LECTURE NOTES ON

SOFTWARE PROJECT MANAGEMENT

III B.TECH I SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA
Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112
(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

SPM UNIT I

Conventional Software Management: The Waterfall Model, Conventional software Management Performance. Evolution of Software Economics: Software Economics, Pragmatic Software Cost Estimation

INTRODUCTION

- Conventional software Management Practices ear sound in theory, but practice is still tied to archaic technology and techniques.
- Conventional software economics provides a benchmark of performance for conventional software management principles.
- The best thing about software is its ***flexibility***: it can be programmed to do almost anything. The worst thing about software is also its flexibility: the "almost anything" characteristic has made it difficult to plan, monitors and control software development.

Three important analyses of the state of the software engineering industry are:

- Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates.
- Management discipline is more of a discriminator in success or failure than are technology advances.
- The level of software scrap and rework is indicative of an immature process.

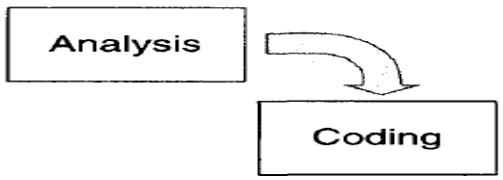
All three analyses reached the same general conclusion: The success rate for software projects is very low. The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.

1.1 THE WATERFALL MODEL (IN THEORY)

Most software engineering texts present the waterfall model as the source of the **"conventional"** software process. 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:

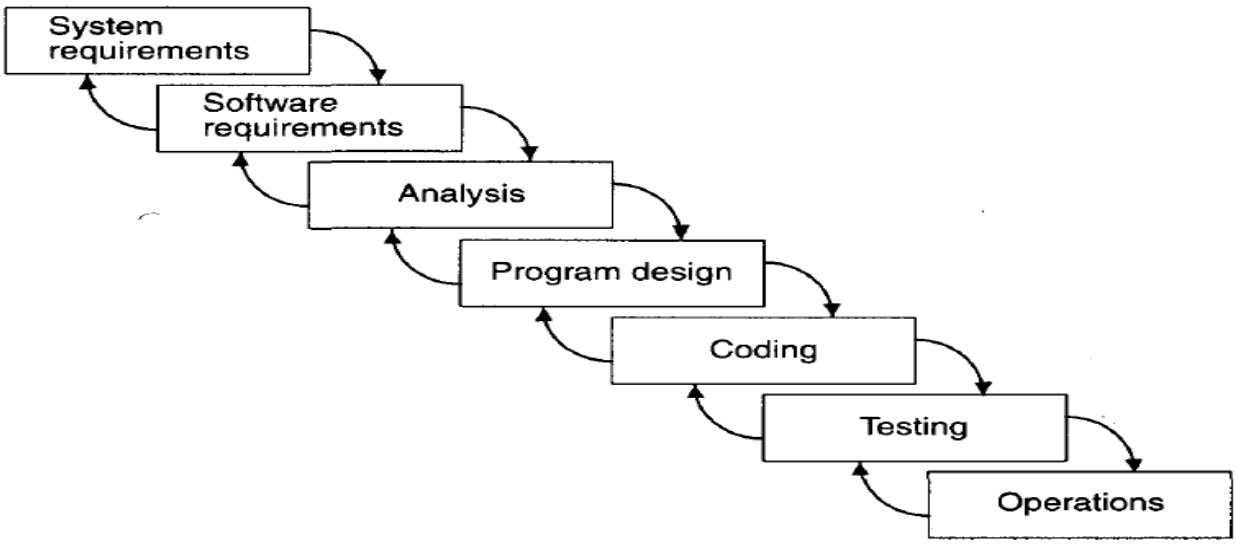
1. There are two essential steps common to the development of computer programs: analysis and coding.
2. In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other "overhead" steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps.
3. The basic framework described in the waterfall model is risky and invites failure. The testing phase that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished form analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violate. Either the requirements must be modified or a substantial design change is warranted.

Waterfall Model Part 1 : The two basic steps to building a program



Analysis and coding both involve creative work that directly contributes to the usefulness of the end product.

Waterfall Model Part 2 : The large-scale system approach



Waterfall Model Part 3 : Five necessary improvements for this approach to work

1. Complete program design before analysis and coding begin.
2. Maintain current and complete documentation.
3. Do the job twice, if possible.
4. Plan, control, and monitor testing.
5. Involve the customer.

Five necessary improvements for waterfall model are (the *risks may be eliminated by making the following five improvements*):-

(a) Program design comes first: The first step is to insert a preliminary program design phase between the software requirements phase and the analysis phase. Hence, by this technique, the software failure will not occur due to the continuous change in storage, timing and data. The designer then urges the storage, timing and operational limitations. On the analyst in such a way, that he notices the results. Resources insufficiently and the design limitations are identified in the early stages before final designing coding and testing. The following steps are required:

- Begin the design process with program designers, not analysts or programmers.
- Design, define, and allocate the data processing modes even at the risk of being wrong. Allocate processing functions, design the database, allocate execution time, define interfaces and processing modes with the operating system, describe input and output processing, and define preliminary operating procedures.
- 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.

(b) Document the design. The amount of documentation associated with the software programs is very large because of the following reasons:

(a) Each designer must communicate with interfacing designers, managers, and possibly customers.
(b) During early phases, the documentation is the design.
(c) The real monetary value of documentation is to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

(c) Do it twice. The Computer program must be developed twice and the second version, which takes into account all the critical design operations, must be finally delivered to the customer for operational development. The first version of the computer program involves a special board competence team, responsible for notifying the troubles in design, followed by their modeling and finally generating an error-free program.

(d) Plan, control, and monitor testing. The test phase is the biggest user of the project resources, such as manpower, computer time and management assessments. It has the greatest risk in terms of cost and schedules and develops at the most point in the schedule, when backup alternatives are least available. Thus, most of the problems need to be solved before the test phase, as it has to perform some other important operations.

(a) Hire a team of test specialists, who are not involved in the original design.
(b) Apply visual inspections to discover the obvious errors, such as skipping the wrong addresses, dropping of minus signs etc.
(c) Conduct a test for every logic path;
(d) Employ the final checkout on the target computer.

(e) **Involve the customer.** The customer must be involved in a formal way, so that, he has devoted himself at the initial stages before final delivery. The customer‟s perception, assessment and commitment can strengthen the development effort. Hence, an initial design step followed by a “preliminary software review", "critical software design reviews", during design and a "final software acceptance review" after testing is performed.

THE WATERFALL MODEL (IN PRACTICE)

- Some software projects still practice the conventional software management approach.
- It is useful to summarize the characteristics of the conventional process as it has typically been applied, which is not necessarily as it was intended. Projects destined for trouble frequently exhibit the following symptoms:
  (a) Protracted integration and late design breakage
  (b) Late risk resolution
  (c) Requirements – driven functional decomposition
  (d) Adversarial (conflict or opposition) stakeholder relationships
  (e) Focus on documents and review meetings.

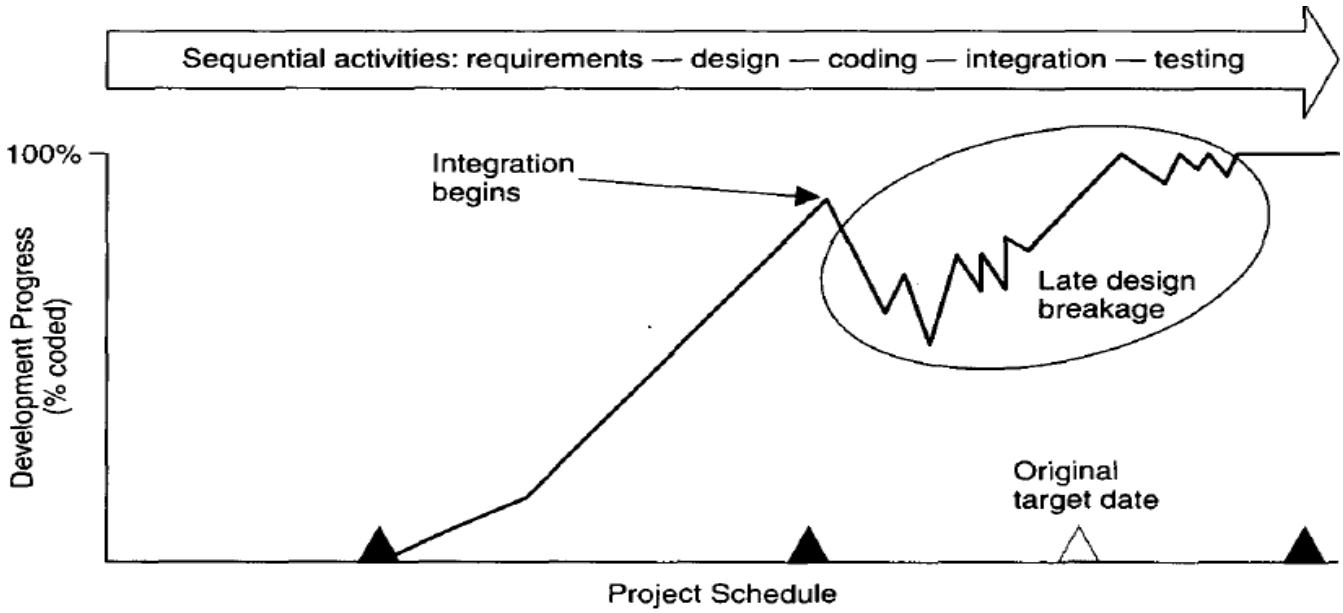
(a) Protracted Integration and Late Design Breakage

The percentage of progress achieved in the development process against time is shown in Figure.
The following sequence was common:

- Early success via paper designs and thorough (often *too* thorough) briefings
- Commitment to code late in the life cycle
- Integration nightmares (unpleasant experience) due to unforeseen implementation issues and inter
- Heavy budget and schedule pressure to get the system working
- Late shoe-homing of no optimal fixes, with no time for redesign
- A very fragile, unmentionable product delivered late

Hence, when a waterfall model is used in the process, late integration and performance degradation occurs.

Format	Ad hoc text	Flowcharts	Source code	Configuration baselines
Activity	Requirements analysis	Program design	Coding and unit testing	Protracted integration and testing
Product	Documents	Documents	Coded units	Fragile baselines



Progress profile of a conventional software project

Expenditures by activity for a conventional software project

ACTIVITY	COST
Management	5%
Requirements	5%
Design	10%
Code and unit testing	30%
Integration and test	40%
Deployment	5%
Environment	5%
Total	100%

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.

(b) Late risk resolution

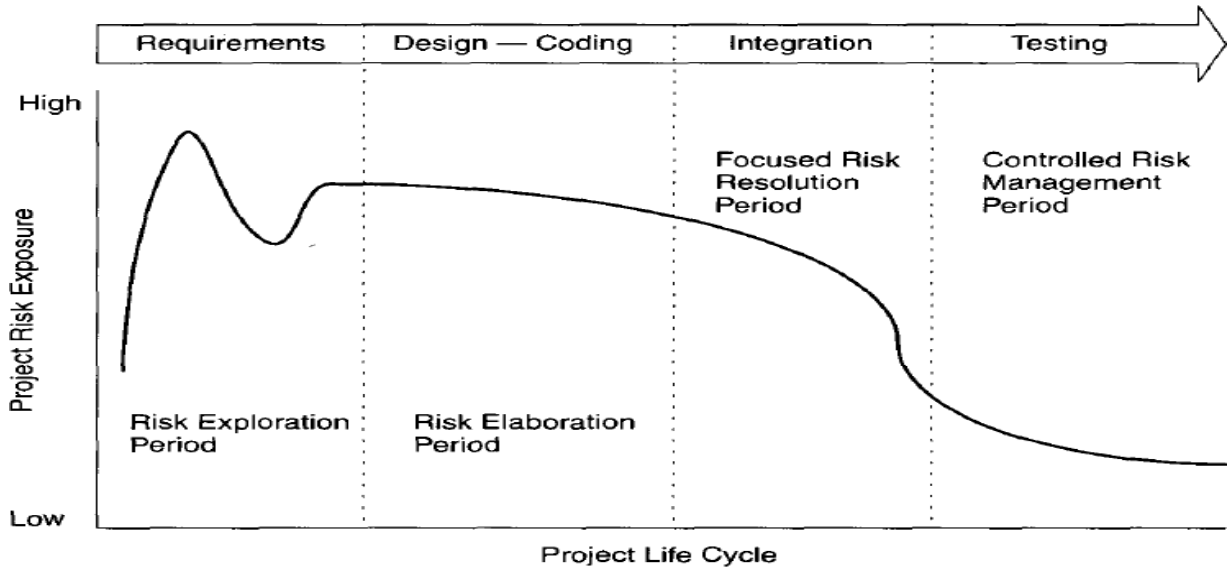
A serious issue associated with the waterfall lifecycle was the lack of early risk resolution.

A ***risk*** is defined as the probability of missing a cost, schedule, feature, or quality goal.

The Figure illustrates a typical risk profile for conventional waterfall model projects. It includes four distinct periods of risk exposure.

- Early in the life cycle, as the requirements· were being specified, the actual risk exposure was highly unpredictable.
- After a design concept is available to balance the understanding the requirements, the risk exposure is stabilized.

- When integration began, during this period real design issues were resolved and engineering tradeoffs were made.



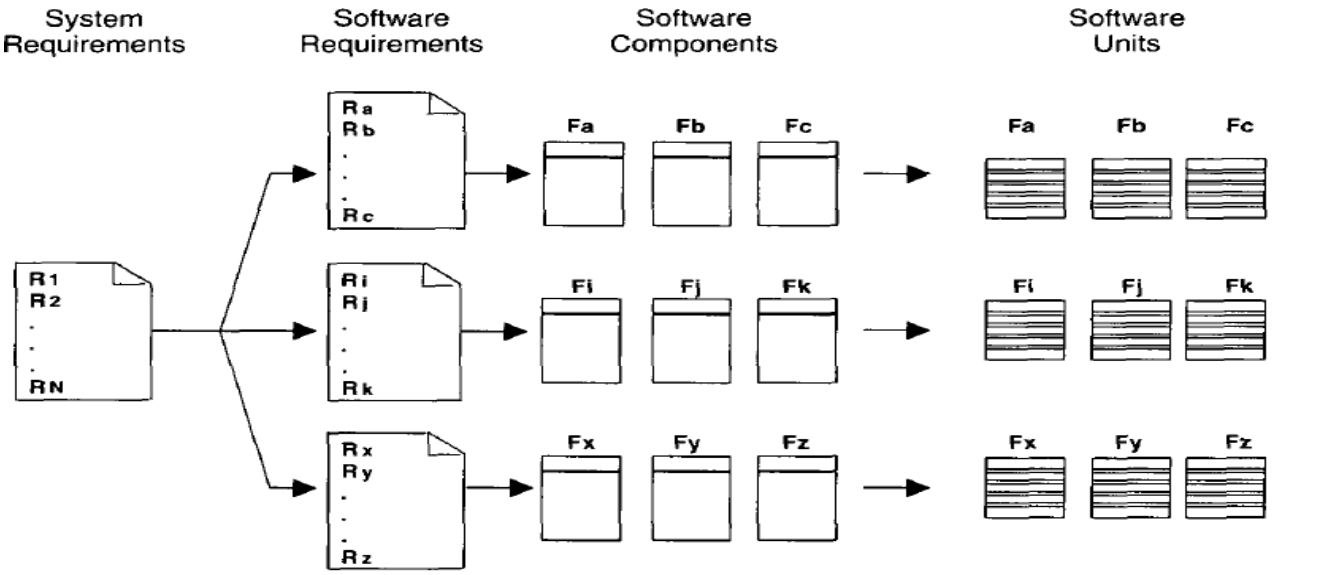
Risk profile of a conventional software project across its life cycle

(c) Requirements – driven functional decomposition

The software development process is requirements driven i.e., initially gives precise definition for the requirements and then provides implementation for them.

The requirements are specified completely and clearly before any other activities in the software development process. It immaturely treats all the requirements, equally. Requirements specification is an important and difficult job in the development process

The basic assumption of the waterfall process is that, requirements are specified in a functional manner, according to when the software can be divided into functions followed by the requirements allocation to the resulting components.



Suboptimal software component organization resulting from a requirements-driven approach

(d) Adversarial (conflict or opposition) stakeholder relationships

The conventional process tended to result in adversarial stakeholder relationships, in large part because of the difficulties of requirement specification and the exchange of information solely through paper documents that captured engineering information in ad hoc formats.

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

(a) The contractor prepared a draft contract-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.
(b) The customer was expected to provide comments (typically within 15 to 30 days).
(c) The contractor incorporated these comments and submitted (typically within 15 to 30 days) a final version for approval.

This one-shot review process encouraged high levels of sensitivity on the part of customers and contractors.

(e) Focus on documents and review meetings.

Emphasis on documents generation while describing a software product causes insufficient focus on producing tangible product increments.

Implementation of the major milestones can be done through documents specification. Contractors, rather than reducing the risks to improve the product quality, produces large amount of paper for creating the documents and only the simple things are reviewed.

Hence, most design reviews have low engineering value and high costs in terms of schedule and effort.

Results of conventional software project design reviews

APPARENT RESULTS	REAL RESULTS
Big briefing to a diverse audience	Only a small percentage of the audience understands the software. Briefings and documents expose few of the important assets and risks of complex software systems.
A design that appears to be compliant	There is no tangible evidence of compliance. Compliance with ambiguous requirements is of little value.
Coverage of requirements (typically hundreds)	Few (tens) are design drivers. Dealing with all requirements dilutes the focus on the critical drivers.
A design considered "innocent until proven guilty"	The design is always guilty. Design flaws are exposed later in the life cycle.

1.2. CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

Barry Boehm"s "***Industrial software Metrics top 10 List***" is a good, objective characterization of the state of software development.

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) **80/20 Principle:** 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.

SOFTWARE ECONOMICS

Most software cost models can be abstracted into a function of five basic parameters: size, process, personal, environment and required quality.

- The **size** of the end product (in human-generated components), which is typically quantified in terms of the number of source instructions or the number of function points required to develop the required functionality.
- The **process** used to produce the end product, in particular the ability of the process to avoid non-value-adding activities (rework, bureaucratic delays, communications overhead).
- The capabilities of software engineering **personnel**, and particularly their experience with the computer science issues and the applications domain issues of the project.
- The **environment**, which is made up of the tools and techniques available to support efficient software development and to automate the process
- The required **quality** of the product, including its features, performance, reliability and adaptability.

The relationships among these parameters and the estimated costs can be written as follows:

$$\textbf{Effort} = (\text{Personnel})(\text{Environment})(\text{Quality})(\text{Size}^{\text{process}})$$

One important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a **diseconomy of scale**. The diseconomy of scale of software development is a result of the process exponent being greater than 1.0. Contrary to most manufacturing processes, the more software you build, the more expensive it is per unit item.

The three generations or software development are defined as follows:

1) Conventional: 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.

2) Transition: 1980s and 1990s, 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, including the operating system, database management system, networking, and graphical user interface.

3) Modern practices: 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the shelf components. Perhaps as few as 30% of the components need to be custom built Technologies for environment automation, size reduction, and process improvement are not independent of one another. In each new era, the

key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.

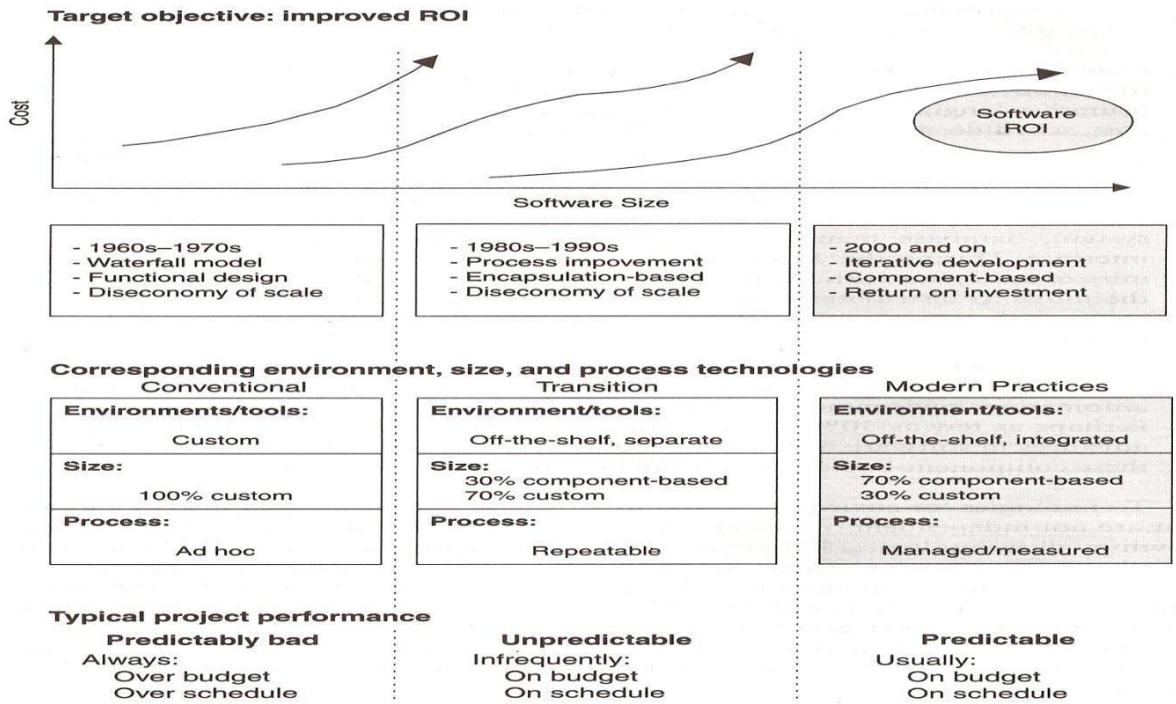


FIGURE 2-1. *Three generations of software economics leading to the target objective*

Organizations are achieving better economies of scale in successive technology eras-with very large projects (systems of systems), long-lived products, and lines of business comprising multiple similar projects. Figure 2-2 provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains.

