UNIT-II
Improving Software Economics

- It is not that much easy to improve the software economics but also difficult to measure and validate.

- There are many aspects are there in order to improve the software economics they are, Size, Process, Personnel, Environment and quality.

- These parameters (aspects) are not independent they are dependent. For example, tools enable size reduction and process improvements, size-reduction approaches lead to process changes, and process improvements drive tool requirements.

- GUI technology is a good example of tools enabling a new and different process. GUI builder tools permitted engineering teams to construct an executable user interface faster and less cost.

- Two decades ago, teams developing a user interface would spend extensive time analyzing factors, screen layout, and screen dynamics. All this would done on paper. Where as by using GUI, the paper descriptions are not necessary.

- Along with these five basic parameters another important factor that has influenced software technology improvements across the board is the ever-increasing advances in hardware
performance.

<table>
<thead>
<tr>
<th>TABLE 3-1. Important trends in improving software economics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COST MODEL PARAMETERS</strong></td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Abstraction and component-based development technologies</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>Methods and techniques</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Personnel</td>
</tr>
<tr>
<td>People factors</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Environment</td>
</tr>
<tr>
<td>Automation technologies and tools</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Quality</td>
</tr>
<tr>
<td>Performance, reliability, accuracy</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**REDUCING SOFTWARE PRODUCT SIZE:**

- By choosing the type of the language
- By using Object-Oriented methods and visual modeling
- By reusing the existing components and building reusable components &
- By using commercial components, we can reduce the product size of a software.
Here UPFs (Universal Function Points) are useful estimators for language-independent in the early life cycle phases. The basic units of function points are:

- External user inputs
- External outputs
- Internal logical data groups
- External data Interfaces
- External inquiries

**OBJECT ORIENTED METHODS AND VISUAL MODELING:**

- There has been a widespread movements in the 1990s toward Object-Oriented technology.
Some studies concluded that Object-Oriented programming languages appear to benefit both software productivity and software quality. One of such Object-Oriented method is UML-Unified Modeling Language.

Booch described the following three reasons for the success of the projects that are using Object-Oriented concepts:

1) An OO-model of the problem and its solution encourages a common vocabulary between the end user of a system and its developers, thus creating a shared understanding of the problem being solved.

2) The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without weaken the entire development effort.

3) An OO-architecture provides a clear separation among different elements of a system, creating firewalls that prevent a change in one part of the system from the entire architecture.

He also suggested five characteristics of a successful OO-Project,

1) A cruel focus on the development of a system that provides a well understood collection of essential minimal characteristics.

2) The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.

3) The effective use of OO-modeling.

4) The existence of a strong architectural vision.

5) The application of a well-managed iterative and incremental development life cycle.

REUSE:

Organizations that translates reusable components into commercial products has the following characteristics:

- They have an economic motivation for continued support.

- They take ownership of improving product quality, adding new features and transitioning to new technologies.
- They have a sufficiently broad customer base to be profitable.

COMMERCIAL COMPONENTS

<table>
<thead>
<tr>
<th>APPROACH</th>
<th>ADVANTAGES</th>
<th>DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial components</td>
<td>Predictable license costs</td>
<td>Frequent upgrades</td>
</tr>
<tr>
<td></td>
<td>Broadly used, mature technology</td>
<td>Up-front license fees</td>
</tr>
<tr>
<td></td>
<td>Available now</td>
<td>Recurring maintenance fees</td>
</tr>
<tr>
<td></td>
<td>Dedicated support organization</td>
<td>Dependency on vendor</td>
</tr>
<tr>
<td></td>
<td>Hardware/software independence</td>
<td>Run-time efficiency sacrifices</td>
</tr>
<tr>
<td></td>
<td>Rich in functionality</td>
<td>Functionality constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration not always trivial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No control over upgrades and maintenance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unnecessary features that consume extra resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Often inadequate reliability and stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple-vendor incompatibilities</td>
</tr>
<tr>
<td>Custom development</td>
<td>Complete change freedom</td>
<td>Expensive, unpredictable development</td>
</tr>
<tr>
<td></td>
<td>Smaller, often simpler implementations</td>
<td>Unpredictable availability date</td>
</tr>
<tr>
<td></td>
<td>Often better performance</td>
<td>Undefined maintenance model</td>
</tr>
<tr>
<td></td>
<td>Control of development and enhancement</td>
<td>Often immature and fragile</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single-platform dependency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drain on expert resources</td>
</tr>
</tbody>
</table>

IMPROVING SOFTWARE PROCESSES:
There are three distinct process perspectives:

1) **Meta process:**
   - It is an Organization’s policies, procedures, and practices for pursuing a software-intensive line of business.
   - The focus of this process is of organizational economics, long-term strategies, and a software ROI.

