[simula . research laboratory]

INF3120 H2006

Problemløsing i team

ved hjelp av

lettvektsmetodikker og -teknikker

Hans Gallis
hansga@simula.no

Pensum

- Kap. 17 i Sommerville (temaet inngår også i kategorien "systemutviklingsprosesser")
- Kursorisk:
  - Kent Beck; *Embracing Change with Extreme Programming*; IEEE Computer, October 1999
  - Barry Boehm; *Get Ready for Agile Methods, with Care*; IEEE Computer, January 2002
- For spesielt interesserte, se bok om agile metoder (pdf): http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf

Mål

- Bevissthet tilknyttet “Agile Software Development”
- Deltaljert kjennskap til én “Agile Method”: eXtreme Programming
- Detaljert kjennskap til én teknikk innen eXtreme Programming: Parprogrammering

Wicked Problems

1. There is no definitive formulation of a wicked problem
2. Wicked problems have no stopping rule
3. Solutions to wicked problems are not true-or-false, but good-or-bad
4. There is no immediate and no ultimate test of a solution to a wicked problem
5. Every solution to a wicked problem is a “one-shot operation”; because there is no opportunity to learn by trial-and-error, every attempt counts significantly
6. Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan
7. Every wicked problem is essentially unique
8. Every wicked problem can be considered to be a symptom of another problem
9. The existence of a discrepancy representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem’s resolution
10. The planner has no right to be wrong

Rittel and Webber, *”Dilemmas in a General Theory of Planning”*, Policy Science **4** (1973)

Historisk perspektiv

- Software Crisis (1960's)
  - Software intensive systems delivered late, over budget and do not meet the quality requirements
- Solution attempt #1: **Structured Methods** (in 1980's)
- Solution attempt #2: **Object-oriented methodologies**
- Chronic Software Crisis (1990's)
  - Software intensive systems still delivered late, over budget and do not meet the quality requirements
- Solution attempt #3: **Software process improvement**
- Solution attempt #4: **Agile development methodologies**
- Solution attempt #n: ?

Pekka Abrahamsson 2005

Ofte tilfelle i den "virkelige" verden……

- Krav endrer seg hele tiden
- Rask time-to-market (Internet)
- Lite langsiktige planer
- 60% av funksjonaliteten blir sjeldent eller aldri benyttet
  - Jim Johnson, Standish Group

Dette medfører:

- Må ha hyppige leveranser
- Må kontinuerlig endre koden (og designet)
  - Forutsetter at systemet er i en "sunn" tilstand (hele tiden!)
- Må kommunisere kontinuerlig (utvikle "riktig" produkt)

Paradigmeskifte?

Før:

- Internett lite "modent"
- (Rask) time-to-market
- **Vannfall vs iterativ/inkrementell utv.**

Nå:

- Internett "modent"
- Raskere time-to-market
- **Agile vs plan-driven utv.**

Agile vs plan-driven methodologies

- Agile (= smidig på norsk?):
  - Planlegger ikke for fremtiden
  - Ikke dokumentdrevet
- Plan-driven:
  - Planlegger for fremtiden
  - Dokumentdrevet

