Abstraktion & Modellering

Palle Nowack
nowack@cse.au.dk
Center for Science Uddannelse, Aarhus Universitet

"Without abstraction we only know that everything is different."
Grady Booch

Introduktion **2**

Hvad er en model? **5**

Systemer & perspektiver **5**

Modelsystem & referentsystem **9**

Metoder **11**

Procesmodeller: vandfald og spiraler **13**

Eksempler **15**

Programmering ***15***

Repræsentation og digitalisering ***15***

Arkitektur ***16***

Referencer **18**

Appendix A: Modelleringskompetencer **19**

Introduktion

Øvelse: Kig på programmet til venstre i figur 1. Prøv at forstå, hvad programmet gør. Kig så på modellen til højre i samme figur. Som du sikkert hurtigt indser, er der en sammenhæng. Hvad synes du er nemmest at forstå? Hvorfor tror du, det er sådan?

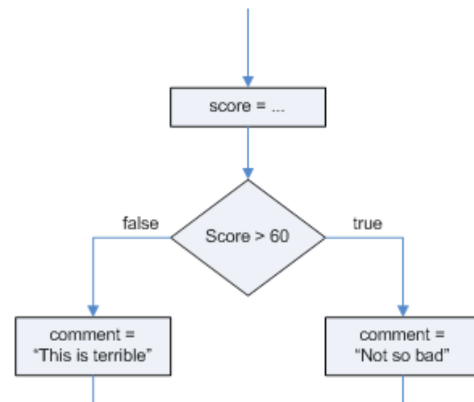
```java
// Description: Evaluate a test score.  Illustrates if statement.
// File  : if/EvalScore.java
// Author: Fred Swartz - 2007-04-09 - Placed in public domain.

import javax.swing.*;

public class EvalScore {
    public static void main(String[] args) {//... Input a score.
        String scoreStr = JOptionPane.showInputDialog(null, "Enter your score?");
        int score = Integer.parseInt(scoreStr);

        //... Create a message.
        String comment;   // Message to the user.
        if (score >= 60) {
            comment = "Not so bad";
        } else {
            comment = "This is terrible";
        }

        //... Output the message.
        JOptionPane.showMessageDialog(null, comment);
    }
}
```



Figur 1: Java kode eksempel og rutediagram[1]

Koden er naturligvis vanskelig at læse, hvis man ikke kender Java programmeringssproget, men bortset fra det, er der en anden central forskel, nemlig at rutedigrammet er en “abstraktion” over koden. Når vi abstraherer, fokuserer vi på visse egenskaber ved et fænomen, og ignorerer andre egenskaber. Koden til venstre skal kunne afvikles på en computer, og det indeholder derfor en masse bogholderi og detaljer, som computeren har brug for, for at kunne udføre programmet. Diagrammet til højre er beregnet til at skulle læses af mennesker, og dette tilfælde har vi valgt at fokusere på logikken i koden til venstre. Vi abstraherer således fra en lang række egenskaber ved koden: syntax, typer, parametre, input/output m.m. Derfor udtrykker rutediagrammet noget langt simplere end koden. Andre åbenlyse forskelle er, at koden er tekstuel (skal læse ovenfra og ned, venstre til højre), hvorimod diagrammet er grafisk. Nogle (de fleste?) har nemmere ved at forstå diagrammer end tekst.

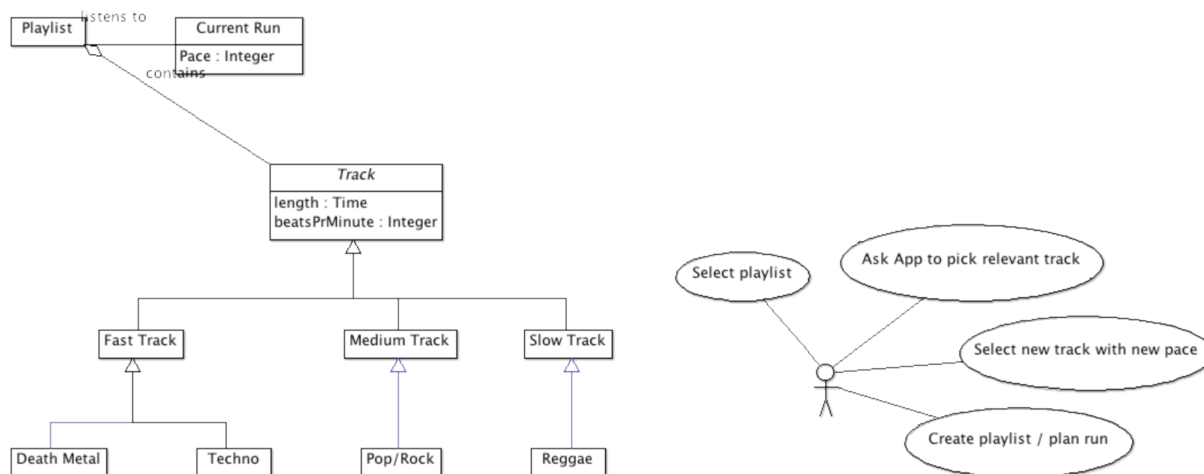
Forestil dig nu, vi skal udvikle en løbe-musik-app til en Smartphone. Når vi skal udvikle systemer, er det generelt en god idé at tale med de kommende brugere, så vi ikke kommer til at lave et system, ingen vil bruge. Vi har derfor interviewet Kim fra løbeklubben, der har bestilt systemet, og spurgt, hvad han kunne tænke sig:

“... jeg vil gerne have, at musikken tilpasser sig min løberytme. Altså, når jeg løber langsomt, skal systemet finde et langsomt nummer, som passer med mine skridt. Hvis jeg løber hurtigt skal den finde et hurtigt nummer. Så musikken skal hele tiden tilpasses den rytme, jeg løber i lige nu. ... Og den skal finde nye numre hver gang, jeg skifter tempo... ikke bare øge tempoet på den sang, jeg hører i forvejen, så Justin Bieber kommer til lyde endnu mere som Mickey Mouse.. det er ikke det jeg mener! ...Omvendt, så kunne det også være smart, hvis man kunne gøre det modsatte.. altså at jeg selv kunne vælge et nummer, som var langsomt, mellem, eller hurtigt.. hvis jeg nu havde lyst til at presse mig selv eller træne i et bestemt tempo en dag. Faktisk ville det være fedt, hvis jeg også kunne lave mine egne playlister hjemmefra, så jeg kunne lægge ud med et langsomt nummer, øge

[1] Fra http://www.leepoint.net/notes-java/flow/if/10if-general.html

rytmen løbende gennem 2-3 numre, maksimal rytme i ét nummer, og så afslutte med et langsomt nummer til sidst...”

Kig på modellerne i figur 2. De er begge udtrykt i et grafisk modelleringsprog, der hedder UML (Unified Modeling Language). UML er et meget omfattende sprog, hvor der findes et hav af notationer til at beskrive mange forskellige dele af og syn på modeller i forbindelse med systemudvikling. De to illustrerede er nogle af de mere almindelige. Det er ikke meningen, at du skal kunne forstå detaljerne i eksemplerne, men du skal have en idé om, hvad der forsøges udtrykt.



Figur 2: UML klassediagram (til venstre) og use-case diagram (til højre).

I klassediagrammet til venstre har jeg gengivet de begreber (klasser), som vores bruger anvendte, da han talte om det ønskede system. Hver kasse udtrykker et begreb (navnet på begrebet står øverst i kassen), og for nogle af kasserne, har jeg også tilføjet såkaldte attributter (dvs. egenskaber for begrebet): f.eks. kan vi se at et musiknummer (Track) har en varighed (length) og en rytme (beats per minute). Pilene med de trekantede hoveder udtrykker, at et begreb kan være en specialisering af et andet begreb. Så i dette tilfælde er Track det generelle begreb, og Fast Track, Medium Track, og Slow Track er specialiseringer af dette begreb. Vi vender tilbage til dette senere i noten. Endelig kan vi også se at stregen mellem Playlist og Track har en lille hvid rhombe i den ene ende: dette udtrykker aggregering, dvs. det forhold, at en Playliste “består af” en række tracks.

