Bliv udvikler

- den kreative programmør

- Normann Aa. Nielsen, 2003 -

At være udvikler er konstant at have øjnene åbne
for de ting, der ikke findes
og for de ting, der findes
men som kan ændres.

Det fortælles at en jøde under anden verdenskrig blev taget til fange af Gestapo.

”Nah, Schweinhund,” brølede obersten, ”her er to æsker med hver sin seddel. På den ene seddel står der at du skal skydes – på den anden seddel står der, at du bliver frigivet. Nu skal du trække en seddel, og din Gud vil bestemme din dom!”

Men SS-officeren var en grusom mand, der havde skrevet dødsdommen på begge sedlerne.

Jøden stak hånden ned i en af æskerne, men før nogen kunne nå at stoppe ham, stak han sedlen i munden og sank. Obersten var rasende.

”Teuffel!” skreg han, ”hvordan skal vi nu finde ud af hvad der stod på sedlen?!”

”Nemt nok,” sagde jøden roligt, ”hvis den anden seddel siger at jeg skal skydes, så må den seddel jeg spiste jo være min frigivelse”.

Indhold

Generelt om udvikling

I den senere tid er der opstået forskellige retninger, når det drejer sig om programudvikling. På mange måder er det mere udtryk for en ændring i arbejdsklimaet i USA (og store firmaer) end egentlig nytænkning, men naturligvis er der metoder som man kan bruge.

Først og fremmest er det dog vigtigt at gøre sig klart hvilken situation, man selv er i. Kun ved denne erkendelse kan man finde det rette niveau at starte på. Dette er selvklart - alligevel er det ikke altid noget, der sker af sig selv. Vi taler her både om en udviklers erfaringsniveau og om vidensniveauet. Disse to niveauer følges ikke ad!

Ligeledes er det vigtigt at kende sine omgivelsers niveau. Dette bliver i særdeleshed vigtigt, når man skal udveksle oplysninger eller anmode om resourcer. Hvis man f.eks. kender sin chefs niveau går man ikke så nemt galt i byen med at formulere en ansøning om et eller andet redskab.

1. Den uvidende
   Man har ikke engang hørt om emnet, og man erkender det, hvis man bliver spurgt. "Jeg aner ikke hvad en computer er."

2. Den interesserede
   Man er netop blevet introduceret for emnet, og aner intet om hvor det kan føre hen. Man ved, at man intet ved, men måske skal man finde ud af det. "Nå, så COBOL er et programmeringssprog - kan man kode spil med det?"

3. Novicen
   Man er igang med at prøve emnet af på egen hånd med den viden, man selv har skaffet sig. Endnu ved man ikke hvor meget, man skal sætte sig ind i. "Det er da meget nemt at kode i Java, men der mangler nu en masse omkring skærmstyring."

4. Amatøren.
   Man har arbejdet ustruktureret med emnet gennem et stykke tid, og elsker det - samtidig har man specialiseret sig inden for bestemte dele, som man selv finder interessante. Inden for disse ting er man kongen, mens man ikke vil indrømme uvidenhed om andre dele af emnet. "Procedurer er ineffektive - makroer kan flere ting, hvis man husker at slå switch -o fra."

5. Den studerende
   Man har mødt andre, og har set at de kan andre ting end een selv. Man begynder nu at lære de systematiske og grundlæggende regler for emnet, samtidig med at man slipper af med fejlagtige anskuelser. "Så man skal altså aldrig bruge goto!"

6. Praktikanten
   De første rigtige opgaver skal udføres, og de idéer, man har lært, skal afprøves. Man begynder at anvende "best practices". "Det er ikke altid at en database skal undgå redundans!"

7. Specialisten
   Man kan det grundlæggende flydende, og kender en hel del teknikker. Nu har man specialiseret sig i et særligt delemne, og andre spørger een til råds omkring dette emne. "Fakturainformationen er spredt på KUFAHEKA og KUFALIKA, men du skal anvende KUFLHEKA for at finde følgesedlen."

8. Eksperten
   Man kan alt omkring emnet, og bliver spurgt om de mest besyndelige dele. En del af arbejdstiden går med at undersøge særlige dele om emnet, og andre emner der støder til det. "En sjov ting ved LOCK er at den ikke låser tabel 0 - det er en fejl, men den udnyttes faktisk af rapportgeneratoren."

9. Guru / arkitekten
   Man ser ikke længere emnet som adskilt fra verden. Man tænker muligheder. Man er kreativ og designer nyt omkring emnet. "Hvordan kan vi udnytte goto-konstruktionen objektorienteret?"

Der findes to nedre niveauer før denne opstilling:

0. Den bedrevidende uvidende
   Man har ikke hørt om emnet - men man prøver at bluffe, for at undgå at tabe ansigt. "Man skal huske at sætte harddisken til skærmen."

-1. Den dominerende bedrevidende uvidende
   Man har ikke hørt om emnet, men det er også komplet ligemeget, for folk skal bare gøre hvad der bliver sagt, og forklaringer er alligevel bare undskyldninger for ikke at ville gøre arbejdet! Meget irriterende, og findes typisk blandt traditionelt opdragede (mænd). "Det kan godt være at du siger at rumtemperaturen er for høj for serveren, men så må du lade være med at bruge den så meget."

Og der findes et enkelt niveau ud over det sidste:

∞. Evangelisten
   Man rejser rundt og fortæller om emnet. Man skriver bøger og giver interviews. Man er blevet kapaciteten bag emnet. "Jamen, Kent Beck siger selv..."

Valget af udviklingsmetode afgøres ikke altid af programmøren selv. Som nyankommen eller i større grupper er der ofte allerede taget en beslutning - der ofte er et blandingsprodukt af nyt og gammelt samt skidt og godt.

Eksempel:
Hvis de første programmører i firmaet var selvlærte blev der sandsynligvis ikke indført moderne metoder i starten. Det afspejler sig helt ned i koden, der bærer præg af mangel på indsigt i metoden. Som nyankommen tør man ikke pille ved noget som helst - hvad værre, man begynder at adaptere de patterns, der bruges, uanset om de er sunde eller ej.

En nyankommen får sjældent lov til at arbejde med nyudvikling af større ting. Som oftest skal man lave fejlrettelser eller arbejde med mindre (allerede designet) opgaver. Man har her en følelse af uvirkelighed - man ved ikke nok om det system, man er ved at modificere, og de test, man udfører, er kun deltest.

Som erfaren arbejder man med langt mere komplekse situationer, hvor man er i tæt kontakt med kunden eller designeren - eller værre, sælgeren... Her er en hemmelighed: Man har her en følelse af uvirkelighed - man ved ikke nok om det system, man er ved at modificere, og de test, man udfører, er kun deltest.

Derfor er det vigtigt at have en struktureret tilgang til udvikling, fra idé til realisering.

Structured development (SD)

Denne model er den traditionelle, som man har benyttet sig af siden midten / slutningen af 1960'erne. Den kaldes ofte for "vandfaldsmodellen", idet en fase skal være færdig før den næste kan begynde. Dens fordel er - set med admistratores øjne - at den er kraftigt dokumentdrevet, og at den arbejder med aktiviteterne svarende til en traditionel produktionslinie / samlebånd. Metoden afspejler naturligvis sin tids hierakiske synspunkt, og blev indført for at modstå den stadigt voksende software krise - programmerne blev større og større, kompleksiteten voksede og fejlraten voksede tilsvarende. I dette synspunkt er programmøren en maskine, der kan udskiftes uden videre - han skal udelukkende følge specifikationerne, for kun derved kan fejl opdages og fjernes.

