Lunds universitet
Historiska institutionen
HISA22 (uppsatskurs, 7,5 hp)
Seminarieledare: Andrés Brink Pinto
10-12 januari 2012

Så tuktas en programmerare

Taylorismens inflytande i modern programvaruutveckling

Patrik Persson

Innehåll

1 Inledning

Vi lever bland datorprogram. De styr tåget du åker i, de spelar musiken du lyssnar på, de hjälper dig att stava, de sätter dig i kontakt med den du älskar. De flesta datorprogram möter vi utan att se dem som sådana; det är helt enkelt de som får saker att fungera – och ibland att sluta fungera, förstås.

En allt större del av vår industri är sysselsatt med att utveckla datorprogram – ofta kollektivt kallade programvara, som om de vore något homogent som kunde säljas efter kilopris. Det påstås att 75 000 svenskar arbetar med programvaruutveckling,[1] som programmerare, projektledare, testare eller något annat, och dessa människor återfinns såväl i de etablerade stora industriföretagen som i små nystartade bolag. Jag är själv en av de 75 000 och har arbetat med olika aspekter av programvaruutveckling i ungefär 15 år, som såväl projektledare som programmerare.

1.1 Syfte och frågeställning

Programvaruutveckling är svårt, inte minst när många människor är inblandade. Utvecklingen bedrivs ofta i projektform, där en grupp människor ska prestera ett gemensamt resultat. Metoder för att organisera sådana programvaruprojekt är av stort intresse, och en kategori av sådana metoder kallas på engelska för *software engineering*.[2] Min erfarenhet är emellertid att dessa metoder inte alltid är så populära bland programmerare: ibland påstås att nutidens software engineering – åtminstone till delar – är föga bättre än det tidiga 1900-talets taylorism, och urholkar den hantverksmässiga sidan av programmerarens roll.

Denna uppsats undersöker dessa påstådda likheter mellan taylorism och software engineering närmare. Närmare bestämt är uppsatsens syfte att undersöka i vilken utsträckning den kritik, som riktats mot taylorismen, också kan riktas mot software engineering. Jämförelsen kommer här att göras utifrån två normer, formulerade av Frederick Taylor respektive Tom DeMarco och närmare presenterade i avsnitt 2.1. Frågan är: *vilka relevanta likheter och olikheter finns mellan Taylors och DeMarcos normer?*

1.2 Software engineering: en kort bakgrund

Inför fortsättningen är en kort bakgrund till software engineering på sin plats. Begreppet kom i allmänt bruk i samband med en NATO-konferens 1968, organiserad för att hantera problemen med stora programvaruprojekt. Projekten resulterade allt oftare i programvara som var

[1] Swedsoft, *Mjukvaran är själen i svensk industri*, http://www.swedsoft.se/Mjukvaran_är_själen_i_svensk_industri.pdf, 2008, hämtad 28 december 2011.

[2] Termen syftar på ingenjörsmässiga metoder för programvaruutveckling; den närmsta svenska termen är ”programvaruteknik”, men den tillmäts ofta en något bredare betydelse och undviks därför här.

försenad, otillförlitlig eller löste fel problem.[3] I rapporten omnämns programmering som produktion (*production, manufacturing, maintenance*), en tydlig vink om hur man nu ville foga in programmering inom de klassiska ingenjörsdisciplinerna.

Ett tidigt försök att möta dessa krav på projektplanering var Winston Royces vattenfallsprocess,[4] som kännetecknas av en linjär sekvens av välavgränsade, specialiserade faser, som design, implementation och testning. Trots att Royce själv problematiserade processen tämligen grundligt i sin artikel kom den den ändå att dominera under lång tid framöver. Ännu i 1990-talets läroböcker i software engineering beskrevs vattenfallsprocessen som en utgångspunkt, om än en förenklad sådan, för programvaruprojekt.[5] DeMarcos *Controlling Software Projects* från 1982 är ett standardverk inom denna software engineering-tradition och refereras ofta.[6]

Kring millenieskiftet kom en annan typ av metoder i bruk. Dessa s.k. agile-metoder fokuserar mindre på formaliserade processer och mer på kontinuerlig målstyrning.[7] De utgör på så vis en reaktion på det sena 1900-talets software engineering.[8]

1.3 Forskningsläge

Harry Braverman utgår från verkstadsindustrin och skriver om den specialisering i arbetsuppgifterna som den tekniska utvecklingen fört med sig. Specialiseringen medför att arbetet splittras upp i beståndsdelar som kan utföras av olika individer. Resultatet är inte bara högre produktivitet, utan även lägre krav på den enskilde arbetaren: istället för att som tidigare bemästra hela arbetsprocessen, räcker det nu att klara av ett enskilt arbetsmoment. När arbetsledningen därmed tar kontroll över arbetsprocessen uppstår en förskjutning i maktbalansen på arbetsmarknaden, från arbetaren till arbetsgivaren, liksom en utarmning av arbetsuppgifterna, en *arbetets degradering*. Med Taylor som källa argumenterar Braverman för att det kapitalistiska systemet oundvikligen leder till sådan degradering, och kritiserar taylorismen som det främsta exemplet på denna utveckling.[9] Paula Mulinari visar emellertid hur Braverman har kritiserats för detta deterministiska synsätt; Klas Åmark menar att degraderingen inte alls är så oundviklig som

[3] Brian Randell och Peter Naur (red.), *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Scientific Affairs Division, NATO, 1968.

[4] Winston Royce, ”Managing the Development of Large Software Systems”, *Proc. IEEE WESCON*, 1970.

[5] Se t. ex. Ian Sommerville, *Software Engineering*, femte utgåvan, Addison-Wesley, Wokingham, 1996, s. 9–11; Watts S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, MA, 1995, s. 12–13, 610.

[6] Tom DeMarco, *Controlling Software Projects: Management, Measurement, and Estimation*, Prentice Hall, Upper Saddle River, NJ, 1982; se t. ex. Humphrey, 1995, s. 147; Barry W. Boehm och Philip N. Papaccio, ”Understanding and Controlling Software Costs”, *IEEE Transactions on Software Engineering* 14(10), 1988; Everald E. Mills, ”Metrics in the software engineering curriculum”, *Annals of Software Engineering* 6, 1998.

[7] Kent Beck och Cynthia Andres, *Extreme Programming Explained: Embrace Change*, andra utgåvan, Addison-Wesley, Boston, MA, 2005; Ken Schwaber och Mike Beedle, *Agile Software Development with Scrum*, Prentice Hall, Upper Saddle River, NJ, 2002.

[8] Jag skiljer alltså här agile-metoderna från software engineering; andra gränsdragningar är möjliga.

[9] Harry Braverman, *Arbete och monopolkapital. Arbetets degradering i det tjugonde århundradet*, Rabén & Sjögren, Stockholm, 1989, s. 75–77, 81–85; om Taylor som källa, s. 87 (fotnot).