I use-case diagrammet til højre, har jeg prøvet at ordne og gruppere den funktionalitet, som brugeren (tændstikmanden) ønskede i såkaldte use-cases (ovalerne). Hver use-case (også kaldet brugsmønstre) navngiver og beskriver en serie af trin i interaktionen ml. en bruger og et system. Så man skal altså forestille sig, at jeg også har lavet en detaljeret beskrivelse af alle ovalerne, f.eks. “Select playlist”, hvor den tilsvarende use-case i tekst, detaljeret vil beskrive hvad henholdsvis brugeren skal gøre, og systemet gør, når man vil vælge playlist.

Nu har du set 3 eksempler på modeller, som benyttes i forbindelse med systemudvikling. Lad os prøve at reflektere lidt over det vi har set:

- Modeller kan udtrykke egenskaber ved systemets **interne** dele: programmer og kode. Man kan lave en sådan model ud fra eksisterende kode for at synliggøre vise dele/aspekter (en analyse model), eller man kan lave en model, inden man begynder at kode (en design model).

- Modeller kan udtrykke egenskaber ved systemets **omgivelser**: problemområde og anvendelsesområde. Problemområdet er det som systemet skal kontrollere eller overvåge (f.eks udtrykt ved vores klassediagram). Anvendelsesområdet er det, der foregår mellem bruger og system (f.eks. udtrykt ved vores use-case diagram).
- Modeller er udtryk for **abstraktion**: når vi laver en model, lægger vi vægt på visse dele (f.eks. logikken i vores rutediagram) og ignorerer andre dele. Det betyder også, at en model altid udtrykker en bestemt **synsvinkel** på systemet eller på dets omgivelser.
- Nogle modeller udtrykker **dynamiske** egenskaber ved et system eller dets omgivelser: f.eks. rutediagrammet eller use-case diagrammet. Her er der således fokus på dynamik, processer, hændelser - noget der sker, som vi skal forstå og/eller holde styr på.
- Nogle modeller udtrykker **statiske** (strukturelle) egenskaber ved et system eller dets omgivelser: f.eks. klassediagrammet. Her er der fokus på begreber og strukturer, som ikke ændrer sig over tid, mens vi bruger systemet.
- Når vi abstraherer, vælger vi som sagt en bestemt synsvinkel at beskrive ud fra. Dette gælder også **størrelsesforholdet** (ofte kaldet modellens granularitet). Dette har vi ikke vist eksempler på, men forestil dig, at vi tager fat i klassen Playlist fra klassediagrammet. Denne kan naturligvis forstørres op og beskrives i yderligere detalje, hvor vi så typisk ser bort fra den kontekst (sammenhæng) som klassen indgår i.
- Sammenfattende: når vi laver modeller, så ofrer vi typisk en detaljerigdom (jvf. koden og interviewet med brugeren) til fordel for en præcision og et bestemt fokus. Vores modeller er fattigere på detaljer end virkeligheden (det vi forsøger at beskrive). Til gengæld bliver det nemmere at holde fokus på bestemte aspekter ved den virkelighed.

Modeller hjælper os med at håndtere **usikkerhed** og **kompleksitet** i vores arbejde med it. Modeller kan understøtte en dialog ml. personer involveret i udviklingen af it-baserede systemer: kunder, brugere, programmører, grafiske designere, interaktionsdesignere, databasedesignere, osv. Uden sådanne konkrete, håndfaste modeller kan det være svært at forstå hvad den anden part mener, og derfor udvikler vi modeller, der udtrykker bestemte egenskaber ved det system, der er under udvikling, såvel som ved de omgivelser et system skal bruges i. På den måde er der noget konkret at forholde sig til, hvilket gør en dialog om ændringer og tilføjelser meget lettere.

Vi kender brugen af modeller fra mange andre områder:

- Når vi har svært ved at finde rundt i den **fysiske** verden, så benytter vi os af kort. Det kan være en kortbog i bilen, et kort over undergrundsbanen i London, et kort over Københavns centrum, et kort over et storcenter, en plantegning over rummene i huset vi overvejer at købe osv. Vi er så vant til disse anvendelser, at vi ofte ikke tænke over at vi bruger dem. Vi glemmer at skelne mellem model og virkelighed, hvilket jo er helt fint, hvis modellen er god, og udtrykker de ting fra virkeligheden, som er vigtige for vores tanker her og nu i situationen. Omvendt så kender vi alle situationer, hvor modellen ikke svarer til virkeligheden (eller er det omvendt?). F.eks. hvis vi har vadet 2 km gennem et storcenter for at finde en bestemt butik, der så er flyttet, når vi kommer frem. Nogle gange er kortet (modellen) forældet. Nogle gange er kortet (modellen) for ny ("Butikken åbner om 14 dage. På gensyn!").
- Når vi skal forklare og visualisere **processer**, så anvender vi også modeller. Det kan være en grafisk fremstilling af en uddannelse med fagpakker, fag, studieretninger, toninger, år og semestre. Eller det kan være en samlevejledning til en reol fra IKEA. Vejledningen udtrykker en model af den proces, som vi skal igennem for at komme fra plader og skruer til et sted at opbevare tøj og bøger.

- I stort set alle **fag** i en uddannelse støder vi ind i modeller: modeller af atomer i fysik, molekyler i kemi, celler i biologi, eksponentielle funktioner i matematik, modeller for dramaturgisk udvikling af en historie i dansk, modeller af den menneskelige pskyke i psykologi, af samfundet i samfundsfagosv. Maslows behovspyramide i historie er også en model. Alle fag benytter sig af modeller, fordi modeller forsimpler virkeligheden og gør det lettere at håndtere den og forstå den.

Så på den ene side, så kender vi i forvejen en masse konkrete modeller og modeltyper. På den anden side, så er det for mange lidt uklart, hvornår vi har fat i en "rigtig" model eller måske blot en beskrivelse eller en forklaring. Da både modelbegrebet og kompetencerne involverede i at kunne bruge, ændre og skabe modeller, begge er så centrale for it-faget, udgør emnet et kernestofområde, som det er fordelagtigt, at vi behersker for at kunne arbejde kreativt og effektivt med it.

I denne note beskriver vi først hvad en model er for en størrelse. For at kunne være lidt mere præcis i vores forståelse, tager vi en tur omkring systemer & perspektiver, fænomener & begreber, og vi forklarer hvad henholdsvis et modelsystem og et referentsystem er, ligesom vi forklarer sammenhængen mellem disse to systemer. Dernæst tager vi et kig på sammenhængen mellem modeller, metoder og processer. Til sidst giver vi en række eksempler på, hvordan modeller og modelleringstankegangen er central for informationsteknologi som fag. I Appendix A har vi desuden opsummeret de forskellige kompetencer, som er vigtige, når man arbejder med modeller.

Hvad er en model?

Et karakterstika ved alle modeller er således, at de fremhæver og udtrykker visse dele af det, som de er en model af, imens de undertrykker og skjuler andre dele. På den måde er modellen en abstraktion over virkeligheden. Modellen reducerer **kompleksiteten** i vores forståelse af virkeligheden.

Et andet karakteristika ved anvendelsen af modeller, er at anvendelsen hjælper os med at reducere **usikkerhed**. Hvis vi f.eks. i en gruppe sidder med en fysisk plastikmodel af et molekyle, så kan vi se de forskellige bindinger og atomer, vi kan fjerne nogle og tilføje andre. På den måde bliver det klart for alle, hvad den enkelte mener, når vedkommende forklarer og fortæller. Vi kan diskutere om det er korrekt eller hensigtsmæssigt, i forhold til det vi ønsker at opnå eller beskrive, men muligheden for misforståelser er klart mindre, end hvis vi bare sidder og taler ud i den blå luft og fægter lidt med armene.

På samme måde anvender f.eks. arkitekter fysiske modeller i deres samarbejde med kunder og ingeniører. Ud fra en model af en ny bygning kan man diskutere om rummene og forbindelserne imellem disse er hensigtsmæssige og funktionelle, ligesom man kan diskutere æstetiske aspekter ved dele og helheden. Denne måde at reducere usikkerhed på, er klart billigere end at lave den virkelige bygning, rive ned og bygge op.

Systemer & perspektiver

Når vi arbejder med it, så taler vi ganske ofte om systemer. Men hvad er et system i denne sammenhæng?

