Practical XP Experiences

Jari Vanhanen
SoberIT
http://www.soberit.hut.fi/sems/

Presentation Outline

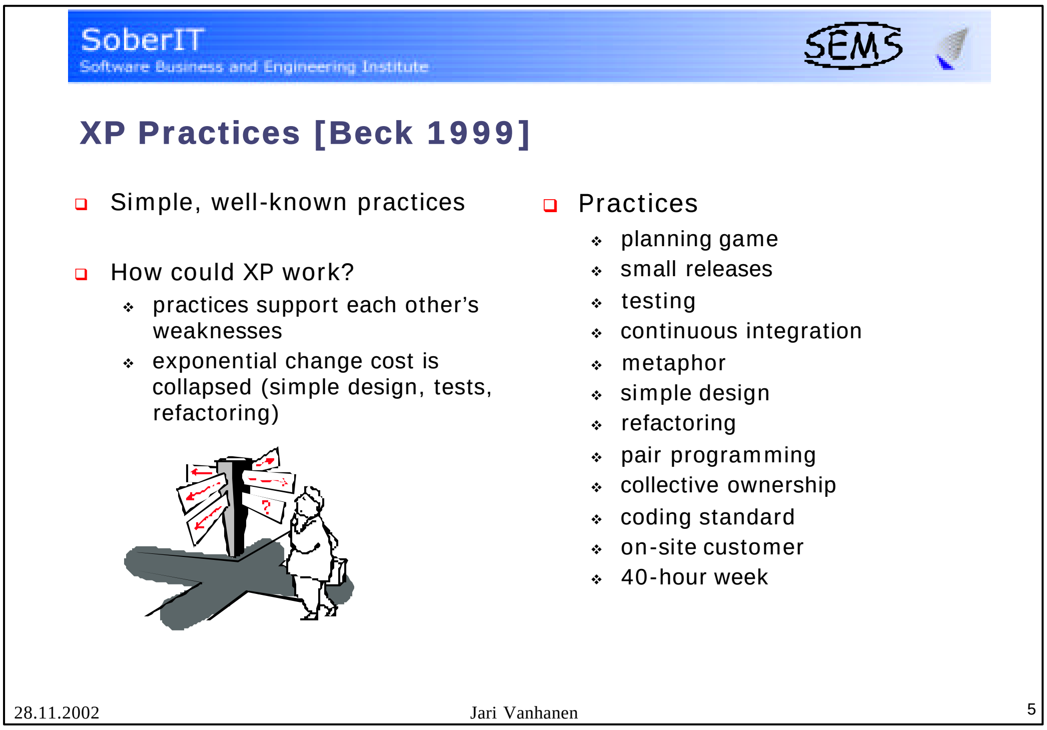
- Introduction
  - the context of the cases
- Case 1
- Case 2
- Summary

Case Descriptions

- Two projects from ”T-76.115 Software Project”–course at HUT
- Based on
  - numerous informal discussions
    - mentoring
  - reported data
    - realized hours per task
    - LOC
  - final reports
    - analysis of experimented XP practices
- Focus on practical experiences gained from the used XP practices

T-76.115 Software Project Course

- Complete software project
- Real customers
- 7 persons in each group
  - 3+ year computer science students
  - most have work experience
- Fixed schedule and effort
  - 7 months
  - 200h per person
  - ~8hrs/week/person
- Fixed process framework
  - traditionally RUP
    - customized by the projects
  - XP pilots 2001-02
    - XP complemented with some mandatory reporting and documentation

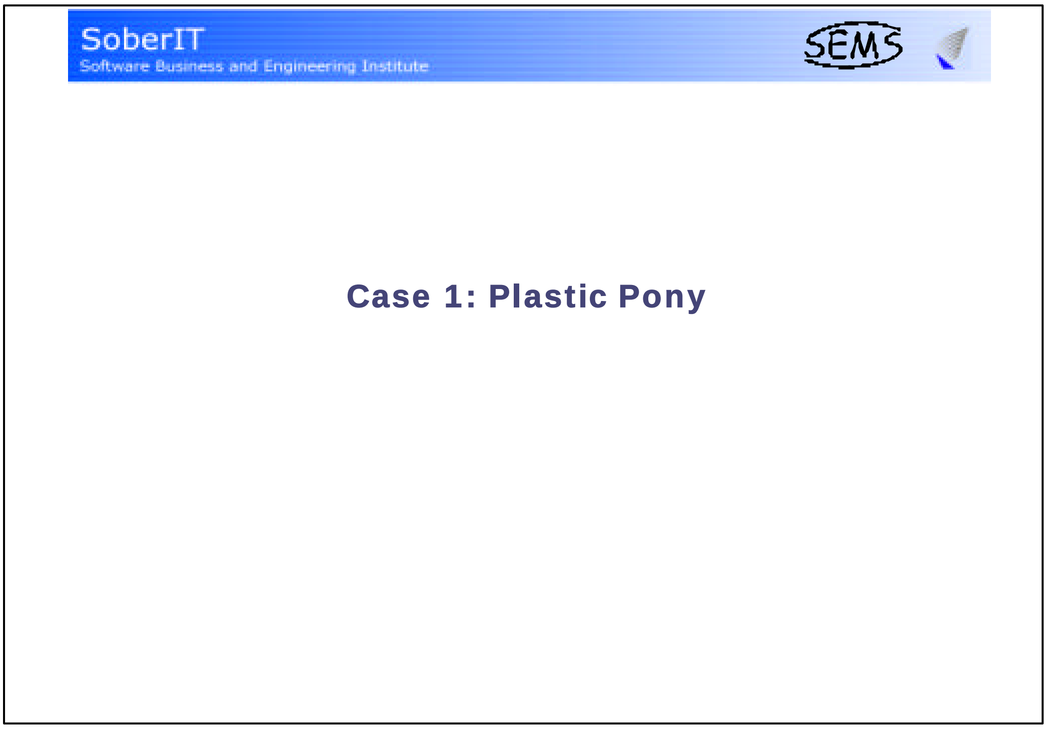


SoberIT
Software Business and Engineering Institute

SEMS

XP Practices [Beck 1999]

- Simple, well-known practices
- How could XP work?
  - practices support each other's weaknesses
  - exponential change cost is collapsed (simple design, tests, refactoring)
- Practices
  - planning game
  - small releases
  - testing
  - continuous integration
  - metaphor
  - simple design
  - refactoring
  - pair programming
  - collective ownership
  - coding standard
  - on-site customer
  - 40-hour week