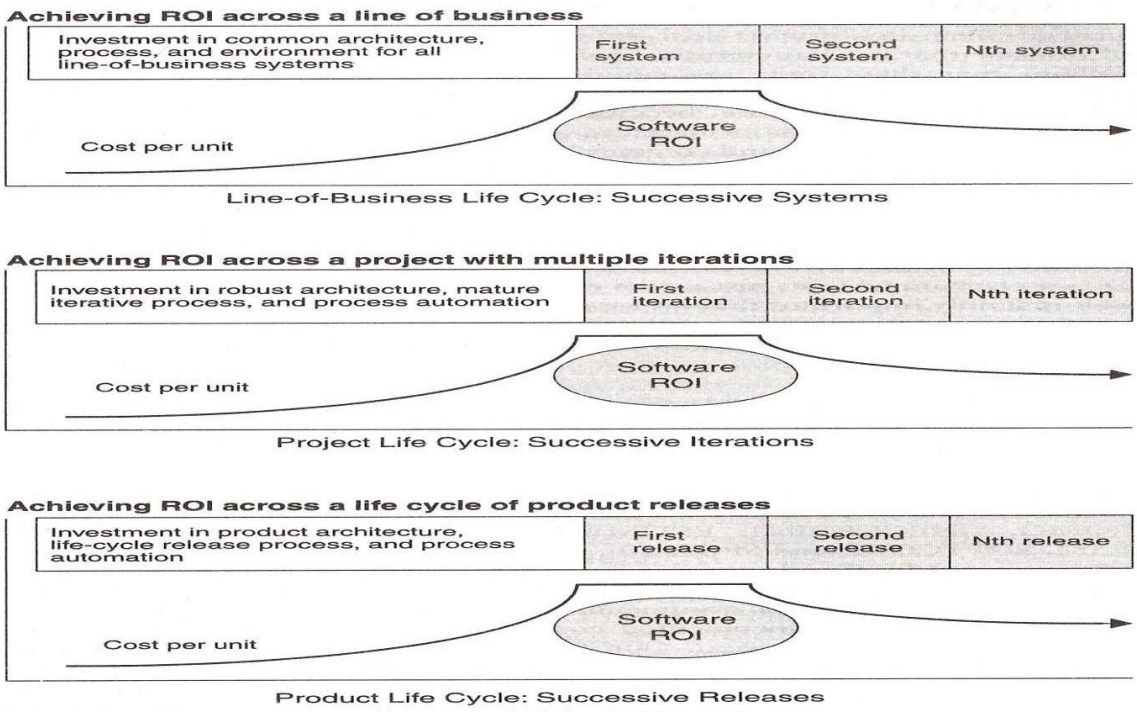


FIGURE 2-2. *Return on investment in different domains*

PRAGMATIC SOFTWARE COST ESTIMATION

One critical problem in software cost estimation is a lack of well-documented case studies of projects that used an iterative development approach. Software industry has inconsistently defined metrics or atomic units of measure, the data from actual projects are highly suspect in terms of consistency and comparability. It is hard enough to collect a homogeneous set of project data within one organization; it is extremely difficult to homogenize data across different organizations with different processes, languages, domains, and so on.
There have been many debates among developers and vendors of software cost estimation models and tools. Three topics of these debates are of particular interest here:

1. **Which cost estimation model to use?**
2. **Whether to measure software size in source lines of code or function points?**
3. **What constitutes a good estimate?**

There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), CO COMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described a "within 20% of actual, 70% of the time."

- Most real-world use of cost models is bottom-up (substantiating a target cost) rather than top-down (estimating the "should" cost). Figure 2-3 illustrates the predominant practice: 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. The rationale for the target cost may be to win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.
- The process described in figure 2-3 is not all bad. In fact, it is absolutely necessary to analyze the cost risks and understand the sensitivities and trade-offs objectively. It forces the software project manager to examine the risks associated with achieving the target costs and to discuss this information with other stakeholders.

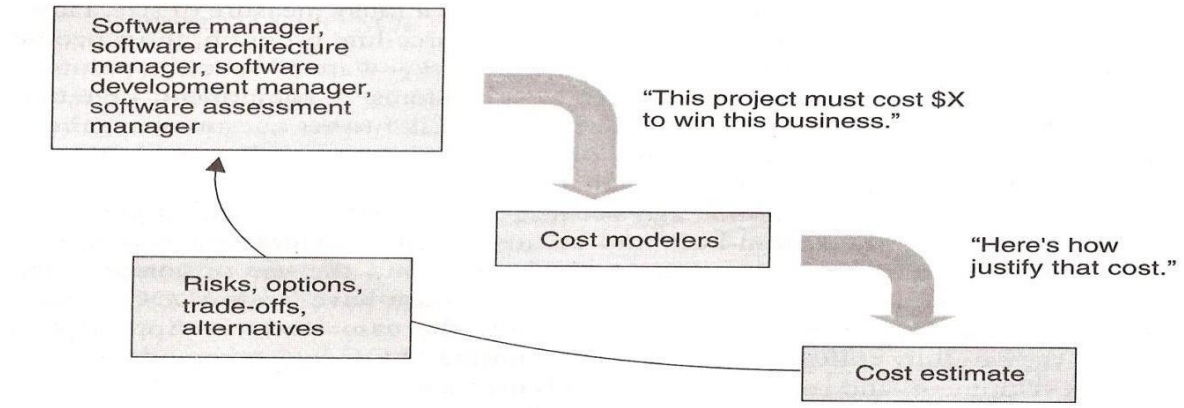


FIGURE 2-3. *The predominant cost estimation process*

A good software cost estimate has the following attributes:

- It is conceived and supported by the project manager, architecture team, development team and test accountable for performing the work
- It is accepted by all stakeholders as ambitious but realizable.
- It is based on a well-defined software cost model with a credible basis.
- It is based on a database of relevant project experience that includes similarprocesses, similar technologies, similar environments, similar quality requirements and similar people.
- It is define din enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

SPM UNIT II (Part-I)

Improving Software Economics: Reducing Software Product Size, Improving software Processes, Improving Team Effectiveness, Improving Automation, Achieving Required Quality, Peer Inspections

INTRODUCTION

Five basic parameters of the software cost model are

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

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

Table: 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

- The most significant way to improve affordability and return on investment (ROI) is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material.
- *Component-based development* is introduced here as the general term for reducing the "source" language size necessary to achieve a software solution.
- Reuse, object oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source.

- This size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic, and fourth-generation languages), automatic code generators (CASE tools, visual modeling tools, GUI builders), reuse of commercial components (operating systems, windowing environments, database management systems, middleware, networks), and object-oriented technologies (Unified Modeling Language, visual modeling tools, architecture frameworks).
- In general, when size-reducing technologies are used, they reduce the number of human-generated source lines.

LANGUAGES

- Universal function points (UFPs) are useful estimators for language-independent, early life-cycle estimates.
- The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries.
- SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Substantial data have been documented relating SLOC to function points.

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

a) Visual basic is very powerful and expressive in building simple interactive applications but it would not be used for real time, embedded.
b) Ada 95 might be the best language for a catastrophic, cost of failure system that controls nuclear power plant but it would not be used for highly parallel, scientific, number crunching program running on a super computer.
c) Ada 83 is used by Department of Defense (DOD) to increase it would provide it expressiveness.
d) C++ incorporated several advances with in Ada as well as advanced support for object oriented programming.
e) C compatability made easy for C programmer to transition to C++
f) The evolution of Java has eliminated many problems in C++, while conserving object oriented features and adding further support for portability and support.
UPFs (Universal Function Points) are useful estimators for language-independent in the early life cycle phases.

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++

The values indicate the relative expressiveness provided by various languages.

OBJECT-ORIENTED METHODS AND VISUAL MODELING

- There has been at widespread movement in the 1990s toward object-oriented technology. The advantages of object-oriented methods include improvement in software productivity and software quality. The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed.

Booch describes the following <u>three</u> reasons for the success of the projects that are using Object Oriented concepts:

1. An object-oriented model of the problem and its solution encourages a common vocabulary between the end users 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 destabilizing the entire development effort.
3. An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture**.**

Booch also summarized <u>five</u> characteristics of a successful object-oriented project:

1. A ruthless focus on the development of a system that provides a well understood collection of essential minimal characteristics.
2. The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.
3. The effective use of object-oriented modeling
4. The existence of a strong architectural vision
5. The application of a well-managed iterative and incremental development life cycle.

REUSE

- Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages.
- Software design methods have always dealt implicitly with reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality.
- Most truly reusable components of value are transitioned to commercial products supported by organizations with the following characteristics:
  1. They have an economic motivation for continued support.
  2. They take ownership of improving product quality, adding new features, and transitioning to new technologies.
  3. They have a sufficiently broad customer base to be profitable.
- Reuse is an important discipline that has an impact on the efficiency of all workflows and the quality of most artifacts. The cost of developing a reusable component is not trivial. The following Figure examines the economic tradeoffs. The steep initial curve illustrates the economic obstacle to developing reusable components.

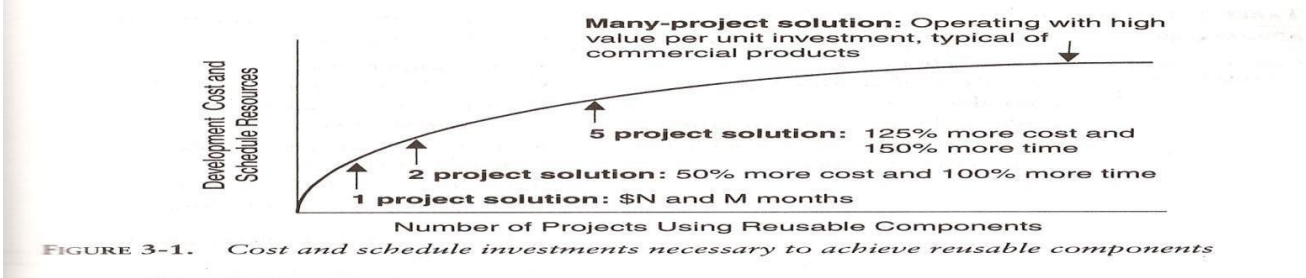


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

COMMERCIAL COMPONENTS

- A common approach being pursued today in may domains is to maximize integration of commercial components and off-the-shelf products.
- While the use of commercial components is certainly desirable as a means of reducing custom development, it has not proven to be straight forward in practice.
- The following Table identifies some of the advantages and disadvantages of using commercial components.

APPROACH	ADVANTAGES	DISADVANTAGES
Commercial components	1. Predictable license costs 2. Broadly used, mature technology Available now 3. Dedicated support organization 4. Hardware/ Software independence 5. Rich in functionality	1. Frequent upgrades 2. Up-front license fees 3. Recurring maintenance fees 4. Dependency on vendor 5. Run-time efficiency sacrifices 6. Functionality constraints. 7. Integration not always trivial 8. No control over upgrades and maintenance 9. Unnecessary features that consume extra resources 10. Often Inadequate reliability and Stability. 11. Multiple-vendor incompatibilities
Custom development	1. Complete change freedom 2. Smaller, often simpler implementations 3. Often better performance 4. Control of development and enhancement	1. Expensive, unpredictable development 2. Unpredictable availability date 3. Undefined maintenance model 4. Often immature and fragile 5. Single-platform dependency 6. Drain on expert resources

IMPROVING SOFTWARE PROCESSES

- Process is an overloaded term. For software-oriented organizations, there are many processes and sub processes. Three distinct process perspectives are:
  1. **Metaprocess:** an organization's policies, procedures, and practices for pursuing a software intensive line of business. The focus of this process is on organizational economics, long-term strategies, and software ROI.
  2. **Macroprocess:** a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The focus of the macro process is on creating an adequate instance of the Meta process for a specific set of constraints.
  3. **Microprocess:** a project team's policies, procedures, and practices for achieving an artifact of the software process. The focus of the micro process is on achieving an intermediate product baseline with adequate quality and adequate functionality as economically and rapidly as practical.

Table: 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 budge, 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/ 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

- Teamwork is much more important than the sum of the individuals. With software teams, a project manager needs to configure a balance of solid talent with highly skilled people in the leverage positions.
- Some maxims of team management include the following:
  1. A well-managed project can succeed with a nominal Engineering team.
  2. A mismanaged project will almost never succeed, even with an expert team of Engineers. A well-architected system can be built by a nominal team of software builders.
  3. A poorly architected system will flounder even with an expert team ofbuilders.
- **Boehm five staffing principles are:**
  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 harmonize with one another
  5. *The principle of phase-out*: Keeping a misfit on the team doesn't benefit anyone.
- **Software project managers need many leadership qualities** in order to enhance team effectiveness. The following are some crucial attributes of successful software project managers that deserve much more attention:
  1. **Hiring skills**: Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
  2. **Customer-interface skill**: Avoiding adversarial relationships among stakeholders is a prerequisite for success.

3. **Decision-Making skill**: The jillion books written about management have failed to provide a clear definition of this attribute.
4. **Team- building skill**: Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
5. **Selling skill**: Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives.

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

- The tools and environment used in the software process generally have a linear effect on the productivity of the process.
- Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces provide crucial automation support far evolving the software engineering artifacts.
- At first order, the isolated impact of tools and automation generally allows improvements of **20% to 40%** in effort.
- However, tools and environments must be viewed as the primary delivery vehicle for process automation and improvement, so their impact can be much higher.
- Automation of the design process provides payback in quality. The ability to estimate costs andschedules, and overall productivity using a smaller team. Integrated toolsets play an increasingly important role in incremental/iterative development by allowing the designers to traverse quickly among development artifacts and keep them up-to-date.
- ***Round-trip Engineering*** is a term used to describe the key capability of environments thatsupport iterative development.
- ***Forward Engineering*** is the automation of one engineering artifact from another, more abstract representation. For example, compilers and linkers have provided automated transition of source codeinto executable code.
- ***Reverse engineering*** is the generation or modification of a more abstract representation from an existing artifact.
- Economic improvements associated with tools and environments. It is common for tool vendors to make relatively accurate individual assessments of life-cycle activities to support claims about the potential economic impact of their tools. For example, it is easy to find statements such as the following from companies in a particular tool:
  1. Requirements analysis and evolution activities consume 40% of life-cycle costs.
  2. Software design activities have an impact on more than 50% of the resources.
  3. Coding and unit testing activities consume about 50% of software development effort and schedule.
4. Test activities can consume as much as 50% of a project's resources.
5. Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.
6. Documentation activities can consume more than 30% of project engineering resources.
7. Project management, business administration, and progress assessment can consume as much as 30% of project budgets.

ACHIEVING REQUIRED QUALITY

Software best practices are derived from the development process and technologies. Key practices that improve overall software quality include the following:

1. ***Focusing on driving requirements*** and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the life cycle, and focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan evolution.
2. ***Using metrics and indicators*** to measure the progress and quality of architecture as it evolves from a high-level prototype into a fully compliant product.
3. ***Providing integrated life-cycle environments*** that support early and continuous configuration control, change management, rigorous design methods, document automation, and regression test automation.
4. ***Using visual modeling and higher level languages*** that support architectural control, abstraction, reliable programming, reuse, and self-documentation
5. ***Early and continuous insight*** into performance issues through demonstration-based evaluations.

Conventional development processes stressed early sizing and timing estimates of computer program resource utilization. However, the typical chronology of events in performance assessment was as follows:

1. **Project Inception:** The proposed design was asserted to be low risk with adequate performance origin.
2. **Initial design review:** Optimistic assessments of adequate design margin were based mostly on paper analysis or ought simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood.
3. **Mid-life-cycle design review:** The assessments started whittling away at the margin, as early benchmarks and initial tests began exposing the optimism inherent in earlier estimates.
4. **Integration 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 poorlyunderstood early design trade-offs.

PEER INSPECTIONS: A PRAGMATIC VIEW

- Peer inspections are frequently over hyped as the key aspect of a quality system. In my experience, peer reviews are valuable as secondary mechanisms, but they are rarely significant contributors to quality compared with the following primary quality mechanisms and indicators, which should be emphasized in the management process:
  1. Transitioning Engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in theengineering artifacts.
  2. Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases
  3. Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control
  4. Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria andrequirements compliance.
  5. Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals.

Inspections are also a good vehicle for holding authors accountable for quality products. All authors of software and documentation should have their products scrutinized as a natural by product of the process. Therefore, the coverage of inspections should be across all authors rather than across all components.

SPM UNIT II (Part-II) and UNIT-III (Part-I)

The Old way and the NEW way: Principles of Conventional Software Engineering, Principles of Modern Software Management, Transitioning to an Iterative Process.

Life Cycle Phases: Engineering and Production Stages, Inception. Elaboration, Construction, Transition Phases.

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:

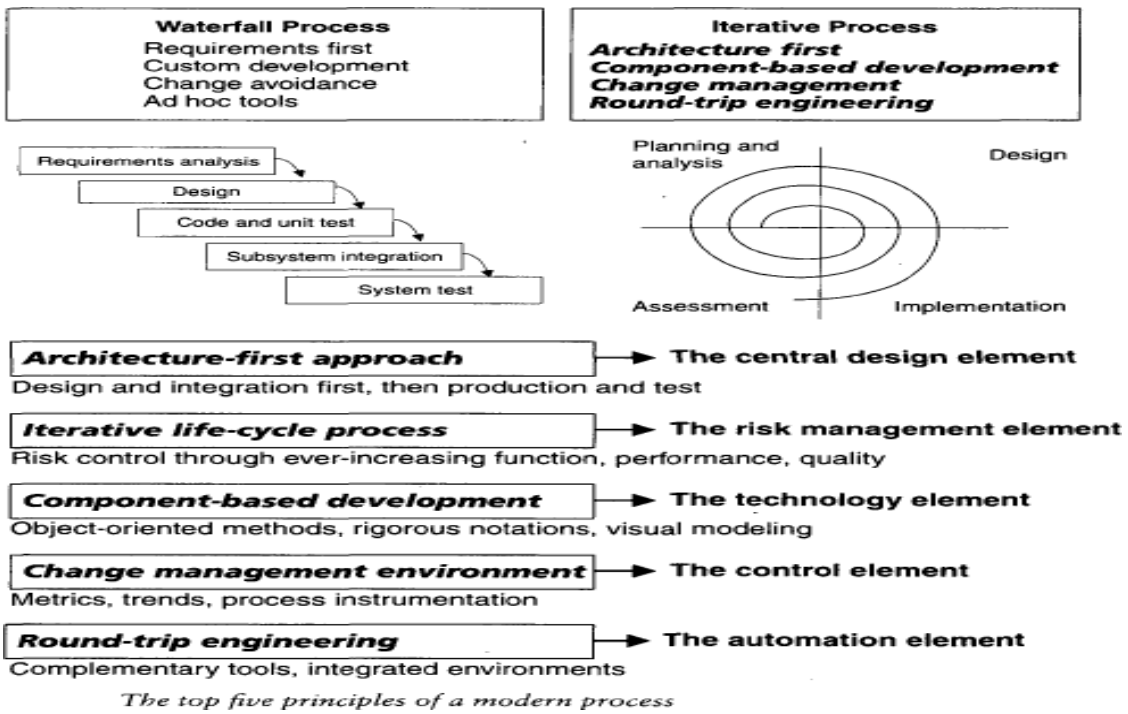
1. **Make quality #1:** Quality must be quantified and mechanisms put into place to motivate its achievement.
2. **High-quality software is possible:** Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people.
3. **Give products to customers early:** No matter how hard you try to learn users needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it.
4. **Determine the problem before writing the requirements:** When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.
5. **Evaluate Design Alternatives:** After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model:** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases:** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle.
8. **Minimize Intellectual Distance:** To minimize intellectual distance, the software's structure should be as close as possible to the real –world structure.
9. **Put techniques before tools:** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software Engineer.
10. **Get it right before you make it faster:** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding.
11. **Inspect Code:** Inspecting the detailed design and code is a much better way to find errors than testing.
12. **Good Management is more important than good technology:** Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success:** Highly skilled people with appropriate experience, talent, and training are key.
14. **Follow with Care:** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.
15. **Take responsibility:** When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs
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 thy 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 crate 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 Total 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 will become more and more disorganized.
29. **People and time are not interchangeable:** Measuring a project solely by person-months makes little sense.
30. **Expect Excellence:** Your employees will do much better if you have high expectations for them.

PRINCIPLES OF MODERN' SOFTWARE MANAGEMENT

Top 10 principles of modern software management are:

1. **Base the process on an architecture-first approach:** This requires that a demonstrable balance be achieved among the driving requirements, the architecturally significant design decisions, and the life-cycle plans before the resources are committed for full-scale development.
2. **Establish an iterative life-cycle process that confronts risk early that confronts risk early:** With today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, then test the end product in sequence. Instead, an iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages a balanced treatment of all stakeholder objectives. 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:** Moving from a line-of-code mentality to a component-based mentality is necessary to reduce the amount of human-generated source code and custom development.



The top five principles of a modern process

4. **Establish a change Management Environment:** the dynamics of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baselines.
5. **Enhance change freedom through tools that support round-trip Engineering:** Round trip engineering is the environment support necessary to automate and synchronize engineering information in different formats (such as requirements specifications, design models, source code, executable code, test cases).
6. **Capture design artifacts in rigorous, model-based notation:** A model based approach (such as UML) supports the evolution of semantically rich graphical and textural design notations.
7. **Instrument the process for objective quality control and progress assessment:** Life-cycle assessment of the progress and the quality of all intermediate products must be integrated into the process.
8. **Use a demonstration-based approach:** to assess intermediate artifacts.
9. **Plan intermediate releases in groups of usage scenarios with evolving levels or detail:** It is essential that the software management process drive toward early and continuous demonstrations within the operational context of the system, namely its use cases.
10. **Establish a configurable process that is economically scalable:** No single process suitable for all software developments.

Modern process approaches for solving conventional problems

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

TRANSITIONING TO AN ITERATIVE PROCESS

- Modern software development processes have moved away form the conventional waterfall model, in which each stage of the development process is dependent on completion of the previous stage.
- The economic benefits inherent in transitioning from the conventional waterfall model to an iterative development process are significant but difficult to quantify.

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

The following paragraphs map the process exponent parameters of CO COMO II to my top 10 principles of a modern process:

1. **Application Precedentedness:** domain experience is a critical factor in understanding how to plan and execute a software development project. Early iterations in the life cycle establish precedents from which the product, the process and the plans can be elaborated in evolving levels of detail.
2. **Process flexibility:** Development of modern software is characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. A configurable process that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.
3. **Architecture Risk Resolution**: Architecture-first development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes architecture before developing all the components that make up the entire suite of applications components. An Architecture-first and component-based development approach forces tile infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architectureprocess.
4. **Team Cohesion**: Successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Advances in technology (such as programming languages, UML, and visual modeling) have enabled more rigorous and understandable notations for communicating software engineering information, particularly in the requirements and design artifacts that previously were ad hoc and based completely on paper exchange. These model-based formats have also enabled the round-trip engineering support needed to establish change freedom sufficient for evolving design representations.
5. **Software Process Maturity**: The Software Engineering Institute's Capability Maturity Model (CMM) is a well-accepted benchmark for software process assessment. One of key themes is that truly mature processes are enabled through an integrated environment that provides the appropriate level of automation to instrument the process for objection quality control.

LIFE CYCLE PHASES

A modern software development process must be defined to support the following:

1. Evolution of the plans, requirements, and architecture, together with well defined synchronization points
2. Risk management and objective measures of progress and quality
3. Evolution of system capabilities through demonstrations of increasing functionality

Engineering and Production Stages

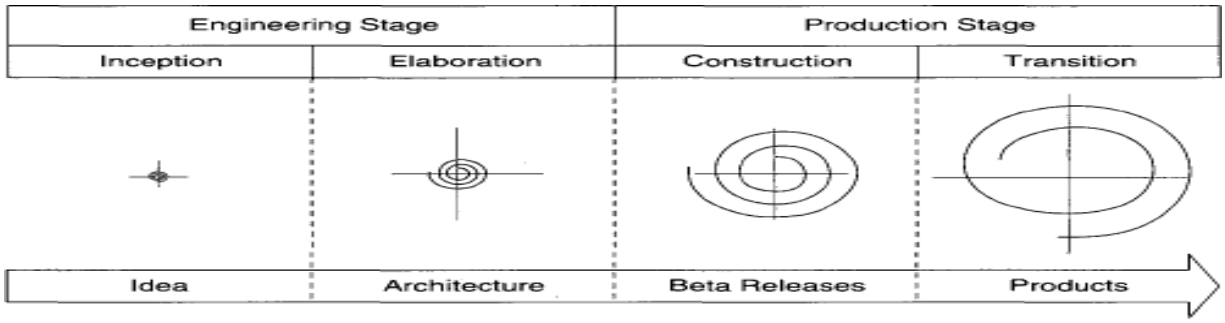
To achieve economies of scale and higher returns on investment, we must move toward a software manufacturing process driven by technological improvements in process automation and component based development. Two stages of the life cycle are:

1. The **Engineering stage,** driven by less predictable but smaller teams doing design and synthesis activities.
2. The **Production stage,** driven by more predictable but larger teams doing construction, test, and deployment activities.