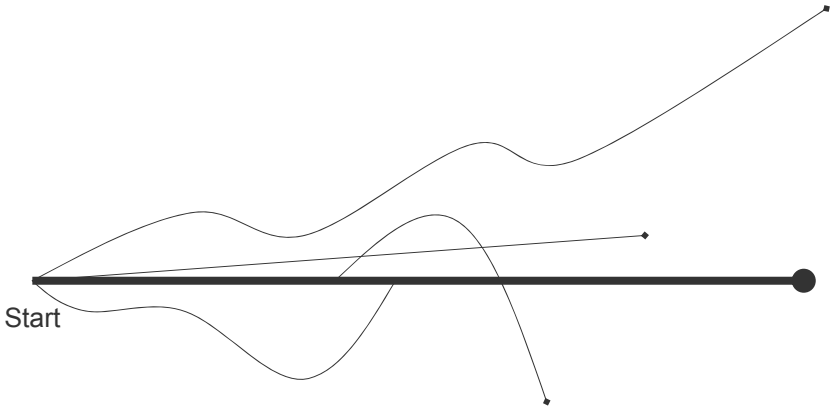
Modell vs metodikk vs prosess

Modell/filosofi

Metodikk

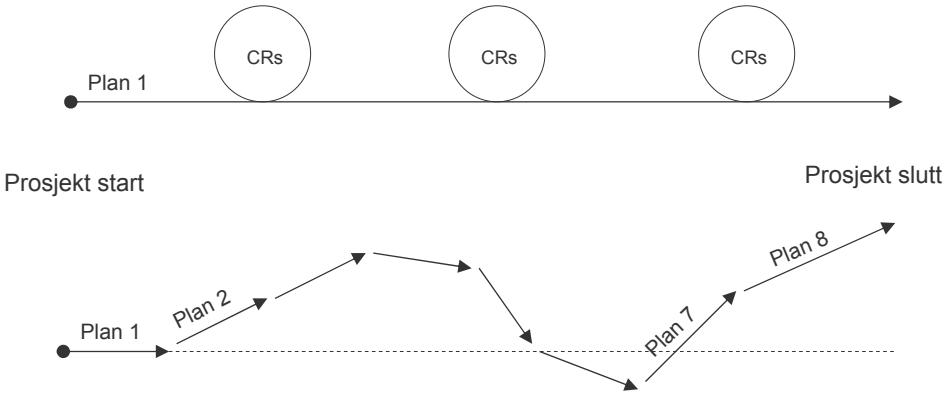
Prosess

Planlegging og fremtidige mål



→ = Mål

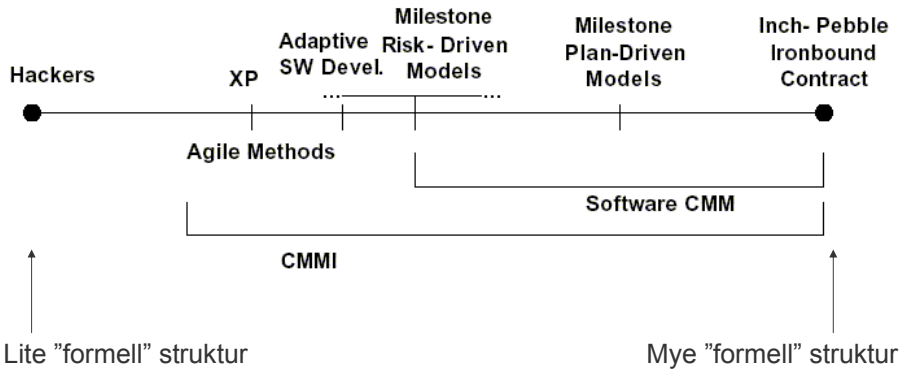
Agile vs plan-driven



Evolusjon og læring

- Usikkerhet krever læring (hyppige tilbakemeldinger)
  - Fra kollegaer
  - Fra iterasjoner i prosjektet
  - Fra testing av kode/system
  - Fra andre prosjekter
  - Fra kunder og sluttbrukere
  - Fra andre organisasjoner/systemer
  - Fra forskning!

Agile vs plan-driven methodologies



Agile Home Ground	Plan-Driven Home Ground
• Agile, knowledgeable, collocated, collaborative developers	• Plan-oriented developers, mix of skills
• Above plus representative, empowered customers	• Mix of customer capability levels
• Reliance on tacit interpersonal knowledge	• Reliance on explicit documented knowledge
• Largely emergent requirements, rapid change	• Requirements knowable early; largely stable
• Architected for current requirements	• Architected for current and foreseeable requirements
• Refactoring inexpensive	• Refactoring expensive
• Smaller teams, products	• Larger teams, products
• Premium on rapid value	• Premium on high-assurance

The Agile Manifesto

- **Individer og interaksjon** fremfor prosesser og verktøy
- **Fungerende software** fremfor utstrakt (omfattende) dokumentasjon
- **Kundemedvirkning** fremfor kontraktsforhandlinger
- **Reagere på endringer** (fleksibilitet) fremfor å følge en plan
- Se www.agilemanifesto.org og www.agilealliance.org

NB! Uthevede verdier er vektlagt, men verdiene til høyre i setningene er også viktig!

Eksisterende agile metoder

- Methods for agile software development:
  - Agile software process model [Ayoama, 1998]
  - Adaptive Software Development [Highsmith, 2000]
  - Crystal Family of Methodologies [Cockburn, 2000]
  - Dynamic Systems Development Method [Stapleton, 1997]
  - Extreme Programming [Beck, 1999]
  - Feature-Driven Development [Palmer & Felsing, 2002]
  - Lean software development [Poppendieck x 2, 2003]
  - Scrum [Schwaber, 1995; 2002]
  - … the list is growing every year…
- Combination of approaches:
  - Agile Modeling [Ambler, 2002]
  - Internet-Speed Development [Cusumano & Yoffie, 1999; Baskerville et al., 2001; Truex et al., 1999]
  - Pragmatic Programming [Hunt & Thomas, 2000]

Pekka Abrahamsson 2005

eXtreme Programming (XP)

- Utviklet av Kent Beck og Ward Cunningham (1996)
- En kodenær disiplin for programvare-utvikling
  - 4 verdier
  - 12 prinsipper/praksiser
  - 4 kjerneaktiviteter
- Kode er det eneste man er helt avhengig av å produsere!
- Et sett av ”best practises” som sees i sammenheng og støtter hverandre (ikke noen nye praksiser!)

De fire verdiene

- Kommunikasjon
  - Problemer i et prosjekt kan ofte spores tilbake til en eller flere som ikke snakket med en eller flere andre!
- Enkelhet
  - Det er bedre å gjøre en enkel ting i dag, og betale litt mer i morgen, enn å gjøre en komplisert ting i dag som kanskje må endres i morgen (eller som kanskje aldri blir brukt)
- Tilbakemelding
  - Tilbakemeldinger på alle nivåer (hele tiden) hjelper oss å holde prosjektet på rett vei
- Mot
  - Sammen med de tre første verdiene hjelper 'mot' oss til å gjøre høyrisiko eksperimentering (gjøre ting på en annen måte), noe som også kan gi høy belønning/avkastning hvis det lykkes.

