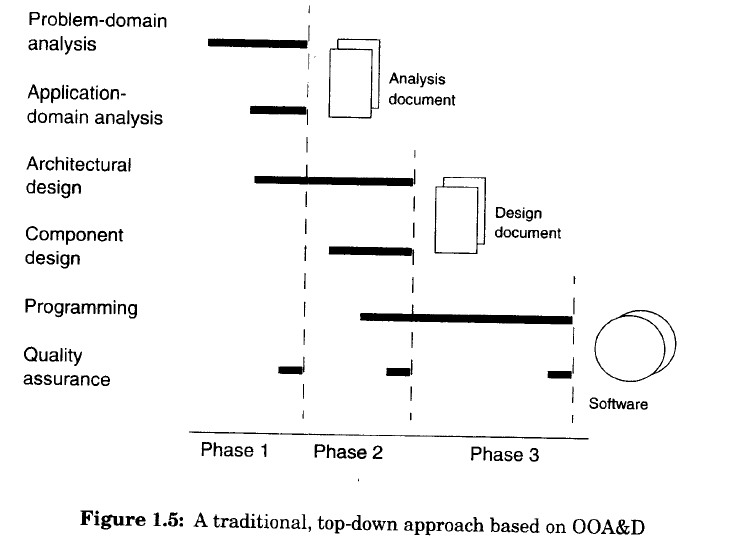
## Applications domæne analyse:

## Problem domæne analyse:

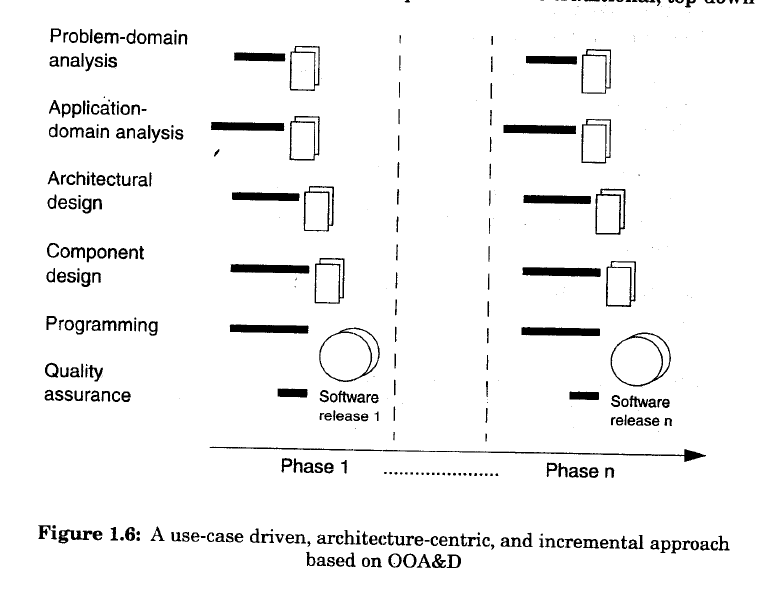
## Komponent design:

**Fremgangsmåder:**

**Traditionel top-down:**



**UML**



Top down er mere drevet af problem domæne analyse, mens den anden er drevet af application domæne analyse

**Dokumentation:** giver kontinuitet i løbet af systemudvikling.

**Exhaustive seach = brute force,** buttom up problem solving

**Metode:** Retningslinjer for at udføre udviklings aktiviteter(fx analyse og/eller design).

# System valg:

Formål: Være enig I de overordnede karakteristika af systemet.

Konsept: definition af systemet. En konkret beskrivelse af et computersystem i et normalt sprog.

Principper: forstå situationen. Udvikle nye idéer. Definere alternative systemer.

Resultat: en system definition der udfylder FACTOR criterier.

Når man udvikler et system skal man overveje følgende:

* Hvilke problemer er det vi vil løse
* Er det planlagte system en brugbar løsning
* Hvad ville der ske hvis vi udviklede et helt andet system?

## System definition:

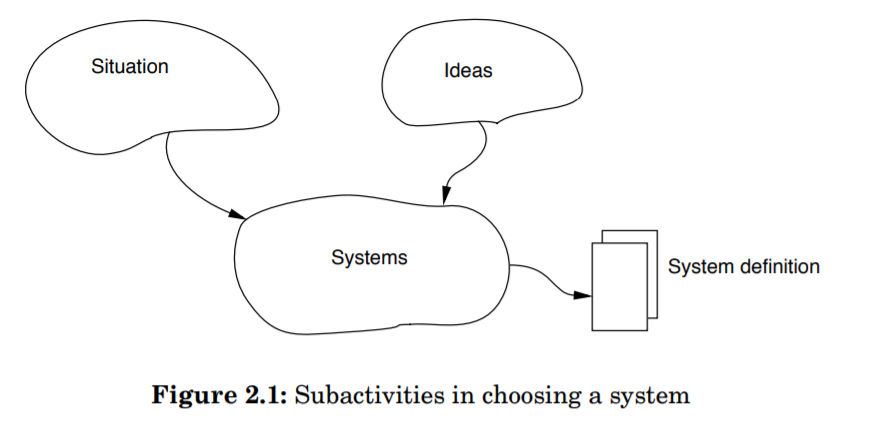
Nogle gange er problemet systemet udvikles til præcist defineret, mens det andre gange er mere løst. Et problem kan altid blive set på forskellige måder, enten fordi det er af forskellige personer, eller forskellige vinkler. SD udtrykker de fundamentale egenskaber af systemudvikling og brug. Det viser systemet i kontekst, med hvilken information de burde indeholde, hvilke funktioner det skulle tilbyde, hvor det skal bruges og hvilke udviklings betingelser der gælder. Idéen er at man finder forskellige måder og muligheder at løse problemet på. Du kan se forskellige løsninger, og sammenligne alternativer. System definitionen der bliver valgt skulle lægge et godt fundament for den videre analyse og design aktiviteter. Skal være kort of præcis, og indeholde de fundamentale valg omkring systemet. Det giver overblik, og gør det let at sammenligne.

Hvis kriterier skal genovervejes, brug fx en iterativ fremgangsmåde, hvis ikke brug vandfald.

Uklare kriterier er blevet antaget som hovedkilden til problemer i systemudviklingen.

Når man skal overveje om ting i app og problem domænet er relevante, kigger man på SD

## Valg af system:



De tre underkategorier når man skal vælge et system er:

1. Situationen. Prøv at danne overblik over situationen og de forskellige måder at tolke denne.
2. Idéer. Teknikker til at understøtte kreativitet og introducere nye måder at tænke på.
3. Definition. Diskussion og evaluering af alternative definitioner af vores situation

En definition vælges ved at gå igennem de tre underkategorier. Når man beskriver en situation giver det ofte muligheder for nye idéer. Denne proces påvirker de andre underkategorier, og giver omstændigheder der skal undersøges, og idéer der skal yderligere undersøges.

Valg af system kræver: fantasi, gode idéer, udviklingsmæssige evner og kritisk sans.

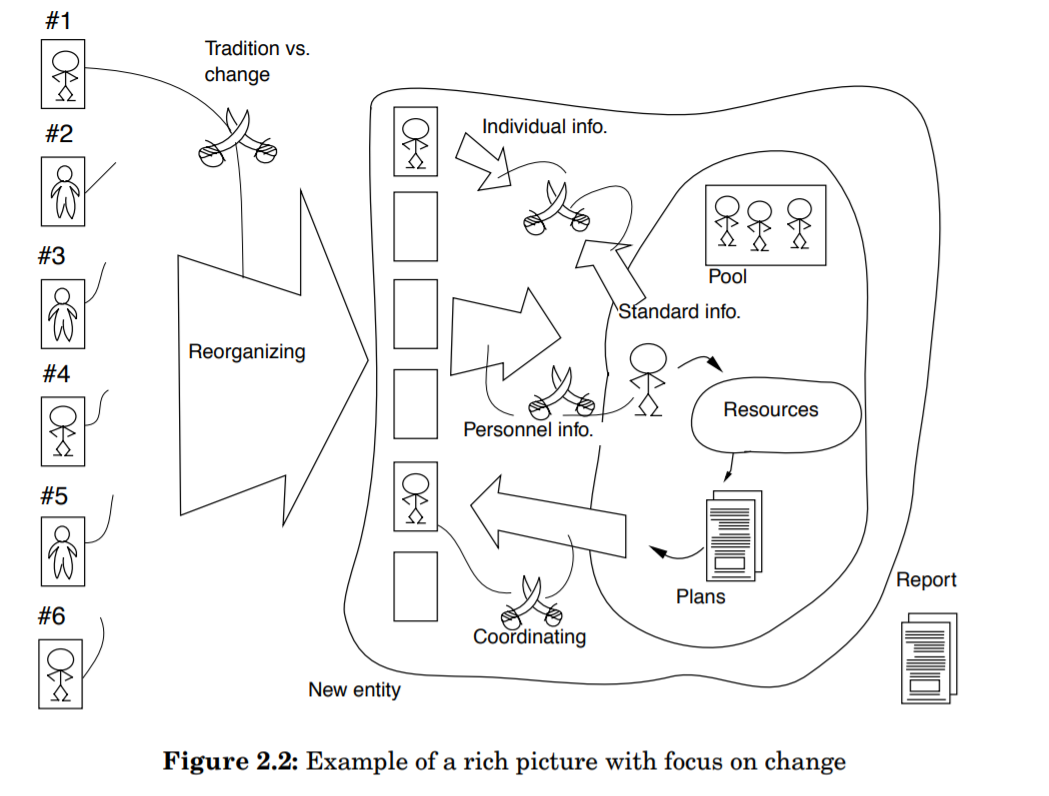
Man skal starte med at undersøge brugerens situation, og forstå hvori udfordringen ligger. Dette kræver tæt samarbejde med brugeren og kunden. Man skal reflektere kritisk over brugerens ønsker og idéer, især hvis de peger mod en bestemt løsning. Brugeren har ofte ikke tilstrækkelig forståelse af udviklingsprocessen.

## Beskriv situationen:

Man skal forstå brugerens situation ordentligt. For at opnå dette skal man være klar til en konstruktiv diskussion.

### ’**Rich** Pictures’:

Et ”rich picture” er en beskrivende tegning der repræsenterer illustrators forståelse af en situation.

Eksempel på et rp fra et hospitals udvikling af et projekt(side 27): 

Billedet beskriver en ændring i seks autonome afdelinger til en ny struktur hvor de bliver lagt sammen. Formålet er at øge koordination og samarbejde mellem afdelingerne.