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

- The transition between engineering and production is a crucial event for the various stakeholders. The production plan has been agreed upon, and there is a good enough understanding of the problem and the solution that all stakeholders can make a firm commitment to go ahead with production.
- Engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition. These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model as shown in Figure:



The phases of the life-cycle process

INCEPTION PHASE

- The goal of this phase is to achieve concurrence among stakeholders on the lifecycle objectives for the project.

❑ **Primary Objectives**

- Establishing the project's software scope and boundary condition, including all operational concept, acceptance criteria, and a clear understanding of what is and is not intended to be in the product.
- Discriminating the critical use cases of the system and the primary scenarios of operation that will drive the major design trade-offs.
- Demonstrating at least one candidate architecture against some of the primary scenarios.
- Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase).
- Estimating potential risks (sources of un predictability)

❑ **Essential Activities**

- Formulating the scope of the project. The information repository should be sufficient to define the problem space and derive the acceptance criteria for the end product.
- Synthesizing the architecture: An information repository is created that is sufficient to demonstrate the feasibility of at least one candidate architecture and an, initial baseline of make/buy decisions so that the cost, schedule, and resource estimates can be derived.
- Planning and preparing a business case. Alternatives for risk management, staffing, iteration plans, and cost/schedule/profitability trade-offs are evaluated.

❑ **Primary Evaluation Criteria**

- Do all stakeholders concur on the scope definition and cost and schedule estimates?
- Are requirements understood, as evidenced by the fidelity of the critical use cases?
- Are the cost and schedule estimates, priorities, risks, and development processes credible?
- Do the depth and breadth of an architecture prototype demonstrate the preceding criteria?
- Are actual resource expenditures versus planned expenditures acceptable?

ELABORATION PHASE

- At the end of this phase, the "Engineering" is considered complete. The elaboration phase activities must ensure that the architecture, requirements, and plans are stable enough, and the risks sufficiently mitigated, that the cost and schedule for the completion of the development call be predicted within an acceptable range. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size and risk.

❑ **Primary Objectives**

- Base lining the architecture as rapidly as practical (establishing a configuration-managed snapshot in which all changes are rationalized, tracked, and maintained)
- Base lining the vision
- Base lining a high-fidelity plan for the construction phase
- Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonabletime

❑ **Essential Activities**

- Elaborating the vision.
- Elaborating the process and infrastructure.
- Elaborating the architecture and selecting components.

❑ **Primary Evaluation Criteria**

- Is the vision stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
- Is the construction phase plan of sufficient fidelity, and is it backed up with a credible basis of estimate?
- Do all stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system in the context of the current architecture?
- Are actual resource expenditures versus planned expenditures acceptable?

CONSTRUCTION PHASE

- During the construction phase, all remaining components and application features are integrated into the application, and all features are thoroughly tested. Newly developed software is integrated where required. The construction phase represents a production process, in which emphasis is placed on managing resources and controlling operations to optimize costs, schedules and quality.

❑ **Primary Objectives**

- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework
- Achieving adequate quality as rapidly as practical
- Achieving useful versions (alpha, beta and other test releases) as rapidly as practical

❑ **Essential Activities**

- Resource management, control and process optimization
- Complete component development and testing against evaluation criteria.
- Assessment of product releases against acceptance criteria of the vision.

❑ **Primary Evaluation Criteria**

- Is this product baseline mature enough to be deployed in the user community?
- Is this product baseline stable enough to be deployed in the user community?
- Are the stakeholders ready for transition to the user community?
- Are actual resource expenditures versus planned expenditures acceptable?

TRANSITION PHASE

- The transition phase is entered when a baseline is mature enough to be deployed in the end-user domain. This typically requires that a usable subset of the system has been achieved with acceptable quality levels and user documentation so that transition to the user will provide positive results. This phase could include any of 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 user and maintainers- transition phase concludes when the deployment baseline has achieved the complete vision.

❑ **Primary Objectives**

- Achieving user self-supportability
- Achieving stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision
- Achieving final produce baselines as rapidly and cost-effectively as practical.

❑ **Essential Activities**

- Synchronization and integration of concurrent construction increments into consistent deployment baselines
- Deployment-specific engineering Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set.

❑ **Primary Evaluation Criteria**

- Is the user satisfied?
- Are actual resource expenditures versus planned expenditures acceptable?

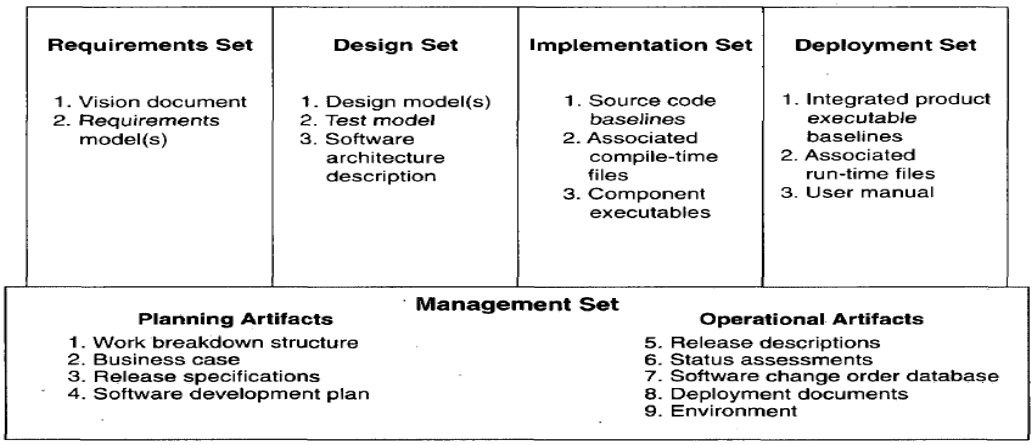
SPM UNIT III (Part-II)

Artifacts of the Process: The Artifact Sets. Management Artifacts, Engineering Artifacts, Programmatic Artifacts. Model Based Software Architectures: A Management Perspective and Technical Perspective.

THE ARTIFACT SETS

- To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. Artifact represents cohesive information that typically is developed and reviewed as a single entity.
- Life-cycle software artifacts are organized into ***five distinct sets*** that are roughly partitioned by the underlying language of the set:
  1. Management (ad hoc textual formats),
  2. Requirements (organized text and models of the problem space,
  3. Design (models of the solution space),
  4. Implementation (human-readable programming, language and associated source files), and
  5. Deployment (machine-process able languages and associate files).

The artifact sets are shown in the following figure:



Overview of the artifact sets

The Engineering sets consist of the requirements set, the design set, the implementation set, and the deployment set.

The Management Set:

- The management set captures the artifacts associated with process planning and execution.
- These artifacts use ad hoc notations, including text, graphics, or whatever representation is required to capture the "contracts" among project personnel (project management, architects, developers, testers, marketers, administrators), among stakeholders (funding authority, user, software project manager, organization manager, regulatory agency), and between project personnel and stakeholders.
- Specific artifacts included in this set are the work breakdown structure (activity breakdown and financial tracking mechanism), the business case (cost, schedule, profit expectations), the release specifications (scope, plan, objectives for release baselines), the software development plan (project process instance), the release descriptions (results of release baselines), the status assessments (periodic snapshots of project progress), the software change orders (descriptions of discrete baseline changes), the deployment documents (cutover plan, training course, sales rollout kit), and the environment (hardware and software tools, process automation & documentation).
- Management set artifacts are evaluated, assessed, and measured through a combination of the following:
  - Relevant stakeholder review.
  - Analysis of changes between the current version of the artifact and previous versions.
  - Major milestone demonstrations of the balance among all artifacts and, in particular, the accuracy of the business case and vision artifacts.

Requirements Set:

Requirements artifacts are evaluated, assessed, and measured through a combination of the following:

- Analysis of consistency with the release specifications of the management set.
- Analysis of consistency between the vision and the requirements models.
- Mapping against the design, implementation, and deployment sets to evaluate the consistency and completeness and the semantic balance between information in the different sets.
- Analysis of changes between the current version of requirements artifacts and previous versions (scrap, rework, and defect elimination trends).
- Subjective review of other dimensions of quality.

Design Set

UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design set is evaluated, assessed and measured through a combination of the following:

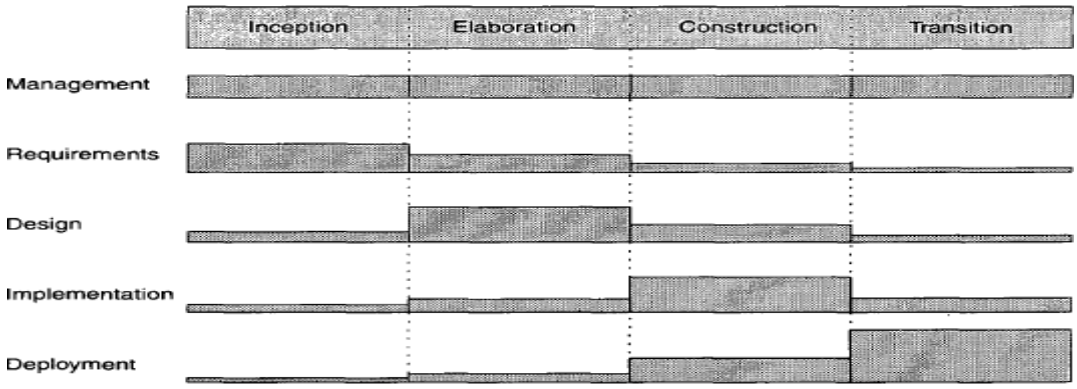
- Analysis of the internal consistency and quality of the design model
- Analysis of consistency with the requirements models
- Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets.
- Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends).
- Subjective review of other dimensions of quality.

Implementation Set

- The implementation set includes source code (programming language notations) that represents the tangible implementations of components (their form, interface, and dependency relationships).
- Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of the following:
  - Analysis of consistency with the design models.
  - Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets.
  - Assessment of component source or executable files against relevant evaluation criteria through inspection, analysis, demonstration, or testing
  - Execution of stand-alone component test cases that automatically compare expected results with actual results.
  - Analysis of changes between the current version of the implementation set and previous versions (scrap, rework, and defect elimination trends).
  - Subjective review of other dimensions of quality.

Deployment Set

- The deployment set 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 the following:
  - 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.
  - Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system (platform type, number, network topology).
  - Testing against the defined usage scenarios in the user manual such as installation, user oriented dynamic reconfiguration, mainstream usage, and anomaly management
  - Analysis of changes between the current version of the deployment set and previous versions (defect elimination trends, performance changes).
  - Subjective review of other dimensions of quality.



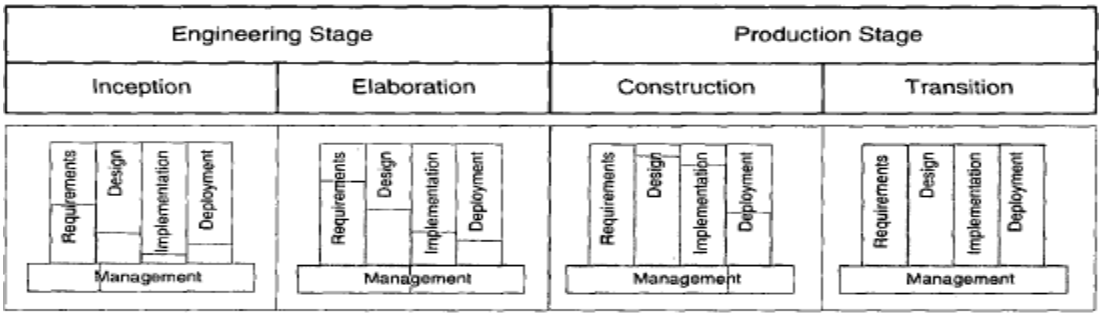
Life-cycle focus on artifact sets

Most of today"s software development tools map closely to one of the five artifact sets.

1. ***Management***: scheduling, workflow, defect tracking, change management, documentation, spreadsheet resource management, and presentation tools.
2. ***Requirements***: requirements management tools.
3. ***Design***: visual modeling tools.
4. ***Implementation***: compiler/debugger tools, code analysis tools, test coverage analysis tools, and test management tools.
5. ***Deployment***: test coverage and test automation tools, network management tools, commercial components (OS, GUIs, RDBMS, networks, middleware), and installation tools.

Artifact Evolution over the Life Cycle

Each state of development represents a certain amount of precision in the final system description. Early in the lif e cycle, precision is low and the representation is generally high. Eventually, the precision of representation is high and everything is specified in full detail. Each phase of development focuses on a particular artifact set. At the end of each phase, the overall system state will have progressed on all sets, as illustrated in following figure:



Life-cycle evolution of the artifact sets

The ***inception*** phase focuses mainly on critical requirements usually with a secondary focus on an initial deployment view. During the ***elaboration*** phase, there is much greater depth in requirements, much more breadth in the design set, and further work on implementation and deployment issues. The main focus of the ***construction*** phase is design and implementation. The main focus of the ***transition*** phase is on achieving consistency and completeness of the deployment set in the context of the other sets.

Test Artifacts

- The test artifacts must be developed concurrently with the product from inception through deployment. Thus, testing is 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.
    - **Management Set**: The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone.
    - **Requirements Set**: The system-level use cases capture the operational concept for the system and the acceptance test case descriptions, including the expected behavior of the system and its quality attributes.
    - **Design Set**: A test model for non deliverable components needed to test the product baselines is captured in the design set.
    - **Implementation Set**: Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts.
    - **Deployment Set:** Executable versions of test components, test drivers, and data files are provided.

MANAGEMENT ARTIFACTS

The management set includes several artifacts that capture intermediate results and ancillary information necessary to document the product/process legacy, maintain the product, improve the product and improve the process.

Management Set

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

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.
- The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment.

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

Typical business case outline

Work Breakdown Structure:

- Work breakdown structure (WBS) is the vehicle for budgeting and collecting costs.
- To monitor and control a project‟s financial performance, the software project manager must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

Software Change Order Database:

Managing change is one of the fundamental primitives of an iterative development process. With greater change freedom, a project can iterate more productively. This flexibility increases the content, quality and number of iterations that a project can achieve within a given schedule. Change freedom has been achieved in practice through automation,

and today"s iterative development environments carry the burden of change management. Organizational processes that depend on manual change management techniques have encountered major inefficiencies.

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 understanding matures.

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

Typical release specification outline

Software Development Plan:

The software development plan (SDP) elaborates the process framework into a fully detailed plan.
Two indications of a useful SDP are periodic updating (it is not stagnant shelf ware) and understanding and acceptance by managers and practitioners alike.

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)

Typical software development plan outline

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.

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

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. Typical status assessments should include a review of resources, personnel staffing, financial data (cost and revenue), top 10 risks, technical progress (metrics snapshots), major milestone plans and results, total project or product scope & action items.

Environment:

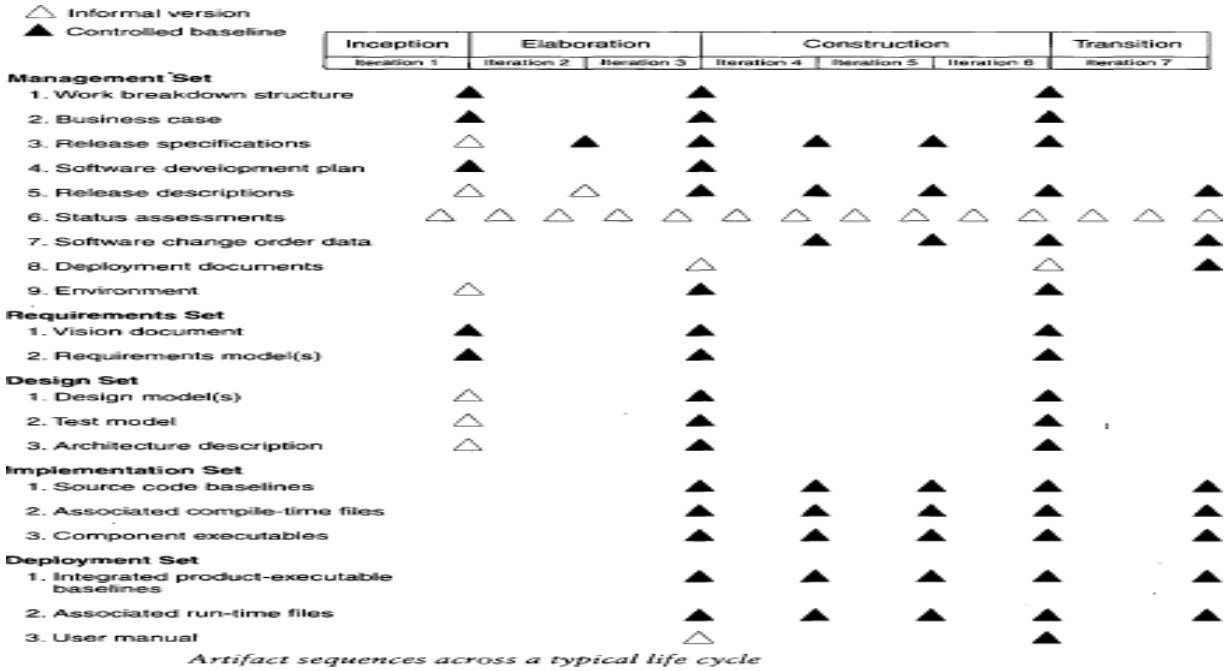
An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support automation of the development process. This environment should include requirements management, visual modeling, document automation, host and target programming tools, automated regression testing, and continuous and integrated change management, and feature and defect tracking.

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 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.

Management Artifact Sequences

In each phase of the life cycle, new artifacts are produced and previously developed m1ifacts are updated to incorporate lessons learned and to capture further depth and breadth of the solution. The following figure identifies a typical sequence of artifacts across the life-cycle phases.



Artifact sequences across a typical life cycle

4.3. ENGINEERING ARTIFACTS

Most of the engineering artifacts are captured in rigorous engineering notations such as UML, programming languages, or executable machine codes. Three engineering artifacts are explicitly intended for more general review, and they deserve further elaboration.

Vision document

- The vision document provides a complete vision for the software system under development and supports the contract between the funding authority and eth development organization.
- A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans and technology.
- A good visions document should change slowly.

The following figure provides a default outline for a visions 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)

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 et will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. The following figure provides a default outline form an architecture description.

Software Use Manual

- The software user manual provides the user with the reference documentation necessary to support 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 interfaced, this manual should be developed early in the life cycle because it is a necessary mechanism for communication 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.

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**

Typical architecture description outline

PRAGMATIC ARTIFACTS

a) **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 is not uncommon to find people (such as veteran software managers, veteran quality assurance specialists, or an auditing authority from a regulatory agency) who react as follows: "I"m not going to learn UML, but I want to review the design of this software, so give me a separate description such as some flowcharts and text that I can understand."
b) **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 is extremely rare that the/other stakeholders have any capability to review the engineering artifacts on-line. Consequently, organization is forced to exchange paper documents. Standardized formats (such as UML, spreadsheets, Visual Basic, C++ and Ada 95), visualization tools, and the web are rapidly making it economically feasible for all stakeholders to exchange information electronically.
c) **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. Acronyms and abbreviations should be used only where they are well accepted jargon in the context of the component"s usage. Readability should be emphasized and the use of proper English words should be required in all engineering artifacts. This practice enables understandable representations, browse able formats (paperless review), more-rigorous notations, and reduced error rates.
d) **Useful documentation is self-defining:** It is documentation that gets used.
e) **Paper is tangible; electronic artifacts are too easy to change.** On-line and Web-based artifacts can be changed easily and are viewed with more skepticism because of their inherent volatility.

ARCHITECTURE: A MANAGEMENT PERSPECTIVE

- The most critical technical product of a software project is its architecture: the infrastructure, control and date interfaces that permit software components to co-operate as a system and software designers to co-operate efficiently as a tem. When the communications media include multiple languages and inter group literacy varies, the communications problem can become extremely complex and even unsolvable. If a software development team is to be successful, the inter project communications, as captured in the software architecture, must be both accurate and precise.
- From a management perspective, there are three difference aspects of architecture.
    - ***An architecture*** (the intangible design concept) is the design of a software system this includes all engineering necessary to specify a complete bill of materials.
    - ***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 and people).
    - ***An architecture description*** (a human-readable representation of an architecture, which is one of the components of an architecture baseline) is an organized subset of information extracted form the design set model(s). The architecture description communicates how the intangible concept is realized in the tangible artifacts.

The importance of software architecture and its close linkage with modern software development processes can be summarized as follows:

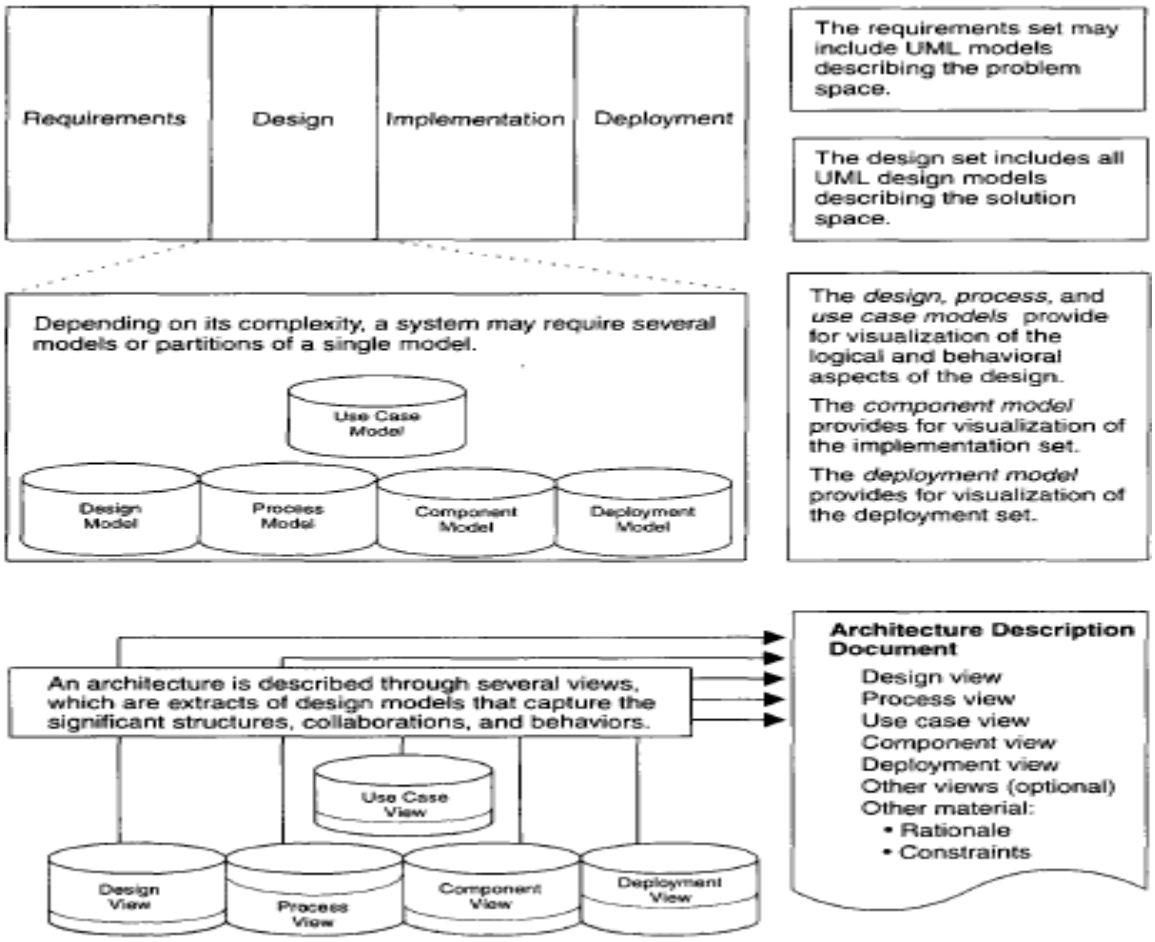
- Achieving stable software architecture represents a significant project milestone at which the critical make/buy decisions should have been resolved.
- Architecture representations provide a basis for balancing the trade-offs between the problem space (requirements and constraints) and the solution space (the operational product).
- The architecture and process encapsulate many of the important (high-payoff or high-risk) communications among individuals, teams, organizations and stakeholders.
- Poor architectures and immature processes are often given as reasons for project failures.
- A mature process, an understanding of the primary requirements, and a demonstrable architecture are important prerequisites fro predictable planning.
- 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

- An architecture framework is defined in the terms of views that are abstractions of the UML models in the design set. The design model includes the full breadth and depth of information. An architecture view is an abstraction of the design model; it contains only the architecturally significant information. Most real-world systems require four views: design, process, component and deployment. The purposes of these views are as follows:
  - ***Design:*** Describes architecturally significant structures and functions of the design model.
  - ***Process:*** Describes concurrency and control thread relationship among the design, component and deployment views.
  - ***Component:*** Describes the structure of the implementation set.
  - ***Deployment:*** Describes the structures of the deploy.