SoberIT
Software Business and Engineering Institute

SEMS

Case 1: Plastic Pony

Case 1: Overview

- Project
  - graphical www-sitemap editor for Accenture
  - 1500 hours
  - 7 persons
- Technologies
  - Java (JFC, JGraph), XML
- Development tools
  - JDK, JUnit, CVS, Ant
- Project Management tools
  - forced by the course
    - MS Project
    - time reporting system
    - metrics visualization tool
  - Wiki
    - web collaboration tool
- No previous XP experience

Case 1: Unit Testing

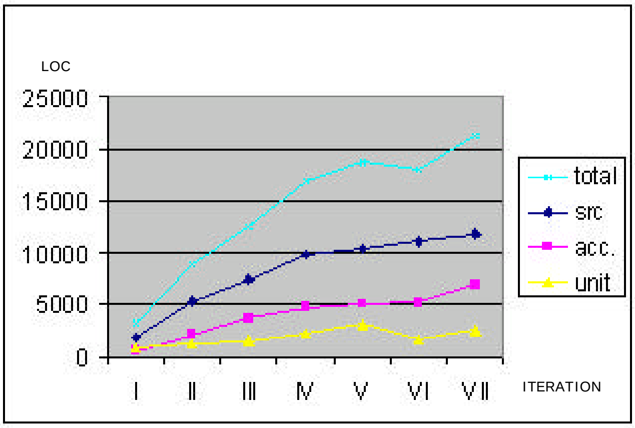
- Adoption goal
  - strictly XP
- Tests were written but not before the real code
  - test-first hard with experimental, continuously changing code
- Confidence on tests improved as the project progressed
  - new tests for found bugs
    - replaces bug reporting
- JUnit
  - useful and working tool
- JFCUnit
  - good concept
  - buggy implementation
- Most important benefits
  - bugs caused by refactoring found soon
  - own new code verified immediately
- Not much aid for communication
  - code comments, pair programming, and coding standard more important

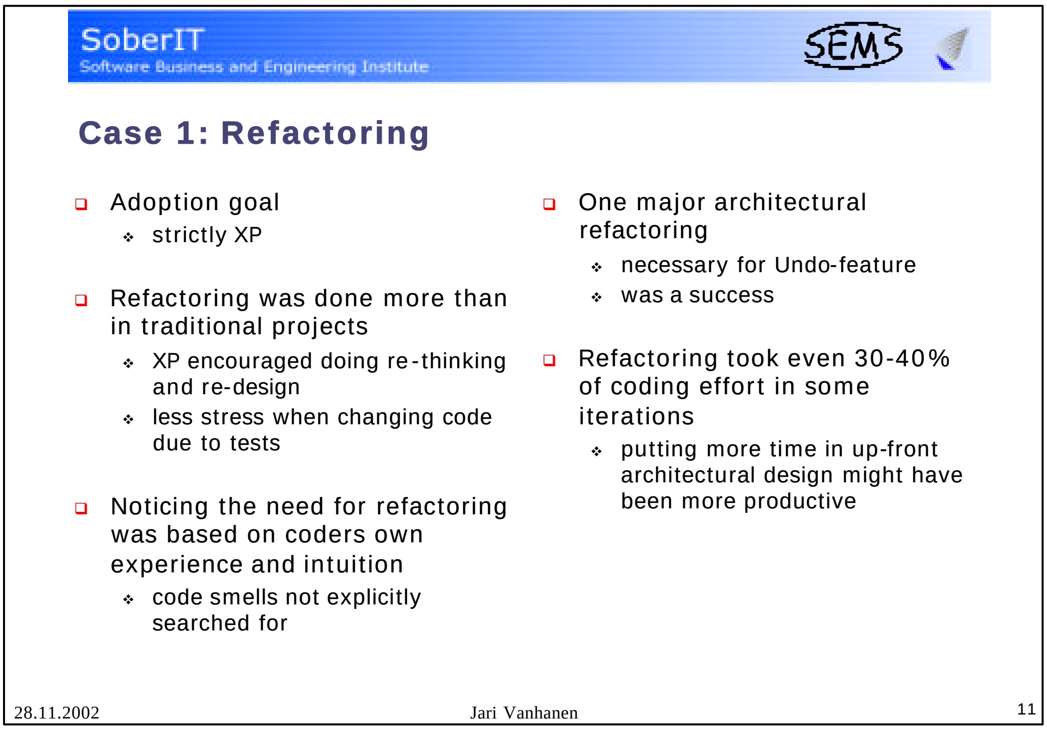
Case 1: Acceptance Testing

- Adoption goal
  - strictly XP
- Developers specified test cases, customer accepted them
  - the gap between customers real expectations and tests narrowed using trial-and-error method
- All test cases automated
  - GUI testing easier than expected
    - no previous experience
    - 25% of programming effort in early iterations
- Acceptance tests survived a major architectural refactoring of code

Case 1: Amount of Test Code

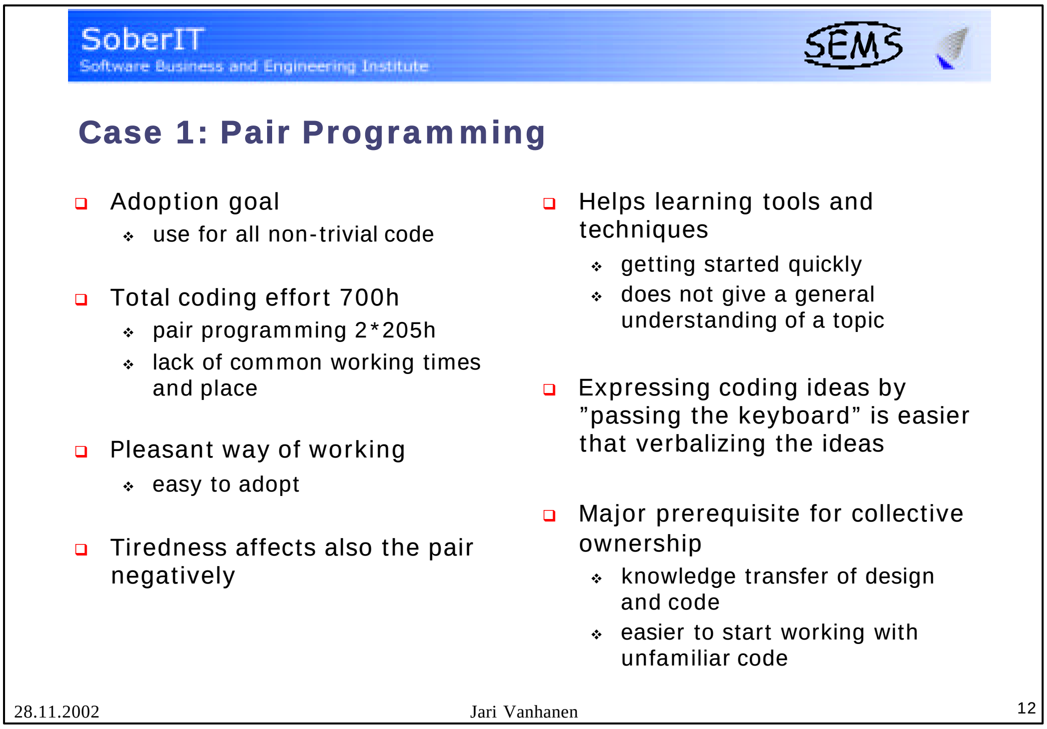
- Final release
  - real code 56%
  - acceptance tests 32%
  - unit tests 12%
- In the 6th iteration refactoring invalidated lots of unit tests
  - new architecture was hard to unit test
  - tested using old acceptance tests





