Unit 1:

SOFTWARE PROCESS MATURITY

Software maturity Framework, Principles of Software Process Change, Software Process Assessment, The Initial Process, The Repeatable Process, The Defined Process, The Managed Process, The Optimizing Process. Process Reference Models Capability Maturity Model (CMM), CMMI, PCMM, PSP, TSP).

IMPORTANT QUOTES:

If you don't know where you are going, any road will do." Chinese Proverb

"If you don't know where you are, a map won't help." Watts Humphrey

"If you don't know where you are going, a map won't get you there any faster." Anonymous

"You can't expect to be a functional employee in a dysfunctional environment" Watts Humphrey

WHY SHOULD WE MANAGE THE SOFTWARE PROCESS?

Individuals, Teams, and Armies:

History of software is one of increasing scale

Initially a few people could craft small programs

Today large projects require the coordinated work of many teams

The increase in scale requires a more structured approach to software process management

People and the Software Process

- Talented people are the most important element in a software organization
- Successful organizations provide a structured and disciplined environment to do cooperative work
- Alternative
  - Endless hours of repetitively solving technically trivial problems
  - Time is consumed by mountains of uncontrolled detail
- If the details are not managed, the best people cannot be productive

- First class people need the support of an orderly process to do first-class work

MYTH OF THE SUPER PROGRAMMERS:

- Common view: First-class people intuitively know how to do first-class work
  - Implication: No orderly process framework is needed
  - Conclusion: Organizations with the best people should not suffer from software quality and productivity problems
- However, studies show that companies with top graduates from leading universities are still plagued with the same problems
  - New Conclusion: The best people need to be supported with an effectively managed software process

MYTH OF TOOLS AND TECHNOLOGY:

- Common View: Some technically advanced tool or method will provide a magic answer to the software crisis
- Reality: Technology is vital, but unthinking reliance on an undefined "silver bullet" will divert attention from the need for better process management

MAJOR CONCERNS OF SOFTWARE PROFESSIONALS:

- Open-ended requirements
- Uncontrolled change
- Arbitrary schedules
- Insufficient test time
- Inadequate training
- Unmanaged system standards

LIMITING FACTORS IN USING SOFTWARE TECHNOLOGY:

- Poorly-defined process
- Inconsistent implementation
- Poor process management

FOCUSING ON SOFTWARE PROCESS MANAGEMENT:

- Software process: the set of actions required to efficiently transform a user's need into an effective software solution
- Many software organizations have trouble defining and controlling this process
  - Even though this is where they have the greatest potential for improvement
- This is the focus of the book "Managing the Software Process"

A SOFTWARE MATURITY FRAMEWORK:

Software maturity Framework: Fundamentally, software development must be predictable. The software process is the set of tools, methods, and practices we use to produce a software product. The objectives of software process management are to produce products according to plan while simultaneously improving the organization’s capability to produce better products. The basic principles are those of statistical process control. A process is said to be stable or under statistical control if its future performance is predictable within established statistical limits.

When a process is under statistical control, repeating the work in roughly the same way will produce roughly the same result. To obtain consistently better results, it is necessary to improve the process. If the process is not under statistical control, sustained progress is not possible until it is.

Lord Kelvin - “When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced the stage of science.” (But, your numbers must be reasonably meaningful.)

The mere act of measuring human processes changes them because of people's fears, and so forth. Measurements are both expensive and disruptive; overzealous measurements can disrupt the process under study.

Principles of Software Process Change:

People:

•The best people are always in short supply

•you probably have about the best team you can get right now.

•With proper leadership and support, most people can do much better than they are currently doing Design:

•Superior products have superior design. Successful products are designed by people who understand the application (domain engineer).

•A program should be viewed as executable knowledge. Program designers should have application knowledge.

The Six Basic Principles of Software Process Change:

•Major changes to the process must start at the top

. •Ultimately, everyone must be involved.

•Effective change requires great knowledge of the current process

•Change is continuous

•Software process changes will not be retained without conscious effort and periodic reinforcement

•Software process improvement requires investment.

Continuous Change:

•Reactive changes generally make things worse

•Every defect is an improvement opportunity

•Crisis prevention is more important than crisis recovery

SOFTWARE PROCESSES CHANGES WON'T STICK BY THEMSELVES

The tendency for improvements to deteriorate is characterized by the term entrophy

(Webster's: a measure of the degree of disorder in a…system; entrophy always increases and
available energy diminishes in a closed system.). New methods must be carefully introduced
and periodically monitored, or they to will rapidly decay. Human adoption of new process
involves four stages:

- Installation - Initial training
- Practice - People learn to perform as instructed
- Proficiency - Traditional learning curve
- Naturalness - Method ingrained and performed without intellectual effort.

It Takes Time, Skill, and Money!

- To improve the software process, someone must work on it
- Unplanned process improvement is wishful thinking
- Automation of a poorly defined process will produce poorly defined results
- Improvements should be made in small steps
- Train!!!!

Some Common Misconceptions about the Software Process

- We must start with firm requirements
- If it passes test it must be OK
- Software quality can't be measured
- The problems are technical
- We need better people
- Software management is different

SOFTWARE PROCESS ASSESSMENT

Process assessments help software organizations improve themselves by identifying their
crucial problems and establishing improvement priorities. The basic assessment objectives

are:

•Learn how the organization works

•Identify its major problems

•Enroll its opinion leaders in the change process

The essential approach is to conduct a series of structured interviews with key people in the
organization to learn their problems, concerns, and creative ideas.

ASSESSMENT OVERVIEW:

A software assessment is not an audit. Audits are conducted for senior managers who suspect problems and send in experts to uncover them. A software process assessment is a review of a software organization to advise its management and professionals on how they can improve their operation.

The phases of assessment are:

•Preparation - Senior management agrees to participate in the process and to take actions on the resulting recommendations or explain why not. Concludes with a training program for the assessment team

•Assessment - The on-site assessment period. It takes several days to two or more weeks. It concludes with a preliminary report to local management.

•Recommendations - Final recommendations are presented to local managers. A local action team is then formed to plan and implement the recommendations.

Five Assessment Principles:

•The need for a process model as a basis for assessment

•The requirement for confidentiality

•Senior management involvement

•An attitude of respect for the views of the people in the organization be assessed

•An action orientation

Start with a process model - Without a model, there is no standard; therefore, no measure of change. Observe strict confidentiality - Otherwise, people will learn they cannot speak in confidence. This means managers can't be in interviews with their subordinates. Involve senior management - The senior manager (called site manager here) sets the organizations priorities. The site manager must be personally involved in the assessment and its follow-up actions. Without this support, the assessment is a waste of time because lasting improvement must survive periodic crises. Respect the people in the assessed organization - They probably work hard and are trying to improve. Do not appear arrogant; otherwise, they will not cooperate and may try to prove the team is ineffective. The only source of real information is from the workers.

Assessment recommendations should highlight the three or four items of highest priority. Don't overwhelm the organization. The report must always be in writing. Implementation Considerations - The greatest risk is that no significant improvement actions will be taken (the "disappearing problem" syndrome). Superficial changes won't help. A small, full-time group should guide the implementation effort, with participation from other action plan working groups. Don't forget that site managers can change or be otherwise distracted, so don't rely on that person solely, no matter how committed.

THE INITIAL PROCESS(LEVEL1)

Usually ad hoc and chaotic - Organization operates without formalized procedures, cost estimates, and project plans. Tools are neither well integrated with the process nor uniformly applied. Change control is lax, and there is little senior management exposure or understanding of the problems and issues. Since many problems are deferred or even forgotten, software installation and maintenance often present serious problems. While organizations at this level may have formal procedures for planning and tracking work, there is no management mechanism to insure they are used. Procedures are often abandoned in a crisis in favor of coding and testing. Level 1 organizations don't use design and code inspections and other techniques not directly related to shipping a product. Organizations at Level 1 can improve their performance by instituting basic project controls.

The most important ones are

•Project management

•Management oversight

•Quality assurance

•Change control

THE REPEATABLE PROCESS (LEVEL 2)

This level provides control over the way the organization establishes plans and commitments. This control provides such an improvement over Level 1 that the people in the organization tend to believe they have mastered the software problem. This strength, however, stems from their prior experience in doing similar work. Level 2 organizations face major risks when presented with new challenges.

Some major risks:

•New tools and methods will affect processes, thus destroying the historical base on which the organization lies. Even with a defined process framework, a new technology can do more harm than good.

•When the organization must develop a new kind of product, it is entering new territory.

•Major organizational change can be highly disruptive. At Level 2, a new manager has no orderly basis for understanding an organization’s operation, and new members must learn the ropes by word of mouth. Key actions required to advance from Repeatable to the next stage, the Defined Process, are:

•Establish a process group: A process group is a technical resource that focuses heavily on improving software processes. In most software organizations, all the people are generally devoted to product work. Until some people are assigned full-time to work on the process, little orderly progress can be made in improving it.

•Establish a software development process architecture (or development cycle) that describes the technical and management activities required for proper execution of the development process. The architecture is a structural decomposition of the development cycle into tasks, each of which has a defined set of prerequisites, functional decompositions, verification procedures, and task completion specifications.

•Introduce a family of software engineering methods and technologies. These include design and code inspections, formal design methods, library control systems, and comprehensive testing methods. Prototying and modern languages should be considered.

THE DEFINED PROCESS (LEVEL 3)

The organization has the foundation for major and continuing change. When faced with a crisis, the software teams will continue to use the same process that has been defined.

However, the process is still only qualitative; there is little data to indicate how much is accomplished or how effective the process is. There is considerable debate about the value of software process measurements and the best one to use.

The key steps required to advance from the Defined Process to the next level are:

•Establish a minimum set of basic process measurements to identify the quality and cost parameters of each process step. The objective is to quantify the relative costs and benefits of each major process activity, such as the cost and yield of error detection and correction methods.

•Establish a process database and the resources to manage and maintain it. Cost and yield data should be maintained centrally to guard against loss, to make it available for all projects, and to facilitate process quality and productivity analysis. Provide sufficient process resources to gather and maintain the process data and to advise project members on its use. Assign skilled professionals to monitor the quality of the data before entry into the database and to provide guidance on the analysis methods and interpretation.

•Assess the relative quality of each product and inform management where quality targets are
not being met. Should be done by an independent quality assurance group.

THE MANAGED PROCESS (LEVEL 4)

Largest problem at Level 4 is the cost of gathering data. There are many sources of potentially valuable measure of the software process, but such data are expensive to collect and maintain.

Productivity data are meaningless unless explicitly defined. For example, the simple measure of lines of source code per expended development month can vary by 100 times or more,

depending on the interpretation of the parameters When different groups gather data but do not use identical definitions, the results are not comparable, even if it makes sense to compare them. It is rare when two processes are comparable by simple measures. The variations in task complexity caused by different product types can exceed five to one. Similarly, the cost per line of code for small modifications is often two to three times that for new programs.

Process data must not be used to compare projects or individuals. Its purpose is too illuminate the product being developed and to provide an informed basis for improving the process.

When such data are used by management to evaluate individuals or terms, the reliability of the data itself will deteriorate. The two fundamental requirements for advancing from the Managed Process to the next level are:

•Support automatic gathering of process data. All data is subject to error and omission, some data cannot be gathered by hand, and the accuracy of manually gathered data is often poor.

•Use process data to analyze and to modify the process to prevent problems and improve efficiency.

THE OPTIMIZING PROCESS (LEVEL 5)

To this point software development managers have largely focused on their products and will typically gather and analyze only data that directly relates to product improvement. In the Optimizing Process, the data are available to tune the process itself. For example, many types of errors can be identified far more economically by design or code inspections than by testing. However, some kinds of errors are either uneconomical to detect or almost impossible to find except by machine. Examples are errors involving interfaces, performance, human factors, and error recovery.

So, there are two aspects of testing: removal of defects and assessment of program quality. To reduce the cost of removing defects, inspections should be emphasized. The role of functional and system testing should then be changed to one of gathering quality data on the program. This involves studying each bug to see if it is an isolated problem or if it indicates design problems that require more comprehensive analysis. With Level 5, the organization should identify the weakest elements of the process and fix them. Data are available to justify the application of technology to various critical tasks, and numerical evidence is available on the effectiveness with which the process has been applied to any given product.

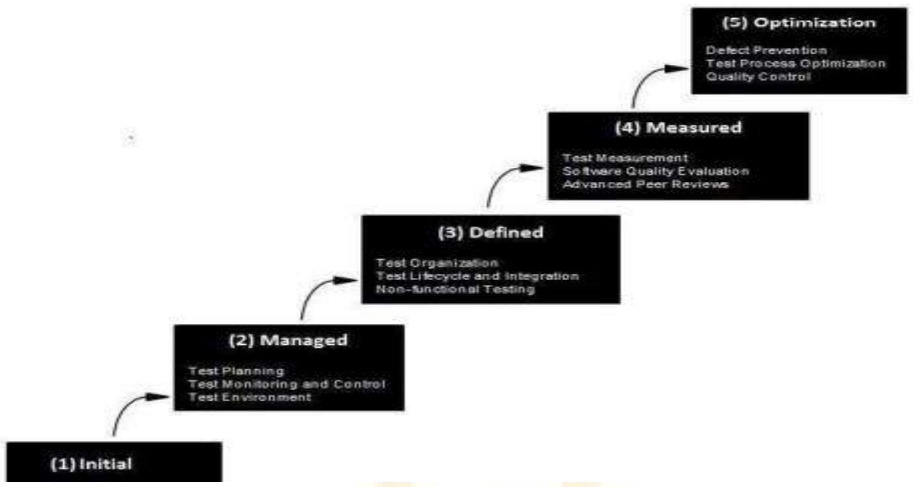
Process reference models; The process framework or reference model acts as an interface between the way the content is organized and the way work is performed. A uniform process model organized under a process reference model makes business modeling and systems designing much easier

CAPABILITY MATURITY MODEL (CMM):

Broadly refers to a process improvement approach that is based on a process model. CMM also refers specifically to the first such model, developed by the Software Engineering Institute (SEI) in the mid-1980s, as well as the family of process models that followed.

The Software Engineering Institute (SEI) Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization. The higher the level, the better the software development process, hence reaching each level is an expensive and timeconsuming process.

Levels of CMM



Level One :Initial - The software process is characterized as inconsistent, and occasionally even chaotic. Defined processes and standard practices that exist are abandoned during a crisis. Success of the organization majorly depends on an individual effort, talent, and heroics. The heroes eventually move on to other organizations taking their wealth of knowledge or lessons learnt with them.

Level Two: Repeatable - This level of Software Development Organization has a basic and consistent project management processes to track cost, schedule, and functionality. The process is in place to repeat the earlier successes on projects with similar applications. Program management is a key characteristic of a level two organization.

Level Three: Defined - The software process for both management and engineering activities are documented, standardized, and integrated into a standard software process for the entire organization and all projects across the organization use an approved, tailored version of the organization's standard software process for developing,testing and maintaining the application.

Level Four: Managed - Management can effectively control the softwaredevelopment effort using precise measurements. At this level, organization set a quantitative quality goal for both software process and software maintenance. At this maturity level, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable.

Level Five: Optimizing - The Key characteristic of this level is focusing on continually improving process performance through both incremental and innovative technological improvements. At this level, changes to the process are to improve the process performance and at the same time maintaining statistical probability to achieve the established quantitative process-improvement objectives.

WHAT IS CMMI ?

CMM Integration project was formed to sort out the problem of using multiple CMMs.

CMMI Product Team's mission was to combine three Source Models into a single improvement framework to be used by the organizations pursuing enterprise-wide process

improvement. These three Source Models are :

- Capability Maturity Model for Software (SW-CMM) - v2.0 Draft C
- Electronic Industries Alliance Interim Standard (EIA/IS) - 731 Systems Engineering
- Integrated Product Development Capability Maturity Model (IPD-CMM) v0.98

CMM Integration:

- builds an initial set of integrated models.
- - improves best practices from source models based on lessons learned.
- - establishes a framework to enable integration of future models.

Following are obvious objectives of CMMI:

Produce quality products or services: The process-improvement concept in CMMI models evolved out of the Deming, Juran, and Crosby quality paradigm:

Quality products are a result of quality processes. CMMI has a strong focus on qualityrelated activities including requirements management, quality assurance, verification, and validation.

Create value for the stockholders: Mature organizations are more likely to make better cost and revenue estimates than those with less maturity, and then perform in line with those estimates. CMMI supports quality products, predictable schedules, and effective measurement to support management in making accurate and defensible forecasts. This process maturity can guard against project performance problems that could weaken the value of the organization in the eyes of investors.

Enhance customer satisfaction: Meeting cost and schedule targets with high-quality products that are validated against customer needs is a good formula for customer satisfaction. CMMI addresses all of these ingredients through its emphasis on planning, monitoring, and measuring, and the improved predictability that comes with more capable processes.

The CMM Integration is a model that has integrated several disciplines/bodies of knowledge. Currently there are four bodies of knowledge available to you when selecting a CMMI model. **SYSTEMS ENGINEERING**

Systems engineering covers the development of complete systems, which may or may not include software. Systems engineers focus on transforming customer needs, expectations, and constraints into product solutions and supporting these product solutions throughout the entire lifecycle of the product.

SOFTWARE ENGINEERING

Software engineering covers the development of software systems. Software engineers focus on the application of systematic, disciplined, and quantifiable approaches to the development, operation, and maintenance of software.

INTEGRATED PRODUCT AND PROCESS DEVELOPMENT

Integrated Product and Process Development (IPPD) is a systematic approach that achieves a timely collaboration of relevant stakeholders throughout the life of the product to better satisfy customer needs, expectations, and requirements. The processes to support an IPPD approach are integrated with the other processes in the organization. If a project or organization chooses IPPD, it performs the IPPD best practices concurrently with other best practices used to produce products (e.g., those related to systems engineering). That is, if an organization or project wishes to use IPPD, it must select one or more disciplines in addition to IPPD.

SUPPLIER SOURCING

As work efforts become more complex, project managers may use suppliers to perform functions or add modifications to products that are specifically needed by the project. When those activities are critical, the project benefits from enhanced source analysis and from monitoring supplier activities before product delivery. Under these circumstances, the supplier sourcing discipline covers the acquisition of products from suppliers. Similar to IPPD best practices, supplier sourcing best practices must be selected in conjunction with best practices used to produce products.

CMMI Discipline Selection Selecting a discipline may be a difficult step and depends on what an organization wants to improve.

- If you are improving your systems engineering processes, like Configuration Management, Measurement and Analysis, Organizational Process Focus, Project Monitoring and Control, Process and Product Quality Assurance, Risk Management, Supplier Agreement Management etc., then you should select Systems engineering (SE) discipline. The discipline amplifications for systems engineering receive special emphasis.
- If you are improving your integrated product and process development processes like Integrated Teaming, Organizational Environment for

Integration, then you should select IPPD. The discipline amplifications for IPPD receive special emphasis.

- If you are improving your source selection processes like Integrated Supplier Management then you should select Supplier sourcing (SS). The discipline amplifications for supplier sourcing receive special emphasis.
- If you are improving multiple disciplines, then you need to work on all the areas related to those disciplines and pay attention to all of the discipline amplifications for those disciplines

The CMMI is structured as follows −

 Maturity Levels (staged representation) or Capability Levels (continuous representation)

 Process Areas

 Goals: Generic and Specific

 Common Features

 Practices: Generic and Specific

This chapter will discuss about two CMMI representations and rest of the subjects will be

covered in subsequent chapters.

A representation allows an organization to pursue different improvement objectives. An

organization can go for one of the following two improvement paths**.**

Staged Representation The staged representation is the approach used in the Software CMM. It is an approach that uses predefined sets of process areas to define an improvement path for an organization. This improvement path is described by a model component called a Maturity Level. A maturity level is a well-defined evolutionary plateau towards achieving improved organizational processes.

CMMI Staged Representation Provides a proven sequence of improvements, each serving as a foundation for the next. Permits comparisons across and among organizations by the use of maturity levels. Provides an easy migration from the SW-CMM to CMMI. Provides a single rating that summarizes appraisal results and allows comparisons among organizations.

Thus Staged Representation provides a pre-defined roadmap for organizational improvement based on proven grouping and ordering of processes and associated organizational relationships. You cannot divert from the sequence of steps.

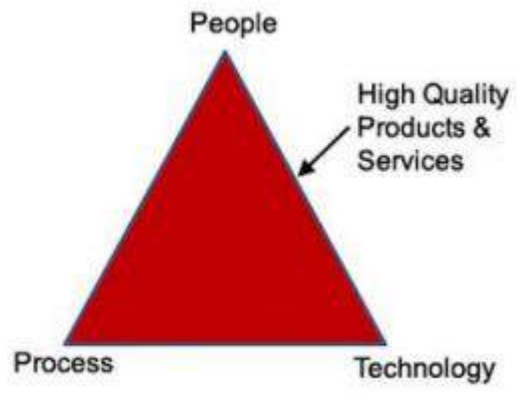
The People Capability Maturity Model (People CMM, P-CMM) is part of the CMMI product family of process maturity models. It is a framework to guide organisations in improving their processes for managing and developing human workforces. It helps organisations to characterize the maturity of their workforce practices, establish a program of continuous workforce development, set priorities for improvement actions, integrate workforce development with Process Improvement, and establish a culture of excellence. PCMM is based on proven practices in fields of human resources, knowledge management, and organisational development. P-CMM is part of the CMMI product family of process maturity models. It describes a progression for continuous improvement and process improvement of the HR processes for managing and developing human workforces. The P-CMM framework enables organisations to incrementally focus on key process areas and to lay foundations for improvement in workforce practices. Unlike other HR models, P-CMM requires that key process areas, improvements, interventions, policies, and procedures are institutionalised across the organisation — irrespective of function or level. Therefore, all improvements have to percolate throughout the organisation, to ensure consistency of focus, to place emphasis on a participatory culture, embodied in a team-based environment, and encouraging individual innovation and creativity. Process Maturity Rating The process maturity rating is from ad hoc and inconsistently performed practices, to a mature and disciplined development of the knowledge, skills, and motivation of the workforce.

