SOFTWARE PROJECT MANAGEMENT

UNIT - 1

Conventional Software Management

The best thing about software is its flexibility: It can be programmed to do almost he worst thing about software is its flexibility: The “almost anything” characteristic has made it difficult to plan, monitor, and control software development. In the mid-1990s, three important analyses were performed on the software engineering industry.

All three analyses given the same general conclusion:

“The success rate for software projects is very low”.

They summarized as follows:

1. Software development is still highly unpredictable. Only 10% of software projects are delivered successfully within initial budget and scheduled time.
2. Management discipline is more differentiator in success or failure than are technology advances.
3. The level of software scrap and rework is indicative of an immature process.

Software management process framework:

WATERFALL MODEL

It is the baseline process for most conventional software projects have used.

We can examine this model in two ways:

i. IN THEORY
ii. IN PRACTICE

IN THEORY:

In 1970, Winston Royce presented a paper called “Managing the Development of Large Scale Software Systems” at IEEE WESCON.

Where he made three primary points:

1. There are two essential steps common to the development of computer programs:

- Analysis
- coding

2. In order to manage and control all of the intellectual freedom associated with software development one should follow

The following steps:

1. System requirements definition
2. Software requirements definition
3. Program design and testing

These steps addition to the analysis and coding steps

a) ***Since*** the testing phase is at the end of the development cycle in the waterfall model, it may be risky and invites failure. So we need to do either the requirements must be modified or a substantial design changes is warranted by breaking the software in to different pieces.

b) ***Complete program design before analysis and coding begin (program design comes first):***

- By this technique, the program designer gives surety that the software will not fail because of storage, timing, and data fluctuations.
- Begin the design process with program designer, not the analyst or programmers.
- Write an overview document that is understandable, informative, and current so that every worker on the project can gain an elemental understanding of the system.

c) ***Maintain current and complete documentation (Document the design):***

- It is necessary to provide a lot of documentation on most software programs.
- Due to this, helps to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

c) ***Do the job twice, if possible (Do it twice):***
- If a computer program is developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned.
- "Do it N times" approach is the principle of modern-day iterative development.

d) Plan, control, and monitor testing:
- The biggest user of project resources is the test phase. This is the phase of greatest risk in terms of cost and schedule.
- In order to carryout proper testing the following things to be done:
 Employ a team of test specialists who were not responsible for the original design.
 Employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses.
 Test every logic phase.
 iv) Employ the final checkout on the target computer.

e) Involve the customer:
- It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery by conducting some reviews such as,

- Preliminary software review during preliminary program design step.
- Critical software review during program design.

- Final software acceptance review following testing.

IN PRACTICE:

- Whatever the advices that are given by the software developers and the theory behind the waterfall model, some Software projects still practice the conventional software management approach.

Projects intended for trouble frequently exhibit the following symptoms:

I) Protracted (delayed) integration

- In the conventional model, the entire system was designed on paper, then implemented all at once, then integrated. Only at the end of this process was it possible to perform system testing to verify that the fundamental architecture was sound.
- Here the testing activities consume 40% or more life-cycle resources.

ACTIVITY COST

- Management 5%
- Requirements 5%
- Design 10%
- Code and unit testing 30%
- Integration and test 40%
- Deployment 5%
- Environment 5%

ii) Late Risk Resolution

- A serious issues associated with the waterfall life cycle was the lack of early risk resolution. The risk profile of a waterfall model is,
- It includes four distinct periods of risk exposure, where risk is defined as "the probability of missing a cost, schedule, feature, or quality goal".

iii) Requirements-Driven Functional Decomposition

-Traditionally, the software development process has been requirement-driven: An attempt is made to provide a Precise requirements definition and then to implement exactly those requirements.

-This approach depends on specifying requirements completely and clearly before other development activities begin.

- It frankly treats all requirements as equally important.
- Specification of requirements is a difficult and important part of the software development process.

iv) Adversarial Stakeholder Relationships

The following sequence of events was typical for most contractual software efforts:

-The contractor prepared a draft contact-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.

-The customer was expected to provide comments (within 15 to 30 days)

-The contractor integrated these comments and submitted a final version for approval (within 15 to 30 days)

Project Stakeholders:

- Stakeholders are the people involved in or affected by project activities
- Stakeholders include the project sponsor and project team

- support staff
- customers
- users
- suppliers
- opponents to the project

v) Focus on Documents and Review Meetings

- The conventional process focused on various documents that attempted to describe the software product.
- Contractors produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software.
- Most design reviews resulted in low engineering and high cost in terms of the effort and schedule involved in their preparation and conduct.

CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

Barry Boehm's Top 10 "Industrial Software Metrics":

1) Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.
2) You can compress software development schedules 25% of nominal (small), but no more.
3) For every $1 you spend on development, you will spend $2 on maintenance.
4) Software development and maintenance costs are primarily a function of the number of source lines of code.
5) Variations among people account for the biggest difference in software productivity.
6) The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.
7) Only about 15% of software development effort is devoted to programming.
8) Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products cost 9 times as much.
9) Walkthroughs catch 60% of the errors.
10) 80% of the contribution comes from 20% of the contributors.

- 80% of the engineering is consumed by 20% of the requirements.
- 80% of the software cost is consumed by 20% of the components.
- 80% of the errors are caused by 20% of the components.
- 80% of the software scrap and rework is caused by 20% of the errors.
- 80% of the resources are consumed by 20% of the components.
- 80% of the engineering is accomplished by 20% of the tools.
- 80% of the progress is made by 20% of the people.

Part-2

Evolution of Software Economics

Economics means System of interrelationship of money, industry and employment.

SOFTWARE ECONOMICS:

The cost of the software can be estimated by considering the following things as parameters to a function.

1) Size: Which is measured in terms of the number of Source Lines of Code or the number of function points required to develop the required functionality?

2) Process: Used to produce the end product, in particular the ability of the process is to avoid nonvalue adding activities (rework, bureaucratic delays, and communications overhead).

3) Personnel: The capabilities of software engineering personnel, and particularly their experience with the computer science issues and the application domain issues of the project.
4) Environment: Which is made up of the tools and techniques available to support efficient software development and to automate the process?
5) Quality: It includes its features, performance, reliability, and flexibility. The relationship among these parameters and estimated cost can be calculated by using,
Effort = (Personnel) (Environment) (Quality) (SizeProcess)
One important aspect of software economics is that the relationship between effort and size
Exhibits a diseconomy of scale and is the result of the process exponent being greater than 1.0.
- Converse to most manufacturing processes, the more software you build, the more expensive it is per unit item.
- There are three generations of basic technology advancement in tools, components, and processes are available.

1) **Conventional:** 1960 and 1970, Craftsmanship. Organizations used custom tools, custom
Processes, and virtually all custom components built in primitive languages. Project performance was highly predictable.
2) **Transition:** 1980 and 1990, software engineering. Organizations used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products like, OS, DBMS, Networking and GUI.
3) **Modern practices:** 2000 and later, software production.
- 70% component-based,

- 30% custom

Conventional Transition Modern Practices

- 1960s – 1970s - 1980s –1990s - 2000 and on
- Waterfall model - Process improvement - Iterative development
- Functional design - Encapsulation - based - Component-based
- Diseconomy of scale- Diseconomy of scale- ROI

Environments /tools:

Custom Off-the-shelf, separate Off-the-shelf, Integrated

Size:

- 100% custom 30% component-based 70% component-based
- 70% custom 30% custom

Process:

- Ad hoc Repeatable Managed/measured

Typical Project Performance:

- Always: Infrequently: Usually:
- Over budget On budget On budget
- Over schedule On schedule On schedule

What Does *Return On Investment - ROI* Mean?

A performance measure used to evaluate the efficiency of an investment or to compare the Efficiency of a number of different investments. To calculate ROI, the benefit (return) of an investment is divided by the cost of the investment; the result is expressed as a percentage or a ratio. The return on investment formula:

• The more the size, the greater are the costs of management overhead, communication, synchronizations among various projects or modules, etc.

Reduce Software Size:

The less software we write, the better it is for project management and for product quality

- The cost of software is not just in the cost of 'coding' alone; it is also in

– Analysis of requirements
– Design
– Review of requirements, design and code
– Test Planning and preparation
– Testing
– Bug fix
– Regression testing
– 'Coding' takes around 15% of development cost

Clearly, if we reduce 15 hrs of coding, we can directly reduce 100 hrs of development effort, and also reduce the project team size appropriately

- Size reduction is defined in terms of human-generated source code. Most often, this might still mean that the computer-generated executable code is at least the same or even more
- Software Size could be reduced by
– Software Re-use
– Use of COTS (Commercial Off-The Shelf Software)
– Programming Languages

PRAGMATIC SOFTWARE ESTIMATION:

- If there is no proper well-documented case study then it is difficult to estimate the cost of the software. It is one of the critical problems in software cost estimation.
- But the cost model vendors claim that their tools are well suitable for estimating iterative Development projects.

In order to estimate the cost of a project the following three topics should be considered,

- Which cost estimation model to use?
- Whether to measure software size in SLOC or function point.
- What constitutes a good estimate?

There is a lot of software cost estimation models are available such as, COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20. Of which COCOMO is one of the most open and well-documented cost estimation models.

The software size can be measured by using 1) SLOC 2) Function points.

- SLOC worked well in applications that were custom built why because of easy to automate and Instrument.
- Now a days there are so many automatic source code generators are available and there are so Many advanced higher-level languages are available. So SLOC is a uncertain measure.

The main advantage of function points is that this method is independent of the technology and is therefore a much better primitive unit for comparisons among projects and organizations.

The main disadvantage of function points is that the primitive definitions are abstract and measurements are not easily derived directly from the evolving artifacts.

- Function points is more accurate estimator in the early phases of a project life cycle. In later phases, SLOC becomes a more useful and precise measurement basis of various metrics perspectives.
- The most real-world use of cost models is bottom-up rather than top-down.
- The software project manager defines the target cost of the software, and then manipulates the parameters and sizing until the target cost can be justified.

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. Whereas 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 3-1. *Important trends in improving software economics*

COST MODEL PARAMETERS	TRENDS
Size Abstraction and component-based development technologies	Higher order languages (C++, Ada 95, Java, Visual Basic, etc.) Object-oriented (analysis, design, programming) Reuse Commercial components
Process Methods and techniques	Iterative development Process maturity models Architecture-first development Acquisition reform
Personnel People factors	Training and personnel skill development Teamwork Win-win cultures
Environment Automation technologies and tools	Integrated tools (visual modeling, compiler, editor, debugger, change management, etc.) Open systems Hardware platform performance Automation of coding, documents, testing, analyses
Quality Performance, reliability, accuracy	Hardware platform performance Demonstration-based assessment Statistical quality control

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 software.

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

1,000,000 lines of assembly language

400,000 lines of C

220,000 lines of Ada 83

175,000 lines of Ada 95 or C++

75,000 lines of Ada 95 or C++ plus integration of several commercial components

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, crating 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.

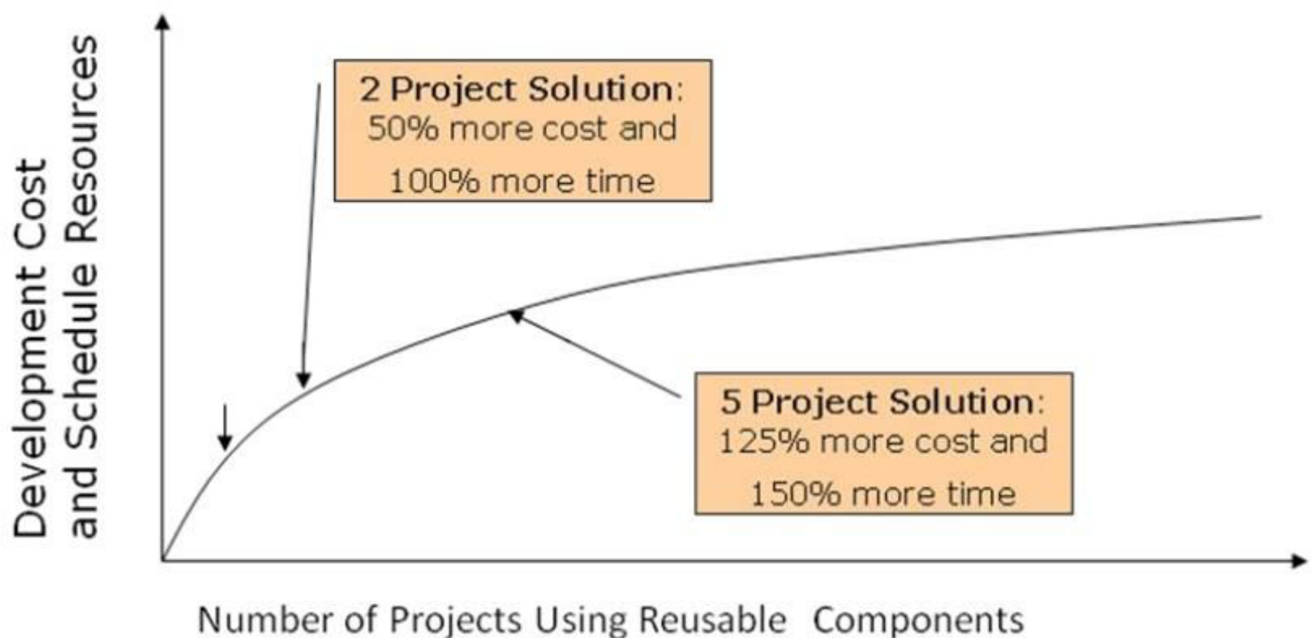
4) The existence of a strong architectural vision.

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 3-3. *Advantages and disadvantages of commercial components versus custom software*

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

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 a 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*

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 Organizational 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 and rework	On budget, on schedule Major milestone progress Release/iteration scrap and 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

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
- 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
2) Mid-life-cycle design review 4) Integration and test

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, straight-forward 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 per-formance feedback, quantitative understanding
Software process rigor	Document-based	Managed, measured, and tool-supported

- 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
- 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

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 was 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:

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, and 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- process able 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 → 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

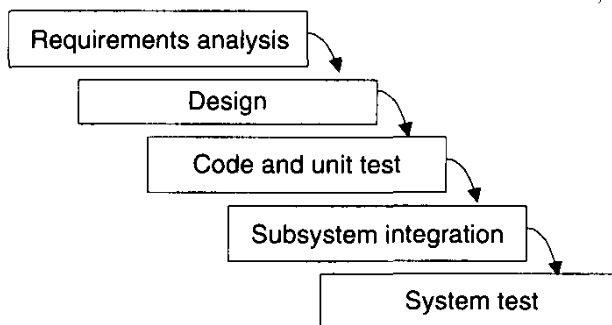
Change management environment → The control element

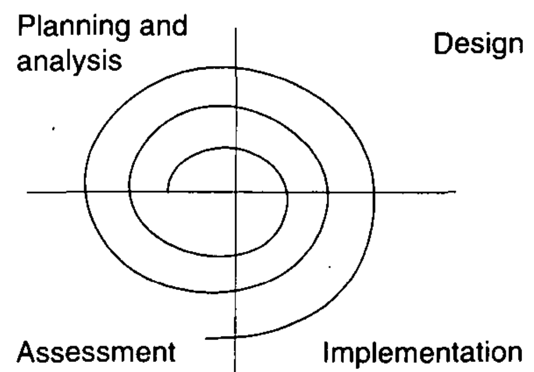
Metrics, trends, process instrumentation

Round-trip engineering → The automation element

Complementary tools, integrated environments

Waterfall Process	**Iterative Process**
Requirements first	***Architecture first***
Custom development	***Component-based development***
Change avoidance	***Change management***
Ad hoc tools	***Round-trip engineering***





UNIT III

- If there is a well defined separation between "research and development" activities and "production" activities then the software is said to be in successful development process.
- Most of the software's fail due to the following characteristics ,
1) An overemphasis on research and development.
2) An overemphasis on production.