*"Et **system** er en opfattelse af en mængde fænomener, der er meningsfyldt relaterede, så det er muligt og hensigtsmæssigt at betragte dem som et hele med særlige systemiske[2] egenskaber."* (Lindgreen, 1990)

Et system er altså udtryk for en bestemt opfattelse[3] eller man kunne sige et bestemt perspektiv:

*"Et **perspektiv** er det middel en person bruger til at strukturere vedkommendes tænkning og forståelse i relation til situationer i et område."* (BETA, 1993)

Eksempler:

- Hvis jeg som bruger betragter Facebook som et system, så kan jeg vælge at opfatte det som et socialt medie, hvor jeg kan kommunikere med mine venner, få nye venner, finde gamle venner, og dele tanker om nyheder, videoer, og musik. Hvis jeg er lidt mere gusten og beregnende, kan jeg se det som et medie til selvpromovering: ved omhyggeligt at udvælge, hvad jeg skriver på min væg, hvilke billeder jeg deler, osv. kan jeg skabe et billede af mig selv, som jeg ønsker at fremstå overfor andre. Hvis jeg er en virksomhed, der reklamerer, kan jeg betragte Facebook som et medie til målrettet at henvende mig til potentielle kunder, baseret på mine oplysninger om disse. Hvis jeg er producent af Facebook kan jeg betragte systemet som en platform, hvor det gælder om at tiltrække så mange brugere og så mange virksomheder som muligt, så de bliver bundet til netop min platform (i stedet for Twitter eller Google+ f.eks.)
- Hvis jeg arbejder som socialrådgiver i en kommune kan jeg vælge at se på de arbejdsløse borgere, som sager, der skal løses, indenfor det budget jeg har, og den lovgivning jeg er underlagt. En anden mulighed er, at betragte dem som klienter, der skal hjælpes. En tredje mulighed er, at betragte dem som kunder, jeg skal sælge kommunens aktiveringstilbud til. Ligegyldigt hvilket perspektiv jeg (mere eller mindre bevidst) vælger, så farver det mit syn på dem og mine arbejdsopgaver i forbindelse med dem - det påvirker min tænkning og min forståelse.

Perspektivet er altså vigtigt for ens forståelse af et system, og dermed også for ens adfærd i forbindelse med systemet. Når vi skaber it-baserede systemer er perspektivet derfor helt afgørende. Det kan i sidste ende afgøre om systemet bliver brugt eller ej.

Abstraktioner, fænomener, begreber

"Et system er en opfattelse af en mængde fænomener", skrev vi ovenfor. Hvis vi igen bruger Facebook som eksempel, så har systemet millioner af brugere. Hvis vi forsøger at forstå dem allesammen individuelt kommer vi ret hurtigt til kort. Derfor klassificerer vi alle de fænomener, der har ensartede egenskaber og adfærd til begreber. Vi taler om brugere, profiler, beskeder, tidslinjer. Ikke bare om Karoline som bruger, Michaels profil, Josephines tidslinje, eller den fatale besked

---

[2] "*En **systemisk** egenskab er en egenskab, man i sin opfattelse af en mængde fænomener som en helhed tillægger helheden i relation til resten af verden, og som hver af fænomenerne bidrager til uden nødvendigvis selv at besidde den"* (Lindgreen, 1990). F.eks. består min hund af hovede, ben, krop, hale, osv., men det at han kan gø og sige vuf er en systemisk egenskab ved (mange dele af) hele hunden, snarere end det er en isoleret egenskab ved hovedet. Det samme gælder i øvrigt hans egenskab med ikke at komme, når jeg kalder, som jeg ikke tror kan henføres til hverken ører eller ben. Muligvis hans diminutive hjerne.

[3] Bemærk at dette er udtryk for såkaldt "Soft Systems Thinking", modsat "Hard Systems Thinking", hvor ens tænkning om systemer er baseret på den antagelse, at systemerne har selvstændig eksistens i virkeligheden uafhængig af en evt. betragter (CoC, 1993). Groft sagt, så er Hard Systems Thinking den typiske naturvidenskabelige ingeniør-tilgang, hvor Soft Systems Thinking har rødder i samfundsvidenskab og konstruktivisme.

vedr. æblet fra Eva til Adam i går kl. 22.57. Det er en vældig handy cognitiv mekanisme, og vi bruger den hvert eneste minut hele dagen uden at tænke over det. Hvis vi udelukkende tænkte i konkrete forskellige fænomener, ville vi kun vide, at alting var forskelligt[4]. Ikke to personer er ens. Ikke to græsstrå er ens. I stedet abstraherer hjernen heldigvis automatisk fra de mindre væsentlige forskelligheder og kategoriserer alle de fænomener, der har væsentlige fælles egenskaber, så vi kan tale om personer og græsstrå, på en måde som er meningsfuld (endda på tværs af sprog og kulturer).

Lad os få defineret, hvad vi mener med henholdsvis et fænomen og et begreb:

"*Et **fænomen** er noget, der har afgrænset, selvstændig eksistens i virkeligheden eller i sindet."* (BETA, 1993)

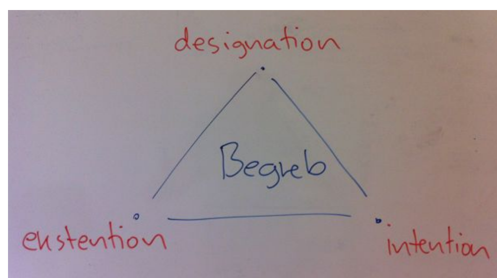
Et fænomen kan f.eks. være naboens bil, dvs. den røde, støjende, ildelugtende rustbunke, som holder i hans indkørsel. Men et fænomen kan også være en en begivenhed (juleaften 2010) eller en proces (studieturen til Norge) udstrakt over tid. Det er meget forskelligt om flere personer kan blive enige om, om de taler om og tænker på det samme fænomen. Med fysiske observerbare fænomener er det ofte let, idet man kan pege på genstanden eller referere til fælles erindringer om eller planer for det fysiske fænomen. Med fænomener i sindet er det ofte vanskeligere. "Harry Potter" er bestemt et fænomen i sindet, og takket været film og bøger (fysiske support-fænomener), kan vi nok ret hurtigt blive enige om, hvad dette fænomen er for noget. Men andre gange er det vanskeligere.

"*Et **begreb** er en generaliseret ide om en samling af fænomener, baseret på viden om fænomenernes fælles egenskaber."* (BETA, 1993)

Et begreb klassificerer således en række fænomener i kraft af deres ensartede egenskaber. F.eks. er vi ikke i tvivl om at min fine nye Mercedes er meget forskellig fra naboens gamle Lada. Men vi kan hurtigt blive enige om, at begge dele (begge fænomener) er eksempler på det generelle begreb "en bil". Egenskaberne er f.eks. at de har 4 hjul, et rat, en gearstang, benzindæksel, et antal sæder, og et antal døre, og at de (ikke mindst - hvilket ofte er rustbunkeejermændenes væsentlige pointe) bringer passageren "fra A til B". Bemærk også, at hvor et fænomen kan være fysisk til stede i virkeligheden eller i sindet, så hører begreber altid hjemme i sindet. Du kan aldrig køre en tur i begrebet "en bil", men altid kun i en bestemt bil.

Et begreb er karakteriseret ved 3 mængder:

- **Ekstension**: mængden af fænomener, som begrebet omfatter og beskriver.
- **Intention**: mængden af alle de egenskaber, der kan bruges til at karakterisere alle de fænomener, der er i et begrebs ekstension.
- **Designation**: mængden af alle de navne (eller symboler og tegn), som begrebet er kendt under.