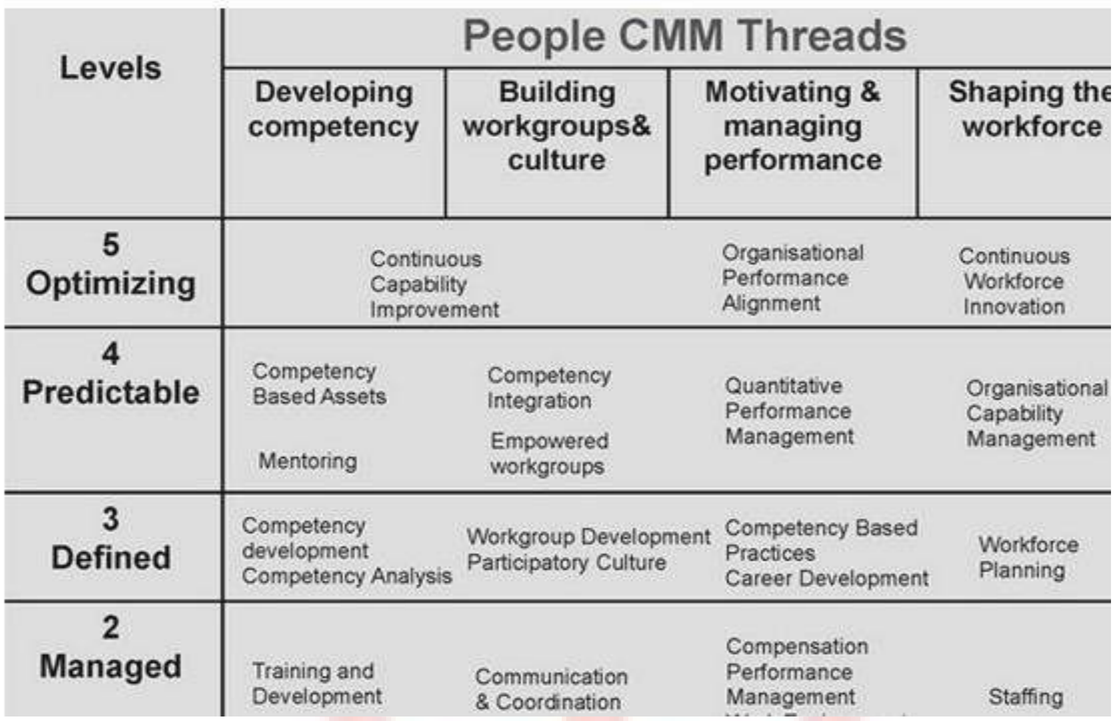
Traditionally, process maturity models like ISO/IEC 15504 or CMMI focus on organisational improvement with respect to operational (Product) Development Processes. PCMM in contrast focus instead on the three prominent factors for operational capability to deliver successful products and services:

1. People

2. Process

3. Products, Technology





Levels	People CMM Threads			
	Developing competency	Building workgroups& culture	Motivating & managing performance	Shaping the workforce
5 Optimizing	Continuous Capability Improvement		Organisational Performance Alignment	Continuous Workforce Innovation
4 Predictable	Competency Based Assets Mentoring	Competency Integration Empowered workgroups	Quantitative Performance Management	Organisational Capability Management
3 Defined	Competency development Competency Analysis	Workgroup Development Participatory Culture	Competency Based Practices Career Development	Workforce Planning
2 Managed	Training and Development	Communication & Coordination	Compensation Performance Management	Staffing

PSP

The Personal Software Process (PSP) is a structured software development process that is designed to help software engineers better understand and improve their performance by bringing discipline to the way they develop software and tracking their predicted and actual development of the code. It clearly shows developers how to manage the quality of their products, how to make a sound plan, and how to make commitments. It also offers them the data to justify their plans. They can evaluate their work and suggest improvement direction by analyzing and reviewing development time, defects, and size data. The PSP was created by Watts Humphrey to apply the underlying principles of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to the software development practices of a single developer. It claims to give software engineers the process skills necessary to work on a team software process (TSP) team.

The PSP aims to provide software engineers with disciplined methods for improving personal

software development processes. The PSP helps software engineers to:

 Improve their estimating and planning skills.

 Make commitments they can keep.

 Manage the quality of their projects.

 Reduce the number of defects in their work.

PSP training follows an evolutionary improvement approach: an engineer learning to integrate the PSP into his or her process begins at the first level – PSP0 – and progresses in process maturity to the final level – PSP2.1. Each Level has detailed scripts, checklists and templates to guide the engineer through required steps and helps the engineer improve their own personal software process. Humphrey encourages proficient engineers to customize these scripts and templates as they gain an understanding of their own strengths and weaknesses.

 Process The input to PSP is the requirements; requirements document is completed and delivered to the engineer.

TSP

The team software process (TSP) provides a defined operational process framework that is designed to help teams of managers and engineers organize projects and produce software the principles products that range in size from small projects of several thousand lines of code (KLOC) to very large projects greater than half a million lines of code. The TSP is intended to improve the levels of quality and productivity of a team's software development project, in order to help them better meet the cost and schedule commitments of developing a software system The initial version of the TSP was developed and piloted by Watts Humphrey in the late

1990s and the Technical Report for TSP sponsored by the U.S. Department of Defense was published in November 2000. The book by Watts Humphrey, Introduction to the Team Software Process, presents a view of the TSP intended for use in academic settings, that focuses on the process of building a software production team, establishing team goals, distributing team roles, and other teamwork-related activities. The primary goal of TSP is to create a team environment for establishing and maintaining a self-directed team, and supporting disciplined individual work as a base of PSP framework. Self-directed team means that the team manages itself, plans and tracks their work, manages the quality of their work, and works proactively to meet team goals. TSP has two principal components: team-building and team-working. Team-building is a process that defines roles for each team member and sets up teamwork through TSP launch and periodical relaunch. Team-working is a process that deals with engineering processes and practices utilized by the team. TSP, in short, provides engineers and managers with a way that establishes and manages their team to produce the high-quality software on schedule and budget.

HOW TSP WORKS:

Before engineers can participate in the TSP, it is required that they have already learned about the PSP, so that the TSP can work effectively. Training is also required for other team members, the team lead and management. The TSP software development cycle begins with a planning process called the launch, led by a coach who has been specially trained, and is either certified or provisional. The launch is designed to begin the team building process, and during this time teams and managers establish goals, define team roles, assess risks, estimate effort, allocate tasks, and produce a team plan. During an execution phase, developers track planned and actual effort, schedule, and defects meeting regularly (usually weekly) to report status and revise plans. A development cycle ends with a Post Mortem to assess performance,revise planning parameters, and capture lessons learned for process improvement. The coach role focuses on supporting the team and the individuals on the team as the process expert while being independent of direct project management responsibility. The team leader role is different from the coach role in that, team leaders are responsible to management for products and project outcomes while the coach is responsible for developing individual and team performance

Unit 2:
Software Project Management Renaissance

Conventional Software Management, Evolution of Software Economics, Improving Software
Economics, The old way and the new way.
Life-Cycle Phases and Process artifacts
Engineering and Production stages, inception phase, elaboration phase, construction phase, transition
phase, artifact sets, management artifacts, engineering artifacts and pragmatic artifacts, model-based
software architectures.

CONVENTIONAL SOFTWARE MANAGEMENT

Conventional software management practices are sound in theory, but practice is still tied to archaic (outdated) 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

1.Software development is still highly unpredictable. Only about **10%** of software projects are delivered **successfully** within initial budget and schedule estimates.

2.Management discipline is more of a discriminator in success or failure than are technology advances.

3.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.

THE WATERFALL MODEL

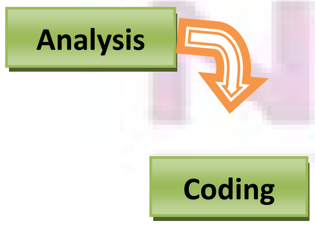
Most software engineering texts present the waterfall model as the source of the "conventional" software process

IN THEORY

It provides an insightful and concise summary of conventional software management.Three main primary points are

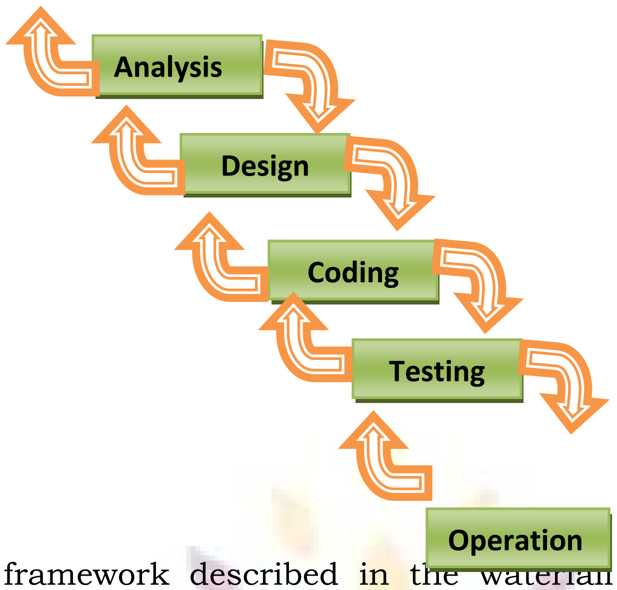
1.There are two essential steps common to the development of computer programs: **analysis** and **coding**.

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.

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. Below Figure illustrates the resulting project profile and the basic steps in developing a large-scale program.



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 from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the requirements must be modified or a substantial design change is warranted.

Five necessary improvements for waterfall model are:-

1. **Program design comes first.** Insert a preliminary program design phase between the software requirements generation phase and the analysis phase. **By this technique, the program designer assures that the software will not fail because of storage, timing, and data flux (continuous change).** As analysis proceeds in the succeeding phase, the program designer must impose on the analyst the storage, timing, and operational constraints in such a way that he senses the consequences. If the total resources to be applied are insufficient or if the embryonic(in an early stage of development) operational design is wrong, it will be recognized at this early stage and the iteration with requirements and preliminary design can be redone before final design,

coding, and test commences. How is this program design procedure implemented?

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.

2. Document the design. The amount of documentation required on most software programs is quite a lot, certainly much more than most programmers, analysts, or program designers are willing to do if left to their own devices. Why do we need so much documentation? (1) Each designer must communicate with interfacing designers, managers, and possibly customers. (2) During early phases, the documentation is the design. (3) 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.

3.Do it twice. If a computer program is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned. Note that this is simply the entire process done in miniature, to a time scale that is relatively small with respect to the overall effort. In the first version, the team must have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives, forget the straightforward aspects of the design that aren't worth studying at this early point, and, finally, arrive at an error-free program.

4.**Plan, control, and monitor testing.** Without question, the biggest user of project resources-manpower, computer time, and/or management judgment-is the test phase. This is the phase of greatest risk in terms of cost and schedule. It occurs at the latest point in the schedule, when backup alternatives are least available, if at all. The previous three recommendations were all aimed at uncovering and solving problems before entering the test phase. However, even after doing these things, there is still a test phase and there are still important things to be done, including: (1) employ a team of test specialists who were not responsible for the original design; (2) employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses (do not use the computer to detect this kind of thing, it is too expensive); (3) test every logic path; (4) employ the final checkout on the target computer**.**

5. **Involve the customer.** It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery. There are three points following requirements definition where the insight, judgment, and commitment of the customer can bolster the development effort. These include a "preliminary software review" following the preliminary program design step, a sequence of "critical software design reviews" during program design, and a "final software acceptance review".

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:

- **Protracted integration and late design breakage.**
- **Late risk resolution.**
- **Requirements-driven functional decomposition.**
- **Adversarial (conflict or opposition) stakeholder relationships.**
- **Focus on documents and review meetings.**

Protracted Integration and Late Design Breakage

For a typical development project that used a waterfall model management process, Figure 1-2 illustrates development progress versus time. Progress is defined as percent coded, that is, demonstrable in its target form.

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 interface ambiguities.
- 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.

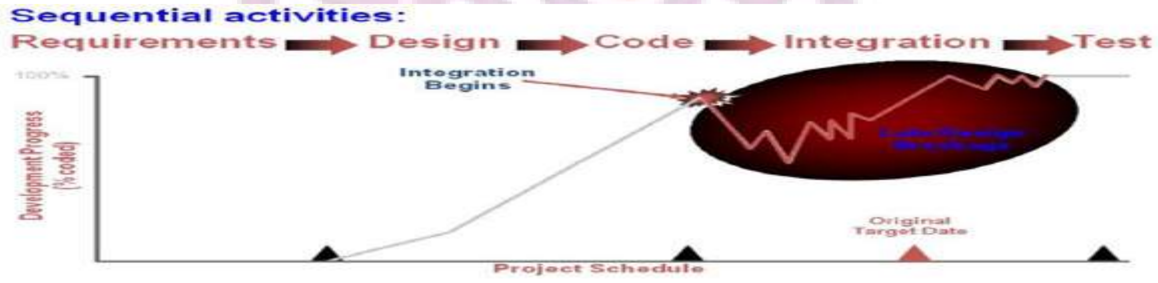
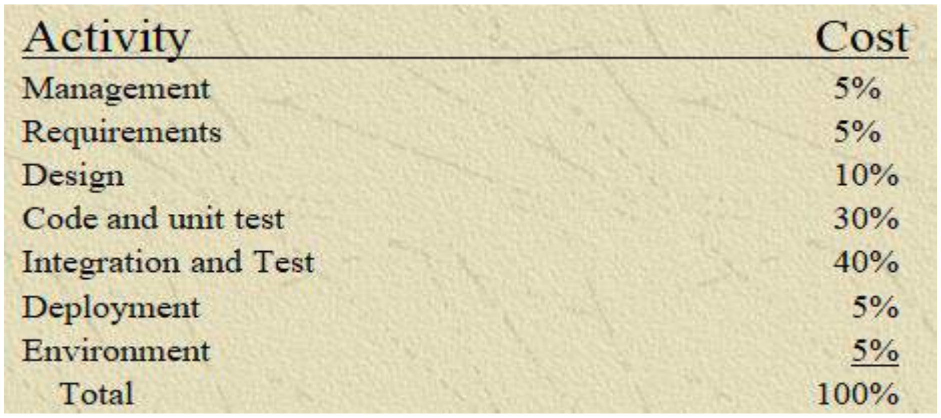


Figure 1-2: Progress profile of a conventional software Project

In the conventional model, the entire system was designed on paper, then implemented all at once, then integrated. Table 1-1 provides a typical profile of cost expenditures across the spectrum of software activities.



Activity	Cost
Management	5%
Requirements	5%
Design	10%
Code and unit test	30%
Integration and Test	40%
Deployment	5%
Environment	5%
Total	100%

Table 1-1: Expenditures of by activity for a conventional software project

Late risk resolution A serious issue associated with the waterfall lifecycle was the lack of early risk resolution. Figure 1.3 illustrates a typical risk profile for conventional waterfall model projects. It includes four distinct periods of risk exposure, where risk is defined as the probability of missing a cost, schedule, feature, or quality goal. Early in the life cycle, as the requirements were being specified, the actual risk exposure was highly unpredictable.

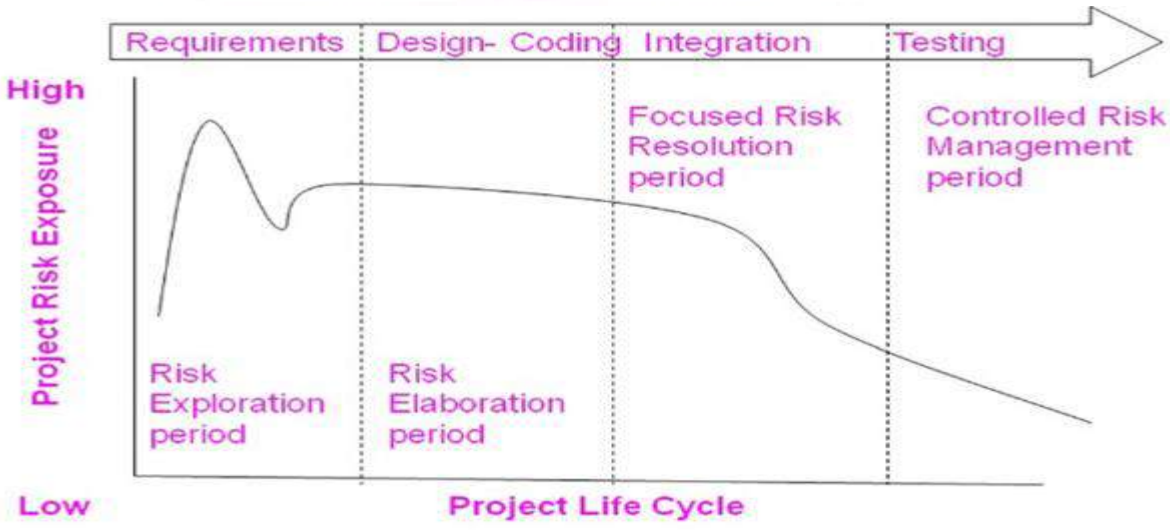


Figure 1.3: risk profile

Requirements-Driven Functional Decomposition: This approach depends on specifying requirements completely and unambiguously before other development activities begin. It naively treats all requirements as equally important, and depends on those requirements remaining constant over the software development life cycle. These conditions rarely occur in the real world. Specification of requirements is a difficult and important part of the software development process**.**

Another property of the conventional approach is that the requirements were typically specified in a functional manner. Built into the classic waterfall process was the
fundamental assumption that the software itself was decomposed into functions; requirements were then allocated to the resulting components. This decomposition was often very different from a decomposition based on object-oriented design and the use of existing components. Figure 1-4 illustrates the result of requirements-driven approaches: a software structure that is organized around the requirements specification structure.

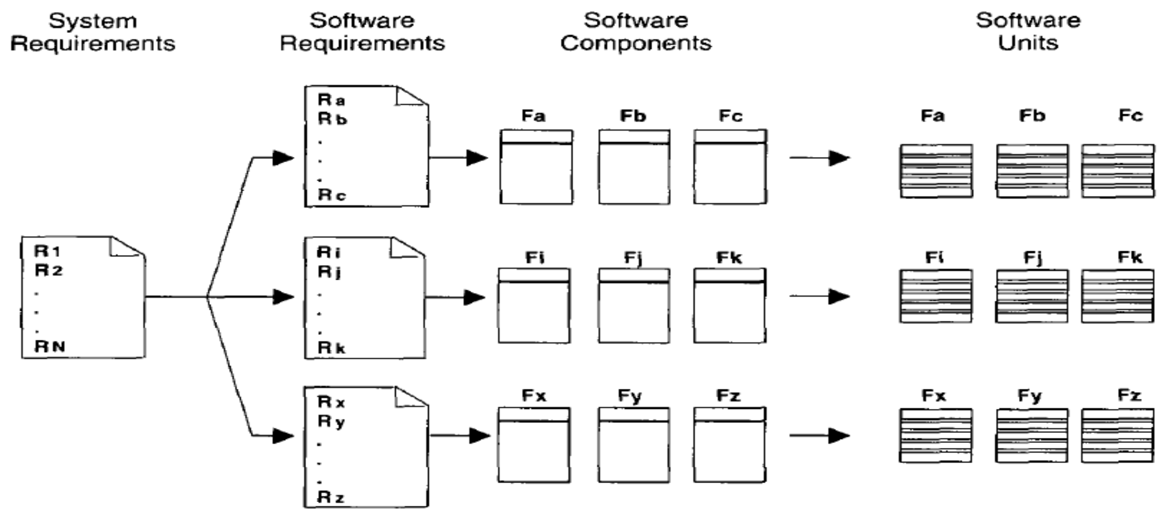


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

Adversarial Stakeholder Relationships:

The conventional process tended to result in adversarial stakeholder relationships, in large part because of the difficulties of requirements 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:

1. The contractor prepared a draft contract-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.
2. The customer was expected to provide comments (typically within 15 to 30 days).
3. 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.

Focus on Documents and Review Meetings:

The conventional process focused on producing various documents that attempted to describe the software product, with insufficient focus on producing tangible increments of the products themselves. Contractors were driven to produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software. Typically,

Presenters and the audience reviewed the simple things that they understood rather than the complex and important issues. Most design reviews therefore resulted in low engineering value and high cost in terms of the effort and schedule involved in their preparation and conduct. They presented merely a facade of progress.

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, 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 differences 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 (i.e., system of sys-tems) cost 9 times as much.
9. Walkthroughs catch 60% of the errors
10. 80% of the contribution comes from 20% of the contributors.

EVOLUTION OF SOFTWARE ECONOMICS

SOFTWARE ECONOMICS:

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

1.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

2.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)

3.The capabilities of software engineering ***personnel***, and particularly their experience with the computer science issues and the applications domain issues of the project

4.The ***environment***, which is made up of the tools and techniques available to support efficient software development and to automate the process

5.The required ***quality*** of the product, including its features, performance, reliability, and adaptability

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

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

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.

Figure 1-5 shows three generations of basic technology advancement in tools, components, and processes. The required levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs (pick your favorite: per SLOC, per function point, per component) realized by an organization.

The three generations of 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.