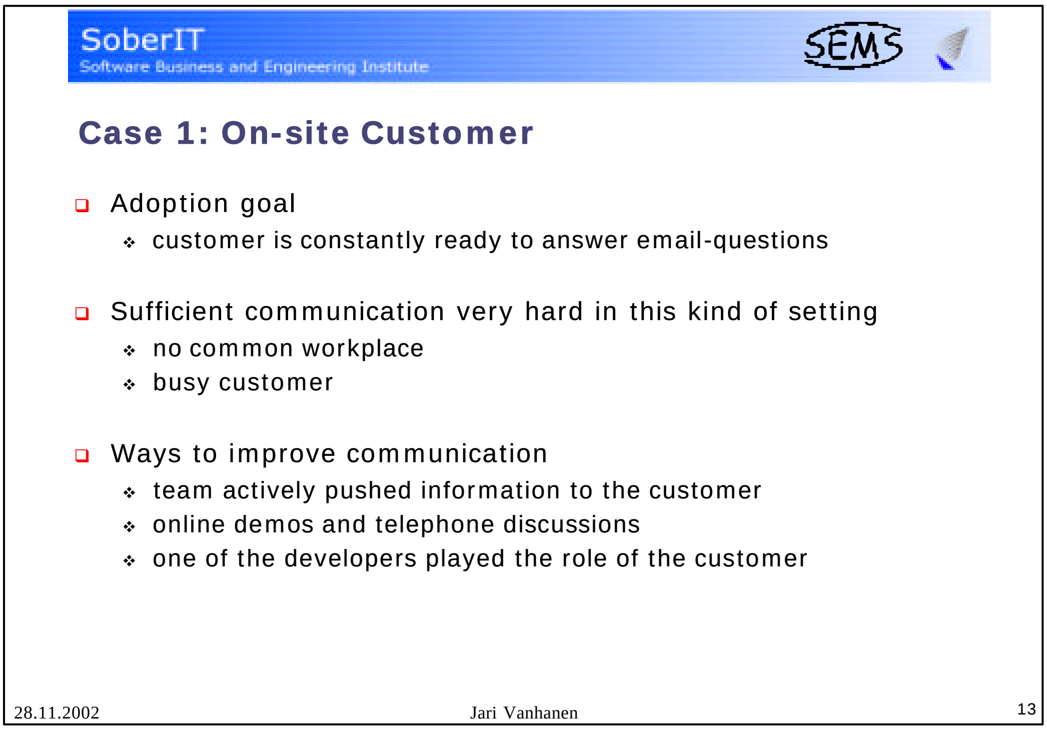
Case 1: Refactoring

- Adoption goal
  - strictly XP
- Refactoring was done more than in traditional projects
  - XP encouraged doing re-thinking and re-design
  - less stress when changing code due to tests
- Noticing the need for refactoring was based on coders own experience and intuition
  - code smells not explicitly searched for
- One major architectural refactoring
  - necessary for Undo-feature
  - was a success
- Refactoring took even 30-40% of coding effort in some iterations
  - putting more time in up-front architectural design might have been more productive



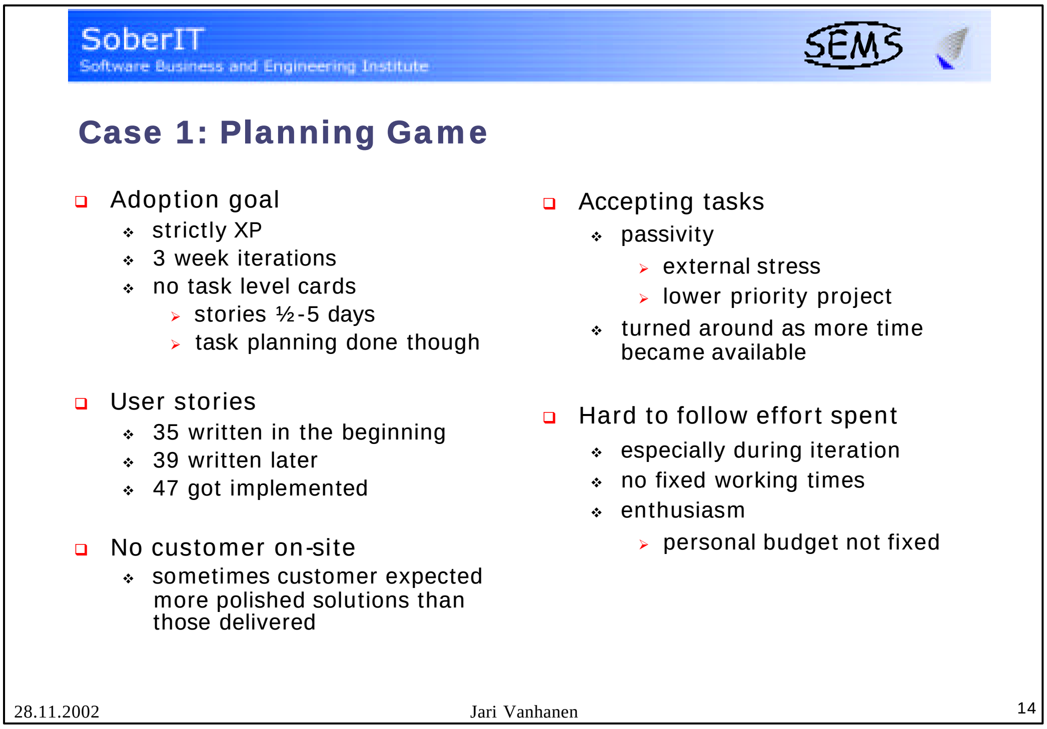
Case 1: Pair Programming

- Adoption goal
  - use for all non-trivial code
- Total coding effort 700h
  - pair programming 2*205h
  - lack of common working times and place
- Pleasant way of working
  - easy to adopt
- Tiredness affects also the pair negatively
- Helps learning tools and techniques
  - getting started quickly
  - does not give a general understanding of a topic
- Expressing coding ideas by ”passing the keyboard” is easier that verbalizing the ideas
- Major prerequisite for collective ownership
  - knowledge transfer of design and code
  - easier to start working with unfamiliar code



