CHAPTER 4

TEAMS
Overview

- Team organization
- Democratic team approach
- Classical chief programmer team approach
- Beyond chief programmer and democratic teams
- Synchronize-and-stabilize teams
- Teams for agile processes
- Open-source programming teams
- People capability maturity model
- Choosing an appropriate team organization

4.1 Team Organization

- A product must be completed within 3 months, but 1 person-year of programming is still needed

- Solution:
  - If one programmer can code the product in 1 year, four programmers can do it in 3 months

- Nonsense!
  - Four programmers will probably take nearly a year
  - The quality of the product is usually lower
Task Sharing

- If one farm hand can pick a strawberry field in 10 days, ten farm hands can pick the same strawberry field in 1 day.

- One elephant can produce a calf in 22 months, but 22 elephants cannot possibly produce that calf in 1 month.

Task Sharing (contd)

- Unlike elephant production, it is possible to share coding tasks between members of a team.

- Unlike strawberry picking, team members must interact in a meaningful and effective way.
Programming Team Organization

• Example:
  - Sheila and Harry code two modules, m₁ and m₂, say

• What can go wrong
  - Both Sheila and Harry may code m₁, and ignore m₂
  - Sheila may code m₁, Harry may code m₂. When m₁ calls m₂ it passes 4 parameters; but m₂ requires 5 parameters
  - Or, the order of parameters in m₁ and m₂ may be different
  - Or, the order may be same, but the data types may be slightly different

Programming Team Organization (contd)

• This has nothing whatsoever to do with technical competency
  - Team organization is a managerial issue
Communications Problems

- Example
  - There are three channels of communication between the three programmers working on a project. The deadline is rapidly approaching but the code is not nearly complete

- “Obvious” solution:
  - Add a fourth programmer to the team

Communications Problems (contd)

- But other three have to explain in detail
  - What has been accomplished
  - What is still incomplete

- Brooks’s Law
  - Adding additional programming personnel to a team when a product is late has the effect of making the product even later
Team Organization

- Teams are used throughout the software production process
  - But especially during implementation
  - Here, the discussion is presented within the context of programming teams

- Two extreme approaches to team organization
  - Democratic teams (Weinberg, 1971)
  - Chief programmer teams (Brooks, 1971; Baker, 1972)

4.2 Democratic Team Approach

- Basic underlying concept — *egoless programming*

- Programmers can be highly attached to their code
  - They even name their modules after themselves
  - They see their modules as extension of themselves
Democratic Team Approach (contd)

- If a programmer sees a module as an extension of his/her ego, he/she is not going to try to find all the errors in “his”/“her” code
  - If there is an error, it is termed a *bug*.
  - The fault could have been prevented if the code had been better guarded against the “bug.”
  - “Shoo-Bug” aerosol spray

Democratic Team Approach (contd)

- Proposed solution

- Egoless programming
  - Restructure the social environment
  - Restructure programmers’ values
  - Encourage team members to find faults in code
  - A fault must be considered a normal and accepted event
  - The team as whole will develop an ethos, a group identity
  - Modules will “belong” to the team as whole
  - A group of up to 10 egoless programmers constitutes a *democratic team*
Difficulties with Democratic Team Approach

- Management may have difficulties
  - Democratic teams are hard to introduce into an undemocratic environment

Strengths of Democratic Team Approach

- Democratic teams are enormously productive
- They work best when the problem is difficult
- They function well in a research environment
- Problem:
  - Democratic teams have to spring up spontaneously
4.3 Classical Chief Programmer Team Approach

- Consider a 6-person team
  - Fifteen 2-person communication channels
  - The total number of 2-, 3-, 4-, 5-, and 6-person groups is 57
  - This team cannot do 6 person-months of work in 1 month

![Diagram](image-url)

Classical Chief Programmer Team

- Six programmers, but now only 5 lines of communication

![Diagram](image-url)
Classical Chief Programmer Team (contd)

- The basic idea behind the concept
  - Analogy: chief surgeon directing an operation, assisted by
    - Other surgeons
    - Anesthesiologists
    - Nurses
    - Other experts, such as cardiologists, nephrologists

- Two key aspects
  - Specialization
  - Hierarchy

Classical Chief Programmer Team (contd)

- Chief programmer
  - Successful manager and highly skilled programmer
  - Does the architectural design
  - Allocates coding among the team members
  - Writes the critical (or complex) sections of the code
  - Handles all the interfacing issues
  - Reviews the work of the other team members
  - Is personally responsible for every line of code
Classical Chief Programmer Team (contd)

- Back-up programmer
  - Necessary only because the chief programmer is human
  - The back-up programmer must be in every way as competent as the chief programmer, and
  - Must know as much about the project as the chief programmer
  - The back-up programmer does black-box test case planning and other tasks that are independent of the design process

Classical Chief Programmer Team (contd)

