Surveys on Software Development

Claus Jensen (Editor)

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
surf@diku.dk

Contributors

Tina A. G. Andersen
Ulrik Schou Jørgensen
Bo Simonsen

CPH STL Report 2008-3, July 2008.

Preface

This volume contains a selection of surveys presented at the software development workshop organized in connection with the course "Selected topics in software development" (taught by Jyrki Katajainen, Lars Yde, and myself). The workshop was held April 11, 2008 at the University of Copenhagen. In the course it was a compulsory requirement to write a survey, and three of the surveys were invited to be included in this collection. The survey could be written on a topic that was discussed earlier in the course or a topic that was omitted by the teachers, but which the student found interesting and which was related to the main theme of the course, communication, understood in a broad sense.

The course home page can be found at: `http://www.diku.dk/forskning/performance-engineering/Courses/Software-development-2008/`. In the lectures the following topics were covered:

- development process models,
- design patterns,
- programming style,
- programming psychology,
- software testing, and
- software tools.

The surveys included in this collection deal with human factors, anti-patterns, and extreme programming, complementing the topics covered by the lectures very well.

I would like to thank all the students for their efforts into making the course a success.

July 2008 Claus Jensen

Table of contents

Den menneskelige faktor i softwareudvikling

Tina A. G. Andersen

Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø, Danmark
taga@diku.dk

Resumé. I dette survey vil jeg studere den menneskelige faktor i system-/softwareudvikling. Da emnet er meget bredt, vil jeg fokusere på at danne et overblik. Udfordringen er, at finde de vinkler der kan udforskes nærmere i fremtidige surveys. Nogle af de spørgsmål jeg har stillet mig selv undervejs har været:

- Hvad betyder de individuelle personer for softwareudvikling?
- Er det en fordel, at investere tid i personlig udvikling, når man beskæftiger sig med softwareudvikling? Hvis ja, for hvem?
- Hvordan opnås personlig udvikling i virksomheder der beskæftiger sig med softwareudvikling?
- Hvordan påvirker personlig udvikling softwareudvikling?
- Er der en sammenhæng mellem faglig og personlig udvikling?

Disse spørgsmål besvares ikke direkte i denne opgave, men de repræsenterer tanker og kilder der kan føre læseren tættere på nogle svar. Fokus er primært lagt på personlig udvikling, og især personlighedstester, da der lægges op til bedst mulig målbarhed ved typeinddeling.

1. Personlig udvikling

Det kunne være meget interessant at undersøge hvilken indflydelse det har på systemudvikling, at der også er fokus på personlig udvikling, ikke mindst vha. de nedenfor skitserede metoder. Spørgsmålet er, om det gør en forskel for:

- kvaliteten og/eller tempoet af udviklingen
- den enkelte deltager i projektet.

Der er altså både det (relativt) kortsigtede perspektiv, der handler om det aktuelle projekt og det langsigtede perspektiv, der handler om fastholdelse og dygtiggørelse af medarbejdere samt virksomhedens udvikling.

Kommunikation er en af de vigtigste parametre i systemudvikling, hvilket bakkes op af [6]. Jeg tror de fleste har prøvet at skulle samarbejde med at andet menneske, men oplever at kommunikationen ikke fungerer. Der kan være mange måder det går galt på.

Et eksempel er, når der anvendes ironi eller den anden person på anden måde ikke siger det der faktisk menes. Dette kan give nogle meget ubehagelige situationer, fordi der i mange tilfælde vil gå en rum tid før man erkender, at den anden ikke udtrykker det han faktisk mener, og fordi det

Skrevet april 2008. Revideret juli 2008.

efterfølgende kan være svært at gætte hvornår han siger det han mener og mener det han siger.

Det kan eksempelvis være, hvis vedkommende konsekvent roser ting han mener er dårlige. I den situation er det ikke til at vide, hvornår han faktisk mener at en metode er brugbar til formålet, og hvornår han blot er ironisk.

Kommunikationen kan også fejle, når to personers begrebsdefinition er forskellig (prøv bare at skrive at du nyder at bruge LaTeX til en udenforstående, og vedkommende vil sikkert tro der er tale om en påklædningsfetich). Denne brist opdages oftest relativt hurtigt, og er sjældent ubehagelig eller pinlig, da det blot er et spørgsmål om at *ensrette* definitioner.

Det sidste eksempel er af "faglig" karakter, mens det første eksempel er et spørgsmål om personlighed og adfærd.

Det er klart, at der er brug for en vis faglig konsensus, for at kommunikationen skal fungere, og jeg finder Clines teori [1] om designmønstre meget interessant og udfordrende, idet der lægges op til, at man skal anvende en pæn del af sin fritid på at holde sig opdateret vedrørende designmønstre. Jeg mener dog, at man kan diskutere, om en arbejdsgiver kan/bør stille krav i den retning, eller om ikke der burde sættes tid af til dette indenfor arbejdstiden.

For langt de fleste (softwareudviklings)virksomheder i dag gælder, at hvis der er faglige udfordringer, så kan der tages kurser, købes bøger, videndeles osv.

De kommunikationsproblemer der skyldes personlighed og adfærd er desværre ofte mere følsomme, og det er især svært, hvis det er en ellers fagligt dygtig person eller en leder der ikke er konstruktiv i/for gruppen. Det kan dels være svært for de kolleger der ser problemet, at udtrykke det overfor vedkommende på en konstruktiv måde, og dels kan det være meget svært for den der får kritikken, at modtage den uden at begynde at forsvare sig selv og sine handlinger.

Det kan eksempelvis være en chef der virker meget aggressiv. Denne aggressivitet kan være en fordel når ting skal gennemføres på et højere ledelsesniveau, men det kan skabe store problemer, hvis medarbejderne føler sig angrebet af chefen.

Det er efter min opfattelse generelt mere tabu at erkende at man har brug for personlig udvikling, end hvis det er faglig udvikling man mangler. Jeg ser dog oftere og oftere at virksomheder prøver at åbne op for personlig udvikling gennem eksempelvis personlighedstest af medarbejderne. Et eksempel på en sådan testtype er MBTI-modellen.

Hvis MBTI-modellen ikke er kendt af læseren anbefales det at læse om det, f.eks. i [2], inden der læses videre her.

1.1 Personlighedstest

Jeg vil starte med at påpege, at jeg ikke er certificeret indenfor personlighedstester, og derfor i høj grad udtaler mig på baggrund af andre kilder, herunder [4] og [2].

En indgangsvinkel til personlig udvikling er, at lave en personlighedstest af medlemmerne i projektgruppen, teamet eller måske hele virksomheden. Som nævnt i [2] skal man dog være varsom med personlighedstester, da der er forskellige faldgrupper. Kvaliteten af en test er bl.a. afhængig af konteksten den tages i, samt hvordan der efterbehandles på de svar der er givet i multiple-choice delen.

Der findes en række gratis udgaver af MBTI, f.eks. på hjemmesiderne `http://www.jobindex.dk/cgi/typeindikator.cgi` og `http://www.kandu.dk/Tip13729.aspx`, hvor enhver frit kan tage en test. Et af problemerne ved disse tester er, at der ikke bliver fulgt op med en personlig samtale med en certificeret konsulent. Som det påpeges i [4] er der desuden ingen sikkerhed for, om testen er pålidelig og korrekt gennemarbejdet.

Som også nævnt i [4] spiller omgivelserne også en stor rolle for resultatet, idet deltagerens seriøsitet kan påvirkes kraftigt af de signaler hjemmesiden sender.