Case 1: On-site Customer

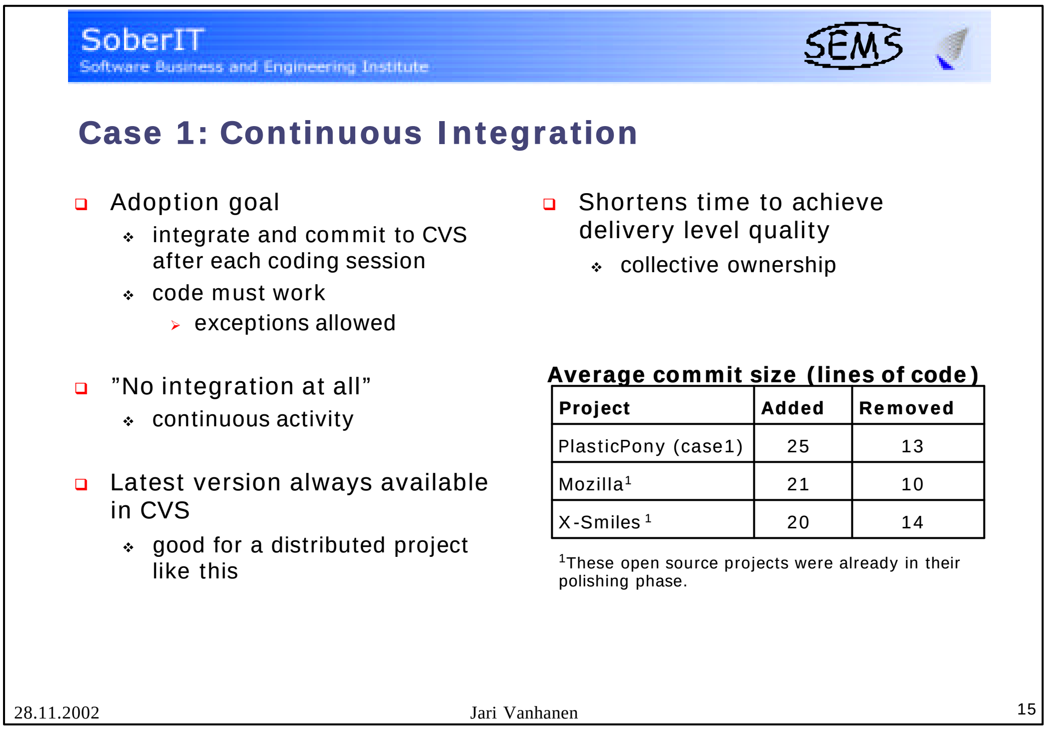
- Adoption goal
  - customer is constantly ready to answer email-questions
- Sufficient communication very hard in this kind of setting
  - no common workplace
  - busy customer
- Ways to improve communication
  - team actively pushed information to the customer
  - online demos and telephone discussions
  - one of the developers played the role of the customer



Case 1: Planning Game

- Adoption goal
  - strictly XP
  - 3 week iterations
  - no task level cards
    - stories ½-5 days
    - task planning done though
- User stories
  - 35 written in the beginning
  - 39 written later
  - 47 got implemented
- No customer on-site
  - sometimes customer expected more polished solutions than those delivered

- Accepting tasks
  - passivity
    - external stress
    - lower priority project
  - turned around as more time became available
- Hard to follow effort spent
  - especially during iteration
  - no fixed working times
  - enthusiasm
    - personal budget not fixed



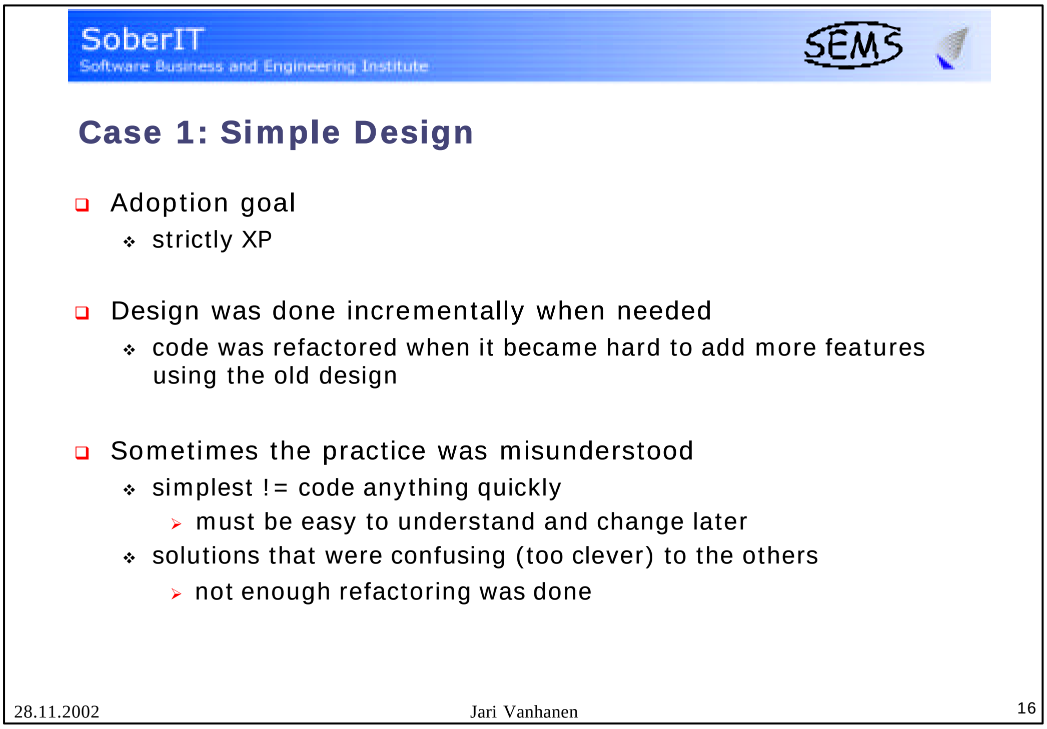
Case 1: Continuous Integration

- Adoption goal
  - integrate and commit to CVS after each coding session
  - code must work
    - exceptions allowed
- "No integration at all"
  - continuous activity
- Latest version always available in CVS
  - good for a distributed project like this
- Shortens time to achieve delivery level quality
  - collective ownership

Average commit size (lines of code)

Project	Added	Removed
PlasticPony (case1)	25	13
Mozilla[1]	21	10
X-Smiles [1]	20	14

[1] These open source projects were already in their polishing phase.



