UNIT-II

Improving Software Economics :-

Improvements in the Economics of Software development have been not only difficult to achieve but also difficult to measure & substantiate.

→ The key to substantial improvement is a balanced attack across several interrelated dimensions. We have structured the presentation of the important dimensions around the five basic parameters of the software cost model

1) Reducing the size or complexity of what needs to be developed
2) Improving the development process.
3) Using more-skilled personnel & better teams (not necessarily the same)
4) Using better environments (tools to automate the process)
5) Trading off or backing off on quality thresholds.

→ These parameters are given in priority order for most software domains. Table below lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of s/w development and integration.

Trading off → situational decision that involves ~~doing~~ losing one quality, quantity or property of a set or design in return for gains in other Aspects.

one thing Increases
other must Decrease

Threshould → setting certain norms & criteria. Any programme, department, or institution, which reaches these norms & criteria, is deemed to be of Quality.

Table: Important trends in improving Software economics

Table

Cost model parameters	Trends
→ Size:- Abstraction & component-based development technologies	Higher order languages (C++, Ada 95, Java, VB etc.) Object oriented (Analysis, design, programming) Reuse Commercial components
→ process:- Methods & Techniques	Iterative development process maturity models Architecture-first development Acquisition reform.
→ personnel:- people factors	Training & personnel skill development Teamwork win-win cultures
→ ~~Auto~~ Environment:- Automation technologies & tools	Integrated tools (visual modeling, compiler, editor, debugger, change Mgmt, etc) Open systems H/w platform performances. Automation of coding, documents, testing, analyses
→ Quality:- performance, reliability, Accuracy	H/w platform performance. Demonstration-based assessment Statistical Quality control.

using further sup

Table: Important trends in improving Software economics

Table

Cost model parameters	Trends
→ Size:- Abstraction & component-based development technologies	Higher order languages (C++, Ada 95, Java, VB etc.) Object oriented (Analysis, design, programming) Reuse Commercial components
→ process:- Methods & Techniques	Iterative development process maturity models Architecture-first development Acquisition reform.
→ personnel:- people factors	Training & personnel skill development Teamwork win-win cultures
→ ~~Auto~~ Environment:- Automation technologies & tools	Integrated tools (visual modeling, compiler, editor, debugger, change Mgmt, etc) Open systems H/w platform performances. Automation of coding, documents, testing, analyses
Quality:- performance, reliability, Accuracy	H/w platform performance. Demonstration-based assessment Statistical Quality control.

→ Consider the domain of user Interface s/w. Two decades ago, teams developing a user Interface would spend extensive time analyzing operations, human factors, screen layout, & screen dynamics

→ GUI (Technology) is a good example of tools enabling a new & different process. As GUI technology matured, the conventional user Interface process became obsolete.

→ GUI builder tools permitted engineering teams to construct an executable user interface faster and at less cost. paper descriptions were now unnecessary; infact, they were an obstacle to the efficiency of the process.

→ operation Analysis & human factors Analysis were still important, but these activities could now be done in a realistic target environment using existing primitives & building blocks.

→ Another important factor that has influenced s/w technology improvements across the board is the Ever-increasing advances in H/w performance. Availability of more cycles, more memory, & more bandwidth has eliminated many sources of s/w Implementation complexity.

3.1. Reducing software product size :-

The most significant way to improve affordability & return on Investment (ROI) is usually to produce a product that achieves the design goals with the minimum amt. of human-generated source material

→ Component-based Development is introduced here as the general term for reducing the "source" language size necessary to achieve a S/W solution.

Re-use, object-oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source directives (statements)

→ Size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic, & Fourth-generation languages), automatic code generator (CASE tools, visual modeling tools, GUI builders), reuse of commercial components (operating systems, windowing environments, database management systems, middleware, networks) and object-oriented technologies (Unified Modeling Language, visual modeling tools, Architecture Frameworks)

4, object-oriented Methods & visual Modeling
3, Reuse
4, commercial components

3.1.1. Languages :-

Universal function points (UFPs) are useful estimators for language-independent, early life-cycle estimates. Basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, & external Enquiries.

→ SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Substantial data have been documented relating SLOC to function points.

TABLE 3-2. *Language expressiveness of some of today's popular languages*

LANGUAGE	SLOC PER UFP
Assembly	320
C	128
FORTRAN 77	105
COBOL 85	91
Ada 83	71
C++	56
Ada 95	55
Java	55
Visual Basic	35

→ The data in the table illustrate why people are interested in modern languages such as C++, Ada 95, Java and visual Basic.

→ Each language has a domain of usage. Visual Basic is very expressive & powerful in building simple interactive applications, but it would not be a wise choice for a real-time, embedded, avionics program.

→ similarly, Ada 95 might be the best language for a catastrophic cost-of-failure system that controls a nuclear power plant, but it would not be the best choice for a highly parallel, scientific number-crunching program running on a super computer.

→ Difference b/w C and C++:

Difference between 'C' & 'C++' is even more profound.

1, C++ incorporated several of the advances within ADA as well as advanced support for OOP's programming.

2, However, C++ was also developed to support 'C' as a subset.

3, one advantage is 'C' compatability made it very easy for 'C' programmers to transition to C++.