Braverman ger sken av, och menar istället att Bravermans främsta bidrag snarare är hans fokus på arbetsprocessen som maktmedel.[10]

Braverman ser en inspiratör till Taylor i matematikern Charles Babbage,[11] som även brukar betraktas som världens första datorkonstruktör. Allan Bromley menar att Babbages maskin till sin uppbyggnad visar slående likheter med dagens datorer, och frågar sig i vilken utsträckning dagens ingenjörer fastnat i Babbages tankesätt.[12] I Babbage finner vi alltså en gemensam nämnare till både taylorismen och dagens datorteknik.

Braverman visar vidare hur Taylors influenser letat sig vidare från verkstadsgolvet in på kontoren,[13] och Mats Greiff följer detta spår i sin undersökning av kontorsarbete i Malmö och Göteborg under 1900-talets första halva. Greiff visar hur tayloristiska influenser långsamt degraderade kontoristens arbete, liksom hur degraderingen gick hand i hand med en sänkt medelålder och en femininisering av arbetet.[14] Här finns både likheter och olikheter med den programvaruindustri som domineras av unga män, liksom paralleller med det utbildningssystem som ibland anklagats för att se den kvinnliga befolkningen som ”begåvningsreserv”.[15]

Michael Mahoney skriver om 1968 års NATO-konferens och visar hur den influerades av andra ingenjörsdiscipliner, som produktionsteknik och maskinteknik. Han visar vidare på likheter med taylorism i 1980-talets software engineering-forskning och ger exempel på hur Taylor uttryckligen betraktats som förebild. Hans jämförelse med Taylor är emellertid relativt ytlig: den bygger på jämförelser med korta utdrag ur dennes bok och diskuterar inte frågan om kontrollen över arbetsprocessen.[16] Mahoney studerar inte heller den litteratur som influerat yrkesverksamma ingenjörer, och därmed inte heller DeMarcos bok.

Det är inte helt ovanligt att Taylor och taylorismen omnämns i sådana böcker. Watts Humphrey, Kent Beck och Cynthia Andres nämner Taylor i mer eller mindre negativa ordalag, men frånsett Becks jämförelse med den moderna kvalitetsavdelningen så sätts sällan dessa begrepp i något egentligt sammanhang, utan får främst representera vad böckerna *inte* handlar om.[17] Dessa återkommande avståndstaganden från Taylor kan nog ses som en indikation på debatt – taylorismen spökar i bakgrunden och är något man gärna distanserar sig tydligt ifrån.

[10] Paula Mulinari, *Maktens fantasier och servicearbetets praktik*, doktorsavhandling, Linköpings universitet, 2007, s. 48; Klas Åmark, ”Genmäle till till Carl-Axel Nilsson”, *Historisk tidskrift* 116, 1996, s. 159–160.

[11] Braverman, 1989, s. 77.

[12] Allan G. Bromley, ”Charles Babbage’s Analytical Engine, 1838”, *IEEE Annals of the History of Computing* 20(4), 1998, s. 45.

[13] Braverman, 1989, s. 278–283.

[14] Mats Greiff, *Kontoristen. Från chefens högra hand till proletär*, doktorsavhandling, Lunds universitet, Mendocino, Lund, 1992, s. 265–271, 282–291.

[15] Om unga män i hackerkulturen, se Jörgen Nissen, ”Datorkulturen – en manlig historia”, Boel Berner (red.), *Vem tillhör tekniken? Kunskap och kön i teknikens värld*, Arkiv, Lund, 2003, s. 86–88; om ”begåvningsreserven”, se Boel Berner, ”Datorkulturen – en manlig historia”, samma volym, s. 119; om dataingenjörers könsfördelning, se t. ex. Minna Salminen-Karlsson, ”Hur skapas den nya teknikens skapare?”, samma volym, s. 150.

[16] Michael S. Mahoney, ”Finding a History for Software Engineering”, *IEEE Annals of the History of Computing* 26(1), 2004, s. 13–15; arbetsprocessen berörs flyktigt vid citatet från Bemer på s. 13, men diskuteras inte vidare.

[17] Humphrey, 1995, s. 15; Beck och Andres, 2005, s. 131–133.

Frederick Brooks förutsade 1987 att programmerarens produktivitet inte skulle nämnvärt det kommande decenniet. Han menar att programmerare sällan gör samma sak två gånger; när ett program väl är skrivet kan det ju upprepas i oändlighet – däri ligger ju själva poängen med datorprogram. Programmerarens arbete koncentreras på nya problem, snarare än upprepade handgrepp, och stora rationaliseringar är därför inte möjliga.[18]

2 Metod och källor

I nästa avsnitt görs en kvalitativ jämförelse mellan Taylors och DeMarcos normer, så som de formulerats i bokform 1911 respektive 1982.[19] Jämförelsen baseras på en närläsning av de båda källorna, i avsikt att fånga upp de idéer, värderingar och tillvägagångssätt de bygger på. Taylor ger oss själv en vink om hur en sådan jämförelse kan organiseras när han strukturerar sina normer kring några grundläggande principer. Braverman visar i sin tur på hur dessa principer kan kritiseras. I jämförelsen mellan de två normerna kommer jag att ta fasta på detta, och på så sätt koncentrera jämförelsen till några centrala aspekter.

En sådan aspekt är huruvida *Taylors principer* kan återfinnas i någon form hos DeMarco, och hur uttalade de i så fall är. Detta kan säga oss något om eventuella likheter i idéerna bakom dessa normer. En sådan likhet, bruket av *kvantitativa mätningar av den enskildes prestationer*, får särskilt utrymme. En annan aspekt är synen på *relationen mellan ledningen och den enskilde*. Vidare är *synen på specialisering* intressant: hur tydlig är den, vad syftar den till, vad resulterar den i? Sammantaget avser jag utifrån dessa parametrar att skapa en bild av i vilken utsträckning DeMarcos normer förorsakar den arbetets degradering som Braverman beskriver.

2.1 Källkritik: Taylor, DeMarco och deras böcker

Identifikation

Taylor var amerikansk ingenjör och organisationskonsult. Då boken gavs ut 1911 var han 55 år gammal med ett antal framgångsrika rationaliseringar bakom sig. Syftet med boken är att utbilda chefer, framförallt i industrin, och den industriella inriktningen låg rätt i tiden – boken gavs ut samtidigt som USA erövrade positionen som världens mest industrialiserade land.[20]

Tom DeMarco är född 1940, ett knappt sekel efter Taylor, och publicerade sin bok vid drygt 40 års ålder 1982. Vid det laget var software engineering en etablerad disciplin. DeMarcos far

[18]Frederick P. Brooks, ”No Silver Bullet—Essence and Accidents of Software Engineering”, *IEEE Computer* 20(4), 1987.

[19]Frederick W. Taylor, *The Principles of Scientific Management*, 1st World Library, Fairfield, IA, 2005; DeMarco, 1982.