ENGINEERING AND PRODUCTION STAGES :

To achieve economies of scale and higher return on investment, we must move toward a software manufacturing process which is determined by technological improvements in process automation and component based development.

There are two stages in the software development process

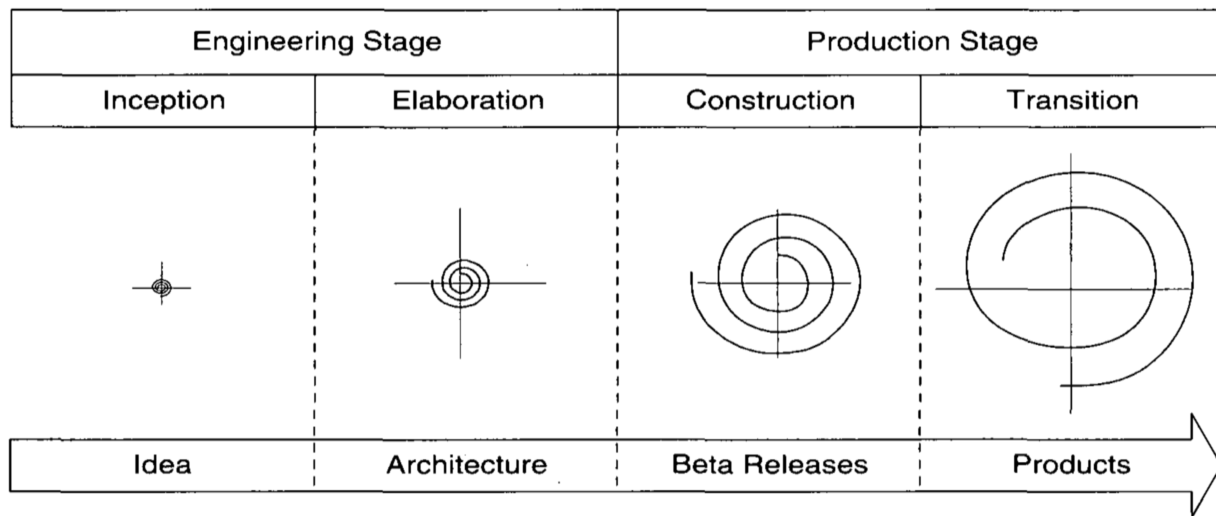
1) **The engineering stage:** Less predictable but smaller teams doing design and production activities. This stage is decomposed into two distinct phases' *inception* and *elaboration*.

2) **The production stage:** More predictable but larger teams doing construction, test, and deployment activities. This stage is also decomposed into two distinct phases' *construction* and *transition*.

TABLE 5-1. *The two stages of the life cycle: engineering and production*

LIFE-CYCLE ASPECT	ENGINEERING STAGE EMPHASIS	PRODUCTION STAGE EMPHASIS
Risk reduction	Schedule, technical feasibility	Cost
Products	Architecture baseline	Product release baselines
Activities	Analysis, design, planning	Implementation, testing
Assessment	Demonstration, inspection, analysis	Testing
Economics	Resolving diseconomies of scale	Exploiting economies of scale
Management	Planning	Operations

These four phases of lifecycle process are loosely mapped to the conceptual framework of the spiral model is as shown in the following figure.



- In the above figure the size of the spiral corresponds to the inactivity of the project with respect to the breadth and depth of the artifacts that have been developed.
- This inertia manifests itself in maintaining artifact consistency, regression testing, documentation, quality analyses, and configuration control.
- Increased inertia may have little, or at least very straightforward, impact on changing any given discrete component or activity.
- However, the reaction time for accommodating major architectural changes, major requirements changes, major planning shifts, or major organizational perturbations clearly increases in subsequent phases.

INCEPTION PAHSE:

The main goal of this phase is to achieve agreement among stakeholders on the life-cycle objectives for the project.

PRIMARY OBJECTIVES

1) Establishing the project's scope and boundary conditions
2) Distinguishing the critical use cases of the system and the primary scenarios of operation
3) Demonstrating at least one candidate architecture against some of the primary scenarios
4) Estimating cost and schedule for the entire project
5) Estimating potential risks

ESSENTIAL ACTIVITIES:

1) Formulating the scope of the project
2) Synthesizing the architecture
3) Planning and preparing a business case

ELABORATION PHASE

- It is the most critical phase among the four phases.
- Depending upon the scope, size, risk, and freshness of the project, an executable architecture prototype is built in one or more iterations.
- At most of the time the process may accommodate changes, the elaboration phase activities must ensure that the architecture, requirements, and plans are stable. And also the cost and schedule for the completion of the development can be predicted within an acceptable range.

PRIMARY OBJECTIVES
1) Base lining the architecture as rapidly as practical
2) Base lining the vision
3) Base lining a high-reliability plan for the construction phase
4) Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonable time.

ESSENTIAL ACTIVITIES
1) Elaborating the vision
2) Elaborating the process and infrastructure
3) Elaborating the architecture and selecting components

CONSTRUCTION PHASE

During this phase all the remaining components and application features are integrated into the application, and all features are thoroughly tested. Newly developed software is integrated where ever required.
- If it is a big project then parallel construction increments are generated.

PRIMARY OBJECTIVES
1) Minimizing development costs
2) Achieving adequate quality as rapidly as practical
3) Achieving useful version (alpha, beta, and other releases) as rapidly as practical

ESSENTIAL ACTIVITIES
1) Resource management, control, and process optimization
2) Complete component development and testing evaluation criteria
3) Assessment of product release criteria of the vision

TRANSITION PHASE

Whenever a project is grown-up completely and to be deployed in the end-user domain this phase is called transition phase. It includes the following activities:
1) Beta testing to validate the new system against user expectations
2) Beta testing and parallel operation relative to a legacy system it is replacing
3) Conversion of operational databases
4) Training of users and maintainers

PRIMARY OBJECTIVES
1) Achieving user self-supportability
2) Achieving stakeholder concurrence
3) Achieving final product baseline as rapidly and cost-effectively as practical

ESSENTIAL ACTIVITIES
1) Synchronization and integration of concurrent construction increments into consistent deployment baselines
2) Deployment-specific engineering
3) Assessment of deployment baselines against the complete vision and acceptance criteria in the requirement set.

Artifacts of the Process

- Conventional s/w projects focused on the sequential development of s/w artifacts:
- Build the requirements
- Construct a design model traceable to the requirements &
- Compile and test the implementation for deployment.
-This process can work for small-scale, purely custom developments in which the design representation, implementation representation and deployment representation are closely aligned.
- This approach is doesn't work for most of today's s/w systems why because of having complexity and are composed of numerous components some are custom, some reused, some commercial products.

THE ARTIFACT SETS

In order to manage the development of a complete software system, we need to gather distinct collections of information and is organized into *artifact sets*.
- *Set* represents a complete aspect of the system where as *artifact* represents interrelated information that is developed and reviewed as a single entity.
- The artifacts of the process are organized into five sets:
1) Management 2) Requirements 3) Design
4) Implementation 5) Deployment
here the management artifacts capture the information that is necessary to synchronize stakeholder expectations. Whereas the remaining four artifacts are captured rigorous notations that support automated analysis and browsing**.**

Requirements Set	Design Set	Implementation Set	Deployment Set
1. Vision document 2. *Requirements* model(s)	1. Design model(s) 2. *Test model* 3. Software architecture description	1. Source code *baselines* 2. Associated compile-time files 3. Component executables	1. Integrated product *executable* baselines 2. Associated run-time files 3. User manual

Management Set

Planning Artifacts	Operational Artifacts
1. Work breakdown structure 2. Business case 3. Release specifications 4. Software development plan	5. Release descriptions 6. Status assessments 7. Software change order database 8. Deployment documents 9. Environment

THE MANAGEMENT SET

It captures the artifacts associated with process planning and execution. These artifacts use ad hoc notation including text, graphics, or whatever representation is required to capture the "contracts" among,
- **project personnel:**
project manager, architects, developers, testers, marketers, administrators
- stakeholders:
funding authority, user, s/w project manager, organization manager, regulatory agency & between project personnel and stakeholders.

Management artifacts are evaluated, assessed, and measured through a combination of
1) Relevant stakeholder review.
2) Analysis of changes between the current version of the artifact and previous versions.
3) Major milestone demonstrations of the balance among all artifacts.

THE ENGINEERING SETS:

1) REQUIREMENT SET:
- The requirements set is the primary engineering context for evaluating the other three engineering artifact sets and is the basis for test cases.
- **Requirement artifacts are evaluated, assessed, and measured through a combination of**
1) Analysis of consistency with the release specifications of the mgmt set.
2) Analysis of consistency between the vision and the requirement models.
3) Mapping against the design, implementation, and deployment sets to
evaluate the consistency and completeness and the semantic balance between information in the different sets.
4) Analysis of changes between the current version of the artifacts and previous versions.
5) Subjective review of other dimensions of quality.

2) DESIGN SET:
- UML notations are used to engineer the design models for the solution.
- It contains various levels of abstraction and enough structural and behavioral information to determine a bill of materials.
- Design model information can be clearly and, in many cases, automatically translated into a subset of the implementation and deployment set artifacts.
The design set is evaluated, assessed, and measured through a combination of
1) Analysis of the internal consistency and quality of the design model.
2) Analysis of consistency with the requirements models.
3) Translation into implementation and deployment sets and notations to evaluate the consistency and completeness and semantic balance between information in the sets.
4) Analysis of changes between the current version of the design model and previous versions.
5) Subjective review of other dimensions of quality.

3) IMPLEMENTATION SET:

- The implementation set include source code that represents the tangible implementations of components and any executables necessary for stand-alone testing of components.
- Executables are the primitive parts that are needed to construct the end product, including custom components, APIs of commercial components.
- Implementation set artifacts can also be translated into a subset of the deployment set. Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of

1) Analysis of consistency with design models
2) Translation into deployment set notations to evaluate consistency and completeness among artifact sets.
3) Execution of stand-alone component test cases that automatically compare expected results with actual results.
4) Analysis of changes b/w the current version of the implementation set and previous versions.
5) Subjective review of other dimensions of quality.

4) DEPLOYMENT SET:

- It includes user deliverables and machine language notations, executable software, and the build scripts, installation scripts, and executable target-specific data necessary to use the product in its target environment.

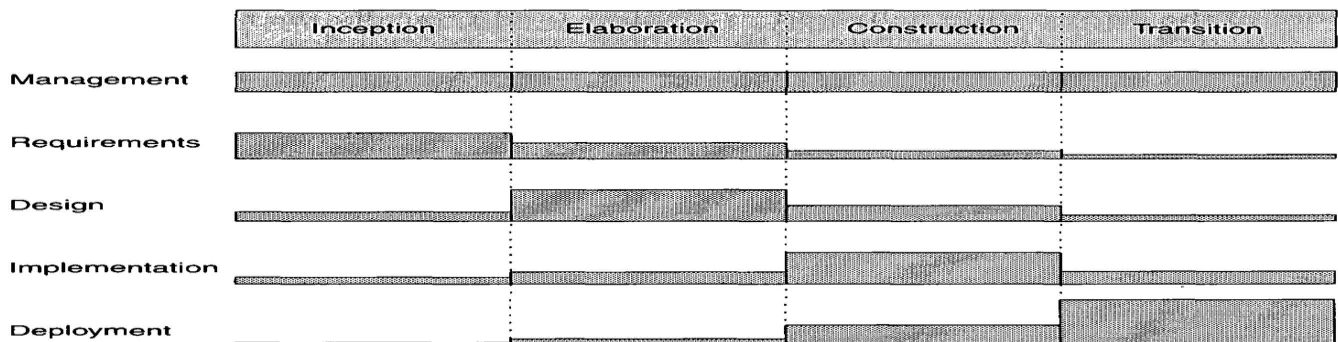
Deployment sets are evaluated, assessed, and measured through a combination of

1) Testing against the usage scenarios and quality attributes defined in the requirements set to evaluate the consistency and completeness and the semantic balance between information in the two sets.
2) Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system.
3) Testing against the defined usage scenarios in the user manual such as installation, user-oriented dynamic reconfiguration, mainstream usage, and anomaly management.
4) Analysis of changes b/w the current version of the deployment set and previous versions.
5) Subjective review of other dimensions of quality.

Each artifact set uses different notations to capture the relevant artifact.

1) **Management set notations** (ad hoc text, graphics, use case notation) capture the plans, process, objectives, and acceptance criteria.

2) **Requirement notation** (structured text and UML models) capture the engineering context and the operational concept.
3) **Implementation notations** (software languages) capture the building blocks of the solution in human-readable formats.
4) **Deployment notations** (executables and data files) capture the solution in machine-readable formats.



IMPLEMENTATION SET VERSUS DEPLOYMENT SET

- The structure of the information delivered to the user (testing people) is very different from the structure of the source code implementation.
- Engineering decisions that have impact on the quality of the deployment set but are relatively incomprehensible in the design and implementation sets include:
1) Dynamically reconfigurable parameters such as buffer sizes, color palettes, number of servers, number of simultaneous clients, data files, run-time parameters.
2) Effects of compiler/link optimizations such as space optimization versus speed optimization.
3) Performance under certain allocation strategies such as centralized versus distributed, primary and shadow threads, dynamic load balancing.
4) Virtual machine constraints such as file descriptors, garbage collection, heap size, maximum record size, disk file rotations.
5) Process-level concurrency issues such as deadlock and race condition.
6) Platform-specific differences in performance or behavior.

ARTIFACTS EVOLUTION OVER THE LIFE CYCLE

- Each state of development represents a certain amount of precision in the final system description.
- Early in the lifecycle, precision is low and the representation is generally high. Eventually, the precision of representation is high and everything is specified in full detail.
- At any point in the lifecycle, the five sets will be in different states of completeness. However, they should be at compatible levels of detail and reasonably traceable to one another.
- Performing detailed traceability and consistency analyses early in the life cycle i.e. when precision is low and changes are frequent usually has a low ROI.

Inception phase: It mainly focuses on critical requirements, usually with a secondary focus on an initial deployment view, little implementation and high-level focus on the design architecture but not on design detail.
Elaboration phase: It include generation of an executable prototype, involves subsets of development in all four sets. A portion of all four sets must be evolved to some level of completion before an architecture baseline can be established.