The following Figure Summarizes the artifacts of the design set, including the architecture views and architecture description:



Architecture, an organized and abstracted view into the design models

The ***requirements model*** addresses the behavior of the system as seen by its end users, analysts, and testers. This view is modeled statically using use case and class diagrams and dynamically using sequence, collaboration, state chart and activity diagrams.

The ***use case view*** describes how the system‟s critical (architecturally significant) use cases are realized by elements of the design model. It is modeled statically using use case diagrams and dynamically using any of the UML behavioral diagrams.

The ***design view*** describes the architecturally significant elements of the design model. This view, an abstraction of the design model, addresses the basic structure and functionality of the solution. It is modeled statically using calls and object diagrams and dynamically using any of the UML behavioral diagrams.

The ***process view*** addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including the logical software network topology (allocation to process and threads of control), inter

process communication and state management. This view is modeled statically using deployment diagrams and dynamically using any of the UML behavioral diagrams.

The ***component view*** describes the architecturally significant elements of the implementation set. This view, an abstraction of the design model, addresses the software source code realization of the system from the perspective of the project‟s integrators and developers, especially with regard to releases and configuration management. It is modeled statically using component diagrams and dynamically using any of the UML behavioral diagrams.

The ***deployment view*** addresses the executable realization of the system, including the allocation of logical processes in the distribution view (the logical software topology) to physical resources of the deployment network (the physical system topology). It is modeled statically using deployment diagrams and dynamically using any of the UML behavioral diagrams.

Generally, an architecture baseline should including the following:

- Requirements: critical use cases system-level quality objectives and priority relationships among features and qualities
- Design: names, attributes, structures, behaviors, groupings and relationships of significant classes and components
- Implementation: source component inventory and bill of materials (number, name, purpose, cost) of all primitive components
- Development: executable components sufficient to demonstrate the critical us cases and the risk associated with achieving the system qualities.

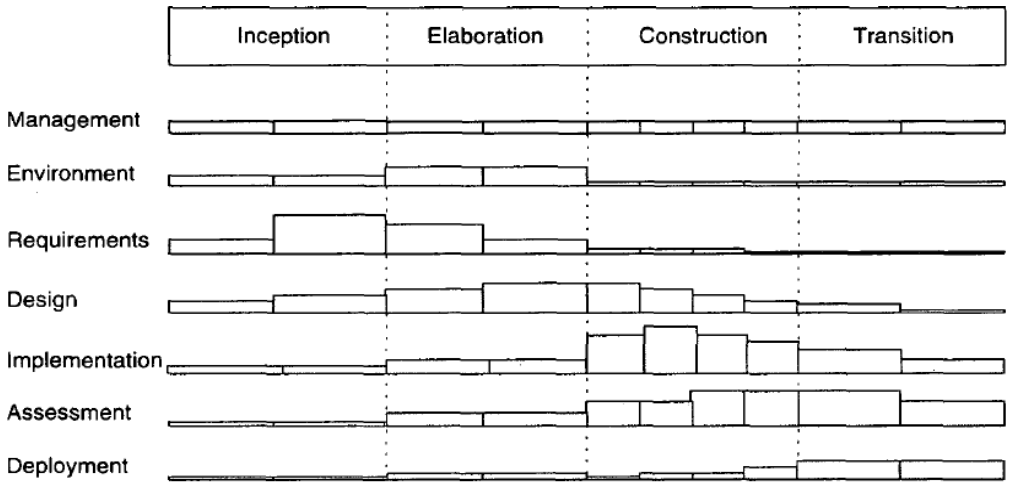
SPM UNIT IV (Part-I)

Flows of the Process: Software Process Workflows. Inter Trans Workflows. Checkpoints of the Process: Major Mile Stones, Minor Milestones, Periodic Status Assessments. Interactive Process Planning: Work Breakdown Structures, Planning Guidelines, Cost and Schedule Estimating. Interaction Planning Process. Pragmatic Planning.

SOFTWARE PROCESS WORKFLOWS

The term ***workflow*** is used to mean a thread of cohesive and mostly sequential activities; Workflows are mapped to product artifacts. There are seven top-level workflows:

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



Activity levels across the life-cycle phases

1. Architecture –first approach: Extensive requirements analysis, design, implementation and assessment activities are performed before the construction phase when full-scale implementation is the focus.

2. Iterative life-cycle process: Some projects may require only one iteration in a phase, others may require several iterations. The point is that the activities and artifacts of any given workflow may require more than one pass to achieve results

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

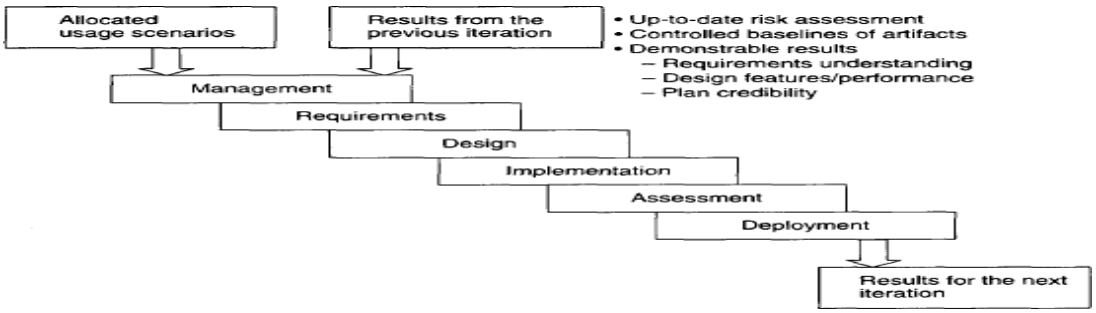
4. Demonstration-based approach: Implementation and assessment activities are initiated early in the life cycle, reflecting the emphasis on constructing executable subsets of the evolving architecture.

The artifacts and life-cycle emphases associated with each workflow

WORKFLOW	ARTIFACTS	LIFE-CYCLE PHASE EMPHASIS
Management	Business case Software development plan Status assessments Vision Work breakdown structure	Inception: Prepare business case and vision Elaboration: Plan development Construction: Monitor and control development Transition: Monitor and control deployment
Environment	Environment Software change order database	Inception: Define development environment and change management infrastructure Elaboration: Install development environment and establish change management database Construction: Maintain development environment and software change order database Transition: Transition maintenance environment and software change order database
Requirements	Requirements set Release specifications Vision	Inception: Define operational concept Elaboration: Define architecture objectives Construction: Define iteration objectives Transition: Refine release objectives
Design	Design set Architecture description	Inception: Formulate architecture concept Elaboration: Achieve architecture baseline Construction: Design components Transition: Refine architecture and components
Implementation	Implementation set Deployment set	Inception: Support architecture prototypes Elaboration: Produce architecture baseline Construction: Produce complete componentry Transition: Maintain components
Assessment	Release specifications Release descriptions User manual Deployment set	Inception: Assess plans, vision, prototypes Elaboration: Assess architecture Construction: Assess interim releases Transition: Assess product releases
Deployment	Deployment set	Inception: Analyze user community Elaboration: Define user manual Construction: Prepare transition materials Transition: Transition product to user

ITERATION WORKFLOWS

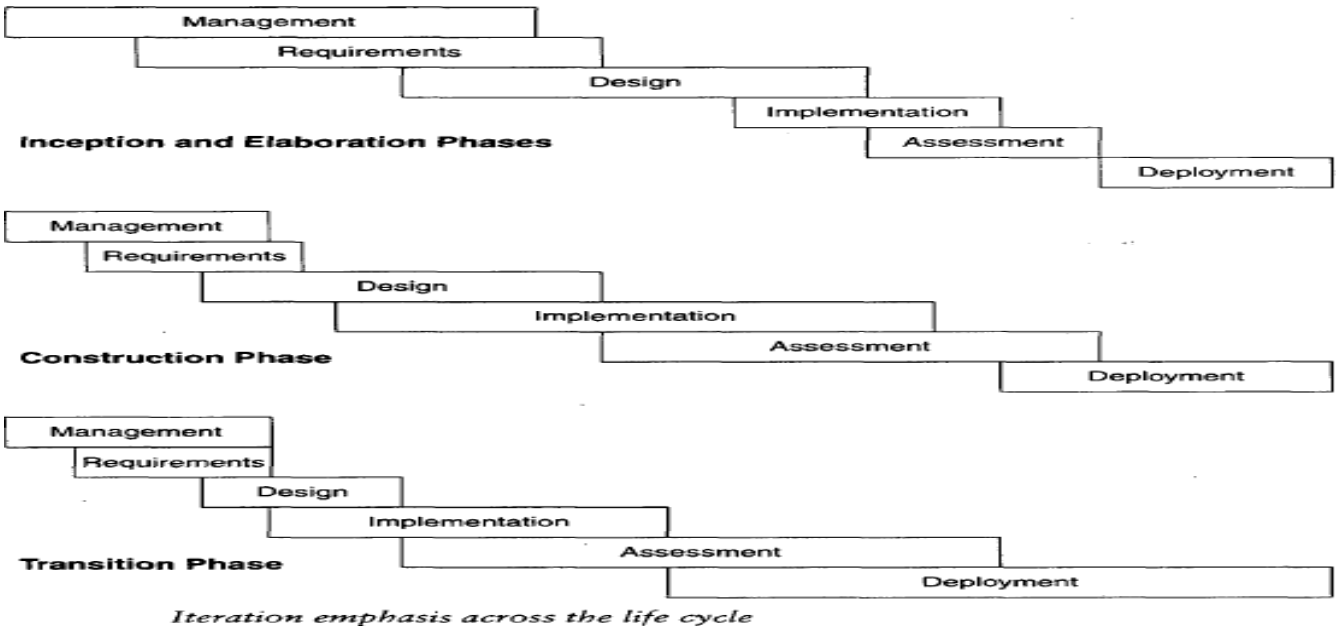
Iteration consists of a loosely sequential set of activities in various proportions, depending on where the iteration is located in the development cycle. Each iteration is defined in terms of a set of allocated usage scenarios.



The workflow of an iteration

An individual iteration's workflow generally includes the following sequence:

- **Management**: iteration planning to determine the content of the release and develop the detailed plan for the iteration; assignment of work packages, or tasks, to the development team.
- **Environment**: evolving the software change order database to reflect all new baselines and changes to existing baselines 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 demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration‟s engineering 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 against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration‟s engineering activities.
  - **Implementation:** developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions).
  - **Assessment:** evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; identifying any rework required and detem1ining whether it should be performed before deployment of this release or allocated to the next release; assessing results to improve the basis of the subsequent iteration‟s plan.
  - **Deployment:** transitioning the release either to an external organization (such as a user, independent verification and validation contractor, or regulatory agency) or to internal closure by conducting a post-mortem so that lessons learned can be captured and reflected in the next iteration.



Iteration emphasis across the life cycle

CHECKPOINTS OF THE PROCESS

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

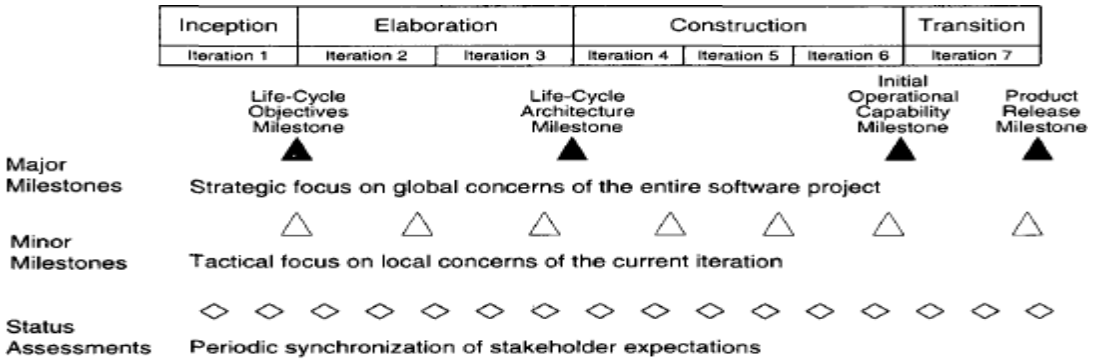
- **Major Milestones**: these system wide events are held ant the end of each development phase. They provide visibility to system wide issues synchronize the management and engineering perspectives and verify that the aims of the phase have been achieved.
- **Minor Milestones**: theses iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work.
- **Status Assessments**: These periodic events provide management with frequent and regular insight into the progress being made.

Each of the four phases-inceptions, elaboration, construction and transition consists of one or more iterations and concludes with a major milestone when a planned technical capability is produced in demonstrable form.

MAJOR MILESTONES

The four major milestones occur at the transition points between life-cycle phases. They can be used in many different process models, including eth conventional waterfall model. In an iterative model, the major milestones are used to achieve concurrence among all stakeholders on the current state of the project.

- **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, trade-off 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:** possibly many other perspectives by stakeholders such as regulatory agencies, independent verification and validation contractors, venture capital investors, subcontractors, associate contractors and sale and marketing teams.



A typical sequence of life-cycle checkpoints

The following Table summarizes the balance of information across them major milestones.

The general status of plans, requirements, and products across the major milestones

MILESTONES	PLANS	UNDERSTANDING OF PROBLEM SPACE (REQUIREMENTS)	SOLUTION SPACE PROGRESS (SOFTWARE PRODUCT)
Life-cycle objectives milestone	Definition of stakeholder responsibilities Low-fidelity life-cycle plan High-fidelity elaboration phase plan	Baseline vision, including growth vectors, quality attributes, and priorities Use case model	Demonstration of at least one feasible architecture Make/buy/reuse trade-offs Initial design model
Life-cycle architecture milestone	High-fidelity construction phase plan (bill of materials, labor allocation) Low-fidelity transition phase plan	Stable vision and use case model Evaluation criteria for construction releases, initial operational capability Draft user manual	Stable design set Make/buy/reuse decisions Critical component prototypes
Initial operational capability milestone	High-fidelity transition phase plan	Acceptance criteria for product release Releasable user manual	Stable implementation set Critical features and core capabilities Objective insight into product qualities
Product release milestone	Next-generation product plan	Final user manual	Stable deployment set Full features Compliant quality

Life-Cycle Objectives Milestone

The life-cycle objectives milestone occurs at tile 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. A successfully completed life-cycle objectives milestone will result in authorization from all stakeholders to proceed with the elaboration phase.

Life-Cycle Architecture Milestone

The life-cycle architecture milestone occurs at the end of the elaboration phase. The primary goal is to demonstrate an executable architecture to all stakeholders. The baseline architecture consists of both a human-readable representation and a configuration-controlled set of software components captured in the engineering artifacts.

I. **Requirements**
   - A. Use case model
   - B. Vision document (text, use cases)
   - C. Evaluation criteria for elaboration (text, scenarios)

II. **Architecture**
   - A. Design view (object models)
   - B. Process view (if necessary, run-time layout, executable code structure)
   - C. Component view (subsystem layout, make/buy/reuse component identification)
   - D. Deployment view (target run-time layout, target executable code structure)
   - E. Use case view (test case structure, test result expectation)
     1. Draft user manual

III. **Source and executable libraries**
   - A. Product components
   - B. Test components
   - C. Environment and tool components

Engineering artifacts available at the life-cycle architecture milestone

Presentation Agenda

I. **Scope and objectives**
   A. Demonstration overview
II. **Requirements assessment**
   A. Project vision and use cases
   B. Primary scenarios and evaluation criteria
III. **Architecture assessment**
   A. Progress
      1. Baseline architecture metrics (progress to date and baseline for measuring future architectural stability, scrap, and rework)
      2. Development metrics baseline estimate (for assessing future progress)
      3. Test metrics baseline estimate (for assessing future progress of the test team)
   B. Quality
      1. Architectural features (demonstration capability summary vs. evaluation criteria)
      2. Performance (demonstration capability summary vs. evaluation criteria)
      3. Exposed architectural risks and resolution plans
      4. Affordability and make/buy/reuse trade-offs
IV. **Construction phase plan assessment**
   A. Iteration content and use case allocation
   B. Next iteration(s) detailed plan and evaluation criteria
   C. Elaboration phase cost/schedule performance
   D. Construction phase resource plan and basis of estimate
   E. Risk assessment

Demonstration Agenda

I. **Evaluation criteria**
II. **Architecture subset summary**
III. **Demonstration environment summary**
IV. **Scripted demonstration scenarios**
V. **Evaluation criteria results and follow-up items**

Default agendas for the life-cycle architecture milestone

MINOR MILESTONES

- Minor milestones are sometimes called as inch-pebbs.
- Minor milestones mainly focus on local concerns of current iteration.
- These iterative focused events are used to review iterative content in a detailed manner & authorize continued work.

Minor Milestone in the life cycle of Iteration: The number of iteration specific milestones is dependent on the iteration length and the content. A one month to six month iterative period requires only two minor milestones

a) **Iteration Readiness review**: This informal milestone is conducted at the start of each iteration to reviewthe detailed iteration plan and evaluation criteria that have been allocated to this iteration.

b) **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 satisfied its evaluation criteria, to review iteration results.

PERIODIC STATUS ASSESSMENTS

- Periodic status assessments are management reviews conducted at regular intervals (monthly, quarterly) to address progress and quality indicators, ensure continuous attention to project dynamics, and maintain open communications among all stakeholders.
- Periodic status assessments are crucial for focusing continuous attention on the evolving health of the project and its dynamic priorities.
- Periodic status assessments serve as project snapshots. While the period may vary, the recurring event forces the project history to be captured and documented. Status assessments provide the following:
  a) A mechanismfor openly addressing, communicating and resolving management issues technical issues and project risks.
  b) Objective data derived directly from on-going activities and evolving product configurations
  c) A mechanism for disseminating process, progress, quality trends, practices, and experience information to and from all stakeholders in an open forum.

Default content of status assessment reviews

TOPIC	CONTENT
Personnel	Staffing plan vs. actuals
	Attritions, additions
Financial trends	Expenditure plan vs. actuals for the previous, current, and next major milestones
	Revenue forecasts
Top 10 risks	Issues and criticality resolution plans
	Quantification (cost, time, quality) of exposure
Technical progress	Configuration baseline schedules for major milestones
	Software management metrics and indicators
	Current change trends
	Test and quality assessments
Major milestone plans and results	Plan, schedule, and risks for the next major milestone
	Pass/fail results for all acceptance criteria
Total product scope	Total size, growth, and acceptance criteria perturbations

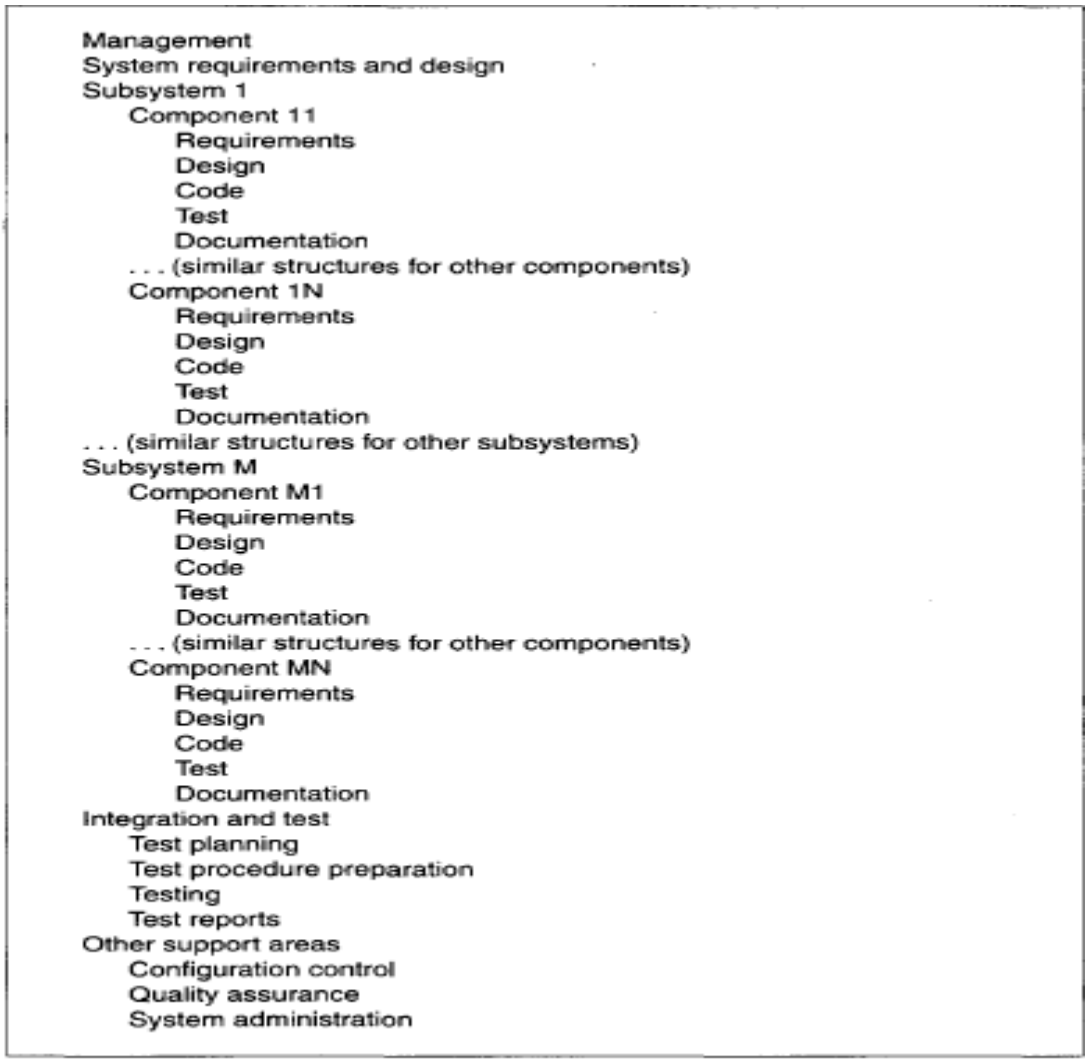
WORK BREAKDOWN STRUCTURES (WBS)

- A good work breakdown structure and its synchronization with the process framework are critical factors in software project success. Development of a work breakdown structure dependent on the project management style, organizational culture, customer preference, financial constraints and several other hard-to-define, project-specific parameters.
- A WBS is simply a hierarchy of elements that decomposes the project plan into the discrete work tasks.
- A WBS provides the following information structure:
  a) A delineation of all significant work
  b) A clear task decomposition for assignment of responsibilities
  c) A framework for scheduling, budgeting, and expenditure tracking

CONVENTIONAL WBS ISSUES

Conventional work breakdown structures frequently suffer from three fundamental flaws.