SD inddeles i følgende faser:

Problemfase -> (problemformulering, idéer til løsning)
- Problemstudiet => opgavebeskrivelse

Analysefase -> (alternativer, konsekvensvurdering)
- Målstudiet => Målbeskrivelse
- Informationsstudiet => Informationsbeskrivelse
- Behandlingsstudiet => Behandligsbeskrivelse

Designfase -> (programmeringsgrundlag)
- Systemstudiet => Systemforslag
- Systemspecificeringen => Systembeskrivelse

Gennemførelsesfase -> (programmeringen, brugevejledning)
- Detailudformningen => Detailbeskrivelse
- Systemindførelsen => Overdragelsesrapport

Driftsfase -> (tilretning af system, mål af godhed)
- Efterstudiet => Godkendelsesrapport

SD danner grundlaget for flere af de team-organiseringer, som siden har været anvendt i mange andre sammenhæng. Især er det værd at hæfte sig ved følgende:

Det egofri team
Ledelse af teamet kan gå på skift, lige som alle andre opgaver. Det vigtigste drejer sig om at undgå *ego*-istisk aktivitet, idet ændringer i kode, som een bestemt person "føler" for kan medføre konflikter på det personlige plan. Teamet er kontrolleret af ledelsen.

Chief-programmer team
Her betyder "Chief" den første blandt ligemænd (og ikke den øverste i et hieraki). Chef-programmøren er en senior, der er teknisk ansvarlig for hele projektet. Andre personer i teamet er bibliotekar (eller teknisk assistent), back up-programmøren, specialister etc. Teamet er autonomt, men melder tilbage til ledelsen.

Begreber som top-down, bottom-up og middle-out programmering fremkommer i denne periode, ligesom ideen om strukturerede flow (goto-fri kodning) er det bærende mantra.

Det største problem med traditionel SD er hvornår det kreative arbejde finder sted. SD lægger op til at det sker i begyndelsen af projektet, med få muligheder for at ændre undervejs. Programmørerne har i denne model kun ansvaret for at implementere det ønskede - men er naturligvis fri til at vælge den algoritme, som er bedst.

Objekt oriented development (OOD)

SIMULA (1968) var fra starten tænkt som et sprog, der kunne beskrive fysiske virkeligheder (klasser), og nærmest som biting kunne man så anvende den resulterede beskrivelse som kode. Dette var unikt - og blev alt for ofte overset.

Imidlertid begyndte arbejdet med objektorienterede metoder især fra midten af 1970'erne. I første omgang teoretisk, men kraftigt støttet af det ene sprog efter det andet ('70-'80-erne var også årene, hvor compilerteknologien blev fintunet).

Lige i hælene på metoder og sprog kom de objektorienterede designs og modeller.

Den objektorienterede udvikling fokuserer på taxonomi, dvs. klassifikation af opgavens forskellige elementer. Denne måde at tænke på kan være betydelig givende, men også noget underlig når man er trænet i den originale SD-metode.

Dogmet i OOD er klassen, den abstrakte datatype. Man må ikke arbejde direkte med klassens indre tilstande, men skal anvende et sæt af metoder / interfaces til dette. Undervejs i designet opstilles et netværk (framework) af sammenhænge mellem de realiserede klasser (dvs. objekterne), og metoderne. Man kan således ikke tale om top-down eller bottom-up udvikling, for abstraktionsniveauet ændres løbende.

Denne metode blev først udviklet på læreanstalterne, og kom derfor ikke ud til virksomhederne i struktureret form. Mange firmaer kender stadig ikke til det grundlæggende i metoden (opdeling af opgaven i abstrakte datatyper), og ser med skepsis på anvendeligheden! Grunden er den simple, at den bryder "vandfaldsmodellen" omkring design- og gennemførelsesfasen. Den stringente (bureaukratiske) trangfølge brydes op.

OOD tillader at designer / kunde og programmør / designer arbejder tæt sammen om idéen i opgaven. Designeren kan anvende abstrakte beskrivelser (klassediagrammer) til at udtrykke tanker over for både kunde og koder. Koderen kan tillade sig at lave prototyper i klasserne. Designeren kan gøre det samme, på et overordnet plan.

Man var sikker på at man ved at anvende disse egenskaber gennemgribende kunne automatisere kodeproduktionen. Dette medførte udviklingen af CASE-værktøjer - meningen var, at designeren skulle lade computeren producere koden ud fra færdigtegnede klassediagrammer. Tankegangen er typisk administrativ (problemerne ligger altid på det nederste niveau), og det viser sig at der endnu ikke har været den store success med CASE-værktøjet.

Model driven architecture / development (MDA / MDD)

Selv om der er flere metodologier at arbejde med, så leder OOD meget direkte hen til tanken om MDA / MDD. Groft taget vil man udspænde OOD til at dække alle systemudviklingens faser, idet man anvender særlige diagrammeringssprog til at beskrive med.

Kernen i MDA / MDD er sproget UML (Universal Modelling Language), som i sin fulde version skal kunne beskrive hvadsomhelst... Det er baseret på en naturlig udvikling af beskrivelsessprog som f.eks. IDL, der blev introduceret i '90-erne i forbindelse med standardiseringen af objekt-interfaces (CORBA, COM+). Men UML er langt mere avanceret, idet det kan beskrive krav til implementationen, som IDL ikke var i stand til (f.eks. timing i forbindelse med realtidsopgaver).

Dybest set er UML at betragte som et højt udviklet grafisk programmeringssprog til objekt-netværket. Og der er derfor også udviklet systemer, der tager en UML-grafik og producerer kode. Så skal programmøren bare udfylde de relevante kasser. Hvis man vil, kan man endog have biblioteker af standard klasser, hvorefter systemet selv kan skrive koden på disse steder. På mange måder er MDA / MDD en moderne CASE-metodologi.

Der er utroligt mange fordele ved UML, hvoraf den vigtigste uden tvivl er standardiseringen. Der var en grund til at flowcharts var så populære i sin tid - alle kunne forstå dem. UML (og dermed MDA / MDD) er fra starten orienteret mod anvendelsen af OD og flere samtidige udviklere på et system. UML indeholder også nok værktøjer til at man kan lave "development by contract", hvor man arbejder med black-box teknikker (det er sådan set bare gammel vin med nyt navn!).

MDA / MDD tvinger ikke organisationen til at ændre på den måde, som udviklingen foregår på. Det er muligt at bibeholde selv den mest rigide vandfaldsmodel. Imidlertid tillader MDA / MDD (rigtigt brugt!) at nye udviklingsmetodologier afprøves og indføres.

Det mest radikale med MDA / MDD er at man udvikler en model over den opgave, der skal løses. Man (designeren) arbejder med modellen indtil den i grundtrækkene gør det, den skal (i beskrivelse), og under dette forløb opstår der naturligt en lang række beskrivelser. Ved anvendelsen af de rette værktøjer vil disse beskrivelser kunne udskrives øjeblikkeligt, eller automatisk videreføres til programmeringsfasen.