Case 1: Simple Design

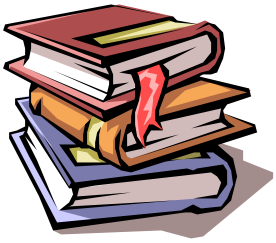
- Adoption goal
  - strictly XP
- Design was done incrementally when needed
  - code was refactored when it became hard to add more features using the old design
- Sometimes the practice was misunderstood
  - simplest != code anything quickly
    - must be easy to understand and change later
  - solutions that were confusing (too clever) to the others
    - not enough refactoring was done

Case 1: Other Practices

- Small releases
  - two releases
  - seven three-week iterations
  - positive experience
    - one cornerstone of XP
- Metaphor
  - quite technical
    - pages, processes, transitions, …
    - technical customer
  - communication tool
- Collective ownership
  - most used in refactoring
  - everyone did not reach equal familiarity with all code
    - short project
- 40-hour-week/sustainable pace
  - not applicable/not used

Case 1: Product Documentation

- Requirements specification
  - 1 page overview of the system
  - user stories
- Source code
  - unit tests
  - acceptance tests



Case 1: Project Evaluation

- Customer very satisfied
  - results did not exactly match original plans
  - results matched the **current needs in the end of the project**
- Overall
  - (one of) the best projects in the course (24 projects)
  - winner of the course’s Quality Award
- Group contained very skilled people
  - the role of used process in the success?

Case 2: RAID

Case 2: Overview

- Project
  - defect tracking system for SoberIT/HUT
  - 7 persons
  - 1200 hours
- Technologies
  - J2EE, JSP
- Development tools
  - JDK, CVS, Junit, Ant
- Project management tools
  - forced by the course
    - MS Project
    - time reporting system
    - metrics visualization tool
- No previous XP experience

Case 2: Practices

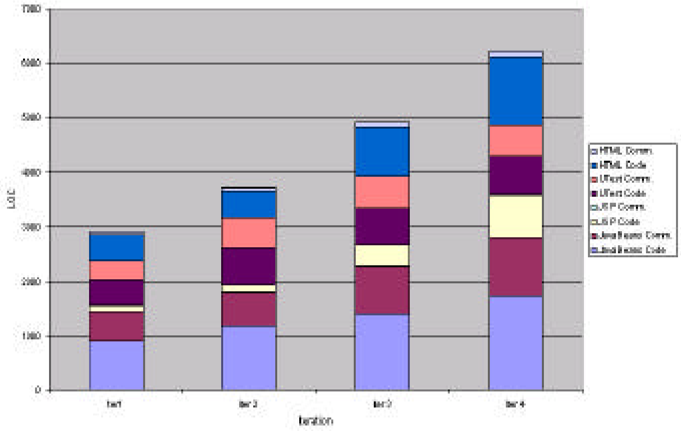
- Planning game
  - customer wrote a lot of stories early
    - 88 user stories
    - too detailed for planning
    - too large for a small project
  - was difficult in the beginning of the project
    - new way of planning
    - unfamiliar technology (J2EE)
    - dependencies between stories were problematic
  - later the practice worked well and was effective in controlling project’s direction
- Small releases
  - good visibility of progress
  - demos anytime
  - earlier releases did not have minimum amount of valuable functionality
    - small project
- On-site customer
  - physically not available
  - quite good communication
    - but mainly with a sub team only

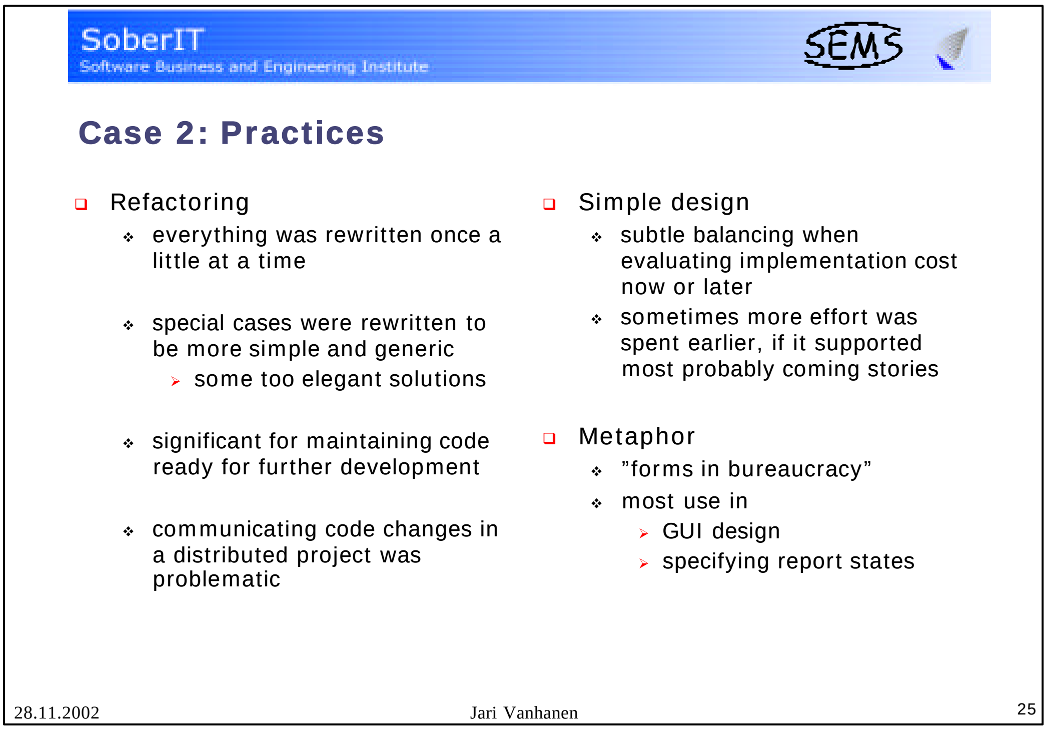
Case 2: Testing

- Unit tests were useful for finding bugs during development and especially while refactoring
- Writing tests before the code was considered a profitable practice
  - however, it was neglected often when it was hard to come up with a good design without building small spikes
- Customer specified acceptance tests
  - group run them at the end of each iteration
  - external testers run the test once
    - a couple of new issues were raised
    - testing by customer herself would have been important
  - tests should have been updated during development