De 12 prinsippene (teknikker)

1. Planning game
2. Små releaser
3. Metaforer
4. Enkelt design
5. Testing
6. Refactoring
7. Parprogrammering
8. Kollektivt eierskap
9. Kontinuerlig integrasjon
10. 40-timers uke
11. On-site kunde
12. Kodestandarder

XP – mer en filosofi enn metodikk!

- XP sier ikke:
  - Hvordan praksisene skal gjennomføres
  - Hvem som skal utføre dem (roller)
  - Hvilke leveranser til hvilken tid
    - Hva skal f.eks. leveransene inneholde?
      - Dokumentasjon, designmodeller, kode, etc. etc.
- XP er:
  - Et sett av 12 enkeltstående praksiser
    - Skal gjennomføres **ekstremt**
  - Funksjonell prototyping?

XP's "extreme" filosofi

- Hvis testing er bra – ja, da gjør vi det hele tiden
- Hvis kodelesing/inspeksjoner er bra – ja, da gjør vi det hele tiden
- Hvis iterasjoner er bra – ja, da gjør vi det så ofte som mulig (dager og uker istedenfor måneder og år)
- Hvis endringer skjer uansett – ja, da tilrettelegger vi for det istedenfor å motvirke at det skjer (rigide planer)
- Hvis kommunikasjon/samarbeid er viktig – ja, da gjør vi det hele tiden

Planning game

- Klarlegge
  - hva er ønskelig (forretningshensyn)
  - hva er mulig (tekniske hensyn)
- Estimere
- Prioritere
- Planlegge

Små releaser

- Minst mulig kode
- Mest mulig forretningsverdi
- 1-2 måneder (mellom hver gang systemet settes i produksjon)
- Risikoreduserende (får hyppig feedback)

Metaforer

- Få et vokabular (et sett av felles begreper) for tekniske ting uten å bruke tekniske termer
- La prosjektet være styrt av én metafor
  - Operativsystem → skrivebord
  - Forhandlersystem → flytog-billettsystem
- Gjør det lettere å kommunisere og utarbeide arkitekturen

Enkelt design

- Møt dagens krav:
  - YAGNI = You aren't gonna need it!
  - Ikke bruk tid på "nice-to-have-features"
  - For mye up-front design vil medføre MYE unødvendig arbeid!
- Det riktige designet
  - kjører alle testene
  - har ikke duplikat logikk
  - kommuniserer godt
  - har færrest mulig metoder

Testing

- Funksjonalitet uten automatiske tester finnes ikke
- Programmerere
  - skriver enhetstester (test-first)
  - gir programmereren tillit til programmet
- Kunder
  - skriver funksjonelle tester
  - gir kunden tillit til programmet

Kontinuerlig integrasjon

- Kode integreres og testes hver dag
  - Skal hele tiden ha et stabilt system
  - Skal finne feil tidligst mulig
- Hvis integrasjon feiler har det å rette opp dette høyeste prioritet

Refactoring

- = forbedre kodestrukturen uten å påvirke dens eksterne oppførsel
- Under implementasjon
  - kan koden endres for å gjøre det enklere å implementere denne funksjonaliteten?
- Etter implementasjon
  - kan koden skrives om til et enklere design, uten at testene ødelegges?

Parprogrammering

- Kommer til slutt i forelesningen!

Kollektivt eierskap

- Alle er ansvarlig for hele systemet
- Alle har rett til å endre all kode
- Likevel, ikke alle kjenner alle deler like godt

On-site kunde

- Kunden
  - Sitter sammen med utviklerne
  - Svarer på spørsmål
  - Gjør prioriteringer (på lavt nivå)
  - Deltar i diskusjon av løsninger
- Ikke nødvendigvis 100% stilling
  - Hvis kunden ikke er villig til å investere skikkelig med tid i prosjektet er det kanskje et tegn på at prosjektet ikke er viktig nok?

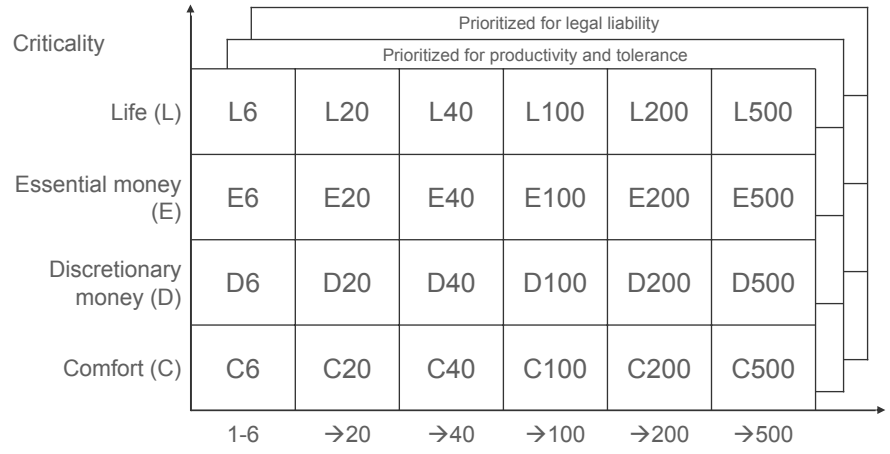
40-timers uke

- XP er krevende
  - Intensivt med korte leveranser
  - Ekstremt mye kommunikasjon
  - Mye tenking
- Viktig å være uthvilt
- 40-timers uke skal være normalen (ingen overtid!)
  - Får du ikke gjort jobben på 40 timer har du for mye å gjøre (justeres i neste release/iterasjon)
  - Mye overtid = symptom på usunt prosjekt (noe er galt!)