Rp fokuserer på vigtige aspekter af situationen. Det skal give en bred beskrivelse af situationen, så man kan have flere forskellige fortolkninger.

Det kan være praktisk at differentiere mellem rp der fokuserer på stabilitet og rp der fokuserer på ændringer. Det er vigtigere at forstå de essentielle aspekter end at arbejde systematisk. For at forstå kulturen og finde ud af hvad der er vigtigt skal man besøge organisationen og se hvordan den virker, tale med forskellige mennesker om hvad der sker og burde ske, og måske lave nogle interviews. RP kan være brugbar til at lave god kommunikation bed brugeren. De kan også bruges til at beskrive syn på situationen mellem udviklere.

### At tegne rich pictures:

Det er en god ide først at tegne vigtige ting så som mennesker, fysiske objekter, steder, organisationer, roller og opgaver. Problem ejere kan også være en del af billedet. De er dem er identificerer problemet, direkte eller indirekte. Kunden er en problem eje, men det er brugeren også måske. Man skal ikke tegne et rp som kunden udtrykker problemet, men som man selv opfatter situationen. Det er vigtigt at finde gode symboler på elementerne.

### Processer:

Når objekterne er tegnet skal relationen imellem disse udarbejdes. Processer er de mest fundementale relationer mellem elementer. En proces beskriver aspekter af situationen der ændrer sig, er ustabile eller under udvikling. Disse kan illustreres med pile. Når man beskriver udveksling af information er det vigtigt at forstå hvordan denne information er produceret, forstået og brugt.

### Strukturer:

Beskriver aspekter af en situation der opfattes som stabile eller svære at ændre. De beskrives enten ved at tegne steger mellem elementer, eller ved at placere relaterede elementer i en fælles figur som firkant eller cirkel.

### Problemer:

Typisk værdier og statements der karakteriserer et problem. Kan repræsenteres som krydsede svær. Problemer kan markerer utilfredshed, i stedet for deciderede problemer. Det er derfor en god idé at knytte et par ord til et problem. Undgå at beskrive løsninger samtidig. Fokus skal være på hvordan bruger ser problemet.

### Praktiske råd:

Et brugbart RP skal:

* Have en masse information og være åben for fortolkning.
* Præsentere proces of struktur i en sammenhængende og velbalanceret måde.
* Vise mindst et problem
* Pege på flere relevante computer systemer.
* Være rigt, ikke kaotisk.
* Belyse nøgle aspekter af situationen så den kan forstås på mange niveauer
* Ikke repræsentere dato og data processer

Det er bedre at tegne flere billeder end at gøre det kaotisk, især hvis situationen er meget kompleks.

# Få idéer:

Vær åben for nye idéer. Gamle idéer er god inspiration. Balancer mellem at udvikle nye idéer og genbruge gamle. Nye idéer udvikles i samarbejde med fremtidige brugere. Uden forståelse for brugerens kontekst vil idéer ofte være generelle og abstrakte.

Det er foreslået at bruge

* Eksempler
* Metaforer
* Eksperimenter

Når man udvikler idéer sammen med brugere.

### Eksempler

Det er en god ide at kigge på systemer der minder om dem brugeren ønsker.

For hver eksisterende løsning besvares følgende spørgsmål:

* Hvilke ideer er fundament for systemet
* Virker ideerne brugbare? hvorfor?
* Vil ideerne virke i din kontekst? Hvorfor?
* Kan ideerne blive ført hen på dit system? Hvordan?

### Metaforer

Her ses systemer i et nyt lys. Dette er kun en intellektuel øvelse, der ikke involverer andre. Kan hjælpe med at overføre ideer fra andre områder. Når en metafor er fundet, følges følgende step:

1. Lav en lang liste af aspekter relateret til metaforen
2. Overfør aspekter til dit system
3. Bestem hvilke aspekter der er brugbare

### Eksperimenter

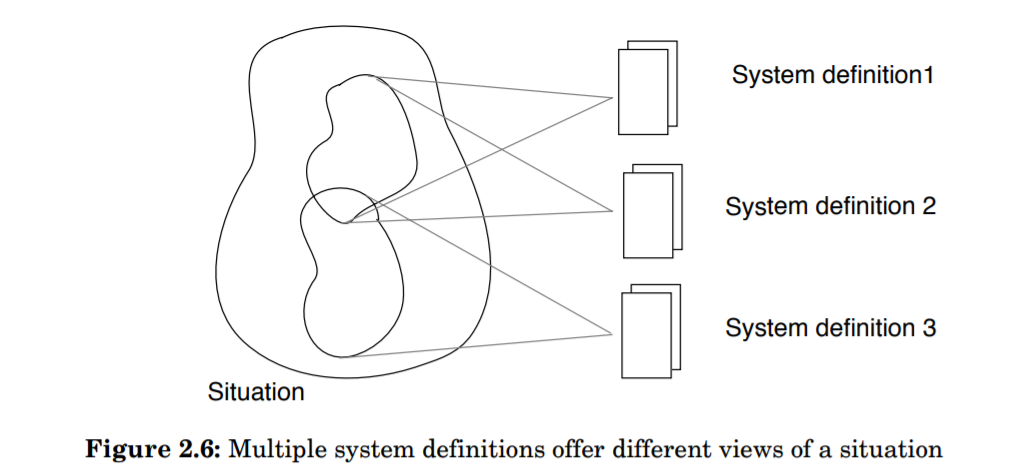
Man kan bruge en simpel prototype til at forsøge sig frem. Dette kan også være med en tegning for at undersøge display. Dog er det bedre hvis muligt at lave computerbaserede prototyper.

# Definer systemer:

Her vælges systemet. Dette gøres ved systematisk at redegøre for fortolkninger, muligheder og konsekvenser af forskellige løsninger. Man kan formulere kvaliteter af potentieller systemer. Denne kan gøres ved at formulere kvaliteterne til systemdefinitioner, der er nemt overskuelige

## Systemer er holistiske syn:

Det samme system kan blive set på forskellige måder alt efter hvordan det er brugt. Hver definition repræsenterer en opfattelse af verden filteret gennem ideer, uddannelse og baggrund for de involverede.



En systemdefinition skal udtrykke systemfokuserede kvaliteter, og ikke detaljerede komponent baserede kvaliteter. Før man vælger et system skal man arbejde med flere forskellige.

## Eksempler på definitioner, side 38.

## Praktiske råd:

* Brug generelle termer, og sæt fokus på vigtige definitioner
* Fokus på ideer og ikke beskriv situationen som den er
* Lav definitionen kort og præcis
* Eksperimenter med forskellige definitioner
* Brug nye måder at tænke på
* Brug udvælgelses processen til at lægge mærke til nye relevante aspekter

# Factor standard:

FACTOR standarden består af seks elementer.

1. Funktionalitet. Systemfunktionerne skal understøtte hovedopgaverne af applikationen
2. Applikations domæne. De dele af organisationen der administrerer, overvåger og kontrollerer problem domænet
3. (Condition)Betingelser. Betingelserne hvorunder systemet skal udvikles.
4. Teknologi. Både den til at udvikle systemer, og den hvorpå systemet vil køre.
5. Objekter(Problem domæne). Hovedobjekterne i problem domænet
6. (Responsibility)Ansvar. Systemets generelle ansvar i forhold til dets kontekst.

FACTOR kan bruges til at understøtte system definition udviklingen, hvor man omhyggeligt undersøger hvert element. Det kan også bruges til at starte beskrivelsen af systemet og bruge kriterier til at se hvordan systemdefinitionen understøtter de seks faktorer. Begge begyndelsessteder kan bruges. Det kan godt være at man flere gange skal formulere definitionen, og derefter sammenholde den med FACTOR.

Hvad gør systemet, og hvad gør brugeren? Definer grænsen mellem disse.

## Evaluering og valg:

Systemudvikleren skal ikke vælge et system, men at lette valget. Et system skal vælges gennem aktive forhandlinger imellem alle involverede parter.

# Principper:

## Genbrug af mønstre:

Ikke lav en ansat og elev klasse, en person kan være i begge roller. Lav i stedet en klasse med et antal elev roller, og en række ansat roller. Find ud af om objekter kan have forskellige roller. Sørg for at alle kan være i et objekt.

# Prototype

## Forskelle:

Hvor et system har interface, funktion og en model, har prototype måske kun en eller to af disse, som fx interfaces. En prototype kan være teknisk begrænset, og mangle fx en sluknings funktion.

Det hedder: Interface prototype og tekniske prototype.

Brugeren kan give feedback på designet før udviklingen. Brugere ved sjældent præcis hvad de vil have.

## Typer af eksperimenter!!!:

### Quick and dirty:

Implementation uden analyse og design Genudviklet indtil brugere er tilfredse. Kan blive meget dyrt

### Throw away

* UI protoryper
* Mock-up

### Design – drevet:

Teknisk

### Mock-up:

Non executional model, kun til visuelt design.

### Evolutionary:

Lave kernen af systemet, og udvikler det stille og roligt. Bliver stille og roligt den sidste version.

Man bygger ovenpå i stedet for at ændre. Man skal være relativt sikker på at de ting man skal lægge på er hvad man skal bruge.

## Explanatory:

Bruger prototype til at udvikle nye design idéer.

Formålet er altid at lære noget.

De steps man skal følge er:

1. planlægning

* Hvad er fokus af eksperimentet
* Hvad er blevet udelukket
* Hvilke forudsætninger er der

1. Udvikling

* Man skal lave den hurtigt

1. Forberedelse

Man skal bestemme sig for følgende ting:

Samarbejde, Man kan fx lade brugere teste prototype under observation

* Realisme (Hvad sker der i den virkelige verden), man kan enten lade brugere bruge prototype i forbindelse med den normale aktivitet, eller separeret fra denne.
* Udvælgelse, hvem skal teste. Hvem repræsenterer de

Er der noget bruger skal forberede

1. Test

Realiser de ting valgt før. Dokumentation kan fx ske ved hjælp af en dagbog brugeren skriver,

1. opsummering

Opsummer på det valgt i planlægning. Her kan man enten vælge at lave flere tests, eller begynde at udvikle videre.

## Evaluating:

Bruger to eller flere prototyper til at vælge design.

## Hvad kan smides væk:

Udvikling

Læg det på en anden platform i forbindelse med prototype hvis lettere. Man skal ikke lave så meget kvalitets sikring og dokumentation(undtagen evolutions)

Produkt:

Skal ikke være så meget kode.

Behøver ikke at være effektiv, pålidelig og rubust hvis det ikke er en teknisk prototype

Ikke vigtigt med errorhandling.