Case 2: Amount of Test Code

- Final release
  - 6100 LOC
  - Unit tests 19%
  - Real code 81%



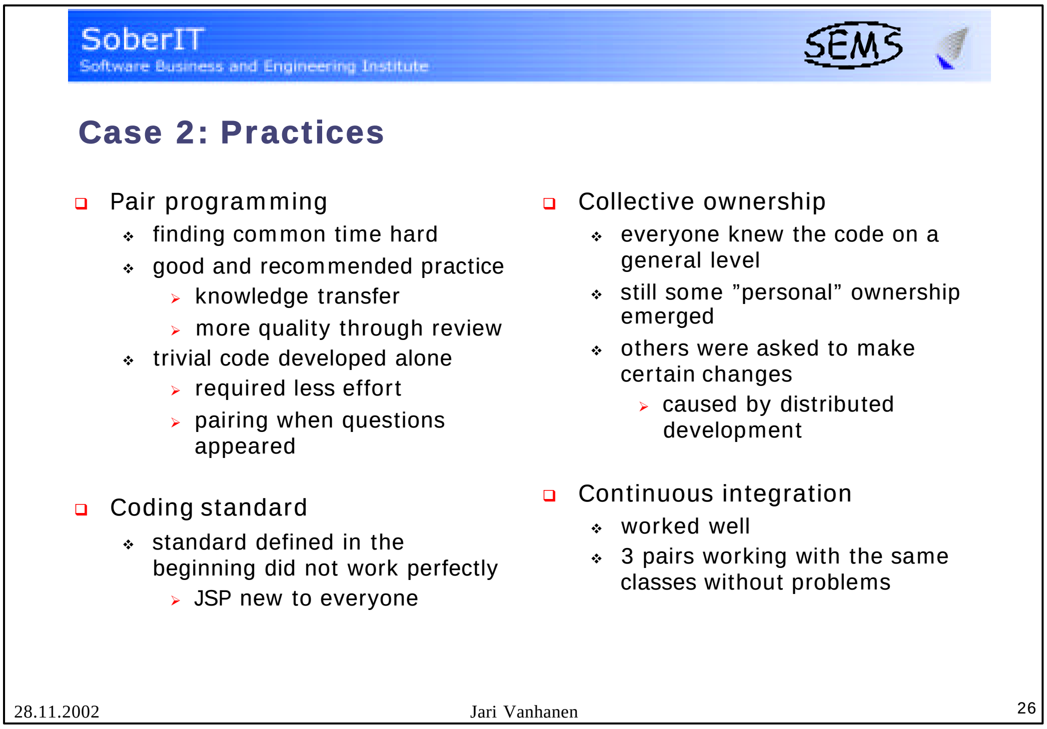


SoberIT
Software Business and Engineering Institute

SEMS

Case 2: Practices

- Refactoring
  - everything was rewritten once a little at a time
  - special cases were rewritten to be more simple and generic
    - some too elegant solutions
  - significant for maintaining code ready for further development
  - communicating code changes in a distributed project was problematic
- Simple design
  - subtle balancing when evaluating implementation cost now or later
  - sometimes more effort was spent earlier, if it supported most probably coming stories
- Metaphor
  - ”forms in bureaucracy”
  - most use in
    - GUI design
    - specifying report states



SoberIT
Software Business and Engineering Institute

SEMS

Case 2: Practices

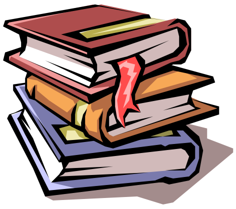
- Pair programming
  - finding common time hard
  - good and recommended practice
    - knowledge transfer
    - more quality through review
  - trivial code developed alone
    - required less effort
    - pairing when questions appeared
- Coding standard
  - standard defined in the beginning did not work perfectly
    - JSP new to everyone
- Collective ownership
  - everyone knew the code on a general level
  - still some ”personal” ownership emerged
  - others were asked to make certain changes
    - caused by distributed development
- Continuous integration
  - worked well
  - 3 pairs working with the same classes without problems