Kodestandarder

- Nødvendig for
  - Parprogrammering
  - Kollektivt eierskap
  - Disiplin
- Skal være enkel men dekkende

Når kan man benytte XP?

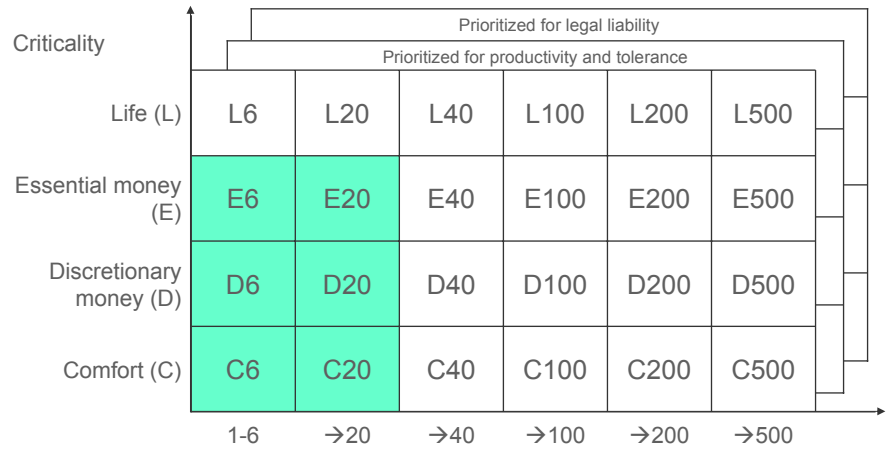


Alistair Cockburn, 2002

35

[simula . research laboratory]
21.09.2006

Når kan man benytte XP?



Alistair Cockburn, 2002

36

XP og Design

- Det er ingen som sier at design er nedprioritert i XP!
  - Gjør så mye design du trenger for å få oversikt og for å forstå
- UML-modeller forekommer på tavle, ark etc.

Oppsummering

- Agile ≈ pragmatisk systemutvikling
  - Velg praksiser/løsninger som passer dine behov (prosjekt)
  - Verktøykasse bestående av teknikker og praksiser som settes sammen til en helhetlig prosess
- XP:
  - Mer en filosofi enn en komplett utviklingsmetodikk
  - Kodenær
  - Menneskeorientert
  - Lettvekt (prøver å fjerne overhead)
  - Praksisene er ikke nye, men ekstreme!
  - Praktisk Knowledge Management

Parprogrammering (PP)

PP – Samarbeid og kommunikasjon

- Opptil 70% av totaltiden i et IT-prosjekt går med til kommunikasjon – designmøter, oppklaring av misforståelser etc. (Rapid Software Development through Team Collocation – Teasley *et al. 2002*)
- En utvikler bruker generelt (Peopleware – DeMarco and Lister 1999):
  - 30% av sin tid til individuelt arbeid,
  - 50% av sin tid til å arbeide med en annen person, og
  - 20% av sin tid til å arbeide med to eller flere personer
- Det er vanlig å bruke 30-70% av totaltiden til å lokalisere og rette feil (Gibson – Managing Computer Projects)

PP – Definisjon

- **To utviklere** (partnere) arbeider på **samme oppgave** (v.h.a. **én skjerm og ett tastatur)**
- Involverer ikke bare **koding**, men også andre faser av systemutviklingsprosessen som f.eks. **design** og **testing**
- "Stand-alone" praksis (uavhengig av type prosess – ikke bare passende for XP)
- Burde heller vært kalt: **Parutvikling**!

PP – Roller

- **Sjåfør**:
  - Sitter med keybordet eller tegner ned designet
- **Navigatør (kartleser)**:
  - Observerer aktivt arbeidet til sjåføren
  - Ser etter taktiske og strategiske feil
  - Tenker og søker etter alternativer
  - Skriver ned ting man må huske å gjøre
  - Slår opp i referanser (manualer, web-sider osv.)

PP – Roller

- Bytting av roller regelmessig (sjåfør vs navigatør)
- Rotere partnere (innad i teamet) → Spre kunnskap/informasjon

Fordeler ved bruk av PP (sammenliknet med individuell programmering)

- Redusert "time-to-market"
- Reduserte utviklingskostnader
- Økt kvalitet (kode og system)
- Økt informasjons- og kunnskapsoverføring
- Redusert kostnad ved trening av nye ansatte
- Redusert kostnad ved rekruttering av nye personer til pågående prosjekter
- Forbedret kommunikasjon mellom utviklerne
- Forbedret tillit mellom utviklerne
- Høyere fokus på oppgaven som skal løses
- Kontinuerlig læring
- Økt motivasjon (trivsel)
- Redusert risiko for å miste nøkkelutviklere
- Enkel teknikk å lære og bruke

Kostnader ved bruk av PP (sammenliknet med indiviuell programmering)

- PP er bare kontinuerlig kodeinspeksjon med lite eller ingen forbedret kvalitet i produktet som utvikles sammenliknet med individuell programmering og kodeinspeksjon
- PP medfører dobbel kostnad (to personer på samme oppgave uten ekstra kvalitet)
- Vanskelig teknikk å innføre og bruke pga stadige forstyrrelser (møter, telefoner, osv.)
- Vanskelig å finne gode og effektive par
- Komplekse oppgaver løses best alene!