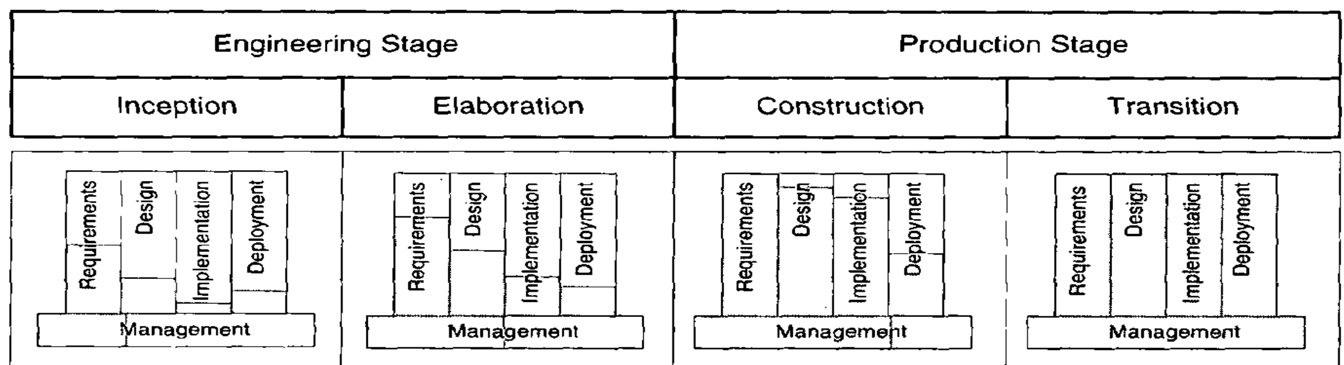


Fig: Life-Cycle evolution of the artifact sets

Construction: Its main focus on design and implementation. In the early stages the main focus is on the depth of the design artifacts. Later, in construction, realizing the design in source code and individually tested components. This stage should drive the requirements, design, and implementation sets almost to completion. Substantial work is also done on the deployment set, at least to test one or a few instances of the programmed system through alpha or beta releases.
Transition: The main focus is on achieving consistency and completeness of the deployment set in the context of another set. Residual defects are resolved, and feedback from alpha, beta, and system testing is incorporated.

TEST ARTIFACTS:

Testing refers to the explicit evaluation through execution of deployment set components under a controlled scenario with an expected and objective outcome.
- Whatever the document-driven approach that was applied to software development is also followed by the software testing people.
- Development teams built requirements documents, top-level design documents, and detailed design documents before constructing any source files or executable files.
- In the same way test teams built system test plan documents, unit test plan documents, and unit test procedure documents before building any test drivers, stubs, or instrumentation.
- This document-driven approach caused the same problems for the test activities that it did for the development activities.
- One of the truly tasteful beliefs of a modern process is to use exactly the same sets, notations, and artifacts for the products of test activities as are used for product development.
- The test artifacts must be developed concurrently with the product from inception through deployment. *i.e. Testing a full-life-cycle activity, not a late life-cycle activity.*
- The test artifacts are communicated, engineered, and developed within the same artifact sets as the developed product.
- The test artifacts are implemented in programmable and repeatable formats as software programs.
- The test artifacts are documented in the same way that the product is documented.
- Developers of the test artifacts use the same tools, techniques, and training as the software engineers developing the product.
- Testing is only one aspect of the evaluation workflow. Other aspects include inspection, analysis, and demonstration.
- The success of test can be determined by comparing the expected outcome to the actual outcome with well-defined mathematical precision.

MANAGEMENT ARTIFACTS:

WORK BREAKDOWN STRUCTURE

Development of WBS is dependent on product management style, organizational culture, custom performance, financial constraints and several project specific parameters.
• The WBS is the architecture of project plan. It encapsulates change and evolve with appropriate level of details.
• A WBS is simply a hierarchy of elements that decomposes the project plan into discrete work task.
• A WBS provides the following information structure
• A delineation of all significant tasks.
• A clear task decomposition for assignment of responsibilities.
• A framework for scheduling, debugging and expenditure tracking.
• Most systems have first level decomposition subsystem. Subsystems are then decomposed into their components
• Therefore WBS is a driving vehicle for budgeting and collecting cost.
• The structure of cost accountability is a serious project planning constraints.

BUSINESS CASE:

The business case artifact provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans, and backup data necessary to demonstrate the risks and realism of the plans. In large contractual procurements, the business case may be implemented in a full-scale proposal with multiple volumes of information. In a small-scale endeavor for a commercial product, it may be implemented in a brief plan with an attached spreadsheet. The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progresses. Figure 6-4 provides a default outline for a business case.

I. **Context (domain, market, scope)**
II. **Technical approach**
    A. Feature set achievement plan
    B. Quality achievement plan
    C. Engineering trade-offs and technical risks
III. **Management approach**
    A. Schedule and schedule risk assessment
    B. Objective measures of success
IV. **Evolutionary appendixes**
    A. Financial forecast
        1. Cost estimate
        2. Revenue estimate
        3. Bases of estimates

Release Specifications

The scope, plan, and objective evaluation criteria for each baseline release are derived from the vision statement as well as many other sources (make/buy analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds). These artifacts are intended to evolve along with the process, achieving greater fidelity as the life cycle progresses and requirements

I. **Context (domain, market, scope)**
II. **Technical approach**
    A. Feature set achievement plan
    B. Quality achievement plan
    C. Engineering trade-offs and technical risks
III. **Management approach**
    A. Schedule and schedule risk assessment
    B. Objective measures of success
IV. **Evolutionary appendixes**
    A. Financial forecast
        1. Cost estimate
        2. Revenue estimate
        3. Bases of estimates

FIGURE 6-4. *Typical business case outline*

- **I. Iteration content**
- **II. Measurable objectives**
  - A. Evaluation criteria
  - B. Followthrough approach
- **III. Demonstration plan**
  - A. Schedule of activities
  - B. Team responsibilities
- **IV. Operational scenarios (use cases demonstrated)**
  - A. Demonstration procedures
  - B. Traceability to vision and business case

FIGURE 6-5. *Typical release specification outline*

Software Development Plan

The software development plan (SDP) elaborates the process framework into a fully detailed plan. It is the defining document for the project's process. It must comply with the contract (if any), comply with organization standards (if any), evolve along with the design and requirements, and be used consistently across all subordinate organizations doing software development. Two indications of a useful SDP are peri-

- **I. Context (scope, objectives)**
- **II. Software development process**
  - A. Project primitives
    1. Life-cycle phases
    2. Artifacts
    3. Workflows
    4. Checkpoints
  - B. Major milestone scope and content
  - C. Process improvement procedures
- **III. Software engineering environment**
  - A. Process automation (hardware and software resource configuration)
  - B. Resource allocation procedures (sharing across organizations, security access)
- **IV. Software change management**
  - A. Configuration control board plan and procedures
  - B. Software change order definitions and procedures
  - C. Configuration baseline definitions and procedures
- **V. Software assessment**
  - A. Metrics collection and reporting procedures
  - B. Risk management procedures (risk identification, tracking, and resolution)
  - C. Status assessment plan
  - D. Acceptance test plan
- **VI. Standards and procedures**
  - A. Standards and procedures for technical artifacts
- **VII. Evolutionary appendixes**
  - A. Minor milestone scope and content
  - B. Human resources (organization, staffing plan, training plan)

FIGURE 6-6. *Typical software development plan outline*

odic updating (it is not stagnant shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-6 provides a default outline for a software development plan.

Release Descriptions

Release description documents describe the results of each release, including performance against each of the evaluation criteria in the corresponding release specification. Release baselines should be accompanied by a release description document that describes the evaluation criteria for that configuration baseline and provides substantiation (through demonstration, testing, inspection, or analysis) that each criterion has been addressed in an acceptable manner. This document should also include a metrics summary that quantifies its quality in absolute and relative terms (compared to the previous versions, if any). The results of a post-mortem review of any release would be documented here, including outstanding issues, recommendations for process and product improvement, trade-offs in addressing evaluation criteria, follow-up actions, and similar information. Figure 6-7 provides a default outline for a release description.

I. **Context**
   A. Release baseline content
   B. Release metrics
II. **Release notes**
   A. Release-specific constraints or limitations
III. **Assessment results**
   A. Substantiation of passed evaluation criteria
   B. Follow-up plans for failed evaluation criteria
   C. Recommendations for next release
IV. **Outstanding issues**
   A. Action items
   B. Post-mortem summary of lessons learned

FIGURE 6-7. *Typical release description outline*

Status Assessments

Status assessments provide periodic snapshots of project health and status, including the software project manager's risk assessment, quality indicators, and management indicators. Although the period may vary, the forcing function needs to persist. The paramount objective of a good management process is to ensure that the expectations of all stakeholders (contractor, customer, user, subcontractor) are synchronized and consistent. The periodic status assessment documents provide the critical mechanism for managing everyone's expectations throughout the life cycle; for addressing, communicating, and resolving management issues, technical issues, and project risks; and for capturing project history. They are the periodic heartbeat for management atten-

Software Change Order Database

Managing change is one of the fundamental primitives of an iterative development process.
• This flexibility increases the content, quality, and number of iterations that a project can achieve within a given schedule.
• Once software is placed in a controlled baseline, all changes must be formally tracked and managed.
• Most of the change management activities can be automated by automating data entry and maintaining change records online.

Deployment

A deployment document can take many forms. Depending on the project, it could include several document subsets for transitioning the product into operational status. In big contractual efforts in which the system is delivered to a separate maintenance organization, deployment artifacts may include computer system operations manuals, software installation manuals, plans and procedures for cutover (from a legacy system), site surveys, and so forth. For commercial software products, deployment artifacts may include marketing plans, sales rollout kits, and training courses.

Environment

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 automation of the development process. This environment should include requirements management, visual modeling, document automation, host and target programming tools, automated regression testing, integrated change management, and defect tracking. A common theme from successful software projects is that they hire good people and provide them with good tools to accomplish their jobs. Automation of the software development process provides payback in quality, the ability to estimate costs and schedules, and overall productivity using a smaller team. By allowing the designers to traverse quickly among development artifacts and easily keep the artifacts up-to-date, integrated toolsets play an increasingly important role in incremental and iterative development.

Management Artifact Sequences

In each phase of the life cycle, new artifacts are produced and previously developed artifacts are updated to incorporate lessons learned and to capture further depth and breadth of the solution. Some artifacts are updated at each major milestone, others at each minor milestone. Figure 6-8 identifies a typical sequence of artifacts across the life-cycle phases.

ENGINEERING ARTIFACTS

Vision Document

The vision document provides a complete vision for the software system under development and supports the contract between the funding authority and the development organization. Whether the project is a huge military-standard development (whose vision could be a 300-page system specification) or a small, internally funded commercial product (whose vision might be a two-page white paper), every project needs a source for capturing the expectations among stakeholders. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans, and technology. A good vision document should change slowly. Figure 6-9 provides a default outline for a vision document.

- I. **Feature set description**
  - A. Precedence and priority
- II. **Quality attributes and ranges**
- III. **Required constraints**
  - A. External interfaces
- IV. **Evolutionary appendixes**
  - A. Use cases
    1. Primary scenarios
    2. Acceptance criteria and tolerances
  - B. Desired freedoms (potential change scenarios)

FIGURE 6-9. *Typical vision document outline*

Architecture Description

The architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. The architecture can be described using a subset of the design model or as an abstraction of the design model with supplementary material, or a combination of both. As examples of these two forms of descriptions, consider the architecture of this book:

- A subset form could be satisfied by the table of contents. This description of the architecture of the book is directly derivable from the book itself.
- An abstraction form could be satisfied by a "Cliffs Notes" treatment. (Cliffs Notes are condensed versions of classic books used as study guides by some college students.) This format is an abstraction that is developed separately and includes supplementary material that is not directly derivable from the evolving product.

Software User Manual

The software user manual provides the user with the reference documentation necessary to support the delivered software. Although content is highly variable across application domains, the user manual should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description, at a minimum. For software products with a user interface, this manual should be developed early in the life cycle because it is a necessary mechanism for communicating and stabilizing an important subset of requirements. The user manual should be written by members of the test team, who are more likely to understand the user's perspective than the development team. If the test team is responsible for the manual, it can be generated in parallel with development and can be evolved early as a tangible and relevant perspective of evaluation criteria. It also provides a necessary basis for test plans and test cases, and for construction of automated test suites.

- **I. Architecture overview**
  - A. Objectives
  - B. Constraints
  - C. Freedoms
- **II. Architecture views**
  - A. Design view
  - B. Process view
  - C. Component view
  - D. Deployment view
- **III. Architectural interactions**
  - A. Operational concept under primary scenarios
  - B. Operational concept under secondary scenarios
  - C. Operational concept under anomalous conditions
- **IV. Architecture performance**
- **V. Rationale, trade-offs, and other substantiation**

FIGURE 6-10. *Typical architecture description outline*

PRAGMATIC ARTIFACTS

Conventional document-driven approaches squandered incredible amounts of engineering time on developing, polishing, formatting, reviewing, updating, and distributing documents. Why? There are several reasons that documents became so important to the process. First, there were no rigorous engineering methods or languages for requirements specification or design. Consequently, paper documents with ad hoc text and graphical representations were the default format. Second, conventional languages of implementation and deployment were extremely cryptic and highly unstructured. To present the details of software structure and behavior to other interested reviewers (testers, maintainers, managers), a more human-readable format was needed. Probably most important, software progress needed to be "credibly" assessed. Documents represented a tangible but misleading mechanism for demonstrating progress.

This philosophy raises the following cultural issues:

- **People want to review information but don't understand the language of the artifact.** Many interested reviewers of a particular artifact will resist having to learn the engineering language in which the artifact is written. It

- **People want to review the information but don't have access to the tools.** It is not very common for the development organization to be fully tooled; it

- **Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner.** Properly spelled English words should be used for all identifiers and descriptions.

- **Useful documentation is self-defining: It is documentation that gets used.**

- **Paper is tangible; electronic artifacts are too easy to change.**

MODEL-BASED SOFTWARE ARCHITECTURES

INTRODUCTION:

Software architecture is the central design problem of a complex software system in the same way an *architecture* is the software system design.

 The ultimate goal of the engineering stage is to converge on a stable architecture baseline.
 Architecture is not a paper document. It is a collection of information across all the engineering sets.
 Architectures are described by extracting the essential information from the design models.
 A *model* is a relatively independent abstraction of a system.
 A *view* is a subset of a model that abstracts a specific, relevant perspective.

ARCHITECTURE: A MANAGEMENT PERSPECTIVE

 the most critical and technical product of a software project is its architecture
 If a software development team is to be successful, the inter project communications, as captured in software architecture, must be accurate and precise.

From the management point of view, three different aspects of architecture

1. An *architecture* (the intangible design concept) is the design of software system it includes all engineering necessary to specify a complete bill of materials. Significant make or buy decisions
are resolved, and all custom components are elaborated so that individual component costs and construction/assembly costs can be determined with confidence.

2. An *architecture baseline* (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision (function and quality) can be achieved within the parameters of the business case (cost, profit, time, technology, people).

3. An *architectural description* is an organized subset of information extracted from the design set model's. It explains how the intangible concept is realized in the tangible artifacts.
The number of views and level of detail in each view can vary widely. For example the architecture of the software architecture of a small development tool**.**

There is a close relationship between software architecture and the modern software development process because of the following reasons:

1. A stable software architecture is nothing but a project milestone where critical make/buy decisions should have been resolved. The life-cycle represents a transition from the engineering stage of a project to the production stage.