4, one Disadvantage, population of programmers using a 'C++' compiler but programming with a 'C' mindset, therefore failing to achieve the expressibility of object-oriented C++.

5, Evolution of Java has eliminated many of the problems in the C++ language while conserving the object-oriented features and adding further support for portability & Distribution.

→ Universal function points can be used to indicate the relative program sizes required to implement a given functionality.

→ For Example,

To achieve a given application with a fixed no. of function points, one of the following program sizes would be Required:

1,000,000 lines of Assembly language.
4,00,000 lines of C.
2,20,000 lines of Ada 83.
1,75,000 lines of Ada 95 or C++.

These values indicate the Relative Expressiveness provided by various languages.

→ Commercial components & Automatic code Generators

↓ CASE tools ↓ GUI builders.

can further reduce the size of human-generated source code, which in turn reduces the size of the team and the time needed for development.

→ Extending this example, Adding a commercial DBMS, commercial GUI builder, and commercial middleware would reduce the effective size of development to the following final size:-

75,000 lines of ADA 95 or C++ plus Integration of several commercial components.

→ Because the difference between large and small projects has a greater than linear impact on the life-cycle cost, the use of the highest level language & appropriate commercial components has a large potential impact on cost.

→ Furthermore, simpler is generally better:

Reducing size usually increases understandability, changeability and Reliability.

3.1.2. Object-oriented Methods & Visual Modeling:

Some studies have concluded that object-oriented programming languages appear to benefit both software productivity & software quality.

→ But an Economic benefit has yet to be demonstrated because of the steep cost of training in object-oriented design methods such as Unified Modeling Language (UML).

→ The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed. Booch has described three other reasons that certain object-oriented projects succeed.

→ These are interesting examples of the interrelationships among the dimensions of improving S/w Economics.

1) An object-oriented model of the problem and its solution encourages a common vocabulary b/w the end users of a system and its developers, thus creating a shared understanding of the

problem being solved.

↳ This is an example of how object-oriented technology permits corresponding improvements in teamwork & interpersonal communications.

2. The use of continuous integration creates opportunities to recognise risk early & make incremental corrections without destabilizing the entire development Effort.

↳ This Aspect of object-oriented technology enables an Architecture-first process, in which integration is an early & continuous life-cycle Activity.

3. An object-oriented Architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire Architecture.

↳ This feature of object-oriented technology is crucial to the supporting languages & environments available to implement object-oriented Architectures.

→ Booch also summarised five charaiterictics of a successful object-oriented project :-

1. A Ruthless 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 object-oriented Modeling.
4. The Existence of a strong Architectural vision
5. The Application of a well-managed iterative & incremental development life cycle.

3.1.3. Reuse :-

Reusing existing components & building reusable components have been natural S/w Engineering activities since the earliest improvements in programming languages.

→ Software design methods have always dealt implicitly with reuse in order to minimise development costs while achieving all the other required attributes of performance, feature set & Quality

→ I try to treat reuse as a mundane part of achieving a return on Investment. common Architectures, common processes, precedent experience, and common environments are all instances of Reuse.

→ One of the biggest obstacles to reuse has been fragmentation of languages, operating systems, notations, machine Architectures, tools & even "standards". As a counter example, the level of reuse made possible by Microsoft's success on the PC platform has been Immense.

→ In general, things get reused for Economic reasons. Therefore, the Key metric in identifying whether a component (or a class of components, or a commercial product) is truly reusable is to see whether some organisation is making money on it.

→ We should also "Beware" of 'open' Reuse libraries sponsored by Non-profit organisations. They lack Economic motivation, trustworthiness

and accountability for Quality, support, improvement & usability.

→ Truly Reusable components of value have some characteristics:

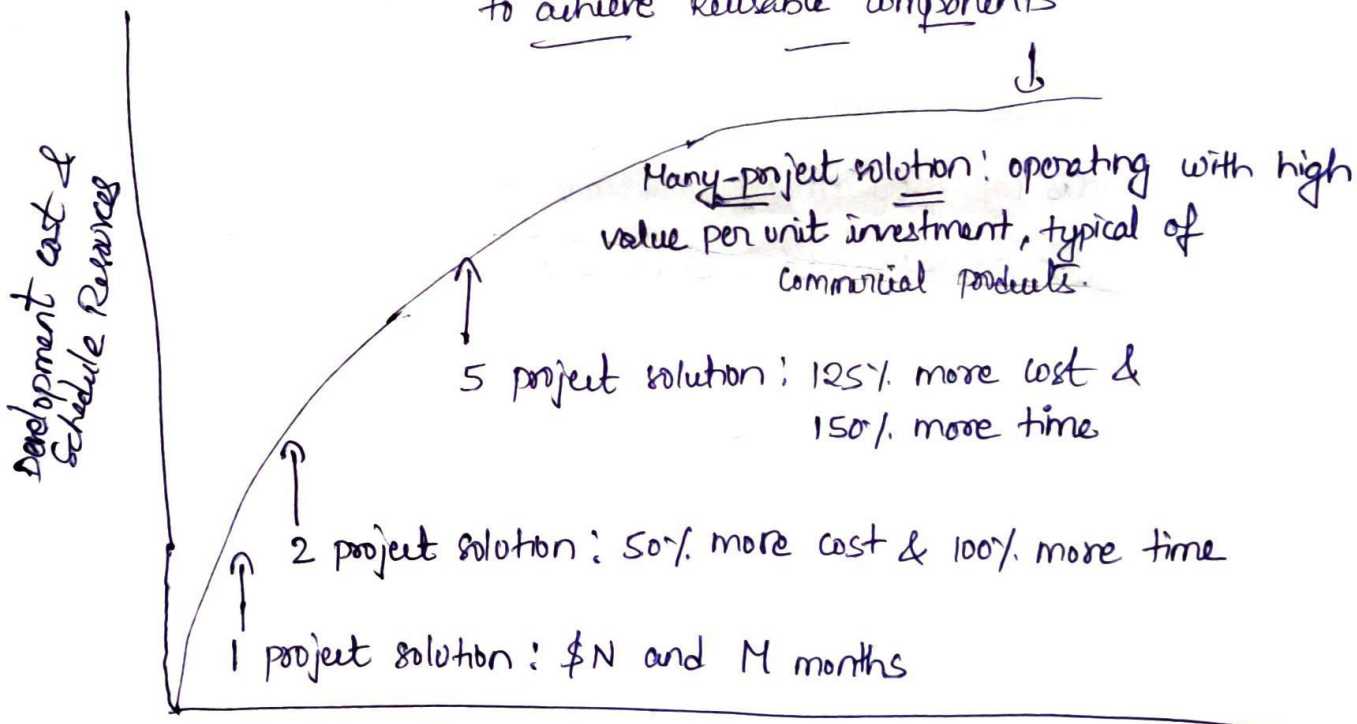
1, They have an Economic motivation for continued support.