2) **Macro process:**
   - A project’s policies, and practices for producing a complete software product within certain cost, schedule, and quality constraints.
   - The focus of the macroprocess is on creating an sufficient instance of the metaprocess for a specific set of constraints.

3) **Micro process:**
   - A projects team’s policies, procedures, and practices for achieving an artifact of a software process.
   - The focus of the microprocess is on achieving an intermediate product baseline with sufficient functionality as economically and rapidly as practical.

The objective of process improvement is to maximize the allocation of resources to productive activities and minimize the impact of overhead activities on resources such as personnel, computers, and schedule.

Schedule improvement has at least three dimensions.

1. We could take an N-step process and improve the efficiency of each step.
2. We could take an N-step process and eliminate some steps so that it is now only an M-step process.
3. We could take an N-step process and use more concurrency in the activities being performed or the resources being applied.
TABLE 3-4. Three levels of process and their attributes

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
<th>METAPROCESS</th>
<th>MACROPROCESS</th>
<th>MICROPROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>Line of business</td>
<td>Project</td>
<td>Iteration</td>
</tr>
<tr>
<td>Objectives</td>
<td>Line-of-business profitability</td>
<td>Project profitability</td>
<td>Resource management</td>
</tr>
<tr>
<td></td>
<td>Competitiveness</td>
<td>Risk management</td>
<td>Risk resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project budget, schedule, quality</td>
<td>Milestone budget, schedule, quality</td>
</tr>
<tr>
<td>Audience</td>
<td>Acquisition authorities, customers, Organizational management</td>
<td>Software project managers</td>
<td>Subproject managers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software engineers</td>
<td>Software engineers</td>
</tr>
<tr>
<td>Metrics</td>
<td>Project predictability</td>
<td>On budget, on schedule</td>
<td>On budget, on schedule</td>
</tr>
<tr>
<td></td>
<td>Revenue, market share</td>
<td>Major milestone success</td>
<td>Major milestone progress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project scrap and rework</td>
<td>Release/iteration scrap and rework</td>
</tr>
<tr>
<td>Concerns</td>
<td>Bureaucracy vs. standardization</td>
<td>Quality vs. financial performance</td>
<td>Content vs. schedule</td>
</tr>
<tr>
<td>Time scales</td>
<td>6 to 12 months</td>
<td>1 to many years</td>
<td>1 to 6 months</td>
</tr>
</tbody>
</table>

IMPROVING TEAM EFFECTIVENESS:

- COCOMO model suggests that the combined effects of personnel skill and experience can have an impact on productivity as much as a factor of four over the unskilled personnel.

- Balance and coverage are two of the most important features of excellent teams. Whenever a team is in out of balance then it is vulnerable.

- It is the responsibility of the project manager to keep track of his teams. Since teamwork is much more important than the sum of the individuals.

Boehm – staffing principles:

1) **The principle of top talent:** Use better and fewer people.

2) **The principle of job matching:** Fit the tasks to the skills and motivation of the people available.

3) **The principle of career progression:** An organization does best in the long run by helping its people to self-actualize.
4) **The principle of team balance:** Select people who will complement and synchronize with one another.

5) **The principle of phase-out:** Keeping a misfit on the team doesn’t benefit anyone.

**In general, staffing is achieved by these common methods:**

- If people are already available with required skill set, just take them
- If people are already available but do not have the required skills, re-train them
- If people are not available, recruit trained people
- If you are not able to recruit skilled people, recruit and train people

**Staffing of key personnel is very important:**

- Project Manager
- Software Architect

**Important Project Manager Skills:**

- **Hiring skills.** Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.

- **Customer-interface skill.** Avoiding adversarial relationships among stake-holders is a prerequisite for success.

- **Decision-making skill.** The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.

- **Team-building skill.** Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.

- **Selling skill.** Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy.
Important Software Architect Skills:

- Technical Skills: the most important skills for an architect. These must include skills in both, the problem domain and the solution domain.

- People Management Skills: must ensure that all people understand and implement the architecture in exactly the way he has conceptualized it. This calls for a lot of people management skills and patience.

- Role Model: must be a role model for the software engineers – they would emulate all good (and also all bad!) things that the architect does.

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The following are the some of the configuration management environments which provide the foundation for executing and implementing the process:

Planning tools, Quality assurance and analysis tools, Test tools, and User interfaces provide crucial automation support for evolving the software engineering artifacts.

Round-trip engineering: is a term used to describe the key capability of environments that support iterative development.

Forward engineering: is the automation of one engineering artifact from another, more abstract representation. Ex: compilers and linkers.

Reverse engineering: is the generation of modification of more abstract representation from an existing artifact. Ex: creating visual design model from a source code.