- Programming secretary
  - A highly skilled, well paid, central member of the chief programmer team
  - Responsible for maintaining the program production library (documentation of the project), including:
    - Source code listings
    - JCL
    - Test data
  - Programmers hand their source code to the secretary who is responsible for
    - Conversion to machine-readable form
    - Compilation, linking, loading, execution, and running test cases (this was 1971, remember!)
Classical Chief Programmer Team (contd)

- Programmers
  - Do nothing but program
  - All other aspects are handled by the programming secretary

The New York Times Project

- Chief programmer team concept
  - First used in 1971
  - By IBM
    - To automate the clippings data bank ("morgue") of the New York Times

- Chief programmer — F. Terry Baker
The New York Times Project (contd)

- 83,000 source lines of code (LOC) were written in 22 calendar months, representing 11 person-years
- After the first year, only the file maintenance system had been written (12,000 LOC)
- Most code was written in the last 6 months
- Only 21 faults were detected in the first 5 weeks of acceptance testing

The New York Times Project (contd)

- 25 further faults were detected in the first year of operation
- Principal programmers averaged one detected fault and 10,000 LOC per person-year
- The file maintenance system, delivered 1 week after coding was completed, operated 20 months before a single failure occurred
- Almost half the subprograms (usually 200 to 400 lines of PL/I) were correct at first compilation

© The McGraw-Hill Companies, 2007
The New York Times Project (contd)

- But, after this fantastic success, no comparable claims for the chief programmer team concept have been made

Why Was the NYT Project Such a Success?

- Prestige project for IBM
  - First real trial for PL/I (developed by IBM)
  - IBM, with superb software experts, used its best people

- Extremely strong technical backup
  - PL/I compiler writers helped the programmers
  - JCL experts assisted with the job control language
Why Was the NYT Project Such a Success?

- F. Terry Baker
  - Superprogrammer
  - Superb manager and leader
  - His skills, enthusiasm, and personality “carried” the project

- Strengths of the chief programmer team approach
  - It works
  - Numerous successful projects have used variants of CPT

Impracticality of Classical CPT

- The chief programmer must be a highly skilled programmer and a successful manager

- There is a shortage of highly skilled programmers

- There is a shortage of successful managers

- The qualities needed to be a highly skilled programmer are unlikely to be found in a successful manager, and vice versa
Impracticality of Classical CPT (contd)

- The *back-up programmer* must be as good as the chief programmer
  - But he/she must take a back seat (and a lower salary) waiting for something to happen to the chief programmer
  - Top programmers, top managers will not do that

- The *programming secretary* does nothing but paperwork all day
  - Software professionals hate paperwork

- Classical CPT is impractical

4.4 Beyond CP and Democratic Teams

- We need ways to organize teams that
  - Make use of the strengths of democratic teams and chief programmer teams, and
  - Can handle teams of 20 (or 120) programmers

- A strength of democratic teams
  - A positive attitude to finding faults

- Use CPT in conjunction with code walkthroughs or inspections
Beyond CP and Democratic Teams (contd)

- Potential pitfall

- The chief programmer is personally responsible for every line of code
  - He/she must therefore be present at reviews

- The chief programmer is also the team manager
  - He/she must therefore not be present at reviews!

Solution

- Reduce the managerial role of the chief programmer

Figure 4.4

© The McGraw-Hill Companies, 2007
Beyond CP and Democratic Teams (contd)

- It is easier to find a team leader than a chief programmer

- Each employee is responsible to exactly one manager — lines of responsibility are clearly delineated

- The team leader is responsible for only technical management

Beyond CP and Democratic Teams (contd)

- Budgetary and legal issues, and performance appraisal are not handled by the team leader

- The team leader participates in reviews — the team manager is not permitted to do so

- The team manager participates in regular team meetings to appraise the technical skills of the team members
Larger Projects

- The nontechnical side is similar
  - For even larger products, add additional layers

Beyond CP and Democratic Teams (contd)

- Decentralize the decision-making process, where appropriate
  - Useful where the democratic team is good
4.5 Synchronize-and-Stabilize Teams

- Used by Microsoft
- Products consist of 3 or 4 sequential builds
- Small parallel teams
  - 3 to 8 developers
  - 3 to 8 testers (work one-to-one with developers)
  - The team is given the overall task specification
  - They may design the task as they wish

Synchronize-and-Stabilize Teams (contd)

- Why this does not degenerate into hacker-induced chaos?
  - Daily synchronization step
  - Individual components always work together
Synchronize-and-Stabilize Teams (contd)

- Rules
  - Programmers must adhere to the time for entering the code into the database for that day’s synchronization

- Analogy
  - Letting children do what they like all day…
  - … but with a 9 P.M. bedtime

Synchronize-and-Stabilize Teams (contd)

- Will this work in all companies?
  - Perhaps if the software professionals are as good as those at Microsoft

- Alternate viewpoint
  - The synchronize-and-stabilize model is simply a way of allowing a group of hackers to develop large products
  - Microsoft’s success is due to superb marketing rather than quality software
4.6 Teams For Agile Processes