Den kreative side af sagen er derfor lagt i hænderne på designeren til at definere modellen over systemet. Programmøren kan anvende sin kreativitet til at udvikle den underliggende framework og fremstille generelle klasser, som kan anvendes på senere tidspunkter.

eXtreme programming (XP)

I forordet til [3] skriver Erich Gamma: *Extreme Programming nominates coding as the key activity throughout a sotware project. This can't possibly work!* Men det er hele idéen bag XP, at programmering er i højsædet.

Traditionel tænkning har alvorlige problemer med denne holdning. Her drejer det sig om at sikre at alting er drøftet og klart før kodningen kan foregå, så den ikke bruger uforholdvist lang tid på eventuelle problemer. Det skulle nødigt være sådan at programmøren spilder tid med at skulle ringe til kunden...

Bortset fra, at dette netop er hvad XP tillader. XP opdeler sin metodologi i følgende praktiske regler:

The Planning Game
Foretage en hurtig afgrænsning af omfanget af den næste release ved at kombinere forretningsprioriteter og tekniske estimater. Opdater planen, når virkeligheden trænger sig på.

Small releases
Sæt hurtigt et simpelt system i produktion, frigiv dernæst nye versioner i hurtig rækkefølge.

Metaphor
Al udvikling skal følges af en simpel, fælles forklaring på hvordan hele systemet fungerer.

Simple design
Systemet skal designes på den på ethvert tidspunkt enkleste måde. Ekstra / overflødig kompleksitet fjernes så snart den opdages.

Testing
Koderne skal løbende udføre test af komponenter, der kræves at køre fejlfrit før udvikling kan fortsætte. Kunderne konstruerer test, der viser om de ønskede faciliteter er færdige.

Refactoring
Koderne foretager omstrukturering af systemet uden at ændre dets virkemåde, for derigennem at fjerne redundans, forbedre kommunikation, forenkle eller tilføje fleksibilitet.

Pair programming
Al produktionskode skrives af to programmører ved een maskine.

Collective ownership
Enhver tillades at kunne ændre hvilken som helst kode hvor som helst i systemet på vilkårlige tidspunkter (*ego*-fri).

Continous integration
Systemet integreres og buildes flere gange dagligt, hver gang en opgave er afsluttet.

40-hour week (eller: 37-timers arbejdsuge)
Som hovedregel: Arbejd ikke mere end den normale arbejdstid. Gå aldrig på overtid to uger i træk.

On-site customer
Involvér en ægte, levende bruger i projektet, som er tilgængelig fuldtid for at besvare spørgsmål.

Coding standards
Programmørerne skriver al kode i overenstemmelse med fastsatte regler som forstærker kommunikationen i koden.

En væsentlig del af XP drejer sig om at opdele en opgave i mindre dele, som man hurtigt kan estimere (""planning game") og efterfølgende løse som en helhed. Man opererer med begrebet TimeBox, som er den tid en delopgave tager. Eftersom disse delopgaver skal kunne løses som en helhed, betragtes en TimeBox som ubrydelig - i det mindste, indtil virkeligheden ændrer på opgaven.

Man bemærker at XP tillader kreativitet efter del-og-hersk princippet på alle tidspunkter i processen. Man bemærker også at kunden skal involveres på alle tidspunkter. Det er her meget tydeligt at man forsøger at ryste den gamle bureaukratiske struktur af sig - i Skandinavien plejer dette trods alt ikke at være et problem...

Ved indførelsen af XP anbefales følgende måde:

1. Udvælg dit værste problem
2. Løs det ved brug af XP-metoden
3. Når det ikke længere er dit værste problem, gå tilbage til 1.

Der er i XP også råd om hvordan man indretter sin arbejdsplads. Det har betydning, især for de steder hvor folk ikke sidder sammen. XP kan således opfattes som en helheds-filosofi.

Pragmatic development (PD)

Denne metodologi er både farlig og fremragende: Den er fremragende, fordi den arbejder med de bedste råd, man kan give - og den er på samme tid farlig, fordi man skal have et vist niveau før man virkelig får noget ud af rådene. De, der har anmeldt [4] på amazon.co.uk, deler sig i to lejre: De begejstrede, som typisk er erfarne professionelle - og de andre, som typisk er mindre erfarne.

PD drejer sig om at finde løsninger og metoder, der synes at virke, og slippe af med dem, der ikke gør. Dette er meget pragmatisk, og PD-folkene har da heller ikke en helstøbt teori for hvad man skal gøre. Fordelen med PD er, at man kan benytte standardværket [4] direkte - ulempen er altså, at det er svært at videregive til studerende.

I [2] omtales 3 adfærdsstadier som man gennemgår, når et nyt emne indlæres. Det drejer sig om følgagtighed (following), frigjorthed (detaching) og flydende (fluent):

Følgagtighed
Man leder efter een metode, der virker. Selv hvis der er 10 andre metoder, der hver især kunne virke, kan man ikke lære alle på een gang - man ønsker at kende een, der "altid" virker. (Ex.: Bubblesort)

Frigjorthed
Man erkender nu svagheden ved kun at kende den ene metode (man lærer begrænsningerne). Man er klar til at lære de 10 andre metode at kende, og man lærer hvornår en metode er bedre end andre. (Ex.: Quisksort)

Flydende
Man har sin viden integreret i sig selv, gennem utallige tanker og tilstande. Det er nu irrellevant hvilken teknik man anvender eller ej. Man kan reelt ikke svare direkte på hvordan man når frem til den metode, man vælger - det er naturligt. (Ex.: Bubblesort)

PD henvender sig til udviklere på det frigjorte niveau, og er kondenseret viden fra udviklere på det flydende niveau. Det giver sig udslag i den måde PD viderekommunikeres på, nemlig som et sæt af gode råd, evt. med programeksempler og case stories. [4] har 70 gode råd og 11 checklister. Den ældre [6] har flere råd, men udmærker sig ved at henvende sig nærmere til det følgagtige niveau.

Her er eksempler på høj-niveau råd fra [6]:

- Use the requirements checklist at the end of the section to assess the quality of your requirements
- Make sure everyone knows the cost of requirements changes
- Set up a change-control procedure
- Use development approaches that accomodate changes

Her er eksempler på høj-niveau råd fra [4]:

- Put abstractions in Code, Details in Metadata
- Always design for concurrency
- Don't use wizard code you don't understand

⑩ Start when you are ready

Der er meget store abstraktioner i PD, og nogle forfattere (bl.a. [2]) har direkte sammenlignet nogle af rådene med Zen.

Den sidste råd i [4] er særdeles interessant, når man sammenligner med SD. Det lyder:

⑩ Sign your work
"Craftsmen of an earlier age were proud to sign their work. You should be, too."

Rådet går umiddelbart direkte imod tanken om det ego-fri programmeringsteam. Men rådet skal forstås på den måde, at man skal have stoltheden tilbage. "Dette er kode, som JEG har været med til at lave, og jeg står ved den." - Dette indebærer ansvarlighed, over for test, dokumentation, undervisning, drift og vedligeholdelse.

PD er blevet beskyldt for at være for vag. Der er nogen grad af sandhed i det, men grunden er, at PD ikke lever i et tomrum. Den lægger ingen krav på omgivelserne, men til gengæld kræver den meget af sine udøvere.