Pair Programming Experiment (PPE)

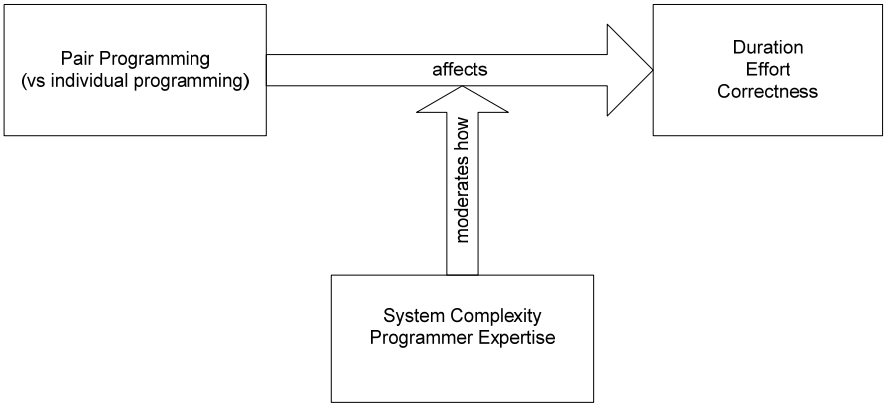
Previous controlled experiments on PP

Exp.	Subj.	N (Ind)	N (Pair)	Tasks	Duration	Effort	Correct
Nosek	Prof.	5	5	Initial Dev.	-29 %	42 %	33 %
Williams	Stud.	13	14	Initial Dev.	-43 %	15 %	15 %
Nawrocki	Stud.	5	5	Initial Dev.	0 %	100 %	0 %
Lui	Prof.	5	5	Multiple Choice	-52 %	-4 %	N/A
Muller	Stud.	19	23	Initial Dev.+ review	-44 %	13 %	29 %
				incl. Rework	-47 %	7 %	N/A

- Previous studies on pair programming have been on initial development tasks
  - For the first time we evaluate PP in the context of software maintenance
- No studies on the moderating effects of the expertise of the programmers or the complexity of the systems to be developed