[20]John P. McKay m. fl., *A History of World Societies*, Bedford/St. Martin’s, Boston/New York, 2009, s. 656–657.

var en stor beundrare av Frederick Taylor; DeMarco menar att detta skilde de båda åt, men understryker också hur viktig fadern varit för hans syn på arbetsledning. [21]

Genre och närhet

Taylors och DeMarcos böcker kan båda sägas tillhöra samma genre, nämligen managementlitteratur med ingenjörer som målgrupp. Detta avspeglas i böckernas informella tilltal; de utgår från att ingenjörsbakgrunden delas med läsaren och vädjar tydligt till gemensamma ideal och erfarenheter. Böckerna har båda tydligt normativ karaktär: de formulerar metoder för hur arbetsprocessen i industriproduktion respektive programvaruutveckling ska byggas upp, och de använder tydligt praktiska erfarenheter för att legitimera metoderna. De praktiska erfarenheterna ligger i det förflutna, och man kan naturligtvis misstänka att författarna tenderar att minnas framgångar framför motgångar. Braverman avfärdar vissa av Taylors framgångsberättelser som ”tvångsföreställningar”.[22] Normerna påverkas inte nödvändigtvis av detta; huvudsaken är ju att Taylor och DeMarco lyckades övertyga sina läsare.

Tendens och urval

De båda böckernas titlar säger oss något om hur normerna legitimeras. Vem kan rimligen invända mot ”vetenskaplig” – eller, i den svenska översättningen, ”rationell” – arbetsledning? På samma sätt tycks mig DeMarcos titel, som handlar om styrning och mätning, anspela inte bara på rationalitet utan även reglerteknik, ett centralt ingenjörsämne; detta framgår än tydligare av bokens ofta citerade första mening: ”you can’t control what you can’t measure”.[23] DeMarcos bok inordnar sig därtill tydligt inom disciplinen software engineering, och vilken ingenjör kan väl invända mot ingenjörsmässig programvaruutveckling? Taylors och DeMarcos retorik vädjar alltså tydligt till ingenjörens ideal.

De båda böckerna har som uttrycklig avsikt att övertyga läsaren om de beskrivna metodernas förträfflighet. Taylor och DeMarco beskriver hur fabriksarbetarna respektive programmerarna anammar de nya metoderna och inser att deras liv blir bättre, även om de kanske stretade emot från början. Dessa utsagor, tillskrivna de enskilda medarbetarna men ensidigt hågkomna, utvalda och återberättade av Taylor och DeMarco, kan naturligtvis ifrågasättas; min egen erfarenhet säger mig att vissa av DeMarcos påståenden inte är så självklara som de kan verka.

[21] Tom DeMarcos förord till Robert L. Glass, *Software Creativity 2.0*, developer.* Books, Atlanta, GA, 2005, s. xii–xiv.

[22] Braverman, 1989, s. 90, se särskilt fotnoten.

[23] ”det man inte kan mäta, kan man inte heller styra” (min övers.), DeMarco, 1982, s. 3; reglerteknik kallas på engelska *automatic control*.

Beroende

Ytterst är beroende mellan källorna av stort intresse. Även om DeMarcos bok knappast är direkt baserad på Taylors, så är ju syftet här att undersöka i vilken utsträckning Taylors idéer lever vidare inom software engineering. Visserligen kan DeMarcos relation till fadern indikera ett möjligt beroende till taylorismen, men för att verkligen förstå beroendet måste källorna och deras idéer studeras närmare. Detta görs i följande avsnitt.

3 En jämförelse

Som redan nämnts bygger Braverman sin analys av arbetets degradering på Taylor som källa. Avsikten är inte här att upprepa Bravermans analys; istället kommer jag här att ta fasta på de aspekter av Taylor respektive DeMarco som jag ser som särskilt intressanta i jämförelsen.

3.1 The Principles of Scientific Management

Taylor beskriver sin rationaliseringsmetod som byggd på tre grundläggande principer: *skapandet av en "vetenskap"* för den uppgift som ska ledas, *urval och utbildning* av arbetare, samt *det "hjärtliga samarbetet"* mellan arbetare och ledning. Den sista punkten innebär specifikt att arbetsprocessen hanteras av arbetsledningen, inte av arbetaren, något Taylor också framhåller som den centrala överordnade strategin för metoden och något som tydligt skiljer den från tidigare metoder.[24]

Princip 1: skapa en vetenskap

Principen innebär, i de exempel Taylor ger oss, att verksamheten beskrivs som matematiskt problem; idag kan man kalla detta en *modell*. Modellen beskriver det system som ska styras – det vill säga arbetaren, hans verktyg och hans material.[25] Taylor tar grävning som exempel och formulerar ett optimeringsproblem: för lätta spadtag är ineffektiva, för tunga är långsamma; optimum ligger alltså någonstans däremellan. Utifrån denna modell experimenterar han sig fram till att det optimala spadtaget lyfter 21 pund, oavsett vem som lyfter.[26] Han ger också exempel på mer komplexa modeller, som den stålskärningsprocess som innefattade tolv obero-

[24]Taylor är inkonsekvent och räknar ibland till tre, ibland till fyra principer, inte alltid desamma; här har jag, liksom Braverman, kombinerat den tredje principen om "hjärtligt samarbete" mellan arbetare och ledning med den fjärde principen om arbetsledningens kontroll över processen. Tre principer, se Taylor, 2005, s. 105; fyra (något varierande) principer, se Taylor, 2005, s. 34, 78–79, 119; vikten av att centralisera arbetsprocessen, se Taylor, 2005, s. 6–7, 30, 59, 95.

[25]Taylor skriver genomgående om verkstadsarbetaren som "han", vilket nog kan anses normalt för perioden. Kvinnor omnämns inte, förutom de unga flickor som anlitades för att sortera metallkulor; se Taylor, 2005, s. 85.

[26]Taylor, 2005, s. 60–61. 21 pund är knappt 10 kg.

ende variabler.[27] Idén är genomgående att abstrahera bort allt Taylor anser ovidkommande för produktiviteten – som exempelvis trivsel och småprat[28] – och optimera det som är kvar.

Braverman beskriver denna princip som ”arbetsprocessens separering från arbetarens yrkesskicklighet”.[29] Istället för att arbetaren har kunskapen om – och kontrollen över – hur spadtagen ska göras eller hur skärmaskinen ska ställas in, så fastslås arbetsprocessen av arbetsledningen. Termen ”vetenskap” får arbetsprocessen att framstå som objektivt självklar, men genom att arbetsledningen formulerar modellen – och därigenom definierar vad man ser som viktigt i sammanhanget – formulerar man också vad den ”vetenskapliga” eller optimala lösningen faktiskt är.