Agile development (AD)

Alle de forskellige metodologier smelter sammen i AD. I foråret 2001 samledes 17 af de mest kendte metode-evangelister i Utah for at diskutere hvad metoderne havde fælles. Ud af dette møde kom "Agile Development Manifestet", som - i min oversættelse - lyder således:

"Vi afdækker bedre måder at udvikle software på ved at gøre det og hjælpe andre til at gøre det. Gennem dette arbejde er vi nået til at værdsætte:

- Personer og relationer over processer og værktøjer
- Software der virker over udførlig dokumentation
- Kundesamarbejde over kontraktforhandlinger
- Ændringsparat over plantænkning

Det vil sige: Selv om der er værdi i emnerne på højre side af skemaet værdsætter vi emnerne på venstre side mest."

Bemærk, hvor kontroversielt AD er i forhold til f.eks. SD eller i det hele taget traditionel udvikling. F.eks. er det for AD mere værdifuldt at fremkomme med software, der virker - frem for at have udførlig dokumentation. Der er mange traditionelt tænkende tekniske chefer, der har fået noget galt i halsen ved denne metodologi!

Men læg vel mærke til, at der er tale om noget helt andet end anarki. Den originale AD-gruppe gør udtrykkeligt opmærksom på, at man ikke er interesseret i at "rive hele huset ned", men at man ønsker som udvikler at være i stand til at kunne give et intelligent modsvar til det pres, der uværgeligt kommer fra omgivelserne, og som er den største plage for systemudvikling. Man gør eksplicit opmærksom på, at der ikke er noget "modsat AD", på samme måde som der ikke er noget "modsat Bengalsk tiger" - der er i stedet alternativer. Endelig er der plads til uenighed i de 4 nævnte værdisætninger.

Ud over de 4 værdisætninger fremførte gruppen 12 anbefalinger, som man forventer modifikationer til. Jeg nævner her nogle, som jeg finder er vigtige:

1) Vores højeste prioritet er at tilfredsstille kunden gennem tidlig og ofte levering af værdifuld software.

3) Software, der virker, er det primære mål af fremdrift.

5) Forretningsfolk og udviklere samarbejder dagligt gennem projektet.

9) Fortløbende opmærksomhed på teknisk udmærkelse og godt design forøger fleksibiliteten.

12) Regelmæssigt vil holdet overveje hvordan det kan blive mere effektiv, og derefter justere sin adfærd i forhold hertil.

Det radikalt forskellige i AD sammenlignet med den traditionelle tænknig er at udvikleren sættes i det samme centrum som kunden. Det er her, at idéerne skal komme, det er her at løsningen skal laves. På mange måder afspejler AD den skandinaviske tankegang, og [2] referer da også til både Peter Naur og Pelle Ehn.

Min egen filosofi...

Jeg er mest inspireret af tankerne i [1] og [4], kombineret. De to vigtigste regler i min karriere har været:

1. Lad være!
2. Vent!! (kun for eksperter...)

Disse to regler er grundlæggende sunde, og stammer helt tilbage fra Michael A. Jackson (Principles of Program Design, 1975). Jackson anvender dem i forbindelse med optimering af programmer, men reglerne rækker langt videre.

3. Opbyg en model før kodning

Man skal ikke forsøge at komplicere verden mere end den er. Når vi skriver et system, så skal det kun gøre det, vi beder det om - og ikke andet. Modellen skal beskrive nøjagtigt det, vi interesserer os for.

4. Gør det generelt

På den anden side skal modellen kunne udvides (inden for sit domæne). Reglen om generalitet er den mest avancerede og vanskelligste, man kan forlange fulgt. Jeg har set mange dygtige udviklere, der ikke har været i stand til at følge denne regel, og kun gøre det halvt og desværre er en halv generel løsning værre end en specialiseret løsning!

5. Gør det enkelt - undgå kompleksitet

Denne regel kan afledes af de to første. Det er den gamle KISS (keep it simple, stupid!), som er den eneste regel man skal huske som amatør.

6. Lær konstant!

Dette er meta-reglen over alle. Den skal være ristet ind i enhvers hjerte.

Indretning af udviklingsområde

Erfaringer fra kreative værksteder fortæller at et almindeligt kontorlokale sjældent er det mest inspirerende for nytænkning. Til gengæld opfylder det den traditionelle bureaukratiske tanke om at medarbejderne kan behandles eens over een kam. Det er ofte slående hvor stor kontrasten kan være mellem ledelsens kontorer og udviklernes.

Nogle udviklere foretrækker at arbejde i enrum. [1] advarer mod denne tendens: "Jo mere isolerede programmører er, jo mere knytter de sig til deres programmer - typisk begynder de at opkalde deres bedste 'værker' efter sig selv." "Problemet er, at når en programmør ser programmer som udvidelse af sit eget jeg, kan det medføre psykologiske problemer, når programmet viser sig at være fejlbehæftet." Her argumenteres kraftigt for ego-fri programmeringsteams. Dette gøres til et kardinal-punkt i XP (se [3]) og dermed også i AD.

Der skal først og fremmest være plads. Plads til at to kodere kan sidde ved siden af hinanden med plads til noter. En af dem skal kunne bruge computeren, begge skal kunne se skærmen. En kunde skal kunne sidde komfortabelt ved siden af udvikleren. Der skal være plads til kaffe / the og andet socialt.

Figur 1: Det kreative hjemmekontor...

Der skal være en stor tavle. Whiteboard eller almindelig tavle er godt, en flip-over kan bruges i nødstilfælde (den har den fordel at man kan tage papiret med sig). De fleste kender problemet med at tage resultatet med sig, men man kan i dag få systemer, der er rimelig i pris og kan overføre grafikken til computer. Eller man kan anvende et digitalt kamera.

En eller flere opslagstavler til hvadsomhelst, men især til de skitser, man når frem til er vigtige. Opslagstavlen er ikke statisk, den skal fornys og ændres konstant, i takt med projekternes udvikling.

Planter og andet miljø er vigtige. Men det skal være uproblematiske planter, for programmørerne gør ikke noget for at holde dem i live.

Hvis vi taler om et rum hvor mange arbejder, skal der være mulighed for at kunne mødes samlet

omkring et fælles bord. Der skal tages hensyn til rygepolitik - det er nemt at definere at ingen må ryge, men inkarnerede rygere har et fysisk problem med det, og der skal også tages hensyn til dem.

Det akustiske niveau er vigtigt. Nogle programmører foretrækker stilhed, andre vil gerne have høj musik kørende. Lave skillevægge kan hjælpe her, men man skal kunne se over dem nemt. Under ingen omstændigheder skal man have kontormuzak - det er kvælende for kreativiteten!

Billeder og kreative småting overalt. Og frem for alt: Biblioteket skal være øjeblikkeligt tilgængeligt. Det drejer sig om fagbøger og -blade, men skal ikke begrænses til dette.

Og resten (lys osv.) skal naturligvis følge arbejdstilsynet...

Hvad er kreativitet?

Jeg er nu igang med den klassiske måde at arbejde på. Jeg har skrevet en overskrift, der udtrykker det spørgsmål, jeg ønsker at besvare. Besvarelsen af spørgsmålet forventes at være fyldestgørende for den målgruppe, jeg henvender mig til.

Er dette nu en kreativ aktivitet?

Udsagn om kreativitet:

Kreativitet er lystbetonet.

Det drejer sig om at gøre noget andet end det faste arbejde.

