CSCW and Software Engineering

Chapter 4: Computer Supported Cooperative Software Engineering
Winter Semester 2005/2006

Dr.-Ing. Stefan Werner
University Duisburg-Essen
Institute for Multimedia and Software Engineering

Content

1. Introduction to CSCW & Software Engineering
   1.1 Motivations
   1.2 History and basic definitions
   1.3 Interpretation of CSCW
   1.4 CSCW Examples
   1.5 Introduction to Distributed Systems
   1.6 History of Software Engineering
   1.7 Introduction to CSCSE

2. Virtual teams
   2.1 Introduction
   2.2 Virtual team organisation
      2.2.1 Virtual team structures
      2.2.2 Team creation
      2.2.3 Team performance
   2.3 Virtual team communications
   2.4 Technology platforms

3. CSCW – A discipline to support teams-
   3.1 Classifications of CSCW-Systems
   3.2 Groups and Group Processes
   3.3 Group Communication
   3.4 Structures of Asynchronous Group Interactions
   3.5 Characteristics of Synchronous Cooperations
   3.6 Shared Context
   3.7 Groupware Awareness
   3.8 Groupware Architectures
   3.9 Concurrency Control

4. CSCSE – A team discipline-
   4.1 Cooperative Software Engineering
   4.2 Software Process Models for distributed teams

Dr.-Ing. Stefan Werner
University of Duisburg-Essen
Institute for Multimedia and Software Engineering
4.1 Cooperative Software Engineering
Introduction (1/2)

- The amount of work achieved through cooperative work lies at around 70% of the total working time of a Software Developer.

- A number of recent studies have highlighted the importance of group working within software engineering.
  - Perry et al., 1994] tried to identify how programmers spend their time
    - Results: Over half the time was spent in interactive activities other than coding, and a significant part of the programmers’ day was spent interacting in various ways with co-workers.
  - Panko, 1996] Development of project team profiles
    - Results: Over half the projects had at least one member from another site, and many had several.

4.1 Cooperative Software Engineering
Introduction (2/2)

- CSCW and Software Engineering follow different strategies towards the organisation of cooperative work:
  - **Software Engineering:**
    - Formal structures and methods for structuring the software process.
  - **Computer Supported Cooperative Working:**
    - Improvement of information processing and cooperation in the group.

- Both the cooperation on the macro level as well as cooperation on the micro level can be supported by CSCW:
  - Macro level: Cooperation between different departments.
  - Micro level: Cooperation within a group.

- For project teams, whose members above all are organised at spatially distributed sites: the term Virtual Teams has established itself in the literature, e.g. Gorton and Motwani 96].
4.1 Cooperative Software Engineering
Summary Virtual Teams

• **Potential advantages, that might result from the application of Virtual Teams:**
  
  – Enabling a 24 hour development
  – Reduction of the Time-to-Market-time period for software-products.
  – Minimisation of duplicate expert knowledge at different branches
  – Profiting from the less bureaucratic and well trained manpower in developing countries.
  – Bringing in global perspectives into the product development during the development itself.
  – Improvement of the software quality by an improved development process.

4.1 Cooperative Software Engineering
Groupware and its potential in SWE

• **Potential benefits include**
  
  – improving the quality of shared information
  – improving the access to shared information
  – improving the speed and accuracy of group decision making
  – reducing the need for time consuming face-to-face meetings

• **Potential risks include**
  
  – overuse of communication tools allows not enough time for thinking,
  – reduced access to information for people who do not have adequate technology, and
  – variable reliability of shared data.
## 4.1 Cooperative Software Engineering

### Groupware and its potential in SWE

<table>
<thead>
<tr>
<th>Software Engineering Activity</th>
<th>Groupware</th>
</tr>
</thead>
<tbody>
<tr>
<td>formulate and exchange ideas</td>
<td></td>
</tr>
<tr>
<td>allocate work among team members</td>
<td></td>
</tr>
<tr>
<td>hold meetings</td>
<td></td>
</tr>
<tr>
<td>develop and edit graphical designs</td>
<td></td>
</tr>
<tr>
<td>develop shared documents and reports</td>
<td></td>
</tr>
<tr>
<td>track the progress of jobs</td>
<td></td>
</tr>
<tr>
<td>chat to each other about the work in hand</td>
<td></td>
</tr>
<tr>
<td>mount presentations and demonstrations</td>
<td></td>
</tr>
</tbody>
</table>