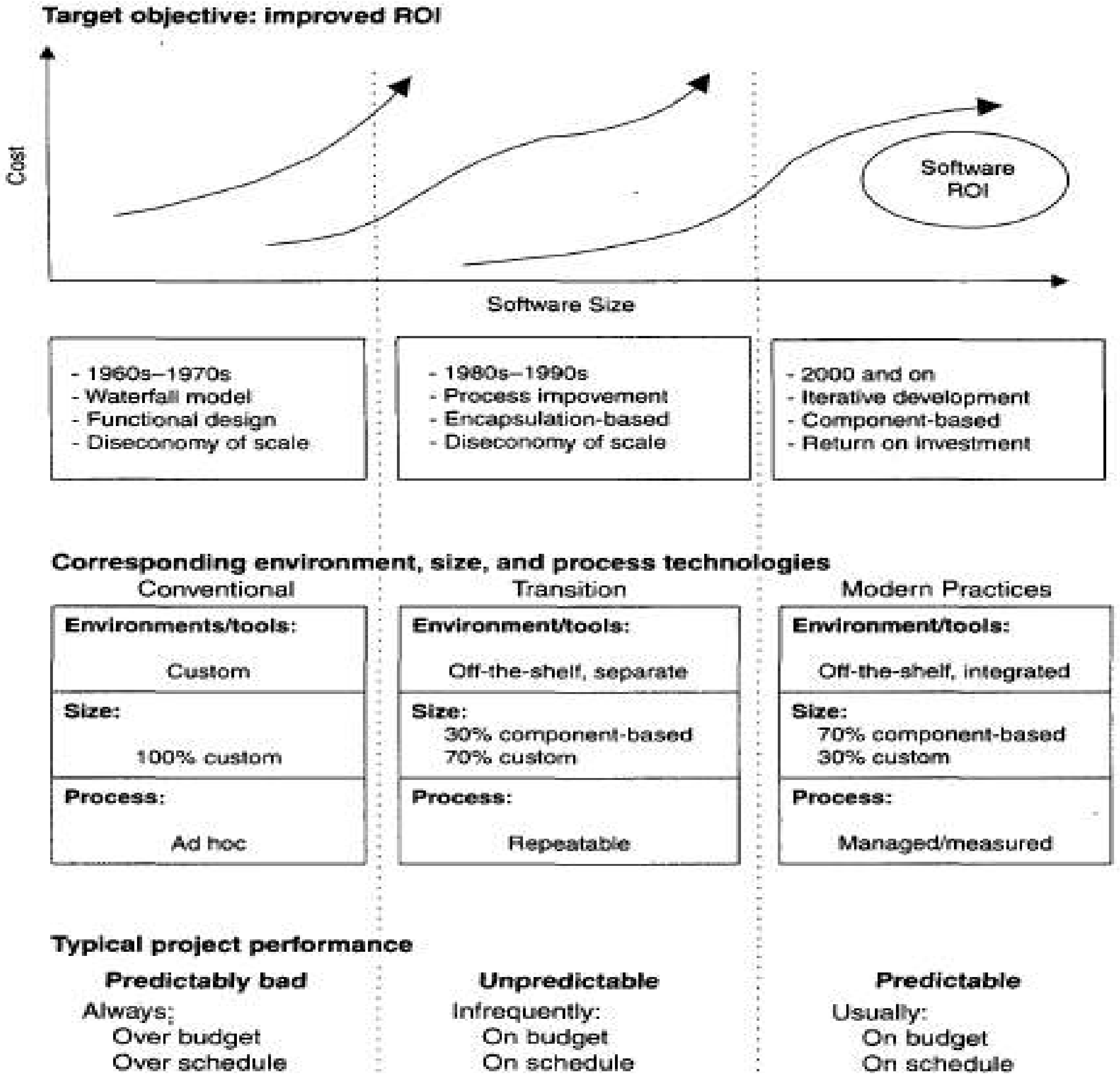


Figure 1-5: Three generations of software economics leading to the target objective

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 1-6 provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains

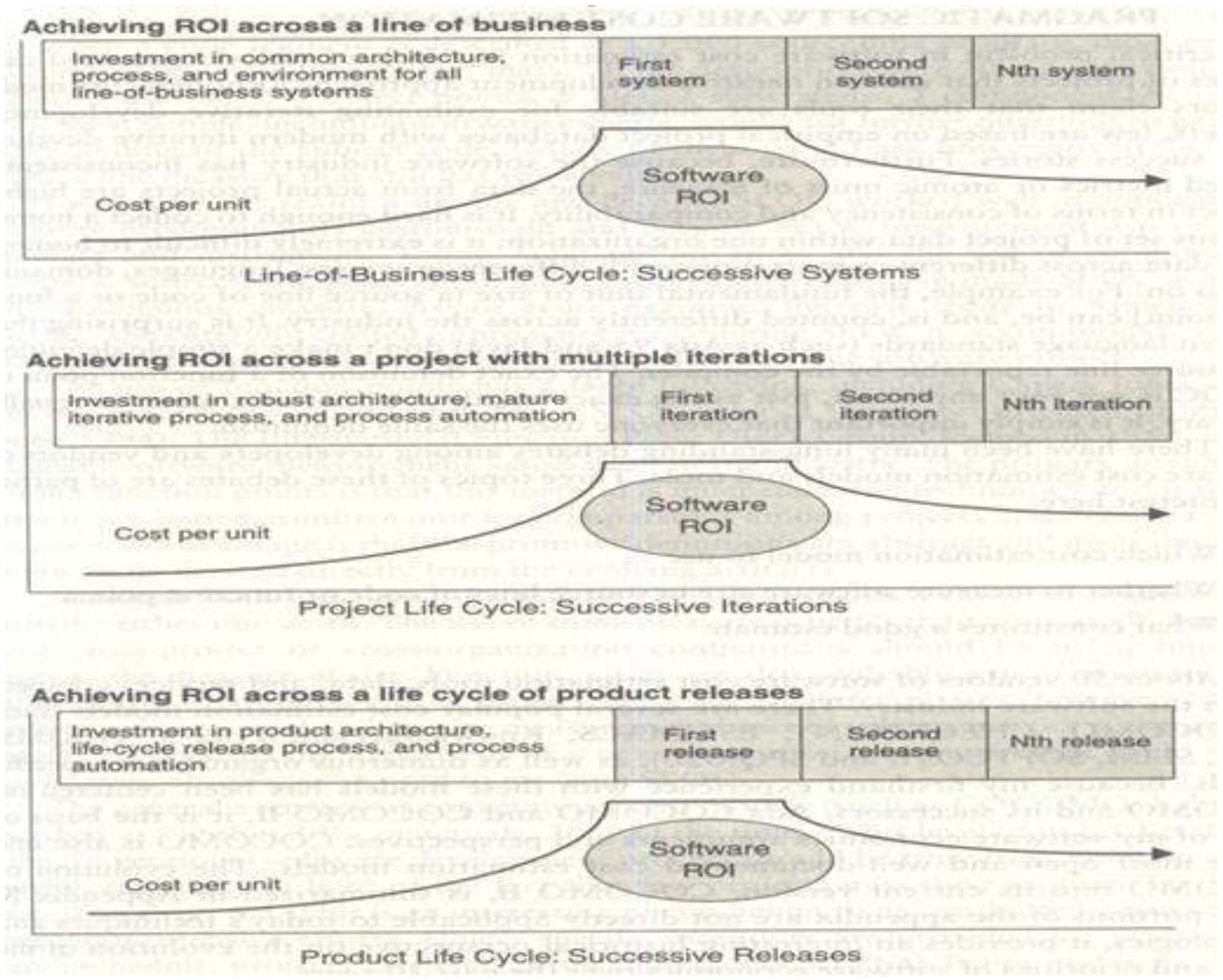

Achieving ROI across a line of business
Investment in common architecture, process, and environment for all line-of-business systems
First system
Second system
Nth system
Software ROI
Cost per unit
Line-of-Business Life Cycle: Successive Systems
Achieving ROI across a project with multiple iterations
Investment in robust architecture, mature iterative process, and process automation
First iteration
Second iteration
Nth iteration
Software ROI
Cost per unit
Project Life Cycle: Successive Iterations
Achieving ROI across a life cycle of product releases
Investment in product architecture, life-cycle release process, and process automation
First release
Second release
Nth release
Software ROI
Cost per unit
Product Life Cycle: Successive Releases

Figure 1-6: 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, KnowledgePlan, 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 as "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 maybe *to* win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.

The process described in Figure 1-7 is not all bad. In fact, it is absolutely neces-sary 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.

A good software cost estimate has the following attributes:

- It is conceived and supported by the project manager, architecture team, development team, and test team 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 similar processes, similar technologies, similar environments, similar quality requirements, and similar people.
- It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Extrapolating from a good estimate, an *ideal* estimate would be derived from a mature cost model with an experience base that reflects multiple similar projects done by the same team with the same mature processes and tools.

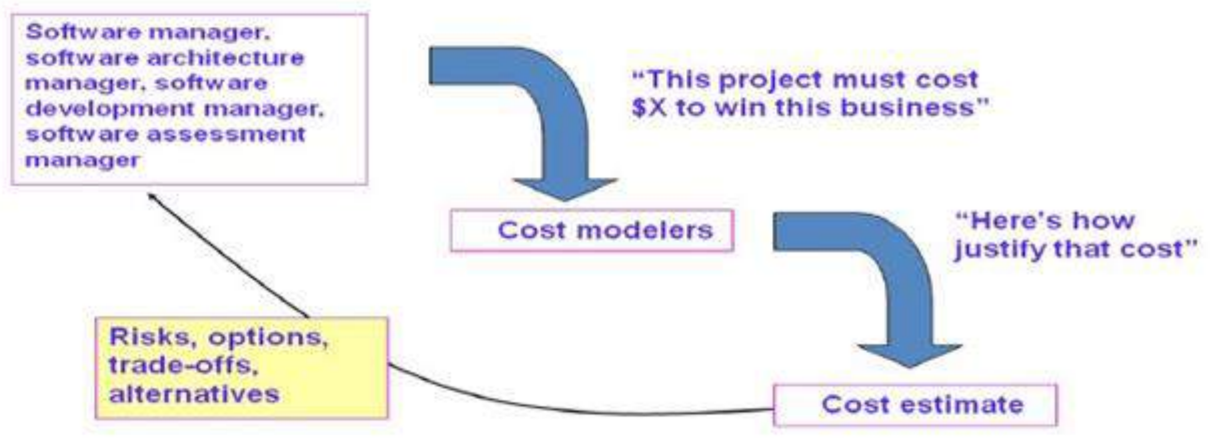

Software manager, software architecture manager, software development manager, software assessment manager
"This project must cost $X to win this business"
Cost modelers
"Here's how justify that cost"
Risks, options, trade-offs, alternatives
Cost estimate

Figure 1-7: The predominant cost estimation process

IMPROVING SOFTWARE ECONOMICS

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. Table 3-1 lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.

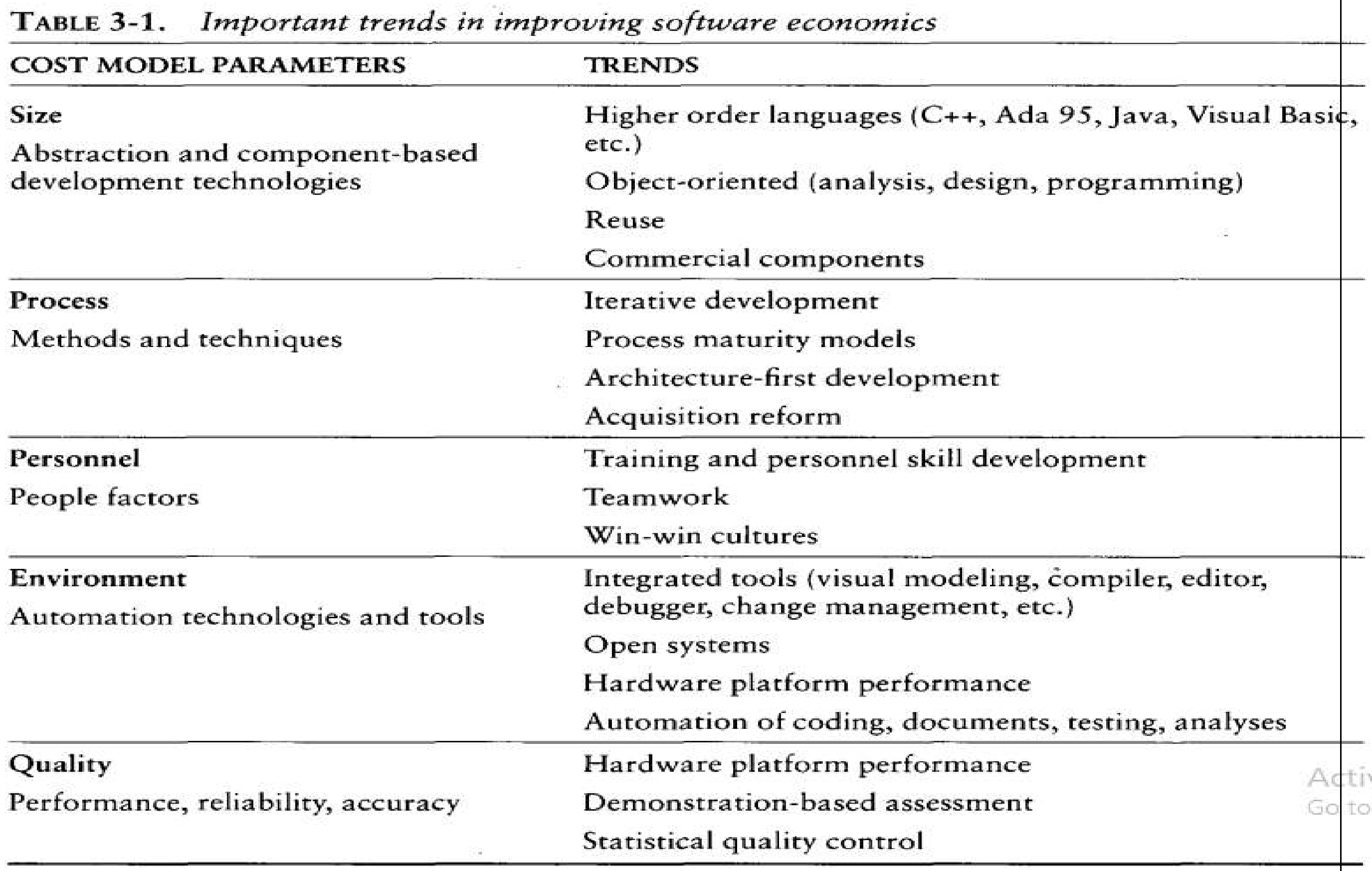


TABLE 3-1. *Important trends in improving software economics*

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

REDUCING SOFTWARE PRODUCT SIZE

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 as the general term for reducing the "source" language size 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 directives (statements).

size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic), 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).
The reduction is defined in terms of human-generated source material. 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. Some of these results are shown in Table 3-2.

Languages expressiveness of some of today's popular languages

LANGUAGES	SLOC per UFP
Assembly	320
C	128
FORTAN77	105

COBOL85	**91**
Ada83	**71**
C++	**56**
Ada95	**55**
Java	**55**
Visual Basic	**35**

Table 3-2

OBJECT-ORIENTED METHODS AND VISUAL MODELING

Object-oriented technology is not germane to most of the software management topics discussed here, and books on object-oriented technology abound. Object-oriented programming languages appear to benefit both software productivity and software quality. The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed.

People like drawing pictures to explain something to others or to themselves. When they do it for software system design, they call these pictures diagrams or diagrammatic models and the very notation for them a modeling language.

These are interesting examples of the interrelationships among the dimensions of improving software economics.

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 five 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. With reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality. Try to treat reuse as a mundane part of achieving a return on investment.

Most truly reusable components of value are transitioned to commercial products supported by organizations with the following characteristics:

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

The cost of developing a reusable component is not trivial. Figure 3-1 examines the economic trade-offs. The steep initial curve illustrates the economic obstacle to developing reusable components.

Reuse is an important discipline that has an impact on the efficiency of all workflows and the quality of most artifacts.

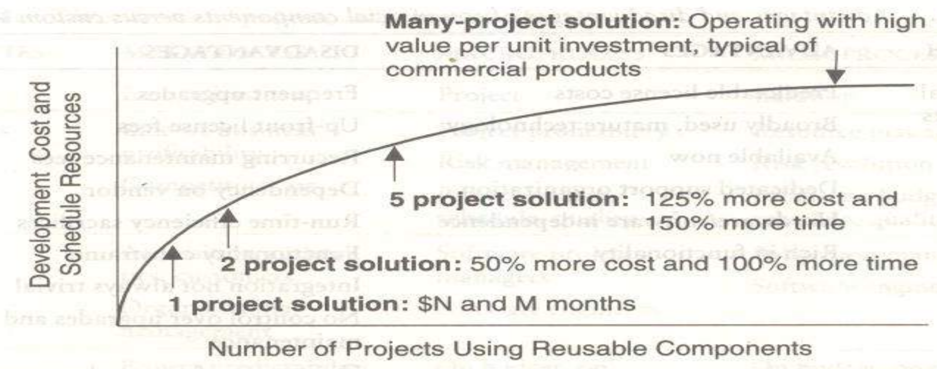


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

COMMERCIAL COMPONENTS

A common approach being pursued today in many 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 straightforward in practice. Table 3-3 identifies some of the advantages and disadvantages of using commercial components.

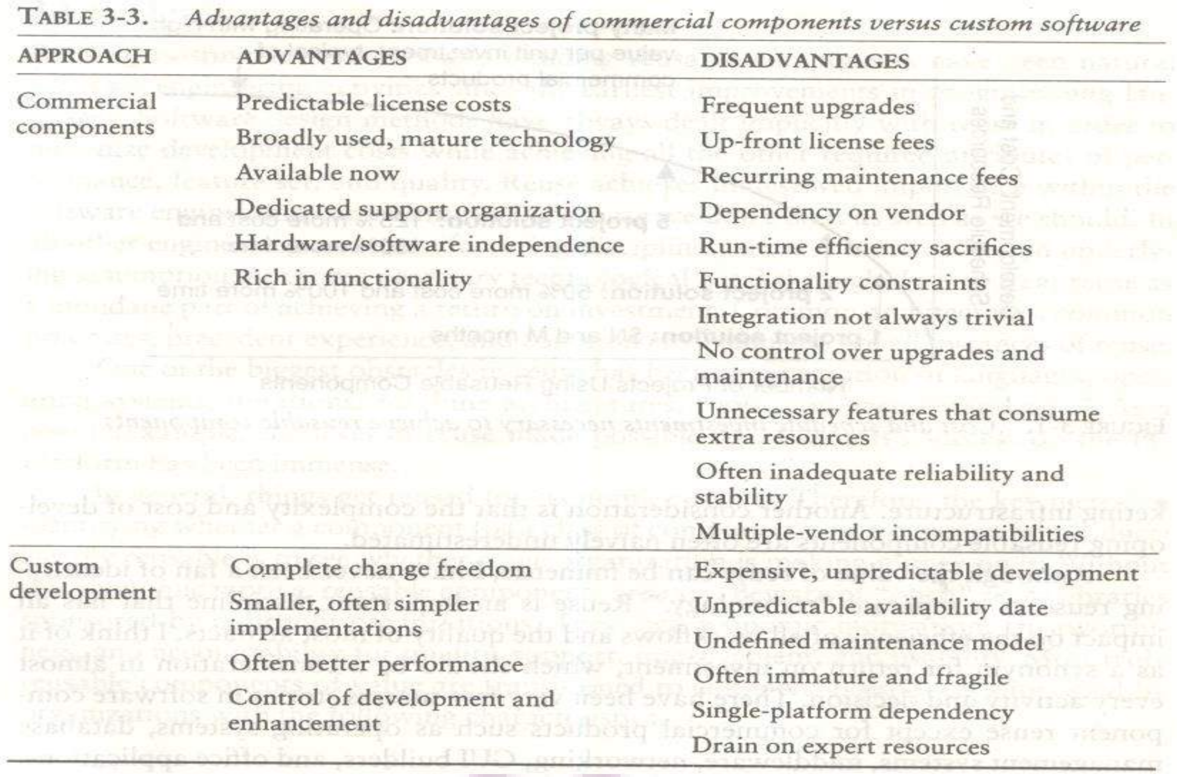


TABLE 3-3. *Advantages and disadvantages of commercial components versus custom software*

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

IMPROVING SOFTWARE PROCESSES

Process is an overloaded term. Three distinct process perspectives are.

- **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.
- **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.
- **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.

Although these three levels of process overlap somewhat, they have different objectives, audiences, metrics, concerns, and time scales as shown in Table 3-4

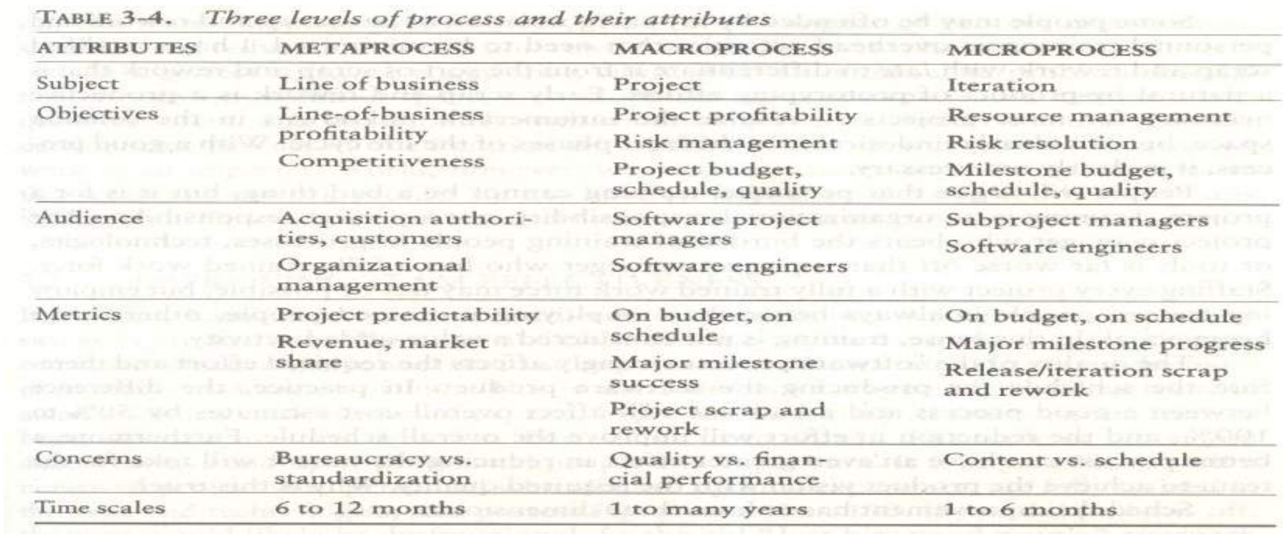


TABLE 3-4. *Three levels of process and their attributes*

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

In a perfect software engineering world with an immaculate problem description, an obvious solution space, a development team of experienced geniuses, adequate resources, and stakeholders with common goals, we could execute a software development process in one iteration with almost no scrap and rework. Because we work in an imperfect world, however, we need to manage engineering activities so that scrap and rework profiles do not have an impact on the win conditions of any stakeholder. This should be the underlying premise for most process improvements.

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:

- A well-managed project can succeed with a nominal engineering team.
- 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.
- A poorly architected system will flounder even with an expert team of builders.

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.

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

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

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

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 for evolving the software engineering artifacts. Above all, configuration management environments provide the foundation for executing and instrument the process. 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 and schedules, and overall productivity using a smaller team.

Round-trip engineering describe the key capability of environments that support iterative development. As we have moved into maintaining different information repositories for the engineering artifacts, we need automation support to ensure efficient and error-free transition of data from one artifact to another. *Forward engineering* is the automation of one engineering artifact from another, more abstract representation. For example, compilers and linkers have provided automated transition of source code into executable code.

Reverse engineering is the generation or modification of a more abstract representation from an existing artifact (for example, creating a .visual design model from a source code representation).

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.

- Requirements analysis and evolution activities consume 40% of life-cycle costs.
- Software design activities have an impact on more than 50% of the resources.
- Coding and unit testing activities consume about 50% of software development effort and schedule.
- Test activities can consume as much as 50% of a project's resources.
- Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.
- Documentation activities can consume more than 30% of project engineering resources.
- 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. Table 3-5 summarizes some dimensions of quality improvement.

