Prepared by:
D.SRINIVAS
Associate Professor,Dept. of CSE

PRINCIPLES

It's Not The Principle That Keeps You Going
It's The Money That They Pay You

Part 1

Improving Software Economics

Improving Team Effectiveness (1)

- The principle of top talent: *Use better and fewer people.*
- The principle of job matching: *Fit the task to the skills an motivation of the people available.*
- The principle of career progression: *An organization does best in the long run by helping its people to self-actualize.*
- The principle of team balance: *Select people who will complement and harmonize with one another.*
- The principle of phase-out: *Keeping a misfit on the team doesn't benefit anyone.*

Part 1

Improving Software Economics

Improving Team Effectiveness (2)

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.

Part 1

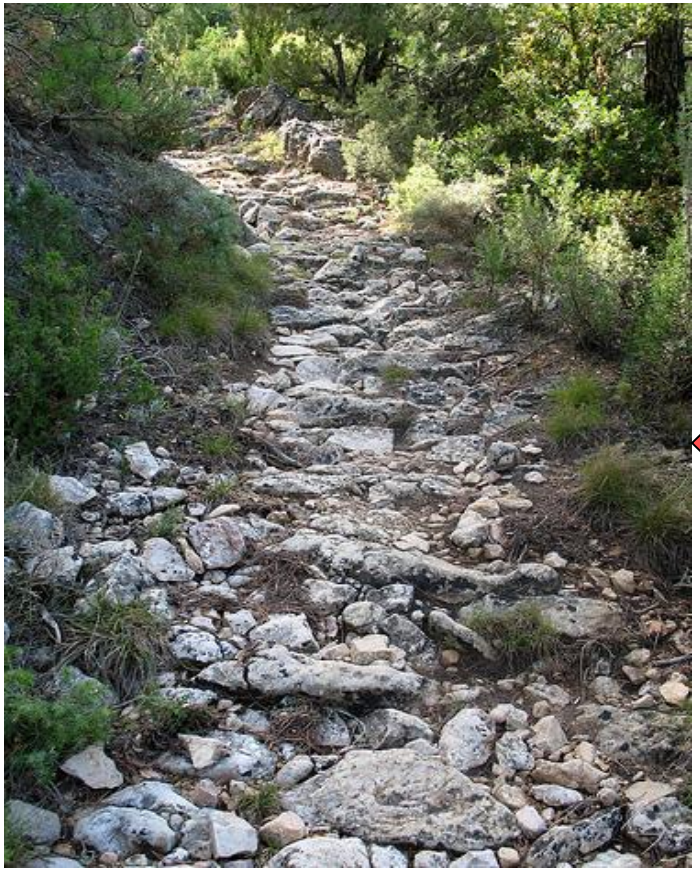
Improving Software Economics

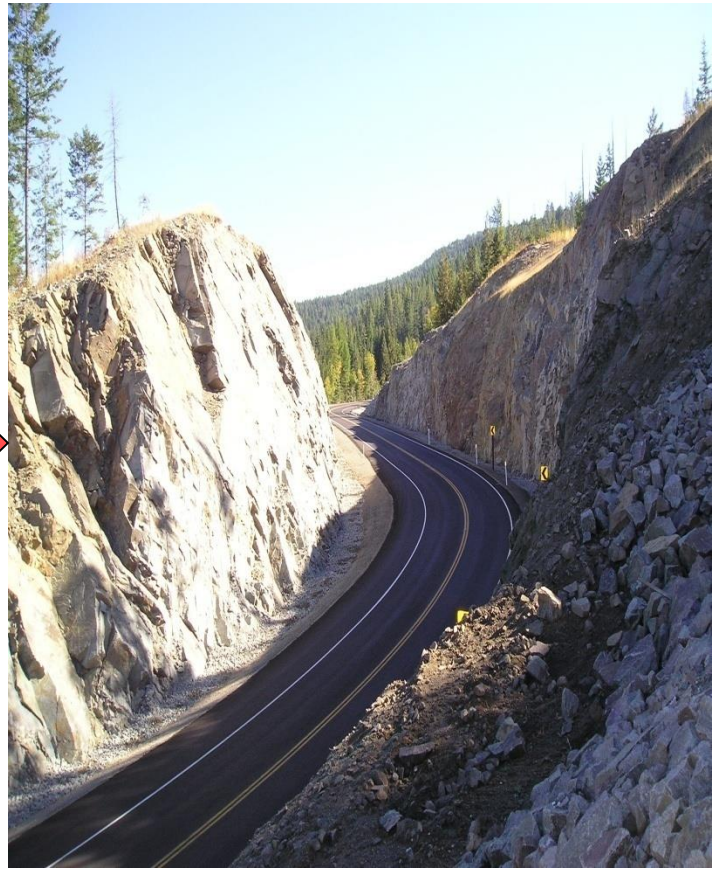
Achieving Required Quality

Key practices that improve overall software quality:

- ✓ Focusing on driving requirements and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the life cycle, and 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 a high-level prototype into a fully compliant 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 language that support architectural control, abstraction, reliable programming, reuse, and self-documentation
- ✓ Early and continuous insight into performance issues through demonstration-based evaluations

Part 1
The Old Way and the New





Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

1. **Make quality #1**. Quality must be quantified and mechanism put into place to motivate its achievement.
2. **High-quality software is possible**. Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people.
3. **Give products to customers early**. No matter how hard you try to learn users' 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**. When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.
5. **Evaluate design alternatives**. After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model**. Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases**. Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation through-out the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?
8. **Minimize intellectual distance**. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure.
9. **Put techniques before tools**. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.
10. **Get it right before you make it faster**. It is far easier to make a working program run than it is to make a fast program work. Don't worry about optimization during initial coding.

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

11. **Inspect code**. Inspecting the detailed design and code is a much better way to find errors than testing.
12. **Good management is more important than good technology**. The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.
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. The wrong people with appropriate tools, languages, and process will probably fail.
14. **Follow with care**. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping-all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.
15. **Take responsibility**. When a bridge collapses we ask, "what did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant 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."

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

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.

Part 1

The Old Way and the New

The Principles of Modern Software Management

Architecture-first approach → The central design element

Design and integration first, then production and test

Iterative life-cycle process → The risk management element

Risk control through ever-increasing function, performance, quality

Component-based development → The technology element

Object-oriented methods, rigorous notations, visual modeling

Change management environment → The control element

Metrics, trends, process instrumentation

Round-trip engineering → The automation element

Complementary tools, integrated environments