47

Conceptual Research Model

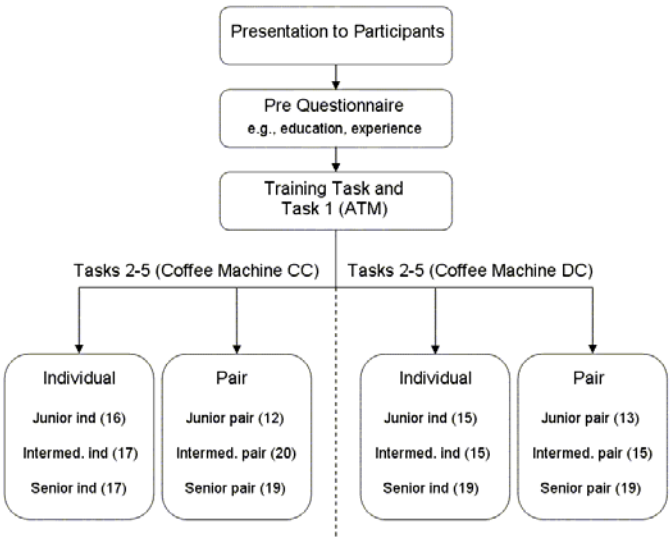


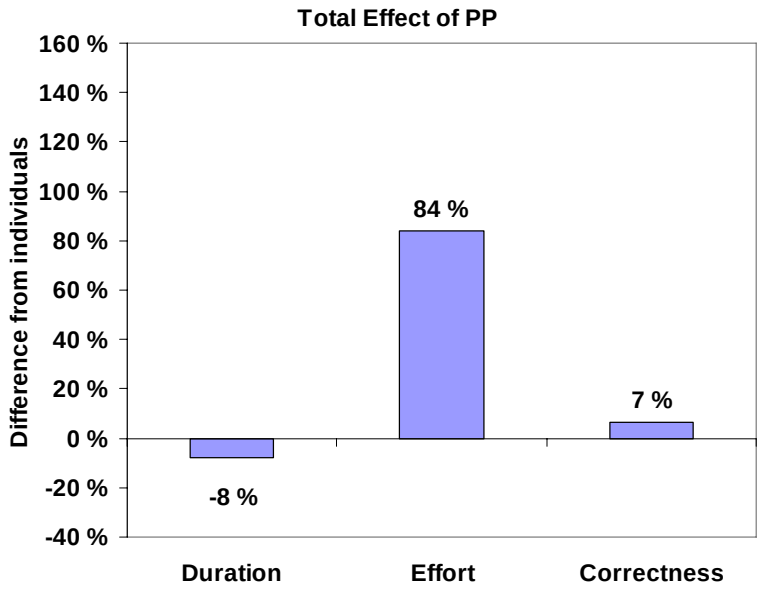
48

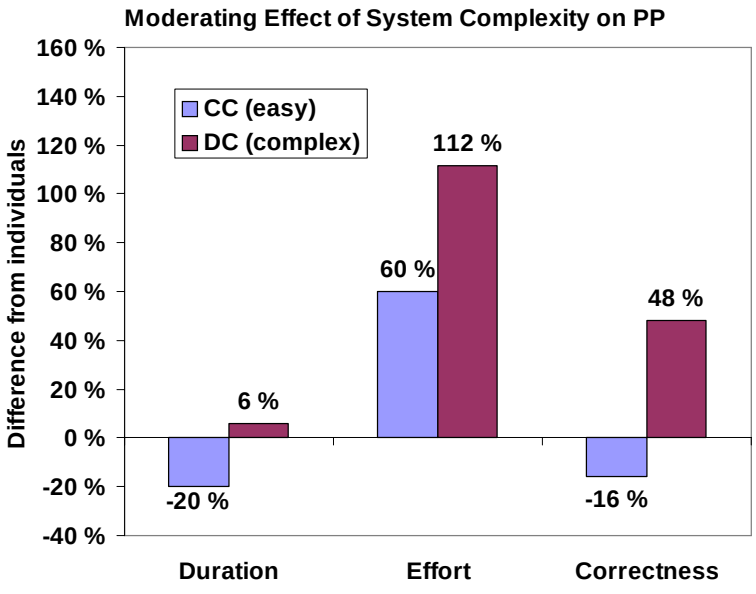
The Experiment

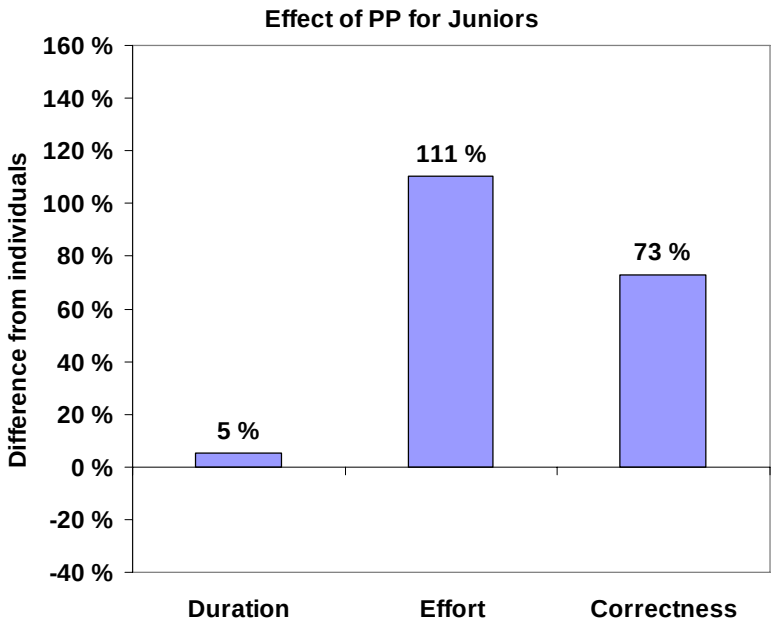
- 295 junior, intermediate and senior professional Java consultants from 29 companies were paid to participate (one work day)
- 99 individuals (conducted in 2001/2002)
- 98 pairs (conducted in 2004/2005)
  - Norway: 41
  - Sweden: 28
  - UK: 29
- The pairs and individuals performed the same Java maintenance tasks on either:
  - a ”simple” system (centralized control style), or
  - a ”complex” system (delegated control style)
- We measured duration (elapsed time), effort (cost) and correctness of their solutions

