# 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.

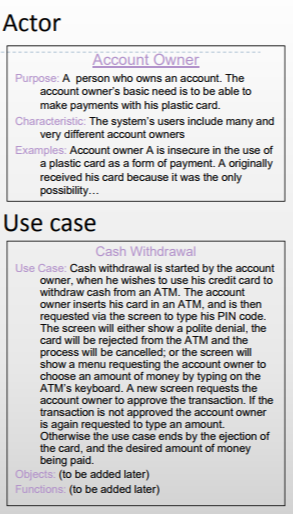
##### Identificer actor:

Hvem er actors? Hvilke menneskelige actors, hvilke systemer?

Lav forskellige eksempler ved samme actor.

#### Use cases:

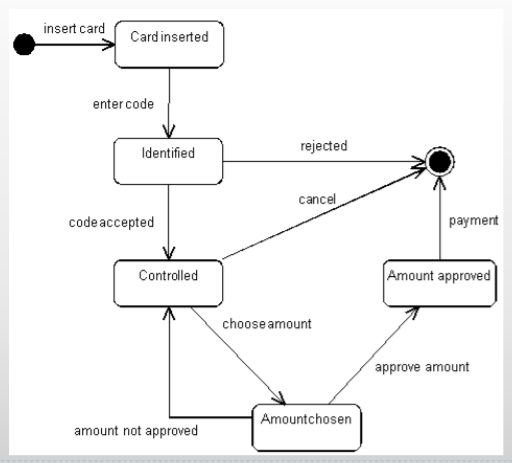
Eksempler på situationer systemet vil bruges i.



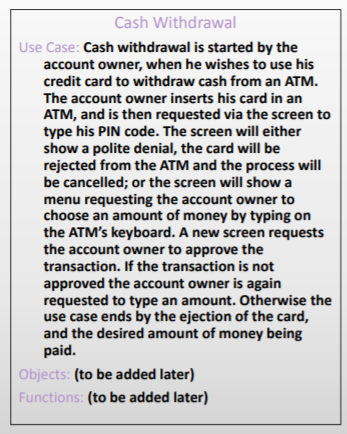
##### Identificer use case:

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:



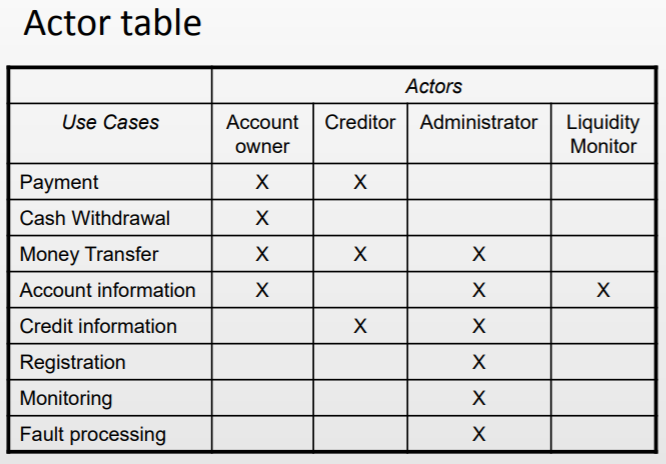
Modellen i et system skrifter ikke så let, da et system i bund og grund mange gange vil gøre det samme. Interfaces skal dog ændres ofte. Funktioner ligger lidt ind imellem.

### 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:

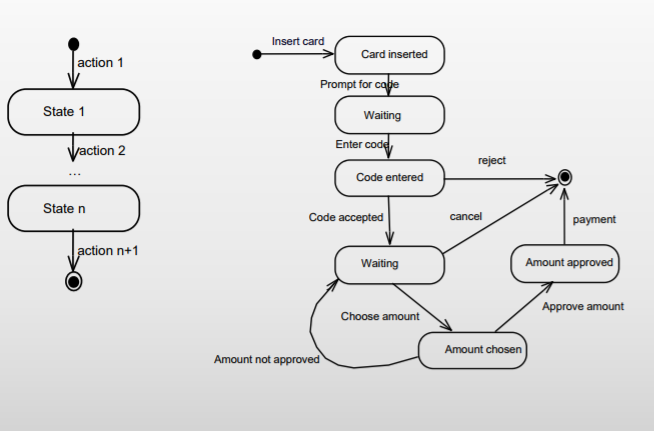
Hvilke opgaver er der i App domæne?

Hvordan er arbejdsfordelingen?

Hvordan er de forskellige opgaver fordelt?