Det er ønskeligt at der kommer et (brugbart) resultat ud af den kreative proces.

Intution er kreativitet og omvendt.

Zen og erkendelsesspring

I Zen-buddismen arbejder man med at forløse den logisk / deduktive tankegang. Det drejer sig om at udløse en erkendelse, der kun kan opnås via intuitionen. Der anvendes mange teknikker for at hjælpe den søgende til det sidste spring ud i erkendelsen.

Den japanske satori-buddisme lader en munk uventet give den mediterende et slag bagfra. Men det sker ikke tilfældigt: Den munk, der uddeler slaget, skal være klar over at den mediterende er ved at nærme sig det oplyste stade. Først i det øjeblik uddeles slaget, og den mediterende bruger dette til at frigøre sig fra det bundne og blive oplyst.

Andre anvender en æstetisk teknik med at udtænke og studere digte, for eksempel *Haiku-digte*, der i høj grad konstrueres med en stringent og regelbundet metode, men som – yderst modsætningsfyldt – indeholder en æstetik eller intuition, som ikke kan udtrykkes sprogligt på anden måde.

En gammel
dam
En frø
springer i
Lyden af vand
Plask

En meget kendt zen-buddistisk metode går ud på at bryde ordenes indbyggede mening. Dette sker ved at misbruge – at *overfortolke* – ord ved at anvende paradokser, de såkaldte **kôan**. Her er et eksempel:

Kôan:

Shuzan held out his short staff and said: "If you call this as short staff, you oppose its reality. If you do not call it a short staff, you ignore the fact. Now what do you wish to call this?"

Mumon's Commentary:

If you call this a short staff, you oppose its reality. If you do not call it a short staff, you ignore the fact. It cannot be expressed with words and it cannot be expressed without words. Now say quickly what it is.

Mumon's Poem:

Holding out the short staff,
He gave an order of life and death.
Positive and negative interwoven,
Even Buddhas and patriarchs cannot escape this attack.

(Douglas R. Hofstadter: Gödel, Escher, Bach)

Rigtigt anvendt kan disse paradoksale metoder udløse en erkendelse. Men det sker ikke med det samme – typisk taler vi om mange års stadig koncentration. Men når det lykkes, så bryder intuitionen igennem, og man ser verden på en ny måde.

Man er blevet oplyst!

Det var ligesom at knuse et islag eller at rive et krystaltåm ned. Da jeg pludselig vågnede og kom til mig selv, fandt jeg mig selv som værende ligesom mester Yen-tou, som i alle tre tider (fortid, nutid, fremtid) ikke mødte nogen lidelser Alle tidligere tvivl var blevet opløst ligesom is, der smelter væk. Med en høj stemme kaldte jeg: "Hvor vidunderligt, hvor vidunderligt!'
(*Jørn Borup: Zen. Levende Japansk Buddhisme. Munksgaard 1998.*)

Nu er Zen ikke det nemmeste at komme ind i for vesterlændinge. Imidlertid har visse vittigheder spor af det samme paradoks, som Zen prøver at forfølge. Vi vil ikke forvente erkendelser komme ud af vittigheder, men som kreativ inspiration kan de bruges.

Et eksempel:

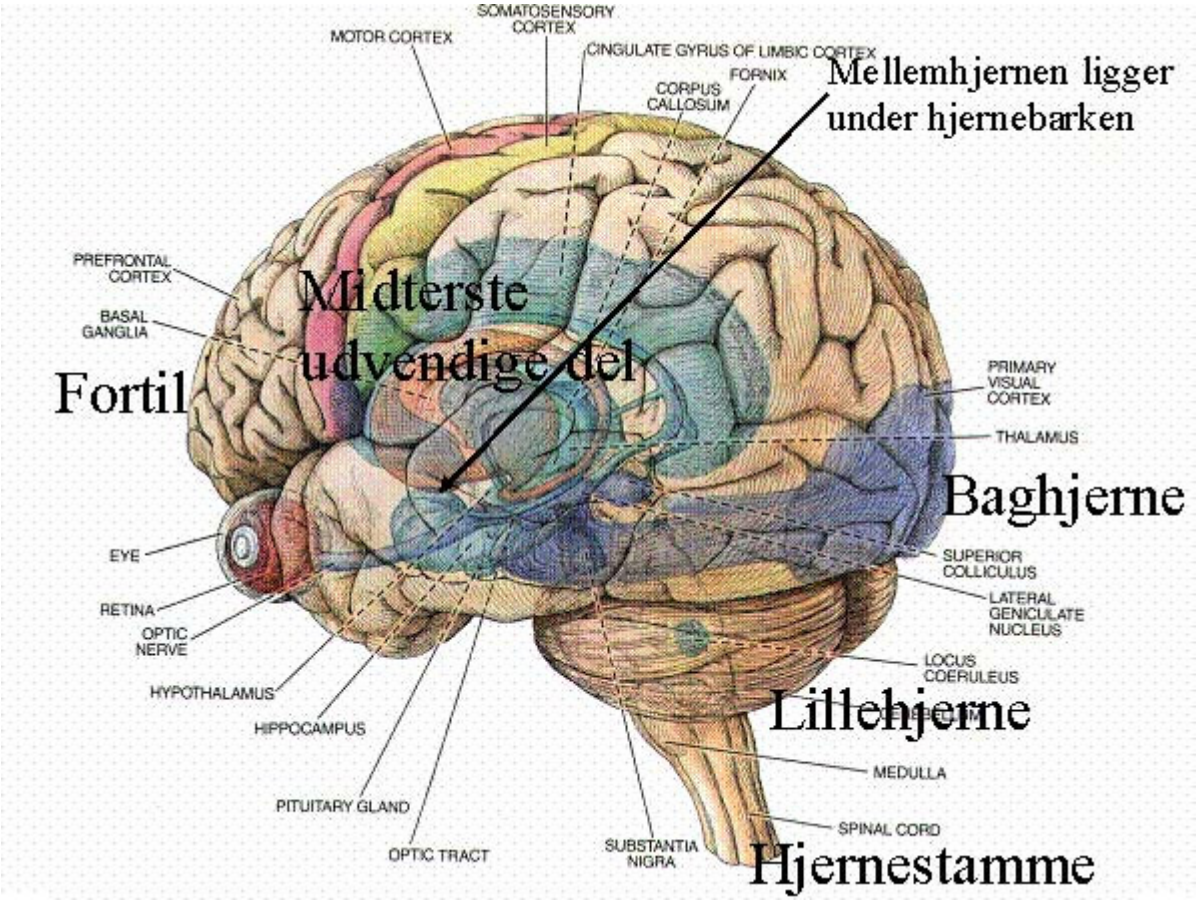
Rosen er rød min ven
Blå er forglemmigej
Hvis digtet ikke rimer, er der dem
der synes det er meget mere kunstnerisk.

Højre / venstre hjernehalvdel og de 10% effektivitet.

Den almindelige myte siger, at vi kun udnytter ca. 10% af vores hjernekapacitet på et givent tidspunkt. Derfor drejer de fleste kurser i kreativitet og selvudvikling sig om at aktivere mere end disse 10%. Det er i hvert fald ofte det, der fortælles i introduktionsmaterialet.