ACHIEVING REQUIRED QUALITY:
Key elements that improve overall software quality include the following:

- Focusing on powerful requirements and critical use case early in the life cycle

- Focusing on requirements completeness and traceability late in the life cycle

- Focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan evolution

- Using metrics and indicators to measure the progress and quality of an architecture as it evolves from high-level prototype into a fully biddable product

- Providing integrated life-cycle environments that support early and continuous configuration control, change management, rigorous design methods, document automation, and regression test automation

- Using visual modeling and higher level languages that support architectural control, abstraction, reliable programming, reuse, and self-documentation

- Early and continuous close look into performance issues through demonstration-based evaluations

**In order to evaluate the performance the following sequence of events are necessary,**

1) Project inception  
2) Initial design review
3) Mid-life-cycle design review  
4) Integration and test
Table 3-5. General quality improvements with a modern process

<table>
<thead>
<tr>
<th>QUALITY DRIVER</th>
<th>CONVENTIONAL PROCESS</th>
<th>MODERN ITERATIVE PROCESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements misunderstanding</td>
<td>Discovered late</td>
<td>Resolved early</td>
</tr>
<tr>
<td>Development risk</td>
<td>Unknown until late</td>
<td>Understood and resolved early</td>
</tr>
<tr>
<td>Commercial components</td>
<td>Mostly unavailable</td>
<td>Still a quality driver, but trade-offs must be resolved early in the life cycle</td>
</tr>
<tr>
<td>Change management</td>
<td>Late in the life cycle, chaotic and malignant</td>
<td>Early in the life cycle, straightforward and benign</td>
</tr>
<tr>
<td>Design errors</td>
<td>Discovered late</td>
<td>Resolved early</td>
</tr>
<tr>
<td>Automation</td>
<td>Mostly error-prone manual procedures</td>
<td>Mostly automated, error-free evolution of artifacts</td>
</tr>
<tr>
<td>Resource adequacy</td>
<td>Unpredictable</td>
<td>Predictable</td>
</tr>
<tr>
<td>Schedules</td>
<td>Overconstrained</td>
<td>Tunable to quality, performance, and technology</td>
</tr>
<tr>
<td>Target performance</td>
<td>Paper-based analysis or separate simulation</td>
<td>Executing prototypes, early performance feedback, quantitative understanding</td>
</tr>
<tr>
<td>Software process rigor</td>
<td>Document-based</td>
<td>Managed, measured, and tool-supported</td>
</tr>
</tbody>
</table>

- Project inception. The proposed design was asserted to be low risk with adequate performance margin.

- Initial design review. Optimistic assessments of adequate design margin were based mostly on paper analysis or rough simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood. However, the infrastructure—including the operating system overhead, the database management overhead, and the interprocess and network communications overhead—and all the secondary threads were typically misunderstood.

- Mid-life-cycle design review. The assessments started whittling away at the margin, as early benchmarks and initial tests began exposing the optimism inherent in earlier estimates.

- Integration and test. Serious performance problems were uncovered, necessitating fundamental changes in the architecture. The underlying infrastructure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.
• Project inception. The proposed design was asserted to be low risk with adequate performance margin.

• Initial design review. Optimistic assessments of adequate design margin were based mostly on paper analysis or rough simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood. However, the infrastructure—including the operating system overhead, the database management overhead, and the interprocess and network communications overhead—and all the secondary threads were typically misunderstood.

• Mid-life-cycle design review. The assessments started whittling away at the margin, as early benchmarks and initial tests began exposing the optimism inherent in earlier estimates.

• Integration and test. Serious performance problems were uncovered, necessitating fundamental changes in the architecture. The underlying infrastructure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.

PEER INSPECTIONS: A PRAGMATIC VIEW:

• Transitioning engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in the engineering artifacts.

• Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases.

• Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control.

• Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria, and requirements compliance.

• Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals.
THE OLD WAY AND THE NEW

- Over the past two decades software development is a re-engineering process. Now it is replaced by advanced software engineering technologies.

- This transition is motivated by the unsatisfactory demand for the software and reduced cost.

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

Based on many years of software development experience, the software industry proposed so many principles (nearly 201 by – Davis’s). Of which Davis’s top 30 principles are:

- Analysis, prototyping, or experimentation
- Constructing design models
- Committing the current state of the design model to an executable implementation
- Demonstrating the current implementation strengths and weaknesses in the context of critical subsets of the use cases and scenarios
- Incorporating lessons learned back into the models, use cases, implementations, and plans
1) **Make quality #1:** Quality must be quantified and mechanisms put into place to motivate its achievement.

2) **High-quality software is possible:** In order to improve the quality of the product we need to involving the customer, select the prototyping, simplifying design, conducting inspections, and hiring the best people.

3) **Give products to customers early:** No matter how hard you try to learn user’s needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it.

4) **Determine the problem before writing the requirements:** Whenever a problem is raised most engineers provide a solution. Before we try to solve a problem, be sure to explore all the alternatives and don’t be blinded by the understandable solution.