2, They take ownership of improving product Quality, adding new features, and transitioning to new technologies.

3, They have a sufficiently broad customer base to be profitable.

→ The cost of developing a Reusable component is not trivial.

Fig: shows cost & schedule Investments necessary to achieve Reusable components



The steep initial curve illustrates the Economic obstacle to developing reusable components. To succeed in the market place for commercial components, an organisation needs three enduring elements:

1, A development Group

2, A support Infrastructure

3, A product-oriented sales and Marketing Infrastructure.

→ Another consideration is that the complexity and cost of developing reusable components are often, naively underestimated. "Reuse" is an important discipline that has an impact on the efficiency of all workflows and the Quality of most Artifacts.

→ Reuse is said to be an synonym for Return on Investment. which should be consideration in almost every activity & decision.

3.1.4. Commercial Components:

A common Approach being persued today in many domains is to maximise integration of commercial components and off-the-shelf products.

→ While the use of commercial components is certainly desirable as a means of reducing custom development, it has not proven to be straight forward in practice.

→ Table below identifies some of the advantages & Disadvantages of using commercial components, Because the trade-offs frequently have global effects on quality, cost, and supportability, the selection of commercial components over development of custom components has significant impact on a project's overall Architecture.

Advantages & DisAdvantages of commercial components versus custom software

Approach	Advantages	DisAdvantages
commercial components	1, predictable license costs 2, Broadly used, mature technology. 3, Available now 4, Dedicated support organisation 5, H/w / S/w Independence 6, Rich in Functionality	1, Frequent upgrades 2, up-front license fees. 3, Recurring Maintenance fees. 4, Dependency on vendor 5, Run-time Efficiency sacrifices. 6, Functionality constraints 7, Integration not always trivial 8, No control over upgrades & maintenance 9, Unnecessary features that consume extra resources 10, often inadequate reliability & Stability 11, Multiple-vendor incompatibilities
• Custom Development	1, complete change freedom 2, Smaller, often simpler implementations 3, often better performance 4, control of development & Enhancement	1, Expensive, unpredictable development. 2, Unpredictable Availability date. 3, undefined Maintenance Model. 4, often immature & fragile. 5, Single-platform dependency. 6, Drain on expert resources.

3.2. Improving software process

process is an overloaded term. For software oriented organisations, there are many processes & subprocesses, but overall, we use three distinct process perspectives

1. Metaprocess
2. Macroprocess
3. Microprocess.

Metaprocess (Business process)

An organization's policies, procedures, and practices for pursuing a s/w-intensive line of business. The focus of this process is on organizational economics, long-term strategies, and a s/w ROI.

Macroprocess

A project's policies, procedures, and practices for producing a complete s/w product within certain cost, schedule, and quality constraints. The focus of the Macroprocess is on creating an adequate instance of the metaprocess for a specific set of constraints.

Microprocess

A project team's policies, procedures, and practices for achieving an artifact of the s/w process. The focus of the microprocess is on achieving an intermediate product baseline with adequate quality & adequate functionality as economically and rapidly as practical.

Three levels of process & their Attributes

Attributes	Metaprocess	Macroprocess	Microprocess
Subject	Line-of-Business	project	Iteration
objectives	Line-of-Business profitability competitiveness	project profitability Risk Management project budget, Schedule, Quality	Resource Management Risk Resolution Milestone budget, Schedule, Quality
Audience	Acquisition authorities, customers organisational Management	Software project Managers Software Engineers	Subproject Managers software Engineers.
Metrics	project predictability Revenue, Market share	on Budget, on Schedule Major milestone success Project scrap & Rework.	on Budget, on Schedule Major milestone progress Release/Iteration scrap & Rework.
Concerns	Bureaucracy vs. Standardization	Quality vs Financial performance.	Content vs Schedule.
Time Scales	6 to 12 months	1 to many years	1 to 6 months.