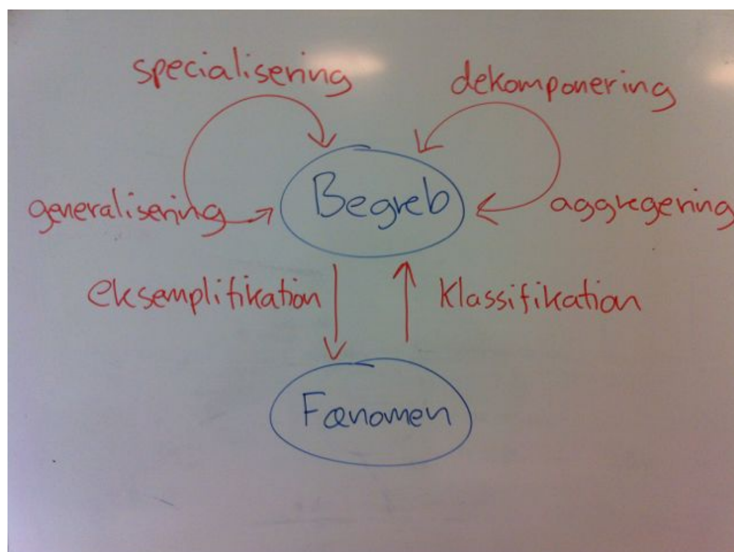


Dette er illustreret i figuren til højre. Et eksempel kunne være begrebet "en hund". Ekstensionen omfatter de konkrete hunde "Fido", "Rolf", "Bimmer" osv. Intentionen omfatter: "kødædende",

[4] Grady Booch: "Without abstraction we only know, that everything is different". Onde tunger påstår så ind i mellem at "With abstraction everything looks the same".

“kan sige vov”, “har 4 ben”, “har en hale”, “kan ikke lide katte” osv. Designationen omfatter “en hund”, “en køter”, “et kræ”, “en vovse”, “Canidae” m.fl.

Når vi tænker på og arbejder med fænomener og begreber, så bruger vi en række begrebsdannelsesprocesser, som jeg har illustreret i figur 3:



Figur 3: Begrebsdannelsesprocesser

- **Eksemplifikation:** Givet et begreb (“en filosof”), så kan vi eksemplificere med en række fænomener (“Kierkegaard”, “Sartre”, “Camus”, “Anders Fogh”).
- **Klassifikation**: Givet en række fænomener (“Daniel Agger”, “Lionel Messi”, “Christiano Ronaldo”, “Sofus Krølben”), så kan vi klassificere dem som værende eksempler på samme begreb (“en fodboldspiller”).
- **Generalisering**: Givet en række begreber (“hund”, “gris”, “ko”) kan vi generalisere disse til et nyt mere generelt begreb (“pattedyr”). Bemærk at alle egenskaberne for det generelle begreb (“pattedyr”) er gældende for alle de specialiserede begreber, men ikke omvendt. F.eks. føder alle pattedyr levende unger, hvilket derfor også gælder for hunde, grise, og køer, men en ko har horn, hvilket ikke gælder alle pattedyr.
- **Specialisering**: Givet et generelt begreb (“en gnaver”) kan vi aflede en række specialiserede begreber (“kanin”, “mus”, “rotte”). Dette er blot den omvendte proces af generalisering.
- **Aggregering**: Givet en række “del-begreber”, kan vi samle disse i et “helhedsbegreb”. F.eks. kan vi samle begreberne “rod”, “stængel”, “blade”, og “blomst” i begrebet “plante”.
- **Dekomponering**: Givet et sammensat begreb, kan vi dekomponere dette i bestanddele. F.eks. kan vi opdele “et fodboldhold” i “forsvar”, “midtbane” og “angreb”. Dette er blot den modsatte proces af aggregering.

Alt dette kan måske virke vældig teoretisk, men når vi arbejder med it, så skal vi huske, at maskinen i udgangspunktet er meget dum. Den forstår ikke vores sprog, og derfor bliver vi nødt til at være meget præcise, når vi programmerer den og udvikler systemer til den. Som en bonus / sidegevinst, så bliver det også nemmere at tale præcist med sine samarbejdspartnere.

Bemærk derfor præcisionen som ovenstående begrebsapparat medfører. I stedet for at vi kan tale om, “at nogle ting har med nogle andre ting at gøre”, kan vi pludseligt blive meget skarpe. Vi kan tale om, at et begreb er en generalisering af 3 andre begreber, og at det samme begreb kan opsplittes

i 5 del-begreber. Vi kan forstå, at "en angriber" ***ikke*** er eksempel på "en fodboldspiller", men at "en angriber" er en specialiering af "en fodboldspiller". Vi kan skelne skarpt mellem konkrete fænomener og abstrakte begreber. Uanset hvilken notation og type af modellering, du anvender, så er det en god generel kompetence at have. Det er afgørende betydning om man vælger den ene eller anden forståelse af fænomener og begreber, når modellen skal repræsenteres i et system.

Øvelse: Tag en model i en notationsform du kender (f.eks. et java-program, et rutediagram, et E/R-diagram, et tilstandsdiagram). Beskriver modellen fænomener eller begreber? Begge dele? Hvis modellen beskriver begreber, så prøv at give eksempler på designation, ekstension, intention. Prøv tilsvarende at finde eksempler på de 6 begrebsdannelsesprocesser. Hvis du ikke kan finde eksempler på dem alle, så prøv at overvej, hvad de kunne være.

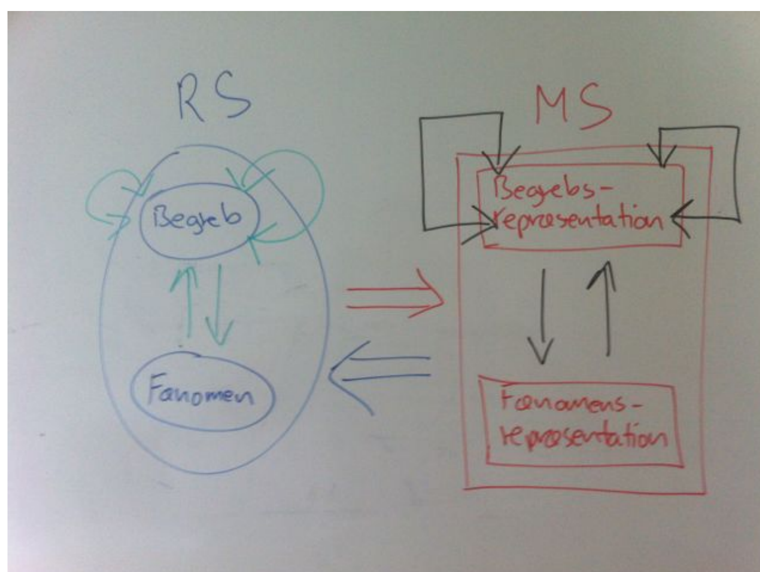
Modelsystem & referentsystem

I forbindelse med modellering skelner vi mellem to typer af systemer: modelsystem og referentsystem. Referentsystemet er det som vi ønsker at lave en model af, og modelsystemet er den model, vi så rent faktisk får lavet. To eksempler:

- På mit 5-stjernede hotel i London får jeg udleveret et kort over undergrundsbanen. Kortet er modelsystemet. Londons undergrundsbane er referentsystemet. Jeg bruger modelsystemet til at forstå referentsystemet. F.eks. til at orientere mig om, hvor jeg er henne pt. eller måske planlægge en rejse i morgen. Modelsystemet er i dette tilfælde ikke det samme som et almindeligt landkort, idet det er tegnet på en måde, så stationerne og linjerne er tydeligt markerede og med pæn indbyrdes afstand, og derfor ikke nødvendigvis ligger geografisk korrekt i forhold til hinanden. Modellen fremhæver altså visse aspekter (stationer og linjer) og skjuler andre (geografisk placering, hundetoiletter, Apple butikker osv.).
- Herning kommune har besluttet at lave nye busruter. Kommunens topplanlægger har fået til opgave at lave en skitse over mulige fremtidige busruter. Skitsen er modelsystemet, og referentsystemet er i dette tilfælde planlæggerens forestilling (vision) om fremtidige busruter i kommunen. Bemærk altså, at i dette tilfælde findes referentsystemet endnu ikke i virkeligheden, men at vi bruger et modelsystem til at diskutere en mulig fremtidig virkelighed.

Ovenstående er faktisk karakteristisk for mange anvendelser af modeller og modellering: hvis man skaber en model for at forstå noget eksisterende, så taler man ofte om **analysemodeller**; hvis man skaber en model for at illustrere muligheden af noget nyt, så taler man ofte om **designmodeller**. I ovenstående eksempler er London-kortet således en analysemodel, og Herning-kortet en designmodel.