---

### 4.1 Cooperative Software Engineering

#### Introduction of Groupware into organizations

Success of groupware usage requires attention to many issues  
[Coleman, 1997]

**Main points:**

1. **Choice of project**
   - Choose a pilot project
   - Choose a bounded project
   - Choose a project with visibility and financial impact
   - Choose a project where there is a specific problem to be solved

2. **Technology**
   - No single groupware product will be adequate,
   - Don’t expect software vendors to offer you all the services you need for groupware.
   - Try to pick software that fits with existing systems
4.1 Cooperative Software Engineering
Introduction of Groupware into organizations

3. Culture
– Groupware changes the corporate culture.
– You can’t change people overnight.
– People take time to change.

4. Economics
– Realise that training, maintenance, and support will be the majority of the cost
– Measure productivity factors.
– Groupware is not a quick fix!

5. Politics
– Find a champion(supporter) of groupware!

4.2 Software process models for distributed teams
4.2.1 The Cluster Model

Cluster Model (Bertrand Meyer)
• subdivides a project among various development teams
  => division into cluster
• is not (mainly) phase oriented, but
  => cluster orientation
  => component orientation
  => object orientation
• One development team per cluster
  => responsible for all phases
• Strict phase distinction is relaxed
  => borders often disappear
• Intensive communication and coordination within and between development teams needed
4.2.1 The Cluster Model

Mini-Lifecycle of a cluster

Object oriented software engineering (according to Meyer) integrates:

- Concurrent Engineering (overlapped planning)
- Decentralisation
- Flexibility
- The advantages of a structured procedure

The Cluster model also contains sequential components with well defined activities in addition to object oriented paradigms.

=> Every cluster can be described by a mini lifecycle

---

4.2.1 The Cluster Model

The Cluster-Model of the Software-Lifecycle consists of two cluster-independent phases:

- Feasibility Study
- Division into Clusters

The Division of the project into clusters

- lies in the responsibility of the Project Manager,
- may integrate experienced team members.
4.2 Software process models for distributed teams
4.2.2 The Synch-and-Stabilise Approach

- Microsoft Strategy
- **Aim:** To let several small teams (each of 3-8 developers) or individual programmers work together as one big team on a big product/project.
- the influence of structured approaches is very small
  => **loosely coupled small teams**
  - are free to a large extend when taking design decisions and work relative autonomously.
  - Incrementally develop product properties and the product as a whole
  - are free to carry out alterations during the introduction period
  - should often „synchronise“ their changes
  - must constantly „stabilise“ the product

---

4.2.2 The Synch-and-Stabilise Approach
Synch-and-Stabilise -Strategies and Principles (1/2)

- Teams begin the process of product development with a description of a vision:
  - Description of the goals
  - Structuring of the user activities
- During the product development:
  Teams,
  - experience more about the product to be developed
  - revise the product properties
  => Experiences at Microsoft show, that at least 30% or more of the product’s properties in the product specification change.

- Teams are responsible for smaller or bigger product properties.
- The programme manager summarises a functional specification after consultations with the developers:
  - for time schedule and milestones
  - for resource planning
- This initial specification doesn’t however cover all the product properties.
- Project managers divide the product and the project into parts.
### 4.2.2 The Synch-and-Stabilise Approach (Microsoft)

**Synch-and-Stabilise -Strategies and Principles- (2/2)**

- The project is divided into three or four milestones (sequential subprojects)
- All teams carry out a complete development cycle until every milestone is reached.
- Throughout the whole project the teams synchronise their work
  - => - daily or weekly builds
  - - find and correct errors.
- At the end of the milestone-subproject, the developers correct all errors in the product that they found.
  => This error correction stabilises the product
- The team of developers continue with the next milestone up until the product can be delivered.