Key practices that improve overall software quality include the following:

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

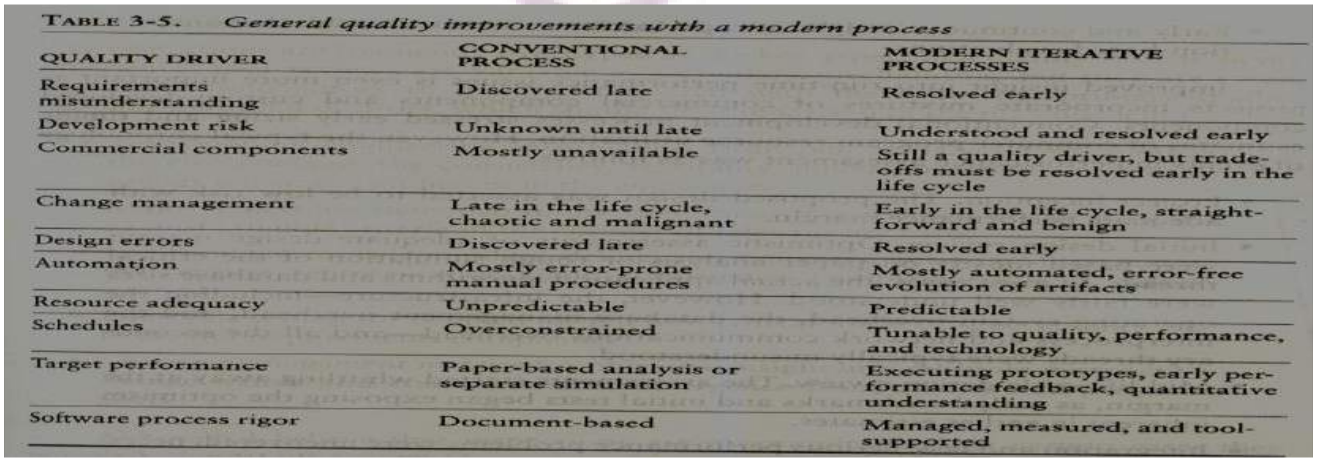


TABLE 3-5. *General quality improvements with a modern process*

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

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

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

- Initial design review. Optimistic assessments of adequate design margin were based mostly on paper analysis or rough simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood.
- Mid-life-cycle design review. The assessments started whittling away at the margin, as early benchmarks and initial tests began exposing the optimism inherent in earlier estimates.
- Integration and test. Serious performance problems were uncovered, necessitating fundamental changes in the architecture. The underlying infrastructure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.

PEER INSPECTIONS: A PRAGMATIC VIEW

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:

- Transitioning engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in the engineering artifacts
- Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases
- Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control
- Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria, and requirements compliance

- Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals

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.

THE OLD WAY AND THE NEW

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

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 demon-strated 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 deter-mine 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 they see, the more they need.** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.

18.**Plan to throw one away.** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.

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

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

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

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

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

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

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

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

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

28.**Realize that software's entropy increases.** Any software system that undergoes continuous change will grow in complexity and 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.

THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

Top 10 principles of modern software management are. (The first five, which are the main themes of my definition of an iterative process, are summarized in Figure 4-1.)

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

2. Establish an *iterative life-cycle process* that confronts risk early. With today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, and 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.

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.

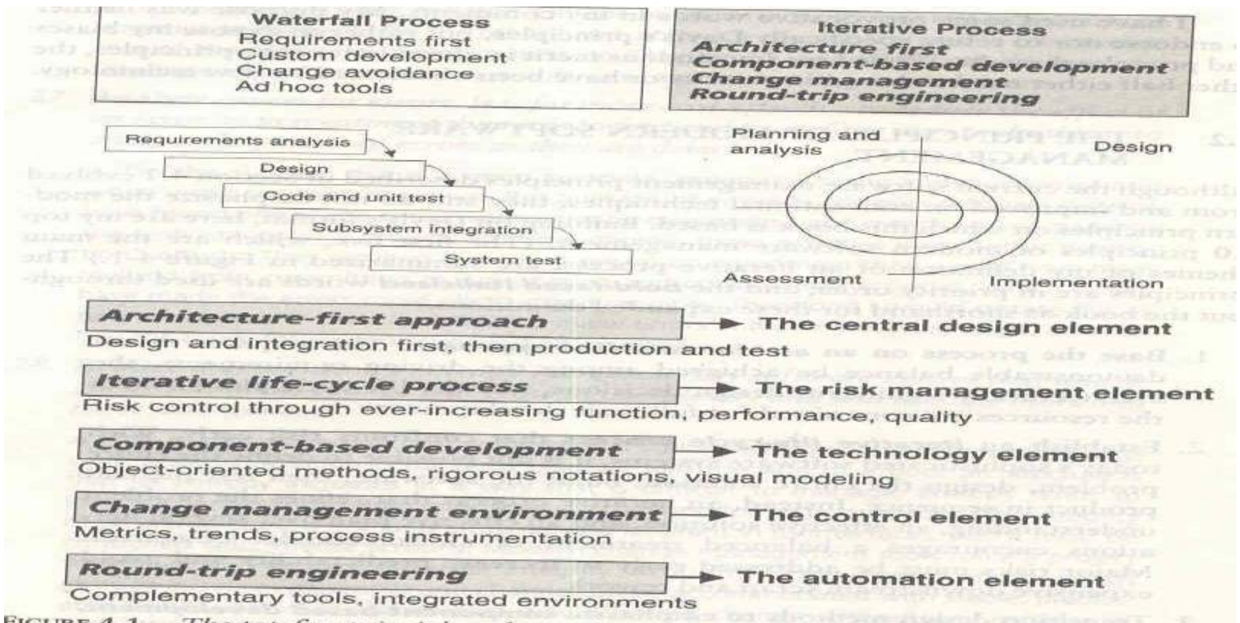

Waterfall Process
Requirements first
Custom development
Change avoidance
Ad hoc tools
Iterative Process
Architecture first
Component-based development
Change management
Round-trip engineering
Requirements analysis
Design
Code and unit test
Subsystem integration
System test
Planning and analysis
Design
Assessment
Implementation
Architecture-first approach
Design and integration first, then production and test
The central design element
Iterative life-cycle process
Risk control through ever-increasing function, performance, quality
The risk management element
Component-based development
Object-oriented methods, rigorous notations, visual modeling
The technology element
Change management environment
Metrics, trends, process instrumentation
The control element
Round-trip engineering
Complementary tools, integrated environments
The automation element

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

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 textual 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 of 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 is suitable for all software developments.

Table 4-1 maps top 10 risks of the conventional process to the key attributes and principles of a modern process

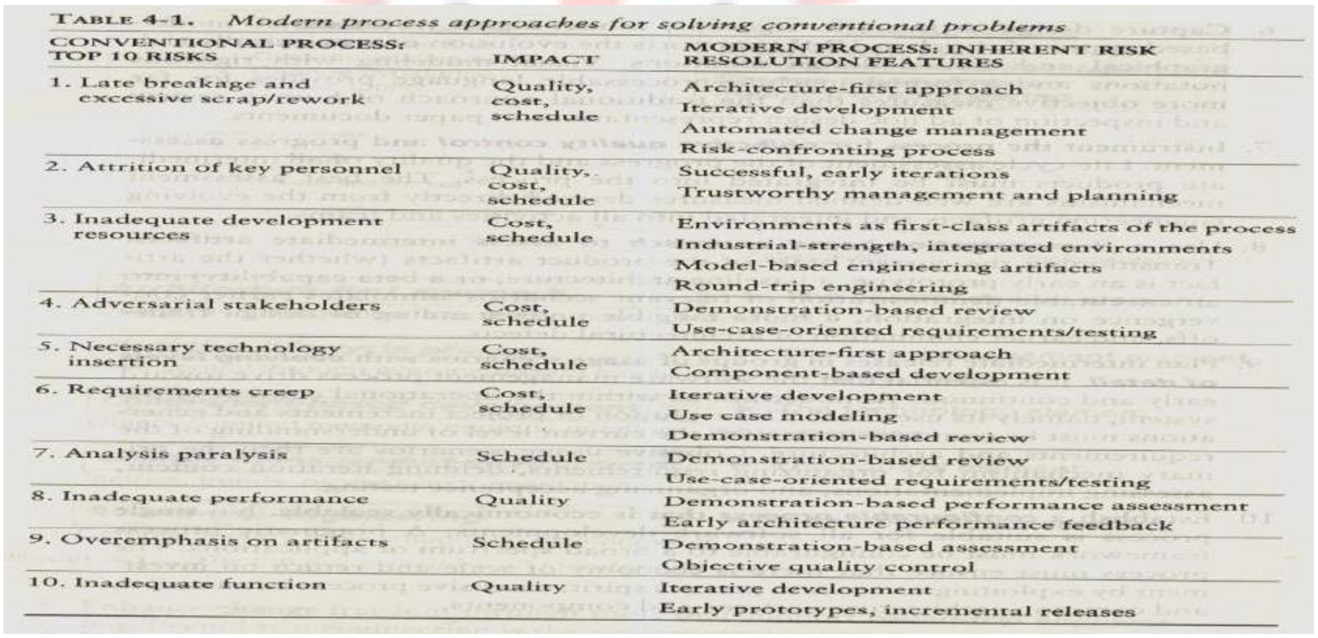


TABLE 4-1. *Modern process approaches for solving conventional problems*

CONVENTIONAL PROCESS: TOP 10 RISKS	IMPACT	MODERN PROCESS: INHERENT RISK RESOLUTION FEATURES
1. Late breakage and excessive scrap/rework	Quality, cost, schedule	Architecture-first approach Iterative development Automated change management Risk-confronting process
2. Attrition of key personnel	Quality, cost, schedule	Successful, early iterations Trustworthy management and planning
3. Inadequate development resources	Cost, schedule	Environments as first-class artifacts of the 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 from 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 model. (Appendix B provides more detail on the COCOMO model) This exponent can range from 1.01 (virtually no diseconomy of scale) to 1.26 (significant diseconomy of scale). The parameters that govern the value of the process exponent are application precedentedness, process flexibility, architecture risk resolution, team cohesion, and software process maturity.

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

- **Application precedentedness.** Domain experience is a critical factor in understanding how to plan and execute a software development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental. This is one of the primary reasons that the software industry has moved to an *iterative life-cycle process.* Early iterations in the life cycle establish precedents from which the product, the process, and the plans can be elab-orated in *evolving levels* of *detail.*
- **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. These changes may be inherent in the problem understanding, the solution space, or the plans. Project artifacts must be supported by efficient *change management*

commensurate with project needs. A *configurable process* that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.

- **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 the infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process.
- **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.
- **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 *objective quality control.*

Life Cycle Phases and Process artifacts:

Introduction:

Characteristic of a successful software development process is the well-defined separation between "research and development" activities and "production" activities. Most unsuccessful projects exhibit one of the following characteristics:

- An overemphasis on research and development
- An overemphasis on production.

Successful modern projects-and even successful projects developed under the conventional process-tend to have a very well-defined project milestone when there is a noticeable transition from a research attitude to a production attitude. Earlier phases focus on achieving functionality. Later phases revolve around achieving a product that can be shipped to a customer, with explicit attention to robustness, performance, and finish.

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

- Evolution of the plans, requirements, and architecture, together with well defined synchronization points
- Risk management and objective measures of progress and quality
- 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

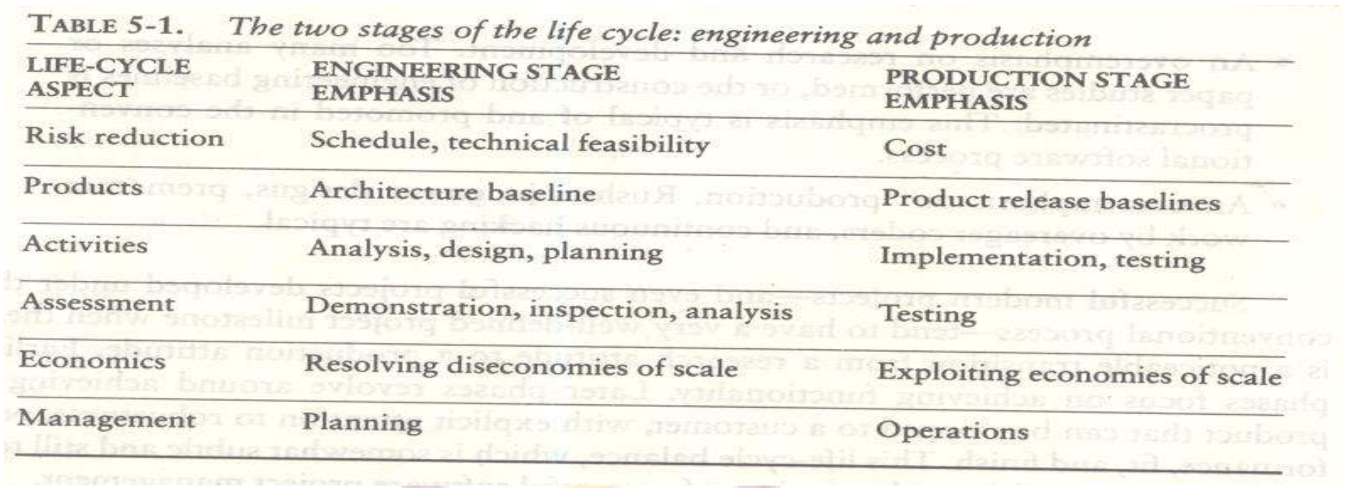


TABLE 5-1. *The two stages of the life cycle: engineering and production*

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

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 5-1

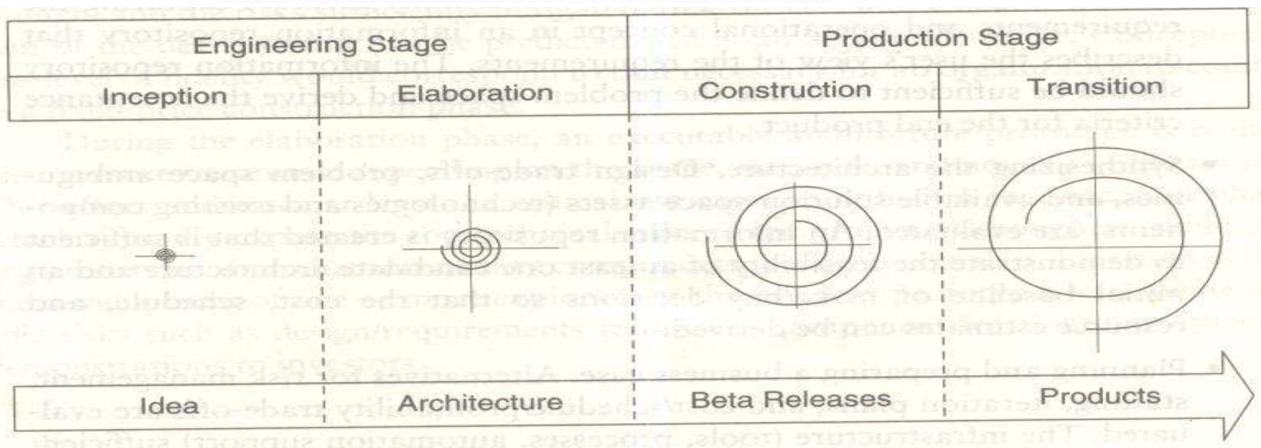


FIGURE 5-1. *The phases of the life-cycle process*

INCEPTION PHASE

The overriding goal of the inception phase is to achieve concurrence among stakeholders on the life-cycle objectives for the project.

PRIMARY OBJECTIVES

- Establishing the project's software scope and boundary conditions, including an 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 scenanos
- Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase)
- Estimating potential risks (sources of unpredictability)

ESSENTIAL ACTIVTIES

- 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? (The primary value of prototyping candidate architecture is to provide a vehicle for understanding the scope and assessing the credibility of the development group in solving the particular technical problem.)

•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 can 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, & risk.

PRIMARY OBJECTIVES

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

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? (Existing defects are not obstacles to achieving the purpose of the next release.)

•Is this product baseline stable enough to be deployed in the user community? (Pending changes are not obstacles to achieving the purpose of the next release.)

•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 users and maintainers

The 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 product baselines as rapidly and cost-effectively as practical

ESSENTIAL ACTIVITIES

•Synchronization and integration of concurrent construction increments into consistent deployment baselines

•Deployment-specific engineering (cutover, commercial packaging and production, sales rollout kit development, field personnel training)

•Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set

EVALUATION CRITERIA

•Is the user satisfied?

•Are actual resource expenditures versus planned expenditures acceptable?

ARTIFACTS OF THE PROCESS

THE ARTIFACT SETS

To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. A*rtifact* 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: management (ad hoc textual formats), requirements (organized text and models of the problem space), design (models of the solution space), implementation (human-readable programming language and associated source files), and deployment (machine-process able languages and associated files). The artifact sets are shown in Figure 6-1.

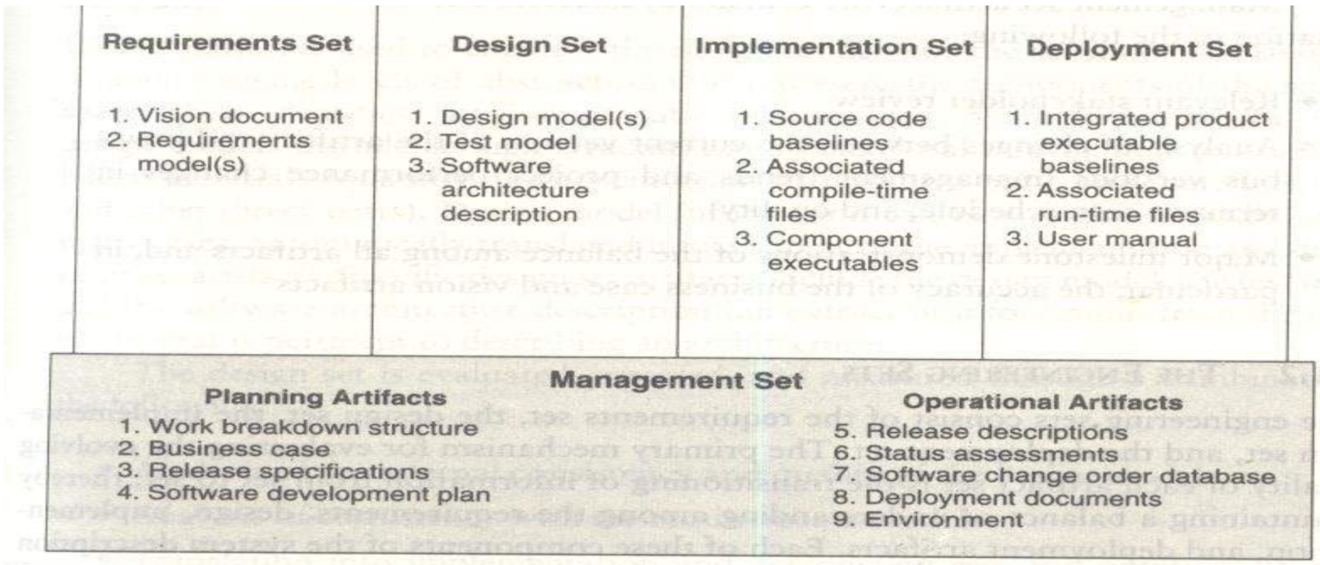


FIGURE 6-1. *Overview of the artifact sets*

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

THE ENGINEERING SETS

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

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

Each artifact set is the predominant development focus of one phase of the life cycle; the other sets take on check and balance roles. As illustrated in Figure 6-2, each phase has a predominant focus: Requirements are the focus of the inception phase; design, the elaboration phase; implementation, the construction phase; and deployment, the transition phase. The management artifacts also evolve, but at a fairly constant level across the life cycle.

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 (operating systems, GUIs, RDBMS, networks, middleware), and installation tools.

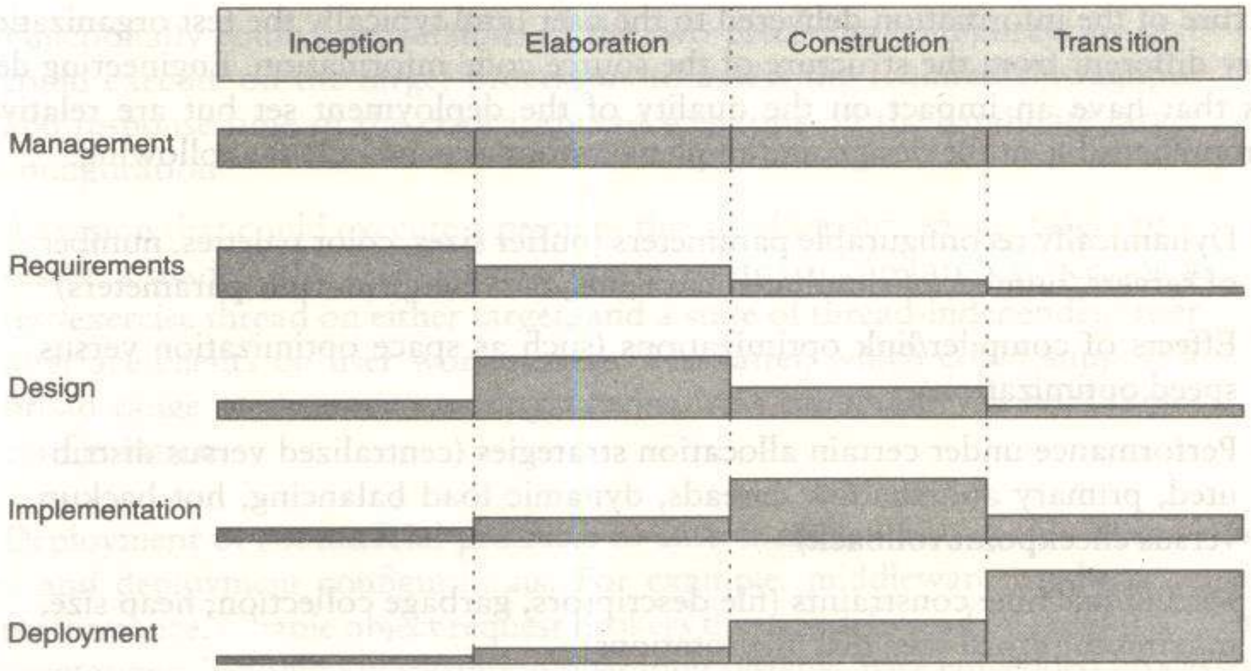


FIGURE 6-2. *Life-cycle focus on artifact sets*

Implementation Set versus Deployment Set

The separation of the implementation set (source code) from the deployment set (executable code) is important because there are very different concerns with each set. The structure of the information delivered to the user (and typically the test organization) is very different from the structure of the source code information. Engineering decisions that have an impact on the quality of the deployment set but are relatively incomprehensible in the design and implementation sets include the following:

•Dynamically reconfigurable parameters (buffer sizes, color palettes, number of servers, number of simultaneous clients, data files, run-time parameters)

•Effects of compiler/link optimizations (such as space optimization versus speed optimization)

•Performance under certain allocation strategies (centralized versus distributed, primary and shadow threads, dynamic load balancing, hot backup versus checkpoint/rollback)

•Virtual machine constraints (file descriptors, garbage collection, heap size, maximum record size, disk file rotations)

•Process-level concurrency issues (deadlock and race conditions)

•Platform-specific differences in performance or behavior

ARTIFACT EVOLUTION OVER THE LIFE CYCLE

Each state of development represents a certain amount of precision in the final system description. Early in the life 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 Figure 6-3.