→ All processes consist of 1, productive Activities &
2, Overhead Activities.

→ productive Activities result in tangible proces toward the End product. For S/w Efforts, these activities include prototyping, modeling, coding, debugging, and User Documentation

→ Overhead Activities that have an intangible impact on the Endproduct are required in plan preparation, documentation, progress monitoring, Risk Assessment, financial Assesment, configuration control, Quality Assesment, Integration, Testing, late scrap and Rework, Management, personnel training, business administration, and other tasks.

→ overhead Activities include many value-added efforts, but in general, the less effort devoted to these activities, the more effort that can be expended in productive Activities.

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

→ Here comes the Concept personnel Training

people will argue that personnel training cannot be a bad thing, but it is for a project. Training is an organisational responsibility, not a project responsibility. staffing every project with a fully trained work force may not be possible, but employing trained people with a is always better than employing untrained people, other things being equal. In this sense, training is not considered a value-added Activity.

→ The Quality of the s/w process strongly affects the required effort and therefore the schedule for producing the s/w product.

→ In practice, the difference b/w a good process and a bad one will affect overall cost estimates by 50% to 100%, and the reduction in effort will improve the overall schedule.

→ For this schedule improvement has atleast 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 & 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.

→ Many organisational time-to-market improvement strategies emphasize the first dimension. In particular, the primary focus of process improvement should be on achieving an adequate solution in the minimum no. of iterations & eliminating as much downstream scrap & rework as possible.

→ Every instance of Rework introduces a sequential set of tasks that must be redone. For example, suppose that a team completes the sequential steps of analysis, design, coding, and testing of a feature, then uncovers a design flaw in testing. Now a sequence of redesign, recode and retest is required.

- Team balance.

→ In a perfect software engineering world with an immaculate problem description, an obvious solution space, a development team of experienced geniuses, adequate resources, and stakeholders with common goals, we could execute a s/w development process in one iteration with almost no scrap and rework.

3.3. Improving Team Effectiveness :- Cocomo model suggests that the combined effects of personal skill & experience have an impact on productivity as much as a factor of ⇒ Balance and coverage characteristics are two of the most important aspects of excellent teams. Teamwork is much more important than the sum of the individuals. With s/w teams, a project manager needs to configure a balance of solid talent with highly skilled people in the leverage positions.

Some maxims of Team management include the following :-

1) A well-managed project can succeed with a Nominal Engineering team.

2) A mismanaged project will almost never succeed, even with an expert team of Engineers.

3) A well-architectured system can be built by a nominal Team of s/w builders.

4) A poorly Architected system will flounder even with an expert team of builders.

Cocomo model calculates a s/w project time, effort, cost & predicts the path and of a s/w project.

Boehm - staffing principles:

→ How to staff a software project, Boehm offered the following five staffing principles.

1) The principle of Top talent:
Use better & fewer people.

2) The principle of Job Matching:
Fit the tasks to the skills & motivaton of the people available.

3) The principle of career progression: An organisation does best in the long run by helping its people to self-actualize. (Complete realization of one's potential)

4) The principle of team balance:
Select people who will complement & harmonise with one another.

5) The principle of phaseout:
Keeping a misfit on the team doesn't benefit anyone.

→ Software development is a team sport. Managers must nurture a culture of teamwork and results rather than individual Accomplishment.

→ Of the five principles, team balance and job matching should be the primary objectives. The top talent and phaseout principles are secondary objectives because they must be applied within the context of Team balance.

Important ~~Software Hons~~ Project Manager Skills:

→ Software project managers need many leadership qualities in order to enhance team effectiveness. The following are some crucial attributes of successful software project managers that deserve much more attention.

1. 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

2. Customer-interface skill:

Avoiding adversarial relationships among stakeholders is a prerequisite for success.

3. Decision-making skill:

A good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.

4. 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

5. Selling skill:

Successful project managers must sell all stakeholders (including themselves) on decisions & 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. (10)

→ 3.4 Improving Automation through Software Environment

The tools and the Environment used in the software process generally have a linear effect on the productivity of the process.

→ planning tools, Requirements management tools, visual modeling tools, compilers, editors, debuggers, Quality assurance analysis tools, test tools, and user interfaces provide crucial automation support for evolving the software Engineering Artifacts.

→ Above all, configuration management environment provide the foundation for executing and instrumenting the process.

→ The Transition to a mature software process introduces, new challenge and opportunities for management control of concurrent activities and for tangible process & Quality Assessments.

→ Project Experience has shown that a highly integrated environment is necessary both to facilitate and to enforce management control of the process. An environment that provides semantic integration (in which the environment understands the detailed meaning of the development artifacts) and process automation can improve productivity, improve software Quality, and accelerate the adoption of Modern techniques.

→ An Environment that supports incremental compilation, automated system builds, and integrated regression testing can provide rapid turnaround for iterative development & allow development teams to iterate more freely.

→ An Important emphasis of a Modern Approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support the automation of the development process.