- They are prematurely structured around the product design.
- They are prematurely decomposed, planned, and budgeted in either too much or too little detail.
- They are project-specific, and cross-project comparisons are usually difficult or impossible.



Conventional work breakdown structure, following the product hierarchy

- *Conventional work breakdown structures are prematurely structured around the product design.* The above Figure shows a typical conventional WBS that has been structured primarily around the subsystems of its product architecture, and then further decomposed into the components of each subsystems. A WBS is the architecture for the financial plan.
- *Conventional work breakdown structures are prematurely decomposed, planned and budgeted in either too little or too much detail.* Large software projects tend to be over planned and small projects tend to be under planned. The basic problem with planning too much detail at the outset is that the detail does not evolve with the level of fidelity in the plan.
- *Conventional work breakdown structures are project-specific and cross-project comparisons are usually difficult or impossible*. With no standard WBS structure, it is extremely difficult to compare plans, financial data, schedule data, organizational efficiencies, cost trends, productivity trends, or quality trends across multiple projects.

EVOLUTIONARY WORK BREAKDOWN STRUCTURES

An evolutionary WBS should organize the planning elements around the process framework rather than the product framework. The basic recommendation for the WBS is to organize the hierarchy as follows:

a) First-level WBS elements are the workflows (management, environment, requirements, design, implementation, assessment, and deployment).

b) Second-level elements are defined for each phase of life cycle (inception, elaboration, construction, and transition).

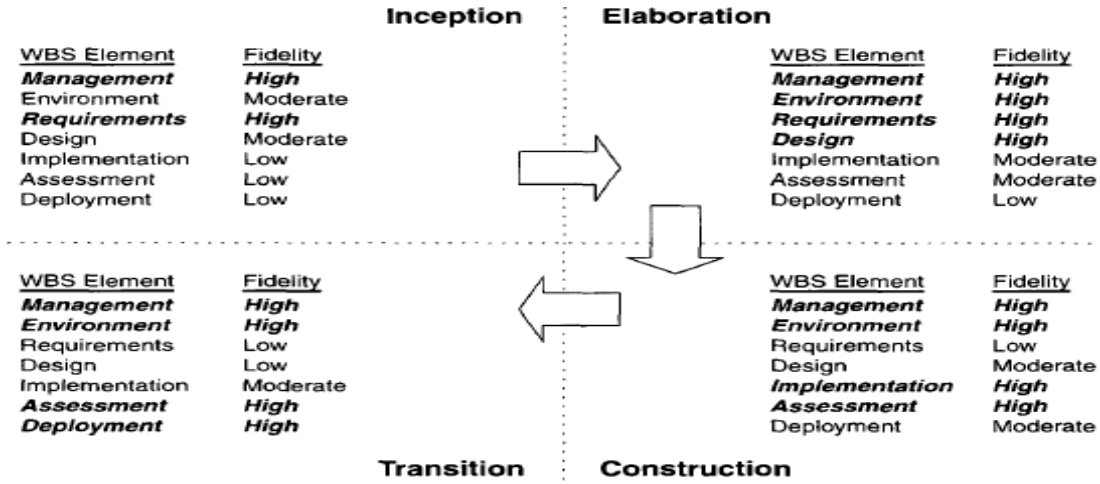
c) Third-level elements are defined for the focus of activities that produce the artifacts of each phase.

A default WBS consistent with the process framework (phases, workflows, and artifacts) is shown in Figure:

- A Management
  - AA Inception phase management
    - AAA Business case development
    - AAB Elaboration phase release specifications
    - AAC Elaboration phase WBS baselining
    - AAD Software development plan
    - AAE Inception phase project control and status assessments
  - AB Elaboration phase management
    - ABA Construction phase release specifications
    - ABB Construction phase WBS baselining
    - ABC Elaboration phase project control and status assessments
  - AC Construction phase management
    - ACA Deployment phase planning
    - ACB Deployment phase WBS baselining
    - ACC Construction phase project control and status assessments
  - AD Transition phase management
    - ADA Next generation planning
    - ADB Transition phase project control and status assessments
- B Environment
  - BA Inception phase environment specification
  - BB Elaboration phase environment baselining
    - BBA Development environment installation and administration
    - BBB Development environment integration and custom toolsmithing
    - BBC SCO database formulation
  - BC Construction phase environment maintenance
    - BCA Development environment installation and administration
    - BCB SCO database maintenance
  - BD Transition phase environment maintenance
    - BDA Development environment maintenance and administration
    - BDB SCO database maintenance
    - BDC Maintenance environment packaging and transition
- C Requirements
  - CA Inception phase requirements development
    - CAA Vision specification
    - CAB Use case modeling
  - CB Elaboration phase requirements baselining
    - CBA Vision baselining
    - CBB Use case model baselining
  - CC Construction phase requirements maintenance
  - CD Transition phase requirements maintenance
- D Design
  - DA Inception phase architecture prototyping
  - DB Elaboration phase architecture baselining
    - DBA Architecture design modeling
    - DBB Design demonstration planning and conduct
    - DBC Software architecture description
  - DC Construction phase design modeling
    - DCA Architecture design model maintenance
    - DCB Component design modeling
  - DD Transition phase design maintenance
- E Implementation
  - EA Inception phase component prototyping
  - EB Elaboration phase component implementation
    - EBA Critical component coding demonstration integration
  - EC Construction phase component implementation
    - ECA Initial release(s) component coding and stand-alone testing
    - ECB Alpha release component coding and stand-alone testing
    - ECC Beta release component coding and stand-alone testing
    - ECD Component maintenance
  - ED Transition phase component maintenance
- F Assessment
  - FA Inception phase assessment planning
  - FB Elaboration phase assessment
    - FBA Test modeling
    - FBB Architecture test scenario implementation
    - FBC Demonstration assessment and release descriptions
  - FC Construction phase assessment
    - FCA Initial release assessment and release description
    - FCB Alpha release assessment and release description
    - FCC Beta release assessment and release description
  - FD Transition phase assessment
    - FDA Product release assessment and release descriptions
- G Deployment
  - GA Inception phase deployment planning
  - GB Elaboration phase deployment planning
  - GC Construction phase deployment
    - GCA User manual baselining
  - GD Transition phase deployment
    - GDA Product transition to user

PLANNING GUIDELINES

Software projects span a broad range of application domains. It is valuable but risky to make specific planning recommendations independent of project context. Project-independent planning advice is also risky. There is the risk that the guidelines may be adopted blindly without being adapted to specific project circumstance. Two simple planning guidelines should be considered when a project plan is being initiated or assessed.



Evolution of planning fidelity in the WBS over the life cycle

The below table prescribes a default allocation of costs among the first-level WBS elements.

WBS budgeting defaults

FIRST-LEVEL WBS ELEMENT	DEFAULT BUDGET
Management	10%
Environment	10%
Requirements	10%
Design	15%
Implementation	25%
Assessment	25%
Deployment	5%
Total	100%

The below table prescribes allocation of effort and schedule across the lifecycle phases.

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. The first is a ***forward-looking***, top-down approach. It starts with an understanding of the general 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 sequence would occur:

- The software project manager (and others) develops a characterization of the overall size, process, environment, people, and quality required for the project.
- A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.
- The software project manager partitions the estimate for the effort into top-level WBS using guidelines.

At this point, subproject managers are given the responsibility for decomposing each of the WBS elements into lower levels using their top-level allocation, staffing profile, and major milestone dates as constraints.

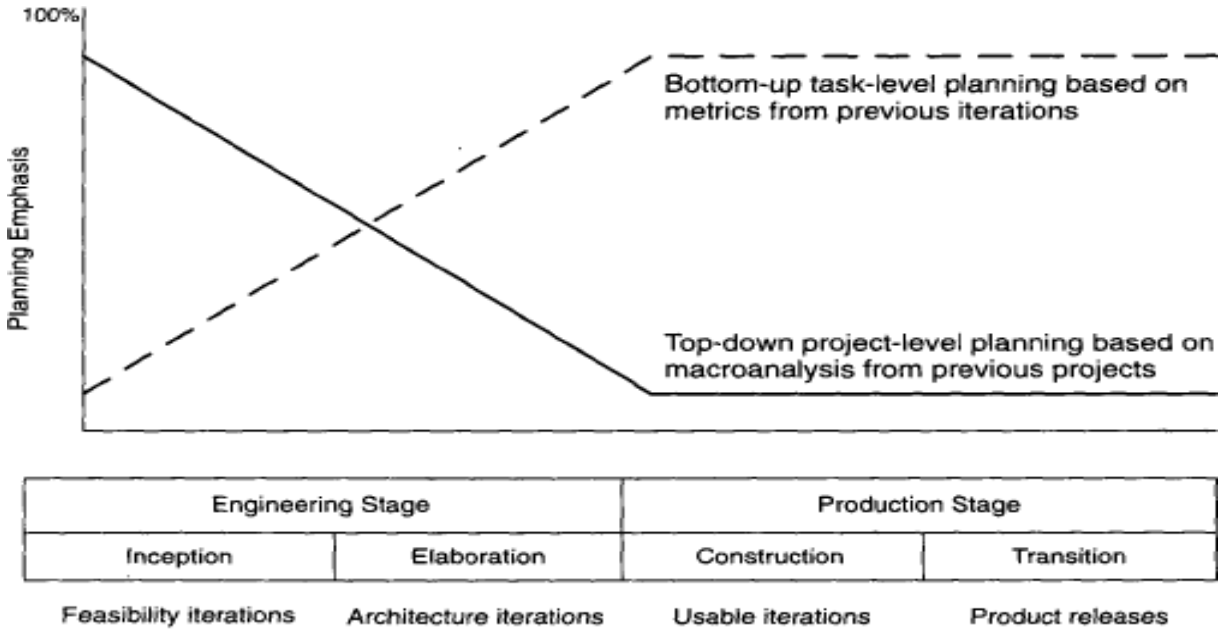
The second perspective is a ***backward-looking***, bottom-up approach. We start with the end in mind, analyze the micro-level budgets and schedules, and then sum all these elements into the higher level budgets and intermediate milestones. This approach tends to define and populate the WBS from the lowest levels upward. From this perspective, the following planning sequence would occur:

a) The lowest level WBS elements are elaborated into detailed tasks

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

c) Comparisons are made with the top-down budgets and schedule milestones.

During the engineering stage top down approach dominates bottom up approach. During the production stage bottom approach dominates top down approach.



Engineering stage planning emphasis:

- Macro-level task estimation for production-stage artifacts
- Micro-level task estimation for engineering artifacts
- Stakeholder concurrence
- Coarse-grained variance analysis of actual vs. planned expenditures
- Tuning the top-down project-independent planning guidelines into project-specific planning guidelines
- WBS definition and elaboration

Production stage planning emphasis:

- Micro-level task estimation for production-stage artifacts
- Macro-level task estimation for maintenance of engineering artifacts
- Stakeholder concurrence
- Fine-grained variance analysis of actual vs. planned expenditures

Planning balance throughout the life cycle

THE ITERATION PLANNING PROCESS

- Planning is concerned with defining the actual sequence of intermediate results.
- Iteration is used to mean a complete synchronization across the project, with a well-orchestrated global assessment of the entire project baseline.
  Inception Iterations: the early prototyping activities integrate the foundation components of candidate architecture and provide an executable framework for elaborating the critical use cases of eth system.
  Elaboration Iteration: These iterations result in architecture, including a complete framework and infrastructure for execution. Upon completion of the architecture iteration, a few critical use cases should be demonstrable: (1) initializing the architecture (2) injecting a scenario to drive the worst-case data processing flow through the system (3) injecting a scenario to drive the worst-case control flow through the system (for example, orchestrating the fault-tolerance use cases).
  Construction Iterations: Most projects require at least two major construction iterations: an alpha release and a beta release.
  Transition Iterations: Most projects use a single iteration to transition a beta release into the final product.
- The general guideline is that most projects will use between four and nine iteration. The typical project would have the following six-iteration profile:
  - **One iteration in inception:** an architecture prototype
  - **Two iterations in elaboration:** architecture prototype and architecture baseline
  - **Two iterations in construction:** alpha and beta releases
  - **One iteration in transition:** product release

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. Aside form bad architectures and misunderstood requirement, inadequate planning (and subsequent bad management) is one of the most common reasons for project failures. Conversely, the success of every successful project can be attributed in part to good planning.

A project‟s plan is a definition of how the project requirements will be transformed into a product within the business constraints. It must be realistic, it must be current, it must be a team product, it must be understood by the stake holders, and it must be used. Plans are not just for mangers. The more open and visible the planning process and results, the more ownership there is among the team members who need to execute it. Bad, closely held plans cause attrition. Good, open plans can shape cultures and encourage teamwork.

SPM UNIT IV (Part-II)

Project Organizations and Responsibilities: Line-of-Business Organizations, Project Organizations, and Evolution of Organizations. Process Automation: Automation Building Blocks, the Project Environment.

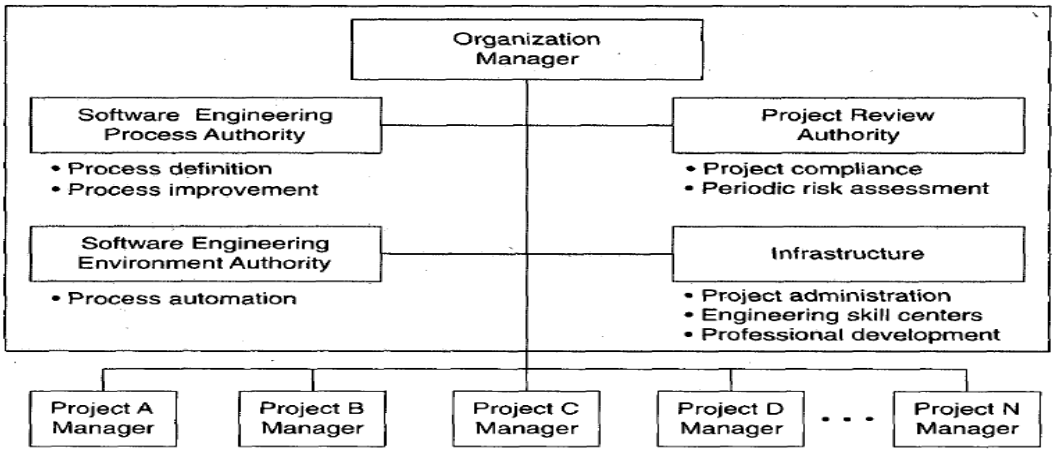
Software lines of business and project teams have different motivations. Software lines of business are motivated by return on investment, new business discriminators, market diversification and profitability.

Software professionals in both types of organizations are motivated by career growth, job satisfaction and the opportunity to make a difference.

LINE -OF-BUSINESS ORGANIZATIONS

The main features of the default organization are as follows:

- Responsibility for process definition and maintenance is specific to a cohesive line of business.
- Responsibility for process automation is an organizational role and is equal in importance to the process definition role.
- Organization roles may be fulfilled by a single individual or several different teams, depending on the scale of the organization.



Default roles in a software line-of-business organization

The line of business organization consists of 4 component teams.

Software Engineering Process Authority (SEPA):

- Responsible for exchanging the information and project guidance to or from the project practitioners.
- Maintains current assessment of organization process maturity.
- Help in initiate and periodically assess project processes.
- Responsible for process definition and maintainence.

Project Review Authority (PRA):

- Responsible for reviewing the financial performance, customer commitments, risks & accomplishments, adherence to organizational policies by customer etc.
- Reviews both project‟s conformance, customer commitments as well as organizational polices, deliverables, financial performances and other risks.

Software Engineering Environment Authority (SEEA):

- SEEA deals with the maintenance or organizations standard environment, training projects and process automation.
- Maintains organization‟s standard environment.
- Training projects to use environment.

- ❖ Maintain organization wide resources support.

Infrastructure:

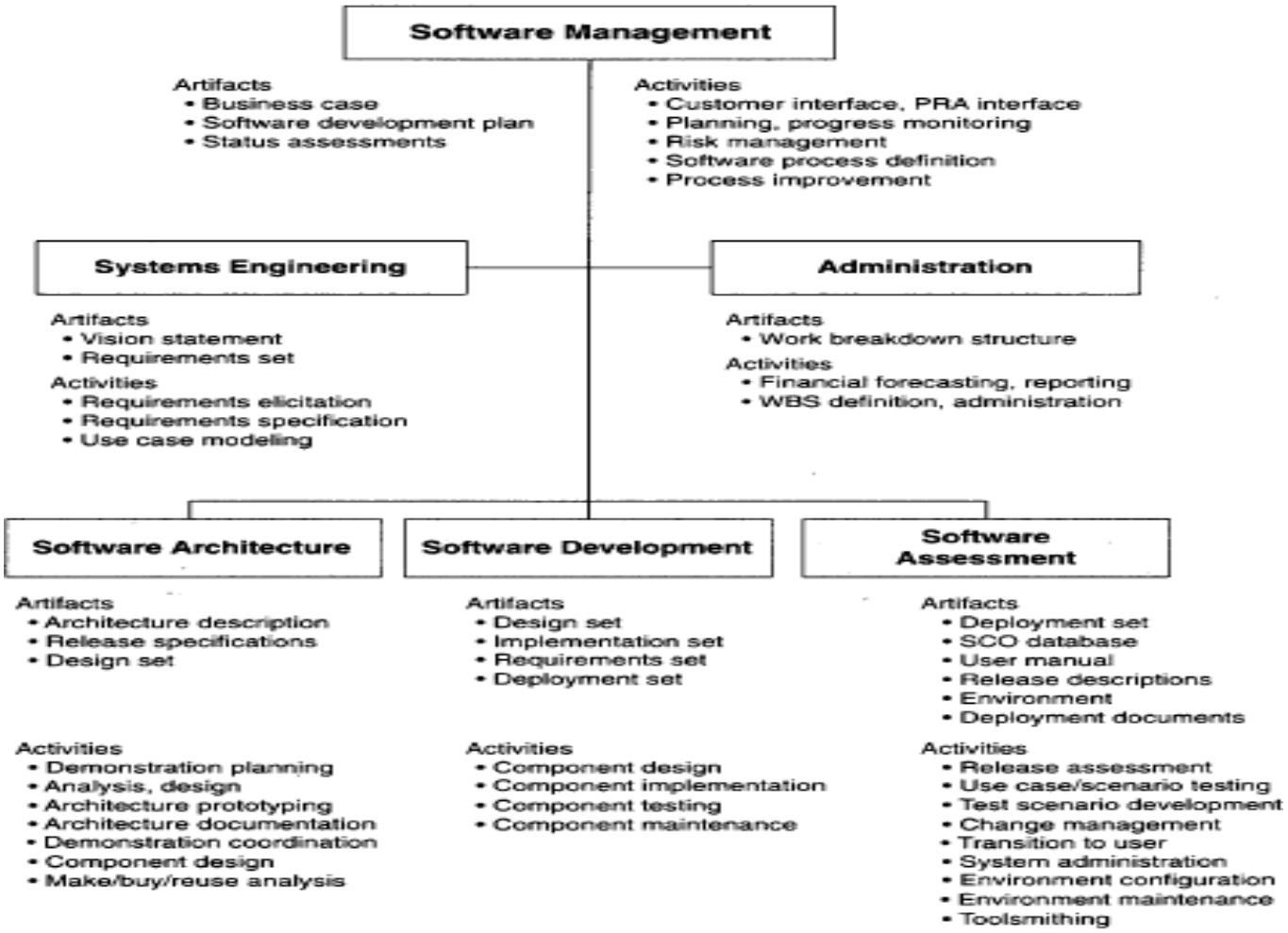
- ❖ An organization‟s infrastructure provides human resources support, project-independent research and development other capital software engineering assets. The typical components of the organizational infrastructure are as follows:
  - ❑ **Project Administration:** Time accounting system; contracts, pricing, terms and conditions; corporate information systems integration.
  - ❑ **Engineering Skill Centers:** Custom tools repository and maintenance, bid and proposal support, independent research and development.
  - ❑ **Professional Development:** Internal training boot camp, personnel recruiting, personnel skills database maintenance, literature and assets library, technical publications.

PROJECT ORGANIZATIONS

The default project organization and maps project-level roles and responsibilities. This structure can be tailored to the size and circumstance of the specific project organization.

The main feature of the default organization is as follows:

- ❑ *The project management team* is an active participant, responsible for producing as well as managing. Project management is not a spectator sport.
- ❑ *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.
- ❑ *Quality is every one job. Each team takes responsibility for a different quality perspective.*



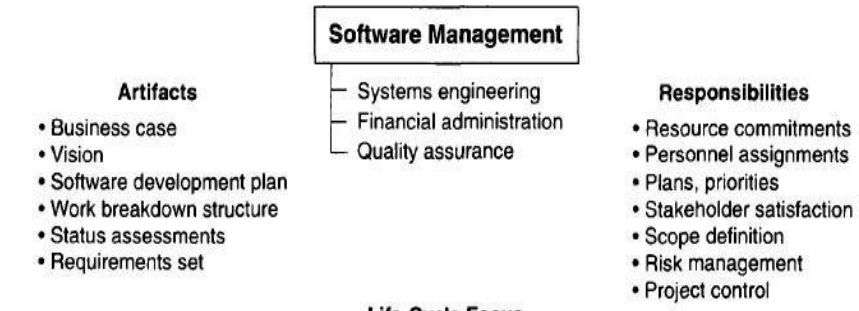
Default project organization and responsibilities

Software Management Team:

- This is active participant in an organization and is incharge of producing as well as managing.
- As the software attributes, such as Schedules, costs, functionality and quality are interrelated to each other, negotiation among multiple stakeholders is required and these are carried out by the software management team.