Taylor skriver i första kapitlets första mening något om vad som menas med ”optimal”, det vill säga vad som är arbetsledningens mål: ”maximalt välstånd för arbetsgivaren, kopplat med maximalt välstånd för arbetaren”.[30] Det är en elegant formulering som utgår ifrån att båda parternas välstånd kan maximeras tillsammans utan att gå ut över varandra. De exempel som ges handlar emellertid entydigt om maximering av produktionen, och Taylor menar att det vore orimligt att höja lönerna alltför mycket, eftersom alltför höga löner gör arbetarna ”initiativlösa, extravaganta och utsvävande”.[31] Mot bakgrund av den arbetets degradering som Braverman beskriver, där arbetaren hamnar i en alltmer trängd position på arbetsmarknaden, kan man nog utan att ta i istället sammanfatta Taylors optimalitetskriterium som *maximalt välstånd för arbetsgivaren utan att arbetarna strejkar*. Taylor framhåller genomgående just det låga antalet strejker som ett mått på arbetarnas acceptans för rationaliseringarna.[32]

Princip 2: urval och utbildning

Taylors andra princip är att placera arbetarna i nya, välavgränsade roller; där de tidigare haft ansvar för sitt eget arbete, har de nu till uppgift att utföra enskilda arbetsmoment enligt arbetsledningens instruktioner. Omställningen anses sällan naturlig eller ens frivillig, något som framgår av Taylors många exempel på hur man måste tvinga arbetarna att följa de nya arbetsprocesserna. Braverman ser i denna specialisering en viktig del av arbetets degradering.[33]

I Taylors beskrivning av denna princip framträder något om synen på enskilde arbetaren. Den idealiske järnverksarbetaren beskrivs som dum, flegmatisk, med en oxes mentalitet; han är dessutom av naturen motvillig och maskar så fort möjligheten ges.[34] När Taylor argumenterar

[27] Taylor, 2005, s. 98–100.
[28] Taylor, 2005, s. 85.
[29] Braverman, 1989, s. 104.
[30] Taylor, 2005, s. 9.
[31] Taylor, 2005, s. 68.
[32] Om avsaknad av strejker då Taylors metoder används, se Taylor, 2005, s. 26, 39, 91, 125; om strejker då Taylors metoder inte används eller används felaktigt, se Taylor, 2005, s. 119, 121–122, 125.
[33] Taylor, 2005, s. 76, 83, 85, 112–113, 116; Braverman, 1989, s. 104–105.
[34] Taylor, 2005, s. 15–16, 35, 54.

för att arbetarna endast kan anförtros enkla arbetsmoment så gör han det alltså utifrån sitt eget antagande om att arbetare av nödvändighet är dumma och lata – människor kan antingen arbeta (motvilligt) eller tänka.

Princip 3: det ”hjärtliga samarbetet”

Taylor skriver genomgående om det ”hjärtliga” eller ”intima” samarbetet mellan arbetare och arbetsledning, men också om disciplinåtgärder, vikten av att leda arbetarna enskilt istället för i grupp, liksom sin tydligt fackfientliga inställning. Det ”hjärtliga samarbetet” uppvisar alltså tydliga drag av disciplinering,[35] och istället framträder en noggrann detaljstyrning av arbetet, med arbetsprocessen tydligt placerad helt under arbetsledningens kontroll.

Taylors arbetsledning var en separat del av organisationen, placerad i egna lokaler – en planeringsavdelning. Tankearbetet hade flyttats från fabriksgolvet till denna planeringsavdelning; detaljerade instruktioner formulerades inför varje arbetsdag och förmedlades till arbetarna genom särskilda postfack. Arbetarens produktion mättes dagligen – i vissa fall oftare – och färgen på lapparna indikerade huruvida föregående dags arbete levde upp till produktionskraven: gul lapp betydde otillräcklig prestation, och omplacering om arbetaren inte bättrade sig.[36]

Noggrann mätning av individens prestationer var alltså en central idé hos Taylor, och med tiden kom sådana mätningar att göras av de tidsstudiemän som ibland förknippas med Taylor. Rollen som tidsstudieman blev ett eget yrke, format av bl. a. Frank Gilbreth.[37] Även om detta var en specialiserad roll, så var det emellertid en roll med kontroll över arbetsprocessen; därför kan tidsstudiemannens specialisering nog inte anses innebära någon degradering i arbetet – snarare tvärtom.

3.2 Controlling Software Projects

DeMarco har inte, som Taylor, några tydligt utstakade principer, utöver möjligen vikten av detaljerade mätningar för projektledningen. Emellertid framträder några viktiga antaganden, liksom konsekvenser av dessa antaganden. Dessa beskrivs här närmare.

Antagande: programvaruutveckling är produktion

DeMarcos bok handlar om metoder för att leda programvaruprojekt och uppnå god kvalitet i resultaten. En central aspekt av sådan projektledning är att uppskatta projektets omfattning,

[35] Om det ”hjärtliga”, ”vänliga”, ”intima” samarbetet, se Taylor, 2005, s. 13, 25, 34, 92, 105, 119, 123, 128, 130; om disciplin, se Taylor, 2005, s. 114; om vikten av individuell ledning, se Taylor, 2005, s. 12, 63–64, 67–68, 76–77; om inställningen till facket, se Taylor, 2005, s. 16, 75; om Bravermans tilltro till Taylors berättelser, se Braverman, 1989, s. 90, särskilt fotnoten.

[36] Om planeringsrummet, se Taylor, 2005, s. 35, 63–64, 112–113; om lappar, postfack och omplacering, se Taylor, 2005, s. 63–64; om återkoppling en gång i timmen vid barnarbete, se Taylor, 2005, s. 87.

[37] Taylor, 2005, s. 68, 106; om Gilbreth, se Taylor, 2005, s. 71–77.

räknat t. ex. i nedlagda arbetstimmar, kalendertid eller kronor. Utan åtminstone en idé om omfattningen kan man ju inte förhålla sig till de ramar – tid och kostnad – som projektet har.

I metoderna för kostnadsuppskattning bok utgår DeMarco tydligt från andra ingenjörsdiscipliner. Han gör jämförelser med betonggjutning och pekar på hur man i byggbranschen använder tabellerade mätdata över vilken arbetstid, mätt i arbetstimmar per kvadratfot, som krävs för olika arbetsmoment. Utifrån dessa tabeller kan en total kostnadsuppskattning för betonggjutningen göras. DeMarcos idé är att använda en sådan lösning också för programvaruutveckling; han förutsäger rentav att den kommer att vara utbredd år 2000. Hans antagande är att software engineering visserligen är i sin linda, men måste förväntas följa samma utvecklingsbana som andra ingenjörsdiscipliner, specifikt sådana som handlar om produktion av fysiska föremål.[38] I antagandet ligger en idé om att programmerarens arbetsmoment kan separeras och standardiseras på samma sätt som betonggjutarens eller aluminiumpressarens.