Lad os lige få koblet det vi har lært om fænomener og begreber, med det vi har lært om referentsystem og modelsystem. Kig på figur 4. Til venstre har jeg vist et referentsystem (RS) med begreber og fænomener, som vi er interesserede i. De grønne pile symboliserer de begrebsdannelsesprocesser, vi beskrev tidligere (aggregering, specialisering, osv.). Til højre har jeg vist et modelsystem (MS). Det interessante er, hvad dette indeholder. Det indeholder nemlig en repræsentation af begreberne fra RS (øverste kasse), en repræsentation af fænomenerne fra RS (nederste kasse), samt en repræsentation af begrebsdannelsesprocesserne fra RS (de sorte pile).



Figur 4: Modellering: Referentsystem og Modelsystem

Så hvis vi tager vores eksempel med Londons undergrundsbane, så har vi stationerne og linjerne som elementer til venstre i RS, men selve symbolerne på kortet er til højre i MS: symbolforklaringen på kortet (altså forklaringen af hvordan en linje og en station ser ud) er vores begrebsrepræsentation (for oven), og selve det fysiske layout af linjer og stationer vores fænomensrepræsentation (for neden).

Kig nu igen på figur 2 med UML diagrammet. Dette er nemlig et andet konkret eksempel på ovenstående generelle model. Begreberne (Track, Playlist, osv. fra vores referentsystem) er repræsenterede som klasser (kasser) i modellen (vores modelsystem). Begrebsdannelsesprocesserne specialisering og agggregring fra RS er repræsenterede som henholdsvis pilene og stregen med rhomben i MS.

Øvelse: Kig nu på figur 1 med javakoden og rutediagrammet. Kan du se, at rutediagrammet er en model af javakoden? Hvilke begreber og hvilke fænomener beskriver det?

Sammenfattende er det vigtigt at forstå, at når vi laver modeller, så anvender vi altid et bestemt perspektiv eller en synsvinkel. Dette perspektiv definerer hvilket referentsystem, vi vælger at beskæftige os med - hvad vi vælger at se og opfatte, når vi kigger på virkeligheden. Altså inden vi overhovedet er begyndt på vores model - når bare vi kigger - så har vi allerede foretaget nogle meget afgørende valg. Det er også vigtigt at forstå, at et referentsystem kan bestå af fænomener i et systems omgivelser (londons undergrund) eller interne dele (javakode). Desuden kan vi vælge at lave vores modelsystem i forskellige notationer, f.eks. UML, naturligt sprog, uformelle tegninger, eller javakode. Dette forhold - at et program kan være både et referentsystem, som vi skal lave en model af, eller i sig selv være et modelsystem for noget andet - kan være temmelig forvirrende.

I øvrigt skal det bemærkes at, hvis man anvender objekt-orienterede programmeringssprog (som f.eks. Java), så stiller sproget primitiver og mekanismer direkte til rådighed til at udtrykke fænomener, begreber, og specialisering, og aggregering. De lærde kan så slås om, hvilke begreber og fænomener, man bør udtrykke i sådanne programmeringssprog: om det skal være fra systemets omgivelser, fra programmørens hjerne (f.eks. i form af design mønstre), eller fra den platform man arbejder på (f.eks. vinduer, knapper, osv.).

Metoder
Når vi arbejder med metoder til systemudvikling, så skelner man ofte mellem "low-ceremony" og "high-ceremony" metoder. I det følgende kigger vi lidt på anvendelsen af modeller og modellering i forbindelse med begge.

High-ceremony metoder anvender mange formelle møder, reviewprocesser, tests, og modeller. Udviklerne fordeler ofte et antal formelt definerede roller i mellem sig (arkitekt, use-case analytiker, databaseansvarlig, GUI-udvikler, tester, m.fl.) - den enkelte udvikler har ofte flere roller. Til modellerne er der som oftest knyttet mere eller mindre præcis semantik (betydning, dvs. hvad betyder en konkret model) og syntax (regler for form, dvs. hvordan må en model se ud, hvis den skal være "lovlig"). Klassiske (typisk lidt ældre) metoder er ofte high-ceremony. High-ceremony metoder har et naturvidenskabeligt udgangspunkt, hvor man vægter præcision, eksplicithed, forudsigelighed, osv. Hvis systemudviklingsprocessen fejler har man en tilbøjelighed til at skyde skylden på udviklerne, snarere end på metoden eller situationen.

Specifikt i forhold til brugen af modeller, så forbindes High-ceremony metoder ofte med omfattende brug af dokumentation, specielt modeller som udarbejdes og vedligeholdes i såkaldte CASE-værktøjer (Computer Aided Software Engineering). Fortalerne for sådanne værktøjer nævnes ofte som to af de vigtigste fordele, at værktøjerne understøtter kodegenerering og reinforcement of syntactical rules. Modstanderne finder værktøjer unødig komplicerede og begrænsende, forstået på den måde, at udover selve modellens kompleksitet (som jo har med det problem, vi forsøger at udtrykke og/eller løse at gøre), så indfører værktøjerne et ekstra lag af kompleksitet, fordi man nu også skal sikre sig, at modellen er syntaktisk korrekt, for at man overhovedet kan udtrykke modellen i værktøjet.

Low-ceremony metoder vægter i kontrast til det ovenstående konkret kode og direkte menneskelig kommunikation højere end udformningen og udveksling af modeller og dokumenter. Som et respons på den stigende brug af mere og mere omfattende metoder i ’80-erne og ’90-erne, opstod der i slutningen af ’90-erne et modsvar fra de systemudviklere, der begyndte at opfatte metoderne, modellerne, notationerne, og værktøjerne, som spændetrøjer. F.eks. formulerede Kent Beck en række principper for det han kaldte “eXtreme Programming” (XP). XP er karakteriseret ved, at man fokuserer på traditionelle håndværksmæssige aspekter af programmering.

- F.eks. ved man erfaringsmæssigt, at det er godt, at programmere to personer sammen ved én maskine, idet to par øjne ser bedre end ét, og to hjerner tænker bedre end én. Den ene person kan taste koden ind, mens den anden hele tiden kigger med og kommer med forslag til forbedringer og ændringer. På den måde bliver ideerne løbende testet af mod en anden persons viden og erfaring i stedet for at ende ukritisk i koden. En anden fordel er, at begge personer således er inde i forståelsen af koden, og derfor er projektet meget mindre sårbart, hvis den ene programmør har barnets første sygedag, eller hvis den anden programmør skifter job til et andet projekt eller en anden virksomhed. Dette kaldes “pair-programming”.
- Et andet eksempel er “Customer On-Site”-princippet, hvor man under hele udviklingsprocessen har en kommende bruger af systemet til rådighed i projektet. Vedkommende er med til at træffe beslutninger, når udviklerne er i tvivl om noget skal laves på den ene eller anden måde. Alternativt til dette, så anvender high-ceremony metoderne ofte et omfattende batteri af kravspecifikationer - dvs. lange og detaljerede (ofte formelle) beskrivelser af brugernes krav til systemet - der udarbejdes i starten af projektet, hvorefter det kastes over muren til en horde af

vilde programmører, der så kan misforstå specifikationerne og komme til at lave det forkerte system.

I begge tilfælde anvender man en praksis, som er velkendt for programmører - f.eks. at hjælpe hinanden med at programmere og at tale med brugerne - men man gør det konsekvent og disciplineret i hele projektforløbet (heraf ordet "ekstrem").

Et nyere eksempel på en low-ceremony metode er SCRUM, som anvendes meget i industrien i dag. i SCRUM benytter man bl.a. daglige ultra-korte morgenmøder, hvor alle udviklerne i en lille gruppe samles, og hvor hver udvikler kort skal svare på spørgsmålene: "Hvad fik jeg lavet i går?", "Hvad vil jeg lave i dag?", "Hvor tror jeg, at jeg kan løbe ind i udfordringer, og kan få brug for hjælp?". På den måde er alle kort orienterede om hinanden arbejde, man får en fornemmelse af den fælles fremdrift, og hvis nogle gør opmærksom på en udfordring, så kan andre byde ind og tilbyde sin hjælp til at få løst problemet. Møderne foretages stående - gerne ved en tavle med post-it notes, der repræsenterer opgaverne, og som kan placeres i en struktur på tavlen - for at forebygge at møderne trækker i langdrag. Et sådant SCRUM-møde tager typisk 15-20 minutter for en lille gruppe af 4-5 udviklere[5].