Et interessant studie kunne være, at analysere den online udgave af MBTI der anvendes på DIKU i forbindelse med førsteårsprojektet, hvor de studerende skal tage en test, for derefter at bestemme rollerne i gruppen ud fra personlighederne. Dette er et konkret emne der ligger op til et studie i sig selv, idet der er mange aspekter at analysere og stille spørgsmål til i den fremgangsmåde. (F.eks. kvaliteten af svarene, de studerendes forudsætninger for at tolke svarene og efterfølgende fordele roller ud fra disse samt hvorvidt det er meningsfyldt at fordele roller ud fra typer, uden at betragte faglige forudsætninger osv.).

Hvis testen er en "autoriseret" udgave af MBTI, blot uden opfølgende samtale, kan man i bedste fald risikere, at præferencer der ikke er helt tydelige ikke bliver vendt. Her forstås, at man kan ligge ca. midt i mellem to præferencer, f.eks. sansning [S] og intuition [N]. Hvis man ikke får resultatet gennemgået personligt i en samtale, er det ikke sikkert, at man er opmærksom på, at man i nogle situationer kan svare til den side der ikke er tydeligt dominerende.

Det svarer lidt til, at man som højrehåndet kan træne sig op til at skrive med venstre hånd, men at det alligevel altid vil være lettere at skrive med højre. Dog findes der, som tidligere omtalt, folk der ligger ca. midt i mellem, og som i princippet kan skrive lige godt med begge hænder. Det er desuden vigtigt, at folk der tager testen bliver gjort bevidst om, at de ikke er fastlåste i den type de testes til, men at den blot, hvis den er taget korrekt, viser hvordan men typisk vil reagere i en presset/stresset situation, og hvad der er "det nemmeste" for en at gøre.

I værre situationer kan man forestille sig, at en test udgiver sig for at være en MBTI-test, men blot er en forsimplet udgave, der mangler den grundige bearbejdning samt standardisering [4]. Det vil ikke kun bevirke, at der ikke kan sammenlignes på resultater mellem forskellige projekter, men kan også bevirke, at den enkelte får et misvisende resultat der ikke kan genskabes i en efterfølgende test.

1.2 Konkret anvendelse af MBTI

Personligt har jeg en ganske god erfaring med MBTI. På min arbejdsplads testes alle professionelt, som det også beskrives i [4], med en efterfølgende to-timers samtale med en certificeret konsulent. Man har mulighed for at kontakte denne konsulent efterfølgende, hvis man har behov for det, idet vedkommende er fastansat i firmaet. Hver medarbejder får desuden udleveret en bog, hvori alle 16 typer gennemgås, og deres indbyrdes forhold også omtales.

Herefter anvendes typerne til, at man ved projektstart hver i sær fortæller lidt om sig selv, ud fra sin type. Det skal nævnes, at sammensætningen af en projektgruppe sker udelukkende efter faglige forudsætninger, og det altså først er efterfølgende typebestemmelserne kommer ind i billedet.

Min rent subjektive opfattelse er, at det er et meget stærkt værktøj for den enkelte, idet der er en "standard" at tale ud fra, og idet vi hver i sær har konstant adgang til beskrivelser af de andre typer, der kan hjælpe os til at forstå og rumme reaktioner, som vi ellers ikke finder rationelle, fra andre typer.

Personligt har jeg oplevet, at potentielle konflikter i en presset tid, f.eks. op til en deadline, er blevet afvæbnet, ved at minde sig selv om, at den andens reaktion betyder noget andet, end det man lige opfatter. Desuden ser jeg også ind i mellem, at kolleger giver støtte på en måde der reelt virker støttende på den enkelte, frem for at give støtte som man selv ville foretrække at modtage den.

Med det i mente vil jeg argumentere for, at den bedste udnyttelse af personlighedstester fås ved at lade kolleger kende hinandens typer, og ikke holde det for sig selv, som der ellers tales om i [2].

En alternativ personlighedstest er *enneagrammet*, der angiver ni forskellige typer, der hver har forskellige kendetegn inden for motivation, adfærd, sprog, kommunikation og beslutningsprocesser. I [5] findes en grundig gennemgang af de ni typer, samt en beskrivelse af, hvordan man finder sin egen type. Efter sigende har enneagrammet en mere holistisk tilgang til menneskets personlighed end mange andre værktøjer. Det er nok i høj grad et spørgsmål om erfaringer og temperament, hvilken typeindeksering man foretrækker. I [4] omtales eksempelvis en form der hedder 16PF. Jeg har ikke kendskab til detaljerne i denne test, men de argumenterer for, at den skulle være at foretrække frem for MBTI.

Ligesom i [6] fokuseres der også på kommunikation i [5], idet de mener, at man gennem enneagrammet kan blive bedre til at kommunikere på en mere konstruktiv måde. Faktisk handler en stor del af bogen om, hvordan de forskellige typer kommunikerer, og hvordan man får det bedste ud af denne kommunikation.

En vigtig pointe i [5] er, at man netop skal dele informationerne om sin type, for bl.a. at kunne sammensætte grupper således, at der dels er de rigtige personlige ressourcer tilstede og dels er harmoni med gruppen. De mener desuden, at enneagrammet kan anvendes til rekruttering af nye folk.

Jeg er dog generelt skeptisk overfor at anvende typeindeksering til rekruttering, idet de faglige forudsætninger ikke må glemmes, og man skal huske, at enhver typeindeksering har sine begrænsninger. Der er ingen garanti for, at personen med den "rette" profil også passer ind i gruppen og virksomheden i øvrigt.

I flere kilder nævnes den generelle kritik af personlighedstester, at indekseringen er begrænsende, hvilket jeg er enig i og derfor vil opfordre til at man husker når man anvender dem.

Slutteligt indeholder [5] et helt kapitel om coaching i team.

1.3 Coaching

Der findes andre afsæt til personlig udvikling end typeindeksering; f.eks. coaching. Coaching ses anvendt i en række større virksomheder, ligesom der også er et reelt marked for privatpersoner der søger coaching. I coaching bliver personen ikke grupperet som ved personlighedstest, og coaching kan ikke automatiseres og udføres over nettet, hvilket giver en mere individuel behandling.

Denne individuelle udvikling bevirker, at medlemmerne af et team eller en gruppe ikke opnår indsigt i hinandens "ståsted", idet samtalerne med en coach typisk sker på tomandshånd. Der sker altså muligvis en udvikling med den enkelte, men der er ingen garanti for at der sker en generel udvikling i gruppen, og der lægges ikke op til et stærkere team.

Dette kan være en fordel for den enkelte medarbejder, hvis denne har et stærkt ønske om at beskytte sin intimsfære, og finder det grænseoverskridende at delagtiggøre andre i private tanke og handlemønstre. Man kan dog spørge sig selv, om en sådan person passer ind i et team der skal arbejde sammen, eller om vedkommende egner sig bedre til opgaver der løses individuelt.

Det er vigtigt at tilføje, at typeindeksering og coaching ikke udelukker hinanden, og at der også er eksempler på, at en gruppe kan modtage coaching, hvis de samlet set står overfor en problemstilling.

Det har desværre ikke været muligt, inden for denne opgaves rammer, at finde artikler der omtaler coaching i direkte sammenhæng med softwareudvikling, men der findes talrige konsulenthuse der tilbyder coaching, også rettet mod den type virksomheder samt flere artikler og bøger om coaching i team generelt. Manglen på artikler der belyser dette område åbner op for den spændende udfordring, at der kunne laves en undersøgelse af emnet.

Coaching handler ikke kun om personlig udvikling, idet der ligesåvel kan coaches over faglige problemstillinger. Bogen [7] giver eksempler på dette.