→ This Environment should include requirements management, document automation, host/target programming tools, automated Regression testing, continuous and integrated change management, and feature/defect Tracking

→ Round-Trip Engineering:

It is a term used to describe the key capability of environments that support iterative development. As we have moved into maintaining different information repositories for the Engineering Artifacts, we need automation support to ensure efficient and Error-free transition of data from one artifact to another.

→ Forward Engineering:

It is the Automation of one Engineering artifact from another, more abstract Representation.

For Example, compilers & linkers have provided automated translation of source code into Executable code.

→ Reverse Engineering is the generation or modification of a more abstract representation from an existing artifact

For Ex: creating a visual design model from a source code Representation

→ Round-trip Engineering describes the Environment support needed to change an artifact freely and have other artifacts automatically changed so that consistency is maintained among the entire set of requirements, design, implementation, and deployment Artifacts.

→ one word of caution is necessary in describing the Economic improvements associated with tools & Environments. It is common for tool vendors to make relatively accurate individual Assessments of life-cycle activities to support claims about the potential economic impact of their tools.

→ For Example

It is easy to find statements such as the following from companies in a particular tool Niche:-

1, Requirements Analysis and Evolution Activities consume 40% of Life-cycle costs.

2, Software Design activities have an impact on more than 50% of the Resources.

3, coding & unit Testing activities consume about 50% of software development effort and Schedule.

4, Test Activities can consume as much as 50% of a project's Resources.

5, Configuration control & change management are critical activities that can consume as much as 25% of resources on a large-scale project.

6, Documentation Activities can consume more than 30% of project Engineering resources.

7, project Management, Business Administration, and progress Assessment can consume as much as 30% of project budgets.

3.5. Achieving Required Quality :-

Key practices that improve overall software Quality include the following :-

1, Focusing on driving requirements & critical use cases early in the life cycle, focusing on requirements completeness & traceability late in the life cycle, and focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan Evolution.

2, 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.

3, providing integrated life-cycle environment that support early & continuous configuration control, change management, rigorous design methods, document ~~automation~~ automation, & Regression test automation.

4, Using visual Modeling & Higher level languages that support architectural control, abstraction, reliable programming, reuse, and self-documentation

5, Early and continuous insight into performance issues through demonstration-based evaluation.

TABLE 3-5. *General quality improvements with a modern process*

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

5) Early and continuous insight into performance issues through demonstration-based evaluation.

→ Typical chronology of events in performance Assessment as follows:-

1) project Inception:- → client's strategic business decision making (Transition blw & implementation of project)

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

2) Critical 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 & Network communications overhead and secondary threads were typically misunderstood.

3) 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.

4) Integration & 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.

→ This sequence occurred because early performance insight was based solely on naive engineering judgement of innumerable criteria.

→ In most large-scale distributed systems composed of many interacting components, a demonstration-based approach can provide significantly more-accurate assessments of performance issues.

→ 3.6: Peer Inspections :- A pragmatic view :-

Peer inspections are frequently overhyped as the key aspect of a quality system. peer reviews are valuable as secondary mechanisms, but they are rarely significant contributors to quality compared with the following primary quality mechanisms & indicators, which should be emphasized in the Management process :-

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

2) Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant usecases.

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

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

5) Change management metrics for objective insight into multiple perspective change trends & convergence or divergence from quality and progress goals.

→ An Inspection focused on resolving an existing issue can be effective way to determine cause or arrive at a resolution once the cause is understood.

→ Significant or substantial design errors or ~~undesired~~ architecture issues are rarely obvious from a superficial review unless the inspection is narrowly focused on a particular issue. And most of the inspections are superficial.

→ Today's systems are highly complex, with innumerable components, concurrent execution, distributed resources, and other equally demanding dimensions of complexity.

→ Consequently, random human inspections tend to degenerate into comments on style & first-order semantic issues. They rarely result in the discovery of real performance bottlenecks, serious control issues (such as deadlocks, races, or resource contention), or architectural weaknesses (such as flaws in scalability, reliability, or interoperability).

→ In all but trivial cases, Architectural issues are exposed only through more rigorous engineering activities as the following:-

1) Analysis, prototyping, or Experimentation

2) Constructing Design models.

3) committing the current state of the design model to an Executable implementation

4) Demonstrating the current implementation strengths and weaknesses in the context of critical subsets of the use cases & scenarios.

5) Incorporating lessons learned back into the models, use cases, implementations and plans.

→ Quality Assurance is everyone's responsibility and should be integral to ~~all~~ almost all process activities instead of a separate discipline performed by quality assurance specialists.

→ Evaluating and Assessing the quality of the evolving engineering baselines should be the job of an engineering team that is independent of the architecture and development team. Their life-cycle assessment of the evolving artifacts would typically include change management, Trend Analysis, and testing, as well as Inspection.

The old way and the New.

over the past two decades there have been a significant re-engineering of the software development process. Many of the conventional management and technical practices have been replaced by new approaches that combine recurring themes of successful project experience with advances in software Engineering Technology.

→ In the commercial software Industry, the combination of competitive pressures, profitability, diversity of customers, and rapidly changing technology caused many organisations to initiate new management approaches.

4.1. principles of Conventional Software Engineering:

There are many descriptions of Engineering software 'the old way'. After years of s/w development experience, the s/w industry has learned many lessions & formulated many principles.

→ This section describes one view of Today's s/w Engineering principles as a benchmark for introducing the primary themes.