Horisontal slice, vise fx kun ui, ingen funktioner, flere inteface sider uden funktionelt indhold.

Vertikal slice, Nogle funktioner og lidt model, lidt interface. Fx kun en side i en app.

## Implementation af prototyper:

Papir: godt for visuel, dårligt for funktion. Wizard of Oz hvis illusterer struktur. Godt til at vise design

Power point: lidt hurtigere visuel tilgang.

Tools: marvelapp.com, justinmind.com, Balsamiq(koster penge)

Jo tidligere i processen du er, jo længere er du oppe på listen.

# Boehm’s specific technique/ constructive cost model/ cocomo

Empirisk omkostnings estimat model. Bruger data fra forrige input til at beregne næste. Det endelige estimat bliver dannet ved at korrigerer det første.

## Wizard of Oz prototype:

Få det til at se ud som om noget er funktionelt uden at der virkelig er noget IT teknologi bag

# Iterativ fremgangsmåde:

Grund: Faktiske problemer er sjældent klare og præcise. Hvis man prøver noget gentagende gange, reducerer man usikkerheden. Når man siger at prototypen er god nok kan man begynde at arbejde med det. Mange gange laver man en release, får feedback og korrigerer systemet – software opdateringer. Prototyper er en del af dokumentationen for krav. Her kan man sige ”i godkendte det”.

## Contingency teori:

Hvordan vælger man mellem vandfald og ”life cycle”(iterativ) prototype.

Handler om usikkerhed.

* Organiserings og teknisk kontekst af systemet
* Fremtidig computer system
* Erfaring og evner af bruger
* Erfaring og evner af udvikler

Baseret på usikkerhed. Hvis lav usikkerhed, så kan kriterie bestemmelse baseres på en informeret fremgang på analyse og eksisterende systemer

### Davis

Hvis høj, brug specifikationer og prototyper.

### Burns og Dennis:

Handler om kompleksitet mellem kompleksitet og usikkerhed

Usikkerhed:

* Hvor struktureret er brugers arbejde
* Graden af forståelse brugeren har af deres arbejde
* Hvad ved vi og hvor meget erfaring har vi på området?

(lavere, mere usikkerhed)

Kompleksitet:

- projekt størrelse

- antal af brugere

-volumen af ny info

- kompleksitet af ny info

### Mathiassen og Stage:

Det skal gøres ofte, spørg ofte hvad usikkerhed og kompleksitet.

Hvis høj usikkerhed, prøv igen og igen.

Når man laver løsninger for at reducere kompleksitet, kommer nye kilder for usikkerhed og omvendt. Fx, jo flere man spørger, jo mere kompleksitet og mindre usikkerhed

At dele projekt ind i elementer giver mindre kompleksitet og mere usikkerhed. Man skal ende med nul usikkerhed, og nul kompleksitet

Skal bruge prototype ved høj usikkerhed. Giver mere kompleksitet.

Ved høj usikkerhed:

Ved høj kompleksitet:

Risiko analyse faktorer:

Definer rissisi.

Specifice konsekvenser

Vælg løsnings strategi.

# Problemdomæne analyse:

Formålet er at identificere og modellere den del af konteksten der er administreret og overvåget eller kontrolleret af systemet. Modellen er her en beskrivelse af klasser, objekter, strukturer og opførsel af et problemdomæne. Man skal lave en model over den virkelige verden som fremtidige brugere ville se den. Dette gør man ved først at danne sig et overblik, for så at tilføje detaljer. På den måde får man en sammenhængende model af problemdomænet.

## Klasser:

(side 49)

Når man taler om klasser i en problemdomæne analyse, handler det om at vælge elementerne i et problemdomæne. Man skal så se på de ”events” eller handlinger disse udfører i problemdomænet, og lave en klasse ud fra hvordan disse interagerer med hhv. hinanden og andre.

Først finder man ud af hvilke handlinger der er vigtige i den givne kontekst. Dette kan man gøre ved hjælp af problembeskrivelsen.

Abstraktion 🡪 Klassifikation 🡪 Selektion.

Man gør elementer i problemområdet abstrakte ved at omdanne dem til klasser og events. Vi klassifiserer så disse, og vælger hvilke systemet skal indeholde information om.

En ”eventtable” hjælper med at få overblik over sammenhæng.

(Se figur 3.1 siden 50 for information om eventtable)

Når man udvikler et system skal man ikke bare modellere hvordan det fungerer nu, men beskrive hvordan processen vil virke når systemet er implementeret.

Under problemdomæne analyse er objektet en abstraktion af et fænomen i problemdomænet. Ikke konkret som ellers. Karakteriser objekter gennem deres events.

En event er her en øjeblikkelig hændelse der involverer et eller flere objekter.

Husk kun at introducerer vigtige events. Der vil ofte være en masse uvigtige ting for systemet som kunne implementeres.

### Vælg klasser:

(side 55)

Skriv alle potentielle klasser ned uden at tænke over detaljerne, og så vælg de vigtigste.

Brug først hvad du ved om området, og gå så andre veje. Dette kan være lignende systemer, interview med stakeholders, litteratur om emnet etc.

Brug klassenavne der er:

Simple og letlæseliige

Stammer fra problemdomænet

Indikerer en enkelt instans

### Find events

(side 57)

Se på aktivitetens start- og slutpunkt. Analyser events andre systemmodeller bruger. Læs teknisk litteratur, love og standarder.

Navngivning er stort set som ved klasser.

Verbformen af navnet kan enten være givet i nutid, datid eller igangværende (sidder, sad, sætter). Vær konsistent.

Nogle gange må man se på hvad klasse eller event repræsenterer.

### Systematisk evaluering

(side 60)

En klasse eller event skal være i problemdomænet kun hvis systemets funktioner bruger information om det.

Er klasser/event i systemdefinitionen?

Er klasse/event relevant for problemdomæne modellen?

Hvis man finder ud af at problembeskrivelsen og relevante klasser/events ikke hænger sammen, bør man mødes med kunden og løse problemet. Dette kan enten gøres ved at afvise klasse/event, eller ændre systemdefinitionen.

Kun inkluder klasser/events i problemdomænet, nogle brugere vil administrere, overvåge eller kontrollerer i fremtiden. De skal ikke illustrerer aspekter af applikationsdomænet.

Typisk er bruger IKKE i problemdomæne. Skal kun inkludere klasser/events brugere identificerer (undtagen hvis systemet laver nye måder for brugeren af forstå problemdomænet)

### Udviklingskriterier for klasser:

(side 61)

Når man evaluerer en klasse skal man spørge sig selv:

* Kan jeg identificere objekter i klassen?
* Har den unik information?
* Omfatter den flere objekter?
* Har den en passende og håndterbar mængde events?

Hvis informationen i en klasse kan blive afledt fra en anden klasse, bør den afvises. Nogle gange er det relevant at lave en klasse, selvom denne kun indeholder én instans.

Husk at definer en klasse med mere end et navn, så alle er enige om hvad den indeholder, og så man ikke glemmer detaljer.

Man kan angive ansvaret for en klasse, så formålet med den er klar (se side 62 nederst). Hvis en klasse har for mange ansvar er den ikke sammenhængende, og bør sandsynligvis blive splittet op i mindre klasser.

### Evalueringskriterier for events:

(side 63)

Man skal spørge sig selv:

* Sker eventet øjeblikkeligt?
* Er den atomar? (kan ikke deles ind i underevents)
* Kan man identificere når det sker?

Hvis den har en varrighedstid skal man se på start- og stopeventet, og måske endda nogle ind imellem. Beskriv hvorfor den skal inkluderes.

# Overblik over analyse (Hårsalon):

(side 383)

Situation 🡪 Development task 🡪

## Situation:

Hvilken kontekst skal systemet udvikles til? Hvordan er de relevante omgivelser nu?

## Development task:

Hvordan er problemet nu? Hvad kunne man afhjælpe? Hvad ønsker kunden? Hvilken fremgangsmåde vil vi have for at forstå situationen? Hvorfor?

## Prototype eksperimenter:

Beskriv hvilke prototyper du bruger.

Hvilken udvikling sker der ud fra dette?

Hvilke mangler viser dette?

## System definition:

Lav først den endelige systemdefinition på baggrund af dine undersøgelser af nødvendige funktioner. Inkluder en factor analyse. Bør indeholde grunde til at inkludere eller ekskludere elementer som klasser, events og strukturer.

## Problemdomæne analyse:

Lav en liste af mulige elementer, konkluder om de skal med eller ej, hvorfor?

# Noter – forelæsning d. 25/09

Limited reduction:

Hvis der er høj kompleksitet, skal du skære det ned i mindre dele, løse disse og samle det bagefter. Fx interface, model. Arbejd eksperimentalt. Løfter usikkerhed.

Hvis du vil reducere usikkerhed, kan du lave prototyper, dette gør usikkerhed lavere, men forhøjer kompleksitet.

Problemdomæne: det der hjælper brugeren.

App domæne: det arbejde han rent faktisk gør.

Problemdomæne analyse:

Man starter med at lave en oversigt – klassediagram – og fylder så detaljerne ud.

Resultater:

For hver klasse skal der være et ”behavioral pattern”.

Du kan bruge event table. Viser sammenhæng mellem events og klasser. Det er lige meget om man skriver events horisontalt og klasser vertikalt eller omvendt. Handler om antallet, og hvordan det står bedst på papiret. Kun nøgleklasser. Underklasserne kommer når du begynder at strukturere systemet. En af problemerne i analysen er at finde ud af hvad hovedklasserne i problemdomænet egentlig er.

Et eventnavn skal være unikt – ikke kun i egen klasse, men i det hele taget.

Ikke diskuter under brainstorm. Først skriv alle idéer ned, og derefter diskuter.

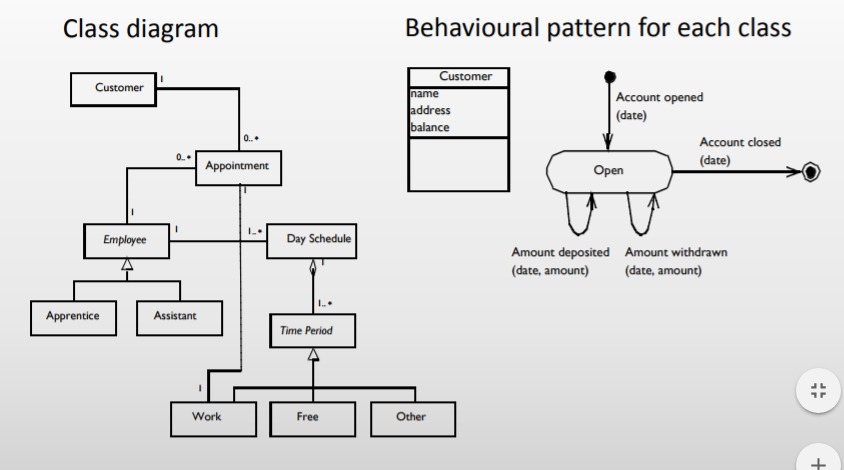
# Forelæsning d. 2/10/2017

Man beskriver klasser i problemdomænet ud fra de events de er involveret i.