XP, SCRUM og andre low-ceremony metoder kaldes ofte for Agile (adrætte) metoder, og nogle af fortalerne har sågar begået et manifest, der udtrykker principperne for de rettroende: http://agilemanifesto.org .

Betyder det så, at man ikke laver modeller, hvis man anvender agile (low ceremony) metoder? Nej, men rollen af modellerne er anderledes. Man laver typisk modellerne med tusch på en tavle, som er i det rum, udviklerne sidder i. På den måde kan alle hele tiden se disse, og man opfordres til at bidrage - hvilket ofte ikke er tilfældet, hvis en kollega sidder med hovedet begravet i et UML-værktøj på sin egen pc. Desuden går man ikke ret meget op i formel syntax og semantik. Hvis ens samtalepartner kan forstå den model, man har lavet, så er den pr. definition god nok - selv om den ikke anvender korrekt UML notation. Endelig så har modellerne ofte kortere levetid. I forbindelse med high-ceremony metoder gemmes alle modeller (og også gerne tidligere versioner af samme model) med henblik på at kunne dokumentere systemudviklingsprocessen. Med agile metoder smider man ofte modellen væk, når den har tjent sit umiddelbare formål, dvs. når forståelsen er på plads og tavlen er ryddet. En helt almindelig praksis, hvis man alligevel ønsker at gemme (eller måske distribuere) sådanne tavle-modeller, er at fotografere og arkivere dem.

Som et eksempel på udvikling og brug af modeller i forbindelse med high-ceremony metoder, kan du kigge på Figur 2 igen. Her har jeg brugt et gratis UML-værktøj (ArgoUML), til at lave modellerne. Som et eksempel på low-ceremony anvendelse af modeller, kan du kigge på de mange digitale fotos i noten, hvor jeg hurtigt har skitseret nogle modeller på et whiteboard og taget et foto med min mobil, for at illustrere en pointe.

Som nævnt så opstod den agile bevægelse som et modsvar fra programmørerne imod den stigende formalisering af udviklingsmetoderne (processen og dokumentationen). Kritikken lød ofte, at metodekonsulenterne havde fået alt for meget magt i virksomhedernes systemudvikling, og at det flyttede fokus fra at lave gode systemer til at følge en bestemt metode til punkt og prikke. Det skal

---

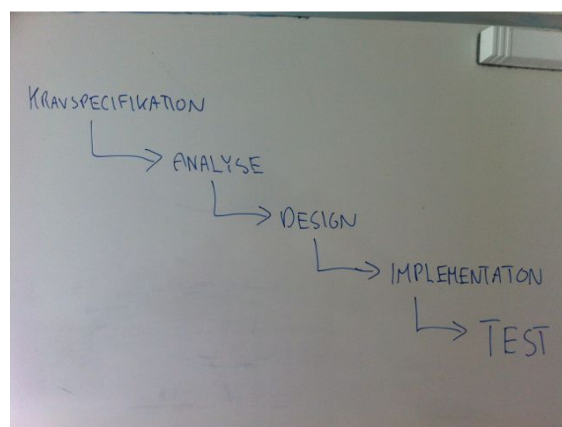
[5] "Scrum" er efter sigende navngivet efter det lille taktik-møde, som rugby-spillere afholder inden de afvikler et set-play i en rugby-kamp.

også indskydes, at metodekonsulenterne ofte var meget mere akademiske og teoretiske end de praktisk anlagte programmører (mange metoder er opstået i forskningsmiljøer og på universiteter). For de dramatisk og religøst anlagte folk, blev Agile metoder derfor opfattet som lidt af en frihedskamp, som gik ud på at give systemudviklerne og programmørerne magten tilbage igen. Det pudsige er så, at historien bider sig selv lidt i halen. I dag bruger man i mange virksomheder uforholdsmæssigt langt tid på at diskutere om SCRUM skal forstås på den ene eller den anden måde. Må man f.eks. godt holde møder i længere end 15 minutter? Kan man godt programmere alene engang i mellem, uden at universet kollapser af den grund? Man har også indført certificering af SCRUM-udøvere, så nu kan du komme på kursus og få sort bælte i SCRUM.

Man kan grine lidt af, at metodekonsulenterne (også kaldet "metodisterne") fra tidligere, nu er erstattet af certificerede SCRUM-masters (det hedder det virkeligt!), og man kan komme til at tænke på både farisæere, ypperstepræster, Yoda og zen-mestre i den anledning. Men det er nok en vigtig pointe, at hvis man vælger en low-ceremony metode frem for en high-ceremony metode, så har man ikke så mange regler, notationer, roller, opskrifter og håndbøger at støtte sig til, og dermed bliver den enkeltes kompetencer meget vigtigere. Både de snævert faglige kompetencer, men også samarbejdskompetencerne. Derfor kan der være en vis mening i, at man skal træne og certificeres for opnå færdigheder og disciplin, inden man indgår i et samarbejde om at konstruere komplicerede software systemer. Groft sagt, så kan man sige, at high-ceremony metoder er designede ud fra hypotesen, om at du og dine kolleger ikke er alt for kvikke, og derfor vil begå fejl som skal fanges og udbedres, og til dette formål anvendes omfattende dokumentation og tilhørende reviewprocesser. Low-ceremony metoder antager, at både du, dine kolleger, kunden og brugerne er særdeles kompetente og disciplinerede, og at I selv løbende fanger fejlene. Begge antagelser kan være rigtige eller forkerte.

Procesmodeller: vandfald og spiraler

I det foregående afsnit diskuterede vi forskellige typer af metoder med specielt fokus på deres anvendelse af modeller. Men som nævnt indledningsvis, så kan vi også lave modeller af processer, og hvorfor ikke kigge på to forskelllige modeller af vores egen udviklingsproces, når vi udvikler software. I det følgende kigger vi derfor på henholdsvis vandfalds- og spiralmodellen.



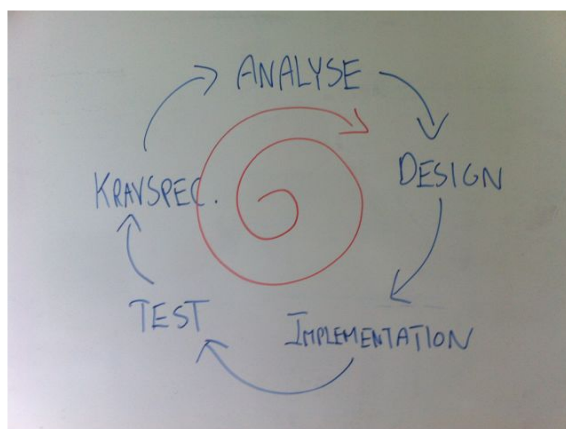
Figur 5: Vandfaldsmodellen

I vandfaldsmodellen (figur 5) opdeler man udviklingsprocessen i et antal faser. Det centrale er, at man færdiggør hver fase, inden man påbegynder den næste. Dvs. først indsamles og beskrives alle kravene til systemet, som sammenfattes i et kravspecifikationsdokument, der reviewes og godkendes; dernæst laves en omfattende analyse af disse krav og resultatet sammenfattes i et

analysedokument, der reviewes og godkendes; herefter designer man en løsning, der sammenfattes i et designdokument, der reviewes og godkendes; så følger selve programmeringen (implementationen), og systemet/koden dokumenteres og reviewes, og endelig testes systemet på forskellig vis, og når det opfylder kravspecifikationen, så er systemet færdigt og tages i brug.

Øvelse: Hvad er fordelen ved vandfaldsmodellen? Hvad er ulemperne?

I spiral modellen (figur 6) opdeler man i stedet processen i et antal aktiviteter: kravspecifikation, analyse, design, implementation, og test[6]. Disse gennemløbes i rækkefølge i et antal **iterationer** indtil systemet vurderes til at være færdigt. I stedet for at starte med den totale kravspecifikation, tager man fat i et enkelt krav til at begynde med. Dette krav analyseres, designes, implementeres og testes. Så sammenholder man det med kravspecifikationen, og hvis testen viser, at den programstump man har fået lavet lever op til kravene, så går man videre med det næste krav. På denne måde vokser systemet **inkrementielt** - stykke for stykke.