2. Faglig udvikling

I [3] fokuseres på, at den primære ressource i systemudvikling er den menneskelige intelligens, og det primære værktøj er programmøren. Det er essentielt at erkende, at den enkelte programmør selv er ansvarlig for at have

gode vaner, og at det ikke handler om hvor intelligent man er, men om hvordan man fokuserer sin intelligens. Faktisk er det en pointe i [3], at de bedste programmører er dem der erkender, hvor små deres hjerner er, idet de er ydmyge og generelt fokuserer på at skrive kode på en måde der gør at den kan læses og forstås af både dem selv og andre.

Prioritering af tid er en anden ting der omtales i [3]. Her tales der dels om, at programmører ofte har så travlt med at arbejde, at de ikke har tid til at være nysgerrige efter måder de måske kan gøre deres arbejde bedre. Desuden tales om den tid der anvendes til at tænke. Når man tænker ser man sjældent travl ud, men en meget vigtig del af arbejdet som programmør er at tænke sig om, og det er derfor vigtigt, at det er acceptabelt at man ikke altid ser travl ud.

Fokus i [3] er på den faglige udvikling, selvom bl.a. vaner også er et spørgsmål om personlighed, og derfor også kan behandles under personlig udvikling.

For at udvikle sig fagligt er det en forudsætning, at man har en idÃ© om hvor man står som udgangspunkt. I [3] angives en "4-trins raket", der starter ved begynder og ender ved guru, og der angives en liste over bøger man bør læse for at udvikle sig fra et givet niveau til det næste. Dette er en interessant inddeling, og jeg tror at den på mange måder kan være nyttig, men den stiller dog et væsentligt krav til den enkelte, nemlig at vedkommende er ærlig og realistisk.

Det ses jævnligt, at programmører har et forbløffende ego, og en selvtillid der får dem til at tro at de er eksperter. [3] og [6] giver eksempler på, at den programmør/udvikler der mener at han er ekspert sjældent vil være motiveret for at udvikle sig fagligt eller personligt, og jeg vil vove den påstand, at vi alle kender en der kender en med netop dette problem.

Som sagt har vaner stor betydning for fagligheden og den faglige udvikling. Det gælder om, at opbygge de bedste vaner fra start, og hvis man har fået en dårlig vane gælder det om at skifte den ud med en god. Definitionen af gode og mindre gode vaner kan diskuteres, idet der findes mange forskellige kodestile, men overordnet set er der enighed om, at man skal huske at man primært skriver koden til et andet menneske og kun sekundært til oversætteren. At ændre den slags vaner kræver for langt de fleste en udfordrende udvikling (se bare på en ryger der ønsker at blive eksryger).

I [1] er fokus også på den faglige udvikling, men igen er der en stærk sammenhæng mellem den faglige og den personlige udvikling, idet der argumenteres for, at (objektorienterede) udviklere konstant må have en personlig interesse i at holde sig opdateret og læse/eksperimentere.

Der argumenteres for, at hvis alle der arbejder på et (objektorienteret) projekt anvender designmønstre, så har alle de samme konventioner, eller vaner, at gå ud fra, hvilket bl.a. letter kommunikationen og sikrer en vis standard. Prisen er dog, at alle skal gennemgå en konstant faglig udvikling, da der konstant udvikles nye designmønstre.

For at holde lysten og energien oppe til at kunne blive i denne faglige udvikling kan der argumenteres for, at det også er nødvendigt med en passende

personlig udvikling. Denne teori finder jeg personligt interessant, og tænker, at det kunne være spændende at undersøge, om der er en sammenhæng mellem lysten til faglig udvikling og lysten til personlig udvikling. Denne sammenhæng berøres dog ikke direkte i de nævnte kilder.

Særligt interessant kunne det være, at se på lederes udvikling kontra medarbejdernes opfattelse af deres lederegenskaber.

3. Konklusion

På baggrund af de i indledningen stillede spørgsmål vil jeg mene, at det er oplagt at se på følgende mulige studier:

- Hvordan fungerer MBTI for DIKUs førsteårsprojekter?
  - Hvad er formålet set fra DIKUs synsvinkel?
  - Opnås formålet?
  - Hvordan opfatter de studerende det?
- Hvad forventer hhv. private og offentlige virksomheder at få ud af at anvende MBTI?
  - Hvad får de faktisk ud af det?
  - Kan disse virksomheder sammenlignes på metoderne?
- Hvilke virksomheder anvender coaching direkte i forbindelse med systemudvikling?
  - Anvender de også coaching til personlig udvikling?
  - Hvilke erfaringer har de gjort?
- Hvad er betydningen af faglig udvikling kontra personlig udvikling?
  - På lederniveau, efter ledelsens egen opfattelse?
  - På lederniveau, efter medarbejdernes opfattelse?
  - Er der sammenhæng mellem lederes opfattelse og medarbejderes opfattelse?

Litteratur

[1] Marshall P. Cline, The pros and cons of adopting and applying design patterns in the real world, *Communications of the ACM* **39**, 10 (1996), 47–49
[2] Christopher Derek Curry og Jyrki Katajainen, *Reengineering a university department*, kapitel 5, Jyrki Katajainen and Company (2006)
[3] Steve McConnell, *Code complete*, 2. udgave, Microsoft Press (2004)
[4] Sharon McDomald og Helen M. Edwards, Who should test whom?, *Communications of the ACM* **50**, 1 (2007), 67–71
[5] Edit Moltke-Leth og Anne Lundgren, *Teamudvikling*, Borgen (2002)
[6] Gerald M. Weinberg, *The psychology of computer programming*, 25-års jubilæumsudgave, kapitel 4, Dorset House Publishing (1996)
[7] John Whitmore, *Coaching på jobbet*, Peter Asschenfeldts Nye Forlag (1998)

Antimønstre i softwareudvikling

Ulrik Schou Jørgensen

Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø

Resumé. Formålet med denne rapport er at sammenfatte, hvad antimønstre er, hvilke forkellige typer der findes og beskrive nogle enkelte af dem. Antimønstre er designmønstres modstykke: *En gentagen måde hvorpå man kommer fra et problem til en dårlig løsning.* Antimønstre er gode at kende til i forbindelse med softwareudvuikling, lige fra når det gælder overordnede designbeslutninger til programmørens daglige arbejde. Et antimønster beskrives på følgende form: Først gives baggrunden for antimønstret, dernæst hvordan det optræder, og til sidst, når det er sket, hvordan man eventuelt kan gå ind og lave en indgreb der fjerner problemet.

1. Antimønstre

Antimønstre er inspireret designmønstre af Gamma et al. [2], men ud over disse objektorienterede antimønstre findes en række andre typer af antimønstre:

- arkitekturantimønstre,
- udviklingsantimønstre,
- genbrugantimønstre,
- ydelseantimønstre og
- ledelsesantimønstre.

I denne opgave vil jeg gennemgå disse forskellige kategorier og for hver kategori beskriver jeg en række antimønstre. Hvis man både skulle nævne og beskrive alle kendte antimønstre, vil det gøre opgaven til en lang punktopstilling. I stedet vil jeg beskrive hver kategori overordnet. Derudover vil jeg redegøre et par af de mest interessante antimønstre i hver kategori.

2. Arkitekturantimønstre

Arkitekturantimønstre sætter fokus på produktet som helhed. Det er vigtigt at forstå, at den objektorienterede model kun beskriver en del af et helt system, og er del af den samlede arkitektur for et softwareprodukt. Ifølge [6] defineres softwarearktitektur således: "all design and implementation aspects, including hardware and technology selection".

Følgende arkitekturantimønstre bliver beskrevet på [6]:

- *autogenerated stovepipe*
- *stovepipe enterprise*
- *jumble*

Skrevet marts 2008. Revideret maj 2008.