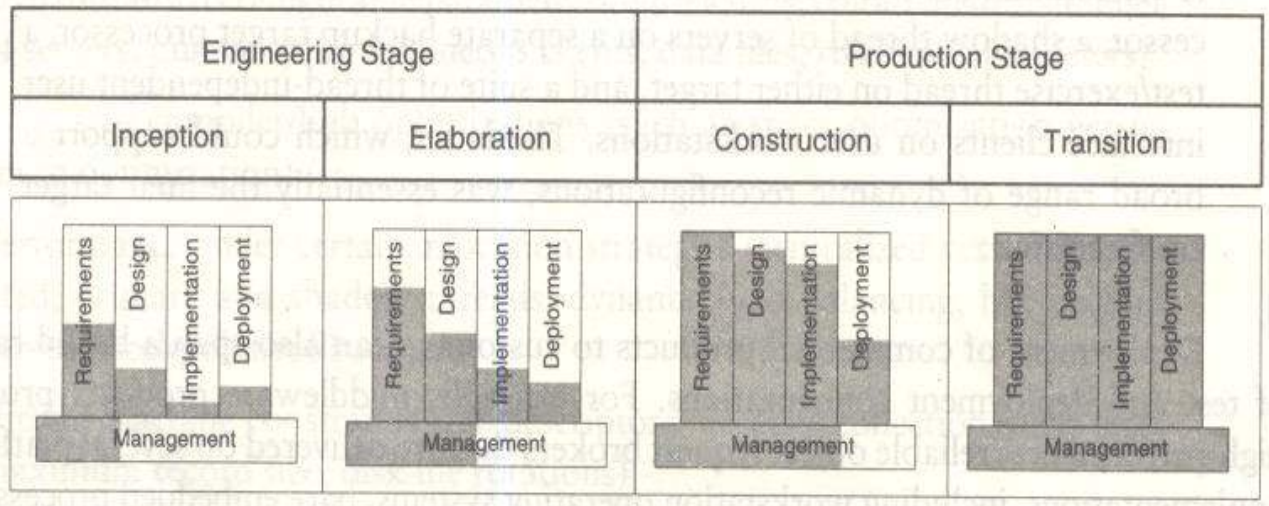


FIGURE 6-3. *Life-cycle evolution of the artifact sets*

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.

Test artifact subsets are highly project-specific, the following example clarifies the relationship between test artifacts and the other artifact sets. Consider a project to perform seismic data processing for the purpose of oil exploration. This system has three fundamental subsystems: (1) a sensor subsystem that captures raw seismic data in real time and delivers these data to (2) a technical operations subsystem that converts raw data into an organized database and manages queries to this database from (3) a display subsystem that allows workstation operators to examine seismic data in human-readable form. Such a system would result in the following test artifacts:

- Management set. The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone. These artifacts are the test plans and test results negotiated among internal project teams. The software change orders capture test results (defects, testability changes, requirements ambiguities, enhancements) and the closure criteria associated with making a discrete change to a baseline.
- 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. The entire requirement set is a test artifact because it is the basis of all assessment activities across the life cycle.
- Design set. A test model for nondeliverable components needed to test the product baselines is captured in the design set. These components include such design set artifacts as a seismic event simulation for creating realistic sensor data; a "virtual operator" that can support unattended, after-hours test cases; specific instrumentation suites for early demonstration of resource usage; transaction rates or response times; and use case test drivers and component stand-alone test drivers.

- Implementation set. Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts. These source files may also include human-readable data files representing certain statically defined data sets that are explicit test source files. Output files from test drivers provide the equivalent of test reports.
- 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.

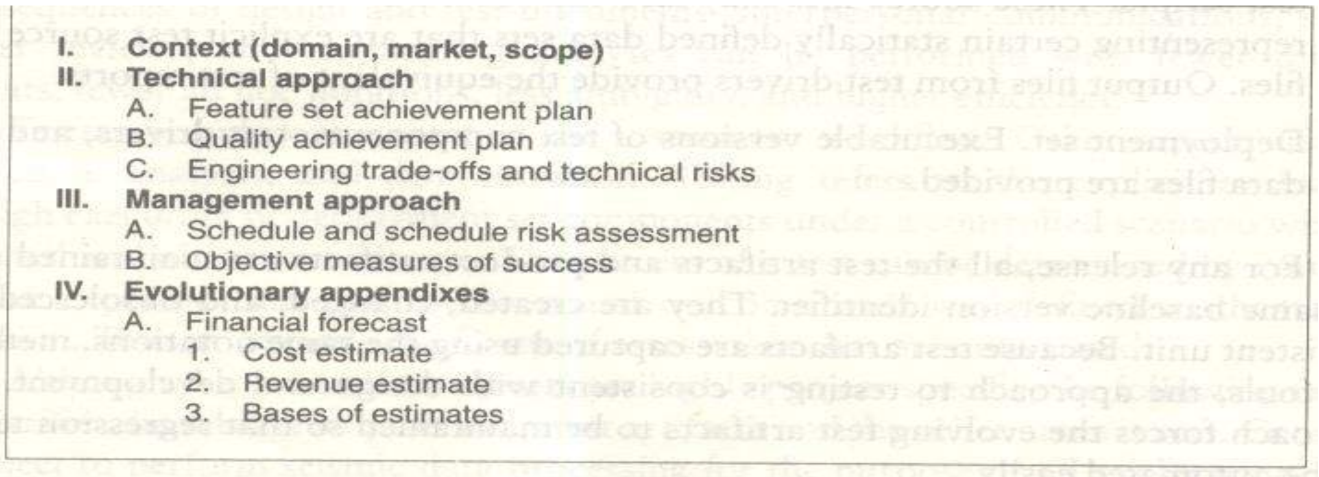
Business Case

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. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progresses. Figure 6-4 provides a default outline for a business case.

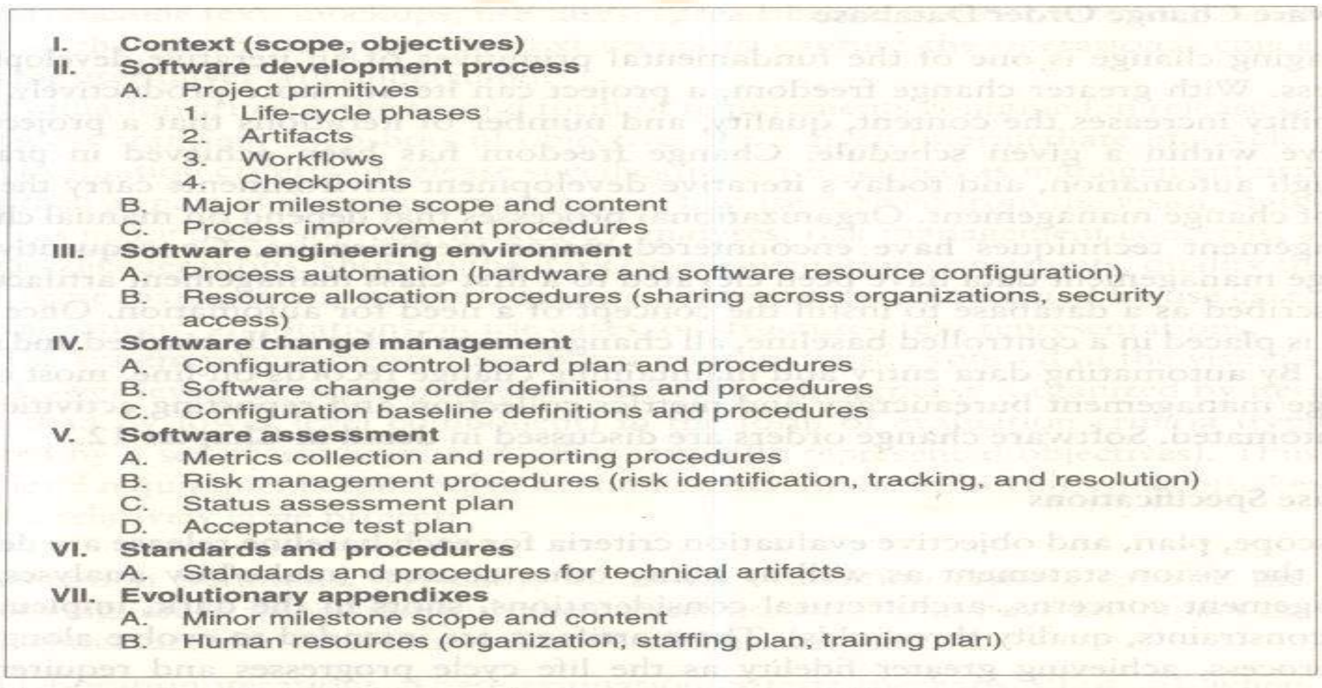
Software Development Plan

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 shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.



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

FIGURE 6-4. *Typical business case outline*



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

FIGURE 6-5. *Typical software development plan 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 man1ger 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. Figure 6-6 provides a default outline for a release specification

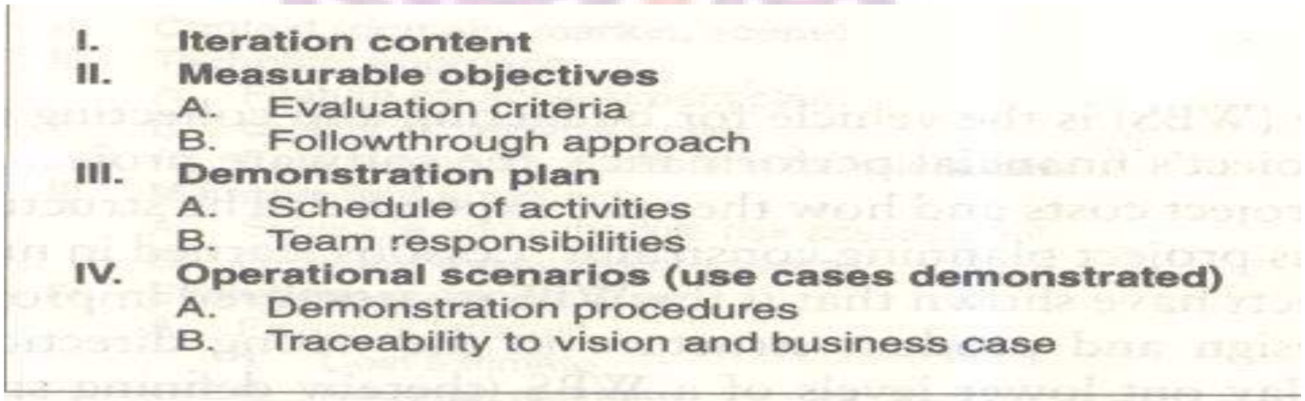

I. Iteration content
II. Measurable objectives
  A. Evaluation criteria
  B. Followthrough approach
III. Demonstration plan
  A. Schedule of activities
  B. Team responsibilities
IV. Operational scenarios (use cases demonstrated)
  A. Demonstration procedures
  B. Traceability to vision and business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

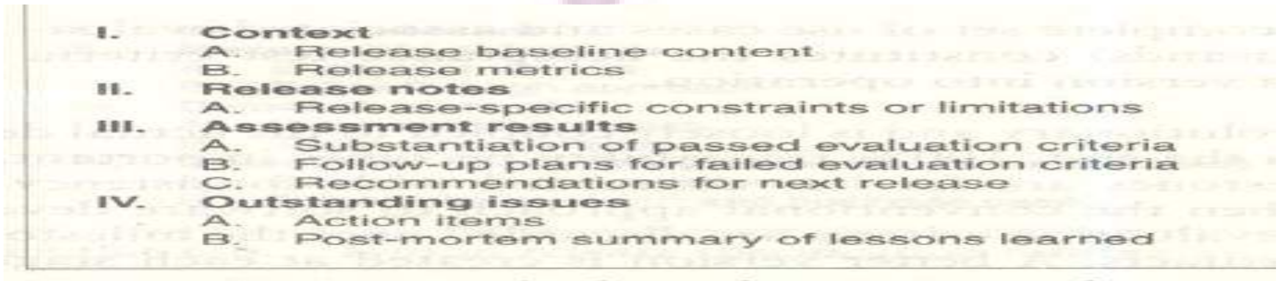
FIGURE 6-6. *Typical release specification outline*

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. Figure 6-7 provides a default outline for a release description.

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



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

FIGURE 6-7. *Typical release description outline*

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 is delivered to a separate maintenance organization, deployment artifacts may include computer system operations manuals, software installation manuals, plans and procedures for cutover (from a legacy system), site surveys, and so forth. For commercial software products, deployment artifacts may include marketing plans, sales rollout kits, and training courses.

Management Artifact Sequences

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

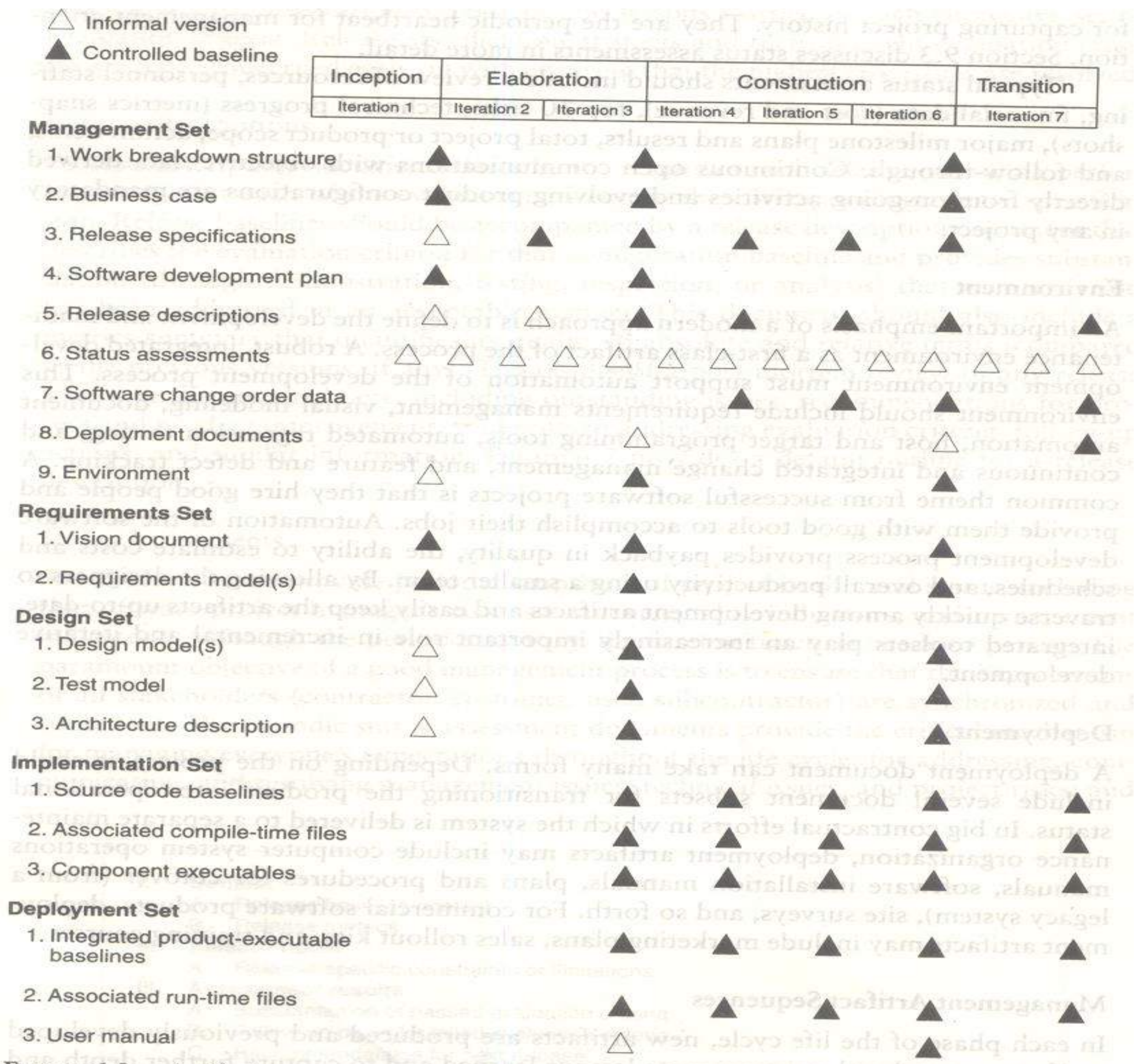


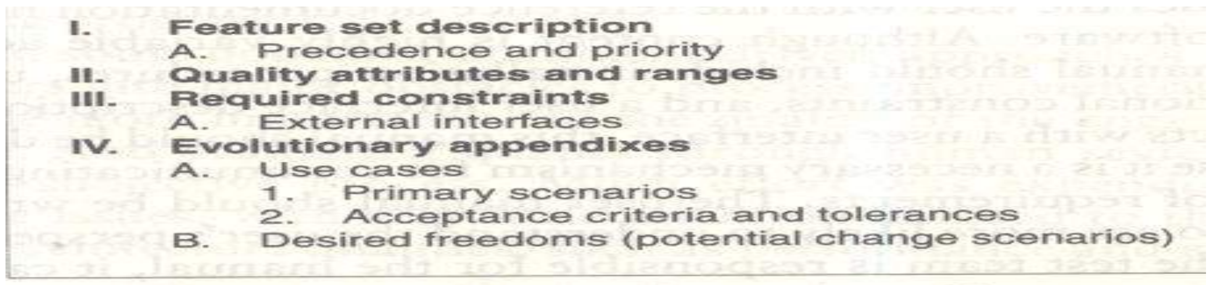
FIGURE 6-8. *Artifact sequences across a typical life cycle*

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 the development organization. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans, and technology. A good vision document should change slowly. Figure 6-9 provides a default outline for a vision document.

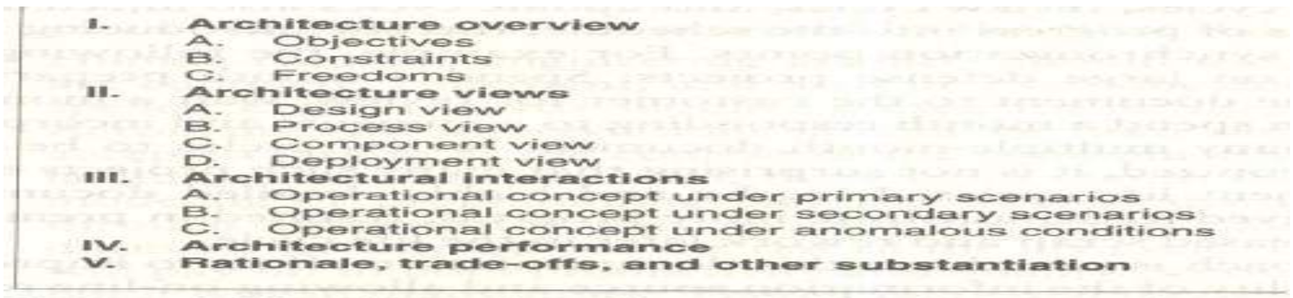


I. Feature set description
   A. Precedence and priority
II. Quality attributes and ranges
III. Required constraints
   A. External interfaces
IV. Evolutionary appendixes
   A. Use cases
      1. Primary scenarios
      2. Acceptance criteria and tolerances
   B. Desired freedoms (potential change scenarios)

FIGURE 6-9. *Typical vision document outline*

Architecture Description

The architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. Figure 6-10 provides a default outline for an architecture description.



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

FIGURE 6-10. *Typical architecture description outline*

Software User Manual

The software user manual provides the user with the reference documentation necessary to support the delivered software. Although content is highly variable across application domains, the user manual should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description, at a minimum. For software products with a user interface, this manual should be developed early in the life cycle because it is a necessary mechanism for communicating and stabilizing an important subset of requirements. The user manual should be written by members of the test team, who are more likely to understand the user's perspective than the development team.

PRAGMATIC ARTIFACTS

- 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."
- 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, organizations are forced to exchange paper documents. Standardized formats (such as UML, spread-sheets, Visual Basic, C++, and Ada 95), visualization tools, and the Web are rapidly making it economically feasible for all stakeholders to exchange information electronically.
- 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.

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

MODEL BASED SOFTWARE ARCHITECTURE

ARCHITECTURE: A MANAGEMENT PERSPECTIVE

The most critical technical product of a software project is its architecture: the infrastructure, control, and data interfaces that permit software components to cooperate as a system and software designers to cooperate efficiently as a team. When the communications media include multiple languages and intergroup 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 different aspects of architecture.

1.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.

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

3.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 from the design set model(s). The architecture description communicates how the intangible concept is realized in the tangible artifacts.

The number of views and the level of detail in each view can vary widely.

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

•Achieving a 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 for 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 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 relationships among the design, component, and deployment views
- •Component: describes the structure of the implementation set
- •Deployment: describes the structure of the deployment set

Figure 7-1 summarizes the artifacts of the design set, including the architecture views and architecture description.