Figur 6: Spiralmodellen

Alternativt - hvis programstumpen ikke lever op til kravene - tager man en tur mere rundt i cirklen: måske var kravet ikke præcist nok formuleret? Måske var analysen ikke grundig nok? Måske valgte vi et forkert design? Måske havde vi blot lavet en simpel programmeringsfejl? Måske var koden faktisk ok, men vi testede forkert?

Denne type udvikling kaldes også for en iterativ (at man gentager aktiviteterne) og inkrementiel (at man opbygger systemets funktionalitet stykvist) proces.

Øvelse: Hvad er fordelen ved spiralmodellen? Hvad er ulemperne?

Der er mange måder at gribe en iterativ og inkrementiel udviklingsproces an på. F.eks. Top-Down, hvor man starter med at lave en abstrakt og generel skelet-applikation (dvs. grundliggende en arkitektur), hvor strukturen er på plads, men alle komponenter er tomme og uden funktionalitet. Dette skelet kan man så gradvist forfine og konkretisere ved at hælde mere kode på. Dette princip kender du måske fra stepwise refinement i programmering.

---

[6] Det skal understreges af faserne fra vandfaldsmodellen og aktiviteterne fra spiralmodellen kunne hedde meget andet - afhængigt af den konkrete metode, der anvendes, og desuden kan der forekomme flere end blot 5.

Alternativt kan man gribe det an som en Bottom-Up proces, hvor man starter med at lave små byggeklodser (komponenter), så man så trinvist sætter sammen i stadig større blokke, der bliver til komponenter på næste niveau, osv.

Den strategi, man vælger, er afhængig af mange faktorer:
- Hvis man f.eks. skal lave et system med en arkitektur, man har prøvet at bygge mange gange før, men hvor f.eks. den teknologi, der er valgt til grænsefladen eller databasen er helt ny, kan man med fordele forsøge at reducere usikkerheden for hele projektet, ved at starte med at den komponent. Arkitekturen ved man jo, at man behersker.
- Hvis man omvendt skal lave et system, hvor man ikke har en god ide om arkitekturen, så skal man måske starte med et lave et antal prototyper på forskellige arkitekturer.
- Hertil kommer alle de lavpraktiske faktorer, som gør sig gældende i hverdagen: hvornår får vi databasen leveret? Hvornår er vores grafiske designere ledige? Har vi komponenter fra andre projekter, vi kan genbruge? Har vi et hold af meget erfarne udviklere eller består holdet af nybegyndere, der skal læres op, mens vi udvikler?

Øvelse: Prøv at tænke på sidste gang du skulle konstruere, bygge eller skabe noget. Det kan f.eks. være et kaninbur, et hus i Sim City, lægge make-up, skrive en dansk stil, eller lave en projektrapport fra et gruppearbejde. Arbejdede du/I efter en vandfaldsmodel? En spiralmodel? Arbejdede du/I Top-Down? Bottom-Up? Var det et fornuftigt valg? Hvorfor (ikke)?

Eksempler

I det følgende karakteriserer og eksemplificerer vi anvendelsen af modellering i centrale dele af it-faget: programmering, repræsentation, samt arkitektur.

Programmering

Når man programmerer, laver man en model (kaldet kildekoden, source code) af en programudførsel, som man giver til et andet (specielt) program, oversætteren (compileren eller fortolkeren). Oversætteren oversætter så kildekoden fra programmeringssprog (som programmører kan læse og skrive) til maskinsprog (som en computer kan læse og skrive). Herefter kan programmet (nu udtrykt i maskinsprog) udføres direkte på computeren. I praksis kan denne proces bestå af flere forskellige trin, hvor repræsentationen af et program gradvist og trinvist oversættes til noget, der er mere maskinnært. Selve det at programmere er derfor en form for modellering.

Repræsentation og digitalisering

Når vi arbejder med it, så arbejder vi også ganske ofte med en lidt anden betydning af ordet “modeller” end det som vi fokuserer på i denne note - nemlig de tilfælde hvor vores informationssystem - vores - software - indeholder en model af et problemområde.

Typisk så indeholder et system en model af det område, som systemet skal bruges til at kontrollere, overvåge, eller styre. Dette kaldes ofte problemområdet.

- F.eks. indeholder et banksystem en (mere eller mindre eksplicit) model af kunder, konti, renter, indsættelser, udbetalinger, saldi, osv.
- Et system til simulering og træning af hjerte- eller ørekirurgi indeholder en model af væv, knogler, blodbaner, osv.
- En social medie applikation såsom twitter eller Facebook indeholder en model af profiler, chatbeskeder, wallposts, reklamer, osv.

- Et sportsspil som FIFA indeholder komplicerede modeller af fysiske legemer og bevægelser, såsom spillere (med arme og ben), bold og bane, såvel som modeller af spillernes samarbejde, taktikker, og strategier for de enkelte hold og dele af holdene.

Denne type af modeller **repræsenterer** fænomener og begreber fra den virkelige eller imaginære verden. Repræsentationen (modellen) kan være lavet på baggrund af en **digitalisering** af virkelige fænomener fra den virkelige verden, eller den kan være baseret på forestillinger om, hvordan vi tror - i følge videnskaben og de pt. gængse teorier - fænomenerne er opbyggede og indbyrdes relaterede.

Arkitektur

Arkitektur handler om strukturen af it-systemer, og der findes mange forskellige varianter og forståelser af arkitekturbegrebet: man kan tale om applikations-arkitektur, som er strukturen af individuelle systemer, om informations-arkitektur, der handler om strukturen af data (repræsentationer), om system-arkitektur, der handler om strukturen ml. en familie af sammenhængende systemer, osv. Fælles for dem alle er, at når man anvender et arkitektursyn på it, så fokuserer man på strukturen og interaktionen mellem delene (og ikke på selve komponenterne, modulerne, programmerne). Grunden til at dette er vigtigt, er at man erfaringsmæssigt har erkendt, at når et system bliver sat i drift, så kommer der uværgeligt rettelser og tilføjelser til systemerne. Hvis systemet har en uhensigtsmæssigt struktur, så bliver det ofte vanskeligt at vedligeholde disse systemer. Nogle gange så vanskeligt, at man må skrotte et system, og lave et helt nyt, hvilket er dyrt. Omvendt, hvis systemet har en god arkitektur - typisk beskrevet som løs kobling mellem delene - så er det nemmere at vedligeholde systemet.

Naturligvis har UML et omfattende beskrivelsesapparat til at modellere arkitektur - faktisk fra en hel række (typisk 4 eller 5) forskellige synsvinkler (ofte kaldet views). I denne note vil vi dog gå lidt mere praktisk og pædagogisk til værks, idet vi vil beskæftige os med en helt generel model for arkitektur, som kan være vældig nyttig, når man skal forstå, bruge, ændre, eller skabe systemer. Modellen er således ikke specifik for et bestemt konkret system, men er generel anvendelig som mental model (forståelse) af alle systemer. Modellen opdeler et system i 3 dele: grænseflade, funktionalitet, repræsentation[7], i det følgende benævnt G, F, og R.

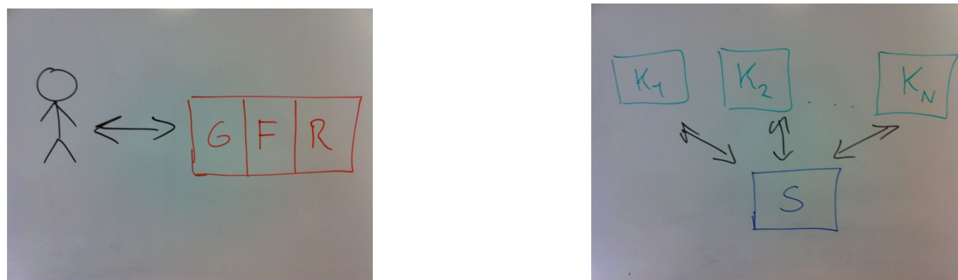
- **Grænsefladen** er den del af systemet (af koden), som brugeren (eller andre eksterne systemer) kommunikerer med. Tænk på knapper, vinduer, scroll-barer, indtastningsfelter - alt det som du kan navigere rundt i med mus og tastatur på en pc eller med fingrene på en smartphone.
- **Repræsentationen** er den del af systemet (koden), som indeholder repræsentationerne af det som systemet handler om - dataene. Hvis det er en musik-app vi kigger på, så drejer det sig om selve musiknumrene, som de er gemt i systemet, og hvad der ellers er gemt af yderligere information i forbindelse med disse musiknumre: coverbillede, titel, albumtitel, kunstner, længde, kvalitet, genre, antal afspilninger, osv.
- **Funktionaliteten** er den del af systemet (koden), som typisk indeholder algoritmerne for det som vi ønsker at gøre ved vores data. Hvis det f.eks. er et program til at lave ringetoner med, vi kigger på, så er der typisk funktioner til at udvælge, redigere og gemme lyden: vælge en del ud, ændre lydstyrken, lave ekko, ændre pitch, sætte delen ind på et andet tidspunkt i nummeret, lave loops, osv.