### 4.2.2 The Synch-and-Stabilise Approach (Microsoft)

**Synch-and-Stabilise Phases –Development phase**

- The product properties will be handled in 3-4 sequential sub-projects
- Every sub-project ends with a Milestone-Release
  - Sub-project I: First third of the product properties
    (Critical properties and shared components)
  - Sub-project II: The second one third of the product properties
  - Semi-project III: Final third of the product properties
    (Least critical properties)
4.2.2 The Synch-and-Stabilise Approach (Microsoft)

**Synch-and-Stabilise Phases -Stabilise-**

- In the stabilisation phase a comprehensive testing will be carried out (internally and externally)
  - **Internal Testing**: testing of the complete product within the company
  - **External Testing**: testing of the complete product outside the company by Beta-users such as OEMs, ISVs and End-users
  - **Tester**: reconstructs errors and isolates them.

- **Programme manager**
  - Coordinating OEMs (Original Equipment Manufacturer) and ISVs (Independent Software Vendors)
  - monitor feedback from the purchaser or the customer respectively.

- **Developers** carry out
  - the last error corrections and
  - code-Stabilisation.

- **Release-Preparation**: The final Release and the documentation will be prepared for delivery.
- **final product will be delivered.**

---

**4.2.2 The Synch-and-Stabilise Approach (Microsoft)**

**Definition of Product and Development processes**

- In order to define a product and to organise a development process, leading Microsoft groups follow a strategy that can be described by „**focus creativity by evolving features and 'fixing' resources**“. This strategy is implemented by the following five principles:
  - Divide big projects into several milestone-cycles with time buffers (20%-50% of the total project time) and without separate product groups for the maintenance.
  - Use a vision description and sketch specifications of product properties, in order to steer the project.
  - The sorting of Basis-Product properties and their priorities follows on the basis of the user activities and data.
  - Develop a modular and horizontal Design-Architecture, so that the product structure is reflected in the project structure.
  - Control through individual commitment to small tasks and fixed project resources.
4.2.2 The Synch-and-Stabilise Approach (Microsoft)  
Product development and Delivery

- For the product development process and delivery Microsoft follows a strategy which can be described as "do everything in parallel with frequent synchronisation".
- Teams follow this strategy by adhering to the following principles:
  - Work in parallel teams with synchronisation and daily error elimination.
  - Always have a product ready for shipping with versions for every important platform and every market.
  - Speak a unifying language in a development subsidiary.
  - Test the product continuously whilst you develop it.
  - Apply metrics, in order to ascertain the completion of Milestones and Product-Releases.

Synch-and-Stabilise vs. Sequential Development

<table>
<thead>
<tr>
<th>Synch-and-Stabilise</th>
<th>Sequential Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product development and tests are carried out in parallel</td>
<td>Separate phases are carried out sequentially</td>
</tr>
<tr>
<td>Description of visions and development of the specification</td>
<td>Complete &quot;frozen&quot; specification and detailed design before the production of the product</td>
</tr>
<tr>
<td>Properties are prioritised and developed in 3 to 4 Milestone-Sub-projects</td>
<td>Attempt, to develop all the parts of the product simultaneously</td>
</tr>
<tr>
<td>Frequent Synchronisation (daily builds) and Stabilisation (Milestone) lying in between</td>
<td>A late and big Integration- and Test phase at the end of project</td>
</tr>
<tr>
<td>Fixed Release- and Delivery date and multiple Release-cycles</td>
<td>Aim of the properties- and product accomplishment in every project, in every project cycle</td>
</tr>
<tr>
<td>Continuous Purchaser-feedback during the development process</td>
<td>Feedback mainly after the development as input for future project</td>
</tr>
<tr>
<td>Product- and Process design, so that bigger teams can be able to work like smaller teams</td>
<td>Work proceeds primarily in big groups of individuals in a separate functional department</td>
</tr>
</tbody>
</table>
4.2 Software process models for distributed teams

4.2.3 The Open Source Model

- The Open-Source-Standard offers no clear guidelines as to how software should be developed in teams.
- Examples:
  1.) LINUX:
      Progress in development due to unexpected stimuli and synergies between developers
  2.) Netscape Web browser:
      - Protected (Registered) commercial product
      - Over five years of development time passed before the company decided to provide the software under Open-Source-conditions to the public.