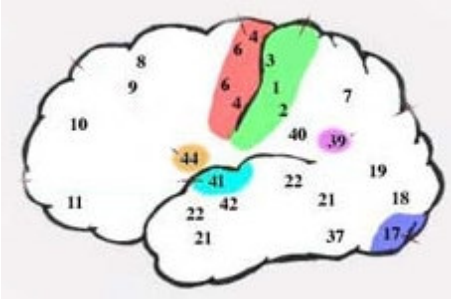
Imidlertid er dette tal i bedste fald grebet ud af luften, og i værste fald underbygget med tvivlsomme præmisser. Det er klart at hjernen arbejder hele tiden, men stofskiftet gennem forskellige centre i hjernen ændrer sig løbende, alt efter hvilke opgavetyper, man arbejder med.

Hvad man *kan* tale om er at hjernen er hovedopdelt i centre (se figuren).



Figur 2: Hjernens forskellige centre (www.dr.dk)

I skematisk form er hjernecentrene fordelt som følgende figur viser:



Figur 3: Hjernens centre, skematisk (www.dr.dk)

1,2,3 - Fornemmelser for berøring samt vægt, form, størrelse, tekstur, temperatur og rystelser.
4 - Primære bevægelsesområde
5,7,40 - Sanserne
8 - Øjenbevægelser, visuelle reflekser, pupillens sammentrækning og udvidelse.
9,10,11 - Dømmekraft og ræsonnering (biologisk intelligens).
17,18,19 - Synet
21,22 - Hørelsen
41 - Hørelsen
42 - Opfattelse og forståelse af tale
44 - Brocas center: Tale

Men hvad siger disse kort over vores vigtigste organ os? Med mindre vi er hjernespecialister, der ønsker at kunne udtale os omkring læsioner i hjernen, så er det begrænset, hvad der umiddelbart kan opnås. Med een undtagelse dog: Øget stofskifte i et af hjernens områder er et udtryk for øget aktivitet i området. Hvis man derfor er interesseret i at stimulere et bestemt område, så kan man forsøge at påvirke et eller flere nabo-områder, og derved i det mindste stimulere randområdet.

Syv stadier for indstudering

Udarbejdet af cellisten og dirigenten Hans Erik Deckert (professor v. Det jyske Musik-konservatorium, Århus).

1. ”Skizze”: Orientere sig i værket, prøver på bedste beskub at spille sig igennem, få en første forestilling om hvad det er.
2. Analysen: Overskue værkets struktur, lave små forøvelser.
3. Detailarbejdet: Systematisk instudering af enkeltheder, teknisk indøvning af fraser.
4. Koordinationen: Samarbejdet mellem uafhængige enheder skal finde sted.
5. Identifikation: Overførsel af ego til instrumentet.
6. Opbygning: Forsøg på at opfatte systemet udefra, opfatte som om man ser sig selv som systemet.
7. Trancendens: Overskridelse af grænsen mellem system og sig selv. ”Nu swinger det!”

Fremlæggelse og kommunikation

Den kreative idé kan ikke stå alene. Den skal bearbejdes og ofte fremlægges i et forum af andre, som ikke nødvendigvis er dine kolleger – eller bare dine fagfæller.

Fremlæggelse for andre ”hjemme i firmaet”.

- Hvordan vil man gøre?
- Hvem kommer og lytter (målgruppe)
- Hvordan følger du op?
- Hvad forventer du som resultat?

Tal sproget:

- Hos kunden: Sproget kan være ”skraldespansk” eller forfinet.
- Chefen fatter ikke en bønne af tekniksnak
- Dine kolleger forstår ikke hvor du vil hen...
- Du kan være i et andet land: Lær alm. høflighedsgloser

Inspirationskilder

Når man skal aktivere sine kreative evner behøver man ikke at starte fra intet. Noget af det mest kreativt aktiverende kommer fra at betragte andre menneskers produktioner. Imidlertid er det vigtigt at undgå situationer, hvor man analyserer frem for at associere. Derfor er det – i modelfasen – vigtigt ikke at tænke i programtekniske baner. Og i kodningsfasen bør man ikke starte med at læse kode. I stedet skal man forsøge at starte sin inspiration et helt andet sted.

Til gengæld skal man heller ikke blive hængende her for længe, før man begynder at arbejde med den konkrete situation. I en startfase skifter man mellem hurtige glimt af forskellige inspirations-kilder, og mellem de enkelte kilder bør man vende sig mod den konkrete opgave.

Faren ved at bruge for lang tid på inspirationskilderne er at man kan komme for langt væk fra den opgave, der i virkeligheden skal løses. Hvis man f.eks. bruger computerspil som associativ hjælp, skal man være parat til at afbryde spillet før det faktisk er færdigt – og alligevel skal man påse at man får tid nok i spillet til at føle, at man er kommet ind i det. Det er en svær balancegang, fordi en kimen fra et minut-ur kan fjerne alle associative idéer før man husker dem. Tænk på, hvor mange drømme der er glemt, når vækkeuret ringer!

Skønlitteratur

Science fiction: Historier af typen ”hvad-nu-hvis...”

Clarke, Arthur C.: Fra den hvide hjort
Delany, Samuel R.: Skæringspunkt
Joseph, M.K.: Hullet i nullet
Silverberg, Robert: Kubikroden af usikkerhed

Fantasy fiction: Historier af typen ”et andet miljø / anden tid”

LeGuin, Ursula: Troldmanden fra Jordhavet
Rice, Anne: Vampyr-bøger
Tolkien, J.R.R.: Eventyret om Ringen

Rejser

Hermed mener jeg ikke ud-at-drikke rejser eller afslapningsrejser, men rejser til steder, man ikke kan have forestillinger om eller fuldkommen information om.

F.eks. rejser til Fjernøsten eller Afrika – eller en tur til London's slumkvarter.

Figur 4: Det uventede i en kultur giver ofte gode associationer.

Eller 14 dage i Lejre som stenaldermenneske.

Figur 5: Glem gamle rutiner! (http://www.lejre-center.dk)

Computerspil

Denne type spil har ofte været undervurderet (”det er bare skydespil”, ”man lærer jo ikke noget af det”, ”man bliver socialt ensom”). Typisk kommer denne type vurderinger fra folk, der ikke har beskæftiget sig med computerspil, eller ikke har afprøvet de nyeste.

Det er korrekt at computerspil ikke er det samme som alm. spil, der jo har en social funktion (hygge, familienærvær osv.). Imidlertid er der ofte et stort socialt netværk (netcaféer, gameparties), også udspredt over Internet (i multi-player spil).

Vi taler her om de nyeste computerspil (ikke simple enspænder-spil som Tetris eller Kabale). Denne type computerspil er interaktion med en ikke-menneskelig intelligens (af varierende kvalitet).

3 hovedtyper:

Actionspil medvirker til at øge reaktionshastighed, korttidshukommelse og koncentration.
Eksempel: No One Lives Forever.

Adventurespil medvirker til at forbedre evnen til problemløsning og øge langtidshukommelse.
Eksempel: Myst / Riven (adventure + strategi).

Strategispil forbedrer strategisk kunnen og rationelle evner. Eksempel: Unreal Tournament.

I det hele taget er det vigtigt at blive udfordret fra uventet kant. Den kunstige intelligens i de mere avancerede spil kan – når den er bedst – give en uventet modstand. Man kan blive stillet over for gåder, som kræver høj grad af indlevelsesevne i den verden, spillet arbejder i.