En central komponent i detta produktionstänkande är alltså arbetsdelning *inom* yrket; de arbetsmoment i betonggjutningen, som DeMarco tar som exempel, kan ju utföras av olika personer. Detta syns tydligt i användningen av tabellerade arbetsmoment – tabellen är given på förhand. Momenten måste även i arbetets genomförande vara exakt de som ges i tabellen, annars stämmer ju inte kostnadsberäkningen. Med andra ord innebär DeMarcos ideal att en detaljerad arbetsprocess centraliseras till arbetsledningen, istället för att lämnas upp till den som ska genomföra arbetet, samt att de ingående arbetsmomenten standardiseras. Just i sådan arbetsdelning inom yrket, där arbetsprocessen kontrolleras av arbetsledningen istället för av de som utför arbetet, och de individuella arbetsmomenten kan utföras av olika personer, ser Braverman själva grunden i arbetets degradering.[39]

Konsekvens: mätgruppen utformar arbetsprocessen tillsammans med projektledaren

Ett problem är att bestämma vilka rader och kolumner som ska ingå i DeMarcos tabell, det vill säga vilka arbetsmoment som finns och hur de mäts. Programvara kan inte mätas i vare sig kvadratfot eller kilogram; visserligen kan ett färdigt program mätas i kilo- eller megabytes, liksom i antal rader programkod, men dessa mått uppstår först när programmet är klart.[40] Man behöver alltså mått på komplexiteten i det problem som ska lösas, men måttet måste kunna formuleras innan programmet är färdigt – annars kan det ju inte användas för projektledningen. Härur kommer DeMarcos ofta citerade fras ”you can’t control what you can’t measure”.[41]

Mätningar och kostnadsuppskattningar är en central del av DeMarcos metod; så central, menar han, att den bör hanteras av en särskild mätgrupp. Denna grupp ansvarar för att avgöra vilka

[38] DeMarco, 1982, s. 7 (betonggjutning), s. 22 (en disciplin i sin linda), s. 26–27 (aluminiumpressning), s. 28 (framtiden), s. 62–64 (statyettgjutning).
[39] Braverman, 1989, s. 72–76.
[40] DeMarco, 1982, s. 29, 163, 166–167.
[41] DeMarco, 1982, s. 3.

mätdata som behövs, insamlandet av dem, och sammanställningen till projektledaren. Exempel på vad som registreras är tidsåtgång i olika arbetsmoment, olika mått på specifikationernas och programmens komplexitet samt antalet fel (”buggar”).[42] DeMarco menar att det är viktigt att gruppen inte bara är separat från projektet; den får inte heller stå under projektledarens kontroll, eftersom dess prediktioner då riskerar avspegla projektledarens önskningar snarare än projektets verklighet. DeMarco bygger här på förebilder inom kvalitetsstyrning i produktionsorienterade industrier, som tryckerier, verkstadsindustri och gruvindustri. För att arbetsdelningen ska fungera får projektledaren inte ens avslöja eventuellt missnöje med mätgruppens prediktioner.[43] Ändå beskrivs mätgruppens uppgift som att bistå projektledaren,[44] och det är svårt att se hur den i praktiken skulle kunna vara oberoende.

DeMarco skriver uttryckligen att mätbehoven påverkar arbetsprocessen och därmed begränsar programmerarnas valfrihet. I detta resonemang ger han ett exempel på en arbetsprocess som *inte* kan accepteras: nämligen när programmerarna själva tar sig an uppgiften, konstruerar en lösning och arbetar med kunden för att göra denne nöjd. Mätgruppen kan inte fungera under sådana omständigheter: den måste kunna kräva metoder som är kvantifierbara och detaljerade – det vill säga kan uttryckas i tabellform – och formulerade på förhand, för att kostnadsuppskattningar ska vara möjliga.[45] När arbetsledningen, genom mätgruppen, på så sätt tar kontrollen över arbetsprocessen och bryter ner den i minimala arbetsmoment framträder återigen paralleller med Bravermans idéer om arbetets degradering.[46]

Antagande: arbetet bör organiseras utifrån en vattenfallsprocess

DeMarcos bok är tydligt bunden vid vattenfallsprocessen.[47] Denna process är ju inte DeMarcos skapelse, utan boken avspeglar i detta avseende snarare dåtidens software engineering-tradition. Vattenfallsprocessen uppvisar likheter med den Taylor-inspirerade flödesprocess som hittade in i kontorsarbetet under 1900-talets första halva; båda går ut på att låta dokument eller produkter passera ett antal stationer inom företaget, och där successivt vidarebehandlas.[48] Termen ”vattenfall” anspelar också här just på ett slags flöde, där det utvecklade programmet ”rinner” genom organisationen i en riktning, från en fas till nästa.

Greiff visar på hur kontorens flödesprocess ledde till en detaljerad arbetsdelning och tydlig degradering i arbetet, med billigare, yngre, utbytbar personal som följd.[49] Skillnader finns

[42] Om mätgruppen, se DeMarco, 1982, s. 6, 9–17; för exempel på insamlade data, se DeMarco, 1982, s. 83, 104, 114–128, 143, 211–213.
[43] DeMarco, 1982, s. 19.
[44] DeMarco, 1982, s. 66, fig. 8.5.
[45] DeMarco, 1982, s. 22–23.
[46] Braverman, 1989, s. 72–76.
[47] Se t. ex. DeMarco, 1982, s. 42, 80, 92–94, 103, 105, 111, 192–193; se även avsnitt 1.2 i denna uppsats.
[48] Se t. ex. Greiff, 1992, s. 173, 246–248; Braverman, 1989, s. 271–174.
[49] Greiff, 1992, s. 243 (om utbytbar personal), 252, 257 (om arbetsdelning).

emellertid från DeMarcos vattenfallsprocess, kanske främst i arbetsmomentens storlek: medan en fakturasorterare på SKF på 1930-talet kanske bara höll i varje faktura i någon sekund,[50] så kan en programmerare mycket väl arbeta med sin del av programmet i veckor inom ramen för vattenfallsprocessen. Med programmerarens större arbetsmoment följer en större frihet över arbetsprocessen *inom* det momentet. Ändå innebär vattenfallsprocessen en specialisering i arbetsrollerna; i en stor organisation, där flera sådana projekt genomförs samtidigt, är det inte ovanligt att personer tillbringar all sin tid i en av faserna, och istället flyttas från projekt till projekt.[51] Istället för att syssla med programvaruutveckling i stort specialiseras dessa då till t. ex. kravanalytiker, designers, implementatörer eller testare.

DeMarcos bok handlar till stor del om modellering av systemet, från tidigt stadium – då bara en kravspecifikation finns – till den färdiga produkten. Mycket av denna modellering syftar till att utifrån en av vattenfallsprocessens faser beräkna uppskattningar för den kommande; exempelvis predikteras implementationens kostnad utifrån mått på designmodellen.[52] De separata faserna är alltså både en förutsättning för DeMarcos kostnadsuppskattningar och en orsak till specialisering av det slag som Greiff och Braverman ser som degraderande.[53]

Konsekvens: specialisering till implementatör, testare och mätare

DeMarco skriver en hel del om specialisering, framförallt i bokens sista del, som handlar om programvarukvalitet. I denna del presenterar han sina idéer om hur projekt ska ledas för att undvika fel i det färdiga resultatet – *zero defect development*.[54] Enligt dessa idéer kategoriseras projektets medlemmar som konstruktörer, testare och mätare; dessa tre grupper görs också till uttryckliga motståndare. Konstruktörerna försöker att minimera antalet fel; testarna försöker istället att finna så många fel som möjligt; de oberoende mätarna registrerar resultaten.[55]

Den specialisering som DeMarco föreskriver är långtgående, inte bara som en bieffekt av vattenfallsprocessen, utan även som önskvärd av disciplinskäl. Konstruktörerna får t. ex. inte provköra det program de skriver; de har inte ens själva tillgång till kompilatorn, utan denna är begränsad till testarna.[56] Detta kan liknas vid att förhindra historiestudenten från att använda stavningskontroll på sin uppsats, och istället ge opponenten detta ansvar. DeMarco motiverar detta med att programmeraren gör slarvfel på grund av fel attityd; när kompileringen överlåts till testarna blir felen tydligare och programmeraren tvingas skärpa sig. Programmerarna tycker

[50] Greiff, 1992, s. 251.