Responsibilities:

- Effort planning
- Conducting the plan
- Adapting the plan according to the changes in requirements and design
- Resource management
- Stakeholders satisfaction
- Risk management
- Assignment or personnel
- Project controls and scope definition
- Quality assurance



Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Elaboration phase planning Team formulation Contract baselining Architecture costs	Construction phase planning Full staff recruitment Risk resolution Product acceptance criteria Construction costs	Transition phase planning Construction plan optimization Risk management	Customer satisfaction Contract closure Sales support Next-generation planning

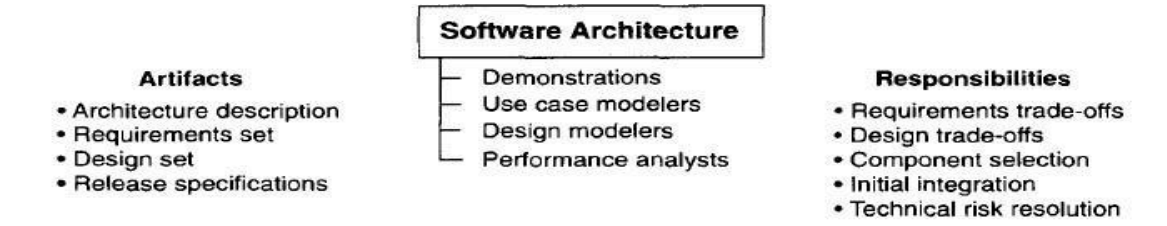
Software management team activities

Software Architecture Team:

- The software architecture team performs the tasks of integrating the components, creating real artifacts etc.
- It promotes team communications and implements the applications with a system-wide quality.
- The success of the development team is depends on the effectiveness of the architecture team along with the software management team controls the inception and elaboration phases of a life-cycle.
- The architecture team must have:
  - Domain experience to generate an acceptable design and use-caseview.
  - Software technology experience to generate an acceptable process view, component and development views.

Responsibilities:

- System-level quality i.e., performance, reliability and maintainability.
- Requirements and design trade-offs.
- Component selection
- Technical risk solution
- Initial integration



Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Architecture prototyping Make/buy trade-offs Primary scenario definition Architecture evaluation criteria definition	Architecture baselining Primary scenario demonstration Make/buy trade-off baselining	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements

Software architecture team activities

Software Development Team:

- The Development team is involved in the construction and maintenance activities. It is most applicationspecific team. It consists of several sub teams assigned to the groups of components requiring a common skill set.
- The skill set include the following:
- ***Commercial component***: specialists with detailed knowledge of commercial components central to a system's architecture.
- ***Database***: specialists with experience in the organization, storage, and retrieval of data.
- ***Graphical user interfaces***: specialists with experience in the display organization; data presentation, and user interaction.
- ***Operating systems and networking***: specialists with experience in various control issues arises due to synchronization, resource sharing, reconfiguration, inter object communications, name space management etc.
- ***Domain applications***: Specialists with experience in the algorithms, application processing, or business rules specific to the system.

Responsibilities:

- The exposure of the quality issues that affect the customer‟s expectations.
- Metric analysis.
- Verifying the requirements.
- Independent testing.
- Configuration control and user development.
- Building project infrastructure.

Software Development

└ Component teams

Artifacts

- Design set
- Implementation set
- Deployment set

Responsibilities

- Component design
- Component implementation
- Component stand-alone test
- Component maintenance
- Component documentation

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Prototyping support Make/buy trade-offs	Critical component design Critical component implementation and test Critical component baseline	Component design Component implementation Component stand-alone test Component maintenance	Component maintenance Component documentation

Software development team activities

Software Assessment Team:

- This team is involved in testing and product activities in parallel with the ongoingdevelopment.
- It is an independent team for utilizing the concurrency of activities.
- The use-case oriented and capability-based testing of a process is done by using two artifacts:
  - Release specification (the plan and evaluation criteria for a release)
  - Release description (the results of a release)

  Responsibilities:
  - The exposure of the quality issues that affect the customer‟s expectations.
  - Metric analysis.
  - Verifying the requirements.
  - Independent testing.
  - Configuration control and user development.
  - Building project infrastructure.

Software Assessment

- Release testing
- Change management
- Deployment
- Environment support

Artifacts

- Deployment set
- SCO database
- User manual
- Environment
- Release specifications
- Release descriptions
- Deployment documents

Responsibilities

- Project infrastructure
- Independent testing
- Requirements verification
- Metrics analysis
- Configuration control
- Change management
- User deployment

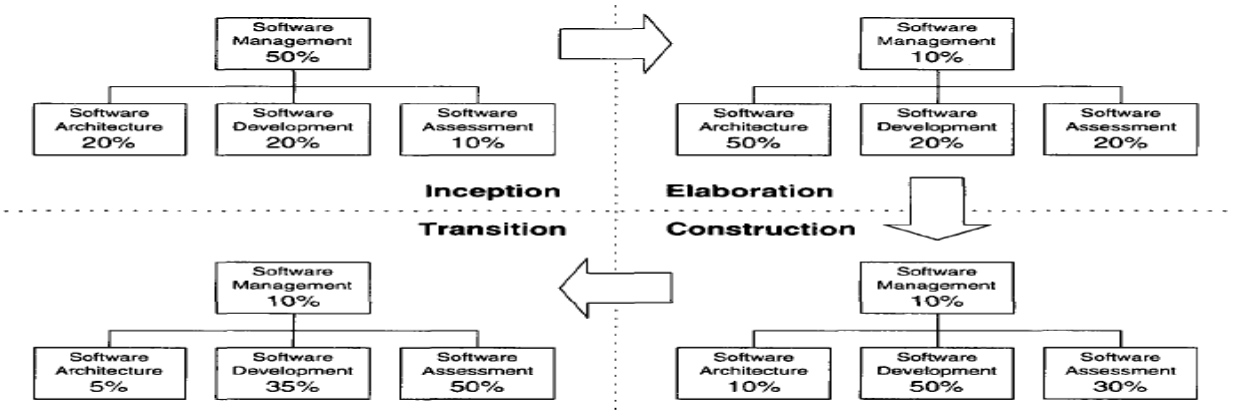
Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Infrastructure planning Primary scenario prototyping	Infrastructure baseline Architecture release testing Change management Initial user manual	Infrastructure upgrades Release testing Change management User manual baseline Requirements verification	Infrastructure maintenance Release baselining Change management Deployment to users Requirements verification

Software assessment team activities

EVOLUTION OF ORGANIZATIONS

- The project organization represents the architecture of the teamand needs to evolve consistent with the project plan captured in the work breakdown structure.
- A different set of activities is emphasized in each phase, as follows:
- **Inception team:** An organization focused on planning, with enough support from the other teams to ensure that the plans represent a consensus of all perspectives.
- **Elaboration team:** An architecture-focused organization in which the driving forces of the project reside in the software architecture team and are supported, by the software development and software assessment teams as necessary to achieve a stable architecture baseline.
- **Construction team:** A fairly balanced organization in which most of the activity resides in the software development and software assessment teams.
- **Transition team:** A customer-focused organization in which usage feedback drives the deployment activities



Software project team evolution over the life cycle

PROCESS AUTOMATION

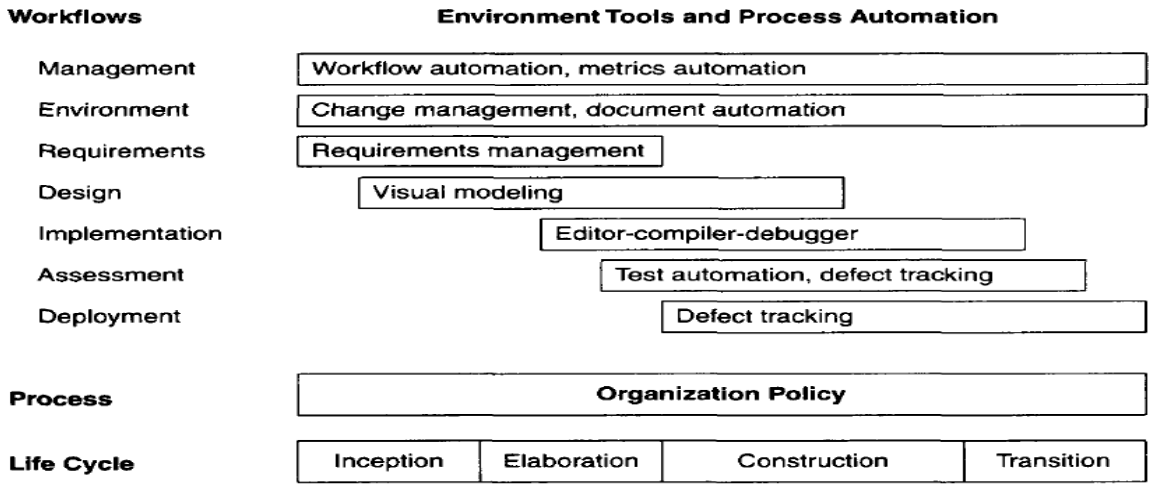
There are 3 levels of process:

1. Metaprocess: An organization‟s policies, procedures, and practices for managing a software intensive line of business.The automation support for this level is called an ***infrastructure***. An infrastructure is an inventory of preferred tools, artifact templates, microprocess guidelines, macroprocess guidelines, project performance repository, database of organizational skill sets, and library of precedent examples of past project plans and results.

2. Macroprocess: A project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The automation support for a project's process is called an ***environment.*** An environment is a specific collection of tools to produce a specific set of artifacts as governed by a specific project plan.

3. Microprocess: A project team's policies, procedures, and practices for achieving an artifact of the software process. The automation support for generating an artifact is generally called a ***tool***. Typical tools include requirements management, visual modeling, compilers, editors, debuggers, change management, metrics automation, document automation, test automation, cost estimation, and workflow automation.

Automation Buliding Blocks



Typical automation and tool components that support the process workflows

Management: Software cost estimation tools and WBS tools are useful for generating the planning artifacts. For managing against a plan, workflow management tools and a software project control panel that can maintain an on-line version of the status assessment are advantageous.

Environment: Configuration management and version control are essential in a modern iterative development process. (change management automation that must be supported by the environment.

Requirements: Conventional approaches decomposed system requirements into subsystem requirements, subsystem requirements into component requirements, and component requirements into unit requirements.

The ramifications of this approach on the environment‟s support for requirements management are twofold:

1. The recommended requirements approach is dependent on both textual and model-based representations
2. Traceability between requirements and other artifacts needs to be automated.

Design: The primary support required for the design workflow is visual modeling, which is used for capturing design models, presenting them in human-readable format, and translating them into source code. Architecture-first and demonstration-based process is enabled by existing architecture components and middleware.

Implementation: The implementation workflow relies primarily on a programming environment (editor, compiler, debugger, and linker, run time) but must also include substantial integration with the change management tools, visual modeling tools, and test automation tools to support productive iteration.

Assessment and Deployment: To increase change freedom, testing and document production must be mostly automated. Defect tracking is another important tool that supports assessment: It provides the change management instrumentation necessary to automate metrics and control release baselines. It is also needed to support the deployment workflow throughout the life cycle.

THE PROJECT ENVIRONMENT

The project environment artifacts evolve through three discrete states:

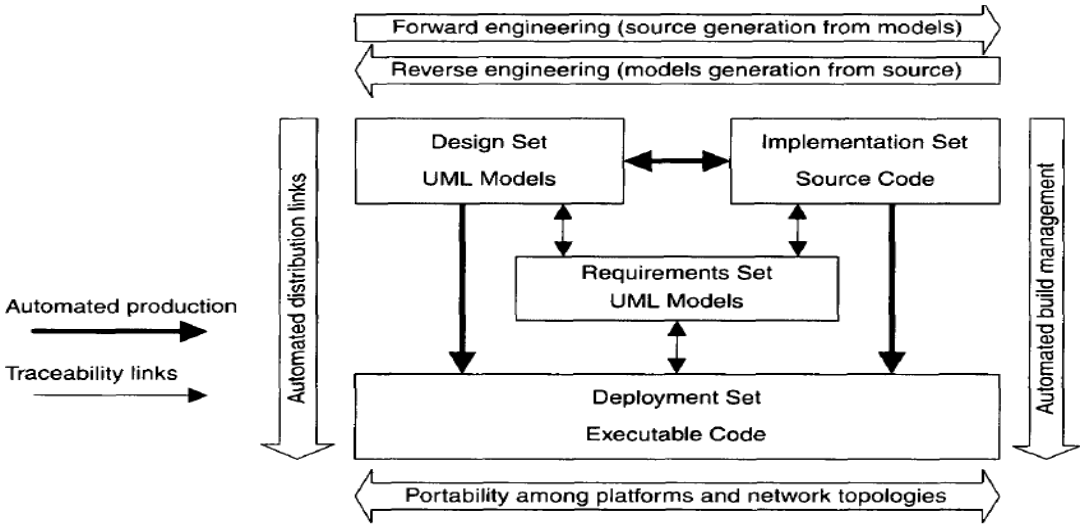
1. The proto typing environment includes an architecture tested for prototyping project architectures to evaluate trade-offs during the inception and elaboration phases of the life cycle. It should be capable of supporting the following activities:
   - technical risk analyses
   - feasibility studies for commercial products
   - Fault tolerance/dynamic reconfiguration trade-offs
   - Analysis of the risks associated implementation
   - Development of test scenarios, tools, and instrumentation suitable for analyzing the requirements.
2. The development environment should include a full suite of development tools needed to support the various process workflows and to support round-trip engineering to the maximum extent possible.
3. The maintenance environment may be a subset of the development environment delivered as one of the project's end products.

Four important environment disciplines that is critical to the management context and the success of a modern iterative development process:

- Tools must be integrated to maintain consistency and traceability. Roundtrip Engineering is the term used to describe this key requirement for environments that support iterative development.
- Change management must be automated and enforced to manage multiple, iterations and to enable change freedom. Change is the fundamental primitive of iterative development.
- Organizational infrastructures A common infrastructure promotes interproject consistency, reuse of training, reuse of lessons learned, and other strategic improvements to the organization's metaprocess.
- Extending automation support for stakeholder environments enables further support for paperless exchange of information and more effective review of engineering artifacts.

Round-Trip Engineering

- Round-trip engineering is the environment support necessary to maintain consistency among theengineering artifacts.
- The primary reason for round-trip engineering is to allow freedom in changing software engineering data sources.



Round-trip engineering

Change Management

- Change management is as critical to iterative processes as planning.
- Tracking changes in the technical artifacts is crucial to understanding the true technical progress trends and quality trends toward delivering an acceptable end product or interim release.
- In a modern process-in which requirements, design, and implementation set artifacts are captured in rigorous notations early in the life cycle and are evolved through multiple generations-change management has become fundamental to all phases and almost all activities.

Software Change Orders (SCO)

- The atomic unit of software work that is authorized to create, modify, or obsolesce components within a configuration baseline is called a software change order (SCO).
- Software change orders are a key mechanism for partitioning, allocating, and scheduling software work against an established software baseline and for assessing progress and quality.

The basic fields of the SCO are title, description, metrics, resolution, assessment and disposition.

a) Title. The title is suggested by the originator and is finalized upon acceptance by the configuration control board.

b) Description: The problem description includes the name of the originator, date of origination, CCB-assigned SCO identifier, and relevant version identifiers of related support software.

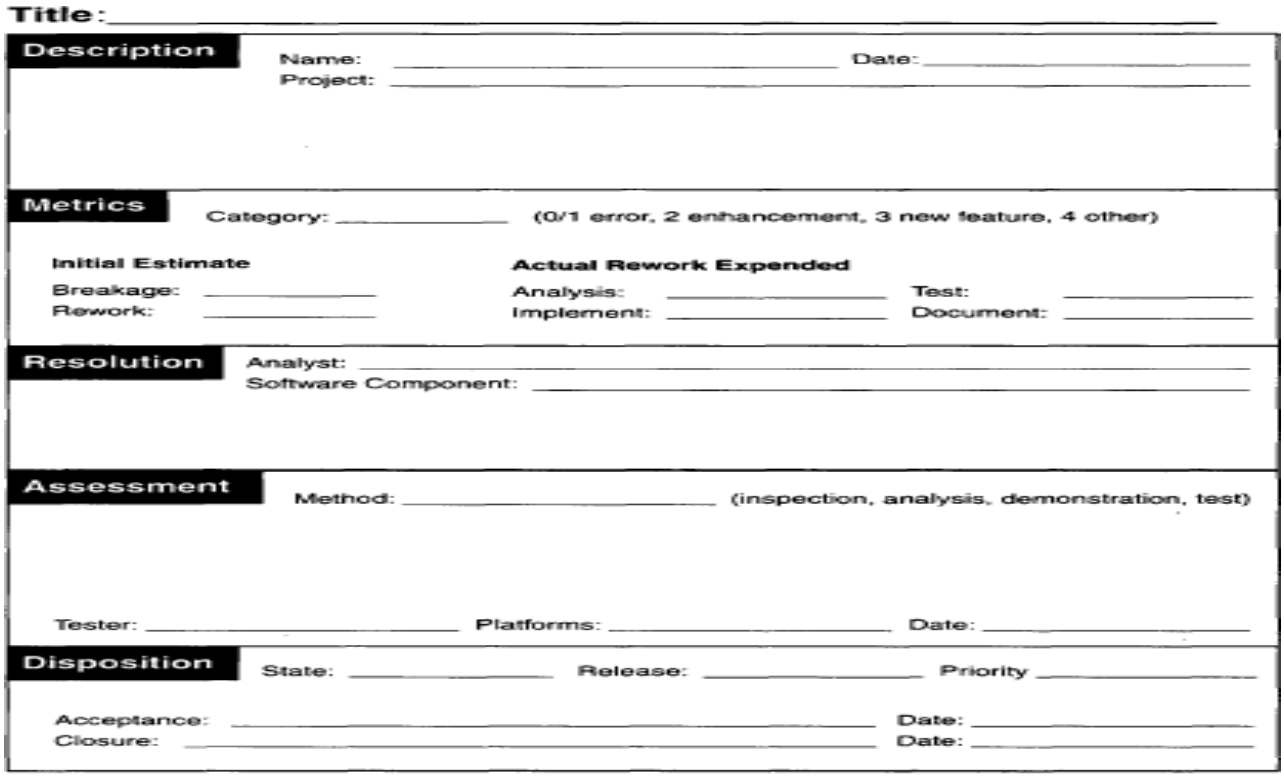
c) Metrics: The metrics collected for each sea are important for planning, for scheduling, and for assessing quality improvement. Change categories are type 0 (critical bug), type 1 (bug), type 2 (enhancement), type 3 (new feature), and type 4 (other)

d) Resolution: This field includes the name of the person responsible for implementing the change, the components changed, the actual metrics, and a description of the change.

e) Assessment: This field describes the assessment technique as either inspection, analysis, demonstration, or test. Where applicable, it should also reference all existing test cases and new test cases executed, and it should identify all different test configurations, such as platforms, topologies, and compilers.

f) Disposition: The SCO is assigned one of the following states by the CCB:

- **Proposed:** written, pending CCB review
- **Accepted:** CCB-approved for resolution
- **Rejected:** closed, with rationale, such as not a problem, duplicate, obsolete change, resolved by another SCO
- **Archived:** accepted but postponed until a later release
- **In progress:** assigned and actively being resolved by the development organization
- **In assessment:** resolved by the development organization; being assessed by a test organization
- **Closed:** completely resolved, with the concurrence of all CCB members.



The primitive components of a software change order

Configuration Baseline

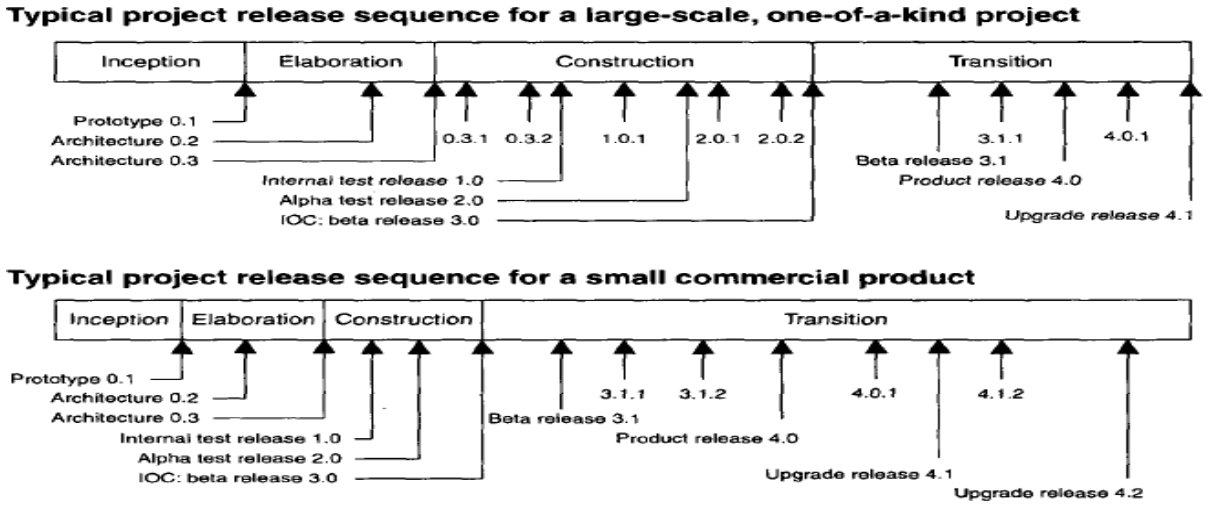
A configuration baseline is a named collection of software components and supporting documentation that is subject to change management and is upgraded, maintained, tested, statused and obsolesced as a unit.

There are general1y two classes of baselines:

1. external product releases and
2. internal testing releases.

A configuration baseline is a named collection of components that is treated as a unit. It is controlled formally because it is a packaged exchange between groups. A project may release a configuration baseline to the user community for beta testing. Once software is placed in a controlled baseline, all changes are tracked. A distinction must be made for the cause of a change. Change categories are as follows:

- Type 0: Critical failures, which are defects that are nearly always fixed before any external release.
- Type 1: A bug or defect that either does not impair 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 an update to the requirements.
- Type 4: changes that are not accommodated by the other categories.



Example release histories for a typical project and a typical product

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 usually includes the software manager, software architecture manager, software development manager, software assessment manager and other stakeholders (customer, software project manager, systems engineer, user) who are integral to the maintenance of a controlled software delivery system.
- The [bracketed] words constitute the state of an SCO transitioning through theprocess.
- [Proposed]: A proposed change is drafted and submitted to the CCB. The proposed change must includea technical description of the problem and an estimate of the resolution effort.
- [Accepted, archived or rejected]: The CCB assigns a unique identifier and accepts, archives, or rejects each proposed change. Acceptance includes the change for resolution in the next release; archiving accepts the change but postpones it for resolution in a future release; and rejection judges the change to be without merit, redundant with other proposed changes, or out of scope.
- [In progress]: the responsible person analyzes, implements and tests a solution to satisfy the SCQ. This task includes updating documentation, release notes and SCO metrics actuals and submitting new SCOs.
- [In assessment]: The independent test assesses whether the SCO is completely resolved. When the independent test team deems the change to be satisfactorily resolved, the SCO is submitted to the CCB for final disposition and closure.
- [Closed]: when the development organization, independent test organization and CCB concur that the SCO is resolved, it is transitioned to a closed status. „

Infrastructures

Organization‟s infrastructure provides the organization capital assets, including two key artifacts:
a) a policy that captures the standards for project software development processes, and
b) an environment that captures an inventory of tools.

Organization Policy

- The organization policy is usually packaged as a handbook that defines the life cycle and the process primitives (major milestones, intermediate artifacts, engineering repositories, metrics, roles and responsibilities). The handbook provides a general framework for answering the following questions:
  - What gets done? (activities and artifacts)
  - When does it get done? (mapping to the life-cycle phases and milestones)
  - Who does it? (team roles and responsibilities)
- How do we know that it is adequate? (Checkpoints, metrics and standards of performance).