### Explore pattern:

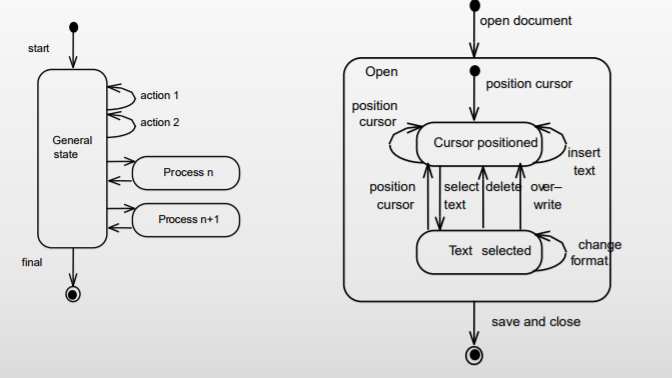
#### Procedural:



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

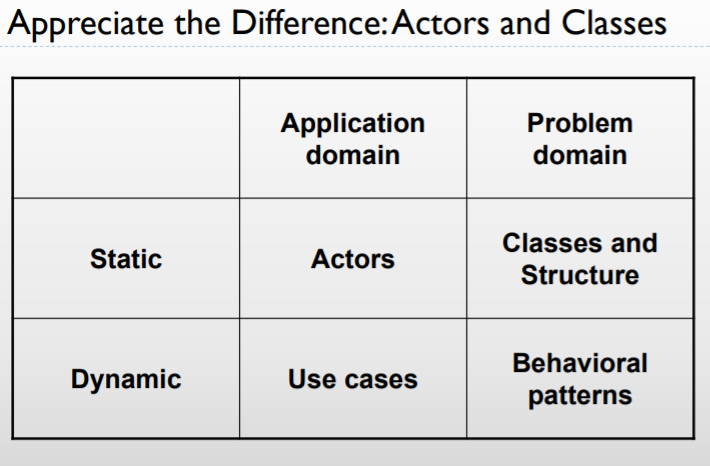


Ingen bestemt rækkefølge. Man ved ikke helt sikkert hvad der sker når man er inde i handlingen. Man kan både gå til helt andre processer eller bare andre handlinger i systemet.

### 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

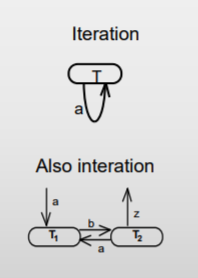
Behavioral patterns: hvordan interagerer klasser i systemet

# 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:

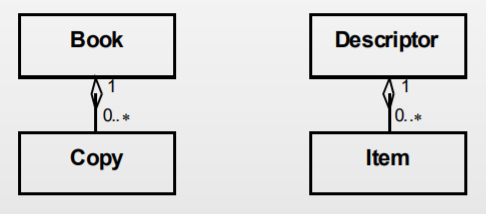


Med nr. to skal den i et andet stadie hvor nogle funktioner er blokerede, men den kan gå tilbage og have de andre mulige events. Ved den første er det når der sker en event hvor objektet stadig har adgang til samme events, fx med indkøb. Nr. et er normalt for abstrakte objekter, mens nr. 2 r for konkrete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



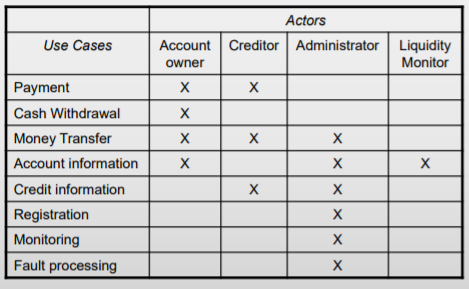
Hvis der fx er forskellige kurser, ville en descriptor repræsentere kurset i forhold til kursusbeskrivelsen, mens et item er hver semesterkursus. Ved en ny kursusbeskrivelse laver man et nyt item. Man kan ændre et descriptor object hvis noget ændrer sig, der ændrer på hvert item, men hvis det ikke repræsenterer 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

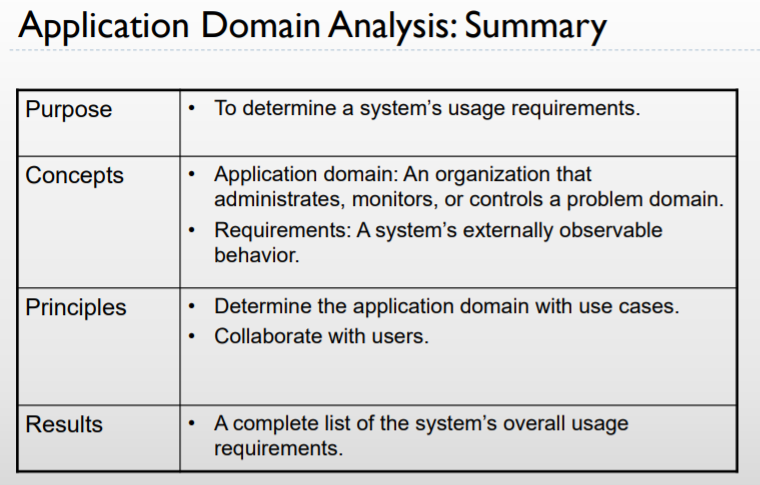
Use case:

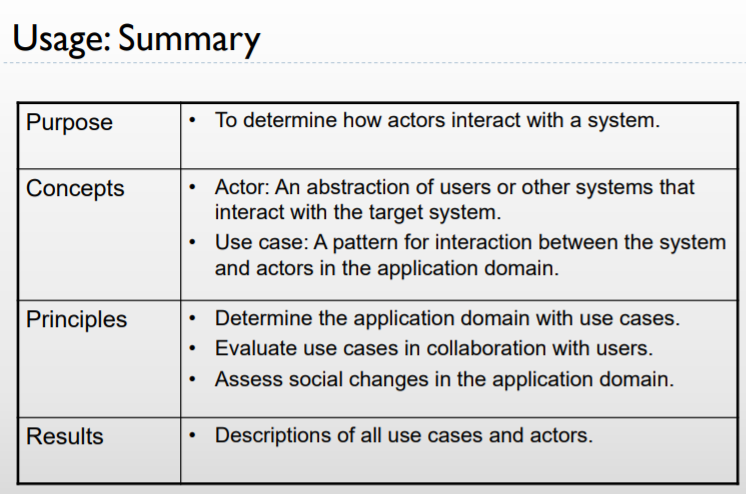
* Use case
* Objects
* Functions

Hvis en usecase mangler functions bør man genoverveje det.

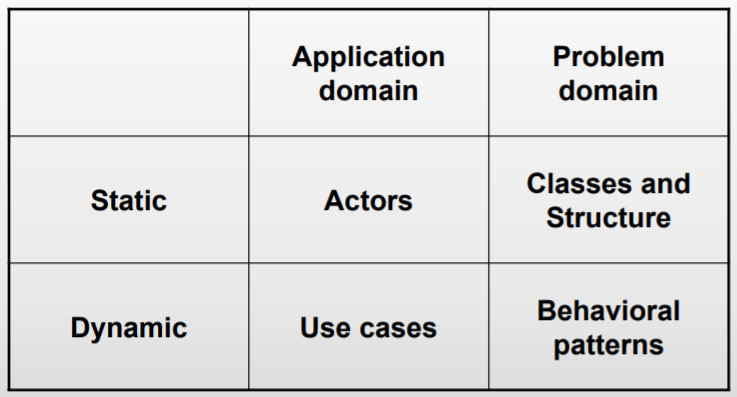
Use case kan beskrives i tekst eller statediagrams

Du finder actors og use cases og analyserer dem.





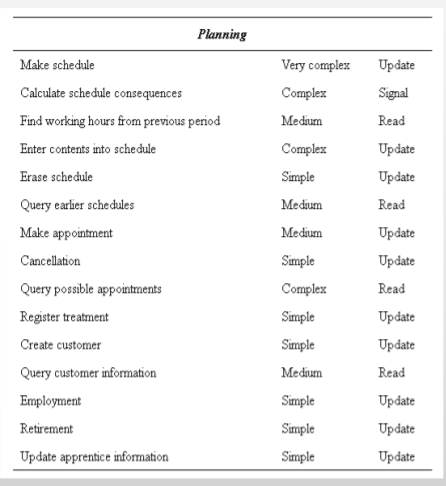
Forskel på app og problemdomæne:



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:

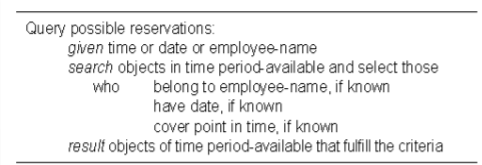
### Resultater:



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.

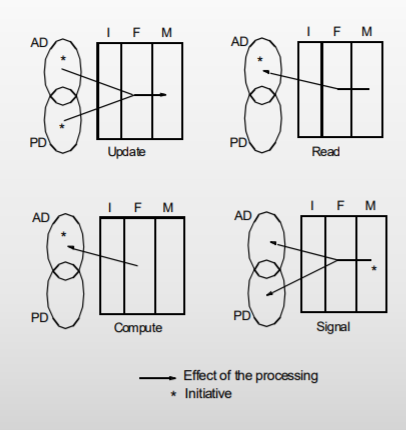
Når der måles en temperatur i drivhus, opdateres modellen og en funktion sættes i gang.

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

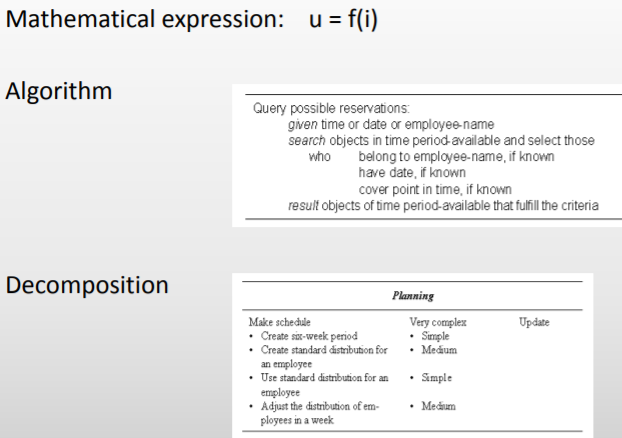
Read: Et behov for information i app domæne om problemdomæne. Actor perspektiv – hvad har actor brug for at vide om modellen? Hvilke read funktioner er der brug for. Redundant er skidt her. Model perspektiv – hvilke objekter og strukturer har actor brug for, hvilke read funktioner er der brug for? Redundant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