2. Architecture representation provide a basis for balancing the trade-offs between the problem space (requirements and constraints) and the solution space (the operational product).

3. The architecture and process encapsulate many of the important communications among individuals, teams, organizations, and stakeholders.

4. Poor architecture and immature process are often given as reasons for project failure.

5. In order to proper planning, a mature process, understanding the primary requirements and demonstrable architecture are important fundamentals.

6. Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints; they require human innovation and cannot be automated.

ARCHITECTURE: A TECHNICAL PERSPECTIVE

 Software architecture include the structure of software systems, their behavior, and the patterns that guide these elements, their collaborations, and their composition.

 An architecture framework is defined in terms of views is the abstraction of the UML models in the design set. Whereas architecture view is an abstraction of the design model, include full breadth and depth of information.

Most real-world systems require four types of views:

1) Design: describes architecturally significant structures and functions of the design model.

2) Process: describes concurrency and control thread relationships among the design, component, and deployment views.

3) Component: describes the structure of the implementation set.

4) Deployment: describes the structure of the deployment set.

The design set include all UML design models describing the solution space.
 The design, process, and use case models provide for visualization of the logical and behavioral aspect of the design.
 The component model provides for visualization of the implementation set.
__ The deployment model provides for visualization of the deployment set.

1. The *use case view* describes how the system's critical use cases are realized by elements of the design model. It is modeled statistically by using use case diagrams, and dynamically by using any of the UML behavioral diagrams.

2. The *design view* describes the architecturally significant elements of the design model. It is modeled statistically by using class and object diagrams, and dynamically using any of the UML behavioral diagrams.

3. The *process view* addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including logical software topology, inter process communication, and state mgmt. it is modeled statistically using deployment diagrams, and dynamically using any of the UML behavioral diagrams.

4. The *component view* describes the architecturally significant elements of the implementation set. It is modeled statistically using component diagrams, and dynamically using any of the UML behavioral diagrams.

5. The *deployment view* addresses the executable realization of the system, including the allocation of logical processes in the distributed view to physical resources of the deployment network. It is modeled statistically using deployment diagrams, and dynamically using any of UML behavioral diagrams.

Architecture descriptions take on different forms and styles in different organizations and domains. At any given time, an architecture requires a subset of artifacts in engineering set.

- An architecture baseline is defined as a balanced subset of information across all sets, whereas an architecture description is completely encapsulated within the design set.

Generally architecture base line include:

1) Requirements 2) Design

3) Implementation 4) Deployment

UNIT IV

Workflows of the Process:

In most of the cases a process is a sequence of activities why because of easy to understand, represent, plans and conduct.
But the simplistic activity sequences are not realistic why because it includes many teams, making progress on many artifacts that must be synchronized, cross-checked, homogenized, merged and integrated.
In order to manage complex software's the workflow of the software process is to be changed that is distributed.
Modern software process avoids the life-cycle phases like inception, elaboration, construction and transition. It tells only the state of the project rather than a sequence of activities in each phase.
The activities of the process are organized in to seven major workflows:
1) Management 2) Environment 3) Requirements
4) Design 5) Implementation 6) Assessment
7) Deployment
These activities are performed concurrently, with varying levels of effort and emphasis as a project progresses through the life cycle.
The management workflow is concerned with three disciplines:
1) Planning 2) Project control 3) Organization

The term *workflow* means a thread of cohesive and mostly sequential activities.

Management workflow: controlling the process and ensuring win conditions for all stakeholders.
Environment workflow: automating the process and evolving the maintenance environment.
Requirements workflow: analyzing the problem space and evolving the requirements artifacts.
Design workflow: modeling the solution and evolving the architecture and design artifacts.
Implementation workflow: programming the components and evolving the implementation and deployment artifacts.
Assessment workflow: assessing the trends in process and product quality.
Deployment workflow: transitioning the end products to the user.

Key Principles Of Modern Software Engineering:
1. **Architecture-first approach:** It focuses on implementing and testing the architecture must precede full- scale development and testing of all the components and must precede the downstream focus on completeness and quality of the entire breadth of the product features. Extensive requirements analysis, design, implementation, and assessment activities are performed before the construction phase if we focus on full scale implementation.
2. **Iterative life-cycle process:** From the above figure each phase describes at least two iterations of each workflow. This default is intended to be descriptive, not prescriptive. Some projects may require only one iteration in a phase; other may require several iterations. *The point here is that*

the activities and artifacts of any given workflow may require more than one pass to achieve adequate results.

3. **Round-trip engineering:** Raising the environment activities to a first-class workflow is critical. The environment is the tangible picture of the project's process, methods, and notations for producing the artifacts.

4. **Demonstration-based approach:** Implementation and assessment activities are initiated early in the life cycle, reflecting the emphasis on constructing executable subsets of the evolving architecture.

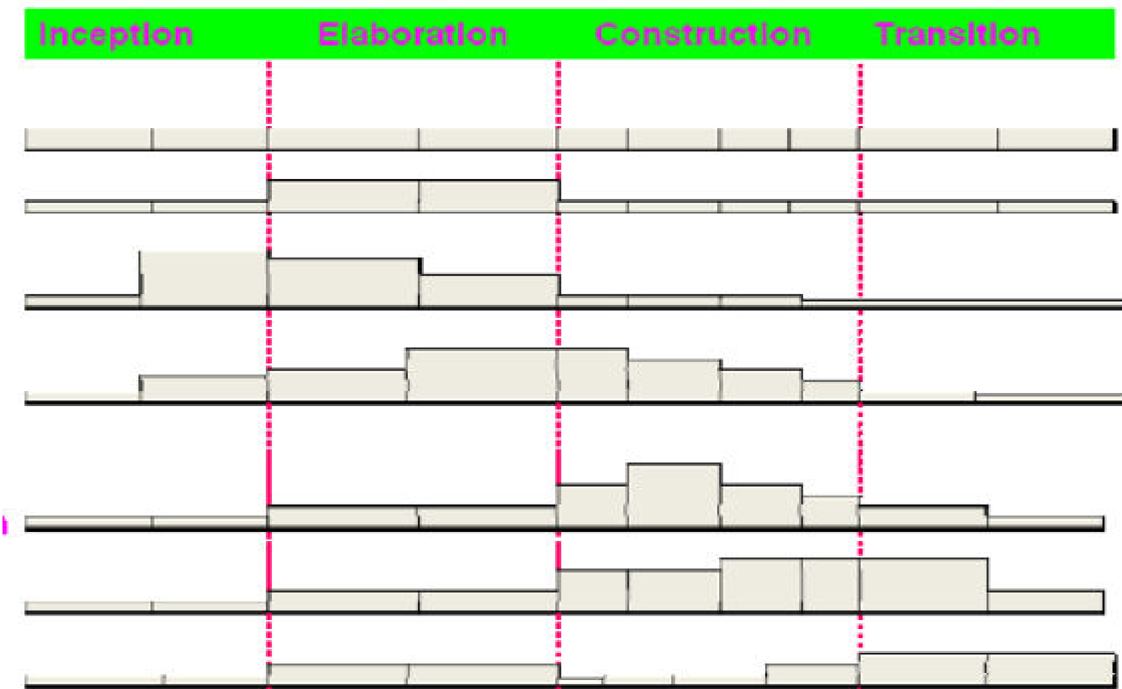


Fig: Activity levels across the life-cycle phases

RKFLOWS:

its of a set of activities in various proportions. An iteration is a set of allocated scenarios. hat are needed in order to implement those scenarios are developed and integrated with the iterations.

Management: Planning to determine the content of the release and develop the detailed plan for the iteration and assigning the tasks, related work to the development team.
Environment: Evolving the software change order database to reflect all new baselines and changes to existing baseline for all product, test, and environment components.
Requirements: Analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be to be demonstrated at the end of this iteration and their evolution criteria. Updating any requirements set artifacts to reflect changes needed by results of this iteration's engine activities.
Design: Evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate at the end of this iteration. Updating design set artifacts to reflect changes needed by the results of this iteration's
engine activities.
Implementation: Developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evolution criteria allocated to this iteration. Integrating and testing all new and modified components with existing baseline (previous versions).
Assessment: Evaluating the results of the iteration; identifying any rework required and determining whether it should be performed before development of this release or allocated to the next release and assessing results to improve the basis of the subsequent iteration's plan.
Deployment: Transitioning the release either to an external organization to internal closure by conducting a post-mortem so that lessons learned can be captured and reflected in the next iteration.
- Iterations in the inception and elaboration phases focus on management, requirement and design activities.
- Iterations in the construction phase focus on design, implementation and assessment activities.
- Iterations in the transition phase focus on assessment and deployment.

Project Organizations and Responsibilities:

 Organizations engaged in software Line-of-Business need to support projects with the infrastructure necessary to use a common process.
 Project organizations need to allocate artifacts & responsibilities across project team to ensure a balance of global (architecture) & local (component) concerns.
 The organization must evolve with the WBS & Life cycle concerns.
 Software lines of business & product teams have different motivation.
 Software lines of business are motivated by return of investment (ROI), new business discriminators, market diversification&profitability.
 Project teams are motivated by the cost, Schedule&quality of specific deliverables

1) Line-Of-Business Organizations:
The main features of default organization are as follows:
• Responsibility for process definition & maintenance is specific to a cohesive line of business.
• Responsibility for process automation is an organizational role & isequal in importance to the process definition role.
• Organizational role may be fulfilled by a single individual or several different teams.

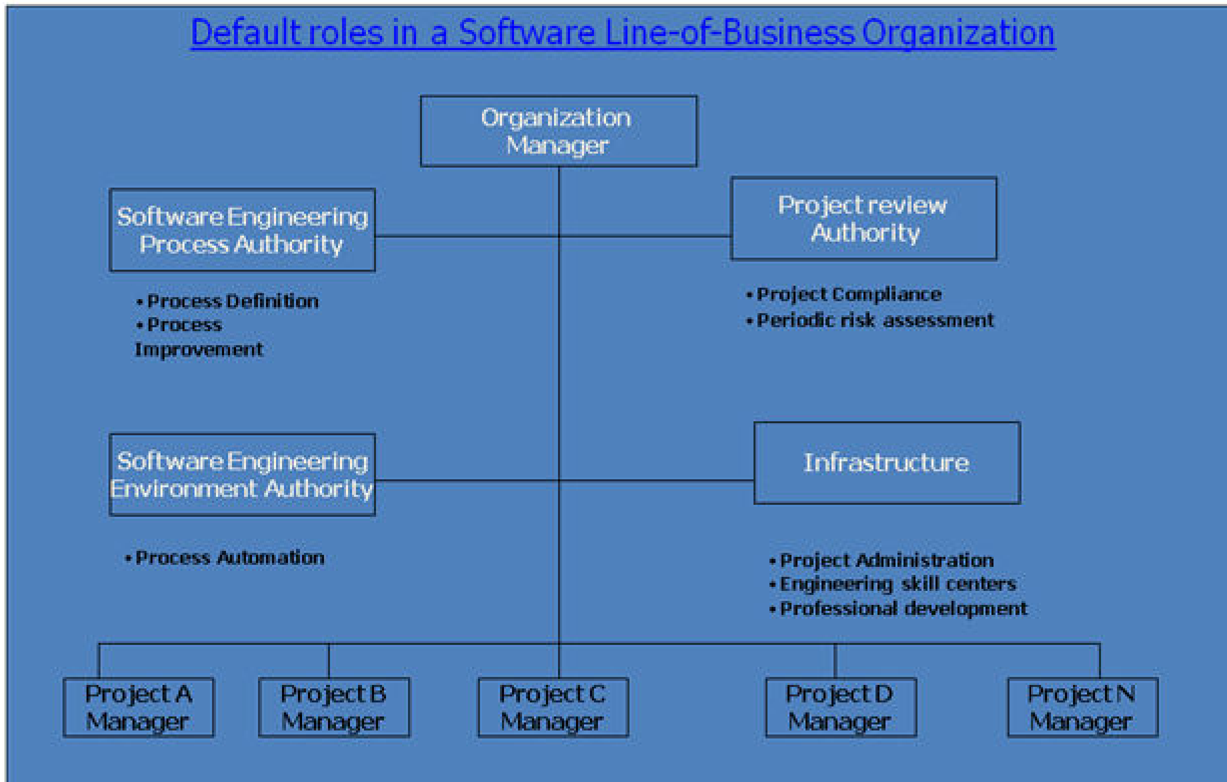


Fig: Default roles in a software Line-of-Business Organization.

Software Engineering Process Authority (SEPA)

The SEPA facilities the exchange of information & process guidance both to & from project practitioners

This role is accountable to General Manager for maintaining a current assessment of the organization's process maturity & its plan for future improvement

Project Review Authority (PRA)

The PRA is the single individual responsible for ensuring that a software project complies with all organizational & business unit software policies, practices & standards

A software Project Manager is responsible for meeting the requirements of a contract or some other project compliance standard

Software Engineering Environment Authority(SEEA)

The SEEA is responsible for automating the organization's process, maintainingthe organization's standard environment, Training projects touse the environment&maintaining organization-wide reusable assets

The SEEA role is necessary to achieve a significant ROI for common process.

Infrastructure

An organization's infrastructure provides human resources support, project-independent research & development, &other capital software engineering assets.

2) Project organizations:

The main features of the default organization are as follows:

- **The project management team** is an active participant, responsible for producing as well as managing.
- **The architecture team** is responsible for real artifacts and for the integration of components, not just for staff functions.
- **The development team** owns the component construction and maintenance activities.
- The assessment team is separate from development.
- Quality is everyone's into all activities and checkpoints.
- Each team takes responsibility for a different quality perspective.

The Process Automation:

The environment must be the first-class artifact of the process.
Process automation& change management is critical to an iterative process. If the change is expensive then the development organization will resist it.
Round-trip engineering& integrated environments promote change freedom & effective evolution of technical artifacts.
Metric automation is crucial to effective project control.
External stakeholders need access to environment resources to improve interaction with the development team & add value to the process.
The three levels of process which requires a certain degree of process automation for the corresponding process to be carried out efficiently.
Metaprocess (Line of business): The automation support for this level is called an infrastructure.
Macroproces (project): The automation support for a project's process is called an environment.
Microprocess (iteration): The automation support for generating artifacts is generally called a tool.