I. **Process-primitive definitions**
   A. Life-cycle phases (inception, elaboration, construction, transition)
   B. Checkpoints (major milestones, minor milestones, status assessments)
   C. Artifacts (requirements, design, implementation, deployment, management sets)
   D. Roles and responsibilities (PRA, SEPA, SEEA, project teams)

II. **Organizational software policies**
   A. Work breakdown structure
   B. Software development plan
   C. Baseline change management
   D. Software metrics
   E. Development environment
   F. Evaluation criteria and acceptance criteria
   G. Risk management
   H. Testing and assessment

III. **Waiver policy**

IV. **Appendixes**
   A. Current process assessment
   B. Software process improvement plan

Organization policy outline

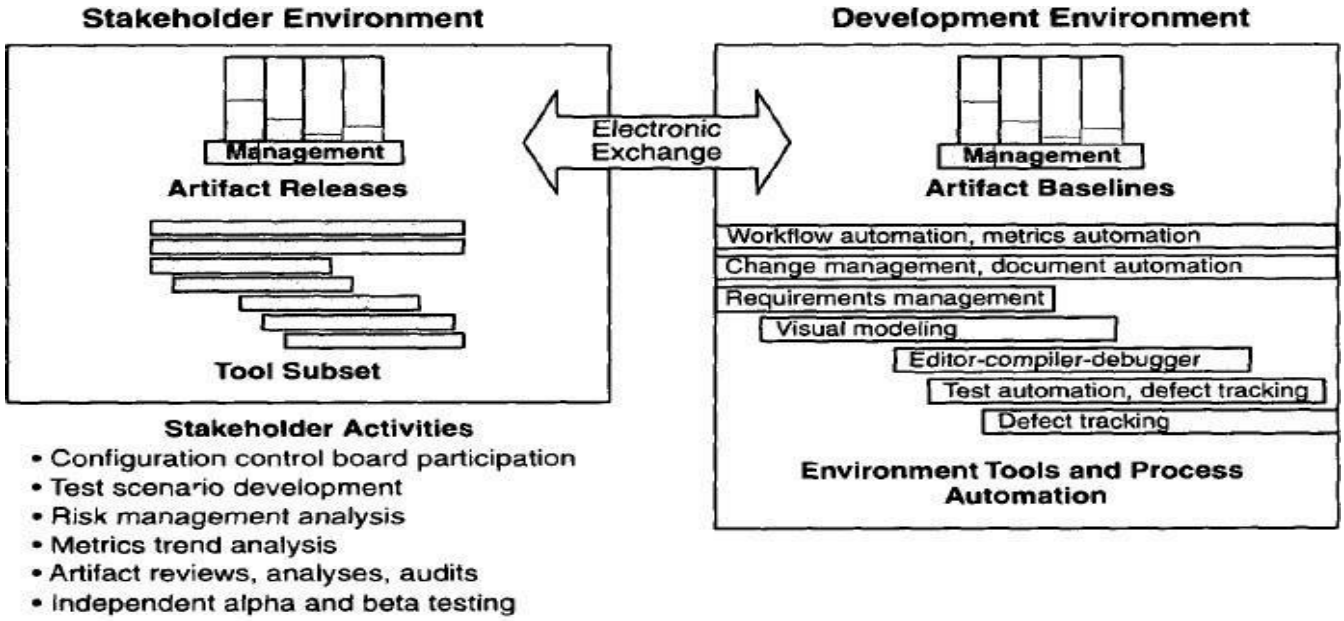
Organization Environment

Some of the typical components of an organization‟s automation building blocks are as follows:

- Standardized tool selections, which promote common workflows and a higher ROI on training.
- Standard notations for artifacts, such as UML for all design models, or Ada 95 for all custom-developed, reliability-critical implementation artifacts.
- Tool adjuncts such as existing artifact templates (architecture description, evaluation criteria, release descriptions, status assessment) or customizations.
- Activity templates (iteration planning, major milestone activities, configuration control boards).

Stakeholder Environments

- An on-line environment accessible by the external stakeholders allows them to participate in the process as follows:
  - Accept and use executable increments for hands-on evaluation.
  - Use the same on-line tools, data and reports that the software development organization uses to manageand monitor the project.
  - Avoid excessive travel, paper interchange delays, format translations, paper and shipping costs and other overhead costs.
- There are several important reasons for extending development environment resources into certainstakeholder domains.
  - Technical artifacts are not just paper.
  - Reviews and inspections, breakage/rework assessments, metrics analyses and even beta testing canbe performed independently of the development team.
  - Even paper documents should be delivered electronically to reduce production costs and turn around time.



Extending environments into stakeholder domains

SPM UNIT V(Part-I)

Project Control and Process Instrumention: Seven Core Metrics, Management Indicators, Quality Indicators, Life Cycle Expectations Pragmatic Software Metrics, Metrics Automation. Tailoring the process: Process Discriminates.

The primary themes of a modern software development process tackle the central management issues of complex software:

- Getting the design right by focusing on the architecture first
- Managing risk through iterative development
- Reducing the complexity with component based techniques
- Making software progress and quality tangible through instrumented change management
- Automating the overhead and bookkeeping activities through the use of round-trip engineering and integrated environments

The goals of software metrics are to provide the development team and the management team with the following:

- An accurate assessment of progress to date
- Insight into the quality of the evolving software product
- A basis for estimating the cost and schedule for completing the product with increasing accuracy over time.

THE SEVEN CORE METRICS

Seven core metrics are used in all software projects. Three are management indicators and four are quality indicators.

a) Management Indicators

- Work and progress (work performed over time)
- Budgeted cost and expenditures (cost incurred over time)
- Staffing and team dynamics (personnel changes over time)

b) Quality Indicators

- Change traffic and stability (change traffic over time)
- Breakage and modularity (average breakage per change over time)
- Rework and adaptability (average rework per change overtime)
- Mean time between failures (MTBF) and maturity (defect rate over time)

Overview of the seven core metrics

METRIC	PURPOSE	PERSPECTIVES
Work and progress	Iteration planning, plan vs. actuals, management indicator	SLOC, function points, object points, scenarios, test cases, SCOs
Budgeted cost and expenditures	Financial insight, plan vs. actuals, management indicator	Cost per month, full-time staff per month, percentage of budget expended
Staffing and team dynamics	Resource plan vs. actuals, hiring rate, attrition rate	People per month added, people per month leaving
Change traffic and stability	Iteration planning, management indicator of schedule convergence	SCOs opened vs. SCOs closed, by type (0,1,2,3,4), by release/component/ subsystem
Breakage and modularity	Convergence, software scrap, quality indicator	Reworked SLOC per change, by type (0,1,2,3,4), by release/component/subsystem
Rework and adaptability	Convergence, software rework, quality indicator	Average hours per change, by type (0,1,2,3,4), by release/component/ subsystem
MTBF and maturity	Test coverage/adequacy, robustness for use, quality indicator	Failure counts, test hours until failure, by release/component/ subsystem

The seven core metrics are based on common sense and field experience with both successful and unsuccessful metrics programs. Their attributes include the following:

- They are simple, objective, easy to collect, easy to interpret and hard to misinterpret.
- Collection can be automated and non intrusive.
- They provide for consistent assessment throughout the life cycle and are derived from the evolving product baselines rather than from a subjective assessment.
- They are useful to both management and engineering personnel for communicating progress and quality in a consistent format.
- They improve fidelity across the life cycle.

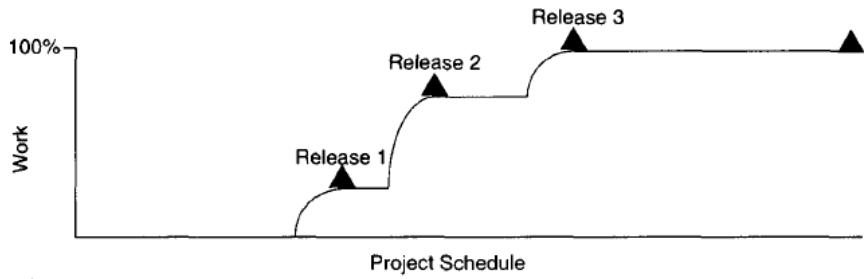
MANAGEMENT INDICATORS

There are three fundamental sets of management metrics; technical progress, financial status staffing progress. By examining these perspectives, management can generally assess whether a project is on budget and on schedule. The management indicators recommended here include standard financial status based on an earned value system, objective technical progress metrics tailored to the primary measurement criteria for each major team of the organization and staff metrics that provide insight into team dynamics.

Work & Progress

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 over time) against that plan), the default perspectives of this metric would be as follows:

- Software architecture team: use cases demonstrated
- Software development team: SLOC under baseline change management, SCOs closed.
- Software assessment team: SCOs opened, test hours executed, evaluation criteria met
- Software management team: milestones completed



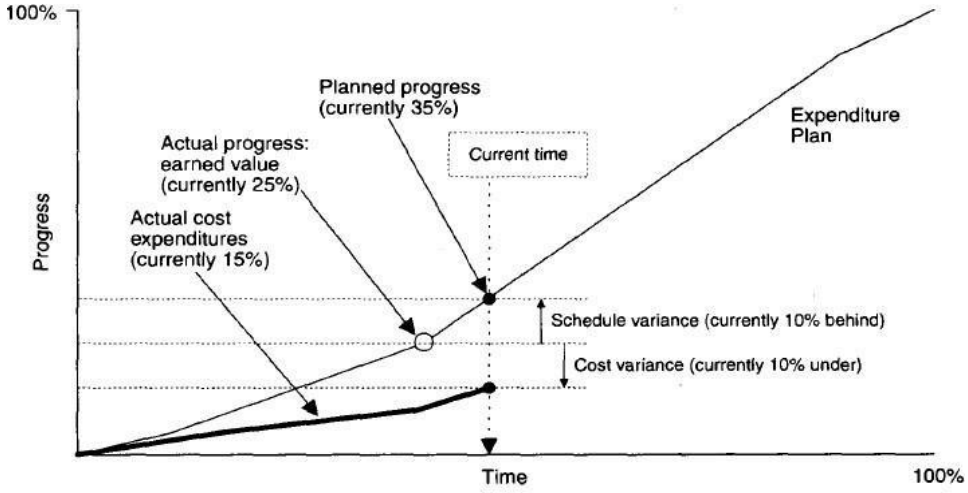
Expected progress for a typical project with three major releases

Budgeted Cost and Expenditures

To maintain management control, measuring cost expenditures over the project life cycle is always necessary. One common approach to financial performance measurement is use of an earned value system, which provides highly detailed cost and schedule insight.

Modern software processes are amenable to financial performance measurement through an earned value approach. The basic parameters of an earned value system, usually expressed in units of dollars, are as follows:

- **Expenditure Plan:** the planned spending profile· for a project over its planned schedule. For most software projects (and other labor-intensive projects), this profile generally tracks the staffing profile.
- **Actual Progress:** the technical accomplishment relative to the planned progress underlying the spending profile. In a healthy project, the actual progress tracks planned progress closely.
- **Actual Cost:** the actual spending profile for a project over its actual schedule. In a healthy project, this profile tracks the planned profile closely.
- **Earned Value:** the value that represents the planned cost of the actual progress.
- **Cost variance:** the difference between the actual cost and the earned value.
- **Positive values** correspond to over - budget situations; negative values correspond to under budget situations.
- **Schedule Variance:** the difference between the planned cost and the earned value. Positive values correspond to behind-schedule situations; negative values correspond to ahead-of-schedule situations.

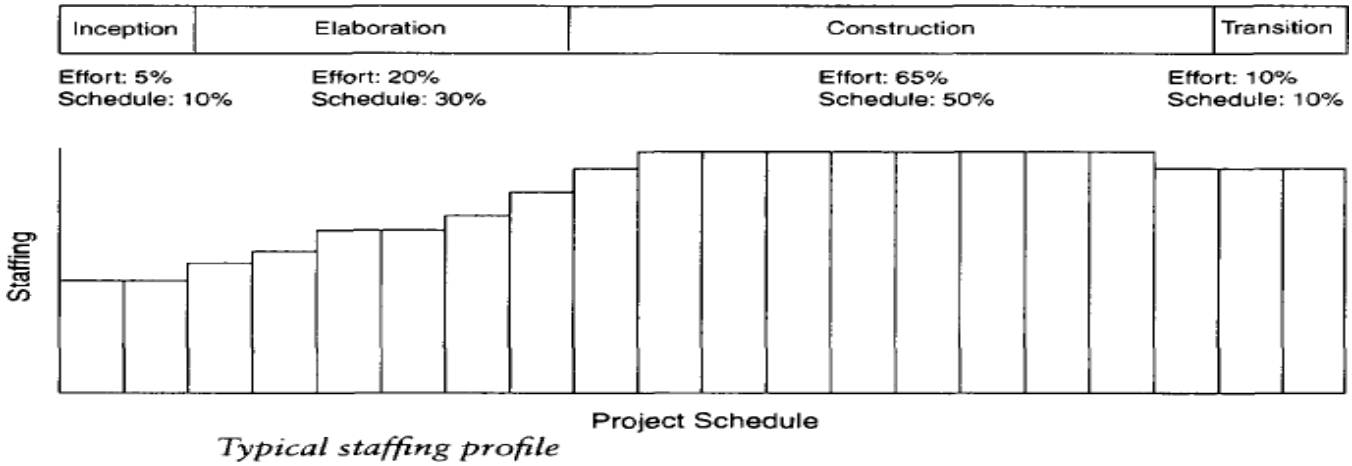


The basic parameters of an earned value system

Staffing and Team Dynamics

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 circumstance, staffing can vary. For discrete, one of-a-kind development efforts (such as building a corporate information system), the staffing profile would be typical.

It is reasonable to expect the maintenance team to be smaller than the development team for these sorts of developments. For a commercial product development, the sizes of the maintenance and development teams may be the same.



Typical staffing profile

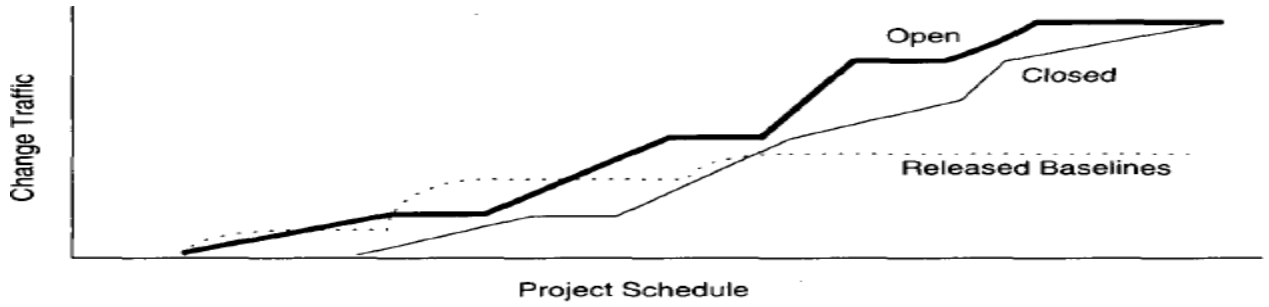
QUALITY INDICATORS

The four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data (such as design models and source code).

Change Traffic and Stability

Overall change traffic is one specific indicator of progress and quality.

Change traffic is defined as the number of software change orders opened and closed over the life cycle This metric can be collected by change type, by release, across all releases, by team, by components, by subsystem, and so forth. ***Stability*** is defined as the relationship between opened versus closed SCOs.

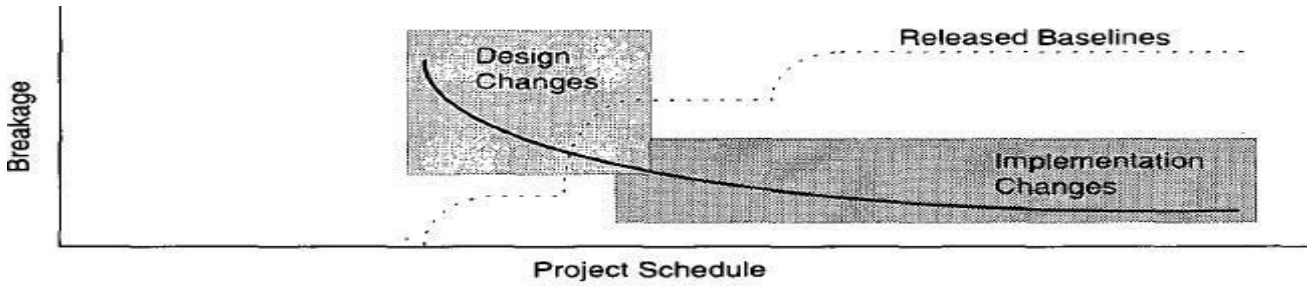


Stability expectation over a healthy project's life cycle

Breakage and Modularity

Breakage is defined as the average extent of change, which is the amount of software baseline that needs rework (in SLOC, function points, components, subsystems, files, etc).

Modularity is the average breakage trend over time. For a healthy project, the trend expectation is decreasing or stable

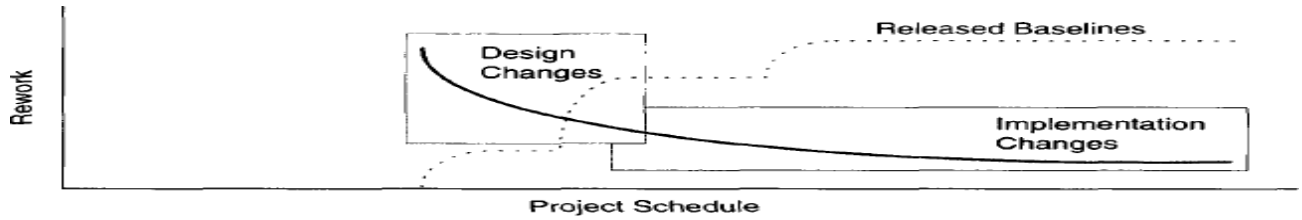


Modularity expectation over a healthy project's life cycle

Rework and Adaptability

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. For a health project, the trend expectation is decreasing or stable.

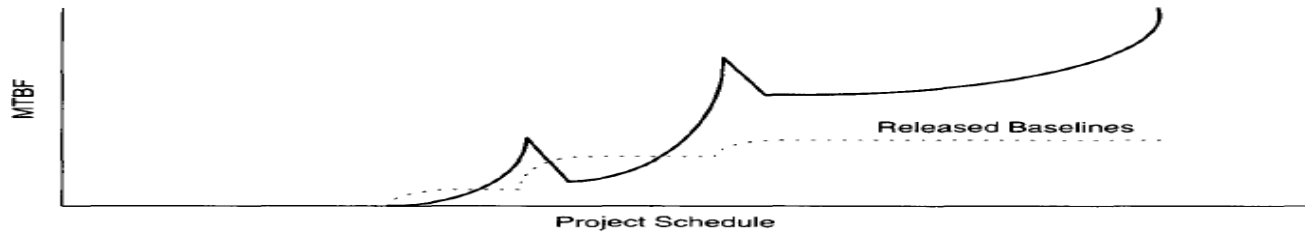


Adaptability expectation over a healthy project's life cycle

MTBF and Maturity

MTBF is the average usage time between software faults. In rough terms, MTBF is computed by dividing the test hours by the number of type 0 and type 1 SCOs. MTBF stands for Mean- Time- Between –Failures.

Maturity is defined as the MTBF trend over time



Maturity expectation over a healthy project's life cycle

LIFE CYCLE EXPECTATIONS

There is no mathematical or formal derivation for using the seven core metrics. However, there were specific reasons for selecting them:

- The quality indicators are derived form the evolving product rather than from the artifacts.
- They provide insight into the waster generated by the process. Scrap and rework metrics are a standard measurement perspective of most manufacturing processes.
- They recognize the inherently dynamic nature of an iterative development process. Rather than focus on the value, they explicitly concentrate on the trends or changes with respect to time.
- The combination of insight from the current value and the current trend provides tangible indicators for management action.

The default pattern of life-cycle metrics evolution

METRIC	INCEPTION	ELABORATION	CONSTRUCTION	TRANSITION
Progress	5%	25%	90%	100%
Architecture	30%	90%	100%	100%
Applications	<5%	20%	85%	100%
Expenditures	Low	Moderate	High	High
Effort	5%	25%	90%	100%
Schedule	10%	40%	90%	100%
Staffing	Small team	Ramp up	Steady	Varying
Stability	Volatile	Moderate	Moderate	Stable
Architecture	Volatile	Moderate	Stable	Stable
Applications	Volatile	Volatile	Moderate	Stable
Modularity	50%–100%	25%–50%	<25%	5%–10%
Architecture	>50%	>50%	<15%	<5%
Applications	>80%	>80%	<25%	<10%
Adaptability	Varying	Varying	Benign	Benign
Architecture	Varying	Moderate	Benign	Benign
Applications	Varying	Varying	Moderate	Benign
Maturity	Prototype	Fragile	Usable	Robust
Architecture	Prototype	Usable	Robust	Robust
Applications	Prototype	Fragile	Usable	Robust

PRAGMATIC SOFTWARE METRICS

Measuring is useful, but it doesn‟t do any thinking for the decision makers. It only provides data to help them ask the right questions, understand the context, and make objective decisions.

The basic characteristics of a good metric are as follows:

1. It is considered meaningful by the customer, manager and performer. Customers come to software engineering providers because the providers are more expert than they are at developing and managing software. Customers will accept metrics that are demonstrated to be meaningful to the developer.
2. It demonstrates quantifiable correlation between process perturbations and business performance. Theonly real organizational goals and objectives are financial: cost reduction, revenue increase and margin increase.
3. It is objective and unambiguously defined: Objectivity should translate into some form of numeric representation (such as numbers, percentages, ratios) as opposed to textual representations (such as excellent, good, fair, poor). Ambiguity is minimized through well understood units of measurement (such as staff-month, SLOC, change, function point, class, scenario, requirement), which are surprisingly hard to define precisely in the software engineering world.
4. It displays trends: This is an important characteristic. Understanding the change in a metric‟s value with respect to time, subsequent projects, subsequent releases, and so forth is an extremely important perspective, especially for today‟s iterative development models. It is very rare that a given metric drives the appropriate actiondirectly.
5. It is a natural by-product of the process: The metric does not introduce new artifacts or overhead activities; it is derived directly from the mainstream engineering and management workflows.

6. It is supported by automation: Experience has demonstrated that the most successful metrics are those that are collected and reported by automated tools, in part because software tools require rigorous definitions of the data they process.