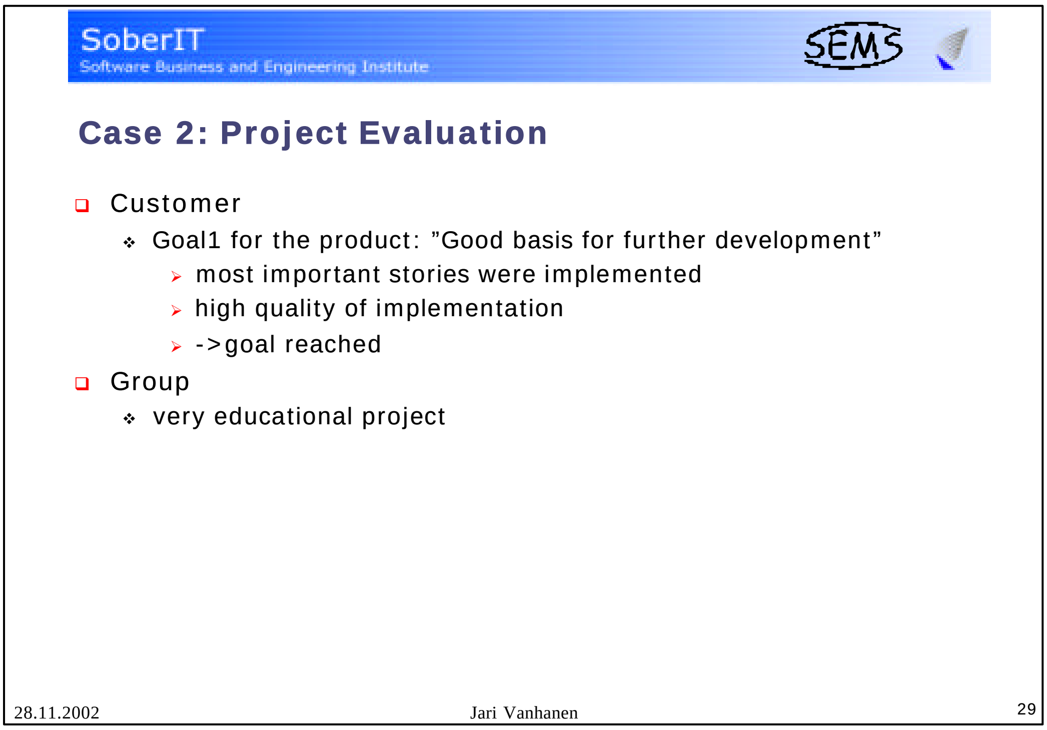
Case 2: General Experiences

- Favorable characteristics for XP project
  - small
  - not too complicated
  - vague requirements
- Most XP practices felt natural and worked well in this project
- A pleasant experience and we are ready to try it again
- XP does not work well with a distributed team
  - same room and common working times required
- Work should be more intensive
  - now about 8hrs/week
  - takes time to restart work

Case 2: Product Documentation

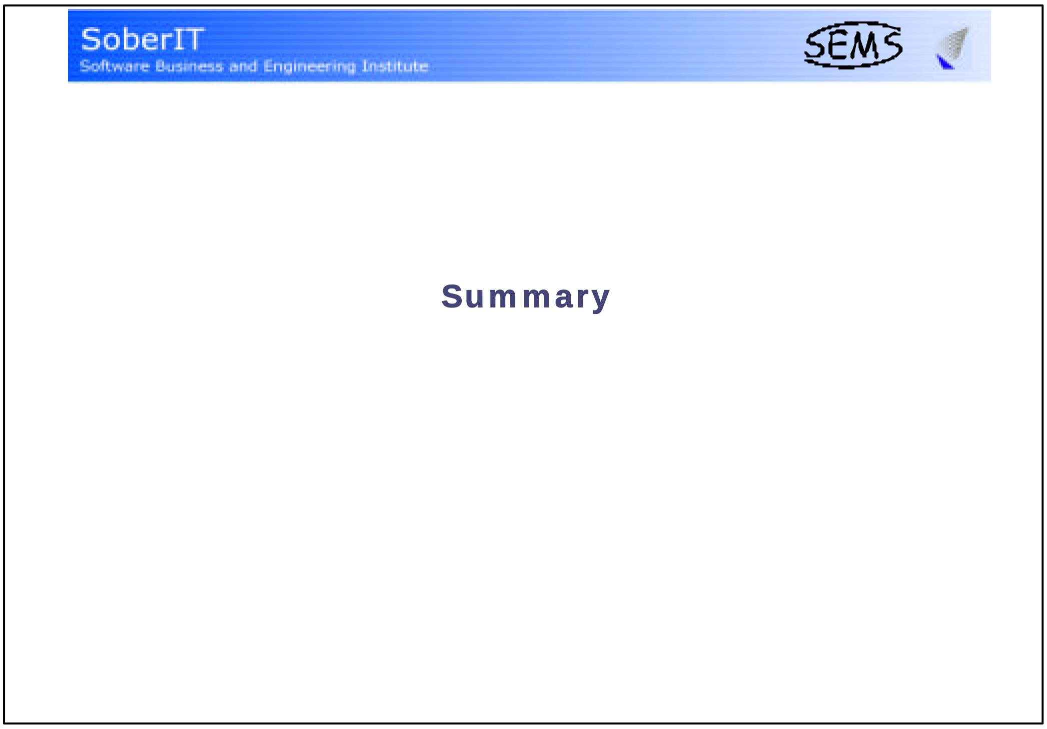
- Technical overview
  - 3 pages
- Installation guide
- User stories
- Acceptance tests
- Acceptance test report
- Open bugs and development ideas



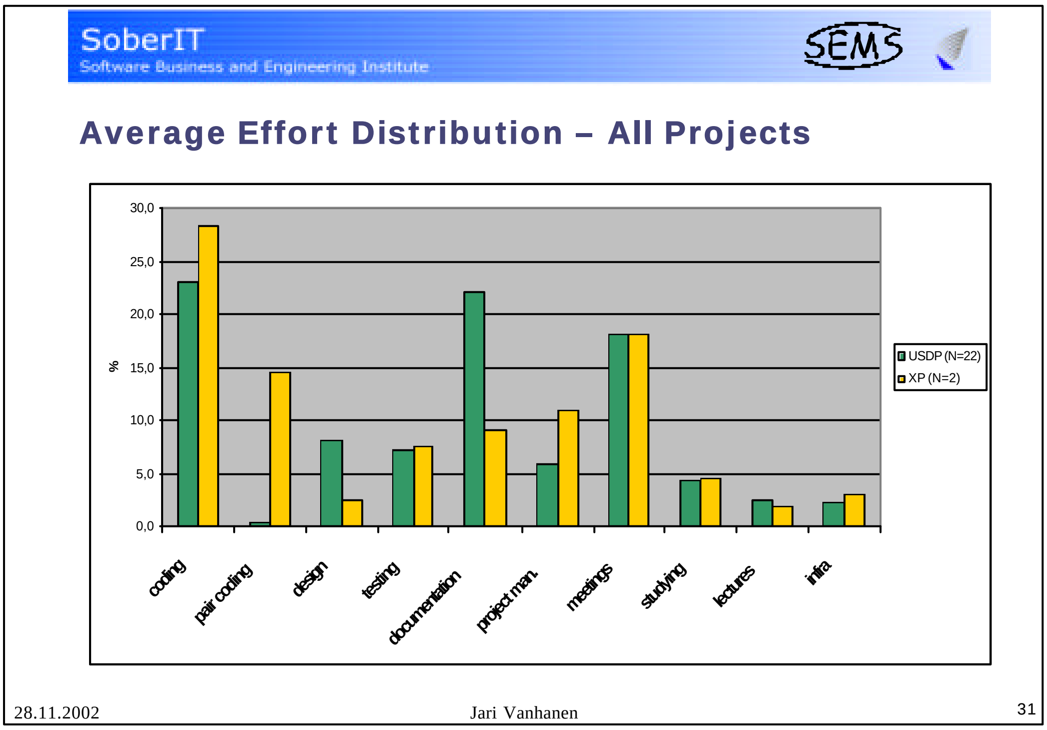


Case 2: Project Evaluation

- Customer
  - Goal1 for the product: ”Good basis for further development”
    - most important stories were implemented
    - high quality of implementation
    - ->goal reached
- Group
  - very educational project