[51] Royce tryckte själv på denna arbetsdelning för bästa utnyttjande av resurserna (det vill säga människorna i projektet); se Royce, 1970, s. 328. DeMarco menar att en person visserligen kan uppfylla två roller, men att dessa roller då måste separeras fullständigt, DeMarco, 1982, s. 92.

[52] DeMarco, 1982, s. 111, 123.

[53] Braverman, 1989, s. 72; Greiff, 1992, s. 261–264.

[54] DeMarco, 1982, kap. 22; se även kap. 19–21.

[55] DeMarco, 1982, s. 219–220.

[56] DeMarco, 1982, s. 217–221, 229. Kompilatorn är programmerarens verktyg: det används för att kontrollera programmets syntax och semantik samt översätta det till de binära maskininstruktioner som datorn kan utföra.

rent av att det är roligt att felsöka, menar DeMarco, och ägnar sig hellre åt detta än nykonstruktion om de har möjlighet.[57] Hans zero defect development har alltså drag av disciplinering: programmerarna tycker att det är roligare att göra fel än att göra rätt; genom att placeras i smala fack och kämpa inbördes ska de förmås att skärpa till sig. Synsättet kan emellertid ifrågasättas. Jag har själv aldrig träffat en programmerare som föredrar felsökning framför nyutveckling, däremot har jag träffat många som tycker tvärtom – det är mycket roligare att skapa nytt än att jaga fel i något gammalt.

Rollen som testare påverkas också av denna specialisering. DeMarco konstaterar att vissa programmerare gör fler fel än andra, och menar att det kan löna sig att låta dessa programmerare bli testare; även om de kanske utför en mindre kvalificerad uppgift, så är deras värde för företaget större i rollen som testare. Utifrån detta argument menar DeMarco att en sådan omplacering inte alls vore negativ för individen.[58] Några alternativ till omplacering – som exempelvis utbildning för uppgiften – nämns emellertid inte, och jag har svårt att föreställa mig en sådan omplacering som något annat än ett yrkesmässigt nederlag. Detta nederlag skiljer sig från den degradering som Braverman beskriver; där degraderas sällan individer direkt, utan istället ställs allt lägre krav på de nyanställda. I fallet här med den omplacerade programmeraren tycks degraderingen emellertid slå direkt mot individen.

Specialiseringen till mätare skiljer sig från den till testare. De är visserligen möjliga specialiseringar för en programmerare, men till skillnad från testarna rekryteras DeMarcos mätare bland de bästa programmerarna. Mätarna avgör dessutom själva vad som mäts och hur man mäter det – med andra ord får de själva kontrollen över sin arbetsprocess.[59] Mot denna bakgrund framstår mätarrollen som ett tydligt steg uppåt i karriären, närmare arbetsledningen.

3.3 Likheter och skillnader

Taylor och DeMarco skriver båda om arbetsledning ur ett ingenjörsperspektiv. De formulerar matematiska modeller för verksamheterna som leds, och utifrån modellerna formas verksamheterna mot arbetsledningens mål. De delar synen på arbetarna/programmerarna som ingående i ett större system, som formas och styrs på ett ingenjörsmässigt sätt.

Taylor och DeMarco styr mot olika mål: Taylor söker produktivitet, DeMarco söker förutsäga kostnad och tidsåtgång. Arbetarens lön är tydligt en del av Taylors beräkningar, och i hans bok återfinns ju också en hel del resonemang om vilken lön arbetaren förtjänar. DeMarco, å andra sidan, berör inte lönefrågan alls – den regleras istället av arbetsmarknaden. DeMarcos målsättning innebär, till skillnad från Taylors, inte nödvändigtvis någon arbetets degradering i sig.

[57] DeMarco, 1982, s. 217, 219.
[58] DeMarco, 1982, s. 210–211.
[59] DeMarco, 1982, s. 22, 37.

Emellertid beror ju lönen på arbetsuppgifterna, och den specialisering av dessa som både Taylor och DeMarco förespråkar: Taylor i verkstadsindustrins smala arbetsroller, DeMarco i testare och programmerare; den senare uppdelningen är tämligen skarp i DeMarcos idéer om zero defect development. I DeMarcos fall innebär också den vattenfallsprocess, som han bygger sitt arbete på, en specialisering av arbetsuppgifterna i stora organisationer. Både Taylor och DeMarco argumenterar för specialiseringen som ett sätt för den enskilde att finna det arbetsmoment som han eller hon är allra bäst på, men denna specialisering är ju också ett centralt element i den arbetets degradering Braverman beskriver.[60]

Kvantitativa mätningar är centrala i både Taylors och DeMarcos idéer, och de upprättar båda speciella funktioner för dessa mätningar i sina respektive organisationer: Taylor skriver om sina tidsstudiemän, DeMarco om sin mätgrupp. I båda fallen blir mätfunktionen arbetsledningens redskap i centraliseringen av arbetsprocessen.[61] I denna centralisering, där individerna fråntas kontrollen över processen, ser Braverman en tydlig källa till arbetets degradering.[62] I båda fallen används mätningarna också för att utöva påtryckningar på arbetarna/programmerarna; i Taylors exempel med gula lappar, i DeMarcos genom att mätare rekommenderar programmerare att bli testare.[63] Greiff menar att redan själva flödesprocessen drev kontorsarbetarna till ett högre arbetstempo.[64] Den kontinuerliga övervakningen av prestationerna gjorde förmodligen också sitt till: Braverman ser kontrollen som central i taylorismen, och även titeln på DeMarcos bok, *Controlling Software Projects*, handlar ju ytterst om kontroll.[65]

En nyansskillnad kan emellertid ses i Taylors respektive DeMarcos syn på relationen mellan arbetsledningen och arbetarna/programmerarna. Taylor har, som redan visats, ett budskap som rubriceras som ett ”hjärtligt samarbete” men istället visar drag av disciplinering. DeMarco visar mestadels inte samma motsättning mellan arbetsledning och programmerare; maximal produktivitet står ju inte i fokus, så den motsättning som föreligger hos Taylor finns inte nödvändigtvis hos DeMarco. I sista delen av DeMarcos bok, där zero defect development beskrivs, beskriver han emellertid programmerare, testare och mätare som varandras motståndare: han beundrar den testgrupp på IBM, *The Black Team*, som påstods älska att få programmerare att gråta, och omnämner sin testgrupp som ”Search and Destroy Team”.[66] Den största skillnaden är kanske att DeMarco framstår som ärligare i sin syn på relationerna.