- *stovepipe system*
- *cover your assets*
- *vendor lock-in*
- *wolf ticket*
- *architecture by implication*
- *design by committee*
- *warm bodies*
- *Swiss-army knife*
- *reinvent the wheel*
- *the grand old Duke of York.*

Vendor lock-in: Et softwareprojekt integrerer en anden udbyders produkter. Integreret forstået på den måde at dele af eksekveringen i den anden software sker via en webservice, eller at man gør sig afhængig af, at brugere benytter en bestemt stykke software f.eks. en bestemt version af en browser. Beslutningen om dette kan være fornuftig for projektet for effektivitetshensyn, men der er en faldgruppe — man kan i uheldige tilfælde gøre sig fuldstændig afhængig af at andenparten er pålidelig.

Hvis den integrerede software opgraderes og dermed skifter funktionalitet skabes regressionsfejl i eget software. Man er ikke længere selv herre over hvornår potentielle fejl kan ske. Det værste der kan ske er at man f.eks. har outsourcet og hostet dele af et softwareprojekt til et firma, der pludselig går konkurs/brænder ned — vel og mærke uden hensyntagen til at dette kunne ske. Ens eget software er nu totalt ubrugeligt.

Man kan gardere sig imod dette antimønster ved at bygge et såkaldt isolationslag mellem ens eget software og andenpartens software. Dette kan anskues som et lag i en protokolstak. Sker der ændringer eller ophører funktionaliteten i andenpartens software fuldstændig, skal man kun rette i isolationslaget for at gøre skaden god igen.

Design by commitiee: Dette antimønster er også kendt som "for mange kokke fordærver maden". Dvs. dette er et meget generelt antimønster, siden at det også er kendt som et godt gammeldags ordsprog.

Opfattelsen af at hele udviklingsteamet skal være med til aktiviterne i forbindelse med design af applikationen er nødvændigvis ikke rigtigt. Møderne bliver lange og effektiveten voldsomt nedsat og derfor går det udover kvaliteten af designet. Det er også farligt at lade alle få hver deres lille forslag gennemført, blot fordi de skal føle at de har haft medbestemmelse. Arkitekturen kan blive inkonsistent, upassende eller direkte fejlbehæftet. Parlamentariske tilstande som i et demokrati, hvor beslutninger foretages ved flertalsafgørelse er potentielt farligt når der skal designes en arkitektur. En arkitektur skal være strømlignet ikke et kompromis.

Er man løbet ind i dette antimønster, gælder det om at optimere ens møder. I artiklen [6] foreslås blandt andet følgende: Placer et synligt ur på væggen. Alle skal vide hvorfor de er til stede og hvad målet med mødet er. Roller skal fordeles blandt deltagerne, her kan drages paralleller til *code reviews*. Der skal være en mødeleder og der bør være fem eller færre tilstede.

Man bør også være klar over hvem der har det sidste ord, og at den person rent faktisk også får det sidste ord. Disse råd kan få møde/design processen til at være mere effektiv og dermed give et bedre resultat.
Warm bodies: En ud af 20 programører besidder et særligt talent, der gør ham/hende mange gange så effektive som den middelmådige programmør. Hvis et firma hyrer en programmør på timebasis, er det for at tjene penge på at have vedkommende siddende pr. time. Så jo flere programmører jo mere profit. Og jo flere programmører jo større kravspecifikation, og igen kan man jo så hyre flere programmører.

Dette antimønster kommer til udtryk ved at, når udviklingsholdet har vokset sig over en bestemt størelse bliver det ineffektivt. Denne ineffektivitet opstår på grund af kommunikationsomkostninger. Dette giver flere varme kroppe der sidder og producerer i højere grad ingenting, og dette medfører mindre profit for virksomheden. Og som kravspecifikationen vokser, er det sværere og sværere for holdet at få produceret et færdigt produkt.

Løsningen på dette er at holde softwareholdende små (fire personer). Er det ikke muligt at nå et færdigt produkt på til en specifik deadline, må flere parallelt arbejdende udviklerhold tages i brug.

3. Udviklingsantimønstre

Udviklingsantimønstre omhandler de mønstre udviklerne skal forsøge at undgå i deres arbejde. Udviklingsantimønstre kan deles op i to grupper: Dem har med dårligt objektorienteret design at gøre:

- *the blob*
- *ambiguous viewpoint*
- *functional decomposition*
- *poltergeist*

og dem der ikke har:

- *lava flow*
- *boat anchor*
- *golden hammer*
- *dead end*
- *input klundge*
- *spaghetti code*
- *continuous obsolescence*
- *mushroom management*
- *walking through a minefield*
- *cut-and-paste programming*

Igen er det for omstændeligt at gennemgå alle de ovennævnte antimønstre, her henvises til [6] for detaljer om resten. Jeg vil gennemgå tre af disse antimønstre: *golden hammer* fordi det er det mest almindelige antimønster i industrien ifølge [6], *spaghetti code* fordi det er ældste ifølge [6] og *the blob*, da jeg finder dette antimønster interessant.

Golden hammer: Antag at i en virksomhed udviklerne kender indgående til en måde at løse tingene på. Som resultat af dette bruges denne metode/værktøj på alle problemer der skal løses. Årsagerne til dette er blandt andet, at udviklere har været gennem omfattende træning inden for et snævert område, eller udviklerne har været isoleret fra industrien. *Golden hammer* resulterer derfor i det modsatte af at bruge det rette værktøj til det rette formål. Dette udmunder i dårlige løsninger, samt at udviklerne på holdet ikke udvider deres horisont mht. metoder/værktøjer som kan anvendes. Løsningen på dette antimønster er selvfølgelig omfattende. Udviklerne må lære nye værktøjer at kende og sørge for at de ikke på ny opfinder en gylden hammer. Dette skal ske ved at ledelsen opfordrer dem til det samt at ledelsen sætter resurser af til, at udviklerne kan afholde workshopper, hvor de sammen kigger efter nye løsningsmetoder/værktøjer.

Spaghetti code: Dette antimønster er måske det ældste af alle antimønstre inden for softwareudvikling. Hvis en programmør i ringe grad strukturerer sit program, opstår det, som før antimønstre kom til blev kaldt for spaghettikode. Dette kan ske hvis programmøren ikke kan finde ud af at strukturere sit program, eller den mere typiske hvor en hurtig prototype, uden struktur, udvikler sig til et større stykke software som andre komponenter/kunder osv. er afhængige af. Ikke afholdte eller dårlige *code-reviews* kan også være en årsag til spaghettikode. Bliver der brugt et objektorienteret sprog, er der typisk få objekter med forkert ansvar.

Konsekvenserne af dette antimønster er mange. Koden bliver gradvist sværere at vedligeholde, fejl opstår nemmere, koden kan ikke genbruges, det er måske kun en udvikler der forstår, hvad der foregår, eller programmet bliver ineffektivt.

Når man står med spaghettikode, er en *refactoring* eller en komplet omskrivning en løsning. Hvor ramt en applikation er, er selvfølgelig forskelligt og skal tages med i overvejelserne. Hvis en del af koden er kludret og ikke struktureret, mens andre dele kan være fint struktureret, kan man overveje en *refactoring*, og hvis den udvikler der konstruerede spaghettidelen stadig er ansat, taler det yderligere for en *refactoring*, da spaghettikode er svær at læse af andre end forfatteren.

The blob: Dette antimønster, også kendt som *god class*, drejer sig udelukkende om objektorienteret design. Antimønstret opstår, når størstedelen eller hele funktionaliteten i en applikation ligger i en klasse og resten af klasserne kun opbevarer data. Flere blobs kan optræde i en komponent, hvis den er stor nok. Hvis en klasse har flere end 60 [6] attributter og metoder, er det ifølge [6] en potentiel *blob*.