4.2.3 The Open Source Model

Democratic Open-Source-Teams

- The democratic Open-Source-Team structure
  - is used for lots of newly started Open-Source-Projects
  - is nearly identical to the traditional democratic team principle (Emphasis on the communication and coordination between team members)
  - leaves coordination and synchronisation open as a random access part of the development.
4.2.3 The Open Source Model
Collective Open-Source-Teams

The Collective Open-Source Team Structure
• has lots of similarities with the traditional Chief-Programmer-Team and the Synch-and-Stabilise-Team.
• Features:
  – flexible number of developers in the collective => additional complexity.
  – free choice of team structure and life cycle by the developers
  – Code-changes must be released and made available to other developers.
  – More than one Chef-Programmer or Programme manager possible
• Application:
  – Linux-Kernel
  – Netscape Web-Browser.

4.2.3 The Open Source Model
Distribution and Communication in Open-Source-Projects

• Open-Source Teams can be globally distributed.
  => Communication between team members is of decisive importance.
  => The internet is a key component for the success of Open-Source-Projects.
• „The real lesson to be learnt from open source communities is the techniques of networked collaboration that they’ve pioneered.“ - Tim O’Reilly
### 4.2.3 The Open Source Model

**Properties of Open-Source-Teams**

An Open-Source-Team can be described by the following properties:

- Users drive the development forward and bring in their own improvements.
- Decentralised teams are distributed over the whole world and use the internet as the primary means of communication and coordination.
- The initial development follows typically a „Code-and-Fix“-procedure.
- The development follows an iterative Rapid-Prototype-process, since the product is always further developing new product requirements.
- Tests and error elimination will be carried out parallel to the development by development society.
- Errors found will be sent to a central place which will take care for solving them.
- New product requirements and Bug-Fixes will be implemented on a willingness basis.

### 4.2 Software process models for distributed teams

**4.2.4 eXtreme Programming**

**eXtreme Programming (XP)**

- Illustrates a process model for object oriented software development
- Is particularly suitable for smaller projects with hazy or changing requirements respectively.
4.2.4 eXtreme Programming

The Basic Rules of eXtreme Programming

The use of XP demands the maintenance of the following 12 Basic Rules:

1. The Planning Game
2. Small Releases
3. System Metaphor
4. Simple Design
5. Continuous Testing
6. Re-factoring
7. Pair Programming
8. Collective Code Ownership
9. Continuous Integration
10. 40-Hour Work Week
11. On-site Customer
12. Coding Standards

4.2.4 eXtreme Programming

Distributed eXtreme Programming (DXP)

From the basic XP rules, the following items can be viewed as being critical due to the intensive demand of communication:

• The Planning Game
  – In a distributed environment there exists two central problematic positions:
    1. The story cards be must accessible by both partners.
    2. The partners must be able to communicate with each other.

• Continuous Testing:
  – Distributed testing profits from a good implementation of the support of the Distributed Pair Programming.
  – Test output must be displayed to all participants of the test session.
  – Since at least two computers take part in the distributed execution of the tests, there comes the question, on which computer should the test code run.
### 4.2.4 eXtreme Programming

**Distributed eXtreme Programming (DXP)**

- **Pair Programming**
  - During the Distributed Pair Programming verbal communication is necessary.
  - Both partners must see the same part of the code.
  - One of them must have the possibility to alter the code.

- **Continuous Integration**
  - Alterations and their dependencies to their own alterations must be visible at all times.

- **On-site Customer:**
  - Additional communication possibilities for communication with the client and further project participants are essential.