- Feature of agile processes
  - All code is written by two programmers sharing a computer
  - “Pair programming”

Strengths of Pair Programming

- Programmers should not test their own code
  - One programmer draws up the test cases, the other tests the code

- If one programmer leaves, the other is sufficiently knowledgeable to continue working with another pair programmer

- An inexperienced programmer can learn from his or her more experienced team member

- Centralized computers promote egoless programming
4.7 Open-Source Programming Teams

- Open-source projects
  - Are generally staffed by teams of unpaid volunteers
  - Who communicate asynchronously (via e-mail)
  - With no team meetings and
  - With no managers
  - There are no specifications or designs, and
  - Little or no other documentation

- So, why have a small number of open-source projects (such as Linux and Apache) attained the highest levels of success?

Open-Source Programming Teams (contd)

- Individuals volunteer to take part in an open-source project for two main reasons

- Reason 1: For the sheer enjoyment of accomplishing a worthwhile task
  - In order to attract and keep volunteers, they have to view the project as “worthwhile” at all times

- Reason 2: For the learning experience
The Open-Source Learning Experience

- Software professionals often join an open-source project to gain new skills
  - For a promotion, or
  - To get a better job elsewhere

- Many employers view experience with a large, successful open-source project as better than additional academic qualifications

Open-Source Programming Teams (contd)

- The members of the open-source team must at all times feel that they are making a contribution

- For all these reasons, it is essential that the key individual behind an open-source project be a superb motivator
  - Otherwise, the project is doomed to inevitable failure
For a successful open-source project, the members of the core group must be top-caliber individuals with skills of the highest order.

Such top-class individuals can thrive in the unstructured environment of an open-source team.

In summary, an open-source project succeeds because of:
- The nature of the target product
- The personality of the instigator
- The talents of the members of the core group

The way that a successful open-source team is organized is essentially irrelevant.
4.8 People Capability Maturity Model

- Best practices for managing and developing the workforce of an organization

- Each maturity level has its own KPAs
  - Level 2: Staffing, communication and coordination, training and development, work environment, performance management, coordination
  - Level 5: Continuous capability improvement, organizational performance alignment, continuous workforce innovation

People Capability Maturity Model (contd)

- P–CMM is a framework for improving an organization’s processes for managing and developing its workforce

- No one specific approach to team organization is put forward
4.9 Choosing an Appropriate Team Organization

- There is no one solution to the problem of team organization

- The “correct” way depends on
  - The product
  - The outlook of the leaders of the organization
  - Previous experience with various team structures

Choosing an Appropriate Team Organization (contd)

- Exceedingly little research has been done on software team organization
  - Instead, team organization has been based on research on group dynamics in general

- Without relevant experimental results, it is hard to determine optimal team organization for a specific product
## Choosing an Appropriate Team Organization (contd)

<table>
<thead>
<tr>
<th>Team Organization</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Democratic teams (Section 4.2)</td>
<td>High-quality code as consequence of positive attitude to finding faults</td>
<td>Experienced staff expect their code being appraised by beginners</td>
</tr>
<tr>
<td>Classical chief programmer teams</td>
<td>Particularly good with hard problems</td>
<td>Cannot be externally imposed</td>
</tr>
<tr>
<td>Classical chief programmer teams (Section 4.3)</td>
<td>Major success of New York Times project</td>
<td>Impractical</td>
</tr>
<tr>
<td>Modified chief programmer teams</td>
<td>Many successes</td>
<td>No successes comparable to the New York Times project</td>
</tr>
<tr>
<td>Modern hierarchical programming teams</td>
<td>Team manager/team leader structure obviates need for chief programmer</td>
<td>Problems can arise unless areas of responsibility of the team manager and</td>
</tr>
<tr>
<td>(Section 4.4)</td>
<td>Scales up</td>
<td>the team leader are clearly delineated</td>
</tr>
<tr>
<td>Synchronised and stabilise teams</td>
<td>Ensures creativity</td>
<td>No evidence so far that this method can be utilized outside Microsoft</td>
</tr>
<tr>
<td>(Section 4.5)</td>
<td>Ensures that a huge number of developers can work toward a common goal</td>
<td></td>
</tr>
<tr>
<td>Agile process teams</td>
<td>Programmers do not test their own code</td>
<td>Still too rare evidence regarding efficacy</td>
</tr>
<tr>
<td>(Section 4.4)</td>
<td>Knowledge is not lost if one programmer leaves</td>
<td></td>
</tr>
<tr>
<td>Open-source teams (Section 4.7)</td>
<td>Group ownership of code</td>
<td>Narrowly applicable</td>
</tr>
<tr>
<td></td>
<td>A few projects are extremely successful</td>
<td>Must be led by a superb motivator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires top caliber participants</td>
</tr>
</tbody>
</table>

Figure 4.7