Årsagerne kan være dårligt design eller at prototypen bliver til en faktisk applikation. *The blob* kan også stamme fra en specifikation som har fremprovokeret det.

En blob klasse er ikke nem at teste og genbruge. Dette antimønster er ikke lige så kritisk som spaghettikode, da koden sagtens kan være velstruktureret inde i objekterne, men bare er fejlagtigt placeret.

For at komme af med et *blob*-objekt kan man omdesigne sig ud af situationen, ved at gruppere de forskellige metoder/attributter i objektet og genplacere dem korrekt. Omdesign af *blob*-objektet indbefatter evt. omdesign af resten af applikationen.

4. Genbrugantimønstre

Med softwaregenbrug menes der at programmøren genbruger dele af kildekoden til et nyt stykke software. Genbrug af kode anbefales generelt i form af moduler og komponenter, som programmører kan give videre til hinanden — CPH STL programbiblioteket er et eksempel på dette. Udover komponenter kan software til genbrug være: eksekverbar kode, design, krav, test cases, templates eller andet udviklingsarbejde gemt i filer. Der er en række måder på hvorledes software genbrug kan udføres dårligt [3]:

- *field of dreams*
- *garbage dump*
- *abracadabra*
- *high noon*
- *used car fiasco*
- *hunter-gatherer*
- *one size fits all*
- *domain-analysis paralysis*
- *object explosion*
- *of course it's reusable!*
- *the cost-cutter.*

Garbage dump: Antimønstret opstår, når alt der muligvis kan genbruges lægges samme sted på filsystemet. Resultatet af dette vil være, at det hurtigt bliver umuligt at finde det man måske kan bruge, og tit skal spørge andre hvor på repositoriet noget er placeret. Sker dette bliver intet genbrugt.
Abracadabra: Dette antimønster sker når ledelsen af et projekt føler sig frustreret over at intet genbrug bliver foretaget. Ledelsen beder medarbejderne om at benytte sig af genbrug, uden at nogle egentlige resurser bliver afsat til at opstarte softwaregenbrug. Resultatet bliver igen at intet bliver genbrugt.
Object explosion: Antimønstret sker når objektorienteret design bliver brugt som grundlag for genbrug på en uhensigtsmæssig måde. Moralen er, at man ikke kan genbruge klasser, man bliver nødt til at genbruge hele komponenter, sagt på en anden måde - genbrug skal introduceres på det rigtige niveau, ellers bliver det ineffektivt eller direkte ubrugeligt.

I artiklen [3] understreges, at softwaregenbrugantimønstre er et alvorligt problem, men for hvert antimønster er der en række metoder, der kan benyttes når man er stødt ind i en genbrugs antimønster. Eksempelvis kan man ved et *garbage dump* benytte del-og-hersk-princippet for hermed at rydde op.

5. Ydelseantimønstre

Ydelseantimønstre beskrives i [4] og [5]. Disse mønstre opstår i klassiske enkelttråede objektorienterede programmer, distribuerede systemer og databasesystemer. En programmør, der har kendskab til denne kategori af antimønstre, kan have nemmere ved at idenficere ydelsesproblemer på det rigtige niveau.

I de to artikler beskrives forskellige antimønstre med følgende skema:

- et navn
- en beskrivelse
- et matematisk udtryk, der beskriver en gevinst ved en evt. *refactoring*/løsning.
- benchmarkgrafer.
- forslag til løsning.

I [5, s. 7] ses en komplet liste over anderkendte og dokumenterede ydelseantimønstre med deres navn, typiske problem og en mulig løsning. De femten antimønstre er:

- *falling dominoes*
- *empty semi-trucks*
- *roundtripping*
- *tower of Babel*
- *unbalanced processing*
- *unnecessary processing*
- *the ramp*
- *Sisyphus database retrieval performance*
- *more is less*
- *god class*
- *excessive dynamic allocation*
- *circuitous treasure hunt*
- *one-lane bridge*
- *traffic jam.*

At beskrive dem alle her er for omstændigt, men to af de ovennævnte vakte min interesse og beskrives her. For flere beskrivelser af performance antimønstre henvises til [4] og [5].

Tower of Babel: Dette antimønster påpeger at hvis to processer ikke taler det samme sprog, sker der et voldsomt tab i ydelse, da der bruges tid på at oversætte fra et format til et anden format af data (såsom XML). Parsing kan hurtigt tage en stor del af den reelle brugte CPU-tid for en proces. Selvfølgelig kan det være nødvændigt at oversætte fra eksempelvis XML i nogle situationer hvor en webservice kræver det, men tit er det tilfældet, at kun meget simpelt information overføres og et meget simpelt udvekslingsformat kunne have været brugt. At skifte enten det interne eller eksterne sprog til udveksling af data kan afhjælpe dette problem.

Excessive dynamic allocation: Antimønstret, som er relevant for CPH STL-projektet, går ud på at minimere brugen af allokering og deallokering af

dynamisk hukommelse. Anvendes der et sprog med spildopsamling, som også er dyrt, skal dette minimeres. Anvendes en bestemt type objekt mange gange i træk (til forskellig brug), kan en løsning være at sørge for, at en klasse er genbrugelig — dvs. at et objekt ikke skal nedlægges og oprettes igen før samme funktionalitet genbruges. En avanceret udgave af denne model kunne være at have en pulje af objekter preinitialiseret inden en programbid startes. Her kan ubrugte objekter ligge, hentes, når de skal bruges, og returneres igen, når de ikke bliver brugt.

6. Ledelsesantimønstre

Ledelsesantimønstre går ud på fejl der gentager sig hos ledelsen af et softwareprojekt. Et eksempel på dette er at bruge e-mail til at give de forkerte type beskeder, f.eks. kan man nemmere få sagt ting på den rigtige måde ansigt til ansigt.

De nævnte antimønstre i projektledelse er [6]:

- *blowhard jamboree*
- *analysis paralysis*
- *viewgraph engineering*
- *death by planning*
- *corncob*
- *fear of success*
- *intellectual violence*
- *irrational management*
- *smoke and mirrors*
- *project mismanagement*
- *throw it over the wall*
- *fire drill*
- *the feud*
- *e-mail is dangerous.*

Fear of succes: Dette måske sære, men interessante, antimønster finder sted i afslutningsfasen af et softwareprojekt — når produktet er ved at være færdigt. Programørerne begynder at bekymre sig ubegrundet over hvad der kan gå galt i afslutningsfasen. Frygten kan sprede sig fra en programmør til hele holdet, og dermed sende negative signaler til omverdenen omkring projektets tilstand. Frygten kan måske begrundes i at når projektet afsluttes, er de enkelte programmørers fremtid efter projektets afslutning måske usikker. Ubegrundet frygt kan få folk til at gøre uventede ting som kan gå hen og skade projektet.

Hvis projektledelsen får fornemmelsen af, eller har opdaget at et *fear of succes*-antimønster er opstået, kan ledelsen gå ind og erklære succes, simpelhen ved at fortælle udviklerne at projektet er succes og meget snart færdigt. Hvis ledelsen overtaler udviklerne til at der ikke er grund til frygt, forsvinder *fear of succes*.

Intellectual violence: Dette antimønster kommer i to udgaver: en mild og en svær form. Begge opstår når en part forstår den teori der ligger bag ved et buzzword/teknologi/koncept, og den anden part ikke gør. Når dette buzzword bruges gang på gang af parten, der forstår det, går kommunikationen i stå. En forstærkerne effekt kan være at udviklere ikke har lyst til at indrømme at de er udvidende omkring et koncept. Dette kan have en kraftig negativ effekt på effektiviteten.