5) **Evaluate design alternatives:** After the requirements are greed upon, we must examine a variety of architectures and algorithms and choose the one which is not used earlier.

6) **Use an appropriate process model:** For every project, there are so many prototypes (process models). So select the best one that is exactly suitable to our project.

7) **Use different languages for different phases:** Our industry’s main aim is to provide simple solutions to complex problems. In order to accomplish this goal choose different languages for different modules/phases if required.

8) **Minimize intellectual distance:** We have to design the structure of a software is as close as possible to the real-world structure.

9) **Put techniques before tools:** An un disciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.
10) **Get it right before you make it faster:** It is very easy to make a working program run faster than it is to make a fast program work. Don’t worry about optimization during initial coding.

11) **Inspect the code:** Examine the detailed design and code is a much better way to find the errors than testing.

12) **Good management** is more important than good technology

13) **People are the key to success:** Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed.

14) **Follow with care:** Everybody is doing something but does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.

15) **Take responsibility:** When a bridge collapses we ask “what did the engineer do wrong?”. Similarly if the software fails, we ask the same. So the fact is in every engineering discipline, the best methods can be used to produce poor results and the most out of date methods to produce stylish design.

16) **Understand the customer’s priorities.** It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.

17) **The more they see, the more they need.** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18) **Plan to throw one away.** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.

19) **Design for change.** The architectures, components, and specification techniques you use must accommodate change.

20) **Design without documentation is not design.** I have often heard software engineers say, “I have finished the design. All that is left is the documentation.”

21. **Use tools, but be realistic.** Software tools make their users more efficient.

22. **Avoid tricks.** Many programmers love to create programs with tricks—constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.

23. **Encapsulate.** Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.

24. **Use coupling and cohesion.** Coupling and cohesion are the best ways to measure software’s inherent maintainability and adaptability.

25. **Use the McCabe complexity measure.** Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tom McCabe’s.

26. **Don’t test your own software.** Software developers should never be the primary testers of their own software.

27. **Analyze causes for errors.** It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.

28. **Realize that software’s entropy increases.** Any software system that undergoes continuous change will grow in complexity and become more and more disorganized.
29. **People and time are not interchangeable.** Measuring a project solely by person-months makes little sense.

30) **Expert excellence.** Your employees will do much better if you have high expectations for them.

---

**THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT**

1) Base the process on an architecture-first approach: (Central design element)

   - Design and integration first, then production and test

2) Establish an iterative life-cycle process: (The risk management element)

   - Risk control through ever-increasing function, performance, quality.

   With today’s sophisticated systems, it is not possible to define the entire problem, design the entire solution, build the software, then test the end product in sequence. Instead, an iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages balanced treatment of all stakeholder objectives.

   Major risks must be addressed early to increase predictability and avoid expensive downstream scrap and rework.

3) Transition design methods to emphasize component-based development: (The technology element)

   Moving from LOC mentally to component-based mentally is necessary to reduce the amount of human-generated source code and custom development. A component is a cohesive set of preexisting lines of code, either in source or executable format, with a defined interface and behavior.

4) Establish a change management environment: (The control element)

   - Metrics, trends, process instrumentation

   The dynamics of iterative development, include concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baseline.
5) Enhance change freedom through tools that support round-trip engineering: (The automation element)

- Complementary tools, integrated environment

Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats. Change freedom is necessary in an iterative process.

6) Capture design artifacts in rigorous, model-based notation:

- A model-based approach supports the evolution of semantically rich graphical and textual design notations.

- Visual modeling with rigorous notations and formal machine-processable language provides more objective measures than the traditional approach of human review and inspection of ad hoc design representations in paper doc.

7) Instrument the process for objective quality control and progress assessment:

- Life-cycle assessment of the progress and quality of all intermediate product must be integrated into the process.

- The best assessment mechanisms are well-defined measures derived directly from the evolving engineering artifacts and integrated into all activities and teams.

8) Use a demonstration-based approach to assess intermediate artifacts:

Transitioning from whether the artifact is an early prototype, a baseline architecture, or a beta capability into an executable demonstration of relevant provides more tangible understanding of the design tradeoffs, early integration and earlier elimination of architectural defects.

9) Plan intermediate releases in groups of usage scenarios with evolving levels of detail:

10) Establish a configurable process that economically scalable:

No single process is suitable for all software developments. The process must ensure that there is economy of scale and ROI.
**Architecture-first approach**

Design and integration first, then production and test

**Iterative life-cycle process**

Risk control through ever-increasing function, performance, quality

**Component-based development**

Object-oriented methods, rigorous notations, visual modeling

**Change management environment**

Metrics, trends, process instrumentation

**Round-trip engineering**

Complementary tools, integrated environments

"Software Project Management"

Walker Royce