The Project Environment:

The project environment artifacts evolve through three discrete states.
(1)Prototyping Environment.(2)Development Environment.(3)Maintenance Environment.
The **Prototype Environment** includes an architecture test bed for prototyping project architecture to evaluate trade-offs during inception & elaboration phase of the life cycle.
The **Development environment** should include a full suite of development tools needed to support various Process workflows & round-trip engineering to the maximum extent possible.
The Maintenance Environment should typically coincide with the mature version of the development.
There are four important environment disciplines that are critical to management context & the success of a modern iterative development process.
Round-Trip engineering
Change Management
Software Change Orders (SCO)
Configuration baseline Configuration Control Board

Infrastructure
Organization Policy
Organization Environment
Stakeholder Environment.
Round Trip Environment
Tools must be integrated to maintain consistency & traceability.
Round-Trip engineering is the term used to describe this key requirement for environment that support iterative development.
As the software industry moves into maintaining different information sets for the engineering artifacts, more automation support is needed to ensure efficient & error free transition of data from one artifacts to another.
Round-trip engineering is the environment support necessary to maintainConsistency among the engineering artifacts
Change Management
Change management must be automated & enforced to manage multiple iterations & to enable change freedom.
Change is the fundamental primitive of iterative Development.
I. Software Change Orders
The atomic unit of software work that is authorized to create,modify or obsolesce components within a configuration baseline is called a software change orders (SCO)
The basic fields of the SCO are Title, description, metrics, resolution, assessment & disposition
II.Configuration Baseline
A configuration baseline is a named collection of software components &Supporting documentation that is subjected to change management & is upgraded, maintained, tested, statuses & obsolesced a unit.
There are generally two classes of baselines
External Product Release
Internal testing Release
Three levels of baseline releases are required for most Systems
1. Major release (N)
2. Minor Release (M)
3. Interim (temporary) Release (X)
Major release represents a new generation of the product or project
A minor release represents the same basic product but with enhanced features, performance or quality.
Major & Minor releases are intended to be external product releases that are persistent & supported for a period of time.
An interim release corresponds to a developmental configuration that is intended to be transient.
Once software is placed in a controlled baseline all changes are tracked such that a distinction must be made for the cause of the change. Change categories are
Type 0: Critical Failures (must be fixed before release)
Type 1: A bug or defect either does not impair (Harm) the usefulness of the system or can be worked around
Type 2: A change that is an enhancement rather than a response to a defect

Type 3: A change that is necessitated by the update to the environment
Type 4: Changes that are not accommodated by the other categories.

III Configuration Control Board (CCB)
A CCB is a team of people that functions as the decision
Authority on the content of configuration baselines
A CCB includes:
1. Software managers
2. Software Architecture managers
3. Software Development managers
4. Software Assessment managers
5. Other Stakeholderswho are integral to the maintenance of the controlled software delivery system?
Infrastructure
The organization infrastructure provides the organization's capitalassets including two key artifacts - Policy & Environment
I Organization Policy:
A Policy captures the standards for project software developmentprocesses
The organization policy is usually packaged as a handbook that defines the life cycles & the process primitives such as

- Major milestones
- Intermediate Artifacts
- Engineering repositories
- Metrics
- Roles & Responsibilities

II Organization Environment:
The Environment that captures an inventory of tools which are building blocks from which project environments can be configured efficiently & economically
Stakeholder Environment
Many large scale projects include people in external organizations that represent other stakeholders participating in the development process they might include

- Procurement agency contract monitors
- End-user engineering support personnel
- Third party maintenance contractors
- Independent verification & validation contractors
- Representatives of regulatory agencies & others.

These stakeholder representatives also need to access to development resources so that they can contribute value to overall effort. These stakeholders will be access through on-line

- An on-line environment accessible by the external stakeholder sallow them to participate in the process a follows
- **Accept & use** executable increments for the hands-on evaluation.
- **Use** the same on-line tools, data & reports that the development organization uses to manage & monitor the project
- **Avoid** excessive travel, paper interchange delays, format translations, paper * shipping costs & other overhead cost.

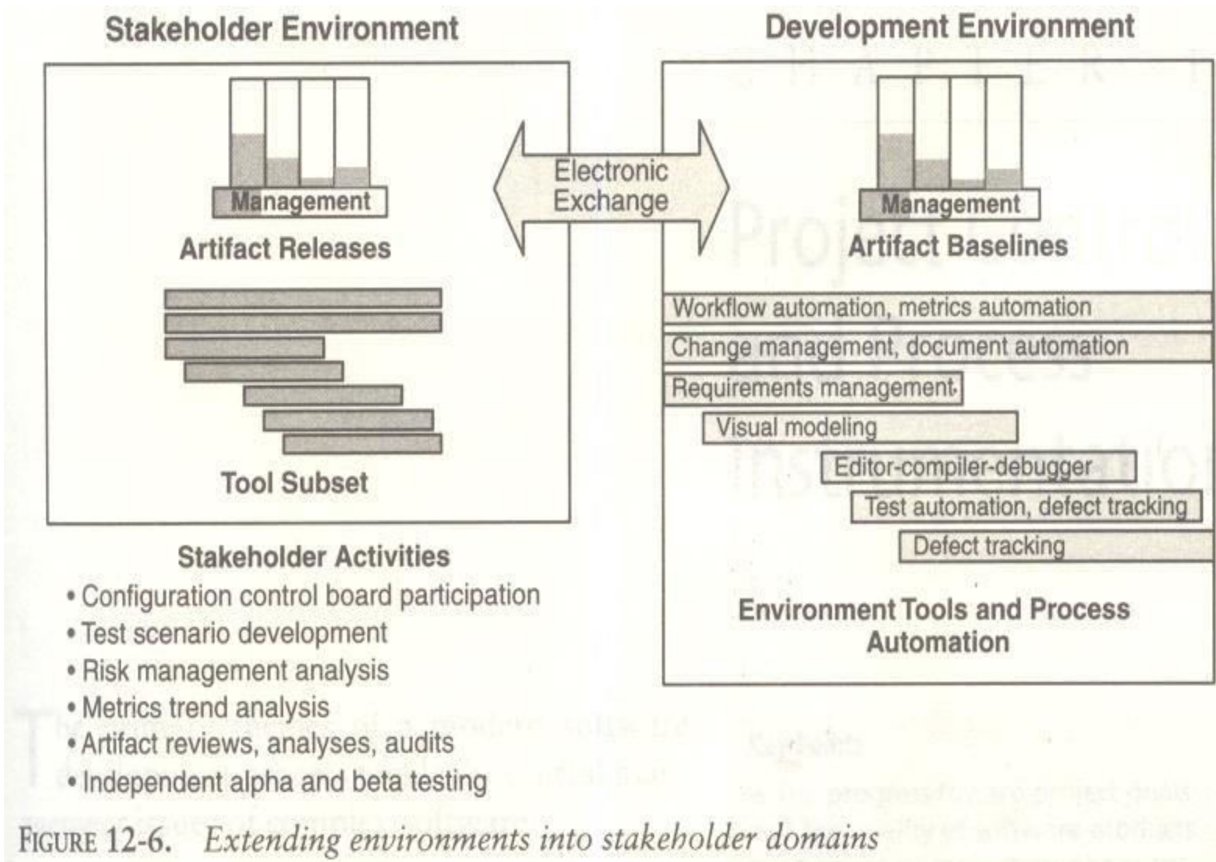


FIGURE 12-6. *Extending environments into stakeholder domains*

Checkpoints of the Process

Introduction:

- It is important to place visible milestones in the life cycle in order to discuss the progress of the project by the stakeholders and also to achieve,

1) Synchronize stakeholder expectations and achieve agreement among the requirements, the design, and the plan perspectives.
2) Synchronize related artifacts into a consistent and balanced state.
3) Identify the important risks, issues, and out-of-tolerance conditions.
4) Perform a global review for the whole life cycle, not just the current situation of an individual perspective or intermediate product.

- Three sequence of project checkpoints are used to synchronize stakeholder expectations throughout the lifecycle:

1) Major milestones 2) Minor milestones 3) Status assessments

- The most important major milestone is usually the event that transitions the project from the elaboration phase into the construction phase.
- The format and content of minor milestones are highly dependent on the project and the organizational culture.
- Periodic status assessments are important for focusing continuous attention on the evolving health of the project and its dynamic priorities.

Three types of joint management reviews are conducted throughout the process:

1) Major milestones: These are the system wide events are held at the end of each development phase. They provide visibility to system wide issues.

2) Minor milestones: These are the iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work.

3) Status assessments: These are periodic events provide management with frequent and regular insight into the progress being made.

- An iteration represents a cycle of activities. Each of the lifecycle phases undergo one or more iterations.

Minor milestones capture two artifacts: a release specification and a release description. Major milestones at the end of each phase use formal, stakeholder-approved evaluation criteria and release descriptions; minor milestones use informal, development-team-controlled versions of these artifacts.

- Most projects should establish all four major milestones. Only in exceptional case you add other major milestones or operate with fewer. For simpler projects, very few or no minor milestones may be necessary to manage intermediate results, and the number of status assessments may be infrequent.

MAJOR MILESTONES:

In an iterative model, the major milestones are used to achieve concurrence among all stakeholders on the current state of the project. Different stakeholders have different concerns:

Customers: schedule and budget estimates, feasibility, risk assessment, requirements understanding, progress, product line compatibility.

Users: consistency with requirements and usage scenarios, potential for accommodating growth, quality attributes.

Architects and systems engineers: product line compatibility, requirements changes, tradeoff analyses, completeness and consistency, balance among risk, quality and usability.

Developers: Sufficiency of requirements detail and usage scenario descriptions, frameworks for component selection or development, resolution of development risk, product line compatibility, sufficiency of the development environment.

Maintainers: sufficiency of product and documentation artifacts, understandability, interoperability. with existing systems, sufficiency of maintenance environment.

Others: regulatory agencies, independent verification and validation contractors, venture capital investors, subcontractors, associate contractors, and sales and marketing teams.

Life-Cycle Objective Milestone: These milestones occur at the end of the inception phase. The goal is to present to all stakeholders a recommendation on how to proceed with development, including a plan, estimated cost and schedule, and expected benefits and cost savings.

Life- Cycle Architecture Milestone: These milestones occur at the end of the elaboration phase. Primary goal is to demonstrate an executable architecture to all stakeholders. A more detailed plan for the construction phase is presented for approval. Critical issues relative to requirements and the operational concept are addressed.

Initial Operational Capability Milestone: These milestones occur late in the construction phase. The goals are to assess the readiness of the software to begin the transition into customer / user sites and to authorize the start of acceptance testing.

Product Release Milestone: Occur at the end of the transition phase. The goal is to assess the completion of the software and its transition to the support organization, if any. The results of acceptance testing are reviewed, and all open issues are addressed and software quality metrics are reviewed to determine whether quality is sufficient for transition to the support organization.

MINOR MILESTONES:

The number of iteration-specific, informal milestones needed depends on the content and length of the iteration.

Iterations which have one-month to six-month duration have only two milestones are needed: the iteration readiness review and iteration assessment review. For longer iterations some other intermediate review points are added.

All iterations are not created equal. Iteration takes different forms and priorities, depending on where the project is in the life cycle.

Early iterations focus on analysis and design. Later iterations focus on completeness, consistency, usability and change management.

Iteration Readiness Review: This informal milestone is conducted at the start of each iteration to review the detailed iteration plan and the evaluation criteria that have been allocated to this iteration.
Iteration Assessment Review: This informal milestone is conducted at the end of each iteration to assess the degree to which the iteration achieved its objectives and to review iteration results, test results, to determine amount of rework to be done, to review impact of the iteration results on the plan for subsequent iterations.

PERIODIC STATUS ASSESSMENTS:

- These are management reviews conducted at regular intervals (monthly, quarterly) to address progress and quality of project and maintain open communication among all stakeholders.
The main objective of these assessments is to synchronize all stakeholders expectations and also serve as project snapshots. Also provide,
1) A mechanism for openly addressing, communicating, and resolving management issues, technical issues, and project risks.
2) A mechanism for broadcast process, progress, quality trends, practices, and experience information to and from all stakeholders in an open forum.
3) Objective data derived directly from on-going activities and evolving product configurations.

Iterative Process Planning:
- Like software development, project planning is also an iterative process.
- Like software, plan is also an intangible one. Plans have an engineering stage, during which the plan is developed, and a production stage, where the plan is executed.
Work breakdown structures:
- Work breakdown structure is the "architecture" of the project plan and also an architecture for financial plan.
- A project is said to be in success, if we maintain good work breakdown structure and its synchronization with the process frame work.
- A WBS is simply a hierarchy of elements that decomposes the project plan into discrete work tasks and it provides:
1) A pictorial description of all significant work.
2) A clear task decomposition for assignment of responsibilities.
3) A framework for scheduling, budgeting, and expenditure tracking.
Conventional *WBS* issues:
Conventional work breakdown structures commonly suffer from three fundamental faults.
1) Conventional WBS is prematurely structured around the product design:
Management
System requirements and design
Subsystem 1
Component 11
{ Requirements, Design, Code, Test, Documentation, . . . }
Component 1N

{ Requirements, Design, Code, Test, Documentation, . . . }
Subsystem M
-do-
Integration and test
{ Test Planning, Test procedure preparation, Testing, Test reports }
Other support areas
{ Configuration control, Quality assurance, System administration }
Fig: Conventional WBS, following the product hierarchy
--
- It is a typical CWBS that has been structured primarily around the subsystem of its product architecture, then further decomposed into the components of each subsystem.
- Once this structure is embedded in the WBS and then allocate to responsible managers with budgets, schedules, and expected deliverables, a concrete planning foundation has been set that is difficult and expensive to change.

2) *CWBS are prematurely decomposed, planned, and budgeted in either too little or too much detail. CWBS are project-specific, and cross-project comparisons are usually difficult or impossible.*

EVOLUTIONARYWBS:
- It organizes the planning elements around the process framework rather than the product framework.
- It better put up the expected changes in the evolving plan.

WBS organizes the hierarchy into three levels:

1) First-level elements: WBS elements are the workflows and are allocated to single team, provides the structure for the purpose of planning and comparison with the other projects.

2) Second-level elements: elements are defined for each phase of the life cycle. These elements allow the faithfulness of the plan to evolve more naturally with the level of understanding of the requirements and architecture, and the risks therein.

3) Third-level elements:
- these elements are defined for the focus of activities that produce the artifacts of each phase.
- These elements may be the lowest level in the hierarchy that collects the cost of discrete artifacts for a given phase, or they may be decomposed further into several lower level activities that, taken together, produce a single artifact.

A Management (1st level)
AA Inception phase management (2nd level)
AAA . . . . (3rd level)
AAB . . . .
. . .
AB Elaboration phase management
ABA . . . .
ABB . . . .
. . .
AC Construction phase management
ACA . . . .

ACB . . . .
AD Transition phase management
ADA . . . .
ABB . . . .
B Environment
-do-
C Requirements
-do-
D Design
-do-
E Implementation
-do-
F Assessment
-do-
G Deployment
-do-

Fig: Default work breakdown structure

- The above structure is a starting point only it need to be tailored to the specifics of a project in many ways:
1) Scale 2) Organizational structure
3) Degree of custom development 4) Business context
5) Precedent experience
- Another important attribute of a good WBS is that the planning fidelity inherent in each element is commensurate with the current life-cycle phase and project state.

PLANNING GUIDELINES:

- Software projects span a broad range of application domains.
- It is valuable but risky to make specific planning suggestions independent of project context.
- Planning provides a skeleton of the project from which the management people can decide the starting point of the project.
- In order to proper plan it is necessary to capture the planning guidelines from most expertise and experience people.
- Project-independent planning advice is also risky. Adopting the planning guidelines blindly without being adapted to specific project circumstances is risk.

Two simple guidelines when a project is initiated or assessed:

1) **A default allocation of costs among the first-level WBS elements.**
   - The above table provides default allocation for budgeted costs of each first-level WBS element.
   - Sometimes these values may vary across projects but this allocation provides a good benchmark for assessing the plan by understanding the foundation for deviations from these guidelines.
   - It is cost allocation table not the effort allocation.