Summary

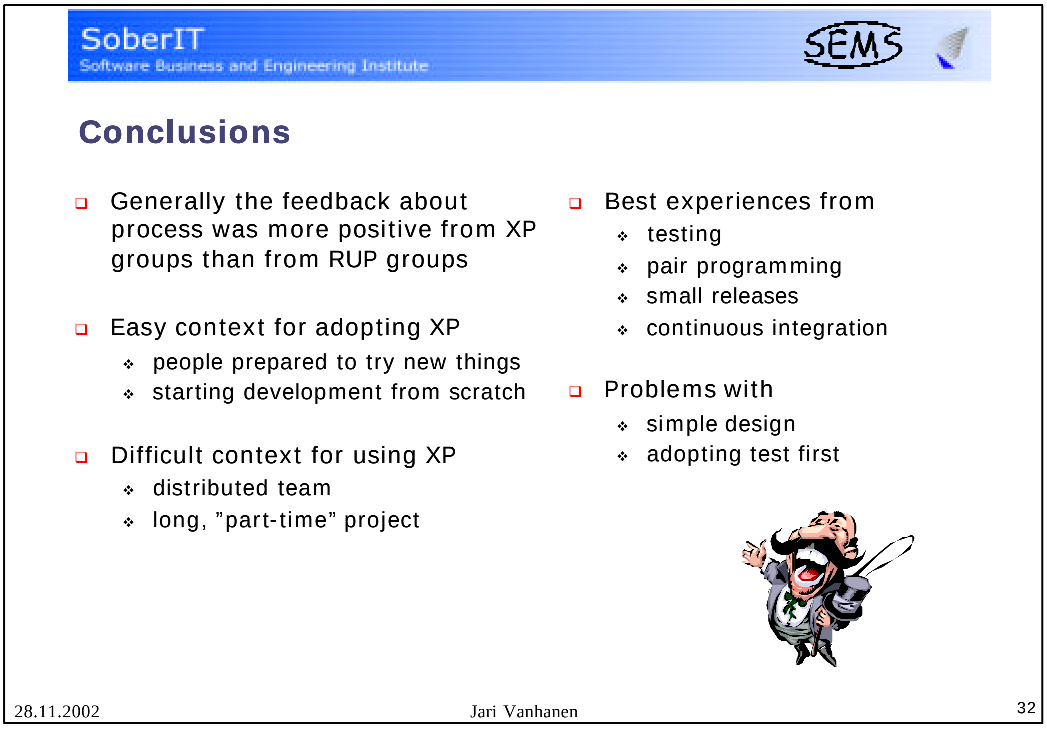


SoberIT
Software Business and Engineering Institute

SEMS

Average Effort Distribution – All Projects

28.11.2002 Jari Vanhanen 31



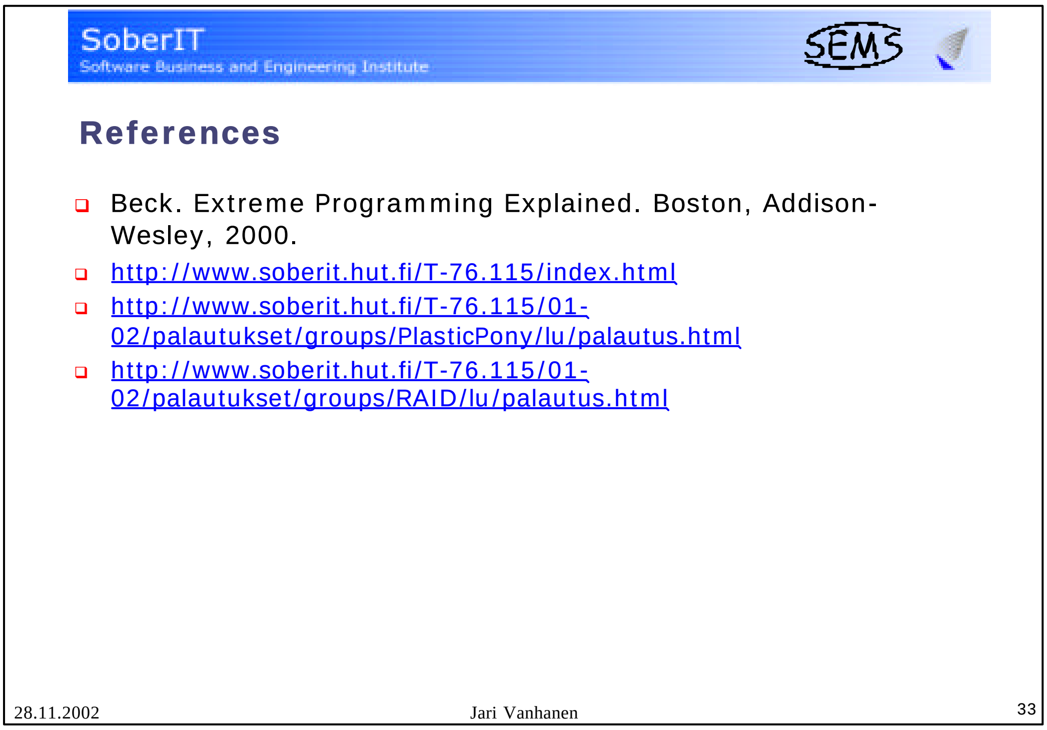
SoberIT
Software Business and Engineering Institute

SEMS

Conclusions

- Generally the feedback about process was more positive from XP groups than from RUP groups
- Easy context for adopting XP
  - people prepared to try new things
  - starting development from scratch
- Difficult context for using XP
  - distributed team
  - long, ”part-time” project
- Best experiences from
  - testing
  - pair programming
  - small releases
  - continuous integration
- Problems with
  - simple design
  - adopting test first

28.11.2002 Jari Vanhanen 32



References

- Beck. Extreme Programming Explained. Boston, Addison-Wesley, 2000.
- http://www.soberit.hut.fi/T-76.115/index.html
- http://www.soberit.hut.fi/T-76.115/01-02/palautukset/groups/PlasticPony/lu/palautus.html
- http://www.soberit.hut.fi/T-76.115/01-02/palautukset/groups/RAID/lu/palautus.html