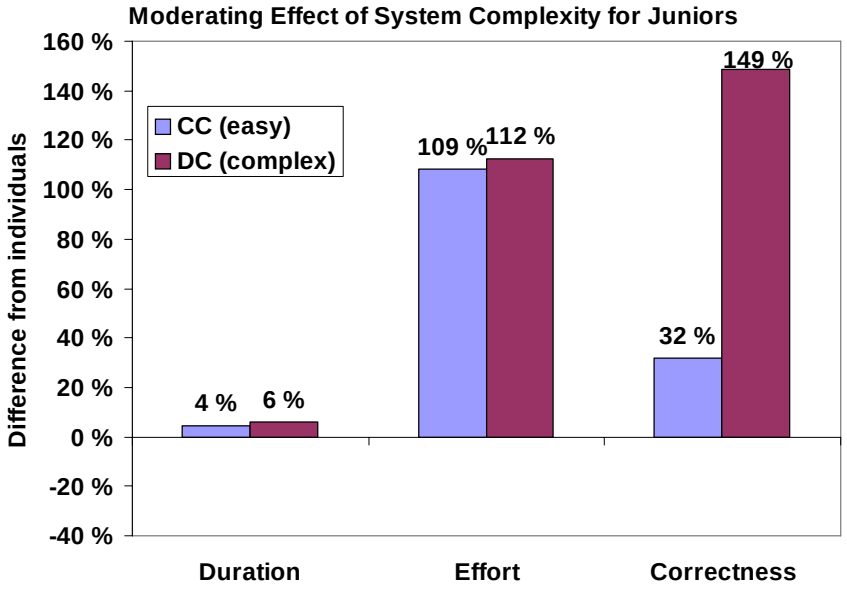
Experiment Design

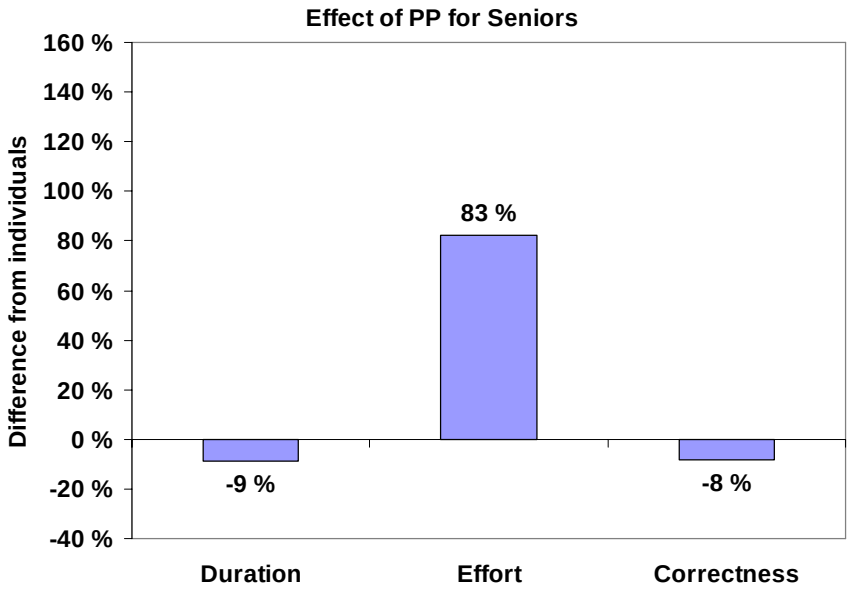



Total Effect of PP
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
84 %
7 %
-8 %
Duration
Effort
Correctness

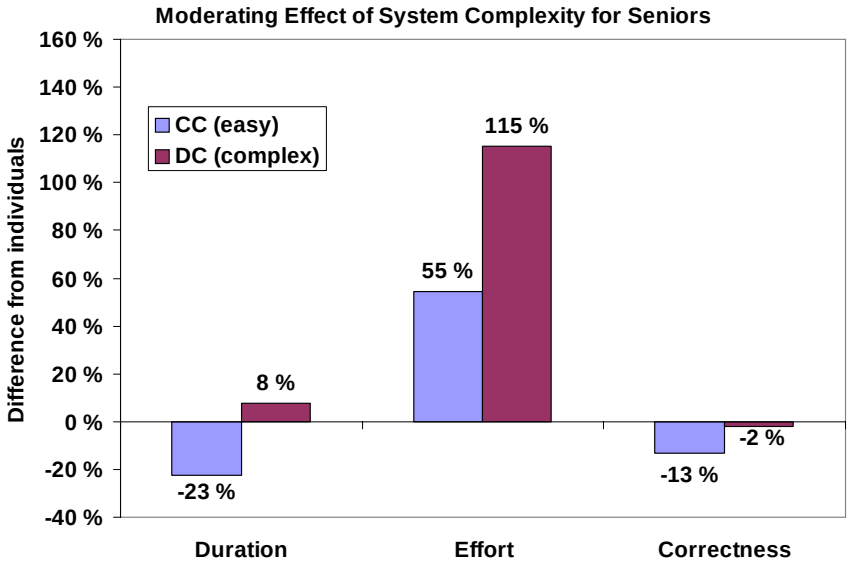

Moderating Effect of System Complexity on PP
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
CC (easy)
DC (complex)
112 %
60 %
48 %
6 %
-20 %
-16 %
Duration
Effort
Correctness


Effect of PP for Juniors
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
111 %
73 %
5 %
Duration
Effort
Correctness