Et en struktur skal kun indeholde de attributter der er relevant for problemdomænet.

## Klasse struktur:

Resultatet af dette vil blive et klassediagram.



Klassestruktur er mellem beskrivelser af klasserne.

Der hvor der peger en pil op, er når de er nedarvede.

Is-a relation – hvis man kan sige at en ”assistant” er en ”Employee”, er der en nedarvet relation.

I dette diagramsystem, når klassen er i kursiv er den abstrakt, og når den ikke har er den ikke.

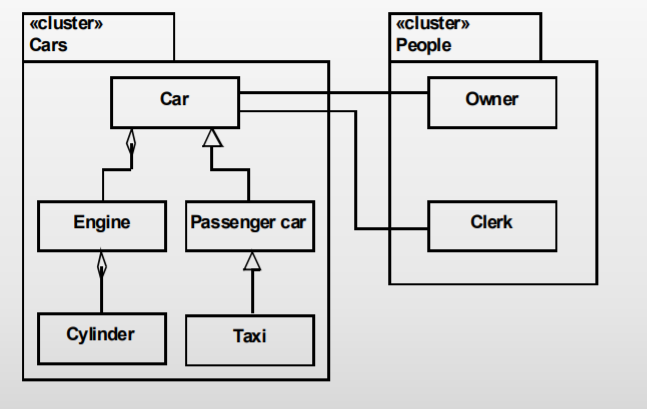
Når der ikke er en pil, er de forbundet af events.

Generalization class er superklasser

Specialixation class er subklasser

Hvis der skal være subklasser, skal superklassen være nødt til at dele sig op, fordi de forskellige subklasser er involveret i forskellige events.

### Cluster:



Sorter klasser i ting der har sammenhæng. Så er det lettere at forklare.

Klasser i en cluster er typisk konnected af en generalisation eller aggregation struktur. Relationer mellem klasser i forskellige clusters er typisk association

### Generalisation

Superklasse, den generelle klasse, beskriver attributter subklaser, en gruppe af specialiserede klasser, har til fælles.

## Objekt struktur:

Beskriver dynamisk relation mellem objekter i problemdomæne. De er beskrevet på klasse niveau med en tildelt ”multiplicity” proporty

### Aggregation:

Beskriver at et objekt er fundementalt, og er en del af det andet.

Decompotion er det modsatte af aggregation, fx kan en bil blive decomposed til et antal hjul, en motor og en krop.

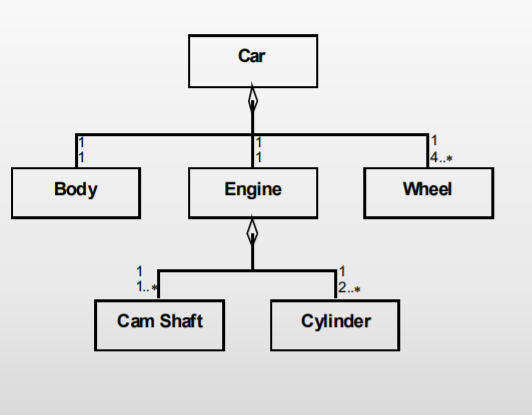
Der er tre typer relevante aggregations:

Whole part: helheden er en sum af delene. Tilføjes eller fjernes der dele, vil fundamentet ændre sig.

Container content: helheden er en beholder for delene. Ændres eller fjernes en del, ændrer helheden sig ikke.

Union – member: Helheden er organiseret af en union af members. De kan fjernes og tilføjes, men der er et minimum da det ikke giver mening uden dette.

#### Physical aggrigation



Relation mellem forskellige objekter i problemdomænet.

Klassen består af… (Has-a, owns-a)

Det handler ikke om sub- og superklasser, men hvad en klasse består af.

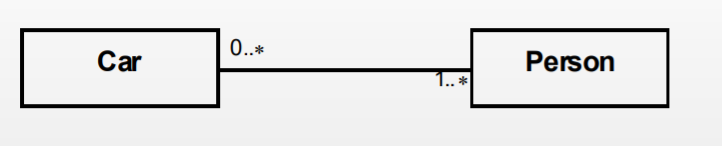
”Engine” er en del af ”car”, ”car” har en…

Det øverste tal betyder hvor mange elementet tilhører, den nederste beskriver hvor mange der er i.

Når der står ..\* betyder det eller flere.

Der er altså MINDST 2 cylinere i engine, den hører til ét sted, mindst 4 hjul i bilen, den hører også til ét sted. Der er præcis én krop i bilen, den hører til et sted.

Det betyder at en basal del af bilen er at den har en motor.



En bil kan være relateret til én eller flere personer, en person kan være relateret til nul eller flere biler, da man kan have x antal biler.

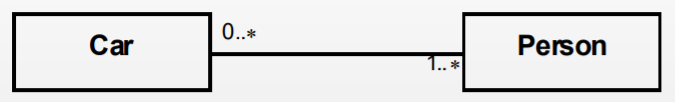
##### Containter-contents!!!:

##### Union – member!!!:

### Association:

En meningsfyldt relation mellem nogle objekter.

Objekter der er assosierede med hinanden, men ikke en del af. Bruges ofte når aggregation ville vise en for tæt sammenhæng. Man kan sige ”associated with”



Ingen pil.

Det er ofte når man har glemt en klasse til at beskrive sammenhængen.

Dette gøres med et ”relation” mønster.

## Find strukturer:

1. Find kandidater – find abstrakte statiske relationer mellem klasser. Find konkrete, dynamiske relationer mellem objekter.

### Indetificer generaliseringer:

Tag alle klasser og afgør om en er indeholdt i en anden. Findes en relevant generalisering mellem to klasser? Tag hver klasse og prøv at finde en relevant generalisering, fx ”arbejdsopgaver” for ansat.

### Indentificer aggregation:

Se i par om objekterne i den ene klasse er indeholdt i den anden. Er det relevant at lave en overklasse at indeholde to klasser i? Kan en klasse deles op til at indeholde flere klasser?

### Identificer association:

Se på de sidste klasser. Er de meningsfuldt relaterede?

Skal laves hvis vi skal overvåge, administrere eller kontrollere relationer mellem objekter der ikke ellers er relaterede.

### Identificer clusters:

Organiser konceptuelt relaterede klasser til clusters.

Brug andre strukturelle relationer til at gennemskue sammenhænge. Er som at putte i mapper. En klasse må kun være i en cluster. Kan du ikke bestemme dig, så lav en yderligere cluster.

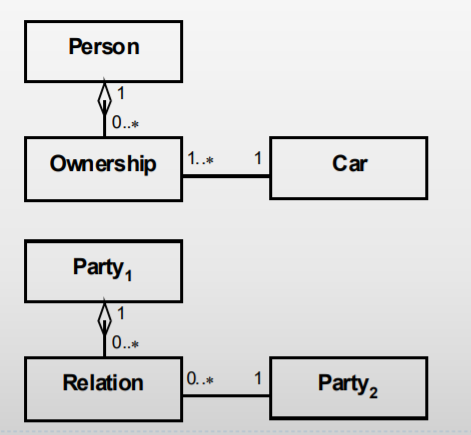
## Explore patterns:

En generaliseret beskrivelse af et problem og tilhørende løsning.

### Role pattern:

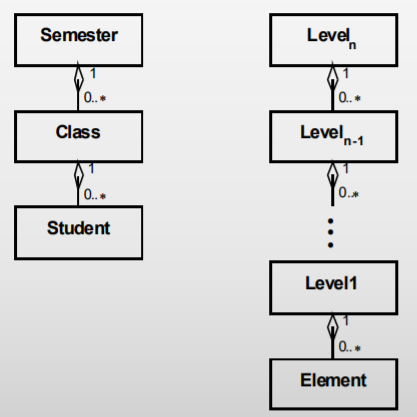
En situation hvor en enkelt person kan have flere roller. Generaliser fx ansatte og kunder til at være personer. Lav så aggregation og lav flere roller, fx ”kunde” og ”ansat” roller. Hver person kan så have en eller flere roller. Attributter alle rollerne har til fælles beskrives i klassen ”roller”, som hver rolle arver fra. Hvis rollerne ikke har noget til fælles behøves der ikke at lave en rolleklasse. Så laves bare normal aggregation.

### Relation pattern:



Det kan også være sådan at klasserne aggregerer relationsobjektet, eller at der kun er en association mellem dem og relationen, altså en løs relation.

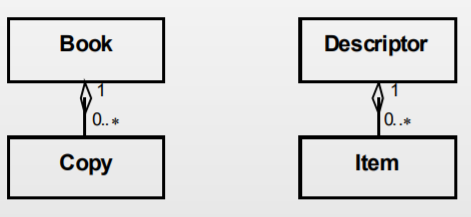
### Hiraki pattern:



Alle studerende er organiseret i klasser, og alle klasser er organiseret i semestre.

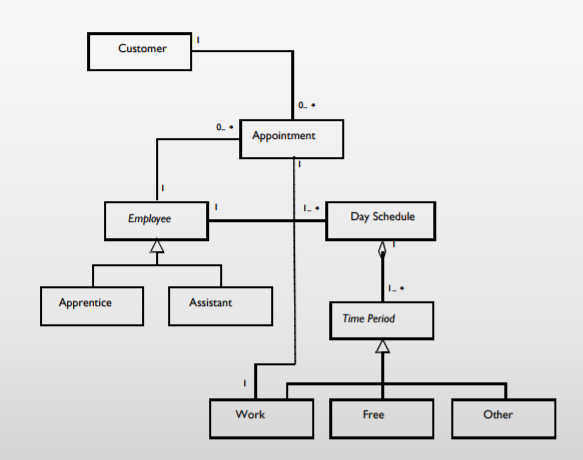
Man kan have en variation hvor et objekt kan høre til flere objekter over.