→ It describes Top 30 principles, of the conventional wisdom within the software Industry.

1) Make Quality #1. Quality must be quantified and mechanisms put into place to motivate its achievement.

Defining quality commensurate with the project at hand is important but is not easily done at the outset of a project.

Consequently, a modern process framework strives to understand the trade-offs among features, quality, cost, and schedule as early in the life cycle as possible.

Until this understanding is achieved, it is not possible to specify or manage the Achievement of Quality.

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.

↳ This is mostly Redundant with the others.

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

This is a key tenet of a Modern process framework, and there must be several mechanisms to involve the customer throughout the life cycle. Depending on the domain, these mechanisms may include demonstrable prototypes, demonstration-based milestones, and alpha/beta Releases.

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.

This gives issues involved with the conventional requirements specification process. parameters of the problem become more tangible as a solution evolves. Modern process framework evolves the problem and the solution together until the problem is well enough understood to commit to full production.

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

It has Two ways!-

1) The Requirements precede the architecture rather than evolving together.

2) The Architecture is incorporated in the requirements specification. while a modern process clearly promotes the analysis of design alternatives, these activities are done concurrently with requirement specification, and the notations and artifacts for requirements and architecture are explicitly decoupled.

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, volatily of requirements, and the extent to which requirements are well understood.

It's true that no individual process is universal. We use process framework to represent a flexible class of processes rather than a single rigid instance.

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 throughout the lifecycle.

This describes an appropriate organisation & recommended languages / notations for the primitive artifacts of the process.

8. Minimise Intellectual distance. To minimise intectuall distance, the software's structure should be as close as possible to the Real-world structure.

This principle has been the primary motivation for the development of object-oriented techniques, component-based development, and visual Modeling.

9. put techniques before tools. An undisciplined software Engineer with a tool becomes a dangerous, undiscipline software Engineer.

It misses two points:

1) A disciplined software Engineer with good tools will outproduce disciplined s/w experts with no tools.

2) one of the best ways to promote, standardize, and deliver good techniques is through automation.

10. Get it right before you make it faster. It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during Initial coding -

Early performance problems in a s/w system are a sure sign of downstream risk. Every successful, non-trivial software project had performance issues arise early in the life cycle.

Almost all immature architectures (especially large-scale ones) have performance issues in their first executable iterations.

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

Today's H/w resources, PC's and automated environments enable automated analyses & testing to be done efficiently throughout the life cycle. Continuous & automated life-cycle testing is a necessity in any modern iterative development.

General, undirected inspections rarely uncover architectural issues or global design trade-offs. When used judiciously and focused on a known issue, inspections are extremely effective at resolving problems.

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, language, 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 asess its applicability to your Environment.

Object orientation, measurement, reuse, process improvement, CASE, prototyping — all these might increase quality, decrease cost, and increase user satisfication.

→ 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 s/w 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 designs.

It takes more than good methods, tools, and components to succeed. It also takes good people, good management, and a learning culture that is focused on forward progress even when confronted with numerous & inevitable intermediate setbacks.

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.

understanding the customer's priorities is important, but only in balance with other stakeholder priorities.

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.

The more users see, the better they understand Not all the stakeholders are 100% driven by greed. They know that they have limited, resources and that developers have constraints.

→ Demonstrating Intermediate results is a high visibility activity that is necessary to synchronise stakeholder expectations

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, ~~architects~~ architectures, interfaces, or algorithms rarely work for the first time.

19: Design for change. The Architectures, components, and specification techniques you use must accomodate change.

Basically it says that we must predict the future & construct a framework that can accomdate change that is not yet well-defined.

→ It is difficult to predict the sorts of changes that are tekely to occur in a system's life cycle is a useful exercise in Risk

management and a recurring theme of successful software projects.

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!

High-level Architecture documents can be extremely helpful if they are written crisply and concisely, but the primary artifacts used by the engineering team are the design notations, source code, and test baselines.

→ Software Artifacts should be mostly self-documenting.

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

A mature process must be well established, automated, and instrumented. Iterative development projects require extensive Automation

22. Avoid tricks. Many programmers love to create programs with Tricky — 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.

Component-based design, object-oriented design, and modern design and programming notations have advanced this principle into mainstream. Encapsulation is as fundamental a technique to a S/w Engineer as mathematics is to a physicist.

24. Use coupling and cohesion. coupling & cohesion are the best ways to measure software's inherent maintainability & adaptability.

Modern metrics for addressing maintainability and adaptability are centered on measuring the amount of s/w scrap and Rework.

→ cohesive components with minimal coupling are more easily adapted with less scrap & Rework.

25. Use the McCabe complexity measure. Although there are many metrics available to report the inherent complexity of s/w, none is as intuitive & easy to use as Tom McCabe's.

The Really complex stuff is obvious, and it is rare to see these complexity measures used in field applications to manage a project or make decisions.

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

On one hand, an independent test team offers an objective perspective. On the other hand, s/w developers need to take ownership of the quality of their products. Developers should test their own software, and so should a separate team.

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 & fix it one way to do this is to analyse the causes of errors as they are detected.

This would be restated as

1) Don't be Afraid to make errors in the Engineering Stage.

2) Analyse the cause for Errors in the production stage.

28. Realise that software's entropy increases. Any software system that undergoes continuous change will grow in complexity and will become more & more disorganised.