Den mere alvorligere form for *intellectual violence* går på at folk med vilje hemmeligholder information. Et motiv bag at en ansat ønsker sig at sætte sig i en magtposition over for en anden ansat. Dette kan i værste fald skabe destruktiv stemning på udviklerholdet, og kan have fatale konsekvenser for produktionen af software.

Løsningen på dette er at man fra ledelsens side sørger for, at medarbejdere og ledere lærer hinanden ting, uanset hierarkisk position, talent og baggrund.
Cornbob : Dette antimønster beskriver en situation hvor en besværlig person er tilstede på udviklerholdet. Personen modsætter sig konsekvent forslag og beslutninger uanset om de er fornuftige eller ej. Grunden til dette er at personen har en skjult dagsorden, der er i konflikt med udviklingsholdets mål. Typisk er sådan en person god til at få trumfet sine ting igennem over for ledelsen, fordi ledelsen typisk lytter til dem der råber højest og længst. Meget teknisk-orienterede medarbejdere kan især lide under dette.

For at løse dette antimønster skal ledelsen først og fremmest være klar over at problemet eksisterer. Derefter kan ledelsen gå ind og løse problemet (reorganisering).

7. Konklusion

Ser man generelt på det gennemgåede litteratur og tilgængelige resurser samt oversigter over antimønstre tilgængelige på internettet, er det svært at danne sig en mening om hvilke kategorier indenfor antimønstre, man bør anerkende. Ydermere ser det tit ud som om at kategorierne overlapper og forskellige antimønstre er fejlplaceret. Eksempelvis *mushroom management* er kategoriseret som en udviklingsantimønster i [6]. En del af forklaringen kan være at feltet er forholdsvis nyt inden for softwareudvikling.

Overalt i datalogi findes gentagne mønstre hvorpå man kommer fra et problem til en løsning. Så inden for datalogi kunne der nemt i alle grene eksistere antimønstre: HCI, teori og praksis i programmeringssprog, billedbehandling, algoritmik og distribuerede systemer, for at nævne de forskellige forskningsgrupper på DIKU. Perspektiverer man videre til andre videnskaber og håndværk, er det muligt at antimønstre kan optræde alle steder hvor der udføres et stykke arbejde.

Litteratur

[1] Alistair Cockburn og andre, Anti pattern, Webdokument tilgængelig fra `http://c2.com/cgi/wiki?AntiPattern`.

[2] Erich Gamma og andre, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
[3] John Long, Software reuse anti-patterns, *Software Engineering Notes* **26**, 4 (2001).
[4] Connie U. Smith og Lloyd G. Williams, Software Performance Anti-Pattern (2000).
[5] Connie U. Smith og Lloyd G. Williams, More new software performance anti-patterns: Even more ways to shoot yourself in the foot, *Performance Engineering Services and Software Engineering Research* (2003).
[6] SourceMaking, Antipatterns, Webdokument tilgængelig fra `http://sourcemaking.com/antipatterns`.

Extreme programming

Bo Simonsen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
bosim@diku.dk

Abstract. Most people who have studied software development knows about the waterfall model, which describes the development process, as a one-way process, where analysis for the entire system is done before programming it. By developing in that way, it can take a long time before you have the product (the system) ready, and defects found late in the development process are very expensive to correct according to [5] (Figure 3 in the appendix).

Another approach is to develop small parts of the system, and at last you have a complete system, i.e. divide the task of building a complete system into smaller tasks. We call these methods *Agile*, and an example is eXtreme Programming (XP), which we will study in this survey. This survey is mainly based on the book Extreme Programming Installed [6].

1. Concepts

We know the concepts from the waterfall model, where development is done in phases: first we do analysis, then we do design, and so on. After each phase we get an output, for the analysis phase we get analysis documentation, for the programming phase we get code.

XP is different. The customer writes stories, where each story is a programming task, and then the developers do minimal documentation (e.g. a few UML diagrams), and at last they write the unit test and implementation. The circle of life in an XP project is shown in Figure 1.

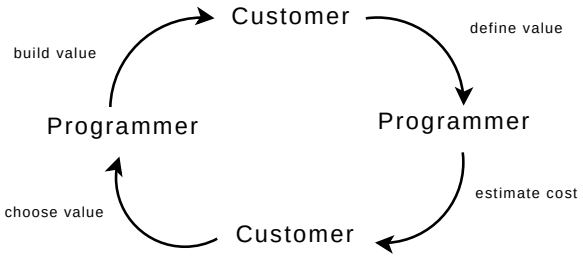

Figure 1. The XP circle of life [6]

Written April 2008. Revised June 2008

The most important *process phase* concepts in XP are:

- **Stories** define a programming task. The stories are written by the customer and each story should increase the business value of the project. This means the customer controls which part of the upcoming system should be implemented, and in which order.
- **Iterations** are used to split up the process of developing the entire project into smaller parts. Each iterator takes 2–3 weeks, and is based on a set of stories. At the beginning of each iteration, an *iteration planning meeting* is held, where the customer presents the stories, and the programmers sign up for them.
- **Releases** are defined by the customer, and should be small. A *release planning meeting* is held, where the stories are considered. All stories should be ready at the beginning of this meeting, and they should define a successful product. At this meeting the estimation and selection of stories is done.

The relationship between these concepts is shown in Figure 4.

XP defines more than just process phases, it also defines some guidelines of how the project team should work:

- **Acceptance test** should be written for each story, and it should be available at the beginning of each iteration. It seems like a lot of testing, but the book [6] recommends that the programmers should use *automated unit testing*. This makes the job easier for the programmers, since they are sure that the code works in a consistent way (most programmers know that changing one part of the program may result in defects in other parts of the program). The acceptance test is done to ensure that the program works in the way the customer wants it.
- **Pair programming** means that programming should be done by two programmers, one programmer should type the code, and the other programmer should observe. In this way potential defects will be corrected (4 eyes see more than 2), and the coding standard is obeyed.
- **Estimation** is done by evaluating the project's history. That is, when the programmers estimate the cost of a story, they will take previous stories into account. Project managers normally use techniques like *function points* [5] for estimation, but this mathematical model is seldom used in XP.

2. Roles

Unlike in other development-process models, the customer is an active part of the development team. This means that he/she should be on-site, and he/she should do some preparation work e.g. writing stories.

2.1 The customer

In the book [6], the rights of the customer are described (Appendix A.1). These rights give the key idea of the customer's position in the development

team; the main points of these rights are: The customer has the right to *control the project*, which means he has the right to have a plan, and get most out of every programming week. He can also change his mind about the system functionality at any time. The customer has the right to get *insight* knowledge of the system, meaning that at any time he can see that the system works.

2.2 The programmer

Like the rights for the customer, there are also rights for the programmers (Appendix A.2). The main idea is that the programmers should communicate with peers (also the customer), and they should produce quality work. Because of that the programmers also have the right to estimate, based on how long they guess it will take them to do quality work.

3. Development

After discussion the concepts of XP and the relationship between the customer and the programmers, we need to know how development is done in XP. Figure 2 gives an overview of how the process works, but in the following sections, we give a more detailed description.