### Item-descriptor pattern:



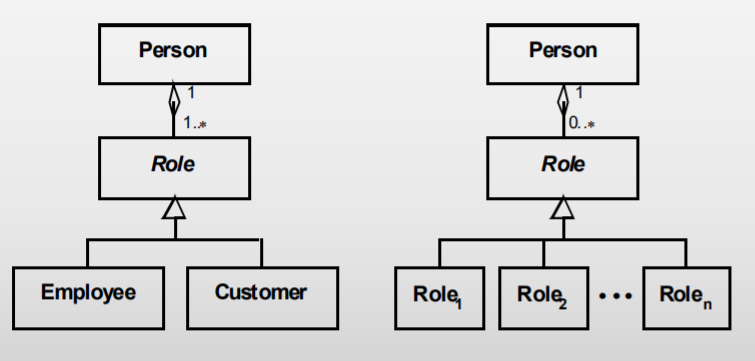
Descriptor har attributter delt i alle items.

## Generalisation vs. aggregation:

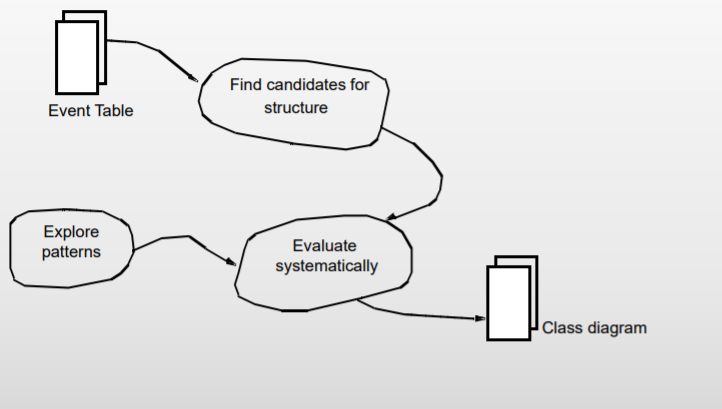


Man kan tilføje aggregation til class diagram hvis de er løst relateret.

Aggregation bruges så man kan flytte på hvor ting hører til, og antallet. Hvis man skal kunne skifte roller kan man bruge aggregation. I den tabel til venstre kan en person enten være ansat eller kunde, og det kan ikke skiftes dynamisk. Hvis den samme person skal være begge, skal den enten have to objekter eller også skal der laves en tredje der hedder costumer and empoyee. Dette er ikke praktisk at gøre. Man skal gøre det så dynamisk som muligt. Hver rolle skal have nogle events der er forskellig fra andre roller.

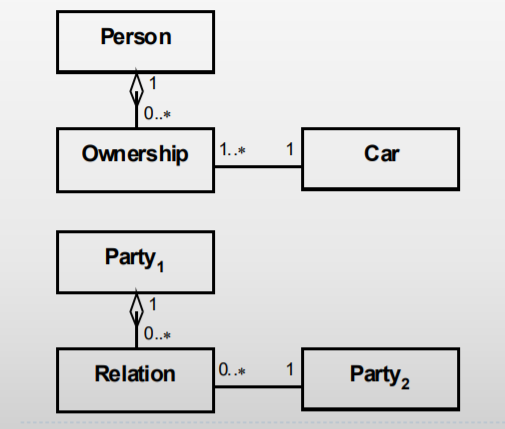


### Aktiviteter:



Patterns bliver vurderet op imod de forskellige kandidater i event table.

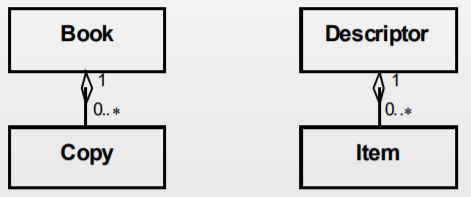
Klasser kan have relationer:



Man skal have en løsning til at se historikken mellem personer og bil. Her kan man have et ”ownership” objekt, der viser historikken i ejerskabet mellem en bil og en person – fra bilen bliver købt til den bliver solgt. På den måde er der et objekt knyttet til denne relation, og man kan i fremtiden se historikken.

### Item- descriptor:

En proporty af et objekt i en klasse (item) beskrives i en anden klasse(descriptor)



Hvis du har en bog, fx ”Illiaden”, og mange kopier af denne, skal man kunne se antallet af bøgerne, og hver enkelt af disse skal have en beskrivelse.

Skal bruges hvis der er forskelle mellem beskrivelsen af et objekt og det faktiske objekt, fx ved en bog.

Med et kursus kan man lave en afholdelse. Man beskriver altså hvad hele kurset går ud på, de ting der ikke ændrer sig, Herefter kan man snakke om de forskellige afholdelser. Hermed kan man se på de forskellige afholdelser af dette kursus, og få en individuel beskrivelse af disse.

## Systematisk evaluering:

Hvis man fx ikke skal se på kursusbeskrivelsen men kun afholdelsen, er dette ikke en relevant klasse.

Generalisering vs. aggregering (is-a/has-a). Se om du kan sige is-a eller has-a. (s. 85)

Man skal have lidt generalisering og lidt aggregering

Se på om de forskellige klasser kan eksistere uden hinanden, fx owner-car. De giver ikke mening uden hinanden.

Vil man sætte den ene over den anden, eller ved siden af hinanden, owner kan være over car, da owner ejer bil, men det kan også være noget andet. Det kan dog være vigtigt at have ”Ownership” klassen.

Man skal undgå at objekter skifter klasser, strukturer skal være simple, især i top levels. Undgå unødvendig generalisering og aggregering. Sammenlign med system definitionen.

Strukturer skal være helt sande. Navne, koncepter og strukturer reflektere den fremtidige brugers forståelse.

Strukturer skal være konceptuelt sande, bruges korrekt og simple.

Tilføjer en sammenhæng noget nyttigt til systemet?

### Aggregering vs. associering:

* Kan objekter eksistere uafhængigt af hinanden?
* Er de ligeligt rangeret?
* Kan relationen skifte fra et objekt til et andet.

Hvis man svarer ja til min to, kan associering overvejes.

## Udfordringer:

Det er svært at vælge den rigtige struktur. Du kan prøve hver af dem, og brug kriterier på begge klasser for at vælge struktur

Man kan nemt få for mange strukturer. Prøv at simulere funktioner, og se om du kan komme til det relevante objekt.

For hvert par af objekter kan man se om det er relevant at have et relationship diagram eller aggregation.

Class structure:

Dette handler om hvordan man kan lave et klassediagram, og forskellige modeller til dette

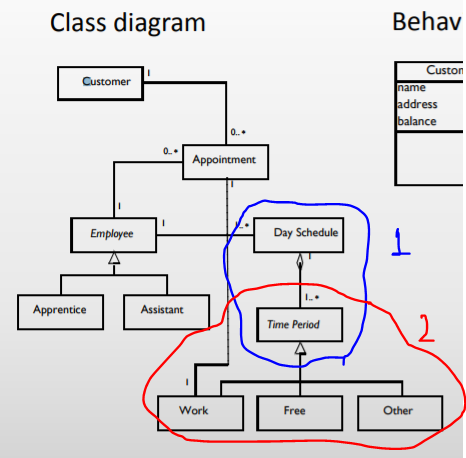
Cluster: her kan du indkapsle en gruppe af elementer der hører sammen for at få bedre overblik.

**Generalization**: et klassisk klassediagram der viser nedarvning. hvad hører til hvad? Viser nedarvning og ting der er relaterede. Nedarvning vises med pile, og sammenspil vises med streger. Er ”is-a” struktur, er en tommelfinger en finger?

**Objekt structures**

//**Association**: hvad hører til hvad? Viser nedarvning og ting der er relaterede. Nedarvning vises med pile, og sammenspil vises med streger. Er ”is-a” struktur, er en tommelfinger en finger?

**Aggregation**: Handler om hvad ting indeholder. Vises med en diamantpil, og til hvert tilhørende element et tal øverst der siger at tingen indeholdt relaterer til x elementer af de objekter vi ser på, og nederst hvor mange af disse elementer der er i objektet.

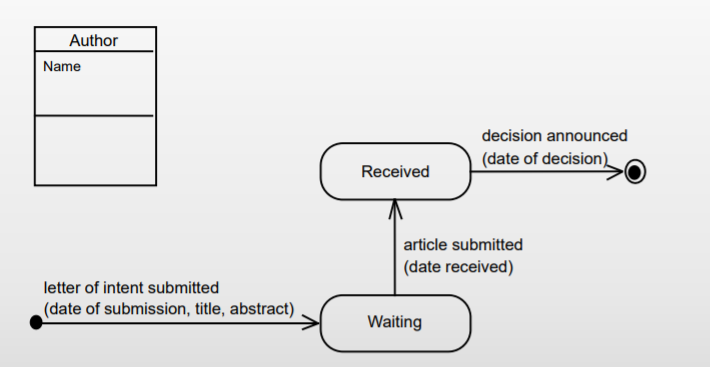


Her viser ”diamanten” ved 1 mellem ”Day Schedule”(DS) og ”Time Periode”(TP), at DS indeholder et antal TP. 1 tallet under DS viser at TP hører til en DS, det nederste 1..\* over TP viser at der er en til mange TP DS, dette er aggregation. I to viser pilen mellem TP, work, free og other at de er nedarvet fra TP, dette er generalisation.

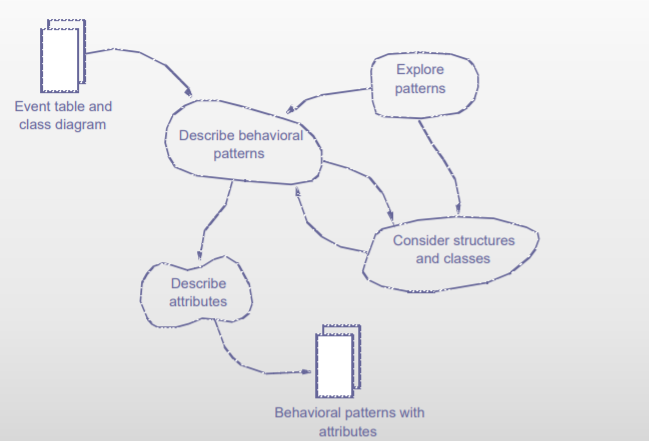
# Forelæsning d. 04-10-2017 Behavior

## Resultat

Event: noget man vil huske i modellen.



I dette eksempel er der en artikel som objekt. En artikel ”fødes” når en forfatter sender et ”letter of intent”. Man venter så indtil artiklen er afleveret. Når den så er modtaget bliver beslutningen offentliggjort.



Behavioral pattern vs. explore patterns.