Kan vi då skönja Taylors principer hos DeMarco? Till stor del, menar jag. Den första principen, ”vetenskapen”, utgör ju själva basen för både Taylor och DeMarco; som goda ingenjörer

[60]Taylor, 2005, s. 9; DeMarco, 1982, s. 210; Braverman, 1989, s. 104–105.
[61]Taylor, 2005, s. 104; DeMarco, 1982, s. 23, 111.
[62]Braverman, 1989, s. 103–104.
[63]Taylor, 2005, s. 63–64; DeMarco, 1982, s. 210–211.
[64]Greiff, 1992, s. 248–249.
[65]Braverman, 1989, s. 85.
[66]Om olika grupper som motståndare, se DeMarco, 1982, s. 219–221; om *The Black Team*, se DeMarco, 1982, s. 221; ”Search and Destroy Team”, se DeMarco, 1982, s. 230, fig. 22.2.

formulerar de matematiska modeller för verksamheten och styr utifrån dessa. De lutar sig båda mot vetenskapens ideal om objektivitet, DeMarco dessutom mot idén om att software engineering självklart bör efterlikna mer traditionella ingenjörsdiscipliner.

Braverman kritiserar denna princip som ”arbetsprocessens separering från arbetarens yrkesskicklighet”,[67] och idén om en allmängiltig, centraliserad process, oberoende av individerna, är ju också något som går igen i vattenfallsprocessen och dess varianter.[68] Denna är förvisso inte DeMarcos uppfinning, utan får ses som representativ för software engineering då boken skrevs; DeMarco bygger emellertid tydligt på den och själva vattenfallsidén är en förutsättning för hans mätningar.

Den andra principen handlar om att foga in individen inom denna ”vetenskapliga” arbetsprocess. Det är inte alltid så lätt att dela in programmerarens arbete i lika smala fack som Taylor placerar arbetaren, men resonemangen om kostnadsuppskattningar för betonggjutning gör klart att DeMarco tar sikte på sådana enkla, standardiserade arbetsmoment även för programmeraren. Hans idéer om zero defect development placerar också in programmerare och testare i smala roller, där deras befogenheter tydligt begränsas.

I just dessa idéer är Bravermans kritik mot Taylors andra princip – ”skilsmässan mellan begrepp och verkställighet”[69] – också som allra mest tillämplig. Kritiken gäller taylorismens sätt att frånta arbetaren överblicken över det sammanhang som arbetet görs i; samma kritik kan också riktas mot DeMarcos idéer om zero defect development, där programmeraren inte ens tillåts provköra det program han eller hon själv skapar.

I Taylors tredje princip – ”det hjärtliga samarbetet” – syns också likheter, kanske framförallt i tidsstudiemännens respektive mätgruppens roller. Dessa båda mätfunktioner fungerar som en del av arbetsledningen och är en viktig komponent i både centraliseringen av arbetsprocessen och uppföljningen av de enskildas prestationer. Relationen mellan projektledaren och programmerarna i DeMarcos värld tycks emellertid mindre spänd än den mellan arbetsledaren och arbetarna i Taylors – förmodligen just eftersom DeMarcos mål, kostnadsuppskattningar och kvalitetsstyrning, inte på samma sätt står i konflikt med programmerarnas intressen.

4 Sammanfattande diskussion

Arbetsdelningen och specialiseringen i industrin har givit oss en produktivitetsökning utan motstycke. Vi har delvis Taylor och hans efterföljare att tacka för vår tillgång till billiga, högkvalitativa massproducerade varor. Just därför är det också viktigt att belysa baksidorna med denna utveckling, så som Braverman gjort genom sitt fokus på arbetsprocessen. Ur Tay-

[67] Braverman, 1989, s. 104.

[68] DeMarco, 1982, s. 131–132 resonerar kring det omöjliga i en allmängiltig process, oberoende av organisation och uppgift; ändå utgår han från vattenfallsidén som självklar.

[69] Braverman, 1989, s. 105.

lors bok, med Braverman som guide, framträder en fabriksarbetare som å ena sidan kanske fått en löneförhöjning, men å andra sidan fastnat i ett monotont, föga givande arbete. Han har fått sälja sin själ, åtminstone den del av den som hörde till arbetet.

Det tycks onekligen som om dessa rationaliseringsidéer hittat in i software engineering också. Mahoney indikerade att Taylors idéer kan spåras i 1968 års NATO-rapport.[70] I denna uppsats har visats mer i detalj hur idéerna tydligt återfinns även i 1980-talets software engineering-litteratur för yrkesverksamma ingenjörer. Exakt hur de hittade dit kan man undra: Greiffs spår via kontoret är en möjlighet, produktionstekniska influenser i 1968 års NATO-konferens är en annan. I DeMarcos fall finns en tydlig strävan att efterlikna andra, etablerade ingenjörsdiscipliner; kanske spelar fadern också in. I alla händelser kan Taylors principer skönjas i DeMarcos bok, och sålunda framträder en nischad programmerare, vars kodkvalitet kvantifieras och övervakas, och som kanske inte ens har befogenhet att köra sitt eget program under utvecklingen.

Jag menar alltså att den kritik, som Braverman riktat mot Taylor, också i väsentliga delar kan riktas mot DeMarco. DeMarcos idéer tycks, på samma sätt som Taylors, medföra en degradering av det slag Braverman beskriver. Programmerarens relativt goda utgångsläge på arbetsmarknaden gör att degraderingen kanske kan avfärdas som ett lyxproblem, men degraderingens *riktning* är densamma. Greiffs beskrivning av kontorsarbetarens roll, som under 1900-talets första halva degraderades från en självständig ”allroundtjänsteman” nära företagsledningen till en specialiserad, detaljstyrd låglönearbetare, ger anledning till eftertanke.[71]

Den beskrivna degraderingen återfinns i Taylors och DeMarcos normer. Effekterna i det faktiska arbetet är en annan fråga. Det är ironiskt att software engineering inte har inneburit tillnärmelsevis samma framsteg som taylorismen i vare sig produktivitet, kostnadsuppskattning eller förmåga att klara deadlines. Till och med den åldrande DeMarco ifrågasätter numera om software engineering egentligen någonsin var någon bra idé.[72] Brooks förklarar varför programmerarens arbete är svårt att bryta ner i standardiserade arbetsmoment;[73] det Babbage ställde till för Braverman blir också programmerarens räddning.

Bravermans bok har undertiteln ”arbetets degradering i det tjugonde århundradet”.[74] Men vad händer i det tjugoförsta? Han har kritiserats för sin deterministiska syn på utvecklingen, där allt fler obönhörligen får se sitt arbete urholkas på innehåll; Åmark menar att marknaden, åtminstone i vissa sammanhang, kan ge utvecklingen en annan riktning.[75] På liknande sätt

[70] Mahoney, 2004, s. 13–15.

[71] Greiff, 1992, s. 97–105 (”allroundtjänsteman”), jämför med t. ex. 315–318.