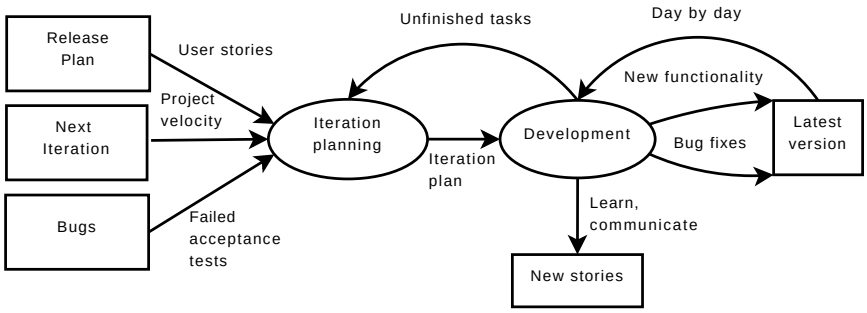

Figure 2. The big picture [1]

3.1 Stories

The stories are the core in XP development, all stories together should define the entire system. The story is written by the customer and it is written on small peaces of paper, we call these peaces for *story cards*. The size of each story card should be kept small, such that the contents of each story is small, since the story defines one programming task.

A sample story is shown below.

Example 1. *Allow the user to attach files, and add them to the composing window such that the user can see how many files there are attached.*

The stories seem a lot like *use cases*, but they do not describe how a user interacts with a system like the *use cases* do; they describe programming assignments.

It may be hard to write stories for a big system. The book [6] recommends that you should try writing stories, and if the story does not fit (e.g. no more space on the story card, or it seems too complicated as a single task), you should tie it up, and split it into two or more stories.

You continue writing stories until the whole system is covered by the stories. During development new stories can be written if some functionality is desired or something is simply missing.

It is often useful to provide fields on the story cards. These fields should be used for estimation and other relevant data. In Figure 5 a sample story from the Xplanner tool is shown. On that story card you can see that fields *priority* and *estimated hours* are present.

3.2 Acceptance test

In the previous section about the customer role, we saw that the customer has certain rights. One of these rights is that the customer should be able to see a running system at any time. To ensure that we need a test for every story, we call this test an *acceptance test*. To ensure consistency, automated tests are recommended. The customer may provide possible input values for these tests, but they are made by the programmer.

3.3 Release planning

The customer has the right to have an overall plan for the entire project, because of that we need to have some kind of planning. We do planning for iterations and releases. The iteration planning is short term planning which is normally a few weeks, and release planning is long term planning, about 5–6 iterations.

The release planning is done before we begin development. The stories should be ready at that time, else you have nothing to base the plan on. A release plan recipe could be:

- Make sure that you have enough stories for a successful product.
- Do necessary exploration.
- Estimate the difficulty of implementing each story.
- Estimate the speed of story implementation.
- Choose stories for the first release, and set a release date.

The planning takes place at a meeting where both the customer and the programmers are present. The customer selects stories, and the programmers estimate the costs. After this meeting we are ready for our first iteration. We know how many iterations there are, because we have set a release date, and an iteration is about two weeks.

3.4 Iteration planning

Iteration planning is similar to release planning. But in iteration planning we will focus on the iteration, especially how and by whom the programming is done. The iteration planning is done at a meeting like the release planning, but the planning is done on a lower level.

The main points in the iteration planning meeting are:

- The customer presents user stories.
- The team brainstorms engineering tasks.
- The programmer signs up for and estimates the work.

We will now look into details of every activity in the meeting.

3.4.1 Story presentation

The customer will present the story for the programmers. The programmers will ask questions if something is unclear.

3.4.2 Brainstorm

When the programmers understand the stories, they do a quick brainstorm, which should result in a design on, how the stories should be implemented. They do it together so they are sure that everybody understands the overall system design. Moreover, that the customer must be sure of that the programmers understood the stories correctly. At this step the stories are broken down into *programming tasks*.

3.4.3 Signing up

After the brainstorming is done, the programmers sign up for a programming task or an entire story. The programmers should have a feeling of how much work they can accomplish within the iteration. After they have signed up for the tasks they estimate the cost of each task. Remember that most programming is done in pairs, so each task must be assigned to two programmers.

3.5 Programming

Extreme Programming is of course about programming. What we have seen so far is how to get the project running, i.e. how to get the system defined and planned. In the following sections we look into details how the programming is done.

3.5.1 Pair programming

In pair programming we have a so-called *driver* and a *partner*. The driver types at the keyboard and the partner is watching for mistakes. This may sound strange to some people that programming should be done by two

persons, where only one of them is typing at the keyboard. To point out some convincing benefits:

- Syntax/design/etc. mistakes are faster discovered and corrected.
- The two persons learn from each other.
- The code is as simple as possible (if it is not the partner should notify the driver about it).

In XP teams, most people work with the same partner, but it should not be a problem to change partner once in a while, even for just a task.

3.5.2 Collective code ownership

Collective code ownership means that every team member has equal rights to change some code even if it is written by somebody else. In other words, if a team member observes a design mistake or simply some bad written code, he should just change it, without asking permission from anybody. One common problem is that people think they own the code they have written, and nobody should change it without asking permissions. Even if people ask permissions, it can still be hard for the original authors of some code, to see there is a problem with their code, especially if it is about readability. It is much easier to correct mistakes when everybody owns all the code collectively.

3.5.3 Simple design

The code should be as simple as possible. A recipe for checking that your code is as simple as possible is:

1. Run all tests. If the tests fails, you made too much refactoring, such that your code is defect.
2. Express every idea you need to express to make the code simple. Write the code as compactly as possible, but still keep it readable. This idea is shown in the following examples, where the first version of the code is shown in Example 2 and an improved version in Example 3. After the change the code is simpler, shorter, and more readable.
3. Remove all duplicate code so that the code has the minimum number of classes and methods.

Example 2.

```
void process_array(Account* array, int num) {
 int i = 0;
 Account* a;

 while(i < num) {
   a = array + i;
   a->amount -= 20;
   i++;
 }
}
```

Example 3.

```
void process_array(Account array[], int num) {
  for(int i = 0; i < num; ++i) {
    array[i].amount -= 20;
  }
}
```

3.5.4 Continuous integration

Continuous integration means that you integrate all changes made to the code, just after making the changes. You should, of course, run the acceptance test before integrating the changes. This means that your code base is consistent such that every change made is in the shared code base.

For people, who have used a version-control system, this is the normal way to work. They do some changes and commit these changes after doing some testing; so for them there is no news in this method. But people, who have not used a version-control system, tend to delay integration of the code and tests. By doing that, it will just get harder to integrate the code, and it will also be hard to get an overview of the system's current state.

3.5.5 Coding standard

Since every team member in XP shares the code, the code needs to be consistent regarding to style. So we need to define a *coding standard* so the code looks identical even that it is written by different project members.

The most essential parts of the coding standard are:

- Indentation: Tabs, bracket placement, etc.
- Commenting: Recommended that the commenting should be sparse — only comment if it is needed. This means that you should not make trivial comments (e.g. traversing an array). You should only do it, if it helps the understanding of a complicated/non-trivial procedure.
- Method size: Recommended that methods should be kept small.
- Names and capitalization: Make it clear how variable and function names should be written, and where to capitalize.

4. Tools

While writing this survey I found some useful tools for XP development:

- **XPlanner [2]** is a complete tool for managing an XP development process. It can handle elementary things like releases, iterations, and stories. But it does also provide statistics and metrics, which can help the programmers to estimate the costs. See Figures 5 and 6.
- **Automated test frameworks** are an essential part of writing tests in XP. In the book [6], they describe how to use the XUnit framework which is used for Smalltalk code. But for every language there is an automated test framework. For C++ there is CppUnit and UnitTest++

and for Java there is JUnit. On Wikipedia there is a list of testing frameworks [7].
– To do continuous integration, it would be a good idea to use a **version-control system** like CVS or Subversion.

5. Concluding remarks

Here are some remarks/thoughts which are not covered by the book.