1) Almost all S/w systems undergo continuous change, and the sign of a poor architecture is that its entropy increases in a way that is difficult to manage.

2) Entropy tends to increase dangerously when interfaces are changed for tactical reasons.

3) The Integrity of an Architecture is primarily Strategic & inherent in its interfaces, and it must be controlled with intense scrutiny. Modern change management tools force a project to respect & enforce interface integrity.

4) A Quality Architecture is one in which entropy increases minimally and change can be accommodated with stable, predictable results. An Ideal architecture would permit change without any increase in Entropy.

29. People and Time are not interchangeable. Measuring a project solely by person-months makes little sense.

This principle is timeless.

30. Expect Excellence. Your employees will do much better if you have high expectations for them.

This principle applies to all disciplines, not just software Management.

4.2 Principles of Modern software Management :

10 principles of Modern software Management.

1, Base the process on an Architecture-first approach. This requires that a demonstrable balance be achieved among the driving requirements, the architecturally significant design decisions, and the life-cycle plans before the resources are committed for full-scale development.

2, Establish an iterative life-cycle process, that confronts risk early. With today's sophisticated software 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 a balanced treatment of all stakeholder objectives.

3, Transition design methods to emphasize component-based development

Moving from a line-of-code mentality to a component-based mentality is necessary to reduce the amount of human generated source code & custom development.

A component is a cohesive set of preexisting lines of code, either in source or executable format, with a defined interface & behaviour.

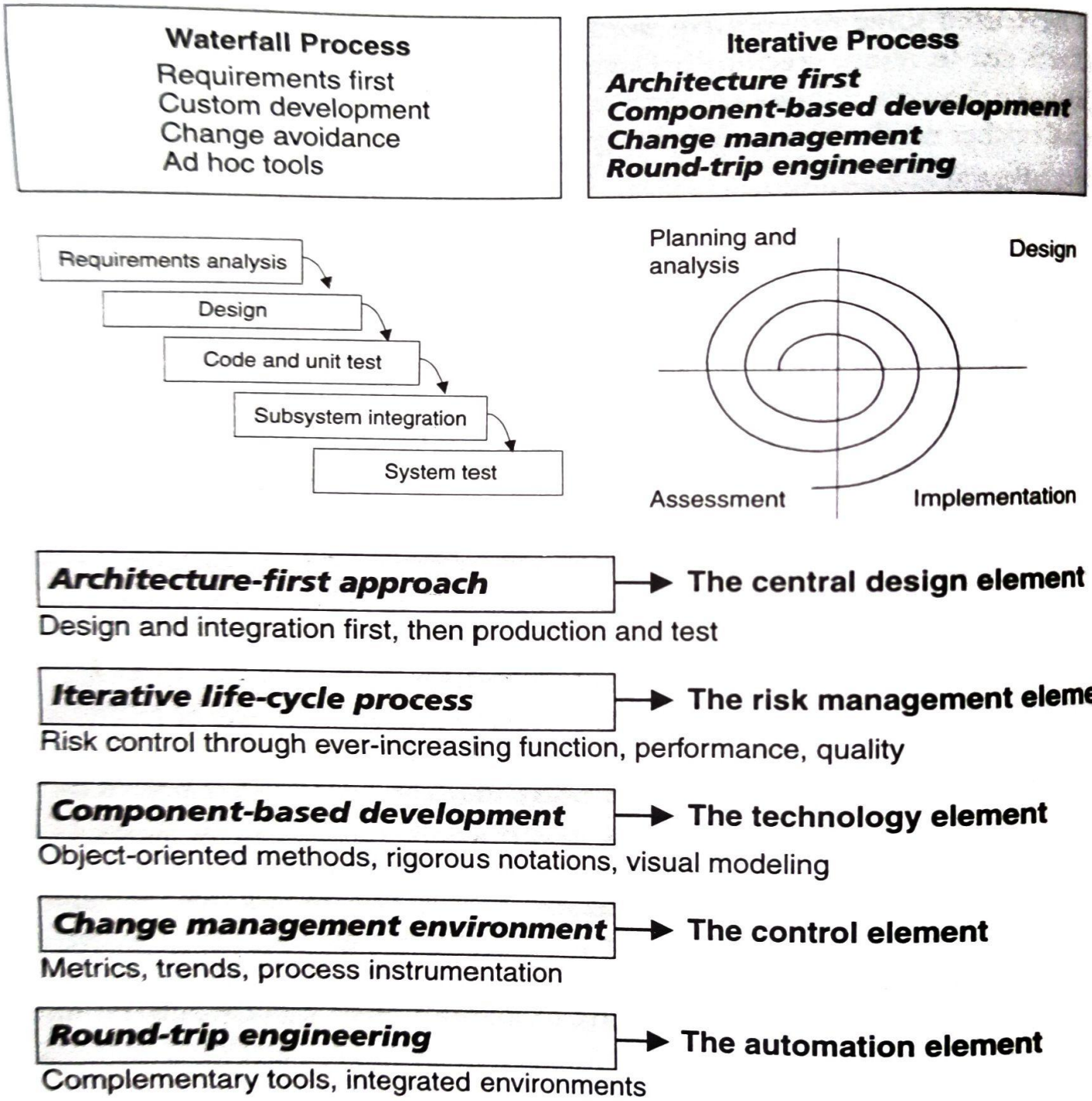


FIGURE 4-1. *The top five principles of a modern process*

integrated into all activities & teams.