2) Allocation of effort and schedule across the life-cycle phases

TABLE 10-2. *Default distributions of effort and schedule by phase*

DOMAIN	INCEPTION	ELABORATION	CONSTRUCTION	TRANSITION
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

THE COST AND SCHEDULE ESTIMATING PROCESS

Project plans need to be derived from two perspectives:

1) Forward-looking, top-down approach: It starts with an understanding requirements and constraints, derives a macro-level budget and schedule, then decomposes these elements into lower level budgets and intermediate milestones.

From this perspective the following planning sequences would occur:

a) The software project manager develops a characterization of the overall size, process, environment, people, and quality required for the project.

b) A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.

c) The software project manager partitions the estimate for the effort into a top-level WBS using guidelines (table 10-1) and also partitions the schedule into major milestone dates and partition the effort into a staffing profile using guidelines (table 10-2).

d) Subproject managers are given the responsibility for decomposing each of the WBS elements into lower levels using their tip-level allocation, staffing profile, and major milestone dates as constraints.

2) Backward-looking, bottom-up approach: We start with the end in mind, analyze the micro-level budgets and schedules, then sum all these elements into higher level budgets and intermediate milestones. This approach tends to define the WBS from the lowest levels upward. From this perspective, the following planning sequences would occur:

a) The lowest level WBS elements are elaborated into detailed tasks. These estimates tend to incorporate the project-specific parameters in an exaggerated way.

b) Estimates are combined and integrated into higher level budgets and milestones.

c) Comparisons are made with the top-down budgets and schedule milestones. Gross differences are assessed and adjustments are made in order to converge on agreement between the topdown and bottom-up estimates.

- These two planning approaches should be used together, in balance, throughout the life cycle of the project.
- During the engineering stage, the top-down perspective will dominate because there is usually not enough depth of understanding nor stability in the detailed task sequences to perform credible bottomup planning.
- During the production stage, there should be enough precedent experience and planning fidelity that the bottom-up planning perspective will dominate.
- By then, the top-down approach should be well tuned to the project specific parameters, so it should be used more as a global assessment technique.

THE ITERATION PLANNING PROCESS:

So far, this discussion has dealt only with the application-independent aspects of budgeting and scheduling. Another dimension of planning is concerned with defining the actual sequence of intermediate results. Planning the content and schedule of the major milestones and their intermediate iterations is probably the most tangible form of the overall risk management plan. An evolutionary build plan is important because there are always adjustments in build content and schedule as early conjecture evolves into well-understood project circumstances.

PRAGMATIC PLANNING:

Even though good planning is more dynamic in an iterative process, doing it accurately is far easier. While executing iteration N of any phase, the software project manager must be monitoring and controlling against a plan that was initiated in iteration N – 1 and must be planning iteration N + 1. The art of good project management is to make trade-offs in the current iteration plan and the next iteration plan based on objective results in the current iteration and previous iterations. This concept seems, and is, overwhelming in early phases or in projects that are pioneering iterative development. But if the planning pump is primed successfully, the process becomes surprisingly easy as the project progresses into the phases in which high-fidelity planning is necessary for success.

UNIT 5
PROJECT CONTROL & PROCESS INSTRUMENTATION

Software Metrics:

Software metrics are used to implement the activities and products of the software development process. Hence, the quality y of the software products and the achievements in the development process can be determined using the software metrics.

Need for Software Metrics:

1. Software metrics are needed for calculating the cost and schedule of a software product with great accuracy.
2. Software metrics are required for making an accurate estimation of the progress.
3. The metrics are also required for understanding the quality of the software product.

Indicators:

An indicator is a metric or a group of metrics that provides an understanding of the software process or software product or a software project. A soft ware engineer assembles measures and produce metrics from which the indicators can be derived. Two types of indicators are:

(i) Management indicators.

(ii) Quality indicators.

Management Indicators The management indicators i.e., technical progress, financial status and staffing progress are used to determine whether a project is on budget and on schedule. The management indicators that indicate financial status are based on earned value system.

Quality Indicators The quality indicators are based on the measurement of the changes occurred in software.

Seven Core Metrics of Software Project

Software metrics instrument the activities and products of the software development/ integration process. Metrics values provide an important perspective for managing the process. The most useful metrics are extracted directly from the evolving artifacts. There are seven core metrics that are used in managing a modern process.

Seven core metrics related to project control:

Management Indicators

- Work and Progress
- Budgeted cost and expenditures
- Staffing and team dynamics

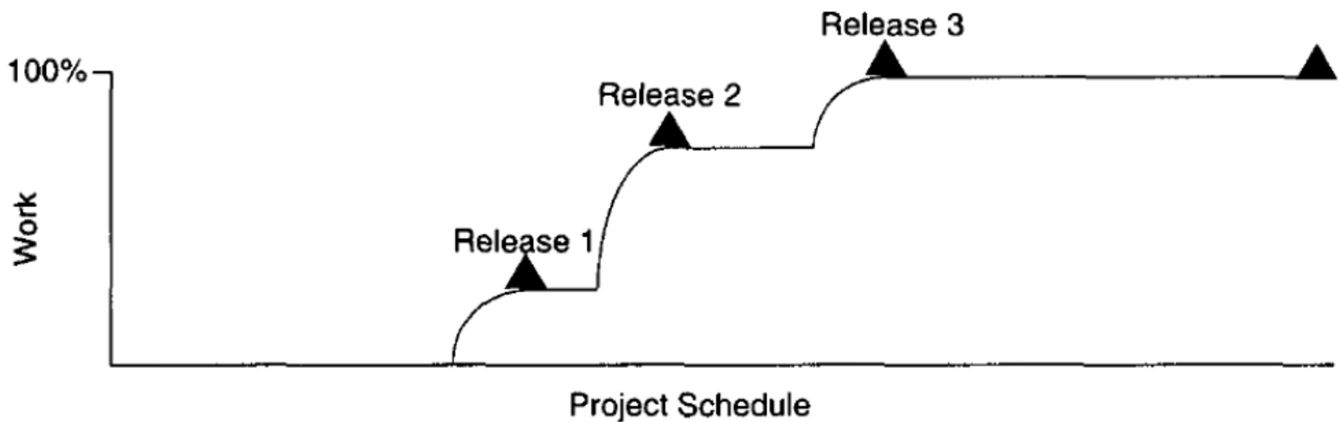
Quality Indicators

- Change traffic and stability
- Breakage and modularity
- Rework and adaptability
- Mean time between failures (MTBF) and maturity

1. Work and progress:

This metric measure the work performed over time. Work is the effort to be accomplished to complete a certain set of tasks. The various activities of an iterative development project can be measured by defining a planned estimate of the work in an objective measure, then tracking progress (work completed overtime) against that plan.

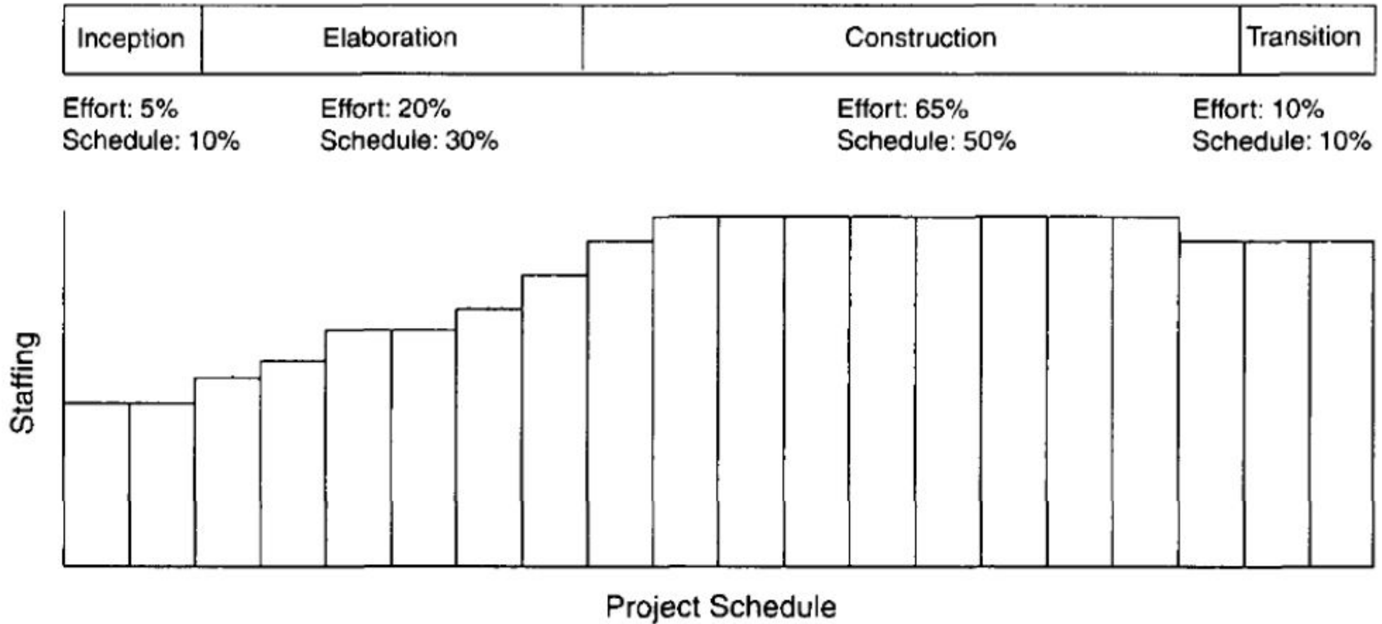
The default perspectives of this metric are:
Software architecture team: - Use cases demonstrated.
Software development team: - SLOC under baseline change management, SCOs closed
Software assessment team: - SCOs opened, test hours executed and evaluation criteria meet.
Software management team: - milestones completed.
The below figure shows expected progress for a typical project with three major releases



2. Budgeted cost and expenditures:
This metric measures cost incurred over time. Budgeted cost is the planned expenditure profile over the life cycle of the project. To maintain management control, measuring cost expenditures over the project life cycle is always necessary. Tracking financial progress takes on an organization - specific format. Financial performance can be measured by the use of an earned value system, which provides highly detailed cost and schedule insight. The basic parameters of an earned value system, expressed in units of dollars, are as follows:
Expenditure Plan - It is the planned spending profile for a project over its planned schedule.
Actual progress - It is the technical accomplishment relative to the planned progress underlying the spending profile.
Actual cost - It is the actual spending profile for a project over its actual schedule.
Earned value - It is the value that represents the planned cost of the actual progress.
Cost variance - It is the difference between the actual cost and the earned value.
Schedule variance - It is the difference between the planned cost and the earned value.
Of all parameters in an earned value system, actual progress is the most subjective assessment. Because most managers know exactly how much cost they have incurred and how much schedule they have used, the variability in making accurate assessments is centered in the actual progress assessment. The default perspectives of this metric are cost per month, full-time staff per month and percentage of budget expended.
3. Staffing and team dynamics:
This metric measure the personnel changes over time, which involves staffing additions and reductions over time. An iterative development should start with a small team until the risks in the requirements and architecture have been suitably resolved. Depending on the overlap of iterations and other project specific circumstances, staffing can vary. Increase in staff can slow overall project progress as new people consume the productive team of existing people in coming up to speed. Low attrition of good people is a sign of

success. **The default perspectives of this metric are people per month added and people per month leaving.** These three management indicators are responsible for technical progress, financial status and staffing progress.

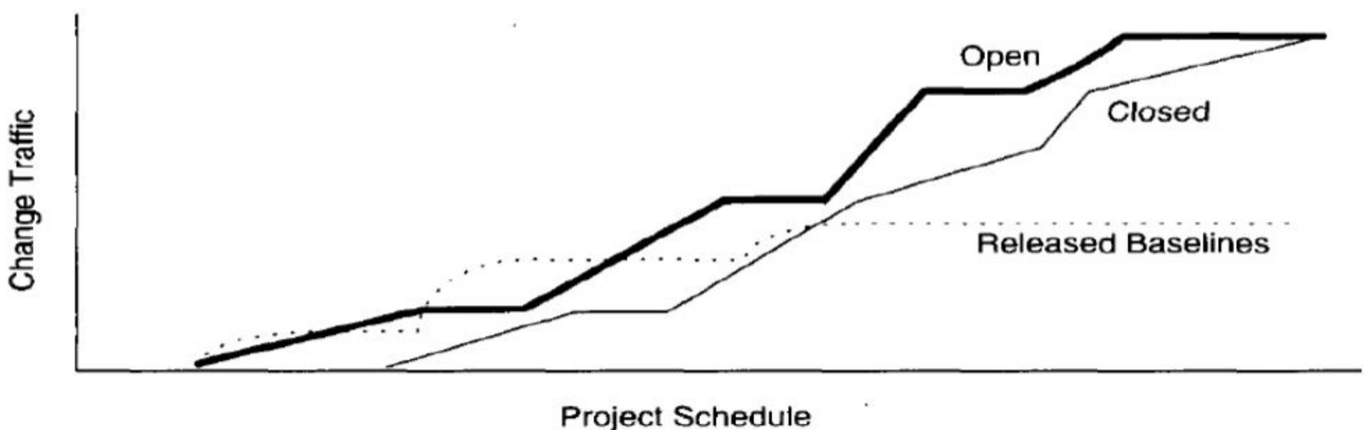
The below figure shows Typical staffing profile



4. Change traffic and stability:

This metric measures the change traffic over time. The number of software change orders opened and closed over the life cycle is called change traffic. Stability specifies the relationship between opened versus closed software change orders. This metric can be collected by change type, by release, across all releases, by term, by components, by subsystems, etc.

The below figure shows stability expectation over a healthy project's life cycle



5. Breakage and modularity:

This metric measures the average breakage per change over time. Breakage is defined as the average extent of change, which is the amount of software baseline that needs rework and measured in source lines of code, function points, components, subsystems, files or other units. Modularity is the average breakage trend over time. This metric can be collected by reworked SLOC per change, by change type, by release, by components and by subsystems.

6. Rework and adaptability:
This metric measures the average rework per change over time. Rework is defined as the average cost of change which is the effort to analyze, resolve and retest all changes to software baselines. Adaptability is defined as the rework trend over time. This metric provides insight into rework measurement. All changes are not created equal. Some changes can be made in a staff hour, while others take staff-weeks. This metric can be collected by average hours per change, by change type, by release, by components and by subsystems.
7. MTBF and Maturity:
This metric measures defect rater over time. MTBF (Mean Time Between Failures) is the average usage time between software faults. It is computed by dividing the test hours by the number of type 0 and type 1 SCOs. Maturity is defined as the MTBF trend over time. Software errors can be categorized into two types deterministic and nondeterministic.
Deterministic errors are also known as Bohr-bugs and nondeterministic errors are also called as Heisen-bugs. Bohr-bugs are a class of errors caused when the software is stimulated in a certain way such as coding errors. Heisen-bugs are software faults that are coincidental with a certain probabilistic occurrence of a given situation, such as design errors. This metric can be collected by failure counts, test hours until failure, by release, by components and by subsystems. These four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data.