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 class 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 processes and threads of control), interprocess 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 include 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
- Deployment: executable components sufficient to demonstrate the critical use cases and the risk associated with achieving the system qualities

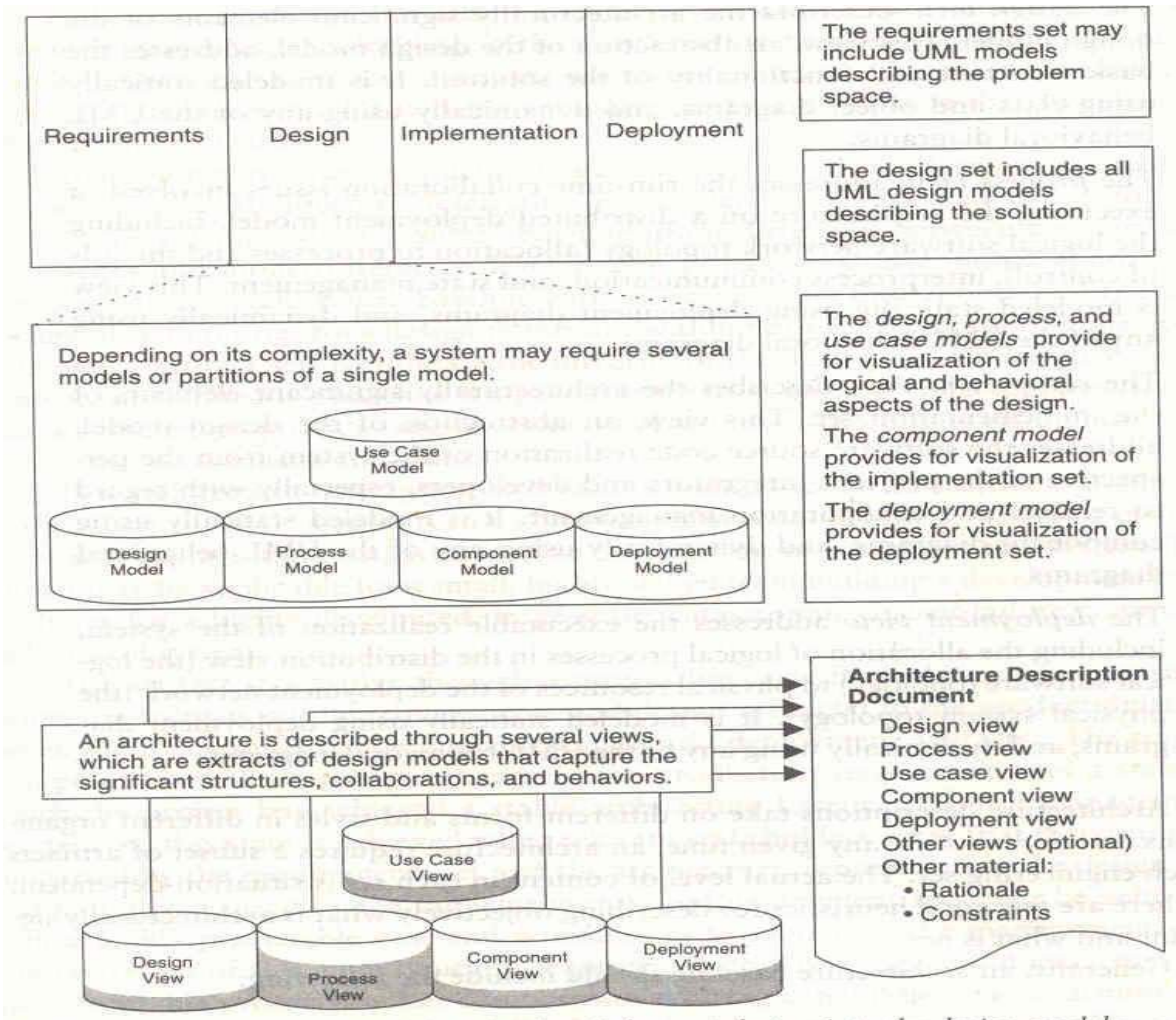

Requirements
Design
Implementation
Deployment
The requirements set may include UML models describing the problem space.
The design set includes all UML design models describing the solution space.
Depending on its complexity, a system may require several models or partitions of a single model.
Use Case Model
Design Model
Process Model
Component Model
Deployment Model
The design, process, and use case models provide for visualization of the logical and behavioral aspects of the design.
The component model provides for visualization of the implementation set.
The deployment model provides for visualization of the deployment set.
An architecture is described through several views, which are extracts of design models that capture the significant structures, collaborations, and behaviors.
Architecture Description Document
Design view
Process view
Use case view
Component view
Deployment view
Other views (optional)
Other material:
• Rationale
• Constraints
Use Case View
Design View
Process View
Component View
Deployment View

FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

UNIT - III

Workflows and Checkpoints of process, Software process workflows, Iteration workflows, Major milestones, minor milestones, periodic status assessments. Process Planning Work breakdown structures, Planning guidelines, cost and schedule estimating process, iteration planning process, Pragmatic planning.

SOFTWARE MANAGEMENT PROCESS FRAMEWORK:

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 with 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 design artifacts
5. Implementation workflow: programming the components and 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

Four basic key principles of the modern process frame work:

Architecture-first approach: implementing and testing the architecture must precede full-scale development and testing and must precede the downstream focus on completeness and quality of the product features.

Iterative life-cycle process: the activities and artifacts of any given workflow may require more than one pass to achieve adequate results.

Roundtrip engineering: Raising the environment activities to a first-class workflow is critical; the environment is the tangible embodiment of the project's process and notations for producing the artifacts.

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

Explain in detail about the iteration workflows of the software process?

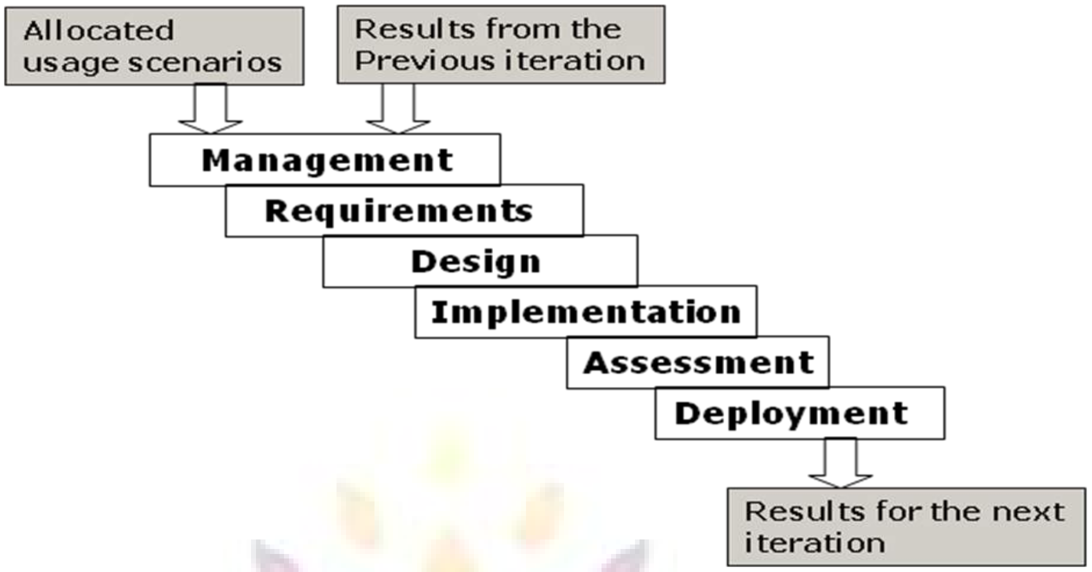
Iteration consists of sequential set of activities in various proportions, depending on where the iteration is located in the development cycle. Each iteration is defined in terms of a se t of allocated usage scenarios. The components needed to implement all selected scenarios are developed and integrated with the results of previous iterations. An individual iteration's workflow illustrated in 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 the demonstrated at the end of the iteration and their evaluation criteria.

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 evolution criteria allocated to this iteration.

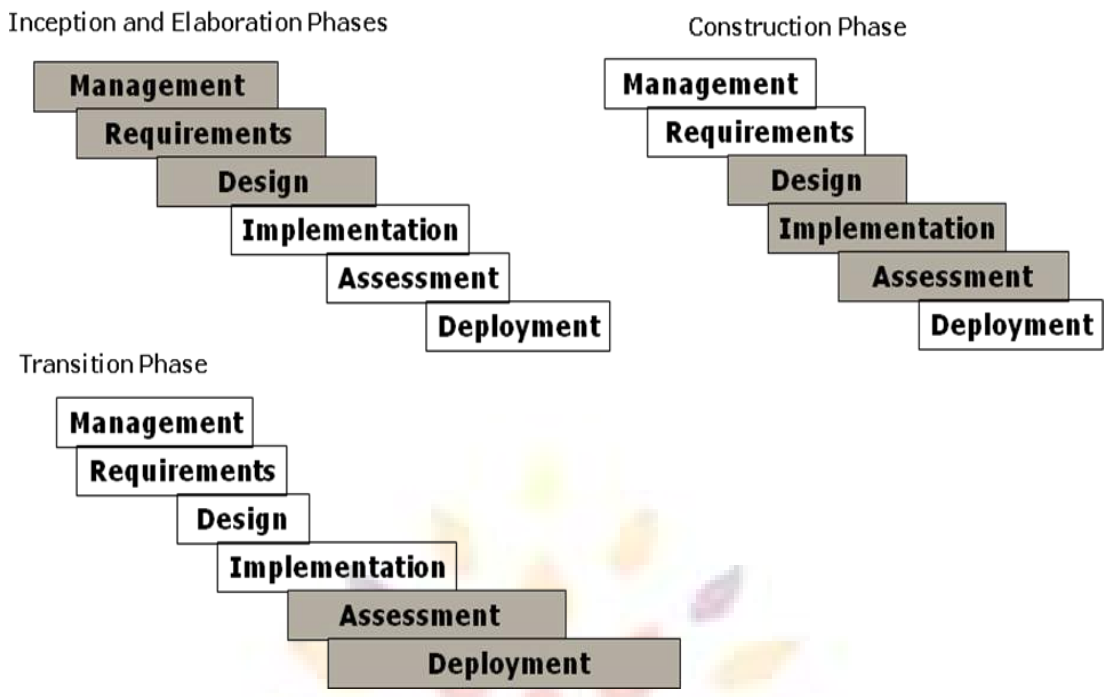

Allocated usage scenarios
Results from the Previous iteration
Management
Requirements
Design
Implementation
Assessment
Deployment
Results for the next iteration

Implementation: developing any new components, and enhancing or modifying any existing components, to demonstrate the evolution criteria allocated to this iteration

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 determining whether it should be performed before deployment of this release or allocated to the next release.

Deployment: transitioning the released either to an external organization or to internal closure by conducting a post mortem so that lessons learned can be captured and reflected in the next iteration.

The following is an example of a simple development life cycle, illustrates the difference between iterations and increments. This example also illustrates a typical build sequence from the perspective of an abstract layered architecture.



Iteration emphasis across the life cycle

CHECK POINTS OF THE PROCESS

It is important to have visible milestones in the life cycle, where various stakeholders meet to discuss progress and planes.

The purpose of this events is to:

Synchronize stakeholder expectations and achieve concurrence on the requirements, the design, and the plan.

Synchronize related artifacts into a consistent and balanced state.

Synchronize related artifacts into a consistent and balanced state Identify the important risks, issues, and out-of-tolerance conditions.

Perform a global assessment for the whole life-cycle.

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

<u>Major milestones</u> –provide visibility to system wide issues, synchronize the management and engineering perspectives and verify that the aims of the phase have been achieved.

<u>Minor milestones</u> – iteration-focused events, conducted to review the content of iteration in detail and to authorize continued work.

<u>Status assessments</u> – periodic events provide management with frequent and regular insight into the progress being made.

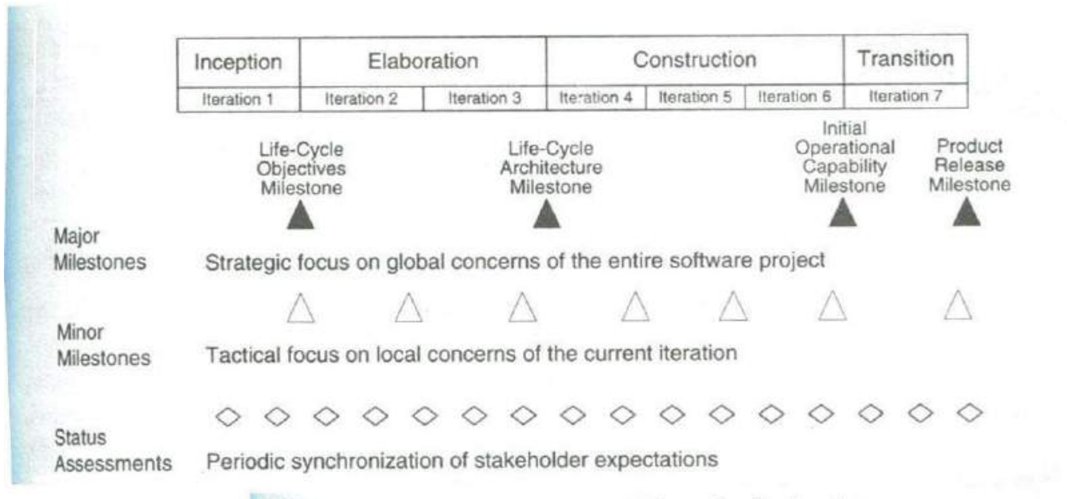

Inception
Elaboration
Construction
Transition
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Life-Cycle Objectives Milestone
Life-Cycle Architecture Milestone
Initial Operational Capability Milestone
Product Release Milestone
Major Milestones
Strategic focus on global concerns of the entire software project
Minor Milestones
Tactical focus on local concerns of the current iteration
Status Assessments
Periodic synchronization of stakeholder expectations

FIGURE 9-1. *A typical sequence of life-cycle checkpoints*

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 the 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. Different stakeholders have very different concerns:

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

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

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

Developers: sufficiency of requirements detail and usage scenario descriptions, frameworks for component selection of development, resolution of development risk, 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 sales and marketing teams.

The milestones may be conducted as one continuous meeting of all concerned parties or incrementally through mostly on-line review of the various artifacts. There are considerable differences in the levels of ceremony for these events depending on several factors.

The essence of each major milestone is to ensure that the requirements understanding, the life-cycle plans, and the product's form, function, and quality are evolving in balanced levels of detail and to ensure consistency among the various artifacts. The following table summarizes the balance of information across the major milestones.

MINOR MILESTONES

All iterations are not created equal. An iteration can take on very different forms and priorities, depending on where the project is in the life cycle. Early iterations focus on analysis and design with substantial elements of discovery, experimentation, and risk assessment. Later iterations focus much more on completeness, consistency, usability, and change management.

Iteration readiness review: this informal milestone is conducted at the start of each iteration to review the detailed iteration plan the evolution criteria that have been allocated to this iteration.

Iteration Assessment review: this informal milestone is conducted at the end of each iteration to assess the degree of which the iteration achieved its objectives and satisfied its evaluation criteria, to review iteration achieved its objectives and satisfied its evaluation criteria, to review iteration results, to review qualification test results, to determine the amount of rework to be done, and to review the impact of the iteration results on the plan for subsequent iterations.

PERIODIC STATUS ASSESSMENTS

Periodic stats assessments are management reviews conducted at regular intervals to address progress and quality indicators, ensure continuous attention to project dynamics, and maintain open communications among all stakeholders.

Status assessments provide the following:

A mechanism for openly addressing, communicating, and resolving management issues, technical issues, and project risks

Objective data directly from on-going activities and evolving product configurations

A mechanism for disseminating process, progress quality trends, practices and experience information to and from all stakeholders in an open forum.

The default content of periodic status assessments should include the topics identified in the following ta

ITERATIVE PROCESS PLANNING

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 delineation of all significant work A clear task decomposition for assignment of responsibilities

A framework for scheduling, budgeting, and expenditure tracking.

The development of a work breakdown structure is dependent on the project management style, organizational culture, customer preference, financial constraints and several other hard- to-define parameters.

Conventional WBS Issues:

Conventional WBS frequently suffer from three fundamental flaws:

Conventional WBS are prematurely structured around the productdesign:

Once this structure is ingrained in the WBS and then allocated to responsible managers with budgets, schedules and expected deliverables, a concrete planning foundation has been set that is difficult and expensive to change.

Conventional WBS are prematurely decomposed, planned, and budgeted in wither too much or too little detail:

Large software projects tend to be over planned and small projects tend to be under planned. The WBS shown in the above figure is overly simplistic for most large-scale systems, where size or more levels of WBS elements are commonplace.

Conventional WBS are project-specific, and cross-project comparisons are usually difficult or impossible:

Most organizations allow individual projects to define their own project-specific structure tailored to the project manager's style, the customer's demands, or other project-specific preferences.

It is extremely difficult to compare plans, financial data, schedule data, organizational efficiencies, cost trends, productivity tends, or quality tends across multiple projects.

Some of the following simple questions, which are critical to any organizational process improvement program, cannot be answered by most project teams that use conventional WBS.

What is the ratio of productive activities to overhead activities?

What is the percentage of effort expanded in rework activities?

What is the percentage of cost expended in software capital equipment?

What is the ration of productive testing versus integration?

What is the cost of release?

Evolutionary Work Breakdown Structures:

An e … process … ndation

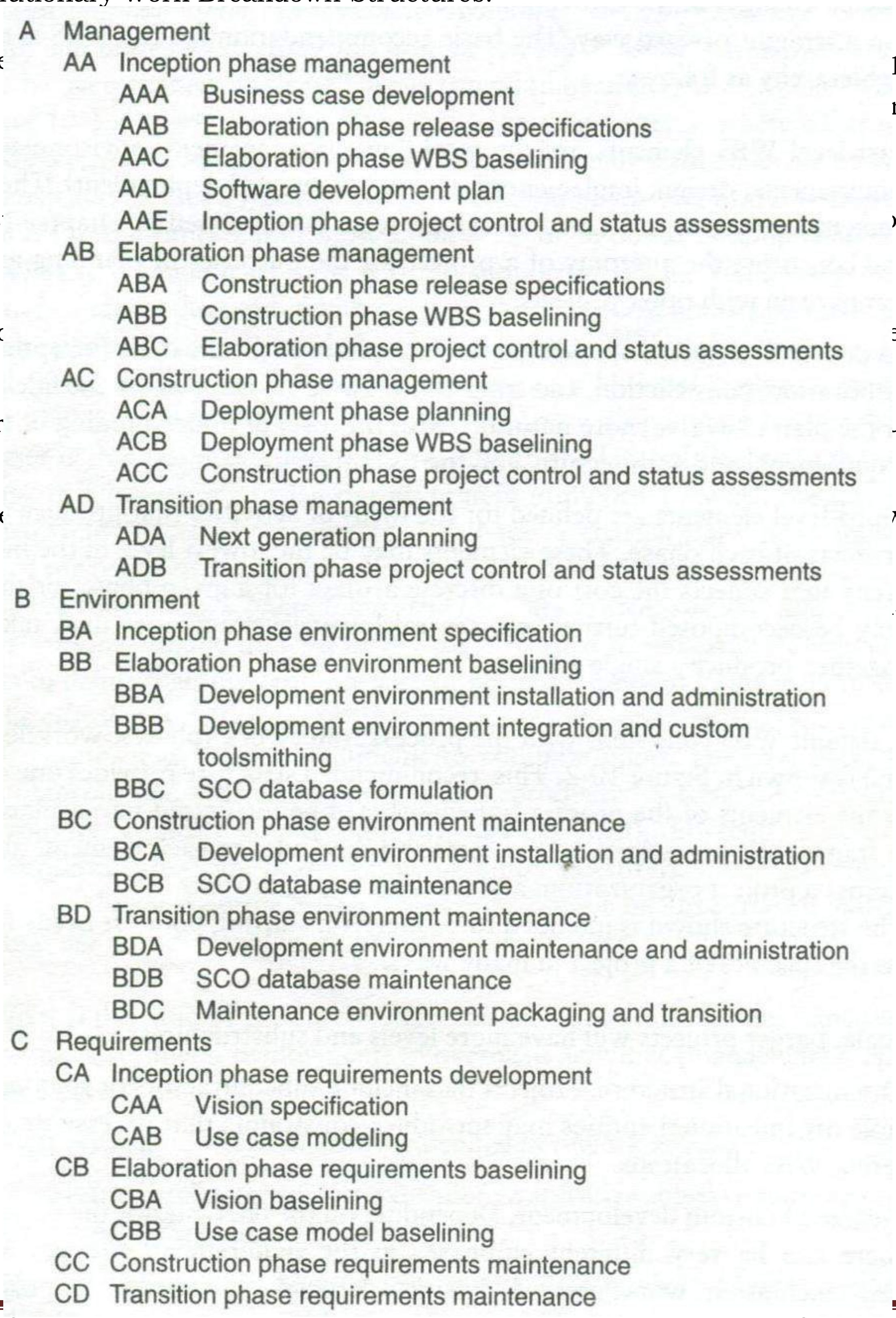
Firs … onment,

Seco … eptions,

Thir … uce the

A de … ws, and

The … s to be



- 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

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. The first guideline, detailed in Table 10-1, prescribes a default allocation of costs among the first-level WBS elements. The second guideline, detailed in Table 10-25, prescribes allocation of effort and schedule across the lifecycle phases.

Web budgeting defaults

First Level WBS Element	**Default Budget**
Management	10%
Environment	10%
Requirement	10%
Design	15%
Implementation	25%
Assessment	25%
Deployment	5%
Total	100%

Table 10-2 Default distributions of effort and schedule by phase

Domain	**Incepti**	**Elaboratio**	**Constructio**	**Transitio**

	on	n	n	n
Effor t	5%	20%	65%	10 %
Sche dule	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.
  - The software project manager partitions the estimate for the effort into top-level WBS using guidelines such as those in Table 10-1.
  - 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, 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:

1. The lowest level WBS elements are elaborated into detailed tasks
2. Estimates are combined and integrated into higher level budgets and milestones.
3. Comparisons are made with the top-down budgets and schedule milestones.

Engineering Stage		**Production Stage**	
Inception	Elaboration	Construction	Transition
Engineering stage planning emphasis:		**Production 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		▪ 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	

THE ITERATION PALNNING PROCESS

- Planning is concerned with defining the actual sequence of intermediate results. An evolutionary build plan is important because there are always adjustments in build content and schedule as early conjecture evolves into well-understood project circumstance. 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. This framework includes existing components, commercial components and custom prototypes sufficient to demonstrate candidate architecture and sufficient requirements understanding to establish a credible business case, vision and software development plan

- **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 (for example, the peak transaction throughput or peak loan scenario) and (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.

UNIT 4:

PROJECT ORGANIZATIONS

Line-of- business organizations, project organizations, evolution of organizations, process automation. Project Control and process instrumentation, The seven-core metrics, management indicators, quality indicators, life-cycle expectations, Pragmatic software metrics, metrics automation.

PROJECT ORGANIZATION AND RESPONSIBILITIES:

INTRODUCTION: 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.

LINES-OF-BUSINESS ORGANIZATIONS: Figure 11-1 maps roles and responsibilities to a default line-of-business organization. This structure can be tailored to specific circumstances.

- 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

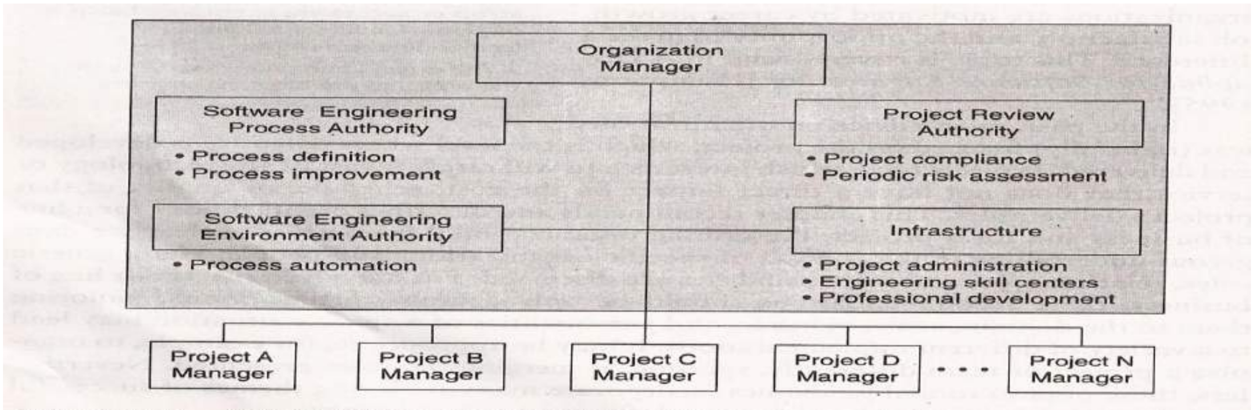


FIGURE 11-1. *Default roles in a software line-of-business organization*

The line of business organization consists of four component teams.