Kunst

Der er grundlæggende to type kunst: Den beskrivende og den abstrakte. For udløsning af det kreative element bør kunsten netop være abstrakt. Her undlader man en række regler, som man ellers skal overholde i det beskrivende forløb, f.eks. perspektiv eller bare kravet om at tingene *skal ligne*. Men umiddelbart fremstår der færre fritløbende associationer ved beskrivende kunst end ved den abstrakte.

Herunder er eksempler på abstrakt billedkunst. Se på billederne enkeltvist et par minutter, og ned-skriv samtidig nogle af de associationer, som dukker op.

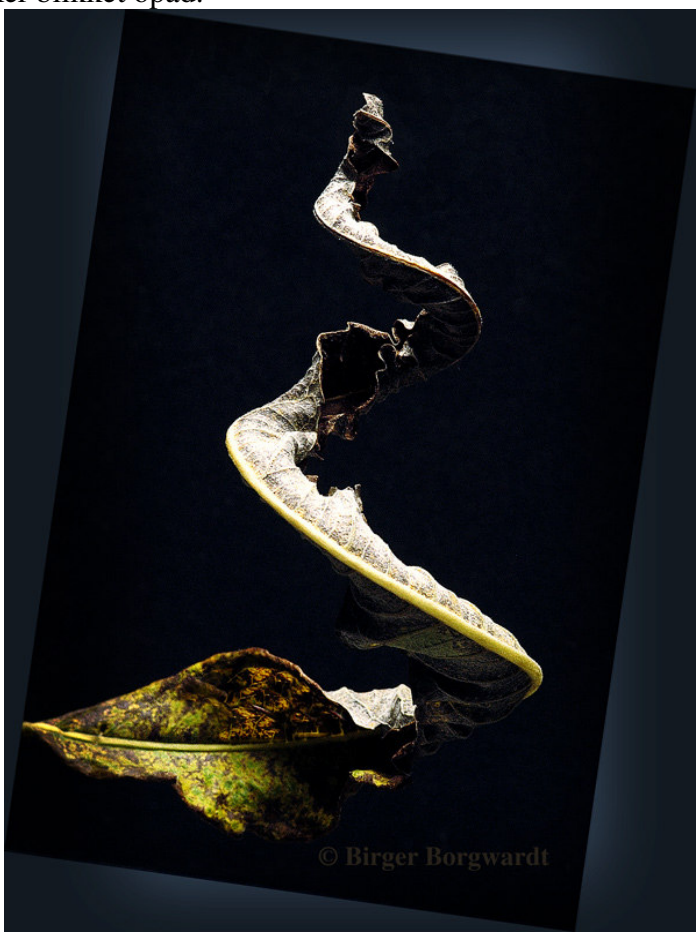
Af tekniske grunde er billederne her ikke trykt i farver. Hvilken betydning har det?

Dette billede er tilsyneladende ikke abstrakt, men hvad forestiller det så (*Gaugain*)?

Dali's bløde og udflydende landskaber har alle dage talt til fantasien. Er dette et billede af tiden, der skal digitaliseres?

Det foruroligende ved *Magritte* synes at være at det er velkendte objekter, der sammenstilles i en ny og dermed spændingsfyldt sammenhæng.

Dette billede af *Birger Borgwardt (www.fotokritik.dk)* udnytter kontrastvirkninger. Spiralen, som dannes af bladet, trækker blikket opad.



Musik

Det er en kendt sag at musik giver mulighed for associationer. Forskellig musik benyttes til forskellige formål. Noget er af streng, rituel karakter, andet er frit fabulerende – grundlæggende er det eens kultur, der afgør hvordan musikken kan anvendes og hvilke associationer, der udløses.

Der er lavet mange målinger i forbindelse med musikalsk påvirkning. Alt afhængigt hvilken musik, man hører, sker der ændringer i hjerterytme / blodtryk og øvrige stofskiftesystemer. Der er påvist at man under musikalsk påvirkning opnår en såkaldt forøget spatial intelligens (rumlig / perspektivisk opfattelse), der dog forsvinder umiddelbart efter påvirkningens ophør.

Det, der er vigtigt, er at finde ud af hvordan man skal anvende en bestemt type musikalsk påvirkning for forøgelse af den kreative evne. Mange udtrykker at de ikke ønsker at lytte til musik mens de arbejder, for "så kan de ikke koncentrere sig". Det er korrekt – men ved stimulering af kreativitet ønsker vi at se bort fra den logiske og dømmende intelligens, og vil i stedet fokusere på den abstraherende og billeddannende intelligens.

De følgende 6 musikalske eksempler spænder vidt. Det er derfor interessant at vide hvilke associationer, som opnås ved at lytte til hvert musikstykke. I virkeligheden bør musikudvalget være større.

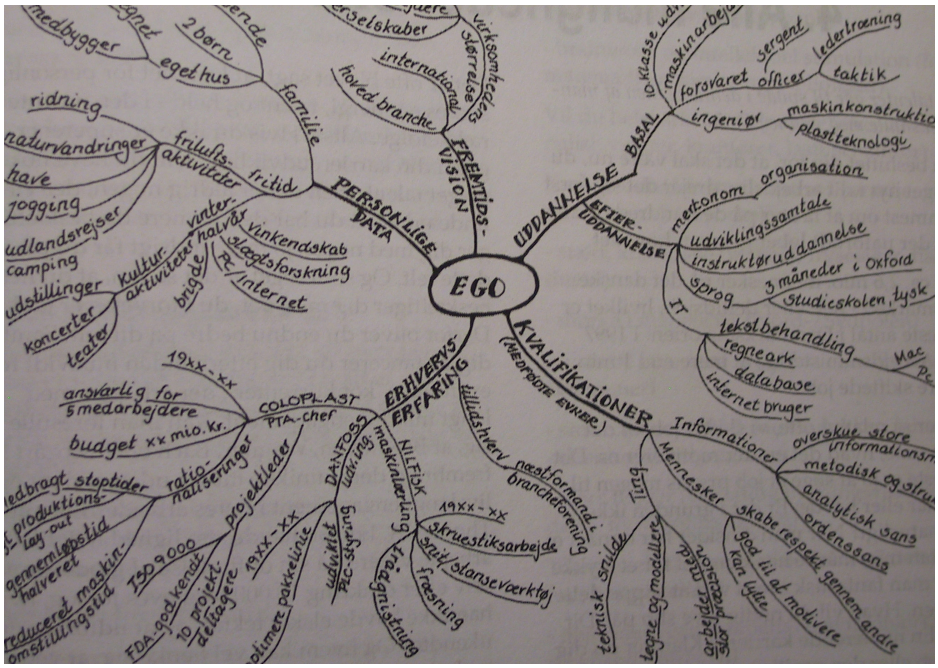
De 6 stykker er:

- Music of the Malinké: Festival of the Circumcision
- Alanis Morisette
- Carl Nielsen: Andante Lamentoso
- Florence Foster Jenkins: Like a Bird
- Mithrandir: Eye of the Lens – #1
- Mithrandir: Eye of the Lens – fanfare

Kreative teknikker

- Mindmap

Et MindMap er en særlig associativ notatteknik uden de sædvanlige lineære grænser. Meningen er at generere ideer og spor, som man skal vende tilbage til på et tidspunkt. Teknikken er simpel: Man tager et meget stort stykke papir (gerne A3 eller A2) eller en tavle.