METRICS AUTOMATION

Many opportunities are available to automate the project control activities of a software project. A **S**oftware **P**roject **C**ontrol **P**anel (**SPCP**) is essential for managing against a plan. This panel integrates data from multiple sources to show the current status of some aspect of the project. The panel can support standard features and provide extensive capability for detailed situation analysis. SPCP is one example of metrics automation approach that collects, organizes and reports values and trends extracted directly from the evolving engineering artifacts.
SPCP:
To implement a complete SPCP, the following are necessary.

- Metrics primitives - trends, comparisons and progressions
- A graphical user interface.
- Metrics collection agents
- Metrics data management server
- Metrics definitions - actual metrics presentations for requirements progress, implementation progress, assessment progress, design progress and other progress dimensions.
- Actors - monitor and administrator.

Monitor defines panel layouts, graphical objects and linkages to project data. Specific monitors called roles include software project managers, software development team leads, software architects and customers. Administrator installs the system, defines new mechanisms, graphical objects and linkages.
The whole display is called a panel. Within a panel are graphical objects, which are types of layouts such as dials and bar charts for information. Each graphical object displays a metric. A panel contains a number of

graphical objects positioned in a particular geometric layout. A metric shown in a graphical object is labeled with the metric type, summary level and insurance name (line of code, subsystem, server1). Metrics can be displayed in two modes – value, referring to a given point in time and graph referring to multiple and consecutive points in time. Metrics can be displayed with or without control values. A control value is an existing expectation either absolute or relative that is used for comparison with a dynamically changing metric. Thresholds are examples of control values.

PROCESS DISCRIMINANTS *Or* TRANSITIONING TO AN ITERATIVE PROCESS *Or TAILORING THE PROCESS*

In tailoring the management process to a specific domain or project, there are two dimensions of discriminating factors: technical complexity and management complexity. A process framework is not a project-specific process implementation with a well-defined recipe for success. The process framework must be configured to the specific characteristics of the project. The process discriminants are organized around six process parameters - scale, stakeholder cohesion, process flexibility, process maturity, architectural risk and domain experience.

1. Scale: the scale of the project is the team size which drives the process configuration more than any other factor. There are many ways to measure scale, including number of sources lines of code, number of function points, number of use cases and number of dollars. The primary measure of scale is the size of the team. Five people are an optimal size for an engineering team. A team consisting of 1 member is said to be trivial, a team of 5 is said to be small, a team of 25 is said to be moderate, a team of 125 is said to be large, a team of 625 is said to be huge and so on. As team size grows, a new level of personnel management is introduced at each factor of 5.10 Trivial - sized projects require almost no management overhead. Only little documentation is required. Workflow is single-threaded. Performance is dependent on personnel skills. Small projects consisting of 5 people require very little management overhead. Project milestones are easily planned, informally conducted and changed. There are a small number of individual workflows. Performance depends primarily on personnel skills. Process maturity is unimportant. Moderate-sized projects consisting of 25 people require very moderate management overhead. Project milestones are formally planned and conducted. There are a small number of concurrent team workflows, each team consisting of multiple individual workflows. Performance is highly dependent on the skills of key personnel. Process maturity is valuable. Large projects consisting of 125 people require substantial management overhead.

Project milestones are formally planned and conducted. A large number of concurrent team workflows are necessary, each with multiple individual workflows. Performance is highly dependent on the skills of the key personnel. Process maturity is necessary. Huge projects consisting of 625 people require substantial management overhead. Project milestones are very formally planned and conducted. There are a very large number of concurrent team workflows, each with multiple individual workflows. Performance is highly dependent on the skills of the key personnel. Project performance is still dependent on average people.

2. Stakeholder Cohesion or Contention: The degree of cooperation and coordination among stakeholders (buyers, developers, users, subcontractors and maintainers) significantly drives the specifics of how aprocess is defined. This process parameter ranges from cohesive to adversarial. Cohesive teams have

common goals, complementary skills and close communications. Adversarial teams have conflicting goals, competing and incomplete skills, and less-than-open communication.

3. Process Flexibility or Rigor: The implementation of the project's process depends on the degree of rigor, formality and change freedom evolved from projects contract (vision document, business case and development plan). For very loose contracts such as building a commercial product within a business unit of a software company, management complexity is minimal. For a very rigorous contract, it could take many months to authorize a change in a release schedule.

4. Process Maturity: The process maturity level of the development organization is the key driver of management complexity. Managing a mature process is very simpler than managing an immature process. Organization with a mature process have a high level of precedent experience in developing software and a high level of existing process collateral that enables predictable planning and execution of the process. This sort of collateral includes well-defined methods, process automation tools, trained personnel, planning metrics, artifact templates and workflow templates.

5. Architectural Risk: The degree of technical feasibility is an important dimension of defining a specific projects process. There are many sources of architecture risk. They are (1) system performance which includes resource utilization, response time, throughout and accuracy, (2) robustness to change which includes addition of new features & incorporation of new technology and (3) system reliability which includes predictable behavior and fault tolerance.

6. Domain Experience: The development organization's domain experience governs its ability to converge on an acceptable architecture in a minimum no of iterations.

NEXT GENERATION SOFTWARE ECONOMICS

MODERN PROJECT PROFILES

Continuous Integration

In the iterative development process, firstly, the overall architecture of the project is created and then all the integration steps are evaluated to identify and eliminate the design errors. This approach eliminates problems such as downstream integration, late patches and shoe-horned software fixes by implementing sequential or continuous integration rather than implementing large-scale integration during the project completion.

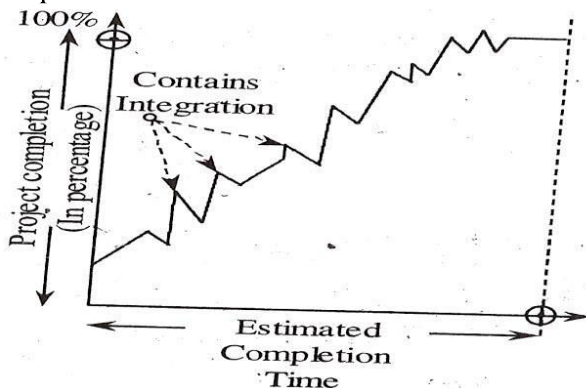


Figure (a): Modern Project Profile

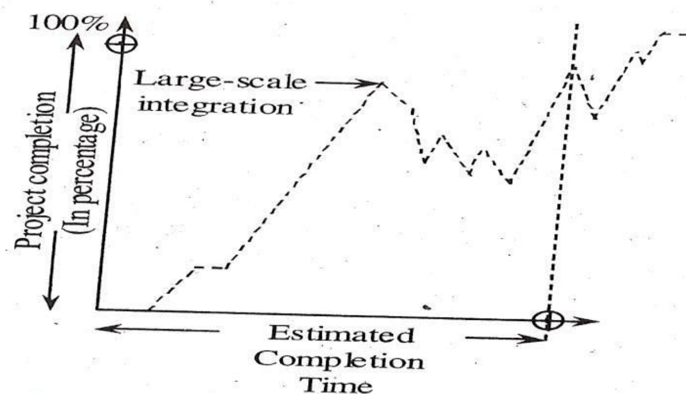


Figure (b): Traditional Project Profile

- Moreover, it produces feasible and a manageable design by delaying the 'design breakage' to the engineering phase, where they can be efficiently resolved. This can be one by making use of project demonstrations which forces integration into the design phase.
- With the help of this continuous integration incorporated in the iterative development process, the quality tradeoffs are better understood and the system features such as system performance, fault tolerance and maintainability are clearly visible even before the completion of the project.

1.2 Early Risk Resolution

In the project development lifecycle, the engineering phase concentrates on identification and elimination of the risks associated with the resource commitments just before the production stage. The traditional projects involve, the solving of the simpler steps first and then goes to the complicated steps, as a result the progress will be visibly good, whereas, the modern projects focuses on 20% of the significant requirements, use cases, components and risk and hence they occasionally have simpler steps.

- To obtain a useful perspective of risk management, the project life cycle has to be applied on the principles of software management. The following are the 80:20 principles.
- The 80% of Engineering is utilized by 20% of the requirements.
- Before selecting any of the resources, try to completely understand all the requirement because irrelevant resource selection (i.e., resources selected based on prediction) may yield severe problems.
- 80% of the software cost is utilized by 20% of the components
- Firstly, the cost-critical components must be elaborated which forces the project to focus more on controlling the cost.
- 80% of the bugs occur because of 20% of the components
- Firstly, the reliability-critical components must be elaborated which give sufficient time for assessment activities like integration and testing, in order to achieve the desired level of maturity.
- 80% of the software scrap and rework is due to 20% if the changes
- The change-critical components r elaborated first so that the changes that have more impact occur when the project is matured.
- 80% of the resource consumption is due to 20% of the components.

- Performance critical components are elaborated first so that, the trade-offs with reliability; changeability and cost-consumption can be solved as early as possible.

- 80% of the project progress is carried-out by 20% of the people

- It is important that planning and designing team should consist of best processionals because the entire success of the project depends upon a good plan and architecture.

The following figure shows the risk management profile of a modern project.

1.3 Evolutionary requirements

 The traditional methods divide the system requirements into subsystem requirements which in turn gets divided into component requirements. These component requirements are further divided into unit requirements. The reason for this systematic division is to simplify the traceability of the requirements.

 In the project life cycle the requirements and design are given the first and the second preference respectively. The third preference is given to the traceability between the requirement and the design components these preferences are given in order to make the design structure evolve into an organization so it parallels the structure of the requirements organization.

 Modern architecture finds it difficult to trace the requirements because of the following reasons.

- Usage of Commercial components
- Usage of legacy components
- Usage of distributed resources
- Usage of object oriented methods.

Principles of Software Management

1. Process must be based on architecture-first approach

If the architecture is focused at the initial stage, then there will be a good foundation for almost 20% of the significant stuff that are responsible for the overall success of the project. This stuff include the requirements, components use cases, risks and errors. In other words, if the components that are being involved in the architecture are well known then the expenditure causes by scrap and rework will be comparatively less.

2. Develop an iterative life-cycle process that identifies the risks at an early stage

An iterative process supports a dynamic planning framework that facilitates the risk management predictable performance moreover, if the risks are resolved earlier, the predictability will be more and the scrap and rework expenses will be reduced.

3. After the design methods in-order to highlight components-based development.

The quantity of the human generated source code and the customized development can be reduced by concentrating on individual components rather than individual lines-of-code. The complexity of software is directly proportional to the number of artifacts it contains that is, if the solution is smaller then the complexity associated with its management is less.

4. Create a change management Environment

Highly-controlled baselines are needed to compensate the changes caused by various teams that concurrently work on the shared artifacts.

5. Improve change freedom with the help of automated tools that support round-trip engineering.

The roundtrip-engineering is an environment that enables the automation and synchronization of engineering information into various formats. The engineering information usually consists requirement specification, source code, design models test cases and executable code. The automation of this information allows the teams to focus more on engineering rather than dealing with over head involved.

Design artifacts must be captured in model based notation.

The design artifacts that are modeled using a model based notation like UML, are rich in graphics and texture.

7. Process must be implemented or obtaining objective quality control and estimation of progress.

The progress in the lifecycle as well as the quality of intermediately products must be estimated and incorporated into the process. This can be done with the help of well defined estimation mechanism that are directly derived from the emerging artifacts. These mechanisms provide detailed information about trends and correlation with requirements.

8. Implement a Demonstration-based Approach for Estimation of intermediately Artifacts

This approach involves giving demonstration on different scenarios. It facilitates earl integration and better understanding of design trade-offs. Moreover, it eliminates architectural defects earlier in the lifecycle. The intermediately results of this approach are definitive.

The Points Increments and generations must be made based on the evolving levels of detail Here, the 'levels of detail' refers to the level of understanding requirements and architecture. The requirements, iteration content, implementations and acceptance testing can be organized using cohesive usage scenarios.

10. Develop a configuration process that should be economically scalable

The process framework applied must be suitable for variety of applications. The process must make use of processing spirit, automation, architectural patterns and components such that it is economical and yield investment benefits.

Best Practices Associated with software Management

 According to airline software council, there are about nine best practices associated with software management. Theses practices are implemented in order to reduce the complexity of the larger projects and to improve software management discipline.

The following are the best practices of software management:

1. Formal Risk Management: Earlier risk management can be done by making use of iterative life cycle process that identifies the risks at early stage.

2. Interface Settlement: The interface settlement is one of the important aspects of architecture first approach because; obtaining architecture involves the selection of various internal and external interfaces that are incorporated into the architecture.

3. Formal Inspections: There are various defect removal strategies available. Formal inspection is one of those strategies. However this is the least important strategy because the cost associated with human recourses is more and is defect detection rate for the critical architecture defects is less.

4.Management and scheduling based on metrics: This principle is related to the model based approach and objective quality control principles. It states to use common notations fro the artifacts so that quality and progress can be easily measured.
5. Binary quality Gates at the inch-pebble level: The concept behind this practice is quite confusing. Most of the organizations have misunderstood the concept and have developed an expensive and a detailed plan during the initial phase of the lifecycle, but later found the necessity to change most of their detailed plan due to the small changes in requirements or architectural. This principle states that first start planning with an understanding of requirements and the architecture. Milestones must be established during engineering stage and inch-pebble must be followed in the production stage.
6. Plan versus visibility of progress throughout the progress: This practice involves a direct communication between different team members of a project so that, they can discuss the significant issues related to the project as well as notice the progress of the project in-comparison to their estimated progress
7. Identifying defects associated with the desired quality: This practice is similar to the architecture-first approach and objective quality control principles of software management. It involves elimination of architectural defects early in the life-cycle, thereby maintaining the architectural quality so as to successfully complete the project.
8. Configuration management: According to Airline software council, configuration management serves as a crucial element for controlling the complexity of the artifacts and for tracing the changes that occur in the artifacts. This practice is similar to the change management principle of software management and prefers automation of components so as to reduce the probability of errors that occur in the large-scale projects.
9.Disclose management accountability: The entire managerial process is disclosed to al the people dealing with the project.

NEXT GENERATION SOFTWARE ECONOMICS

Next generation software cost models

 In comparison to the current generation software cost modes, the next generation software cost models should perform the architecture engineering and application production separately. The cost associated with designing, building, testing and maintaining the architecture is defined in terms of scale, quality, process, technology and the team employed.

 After obtaining the stable architecture, the cost of the production is an exponential function of size, quality and complexity involved.

 The architecture stage cost model should reflect certain diseconomy of scale (exponent less than 1.0) because it is based on research and development-oriented concerns. Whereas the production stage cost model should reflect economy of scale (exponent less than 1.0) for production of commodities.

 The next generation software cost models should be designed in a way that, they can assess larger architectures with economy of scale. Thus, the process exponent will be less than 1.0 at the time of production because large systems have more automated proves components and architectures which are easily reusable.