## Event trace:

Behavior: historie et objekt har af handlinger. Her kan man lave event trace. De ting der har været ved objektet (relateret til systemet) er mulige events. Så kan man se på om der skal være en bestemt rækkefølge mellem dem (sid ned, stå op). Så kan man se på om der er et limit ved events (hæv penge, sæt ind, man kan ikke hæve flere penge end der er til rådighed).

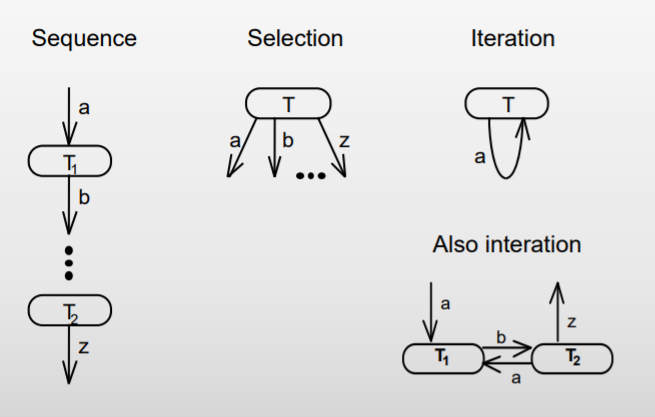
Spørg hvad der fører til at et objekt bliver oprettet. Hvad skal der til. Hvilke events fører til at et objekt er født. Dette er grupperet som den selektion der fører til fødsel af objektet. Samme med ”død” af objektet.

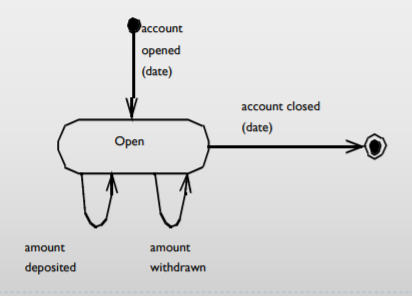
Hvilke events sker sammen i sekvens?

Er der nogen alternative events?

Kan den ske mere end en gang?

Er den overordnede form (objekterne) strukturerede elle ikke? Ved man når én event er sket hvad der vil ske næste gang.





*Bankkonto*

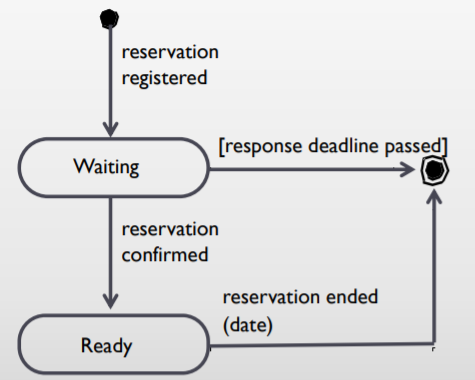
### Kontrolstruktur:

#### Sekvens:

En række events hvor man til sidst afslutter muligheden for at ændre mere på objektet. I bankkonto: åben, lav et antal deposit og withdrawn, og luk konto.

#### Selektion:

Som en if-statement. Flere forskellige events der kan fører til lukning af mulighed for at gentage event. I bankkonto ville det være hvis der var flere måder at lukke konto.



Her vil mulighederne fra ”waiting” være ”ready” eller ”response deadline passed”

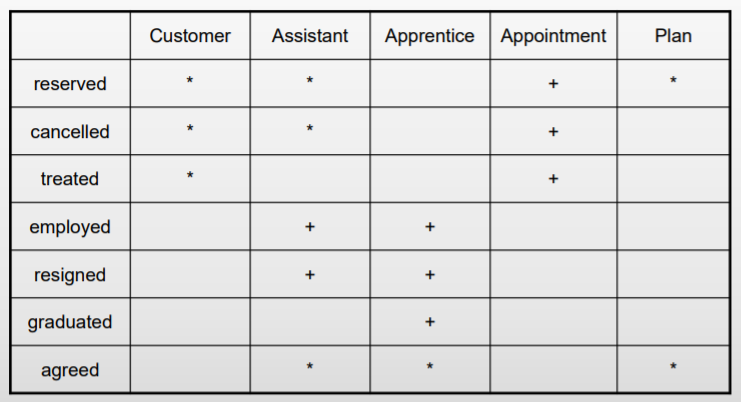
#### Iteration:

Handlinger der sker på kun ét objekt der ikke ændrer stadie. Den bliver ved med at ske. En handling kan også blive ved med at ske, og så på et tidspunkt stoppe muligheden for at gentage. I bankeksempel kan det være deposit 🡪 deposit 🡪 withdraw 🡪 deposit…

### Ulovlig event:

En sekvens der ikke kan ske.

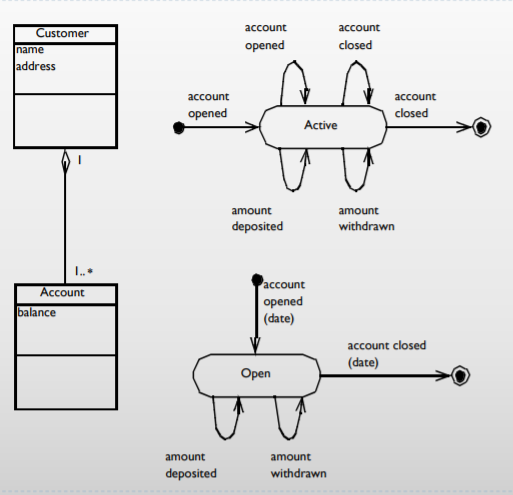
I bankeksempel ville det være close før du har åbnet osv. Handler om hvilke sekvenser der er lovlige, som åben 🡪 deposit 🡪 withdraw 🡪 close, og ulovlige 🡪 åben 🡪 luk 🡪 deposit.



Denne udvidning af eventtable viser om det kan ske én eller flere gange for et objekt. + er én gang, \* er flere gange.

Diagrammerne for events omkring et objekt hedder et ”State chart diagram”, det vises fx i bankeksemplet.

Har man to forskellige diagrammer, er en sekvens kun lovlig hvis den kan foregå i begge.

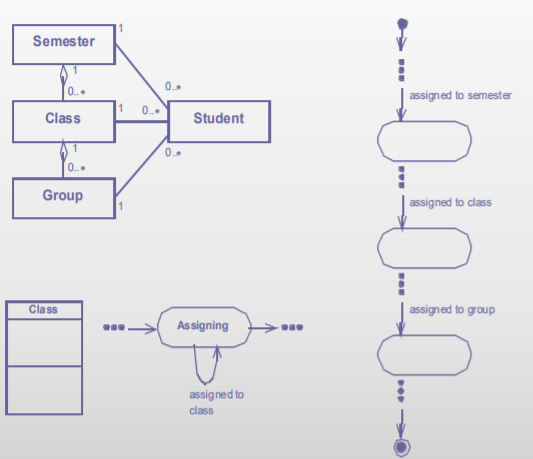


En account kan være åben, men hvis den er lukket kan den ikke åbnes igen, mens en kunde kan åbne flere accounts. Hvis der er en event der indeholder flere objekter men ikke er lovlig i de andre er den ulovlig.

Hvis de har en event til fælles kan den ene være en generalisering af den anden.

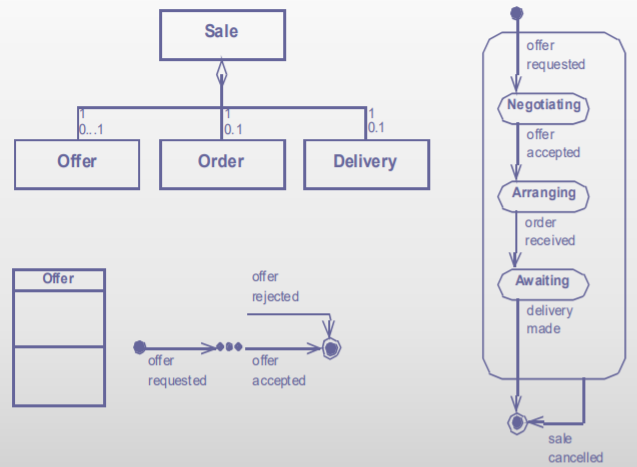
## Explore pattern:

### Stepwise relation:



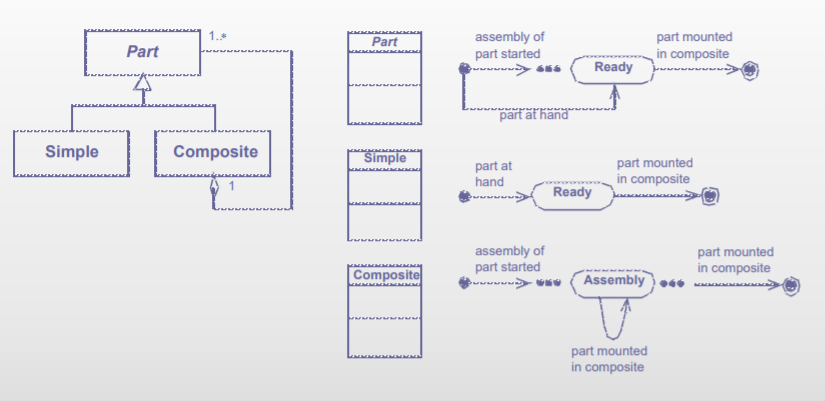
Der skal være et hiraki.

### Stepwise role:



Man skal have noget der ændrer state. Man dele hvert element op i flere events og laver et forløb for hver, og derefter sætter man dem sammen. Boksen rundt om sekvensen siger at ved ethvert stadie kan handlen blive aflyst.

### Composite:



Sammensat af flere dele. Kan beskrive mange forskellige ting. Her kan man se at en del enten kan være simpel eller sammensat af mange.

# Forelæsning d. 04/10/2017 Usage

Problemdomænet beskrives med en objekt orienteret model.

Man skal også beskrive hvad modellen er brugt til i app domænet af actors.

## Application Domain Analysis: results:

Hvem er brugeren? Du er kun en bruger hvis du bruger systemet. Hvis kunden lejer en film og selv checker den ind (som selvbetjening i biblioteket), er kunden bruger, hvis kunden går hen til ansat og får den til at registrere bogen, er denne brugeren.

### Actors and use cases:

#### Actor:

Dette er brugeren og andre systemet der bruger dette system.

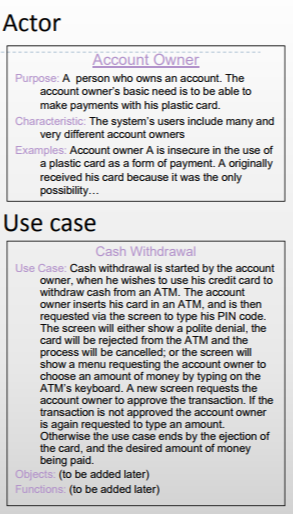
##### Identificer actor:

Hvem er actors? Hvilke menneskelige actors, hvilke systemer?