5.1 The new XP

The survey is based on a book which was published in 2001, but in 2005 Kent Beck publish a revised edition of his book "Extreme Programming Explained" which contains a refined description of XP. The principles are almost the same, but they have different names, and key values are more precisely described.

The article "The New XP" [4] describes the new XP model and compares it with the previous practice.

5.2 Thoughts on XP in practice

As described, in XP the customer is an important member of the development team (unlike in other development process models). This is because it is the customer who pays for the final product. But can development actually be done with high customer involvement?

One of the authors of the book [6] has written an article "Will Extreme Programming kill your customer?" [3]. This article describes how the customer got stress during the development process. The author of the article asks if development can be done and still have a healthy customer, but he does not conclude anything. It seems to be nessersary that the customer is trained to work in the XP style.

References

[1] Don Wells, Extreme programming: A gentle introduction, Website accessible at `http://www.extremeprogramming.org`.
[2] Xplanner Project, Xplanner, Website accessible at `http://www.xplanner.org`.
[3] C. Hendrickson, Will extreme programming kill your customer?, Worldwide Web Document. Available at `http://www.coldewey.com/publikationen/conferences/oopsla2001/agileWorkshop/hendrickson.html`.
[4] M. Marchesi, The new xp, Worldwide Web Document. Available at `http://www.scribd.com/doc/196734/The-New-XP`.
[5] R. S. Pressman, *Software Engineering — A Practitioner's Approach*, McGraw-Hill (2000).
[6] C. H. Ron Jeffries, Ann Anderson, *Exteme Programming Installed*, Addison-Wesley (2001).
[7] Wikipedia, List of automated testing frameworks, Worldwide Web Document. Available at `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`.

Appendix A. Rights

Appendix A.1 Manager and customer rights

1. You have the right to an overall plan, to know what can be accomplished, when, and at what cost.
2. You have the right to get the most value out of every programming week.
3. You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify.
4. You have the right to change your mind, to substitute functionality, and to change priorities without paying exorbitant costs.
5. You have the right to be informed of schedule changes in time to choose how to reduce scope to restore the original date. You can cancel the project at any time and be left with a useful working system reflecting investment to date.

Appendix A.2 Programmer rights

1. You have the right to know what is needed, with clear declarations of priority.
2. You have the right to produce quality work at all times.
3. You have the right to ask for and receive help from peers, superiors, and customers.
4. You have the right to make and update your own estimates.
5. You have the right to accept your responsibilities instead of having them assigned to you.

Appendix B. Figures

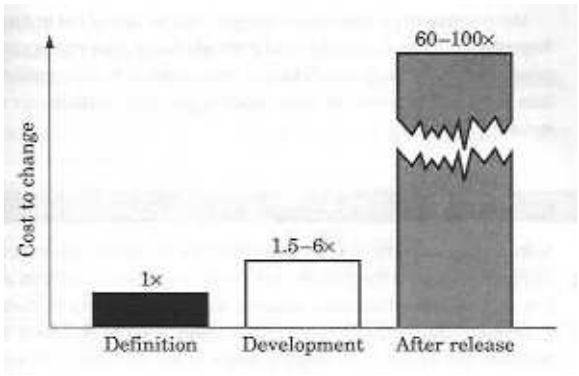

Figure 3. The cost of correcting defects in different phases [5]

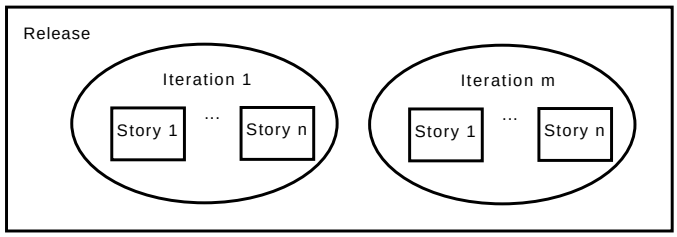

Figure 4. Relationship between release, iteration, and story

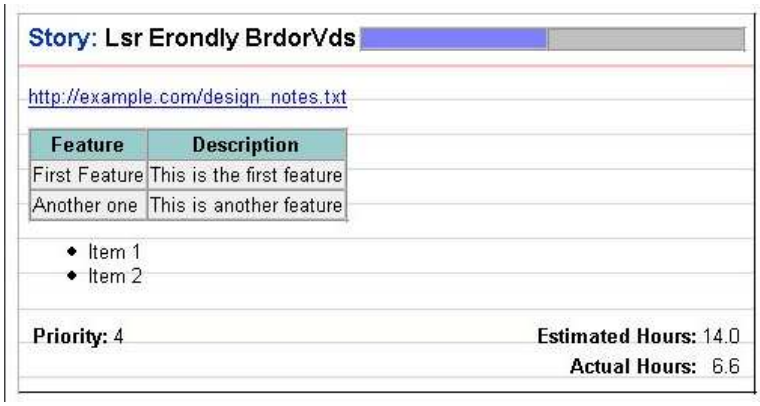

Figure 5. An example of the story card in Xplanner [2]

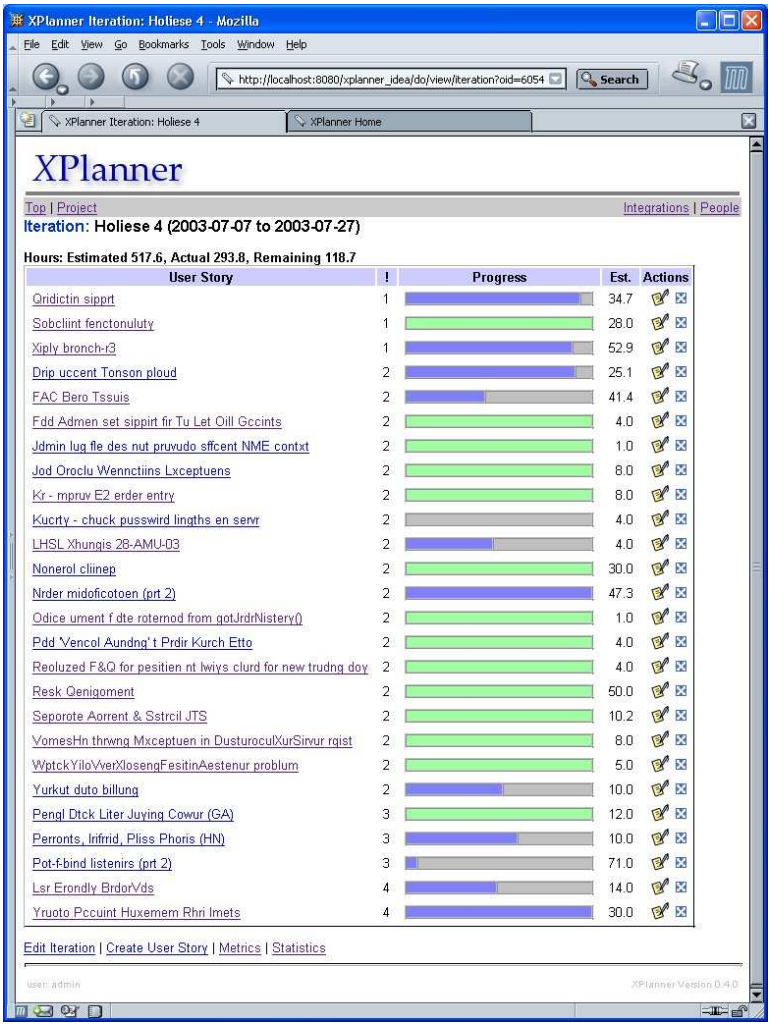

Figure 6. The iteration overview in Xplanner [2]

Author index