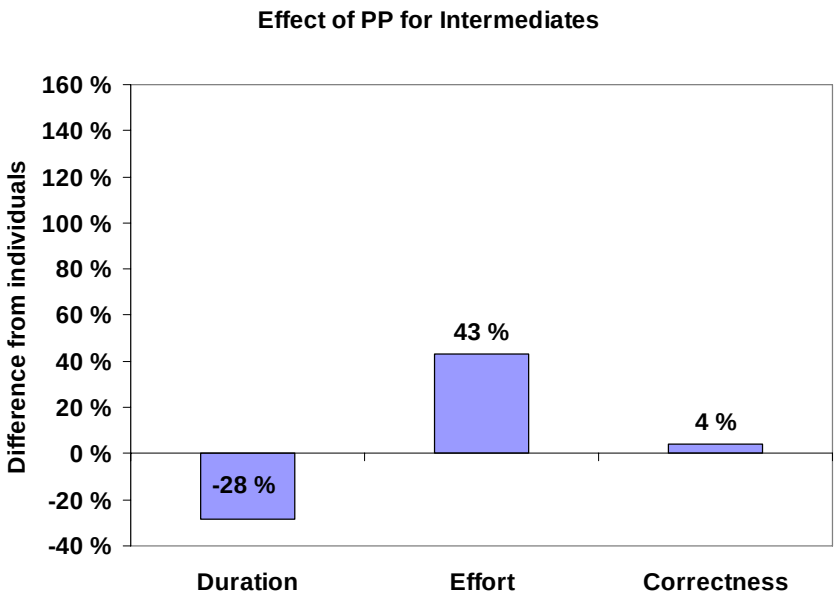

Moderating Effect of System Complexity for Juniors
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
CC (easy)
DC (complex)
4 %
6 %
109 %
112 %
32 %
149 %
Duration
Effort
Correctness


Effect of PP for Seniors
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
83 %
-9 %
-8 %
Duration
Effort
Correctness

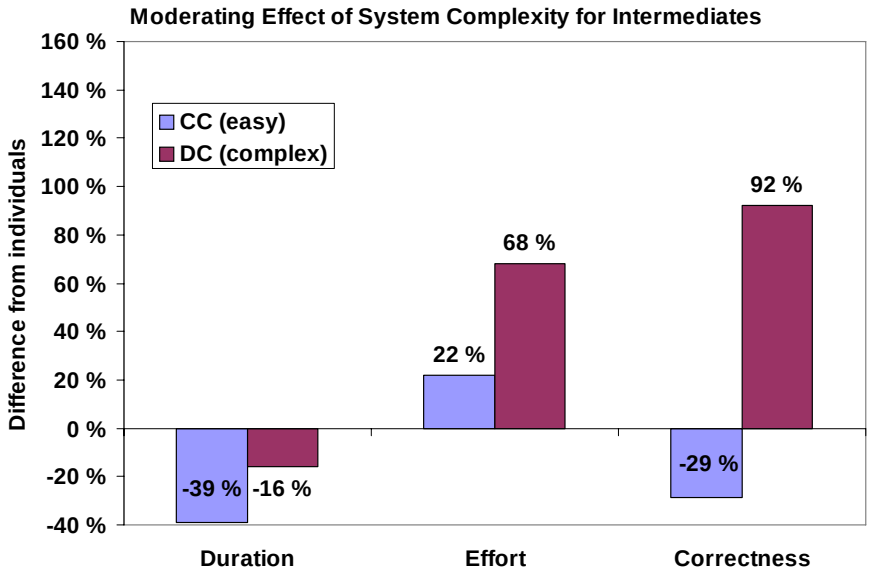
55


Moderating Effect of System Complexity for Seniors
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
CC (easy)
DC (complex)
115 %
55 %
8 %
-2 %
-23 %
-13 %
Duration
Effort
Correctness

56


Effect of PP for Intermediates
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
-28 %
43 %
4 %
Duration
Effort
Correctness

57


Moderating Effect of System Complexity for Intermediates
Difference from individuals
160 %
140 %
120 %
100 %
80 %
60 %
40 %
20 %
0 %
-20 %
-40 %
CC (easy)
DC (complex)
-39 %
-16 %
22 %
68 %
-29 %
92 %
Duration
Effort
Correctness

58

Junior developers benefited most from pair programming, in particular on the complex system!

- The effect of pair programming on duration, effort and correctness depends on the expertise of the developers and the type of task to be solved:
  - The benefits of PP is reduced with increasing programming skills of the individuals
  - The benefits of of PP is reduced with decreasing task complexity
- Pair programming requires significantly more effort than individual programming (regardless of programmer category)
- Juniors:
  - PP significantly improves correctness
  - The effect of PP on *correctness* is highest for the tasks based on the delegated control style
  - Junior pairs obtain almost the same correctness as intermediate and senior pairs (≈ 80 percent correct)
- Seniors:
  - No clear benefits of pair programming

Expertise versus Task Complexity

- On which type of tasks are you a junior, intermediate or senior developer?
  - Programming skills
  - Domain knowledge
- Wicked problems
  - Many solutions!
- Cognitive psychology (group dynamics)
  - Social facilitation
  - Social labouring

Case study: Innføring av PP

- Vanskelig å finne ”uforstyrret” tid
  - Lag kjernetid for parprogrammering, f.eks. fra kl. 10-14
  - Aktivt samarbeid krever regelmessige pauser
- Alle oppgaver egner seg ikke for PP?
  - Bestem graden av parprogrammering (og partnere) gjennom regelmessige og korte stand-up møter
  - Paret deler samme kontorplass og har kontinuerlig dialog
- Kan man droppe QA-rutinene når man benytter PP?
  - Nei, man kan ikke gjøre enten PP eller individuell programmering (riktig spørsmål: når og hvordan skal man benytte PP?)
  - Se opp for at dokumentering av kode etc. kan ”glemmes” ved PP
  - Hvis ”test-first” praktiseres: Lag testene i par → implementer individuelt → gjør QA-rutiner i par til slutt

Takk for oppmerksomheten ☺