- The next generation cost model developed on the basis of architecture-first approach is shown below.
- At architectural engineering Stage
- A Plan with less fidelity and risk resolution
- It is technology or schedule-based
- It has contracts with risk sharing
- Team size is small but with experienced professionals
- The architecture team, consists of small number of software engineers
- The application team consists of small number of domain engineers.
- The output will be an executable architecture, production and requirements
- The focus of the architectural engineering will be on design and integration of entities as well as host development environment.
- It contains two phases they are inspection and elaboration

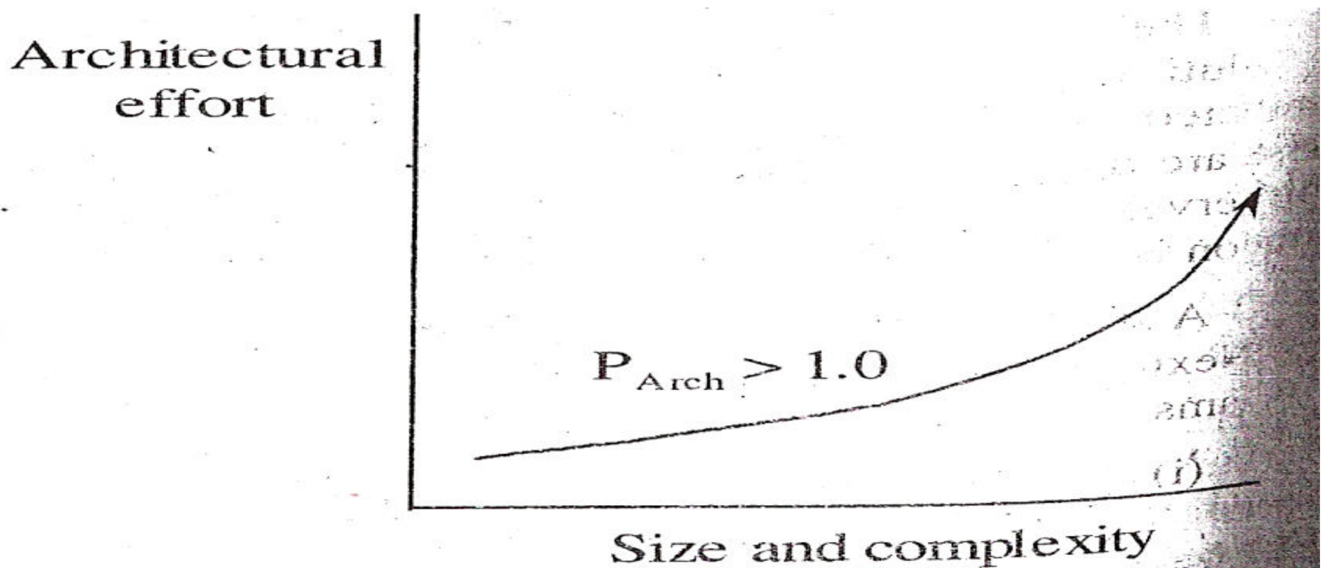


Figure: N-Month Design Phase

- At Application production stage
- A plan with high fidelity and lower risk

- It is cost-based
- It has fixed-priced contracts
- Team size is large and diverse as needed.
- Architecture team consists of a small number of software engineers.
- The Application team may have nay number of domain engineers.
- The output will be a function which is deliverable and useful, tested baseline and warranted quality.

- The focus of the application production will be on implementing testing and maintaining target technology.

It contains two phases they are construction and transition.

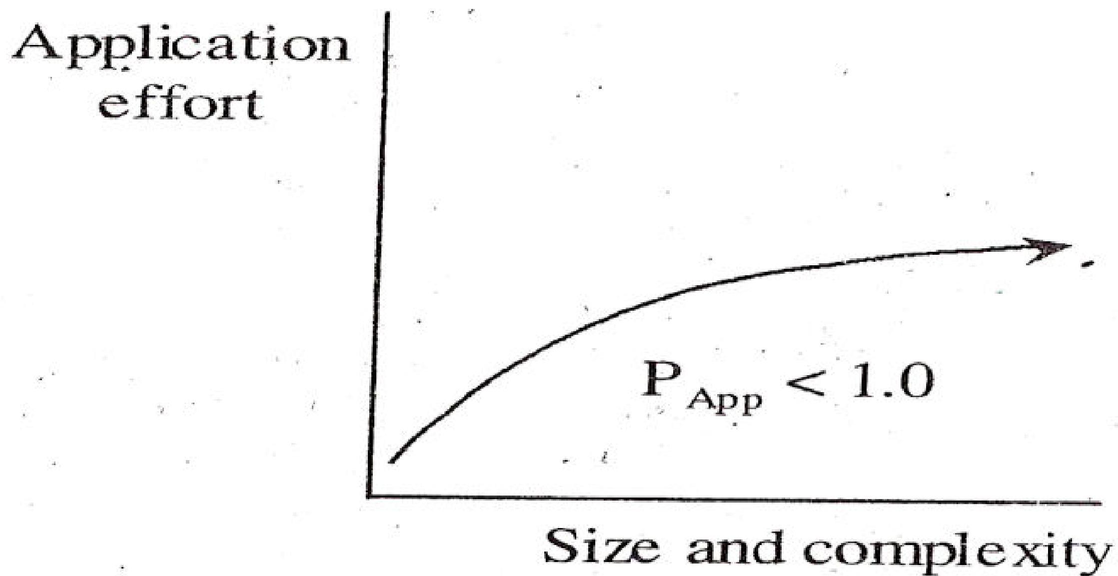


Figure: M-Month Production Increments

Total Effort = Func(TechnologyArch, ScaleArch, Quality Arch, Process Arch) + Func(TechnologyApp, ScaleApp, Quality App, Process App)
Total Time = Func(ProcessArch, EffortArch) + Func(ProcessApp, EffortApp,)

- The next generation infrastructure and environment automated various management activities with low effort. It relieves many of the sources of diseconomy of scale by reusing the common processes that are repetitive in a particular project. It also reuses the common outcomes of the project. The prior experience and matured processes utilized

in these types of models eliminate the scrap rework sources. Here, the economics of scale will be affected.

- The architecture and applications of next generation cost models have difference scales and sized which represents the solution space. The size can be computed inters of SLOC or megabytes of executable code while the scale can be computed in 0-terms of components, classes, processes or nodes. The requirement or use cases of solution space are different from that of a problem space. Moreover, there can be more than one solution to a problem. Where cost serves as a key discriminator. The cost estimates must be determined to find an optimal solution. If an optional solution is not found then different solution s need to be selected or to change the problem statement.
- A strict notation must be applied for design artifacts so, that the prediction of a design scale can be improved. The Next-generation software cost model should automate the process of measuring design scale directly from UML diagrams. There should be two major improvements. There are,
- Separate architectural engineering stage from application production stage. This will yield greater accuracy and more precision of lifecycle estimate.
- The use of rigorous design notations. This will enable the automation and standardization of scale measure so that they can be easily traced which helps to determine the total cost associated with production.

The next generation software process has two potential breakthroughs, they are,

- Certain integrated tools would be available that automates the information transition between the requirements, design, implementation and deployment elements. These tools facilitate roundtrip engineering between various artifacts of engineering.
- It will reduce the four sets of fundamental technical artifacts into three sets. This is achieved by automating the activities related to human-generated source code so as to eliminate the need for a separate implementation set**.**

2.2 'An organizational manager should strive for making the transition to a modern process'.

The transition to a modern process should be made based on the following quotations laid by Boehm.

1. Identifying and solving a software problem in the design phase is almost 100 times cost effective than solving the same problem after delivery.

This quotation or metric serves as a base for most software processes. Modern processes, component-based development techniques and architectural frameworks mainly focus on enhancing this relationship. The architectural errors are solved by implementing an architecture-first approach. Modern process plays a crucial role in identification of risks

2. Software Development schedules can be compressed to a Maximum of 25 percent

If we want a reduction in the scheduled time, then we must increase the personnel resources which inturn increases the management overhead. The management overhead, concurrent activities scheduling, sequential activities conservation along some resource constraints will have the flexibility limit of about 25 percent.

This metric must be acceptable by the engineering phase which consists of detailed system content if we have successfully completed the engineering then compression in the production stage will be automatically flexible. The concurrent development must be possible irrespective of whether a business organization

implements the engineering phase over multiple projects or whether a project implements the engineering phase over multiple incremental stages.

3. The maintenance cost will be almost double the development cost

Most o the experts in the software industry find it difficult to maintain the software than development. The ratio between development and maintenance can be measured by computing productivity cost. One of the interesting fact of iterative development is that the dividing line between the development and maintenance is vanishing. Moreover, a good iterative process and architecture will cause the reduction in the scrap and rework levels so this ratio (i.e.,) 2:1 can be reduced to 1:1.

4. Both the software development cot and the maintenance cost are dependent on the number of lines in the source code.

This metric was applicable to the conventional cost models which were lacking in-terms of commercial components, reusing techniques, automated code generators etc. The implementation of commercial components, reusing techniques and automated code generators will make this metric inappropriate. However, the development cost is still dependent on the commercial components, reuse technique and automatic code generators and their integration.

The next-generation cost models should focus more on the number of components and their integration efforts rather than on the number of lines of code.

5. Software productivity mainly relies on the type of people employed

The personal skills, team work ability and the motivation of employees are the crucial factors responsible for the success and the failure of any project. The next-generation cost models failure should concentrate more on employing a highly skilled team of professionals at engineering stage.

6. The ratio of software to hardware cost is increasing.

As the computers are becoming more and more popular, the need for software and hardware applications is also increasing. The hardware components are becoming cheaper whereas, the software applications are becoming more complicated as a result, highly skilled professionals needed for development and controlling the software applications, the in turn increases the cost. In 1955 the software to hardware cost ratio was 15:85 and in 1985 this ratio was 85:15. This ratio continuously increases with respect to the need for variety of software applications. Certain software applications have already been developed which provides automated configuration control and analysis of quality assurance. The next-generation cost models must focus on automation of production and testing.

7. Only 15% of the overall software development is dedicated process to programming.

The automation and reusability of codes have lead to the reduction in programming effort. Earlier in 1960s, the programming staff was producing about 200 machine instructions per month and in 1970s and 1980s, the machine instruction count has raised to about 1000 machine instructions. Now as days, programmers are able to produce several thousand instructions without even writing few hundreds of them.

8. Software system and products cost three times the cost associated with individual software programs per SLOC software-system products cost 9 times more than the cost of individual software program.

In the software development, the cost of each instruction depends upon the complexity of the software. Modern processes and technologies must reduce this diseconomy of scale. The economy of the scale must be achievable under the customer specific software systems with a common architecture, common environment and common process.

9. 60% of Errors are caught by walkthrough

The walkthrough and other forms of human inspection catch only the surface and style issues. However, the critical issues are not caught by the walkthroughs so, this metric doesn't prove to the reliable.

10. Only 20% of the contributors are responsible for the 80% of the contributions.

This metric is applicable to most of the engineering concepts such as 80:20 principles of software project management. The next generation software process must facilitate the software organizations in achieving economic scale.

MODERN PROCESS TRANSITIONS

Indications of a successful project transition to a modern culture

Several indicators are available that can be observed in order to distinguish projects that have made a genuine cultural transition from projects that only pretends. The following are some rough indicators available.

1. The lower-level managers and the middle level managers should participate in the project development

Any organization which has an employee count less than or equal to 25 does not need to have pure managers. The responsibility of the managers in this type of organization will be similar to that of a project manager. Pure managers are needed when personal resources exceed 25. Firstly, these managers understand the status of the project them, develop the plans and estimate the results. The manager should participate in developing the plans. This transition affects the software project managers.

2. Tangible design and requirements

The traditional processes utilize tons of paper in order to generate the documents relevant to the desired project. Even the significant milestones of a project are expressed via documents. Thus, the traditional process spends most of their crucial time on document preparation instead of performing software development activities.

An iterative process involves the construction of systems that describe the architecture, negotiates the significant requirements, identifies and resolves the risks etc. These milestones will be focused by all the stakeholders because they show progressive deliveries of important functionalities instead of documental descriptions about the project. Engineering teams will accept this transition of environment from to less document-driven while conventional monitors will refuse this transition.

3. Assertive Demonstrations are prioritized

The design errors are exposed by carrying-out demonstrations in the early stages of the life cycle. The stake holders should not over-react to these design errors because overemphasis of design errors will discourage the development organizations in producing the ambitious future iterating. This does not mean that stakeholders should bare all these errors. Infact, the stakeholders must follow all the significant steps needed for resolving these issues because these errors will sometimes lead to serious down-fall in the project.

This transition will unmark all the engineering or process issues so, it is mostly refused by management team, and widely accepted by users, customers and the engineering team.

4. The performance of the project can be determined earlier in the life cycle. The success and failure of any project depends on the planning and architectural phases of life cycle so, these phases must employ high-skilled professionals. However, the remaining phases may work well an average team.

5. Earlier increments will be adolescent

The development organizations must ensure that customers and users should not expect to have good or reliable deliveries at the initial stages. This can be done by demonstration of flexible benefits in successive increments. The demonstration is similar to that of documentation but involves measuring of changes, fixes and upgrades based on the objectives so as to highlight the process quality and future environments.

6. Artifacts tend to be insignificant at the early stages but proves to be the most significant in the later stages

The details of the artifacts should not be considered unless a stable and a useful baseline is obtained. This transition is accepted by the development team while the conventional contract monitors refuse this transition.

7. Identifying and Resolving of real issues is done in a systematic order

The requirements and designs of any successful project arguments along with the continuous negotiations and trade-offs. The difference between real and apparent issued of a successful project can easily be determined. This transition may affect any team of stakeholders.

8. Everyone should focus on quality assurance

The software project manager should ensure that quality assurance is integrated in every aspect of project that is it should be integrated into every individuals role, every artifact, and every activity performed etc. There are some organizations which maintains a separate group of individuals know as quality assurance team, this team would perform inspections, meeting and checklist in order to measure quality assurance. However, this transition involves replacing of separate quality assurance team into an organizational teamwork with mature process, common objectives and common incentives. So, this transition is supported by engineering teams and avoided by quality assurance team and conventional managers.

9. Performance issues crop up earlier in the projects life cycle Earlier performance issues are a mature design process but resembles as an immature design. This transition is accepted by development engineers because it enables the evaluation of performance tradeoffs in subsequent releases.

10. Automation must be done with appropriate investments

Automation is the key concept of iterative development projects and must be done with sufficient funds. Moreover, the stakeholders must select an environment that supports iterative development. This transition is mainly opposed by organizational managers.

11. Good software organizations should have good profit margins.

Most of the contractors for any software contracting firm focus only on obtaining their profit margins beyond the acceptable range of 5% and 15%. They don't look for the quality of finished product as a result, the customers will be affected. For the success of any software industry, the good quality and at a reasonable rate them, customer will not worry about the profit the contractor has made. The bad contractors especially in a government contracting firm will be against this transition.

Characteristics of conventional and iterative software development Process

The characteristics of the conventional software process are listed below:

1. It evolves in the sequential order (requirement design-code-test).
2. It gives the same preference to all the artifacts, components, requirements etc.

3. It completes all the artifacts of a stage before moving to the other stage in the project life cycle.

4. It achieves traceability with high-fidelity for al the artifacts present at each life cycle stage.

The characteristics of the modern iterative development process framework are listed below:

1. It continuously performs round-trip engineering of requirements, design, coding and testing at evolving levels of abstraction.
2. It evolves the artifacts depending on the priorities of the risk management.

3. It postpones the consistency analysis and completeness of the artifacts to the later stages in the life cycle.

4. It achieves the significant drives (i.e. 20 percent) with high-fidelity during the initial stages of the life cycle.