- SOFTWARE ENGINEERING PROCESS AUTHORITY
- The software engineering process authority (SEPA) is responsible for exchanging the information and project guidance to or from the project practitioners.
- PROJECT REVIEW AUTHORITY
- The project review Authority (PRA) is responsible for reviewing the financial performance, customer commitments, risks and accomplishments, adherence to organizational policies by the customer etc.
- SOFTWARE ENGINEERING ENVIRONMENT AUTHORITY

The software Engineering Environment Authority (SEEA) deals with the maintenance or organizations standard environment, training projects and process automation

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

- shows a 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 are 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. *The assessment team* is separate form development

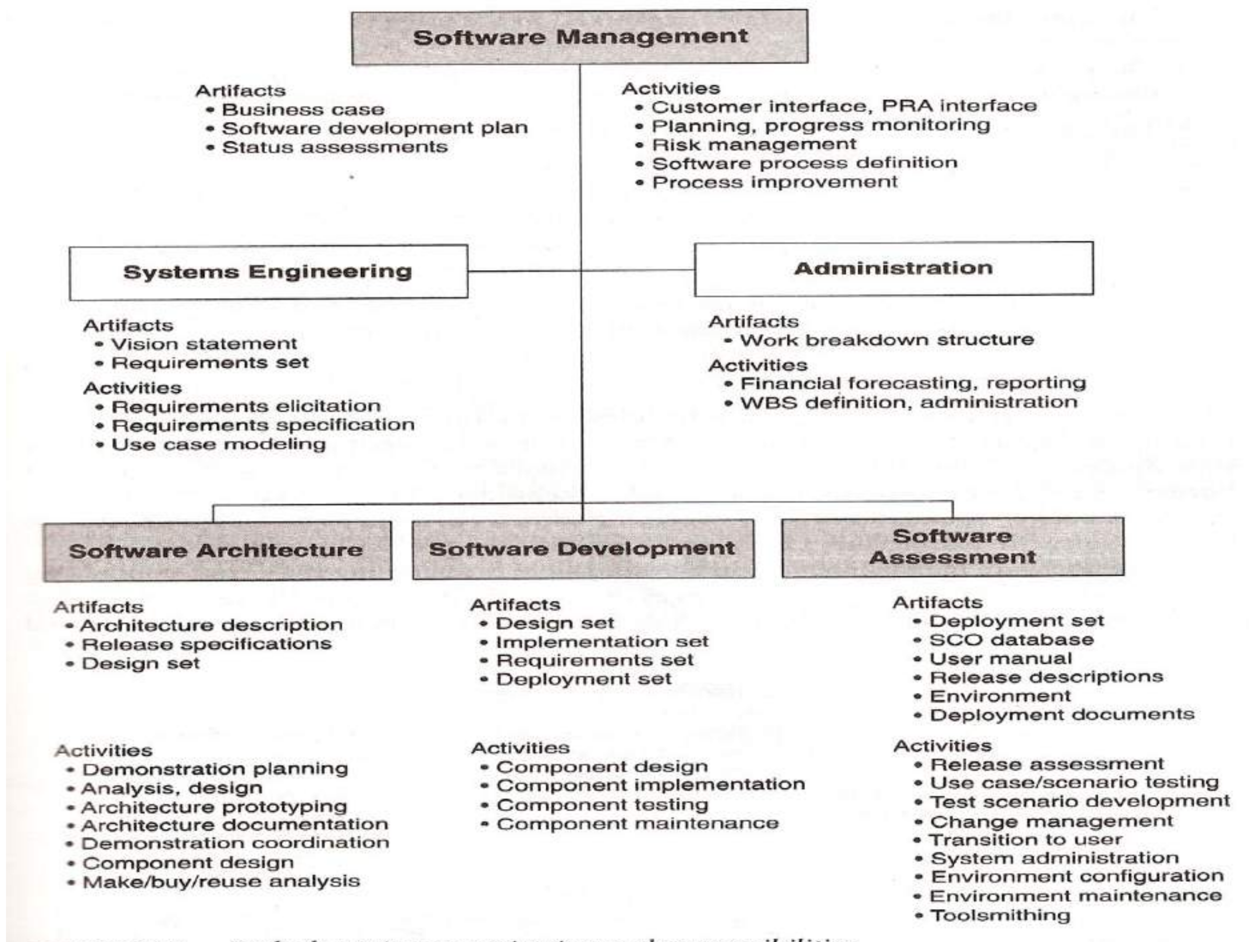


FIGURE 11-2. *Default project organization and responsibilities*

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: Software management team is responsible for:

- 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

SOFTWARE ARCHITECTURE TEAM

- The software architecture team performs the tasks of integrating the components, creating real artifacts etc. The skill possessed by the architecture team is of utmost importance as 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-case view.
- Software technology experience to generate an acceptable process view, component and development views
- Responsibilities: Software architecture team is responsible for:
- System-level quality i.e., performance, reliability and maintainability.
- Requirements and design trade-offs.
- Component selection
- Technical risk solution
- Initial integration

SOFTWARE DEVELOPMENT TEAM

- The Development team is involved in the construction and maintenance activities. It is most application specific 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: Software development team is responsible for
    - Component development, testing and maintenance.
    - Component design and implementation
    - Component documentation

SOFTWARE ASSESSMENT TEAM

- The team is involved in testing and product activities in parallel with the ongoing development. This 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 assessment team is responsible for
    - 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

EVOLUTION OF ORGANIZATIONS

- The project organization represents the architecture of the team and needs to evolve consistent with the project plan captured in the work breakdown structure. Figure 11-7 illustrates how the team's center of gravity shifts over the life cycle, with about 50% of the staff assigned to one set of activities in each phase.
- 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

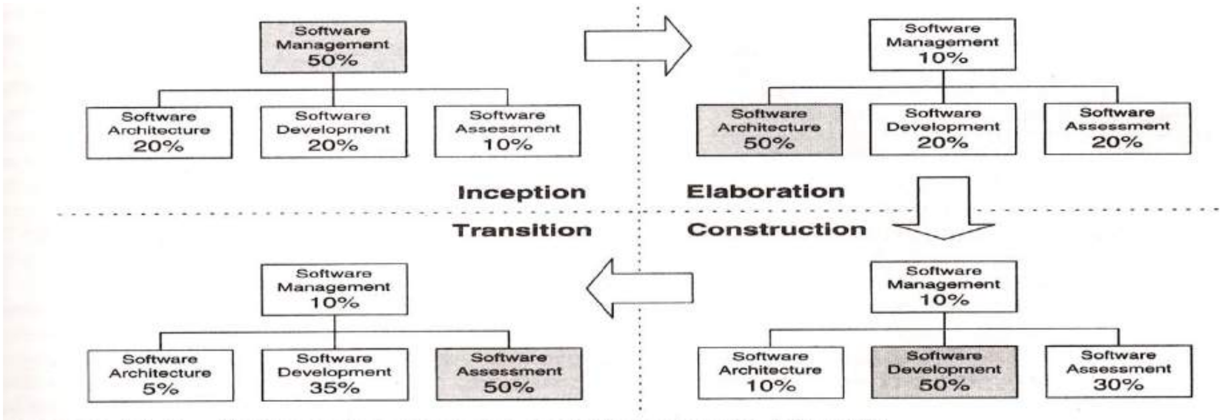


FIGURE 11-7. *Software project team evolution over the life cycle*

PROCESS AUTOMATION

- Three levels of process are
- **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.
- **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.
- **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

TOOLS: AUTOMATION BUILDING BLOCKS

- It introduces some of the important tools that tend to be needed universally across software projects and that correlate well to the process framework. (Many other tools and process automation aids are not included.) Most of the core software development tools map closely to one of the process workflows, as illustrated ill Figure 12-1.

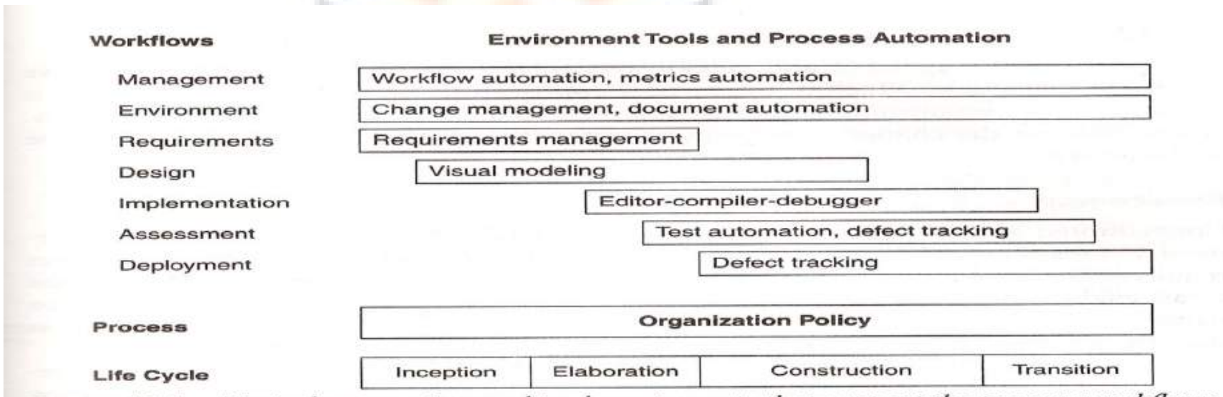


FIGURE 12-1. *Typical automation and tool components that support the process workflows*

- **MANAGEMENT**
  - There are many opportunities for automating the project planning and control activities of the management workflow. 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. This automation support can considerably improve the insight of the metrics collection and reporting concepts.
- **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 equal treatment of all requirements drained away engineering hours from the driving requirements then wasted that time on paperwork associated with detailed traceability that was inevitably discarded later as the driving requirements and subsequent design understanding evolved.
  - 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 too1s that Support the requirements, design, implementation, and assessment workflows are usually used together. 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, 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**

The assessment workflow requires all the tools just discussed as well as additional capabilities to support test automation and test management. 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: the prototyping environment, the development environment, and the maintenance environment.
- 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. This informal configuration of tools should be capable of supporting the following activities:
    - **Performance trade-offs and technical risk analyses**
    - **Make /buy trade-offs and feasibility studies for commercial products**
    - **Fault tolerance/dynamic reconfiguration trade-offs**
    - **Analysis of the risks associated with transitioning to full-scale implementation**
    - **Development of test scenarios, tools, and instrumentation suitable for analyzing the requirements.**

- 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.

The maintenance environment should typically coincide with a mature version of the development environment. In some cases, 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 the engineering artifacts.
- Figure 12-2 depicts some important transitions between information repositories. The automated translation of design models to source code (both forward and reverse engineering) is fairly well established. The automated translation of design models to process (distribution) models is

also becoming straightforward through technologies such as ActiveX and the Common Object Request Broker Architecture (CORBA).

The primary reason for round-trip engineering is to allow freedom in changing software engineering data sources

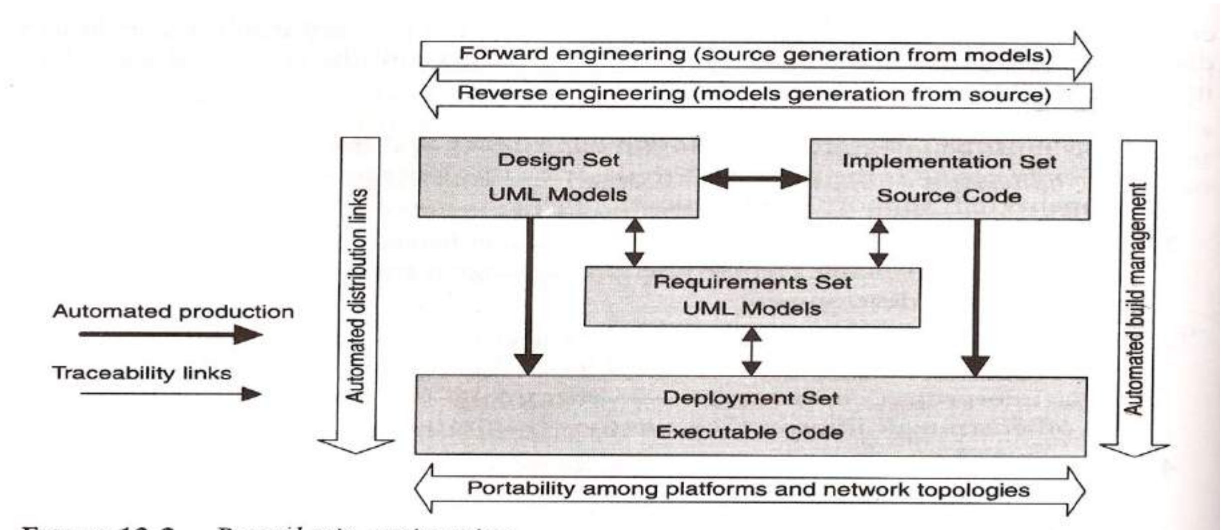

Forward engineering (source generation from models)
Reverse engineering (models generation from source)
Design Set
UML Models
Implementation Set
Source Code
Requirements Set
UML Models
Deployment Set
Executable Code
Automated production
Traceability links
Automated distribution links
Automated build management
Portability among platforms and network topologies

FIGURE 12-2. *Round-trip engineering*

CHANGE MANAGEMENT

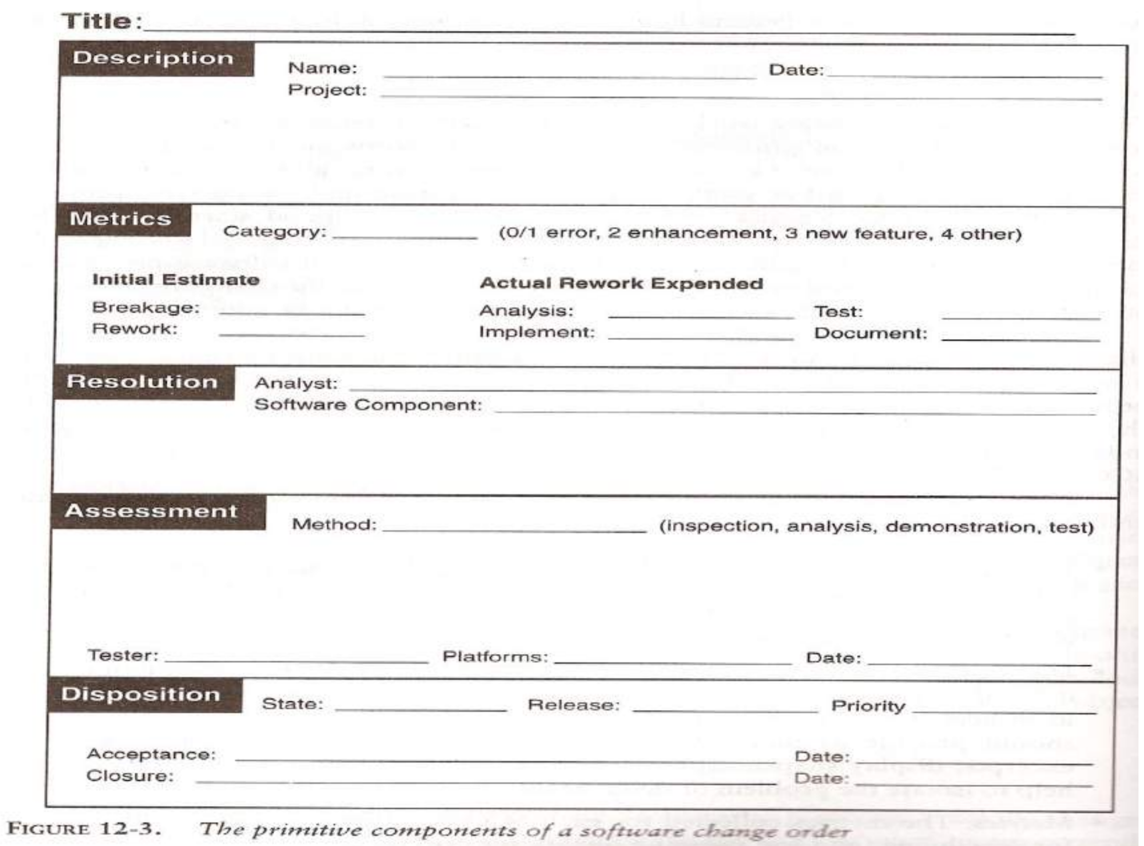
Change management is as critical to iterative processes as planning. Tracking changes in the technical artifacts is crucial to understanding the true technical progress trends and quality trends toward delivering an acceptable end product or interim release. In a modern process-in which requirements, 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

- 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 example SCO shown in Figure 12-3 is a good starting point for describing a set of change primitives. It shows the level of detail required to achieve the metrics and change management rigor necessary for a modern software process.

- The basic fields of the SCO are title, description, metrics, resolution, assessment and disposition.
- **Title.** The title is suggested by the originator and is finalized upon acceptance by the configuration control board (CCB).
- **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.
- **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)
- **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
- **Assessment:** This field describes the assessment technique as 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.
- **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.



Title: ______

Description
Name: ______ Date: ______
Project: ______

Metrics
Category: ______ (0/1 error, 2 enhancement, 3 new feature, 4 other)

Initial Estimate		Actual Rework Expended			
Breakage:	______	Analysis:	______	Test:	______
Rework:	______	Implement:	______	Document:	______

Resolution
Analyst: ______
Software Component: ______

Assessment
Method: ______ (inspection, analysis, demonstration, test)

Tester: ______ Platforms: ______ Date: ______

Disposition
State: ______ Release: ______ Priority ______

Acceptance: ______ Date: ______
Closure: ______ Date: ______

FIGURE 12-3. *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 genera1ly two classes of baselines: external product releases and 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.

Generally, three levels of baseline releases arc required for most systems: major, minor, and interim. Each level corresponds to a numbered identifier such as N.M.X, where N is the major release number, M is the minor release number, and X is the interim release identifier. A major release represents a new generation of the product or project, while a minor release represents the same basic product but with enhanced features, performance, or quality. Major and minor releases are intended to be external product releases that are persistent and supported for a period of time. An interim release corresponds to a developmental configuration that is intended to be transient. The shorter its life cycle, the better. Figure 12-4 shows examples of some release name histories for two different situations

- 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.
- Table 12-1 provides examples of these changes in the context of two different project domains: a large-scale, reliable air traffic control system and a packaged software development tool

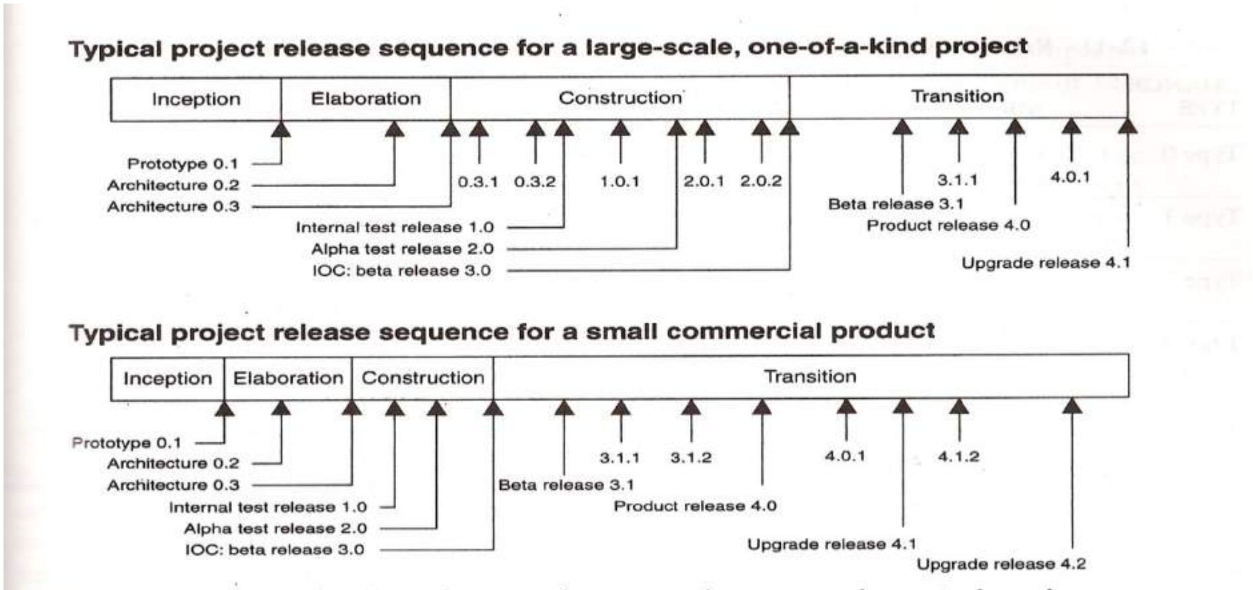


FIGURE 12-4. *Example release histories for a typical project and a typical product*

Change Type	**Air Traffic control Project**	**Packaged visual Modeling Tool**
Type 0	Control deadlock and loss of flight data	Loss of user data
Type 1	Display response time that exceeds the requirement by 0.5 second	Browser expands but does not collapse displayed entries
Type 2	Add internal message field for response time instrumentation	Use of color to differentiate updates from previous version of visual model
Type 3	Increase air traffic management capacity from 1,200 to 2,400 simultaneous flights	Port to new platform such as WinNT

Type 4	Upgrade from Oracle 7 to Oracle 8 to improve query performance	Exception raised when interfacing to MS Excel 5.0 due to windows resource management bug.

CONFIGURATION CONTROL BOARD

- 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 the process.
  [Proposed]: A proposed change is drafted and submitted to the CCB. The proposed change must include a 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 actual 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

- From a process automation perspective, the organization's infrastructure provides the organization capital assets, including two key artifacts: a policy that captures the standards for project software development processes, and 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

- The need for balance is an important consideration in defining organizational policy. Effective organizational policies have several recurring themes:
  - They are concise and avoid policy statements that fill 6-inch-thick documents.
  - They confine the policies to the real shalls, then enforce them.

        - They avoid using the word should in policy statements. Rather than a menu of options (shoulds), policies need a concise set of mandatory standards (shalls).
        - Waivers are the exception, not the rule.
- Appropriate policy is written at the appropriate level.
- The organization policy is the defining document for the organization's software policies. In any process assessment, this is the tangible artifact that says what you do. From this document, reviewers should be able to question and review projects and personnel and determine whether the organization does what it says. Figure 12-5 shows a general outline for an organizational policy.
    - **Process-Primitive definitions**
        - Life-cycle phases (inception, elaboration, construction, transition)
        - Checkpoints (major milestones, minor milestones, status assessments)
        - Artifacts (requirements, design, implementation, deployment, management sets)
        - Roles and responsibilities (PRA, SEPA, SEEA, project teams).
    - **Organization software policies**
        - Work breakdown structure
        - Software development plan
        - Baseline change management
        - Software metrics
        - Development environment
        - Evaluation criteria and acceptance criteria
        - Risk management
        - Testing and assessment.
    - **Walver policy**

  - **Appendixes**
    - Current process assessment
    - Software process improvement plan.
  - Some of the typical components of an organization's automation building blocks are as follows:
    - Standardized tool selections (through investment by the organization in a site license or negotiation of a favorable discount with a tool vendor so that project teams are motivated economically to use that tool), 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).
- Other indirectly useful components of an organization's infrastructure
  - A reference library of precedent experience for planning, assessing and improving process performance parameters; answers for how well? How much? Why?
  - Existing case studies, including objective benchmarks of performance for successful projects that followed the organization process.
  - A library of project artifact examples such as software development plans, architecture descriptions and status assessment histories.
  - Mock audits and compliance traceability for external process assessment frameworks.