Lav forskellige eksempler ved samme actor.

#### Use cases:

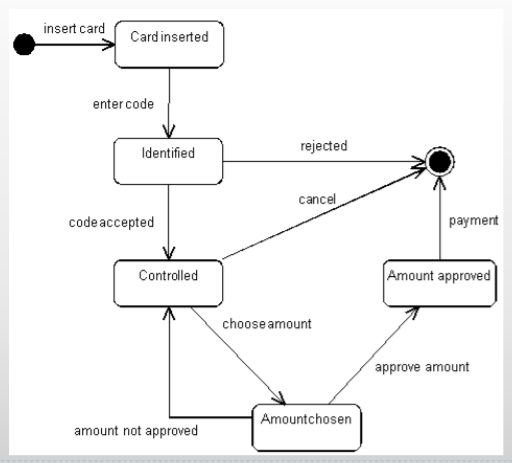
Eksempler på situationer systemet vil bruges i.



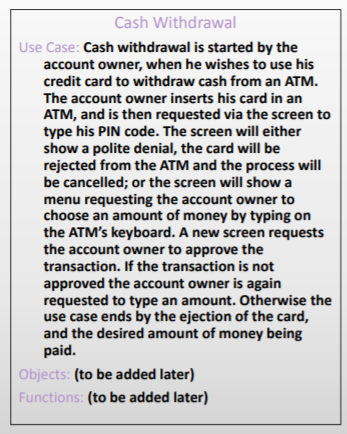
##### Identificer use case:

Beskrivelsen kan foregå med tekst eller med diagram.

Fordelen ved tekst er at det er lettere at relatere til for brugeren end et diagram. Ulempen er at det er svært hurtigt at overskue. Tekst skal skrives overskueligt og hurtigt at forstå, men vil være svær at se om der er problemer. ¨



Figur 1 Diagram type



Figur 2 Tekst type

Diagram typen gør det lettere at overskue hvor der er fejl, og hvad der faktisk sker. De forskellige bokse indeholder events, mens det tekst udenfor er funktioner.

### User interfaces (bliver dækket i design):

Gør actors i stand til at kommunikere med systemet.

### Functions:

### Stabil vs transient properties:



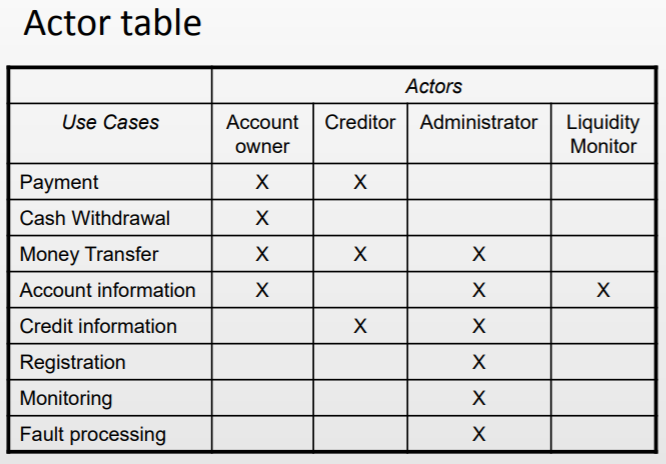
Modellen i et system skrifter ikke så let, da et system i bund og grund mange gange vil gøre det samme. Interfaces skal dog ændres ofte. Funktioner ligger lidt ind imellem.

### Oversigt:

Formålet er at vise hvad der sker i et system udefra. Her vil kriterierne være andre end i problemdomænet.

### Actor table:

Hvad kan blive gjort af hver. IKKE en eventtabel, hvem har ret til at gøre dette, hvem har ikke:



### Actor:

Ved hver actor skal beskrives tre ting:

* Purpose: Hvorfor er denne i systemet? Hvad gør de?
* Karakteristika: Hvad gør dem specielle?
* Eksempel: Fortæl et eksempel på kontekst

### USe case:

Use case:

Objects:

Functions:

### Work task:

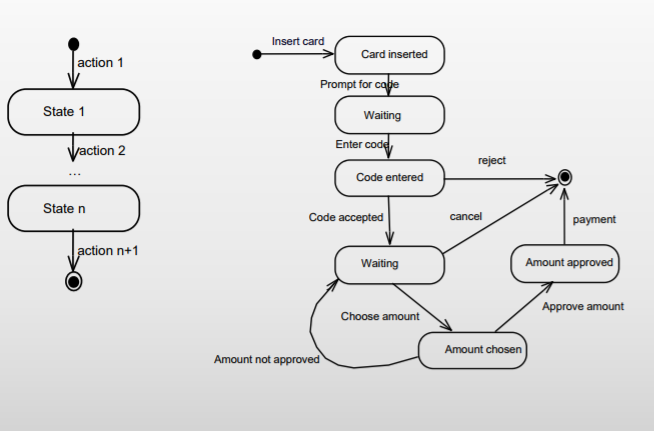
Hvilke opgaver er der i App domæne?

Hvordan er arbejdsfordelingen?

Hvordan er de forskellige opgaver fordelt?

### Explore pattern:

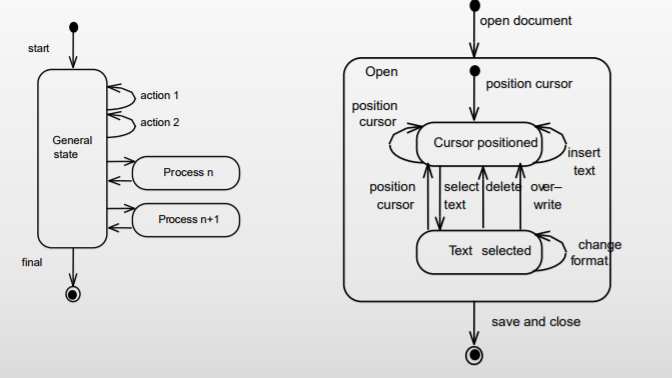
#### Procedural:



I den første går man step-by-step. Dette kan fx være at registrere en bruger. Man skal kunne liste op specifikt hvad der skal til for at noget er afsluttet.

Hoveddelen i den til høre har hovedsageligt procedural

#### Material

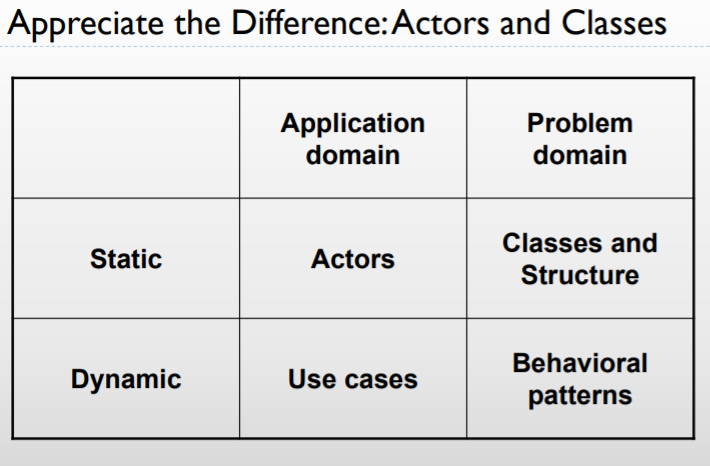


Ingen bestemt rækkefølge. Man ved ikke helt sikkert hvad der sker når man er inde i handlingen. Man kan både gå til helt andre processer eller bare andre handlinger i systemet.

### Evaluer systematisk:

Hold fokus på at holde overblik, ikke for meget på detaljer. Få stadig alt relevant med.

Hvordan interagerer actors med systemet?



Use cases: hvordan opfører actors sig når de bruger systemet

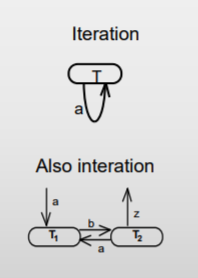
Behavioral patterns: hvordan interagerer klasser i systemet

# Forelæsning d. 11/10/2017 – functions

## Besværligheder:

Final state af et objekt: I analyse – objektet er kommet i et stadie hvor man ikke kan gøre mere ved det, men du kan stadig læse info fra det. I design, alle objekter i problemdomænet skal på et tidspunkt fjernet. I design skal vi vælge hvor længe vi vil gemme information – lov om max 5 år.

Iterative events: Hvornår et det en iterativ event i eventtable, \*m eller ikke, +. Skal det beskrives i statechart diagram med loop pil eller med en ændring i stadiet. Ting der kan ske en gang eller flere:

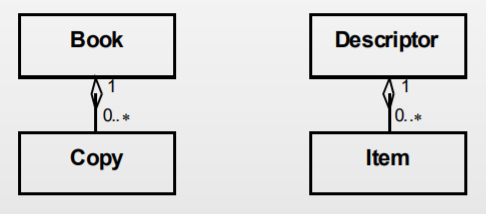


Med nr. to skal den i et andet stadie hvor nogle funktioner er blokerede, men den kan gå tilbage og have de andre mulige events. Ved den første er det når der sker en event hvor objektet stadig har adgang til samme events, fx med indkøb. Nr. et er normalt for abstrakte objekter, mens nr. 2 r for konkrete objekter. Der skal findes en god balance mellem hvornår man bruger de to forskellige objekter.

Hvornår skal de forskellige itterationstyper bruges:

Type to skal kun bruges hvis man sørger for at noget ikke kan ske i den ene tilstand der kan ske i den anden. Ved modellen nedenunder kan bogen få kopier tilføjet og fjernet uden at ændre nogen tilgængelige funktioner, her er det nr. et, mens når du låner en bog kan den ikke lånes før den er afleveret. Her er det nr. to. En descriptor vil typisk have type et, mens item har nummer to. Det er blandt andet fordi descriptor er mere abstrakt.

Item descriptor:



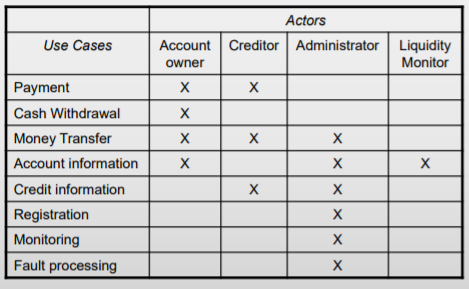
Hvis der fx er forskellige kurser, ville en descriptor repræsentere kurset i forhold til kursusbeskrivelsen, mens et item er hver semesterkursus. Ved en ny kursusbeskrivelse laver man et nyt item. Man kan ændre et descriptor object hvis noget ændrer sig, der ændrer på hvert item, men hvis det ikke repræsenterer det samme objekt mere vil der skulle laves en ny instans. Descriptor repræsenterer en kategori, fx ”bibelen”, og item repræcenterer de specifikke objekter.