Figur 6: Mindmap over emnet "Ego"

Det emne, man er interesseret i, skrives med et eller få ord midt på tavlen. Der opstår herved omgående et sæt af associationer – hovedlinierne danner hovedspor, der forgrenes og danner nye spor. Ingen spor må krydse hinanden. Figuren viser et eksempel (fra [12]).

- Brainstorm

Teknikken ved brainstorm kan udføres på forskellige måder. Hvis det anvendes som et led i problemløsning vil fremgangsmåden være:

- Problemet formuleres kortfattet på skrift (helst kun et problem af gangen).
- Brainstorm på løsningsforslag. Alle ideer og løsningsforslag skal accepteres (i første omgang). Giv hjernen frit spil i eks. 20 minutter, hvor alle indfald af løsningsforslag noteres ned.
- Prioritering og konsolidering af løsningsforslag betyder at løsningsforslagene et for et skal vurderes i forhold til nogle fastsatte kriterier. Her ud fra kan forslaget prioriteres i forhold til om det kan bruges nu, senere eller aldrig

- "Visitkort" / postit

Skriv stikord på visitkort, hvert visitkort får sit stikord. Under ordet skrives en meget kort forklaring (hvis nødvendigt). Hver gang et nyt stikord dukker op, skrives et nyt visitkort. Alle kortene lægges på et bord i alm. uorden. Se på bordet. Træk relevante kort ud, bær dem til et andet bord, og start

analysen af dem der. Når analysen er (næsten) færdig, gå tilbage til den første bord, og betragt igen stikordene – er nogle af dem nu klaret? Er der nu nogle nye, relevante?

- Spørge-Jørgen (Compass, [11], s. 19)

Formuler problemet som en særning, der starter med ordene: Jeg ønsker at... eller tilsvarende formulering. Modparten (eller een selv) spørger nu ”hvorfor?”, og svarene skal registreres. Metoden kendes fra børn...

- Spørg på Internettet

Surf koncentreret i 20 minutter med alle de søgemaskiner og portaler der er til rådighed. Søg bredt. Læs derefter den indfangne høst.

Prøv med problemet: Introduktion af EDB i børnehave

Case stories

I dette emne skal man fremlægge sine egne iagtagelser, historier og erfaringer. Men da det kan være svært at komme igang, har jeg medbragt noget fra mit egen karriere.

1) Sorteringsanlæg
   I første omgang var der flere special løsninger. Det kreative element drejede sig om at se, hvordan man kunne forene løsningerne i een. En objekt orienteret model viste sig at bære frugt; selv om den tog tid at lave, kom tiden tilbage igen ved andre installationer - koden blev genbrugt!

2) ULTRA
   Opgaven drejede sig om at udtænke en måde, hvorpå man kunne komme videre med at kode i en moderne verden, men stadig kunne bruge den proprietære database (og dog alligevel gøre det muligt at fjerne sig fra den også). Løsningen var arkitektonisk ved brug af Java, JDBC og JNI sammen med C++ og et lavniveau bibliotek.

3) DSFL-oversætter
   Kort og andet grafisk materiale der produceres ud fra overflyvninger leveres i dag på det såkaldte DSFL-format. Dette format er platform-uafhængigt. Der skulle skrives et program, der konverterede DSFL-formatet til et konkret grafisk system (Microstation MDL). Opgaven blev lettere, da jeg indså at DSFL skulle opfattes som et programmeringssprog, hvor MDL var den producerede kode.

Tankeopgaver

Forudsætning: Du er udvikler, du er dygtig og erfaren, og du er vant til at finde udveje, hvor der ikke er nogen.

1. Da du vågner er du spærret inde i et nøgent rum. Det eneste, du kan se er to døre og en seddel på gulvet. På sedlen står: "Bag den ene dør er der en frygtelig drage. Bag den anden dør er der en frygtelig drage." Hvordan slipper du ud?

2. Du ønsker at få bedre arbejdsforhold, men din chef aner intet om hvad dit arbejde går ud på, og er ofte ret skræmmende. Hvad gør du?

3. Du dømmes til eksil på en øde ø (og jeg mener øde - ingen MacDonnalds!). Hvilke 5 ting vil du tage med?

4. Det er nu bevist at ligningen:

   a^n + b^n = c^n , a,b,c,n heltal, n>2

   ingen løsninger har. Det tog flere hundrede år at bevise det, og kræver den højeste matematiske kunnen. Hvordan vil du bevise det?

5. Hvordan bygger man en robot, der kan gå ned af trapper?

6. Konstruer en intelligent musefælde.

7. Andrew er 7 år og meget syg. Han er født uden ben, og nu er han ved at dø. Han har kun eet ønske tilbage, og det er at vide, hvordan det er at cykle. Fortæl ham hvordan det er - hvordan holder man balancen, hvordan gør man osv.

8. Mennesket har lavet ting i mange ti-tusinder af år. En af de ældste artefakter (ca. 6000 år gammel), der ikke lader sig forbedre, er en ske. Prøv at foreslå forbedringer.

9. Der er blevet fremsat forslag om at samordne landets børnehaver med skolefritidsordningerne. Der kræves et idékatalog, som viser hvilke IT-muligheder, man har, når man skal iagtage krav om økonomi og tyndt befolkede områder. Giv nogle bud på mulighederne.

Litteratur

[1] Søvndahl, T.; Traberg, J.: Pålidelig programmering
Gyldendal 1977, ISBN: 87-01-55222-8.

[2] Cockburn, A.: Agile Software Development
Addison-Wesley 2002, ISBN: 02-01-69969-9

[3] Beck, K.: eXtreme Programming eXplained
Addison-Wesley 1999, ISBN: 02-01-61641-6

[4] Hunt, A.; Thomas, D.: The Pragmatic Programmer
Addison-Wesley 1999, ISBN: 02-01-61622-X

[5] Glass, R.L.: Facts and Fallacies of Software Engineering
Addison-Wesley 2003, ISBN: 03-21-11742-5

[6] McConnell, S.: Code Complete
Microsoft Press 1993, ISBN: 15-56-15484-4

[7] Fulton, J.: MENSA - Styrk din intelligens
Aschehoug Dansk Forlag A/S 2000, ISBN: 87-11-12909-3

[8] Gamma, E. et al.: Design Patterns
Addison-Wesley 2001, ISBN: 02-01-63361-2

[9] McKeown, M.; Whiteley, P.: Unshrink
Prentice Hall 2002, ISBN: 02-73-65614-7

[10] Brooks Jr., F.P.: The Mythical Man-Month
Addison-Wesley 2003, ISBN: 02-01-83595-9

[11] Clegg, B.; Birch, P.: Instant Creativity
Kogan Page 1999, ISBN: 07-49-42949-6

[12] Lausten, T.: Jobjagt og karriereskift
Libris 2003, ISBN: 87-78-43633-8

Forslag til tidsplan for workshop

9:10- 9:30 Ankomst
9:30-10:00 Præsentation, baggrund
10:00-11:00 Case stories, tankeopgaver
11:00-12:00 Gennemgang af fagets udviklingsmetodologi
12:00-12:30 Frokost
12:30-13:00 Fagets udviklingsmetodologi fortsat
13:00-14:00 Kreative teknikker
14:00-15:00 Opgaver / workshop
15:00-16:00 Case stories, debat, afslutning