[72] Tom DeMarco, ”Software Engineering: An Idea Whose Time Has Come and Gone?”, *IEEE Software* 26(4), 2009.

[73] Brooks, 1987.

[74] Braverman, *Arbete och monopolkapital. Arbetets degradering i det tjugonde århundradet*, 1989.

[75] Åmark, 1996, s. 161.

är 2000-talets agile-metoder sprungna ur marknadskrav på datorprogram som fungerar och levereras i tid. Dessa metoder skiljer sig också på viktiga punkter från den software engineering som diskuterats här. De kommersiella krafter som verkade för Taylors idéer verkar alltså också mot DeMarcos. Programmerarens degradering i software engineering är inte lika entydig som den framstod för Braverman – marknadskrafterna tycks ha dämpat den snarare än förstärkt den.

Fler frågetecken återstår kring idébakgrunden till software engineering. Babbages' idéer känns ju igen i både industrins arbetsdelning och den moderna datortekniken. Kan dessa idéer även skönjas i software engineering? Den större, och svårare, frågan är i så fall om Babbages nedbrytning i elementära uppgifter är det enda sättet att lösa komplexa tekniska problem.

En relaterad fråga gäller de militära influenserna. Braverman ser ett militärt ursprung till arbetsdelningen; 1968 års tongivande software engineering-konferens anordnades av NATO; DeMarcos språk tycks ibland färgat av militära influenser.[76] I vilken utsträckning leds dagens programmerare utifrån militära principer?

Vidare kan man fråga sig i vilken utsträckning agile-metoderna egentligen gör upp med taylorismen. Beck och Andres tar uttryckligen avstånd från taylorismen, men hänvisar samtidigt till DeMarcos *Controlling Software Projects* för exempel på hur man mäter och styr programvaruprojekt. Schwaber och Beedle kritiserar tidigare metoder för deras reglertekniska modeller av utvecklingsprojekten, men menar att lösningen är robustare reglerteknik.[77] Matematiska modeller betyder emellertid inte nödvändigtvis taylorism – Taylor hade ju fler principer än så.

Såväl Taylor som DeMarco tycks dela Bravermans deterministiska syn på utvecklingen, åtminstone i vissa stycken. Taylor är övertygad om att varje bransch strävar mot en universell metod, bättre än alla andra; DeMarco utgår från att även programmering en dag – år 2000, närmare bestämt – kan kostnadsuppskattas från tabelldata, på samma sätt som görs i andra ingenjörsdiscipliner.[78] Ingen av dem – vare sig Braverman, Taylor, eller DeMarco – har fått rätt.

Babbage tycks ha sett likheter i arbetsdelningen i industrin och i datorn. Jämförelsen tål ännu att göras: den moderna datorutvecklingen har delvis möjliggjorts genom en uppdelning i allt enklare maskininstruktioner, utförda enligt en löpande-band-liknande princip.[79] I Taylors industri ruvar arbetsledaren på processen; i datorn utgörs istället denna process av datorprogrammet. Detta är det ju programmerarna som utformar, på nya sätt varje gång, och det verkar faktiskt rätt svårt att helt ta ifrån dem nöjet i att få saker att fungera.

[76] Braverman, 1989, s. 65, även fotnot; DeMarco, 1982, s. 132, 230; kanske passar även "survived" på s. 153 in här.

[77] Beck och Andres, 2005, s. 131–133, 169; Schwaber och Beedle, 2002, s. 94–103.

[78] Taylor, 2005, s. 23; DeMarco, 1982, s. 26, 28.

[79] RISC (*Reduced Instruction Set Computers*) och *instruction pipelining*.

Källor

DeMarco, Tom, *Controlling Software Projects: Management, Measurement, and Estimation*, Prentice Hall, Upper Saddle River, NJ, 1982

Taylor, Frederick W., *Principles of Scientific Management*, 1st World Library, Fairfield, IA, 2005

Litteratur

Beck, Kent och Cynthia Andres, *Extreme Programming Explained: Embrace Change*, andra utgåvan, Addison-Wesley, Boston, MA, 2005

Berner, Boel, ”Datorkulturen – en manlig historia”, Berner, Boel (red.), *Vem tillhör tekniken? Kunskap och kön i teknikens värld*, Arkiv, Lund, 2003

Boehm, Barry W. och Philip N. Papaccio, ”Understanding and Controlling Software Costs”, *IEEE Transactions on Software Engineering* 14(10), 1988

Braverman, Harry, *Arbete och monopolkapital. Arbetets degradering i det tjugonde århundradet*, Rabén & Sjögren, Stockholm, 1989

Bromley, Allan G., ”Charles Babbage’s Analytical Engine, 1838”, *IEEE Annals of the History of Computing* 20(4), 1998

Brooks, Frederick P., ”No Silver Bullet—Essence and Accidents of Software Engineering”, *IEEE Computer* 20(4), 1987

DeMarco, Tom, ”Software Engineering: An Idea Whose Time Has Come and Gone?”, *IEEE Software* 26(4), 2009

Glass, Robert L., *Software Creativity 2.0*, developer.* Books, Atlanta, GA, 2005

Greiff, Mats, *Kontoristen. Från chefens högra hand till proletär*, doktorsavhandling, Lunds universitet, Mendocino, Lund, 1992

Humphrey, Watts S., *A Discipline for Software Engineering*, Addison-Wesley, Reading, MA, 1995

Mahoney, Michael S., ”Finding a History for Software Engineering”, *IEEE Annals of the History of Computing* 26(1), 2004

McKay, John P. m. fl., *A History of World Societies*, Bedford/St. Martin’s, Boston/New York, 2009

Mills, Everald E., ”Metrics in the software engineering curriculum”, *Annals of Software Engineering* 6, 1998

Mulinari, Paula, *Maktens fantasier och servicearbetets praktik*, doktorsavhandling, Linköpings universitet, 2007

Nissen, Jörgen, ”Datorkulturen – en manlig historia”, Berner, Boel (red.), *Vem tillhör tekniken? Kunskap och kön i teknikens värld*, Arkiv, Lund, 2003

Randell, Brian och Peter Naur (red.), *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Scientific Affairs Division, NATO, 1968

Royce, Winston, ”Managing the Development of Large Software Systems”, *Proc. IEEE WESCON*, 1970

Salminen-Karlsson, Minna, ”Hur skapas den nya kritikens skapare?”, Berner, Boel (red.), *Vem tillhör tekniken? Kunskap och kön i teknikens värld*, Arkiv, Lund, 2003

Schwaber, Ken och Mike Beedle, *Agile Software Development with Scrum*, Prentice Hall, Upper Saddle River, NJ, 2002

Sommerville, Ian, *Software Engineering*, femte utgåvan, Addison-Wesley, Wokingham, 1996

Swedsoft, *Mjukvaran är själen i svensk industri*, http://www.swedsoft.se/Mjukvaran_är_själen_i_svensk_industri.pdf, 2008, hämtad 28 december 2011

Åmark, Klas, ”Genmäle till till Carl-Axel Nilsson”, *Historisk tidskrift* 116, 1996