Ved book(descriptor): opretter, fjernet

Item: Borrowd, returned.

Common events(både til copy og descriptor): copy bought, copy destroyed.

Actor: Hvad er en actor og hvad er ikke. En abstraktion af en bruger eller et andet system der direkter interagerer med systemet og ikke individuelt igennem andre brugere eller systemer.



Her er ”Liquidity monitor” en actor da det er et andet system der interagerer med systemet.

Actors i problemdomænet bruger information fra objekter i problemdomænet.

## Results:

Actor:

* Purpose
* Chacacteristic
* Example

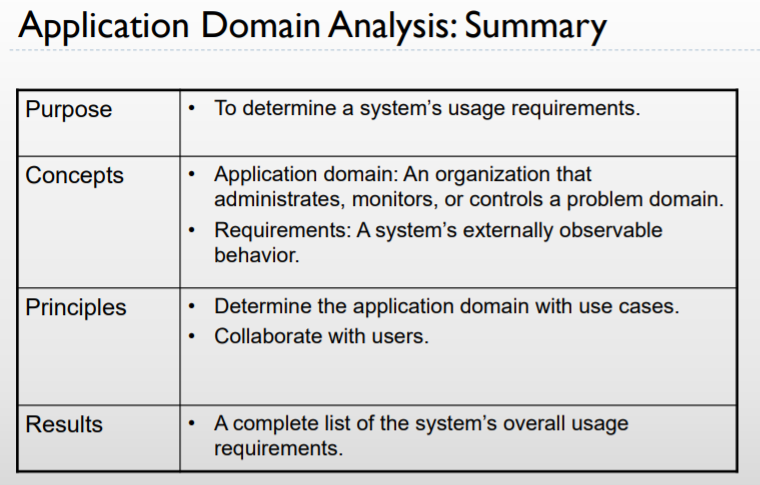
Use case:

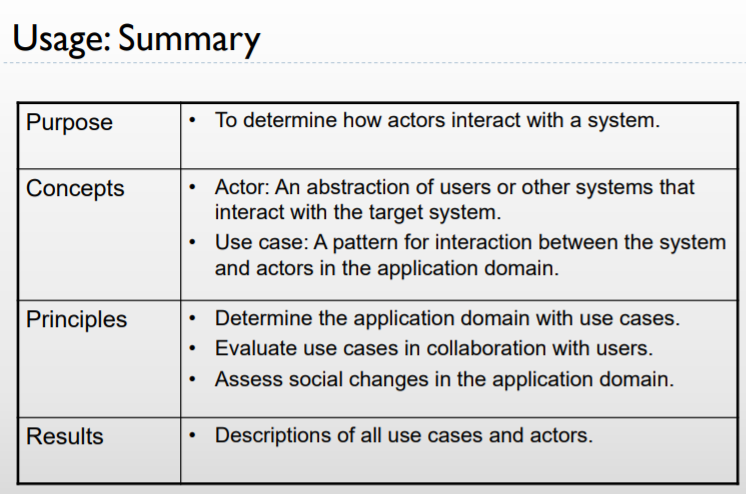
* Use case
* Objects
* Functions

Hvis en usecase mangler functions bør man genoverveje det.

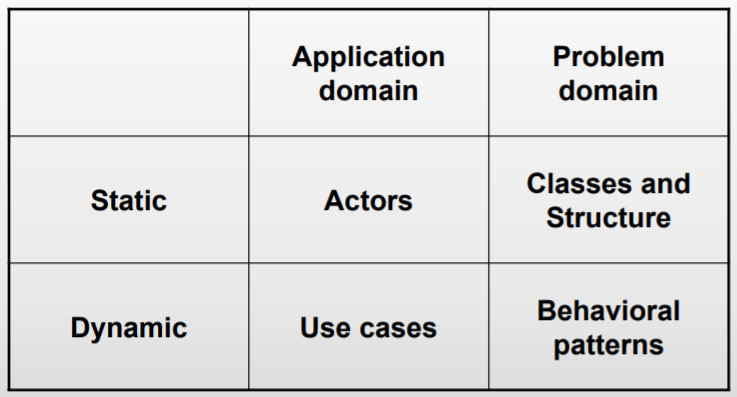
Use case kan beskrives i tekst eller statediagrams

Du finder actors og use cases og analyserer dem.





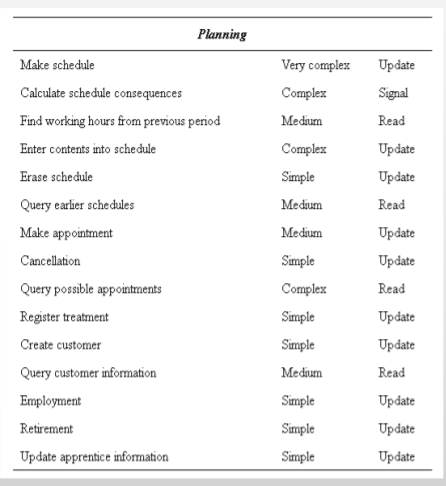
Forskel på app og problemdomæne:



Brugeren af systemet skal ikke være i systemet med mindre man vil overvåge eller kontrollere dem. De overstående begreber beskriver forskellen mellem de forskelige domæner

## Functions:

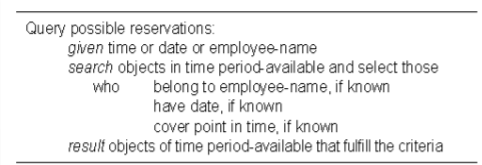
### Resultater:



Det primære resultat er en komplet liste af funktioner. Det skal altså være alle funktioner i systemet.

Det sekundære resultat er at specificere hvor kompleks de forskellige funktioner er i forhold til hinanden. Dette fortælle hvor risikabelt en funktion er.

Nogle funktioner er meget lette at gøre og nogle er besværlige. Man behøver ikke en lang beskrivelse af funktionen.



Beskrivelse og resultat af en funktion, altså hvad gør den, og hvad skal det resultere i. Beskriv til det fjerne usikkerhed.

En funktion: at tage information med modellen til actor.

Spørgsmål:

* Hvem sætter funktionen i gang
* Hvor går data.

Noget er sket, vi vil opdatere modellen til at reflektere dette.

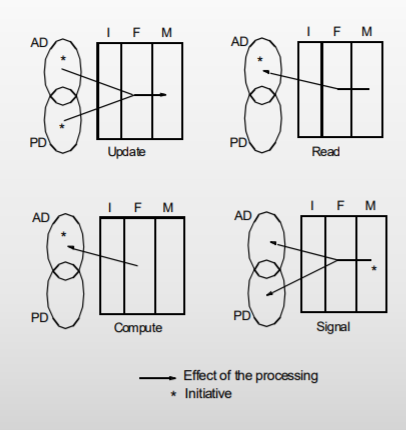
Når der måles en temperatur i drivhus, opdateres modellen og en funktion sættes i gang.

Når en bruger sender registrerings information opdateres modellen. Den kan altså sættes i gang af begge domæner.

Meningen med modellen er at give actor information fra problemdomænet.

### Functions og function typer:

Update, read, compute, signal.



Update: noget bliver opdateret i modellen fra et af domænerne.

Read: actor modtager information.

Signal: når man har sendt en update på grund af noget modtaget information i modellen. Fx kan modellen hvis temperaturen i drivhus bliver forkert kan modellen enten signalere gartner eller åbne/ lukke vindue.

Compute: actor sender information og modtager resultat uden inkludering af problemdomæne!?!?

Funktionen binder modellen til domænerne.

### Find funktioner:

Update: det er en update funktion når noget i et af domænerne bare sender information til modellen. Hvordan bliver en event opserveret i problemdomænet? Hvordan bliver use case understøttet af update funkction? Hvilke objekter, attributter og objektstukturer bliver påvirket af event, hvilke requirements har dette af update funktion

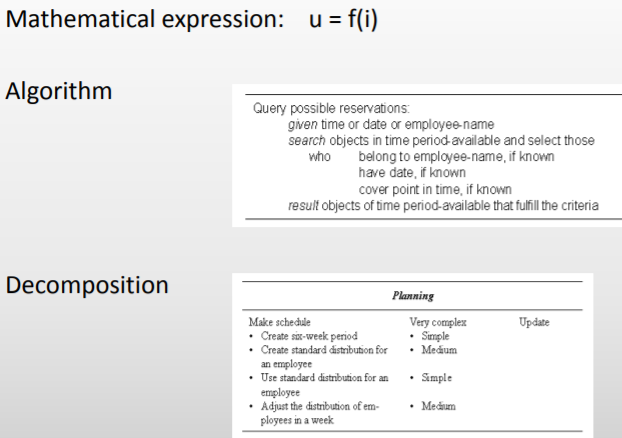
Read: Et behov for information i app domæne om problemdomæne. Actor perspektiv – hvad har actor brug for at vide om modellen? Hvilke read funktioner er der brug for. Redundant er skidt her. Model perspektiv – hvilke objekter og strukturer har actor brug for, hvilke read funktioner er der brug for? Redundant kan bruges til at tjekke ting mod hinanden.

Compute: generer yderligere info.

Signaling: kritiske states af model. Hvad er de kritiske states jeg vil signaleres om? For høj temperatur.

Gå gennem use cases og se hvilke funktionstyper dette har brug for.

### Specificer komplekse funktioner:



Lige meget hvordan man gør det. Vær sikker på at du kan programmere det. Når du er sikker på at du kan programmere dette og det opfylder klassediagram, er du færdig.

### Evaluer systematisk: !!

Bed bruger om at se liste af funktioner.

### Udfordringer:

Er det noget jeg vil huske? Er det virkelig relevant? Ikke indsaml info der ikke er vigtig.

Events – use cases – funktioner. Det er ikke det samme. De hører dog sammen. De passer til forskellige domæner. Hvis vi vil have information om noget er det en event, ellers er det en funktion?!? Hold dem hver for sig. De hører til forskellige dele af systemet.

Event: Ordered: En bruger har underskrevet noget

Use case: enter order: en bruger bestiller noget

Fuction: Create order: I systemets model bliver et objekt af klassen order oprettet.