Øvelse: Hvorfor tror du disse 3 aspekter typisk er adskilt fra hinanden i et system?

---

[7] Repræsentationen kaldes ofte for “modellen”, idet den er en model af systemets problemområde, men det forvirrer begreberne i forhold til det, vi indtil videre har sagt om modeller og modellering.

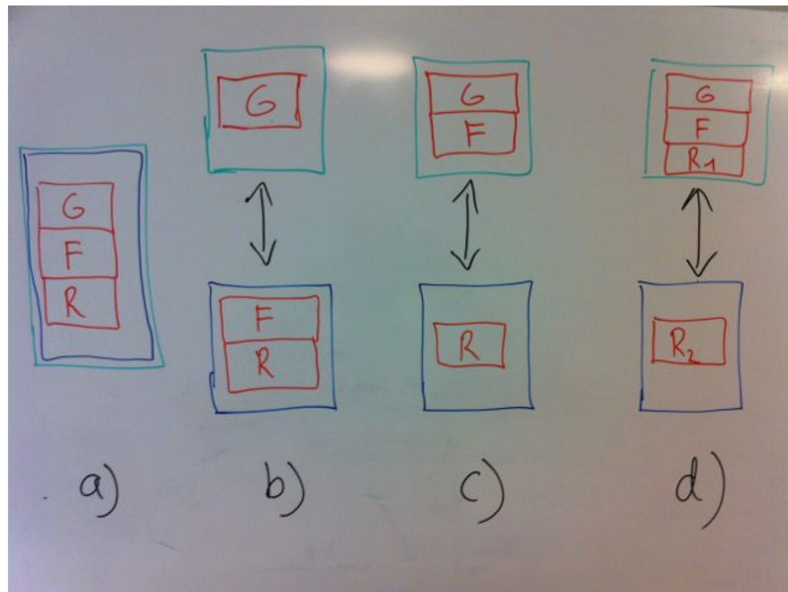
Ovenfor har vi ikke taget stilling til, hvor systemdelen (koden) rent fysisk befinder sig: om det hele er placeret på samme computer eller fordelt på flere forskellige maskiner. Der er nemlig flere muligheder. Betragt figur 7 nedenfor.



Figur 7: Logisk versus fysisk opdeling af systemdele.

Figuren viser to klassiske mønstre i arkitektur. Til venstre har vi ideen om **lagdeling**, hvor vi har opdelt et system i grænseflade, funktionalitet, og repræsentation, som forklaret tidligere. Til højre har vi ideen om **klienter og servere**: pointen i dette mønster er, at en række forskellige brugere af systemet kan benytte hver deres klient til at forbinde sig til og kommunikere med en fælles server. Dels betyder det, at flere brugere kan bruge deres klienter samtidigt (uden at skulle slås om tastatur og skærm), dels betyder det, at brugerne og klienterne kan være geografisk distribuerede.

Forskellige kombinationer af disse to simple mønstre giver en god forståelse for mange af de systemer, vi bruger i det daglige. Betragt Figur 8. De grønne kasser er klienter, de blå er servere, og de røde er vores forskellige logiske systemdele (G, F, R).



Figur 8: Varianter af distribuerede systemer.

I a-varianten har vi alle tre systemdele kørende på samme computer. Et eksempel er applikationerne fra officepakken, hvor vi kan bruge henholdsvis Word, Powerpoint, og Excel isoleret på vores egen maskine uden at være på nettet.
I b-varianten har vi vores grænsefladekomponent placeret på en klient, hvorimod funktionaliteten og repræsentationen er placerede på en server. Dette er typisk for mange web-baserede systemer, at

vi har vores browser kørende på vores egen computer, men at vi så kan koble op på en webside, der tilbyder noget funktionalitet og noget data vi kan arbejde på. F.eks. Facebook.
I c-varianten har vi både grænsefladen og funktionaliteten placeret på vores egen klient. Hvis du har prøvet at arbejde med et "fællesdrev", som bruges til at dele dokumenter og filer på en skole eller en arbejdsplads, så kender du også denne variant. Dette gælder f.eks. hvis du bruger Word til at rette i et dokument som ligger på et fællesdrev. Eller hvis du har adgang til en database, som du kan bruge eller opdatere. Hvis du har prøvet dette, så kender du også fornemmelsen, når netværksforbindelsen mistes - enten pga. en fejl, eller fordi du sidder i sommerhuset uden internet og skulle have lagt sidste hånd på en opgave, der skulle afleveres i morgen.
Endelig er der d-varianten, som skal forstås på den måde, at repræsentationen (vores data) er fordelt på både klient og server. Dette kender du sikkert fra de mange såkaldte cloud-baserede services, som er dukket op de senere år: Google Drive, Microsoft Skydrive, Apple's iCloud, eller DropBox. Her er idéen, at man kan arbejde videre på sine data, selv om man ikke har netforbindelse. Klient-computeren (f.eks. din bærbare) opretholder nemlig en lokal kopi af dine data fra serveren, som du altid kan arbejde på. Når computeren så igen får netforbindelse, så opdaterer den automatisk både din lokale kopi (hvis der er ændringer på serveren, f.eks. fra andre brugere af DropBox, som har uploadet nye filer til jeres delte dropbox) og serveren (hvis du har ændret i eller oprettet nye dokumenter i din lokale kopi af jeres dropbox). Dropbox synkroniserer således klienter og serveren.

Øvelse:
- *Vælg en håndfuld applikationseksempler, som du kender fra pc/mac, ipads, og smartphones. Overvej for hver enkelt applikation, hvilken af varianterne a-d du tror, der bliver anvendt.*
- *For hver variant a-d, overvej fordele og ulemper ved denne arkitektur.*
- *Overvej om der er flere varianter end de nævnte a-d.*

Referencer

(BETA, 1993) Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
(CiC, 1993) Dahlbom, Bo and Mathiassen, Lars: *Computers in Context*. Blackwell Publishing, 1993.
(Lindgreen, 1990) Lindgreen, P.: *Systemanalyse*. Jurist- og Økononomforbundets Forlag, 1990.

Appendix A: Modelleringskompetencer

Vi skelner mellem tre forskellige former for kompetence i arbejdet med modeller:

- A**nvendelseskompetence**. Man kan forstå og anvende en model.
    - *Eksempelvis at man kan læse og forstå dele af et UML klassediagram.*
- **Ændringskompetence**. Man kan forstå, anvende og ændre/modificere en model under vejledning og instruktion. Jvf. ideerne om faded guidance og scaffolding.
    - *Eksempelvis at man kan forstå en E/R-model og ændre udvalgte dele af denne så resultatet er korrekt syntaktisk og semantisk.*
- **Skaberkompetence**. Man kan anvende, ændre og skabe nye modeller.
    - *Eksempelvis at man kan udforme et UML sekvensdiagram, der udtrykker samarbejdet mellem en række objekter.*

Vi skelner yderligere mellem tre typer af ændringskompetencer:

- **Konkretisering**. At gøre en abstrakt model mere konkret.
    - *Eksempelvis at erstatte abstrakte klasser med konkrete klasser i et klassediagram.*
- **Udvide**. At tilføje noget til en model.
    - *Eksempelvis at tilføje klasser til et klassediagram.*
- **Restrukturere**. At ændre modellen, så den funktionelt udtrykker det samme, men udtrykker ændrede ikke-funktionelle egenskaber.
    - *Eksempelvis at indføre en række design patterns i sit klassediagram.*