8. Use a demonstration-based approach to access intermediate artifacts.

Transitioning the current state-of-the product artifacts (whether the artifact is an early prototype, a base-line architecture, or a beta capability) into an executable demonstration of relevant scenarios stimulates earlier convergence on integration, a more tangible understanding of design toade-offs, and earlier elimination of Architectural defects.

9. plan Intermediate releases in groups of usage scenarios with evolving levels of detail

It is essential that the S/w management process drive toward early and continuous demonstrations within the operational context of the system, namely its use cases.

The evolution of project increments and iterations must be commensurate with the current level of understanding of the requirements & Architecture.

10. Establish a configurable process that is economically scalable.

No single process is suitable for all software developments. The process must ensure that there is economy of scale and return on investment by exploiting a common process spirit, extensive process automation, and common architecture patterns and components.

TABLE 4-1. *Modern process approaches for solving conventional problems*

CONVENTIONAL PROCESS: TOP 10 RISKS	IMPACT	MODERN PROCESS: INHERENT RISK RESOLUTION FEATURES
1. Late breakage and excessive scrap/rework	Quality, cost, schedule	Architecture-first approach Iterative development Automated change management Risk-confronting process
2. Attrition of key personnel	Quality, cost, schedule	Successful, early iterations Trustworthy management and planning
3. Inadequate development resources	Cost, schedule	Environments as first-class artifacts of the proc Industrial-strength, integrated environments Model-based engineering artifacts Round-trip engineering
4. Adversarial stakeholders	Cost, schedule	Demonstration-based review Use-case-oriented requirements/testing
5. Necessary technology insertion	Cost, schedule	Architecture-first approach Component-based development
6. Requirements creep	Cost, schedule	Iterative development Use case modeling Demonstration-based review
7. Analysis paralysis	Schedule	Demonstration-based review Use-case-oriented requirements/testing
8. Inadequate performance	Quality	Demonstration-based performance assessment Early architecture performance feedback
9. Overemphasis on artifacts	Schedule	Demonstration-based assessment Objective quality control
10. Inadequate function	Quality	Iterative development Early prototypes, incremental releases

4.3 Transitioning to an Iterative process

Modern software development processes have moved away from the conventional waterfall model, in which each stage of the development process is dependent on completion of the previous stage.

→ Development then proceeds as a series of iterations, building on the core architecture until the desired levels of functionality, performance, and robustness are achieved. (These iterations have been called spirals, increments, generations, or releases)

→ An iterative process emphasizes the whole system rather than the individual parts. Risk is reduced early in the life cycle through continuous integration & refinement of requirements, architecture, and plans.

→ The Economic benefits inherent in transitioning from the conventional waterfall model to an iterative development process are significant but difficult to quantify.

→ As one benchmark of the expected economic impact of process improvement, consider the process exponent parameters of the COCOMO II model. This model can range from 1.01 (virtually no diseconomy of scale) to 1.26 (significant diseconomy of scale).

→ The parameters that govern the value of the process exponent are application precedentedness, process flexibility, architecture risk resolution, team cohesion, and S/W process maturity.

→ Following map the process exponent parameters of COCOMO II to Top 10 principles of a modern process.

1, Application precendentedness:-

Domain experience is a critical factor in understanding how to plan, and execute a s/w development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental.

This is one of the primary reasons that the s/w industry has moved to an iterative life-cycle process.

2, process flexibility:-

Development of modern s/w is characterised by such a broad solution space & so many interrelated concerns that there is a paramount need for continuous incorporation of changes.

These changes may be inherent in the problem understanding, the solution space, or the plans. project artifacts must be supported by efficient change Management commensurate with project needs.

→ A configurable process that allows a common framework to be adapted across a range of projects is necessary to achieve a s/w return on Investment.

3, Architecture Risk Resolution:-

Architecture-first development is a crucial theme underlying a successful iterative development. A project team develops & stabilizes an architecture before developing all the components

that make up the entire suite of applications components.

→ An Architecture-first and component-based development approach forces the infrastructure, common mechanisms, & control mechanisms to be elaborated early in the lifecycle & drives all component make/buy decisions into the Architecture process

4. Team Cohesion:-

Successful teams are cohesive, and cohesive teams are successful. Cohesive teams avoid sources of project turbulence and entropy that may result from difficulties in synchronising project stakeholder expectations.

→ The main reason for this is miscommunication through paper documents & advances in technology (such as programming lang, UML & visual modeling) have enabled more rigorous & understandable notations for communicating S/w Engineering information, particularly in the requirements & design artifacts that previously were adhoc & based completely on paper exchange.

5. Software process Maturity:-

The Software Engineering Institute's capability Maturity Model (CMM) is a well-accepted benchmark for s/w process assessment. Software process maturity is crucial for avoiding s/w development risks & exploiting the organisation's s/w assets & lessons learned. Just as domain experience is crucial for avoiding the application risks & exploiting the available domain assets & lessons learned, ~~process maturity~~ is The truly mature processes are enabled through an integrated environment that provides the appropriate level of automation to instrument the process for objective quality control.