METRICS AUTOMATION

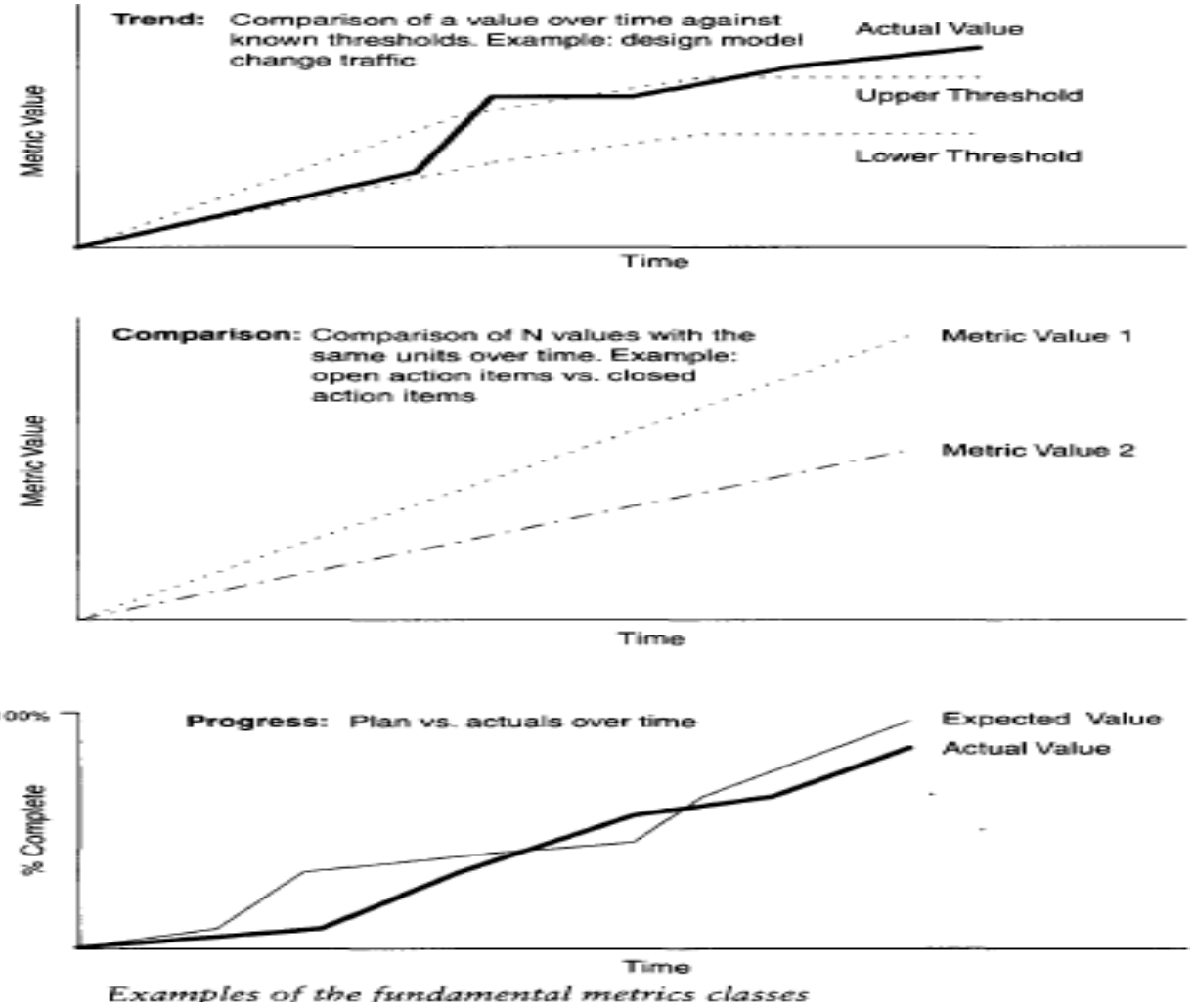
There are many opportunities to automate the project control activities of a software project. For managing against a plan, a software project control panel (SPCP) that maintains an on-line version of the status of evolving artifacts provides a key advantage.

To implement a complete SPCP, it is necessary to define and develop the following:

- ***Metrics primitives***: indicators, trends, comparisons, and progressions.
- ***A graphical user interface***: GUI support for a software project manager role and flexibility to support other roles
- ***Metric collection agents:*** data extraction from the environment tools that maintain theengineering notations .for the various artifact sets.
- ***Metrics data management server***: data management support for populating the metric displays of the GUI and storing the data extracted by the agents.
- ***Metrics definitions***: actual metrics presentations for requirements progress (extracted from requirements set artifacts), design progress (extracted from design set artifacts), implementation progress (extracted from implementation set artifacts), assessment progress (extracted from deployment set artifacts), and other progress dimensions (extracted from manual sources, financial management systems, management artifacts, etc.)
- ***Actors***: typically, the monitor and the administrator

Specific monitors (called roles) include software project managers, software development team leads, software architects, and customers.

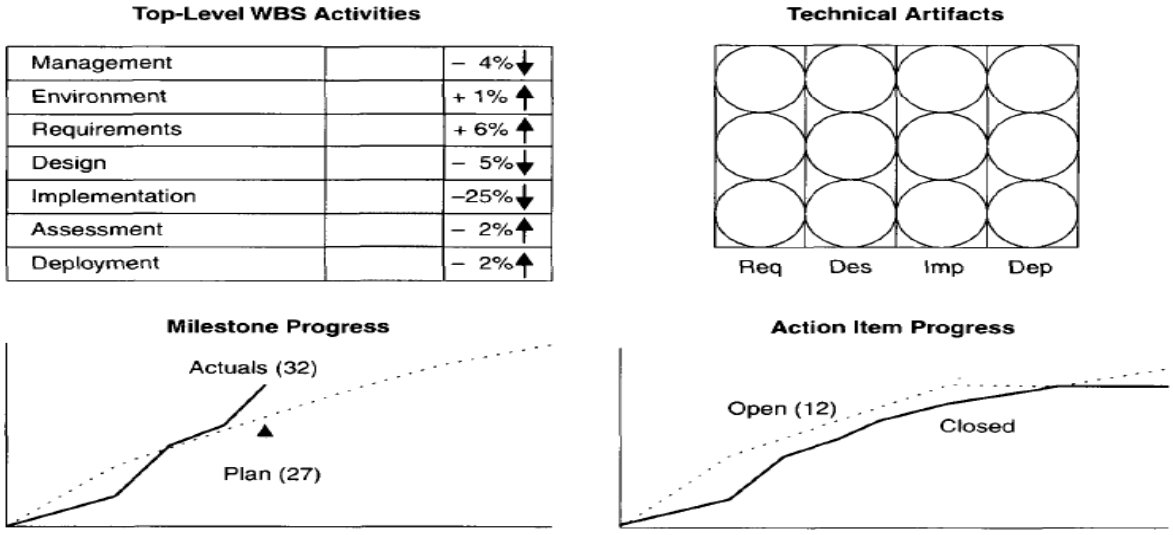
- ***Monitor:*** defines panel layouts from existing mechanisms, graphical objects, and linkages to project data; queries data to be displayed at different levels of abstraction
- ***Administrator:*** installs thesystem; defines new mechanisms, graphical objects, and linkages; archiving functions; defines composition and decomposition structures for displaying multiple levels of abstraction.



Examples of the fundamental metrics classes

In this case, the software project manager role has defined a top-level display with four graphical objects.

1. ***Project activity Status***: the graphical object in the upper left provides an overview of the status of the top-level WBS elements. The seven elements could be coded red, yellow and green to reflect the current earned valuestatus. (In Figure they are coded with white and shades of gray). For example, green would represent ahead of plan, yellow would indicate within 10% of plan, and red would identify elements that have a greater than 10% cost or schedule variance. This graphical object provides several examples of indicators: tertiary colors, the actual percentage, and the current first derivative (up arrow means getting better, down arrow means getting worse).
2. ***Technical artifact status***: the graphical object in the upper right provides an overview of the status of the evolving technical artifacts. The Req light would display an assessment of the current state of the use case models and requirements specifications. The Des light would do the same for the design models, the Imp light for the source code baseline and the Dep light for the test program.
3. ***Milestone progress***: the graphical object in the lower left provides a progress assessment of the achievementof milestones against plan and provides indicators of the current values.
4. ***Action item progress***: the graphical object in the lower right provides a different perspective of progress, showing the current number of open and close issues.



Example SPCP display for a top-level project situation

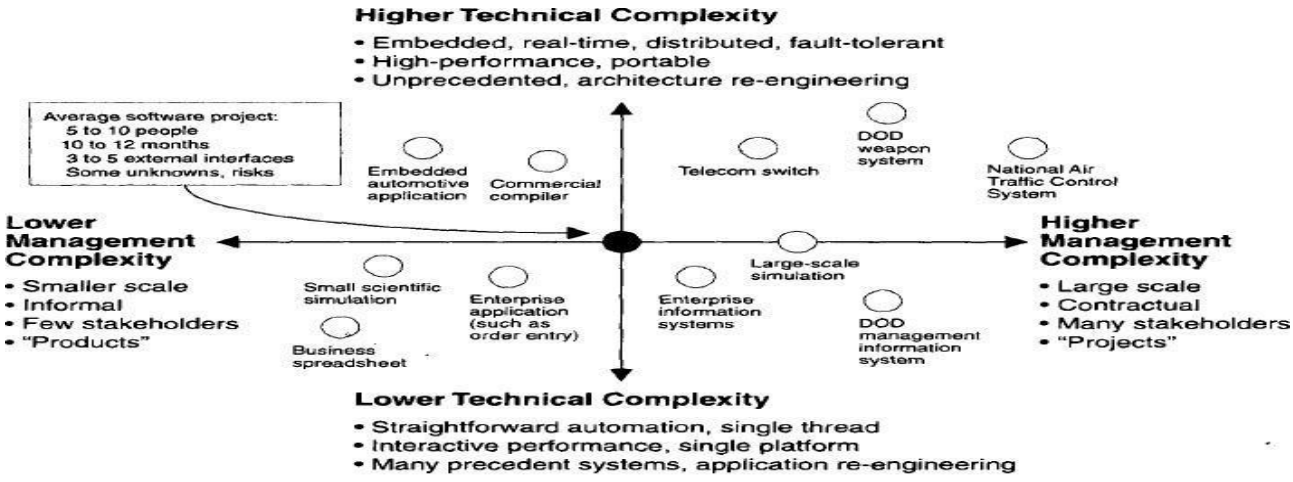
The following top-level use case, which describes the basic operational concept of an SPCP, corresponds to a monitor interacting with the control panel:

- Start the SPCP. The SPCP starts and shows the most current information that was saved when the user last used the SPCP.
- Select a panel preference. The user selects from a list of previously defined default panel preference. The SPCP displays the preference selected.
- Select a value or graph metric. The user selects whether the metric should be displayed for a given point in time or in a graph, as a trend. The default for trends is monthly.
- Select to superimpose controls. The user points to a graphical object and requests that the control values for that metric and point in time be displayed.
- Drill down to trend. The user points to a graphical object displaying a point in time and drills down to view the trend for the metric.
- Drill down to point in time. The user points to a graphical object displaying a trend and drills down to view the values for the metric.
- Drill down to lower levels of information. The user points to a graphical object displaying a point in time and drills down to view the next level of information.
- Drill down to lower level of indicators. The user points to a graphical object displaying an indicator and drills down to view the breakdown of the next level of indicators.

PROCESS DISCRIMINATES

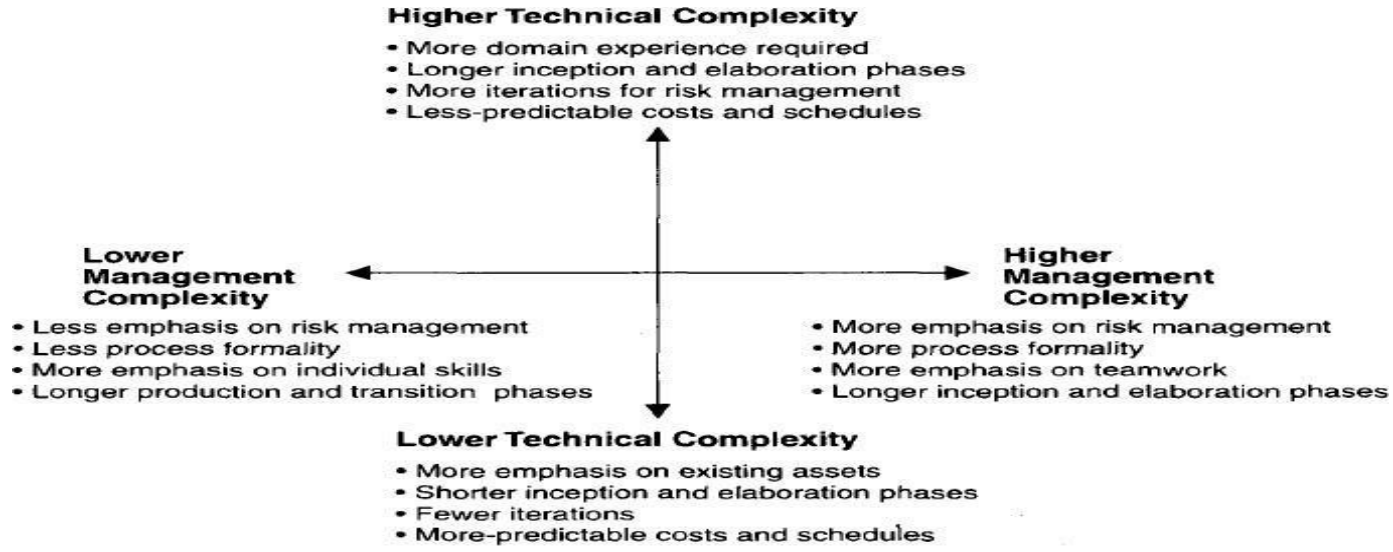
In tailoring the management process to a specific domain or project, there are tow dimensions of discriminating factors: technical complexity and management complexity.

The Figure illustrates discriminating these tow dimensions of process variability and shows some example project applications. The formality of reviews, the quality control of artifacts, the priorities f concerns and numerous other process instantiation parameters are governed by the point a project occupies in these two dimensions.



The two primary dimensions of process variability

Figure summarizes the different priorities along the two dimensions.



Priorities for tailoring the process framework

Scale

- There are many ways to measure scale, including number of source lines of code, number of function points, number of use cases, and number of dollars. From a process tailoring perspective, the primary measure of scale is the size of the team. As the headcount increases, the importance of consistent interpersonal communications becomes paramount. Otherwise, the diseconomies of scale can have a serious impact on achievement of the project objectives.
- A team of 1 (trivial), a team of 5 (small), a team of 25 (moderate), a team of 125 (large), a team of 625 (huge), and so on. As team size grows, a new level of personnel management ins introduced at roughly each factor of 5. This model can be sued to describe some of the process differences among projects of different sizes.
- Trivial-sized projects require almost no management overhead (planning, communication, coordination, progress assessment, review, administration).
- Small projects (5 people) require very little management overhead, but team leadership toward a common objective is crucial. There is some need to communicate the intermediate artifacts among teammember.
- Moderate-sized projects (25 people) require moderate management overhead, including a dedicated software project manager to synchronize team workflows and balance resources.
- Large projects (125 people) require substantial management overhead including a dedicated software project manager and several subproject managers to synchronize project-level and subproject-level workflows and to balance resources.Project performance is dependent on average people, for two reasons:

a) There are numerous mundane jobs in any large project, especially in the overhead workflows.
b) The probability of recruiting, maintaining and retaining a large umber of exceptional people is small.

- Huge projects (625 people) require substantial management overhead, including multiple software project managers an many subproject managers to synchronize project-level and subproject-level workflows and to balance resources.

Process discriminators that result from differences in project size

PROCESS PRIMITIVE	SMALLER TEAM	LARGER TEAM
Life-cycle phases	Weak boundaries between phases	Well-defined phase transitions to synchronize progress among concurrent activities
Artifacts	Focus on technical artifacts Few discrete baselines Very few management artifacts required	Change management of technical artifacts, which may result in numerous baselines Management artifacts important
Workflow effort allocations	More need for generalists, people who perform roles in multiple workflows	Higher percentage of specialists More people and teams focused on a specific workflow
Checkpoints	Many informal events for maintaining technical consistency No schedule disruption	A few formal events Synchronization among teams, which can take days
Management discipline	Informal planning, project control, and organization	Formal planning, project control, and organization
Automation discipline	More ad hoc environments, managed by individuals	Infrastructure to ensure a consistent, up-to-date environment available across all teams Additional tool integration to support project control and change control

Stakeholder Cohesion or Contention

The degree of cooperation and coordination among stakeholders (buyers, developers, users, subcontractors and maintainers, among others) can significantly drive the specifies of how a process is defined. This process parameter can range from cohesive to adversarial. Cohesive teams have common goals, complementary skills and close communications. Adversarial teams have conflicting goals, completing or incomplete skills and less-than-open communications.

Process discriminators that result from differences in stakeholder cohesion

PROCESS PRIMITIVE	FEW STAKEHOLDERS, COHESIVE TEAMS	MULTIPLE STAKEHOLDERS, ADVERSARIAL RELATIONSHIPS
Life-cycle phases	Weak boundaries between phases	Well-defined phase transitions to synchronize progress among concurrent activities
Artifacts	Fewer and less detailed management artifacts required	Management artifacts paramount, especially the business case, vision, and status assessment
Workflow effort allocations	Less overhead in assessment	High assessment overhead to ensure stakeholder concurrence
Checkpoints	Many informal events	3 or 4 formal events Many informal technical walkthroughs necessary to synchronize technical decisions Synchronization among stakeholder teams, which can impede progress for weeks
Management discipline	Informal planning, project control, and organization	Formal planning, project control, and organization
Automation discipline	(insignificant)	On-line stakeholder environments necessary

Process Flexibility or Rigor

The degree of rigor, formality and change freedom inherent in a specific project‟s “contract” (vision document, business case and development plan) will have a substantial impact on the implementation of the project‟s process. For very loose contracts such as building a commercial product within a business unit of a software company (such as a Microsoft application or a rational software corporation development tool), management complexity is minimal. In these

sorts of development processes, feature set, time to market, budget and quality can all be freely traded off and changed with very little overhead.

Process discriminators that result from differences in process flexibility

PROCESS PRIMITIVE	FLEXIBLE PROCESS	INFLEXIBLE PROCESS
Life-cycle phases	Tolerant of cavalier phase commitments	More credible basis required for inception phase commitments
Artifacts	Changeable business case and vision	Carefully controlled changes to business case and vision
Workflow effort allocations	(insignificant)	Increased levels of management and assessment workflows
Checkpoints	Many informal events for maintaining technical consistency	3 or 4 formal events Synchronization among stakeholder teams, which can impede progress for days or weeks
Management discipline	(insignificant)	More fidelity required for planning and project control
Automation discipline	(insignificant)	(insignificant)

Process Maturity

The process maturity level of the development organization, as defined by thesoftware engineering Institute‟s capability maturity model is another key driver of management complexity. Managing a mature process (level 3 or higher) is far simpler than managing an immature process (level 1 and 2). Organizations with a mature process typically 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. Tailoring a mature organization‟s process for a specific project is generally a straight forward task.

Process discriminators that result from differences in process maturity

PROCESS PRIMITIVE	MATURE, LEVEL 3 OR 4 ORGANIZATION	LEVEL 1 ORGANIZATION
Life-cycle phases	Well-established criteria for phase transitions	(insignificant)
Artifacts	Well-established format, content, and production methods	Free-form
Workflow effort allocations	Well-established basis	No basis
Checkpoints	Well-defined combination of formal and informal events	(insignificant)
Management discipline	Predictable planning Objective status assessments	Informal planning and project control
Automation discipline	Requires high levels of automation for round-trip engineering, change management, and process instrumentation	Little automation or disconnected islands of automation

Architectural Risk

The degree of technical feasibility demonstrated before commitment to full-scale production is an important dimension of defining a specific project‟s process. There are many sources of architectural risk. Some of the most important and recurring sources are system performance (resource utilization, response time, throughput, accuracy), robustness to change (addition of new features, incorporation of new technology, adaptation to dynamic operational conditions) and

system reliability (predictable behavior, fault tolerance). The degree to which these risks can bed eliminated before construction begins can have dramatic ramifications in the process tailoring.

Process discriminators that result from differences in architectural risk

PROCESS PRIMITIVE	COMPLETE ARCHITECTURE FEASIBILITY DEMONSTRATION	NO ARCHITECTURE FEASIBILITY DEMONSTRATION
Life-cycle phases	More inception and elaboration phase iterations	Fewer early iterations More construction iterations
Artifacts	Earlier breadth and depth across technical artifacts	(insignificant)
Workflow effort allocations	Higher level of design effort Lower levels of implementation and assessment	Higher levels of implementation and assessment to deal with increased scrap and rework
Checkpoints	More emphasis on executable demonstrations	More emphasis on briefings, documents, and simulations
Management discipline	(insignificant)	(insignificant)
Automation discipline	More environment resources required earlier in the life cycle	Less environment demand early in the life cycle

Domain Experience

The development organization"s domain experience governs its ability to converge on an acceptable architecture in a minimum number of iterations. An organization that has built five generations of radar control switches may be able to converge on adequate baseline architecture for a new radar application in two or three prototype release iterations. A skilled software organization building its first radar application may require four or five prototype releases before converging on an adequate baseline.

Process discriminators that result from differences in domain experience

PROCESS PRIMITIVE	EXPERIENCED TEAM	INEXPERIENCED TEAM
Life-cycle phases	Shorter engineering stage	Longer engineering stage
Artifacts	Less scrap and rework in requirements and design sets	More scrap and rework in requirements and design sets
Workflow effort allocations	Lower levels of requirements and design	Higher levels of requirements and design
Checkpoints	(insignificant)	(insignificant)
Management discipline	Less emphasis on risk management Less-frequent status assessments needed	More-frequent status assessments required
Automation discipline	(insignificant)	(insignificant)

EXAMPLE: SMALL-SCALE PROJECT VERSUS LARGE-SCALE PROJECT

- An analysis of the differences between the phases, workflows and artifacts of two projects on opposite ends of the management complexity spectrum shows how different two software project processes can be Table 14-7 illustrates the differences in schedule distribution for large and small project across the life-cycle phases. A small commercial project (for example, a 50,000 source-line visual basic windows application, built by a team

of five) may require only 1 month of inception, 2 months of elaboration, 5 months of construction and 2 months of transition. A large, complex project (for example, a 300,000 source-line embedded avionics program, built by a team of 40) could require 8 months of inception, 14 months of elaboration, 20 months of construction, and 8 months of transition. Comparing the ratios of the life cycle spend in each phase highlights the obvious differences.

- One key aspect of the differences between the two projects is the leverage of the various process components in the success or failure of the project. This reflects the importance of staffing or the level of associated risk management.

Differences in workflow priorities between small and large projects

RANK	SMALL COMMERCIAL PROJECT	LARGE, COMPLEX PROJECT
1	Design	Management
2	Implementation	Design
3	Deployment	Requirements
4	Requirements	Assessment
5	Assessment	Environment
6	Management	Implementation
7	Environment	Deployment

The following list elaborates some of the key differences in discriminators of success.

- Design is key in both domains. Good design of a commercial product is a key differentiator in the marketplace and is the foundation for efficient new product releases. Good design of a large, complex project is the foundation for predictable, cost-efficient construction.
- Management is paramount in large projects, where the consequences of planning errors, resource allocation errors, inconsistent stakeholder expectations and other out-of-balance factors can have catastrophic consequences for the overall team dynamics. Management is far less important in a small team, where opportunities for miscommunications are fewer and their consequences less significant.
- Deployment plays a far greater role for a small commercial product because there is a broad user base of diverse individuals and environments.

Differences in artifacts between small and large projects

ARTIFACT	SMALL COMMERCIAL PROJECT	LARGE, COMPLEX PROJECT
Work breakdown structure	1-page spreadsheet with 2 levels of WBS elements	Financial management system with 5 or 6 levels of WBS elements
Business case	Spreadsheet and short memo	3-volume proposal including technical volume, cost volume, and related experience
Vision statement	10-page concept paper	200-page subsystem specification
Development plan	10-page plan	200-page development plan
Release specifications and number of releases	3 interim release specifications	8 to 10 interim release specifications
Architecture description	5 critical use cases, 50 UML diagrams, 20 pages of text, other graphics	25 critical use cases, 200 UML diagrams, 100 pages of text, other graphics
Software	50,000 lines of Visual Basic code	300,000 lines of C++ code
Release description	10-page release notes	100-page summary
Deployment	User training course Sales rollout kit	Transition plan Installation plan
User manual	On-line help and 100-page user manual	200-page user manual
Status assessment	Quarterly project reviews	Monthly project management reviews