- Such as the software Engineering Institute's Capability Maturity Model (SEI CMM)

STAKEHOLDER ENVIRONMENTS

- The transition to a modern iterative development process with supporting automation should not be restricted to the development team. many large scale contractual projects include people in external organization that represent other stakeholders participating in the development process.
- 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 manage and monitor the project.

Avoid excessive travel, paper interchange delays, format translations, paper and shipping costs and other overhead costs

- **FIGURE 12-6:** Illustrates some of the new opportunities for value-added activities by external stakeholders in large contractual efforts. There are several important reasons for extending development environment resources into certain stakeholder domains.
  - Technical artifacts are not just paper. Electronic artifacts in rigorous notations such as visual models and source code are viewed far more efficiently by using tools with smart browsers.
  - Independent assessments of the evolving artifacts are encouraged by electronic read-only access to on-line data such as configuration baseline libraries and the change management database. Reviews and inspections, breakage/rework assessments, metrics analyses and even beta testing can be performed independently of the development team.

Even paper documents should be delivered electronically to reduce production costs and turn around time.

PROJECT CONTROL & PROCESS INSTRUMENTATION

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

products of the software development process. Hence, the quality of the software products and the achievements in the development process can be determined using the software metrics.

Need for Software Metrics:

Software metrics are needed for calculating the cost and schedule of a software product with great accuracy.

Software metrics are required for making an accurate estimation of the progress.

The metrics are also required for understanding the quality of the software product.

INDICATORS:

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

Two types of indicators are:

Management indicators.

Quality indicators.

Management Indicators

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

Quality Indicators

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

SEVEN CORE METRICS OF SOFTWARE PROJECT

Software metrics instrument the activities and products of the software development/integration process. Metrics values provide an important perspective for managing the process. The most useful metrics are extracted directly from the evolving artifacts.

There are seven core metrics that are used in managing a modern process.

Seven core metrics related to project control:

Management Indicators

1. Work and Progress
2. Budgeted cost and expenditures
3. Staffing and team dynamics

Quality Indicators

4. Change traffic and stability
5. Breakage and modularity
6. Rework and adaptability
7. Mean time between failures (MTBF) and maturity

MANAGEMENTINDICATORS:

1. Work and progress

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

The default perspectives of this metric are: Software architecture team: - Use cases demonstrated.

Software development team: - SLOC under baseline change management, SCOs closed Software assessment team: - SCOs opened, test hours executed and evaluation criteria meet. Software management team: - milestones completed.

Budgeted cost and expenditures

This metric measure cost incurred over time. Budgeted cost is the planned expenditure profile over the life cycle of the project. To maintain management control, measuring cost expenditures over the project life cycle is always necessary. Tracking financial progress takes on an organization - specific format. Financial performance can be measured by the use of an earned value system, which provides highly detailed cost and schedule insight. The basic parameters of an earned value system, expressed in units of dollars, are as follows:

Expenditure Plan - It is the planned spending profile for a project over its planned schedule. Actual progress - It is the technical accomplishment relative to the planned progress underlying thespending profile.

Actual cost: It is the actual spending profile for a project over its actual schedule.

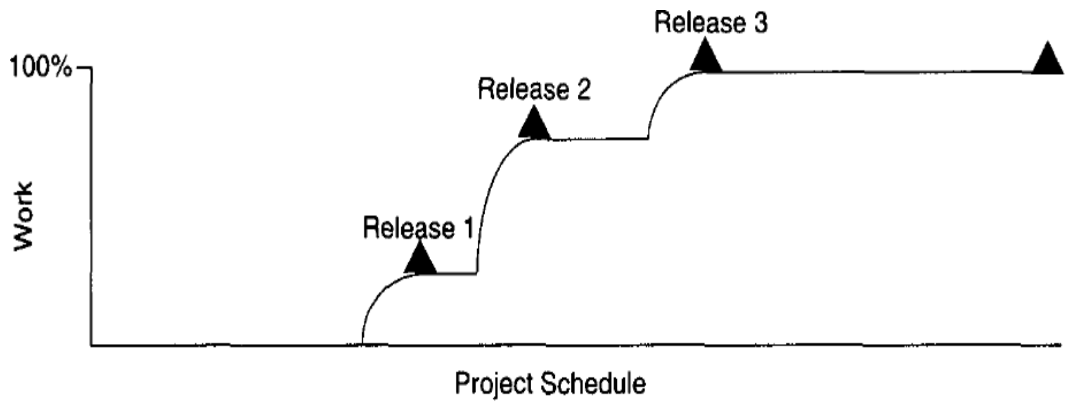
Earned value: It is the value that represents the planned cost of the actual progress.

Cost variance: It is the difference between the actual cost and the earned value.

staff per month and percentage of budget expended.

Staffing and team dynamics

This metric measures the personnel changes over time, which involves staffing additions and reductions over time. An iterative development should start with a small team until the risks in the requirements and architecture have been suitably resolved. Depending on the overlap of iterations and other project specific circumstances, staffing can vary. Increase in staff can slow overall project progress as new people consume the productive team of existing people in coming up to speed. Low attrition of good people is a sign of success.



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

Budgeted cost and expenditures

This metric measure cost incurred over time. Budgeted cost is the planned expenditure profile over the life cycle of the project. To maintain management control, measuring cost expenditures over the project life cycle is always necessary. Tracking financial progress takes on an organization - specific format. Financial performance can be measured by the use of an earned value system, which provides highly detailed cost and schedule insight. The basic parameters of an earned value system, expressed in units of dollars, are as follows:

Expenditure Plan - It is the planned spending profile for a project over its planned schedule. Actual progress - It is the technical accomplishment relative to the planned progress underlying the spending profile.

Actual cost: It is the actual spending profile for a project over its actual schedule.

Earned value: It is the value that represents the planned cost of the actual progress.

Staffing and team dynamics

This metric measures the personnel changes over time, which involves staffing additions and reductions over time. An iterative development should start with a small team until the risks in the requirements and architecture have been suitably resolved. Depending on the overlap of iterations and other project specific circumstances, staffing can vary. Increase in staff can slow overall project progress as new people consume the productive team of existing people in coming up to speed. Low attrition of good people is a sign of success. The default perspectives of this metric are people per month added and people per month leaving. These three management indicators are responsible for technical progress, financial status and staffing progress.

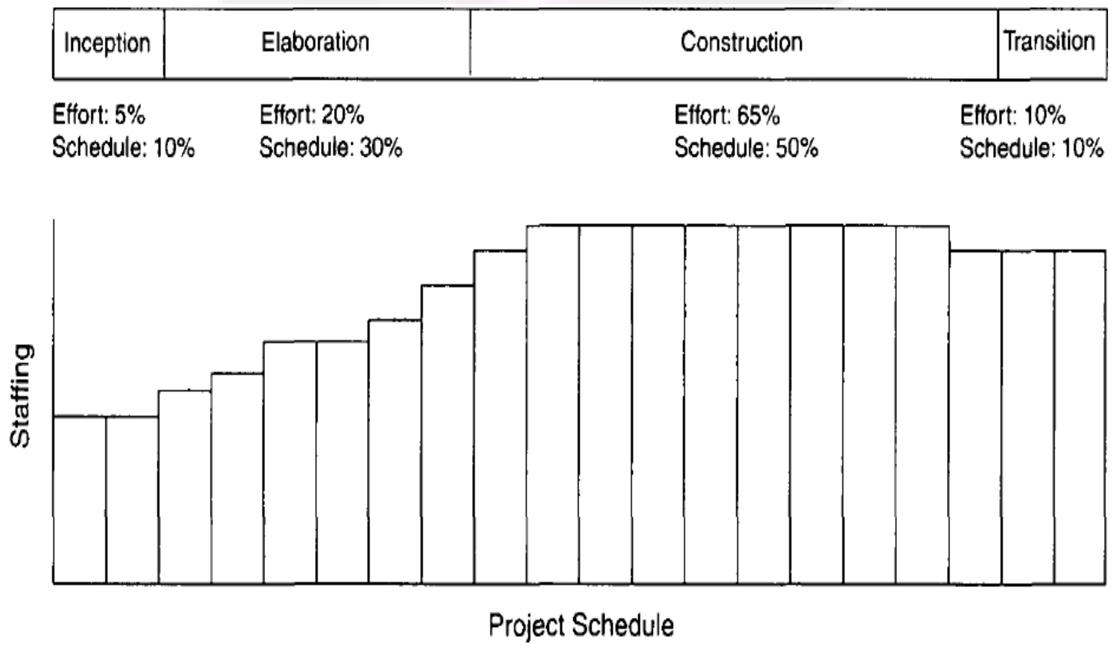


Fig: staffing and Team dynamics

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

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

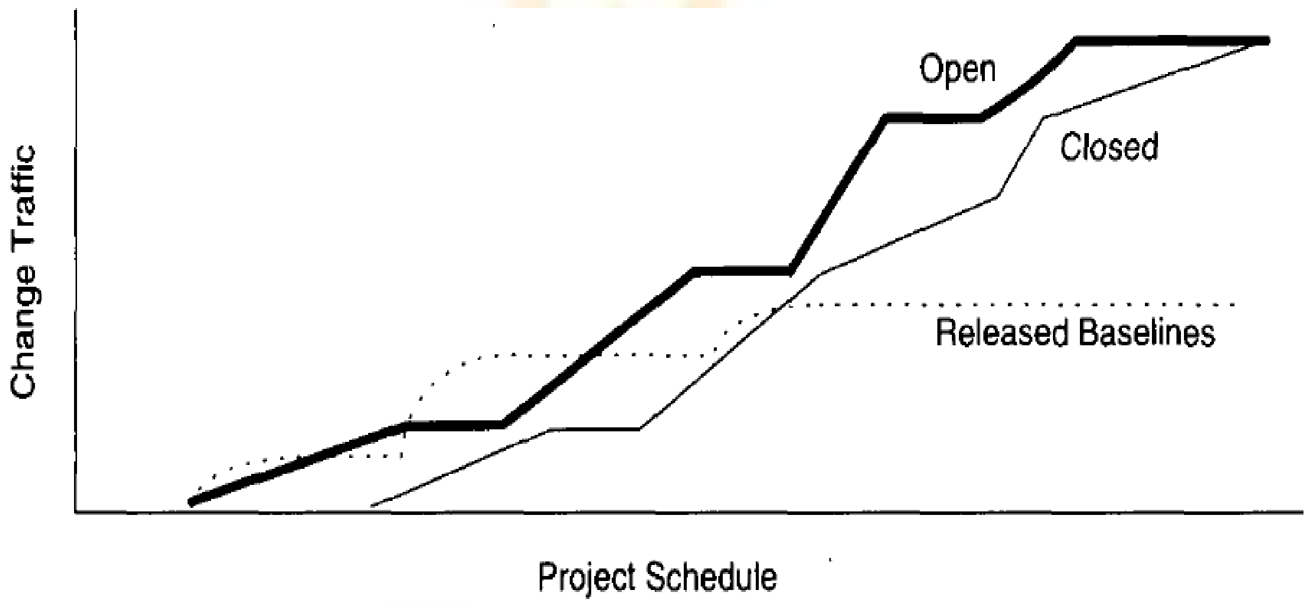


Fig: Change traffic and stability

<u>Breakage and modularity</u>

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

<u>Rework and adaptability:</u>

This metric measures the average rework per change over time. Rework is defined as the average cost of change which is the effort to analyze, resolve and retest all changes to software baselines. Adaptability is defined as the rework trend over time. This metric provides insight into rework measurement. All changes are not created equal. Some changes can be made in a staff- hour, while others take staff-weeks. This metric can be collected by average hours per change, by change type, by release, by components and by subsystems.

<u>MTBF and Maturity</u>

This metric measure defect rather over time. MTBF (Mean Time Between Failures) is theaverage usage time between software faults. It is computed by dividing the test hours by the number of type 0 and type 1 SCOs. Maturity is defined as the MTBF trend over time. Software errors can be categorized into two types deterministic and nondeterministic. Deterministic errors are also known as Bohr-bugs and nondeterministic errors are also called as Heisen-bugs. Bohr-bugs are a class of errors caused when the software is stimulated in a certain way such as coding errors. Heisen-bugs are software faults that are coincidental with a certain probabilistic occurrence of a given situation, such as design errors. This metric can be collected by failure counts, test hours until failure, by release, by components and by subsystems. These four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data.

LIFE -CYCLE EXPECTATIONS:

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

The quality indicators are derived from the evolving product rather than the artifacts. They provide inside into the waste 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 and the current trend provides tangible indicators for management action.

Table: The default pattern of life cycle 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%

METRICS AUTOMATION:

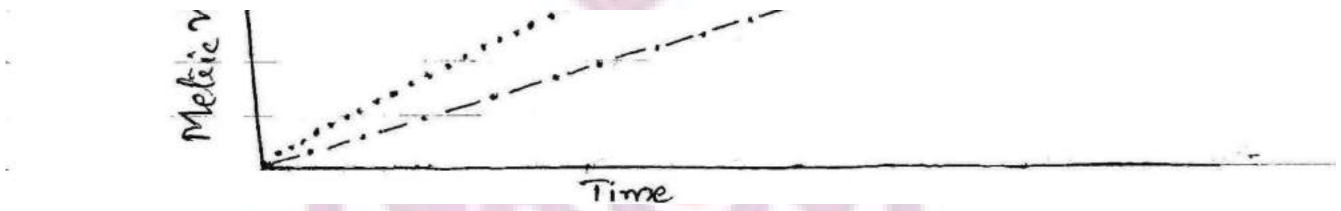
Many opportunities are available to automate the project control activities of a software project. A Software Project Control Panel (SPCP) is essential for managing against a plan. This panel integrates data from multiple sources to show the current status of some aspect of the project. The panel can support standard features and provide extensive capability for detailed situation analysis. SPCP is one example of metrics automation approach that collects, organizes and reports values and trends extracted directly from the evolving engineering artifacts.

SPCP:

To implement a complete SPCP, the following are necessary.

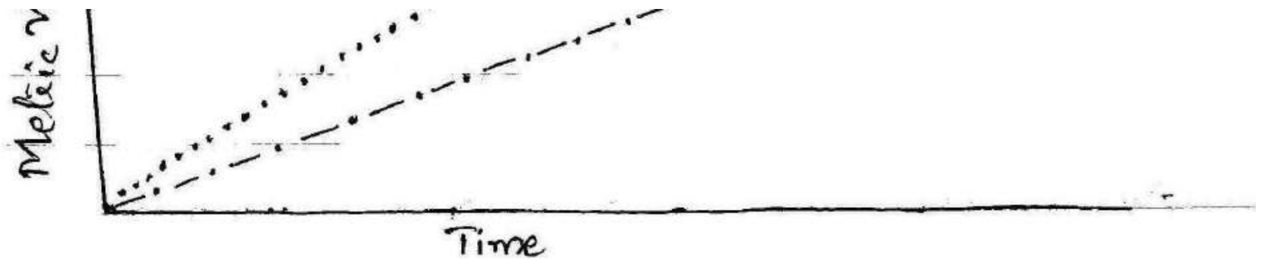
- Metrics primitives - trends, comparisons and progressions
- A graphical user interface.
- Metrics collection agents
- Metrics data management server
- Metrics definitions - actual metrics presentations for requirementsprogress, implementation progress, assessment progress, design progress and other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

Monitor defines panel layouts, graphical objects and linkages to project data. Specific monitors called roles include software project managers, software development team leads, software architects and customers. Administrator installs the system, defines new mechanisms, graphical objects and linkages. The whole display is called a panel. Within a panel are graphical objects, which are types of layouts such as dials and bar charts for information. Each graphical object displays a metric. A panel contains a number of graphical objects positioned in a particular geometric layout. A metric shown in a graphical object is labelled with the metric type, summary level and insurance name (line of code, subsystem, server1). Metrics can be displayed in two modes – value, referring to a given point in time and graph referring to multiple and consecutive points in time. Metrics can be displayed with or without control values. A control value is an existing expectation either absolute or relative that is used for comparison with a dynamically changing metric. Thresholds are examples of control values.



The basic fundamental metrics classes are trend, comparison and progress.

The format and content of any project panel are configurable to the software project manager's preference for tracking metrics of top-level interest. The basic operation of an SPCP can be described by



the following top - level use case.

i. Start the SPCP
ii. Select a panel preference
iii. Select a value or graph metric
iv. Select to superimpose controls
v. Drill down to trend
vi. Drill down to point in time.
vii. Drill down to lower levels of information
viii. Drill down to lower level of indicators.

,

(3) 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.

Do it twice. If a computer program is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned. Note that this is simply the entire process done in miniature, to a time scale that is relatively small with respect to the overall effort. In the first version, the team must have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives, forget the straightforward aspects of the design that aren't worth studying at this early point, and, finally, arrive at an error-free program.

Plan, control, and monitor testing. Without question, the biggest user of project resources-manpower, computer time, and/or management judgment-is the test phase.

This is the phase of greatest risk in terms of cost and schedule.

It occurs at the latest point in the schedule, when backup alternatives are least available, if at all. The previous three recommendations were all aimed at uncovering and solving problems before entering the test phase. However, even after doing these things, there is still a test phase and there are still important things to be done, including:

(1) employ a team of test specialists who were not responsible for the original design;

(2) employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses (do not use the computer to detect this kind of thing, it is too expensive);

(3) test every logic path;

(4) employ the final checkout on the target computer.

1. Involve the customer. It is important to involve the customer in a formal way so that hehas committed himself at earlier points before final delivery. There are three points following requirements definition where the insight, judgment, and commitment of the customer can bolster the development effort. These include a "preliminary software review" following the preliminary program design step, a sequence of "critical software design reviews" during program design, and a "final software acceptance review".

UNIT 5:

CCPDS-R Case Study and Future Software Project Management Practices Modern Project Profiles, Next-Generation software Economics, Modern Process Transitions

COMMAND CENTER PROCESSING AND DISPLAY SYSTEM-REPLACEMENT (CCPDS-R)

- The Command Center Processing and Display Sys-tem-Replacement (CCPDS-R) project was performed for the U.S. Air Force by TRW Space and Defense in Redondo Beach, California. The entire project included systems engineering, hardware procurement, and software development, with each of these three major activities consuming about one-third of the total cost. The schedule spanned 1987 through 1994.

a The metrics histories were all derived directly from the artifacts of the project's process. These data were used to manage the project and were embraced by practitioners, managers, and stakeholders.

There are very few well-documented projects with objective descriptions of what worked, what didn't, and why. This was one of my primary motivations for providing the level of detail contained in this appendix. It is heavy in project-specific details, approaches, and results, for three reasons:

1. Generating the case study wasn't much work. CCPDS-R is unique in its detailed and automated metrics approach. All the data were derived directly from the historical artifacts of the project's process.

2. This sort of objective case study is a true indicator of a mature organization and a mature project process. The absolute values of this historical perspective are only marginally useful. However, the trends, lessons learned, and relative priorities are distinguishing characteristics of successful software development.

3. Throughout previous chapters, many management and technical approaches are discussed generically. This appendix provides in a real-world example at least one relevant benchmark of performance

- The CCPDS-R project produced a large-scale, highly reliable command and control system that provides missile warning information used by the National Command Authority. The procurement agency was Air Force Systems Command Headquarters, Electronic Systems Division, at Hanscom Air Force Base, Massachusetts. The primary user was US Space Command, and the full-scale development contract was awarded to TRWs Systems Integration Group in 1987. The CCPDS-R contract called for the development of three subsystems:

1. The Common Subsystem was the primary missile warning system within the Cheyenne Mountain Upgrade program. It required about 355,000 source lines of code, had a 48-month software development schedule, and laid the foundations for the subsystems that followed (reusable components, tools, environment, process, procedures). The Common Subsystem included a primary installation in Cheyenne Mountain, with a backup system deployed at Offutt Air Force Base, Nebraska.

2. The Processing and Display Subsystem (PDS) was a scaled-down missile warning display system for all nuclear-capable commanders-in-chief. The PDS software (about 250,000 SLOC) was fielded on remote, read-only workstations that were distributed worldwide.

3. The STRATCOM Subsystem (about 450,000 SLOC) provided both missile warning and force management capability for the backup missile warning center at the command center of the Strategic Command

CCPDS-R LIFE-CYCLE OVERVIEW

- The CD phase was very similar in intent to the inception phase. The primary products were a system specification (a vision document), an FSD phase proposal (a business case, including the technical approach and a fixed-price-incentive and award-fee cost proposal), and a software development plan. The CD phase also included a system design review, technical interchange meetings with the government stakeholders (customer and user), and several contract-deliverable documents. These events and products enabled the FSD source selection to be based on demonstrated performance of the contractor-proposed team as well as the FSD proposal.

- From a software perspective, there was one additional source selection criterion included in the FSD proposal activities: a software engineering exercise. This was a unique but very effective approach for assessing the abilities of the two competing contractors to perform software development. The Air Force was extremely concerned with the overall software risk of this project: Recent projects had demonstrated dismal software development performance. The Air Force acquisition authorities had also been frustrated with previous situations in which a contractor's crack proposal team was not the team committed to perform after contract award, and contractor proposals exaggerated their approaches or capabilities beyond what they could deliver.

  CCPDS-R was also a very large software development activity and was one of the first projects to use the Ada programming language. There was serious concern that the Ada development environments, contractor processes, and contractor training programs might not be mature enough to use on a full-scale development effort. The purpose of the software engineering exercise was to demonstrate that the contractor's proposed software process, Ada environment, and software team were in place, were mature, and were demonstrable

- The software engineering exercise occurred immediately after the FSD proposals were submitted. The customer provided both bidders with a simple two-page specification of a "missile warning simulator." This simulator had some of the same fundamental requirements as the CCPDS-R full-scale system, including a distributed architecture, a flexible user interface, and the basic processing scenarios of a simple CCPDS-R missile warning thread. The exercise requirements included the following:
- Use the proposed software team.
- Use the proposed software development techniques and tools.
- Use the FSD-proposed software development plan.
- Conduct a mock design review with the customer 23 days after receipt of the specification.
- Four primary use cases were elaborated and demonstrated.
- A software architecture skeleton was designed, prototyped, and documented, including two executable, distributed processes; five concurrent tasks (separate threads of control); eight components; and 72 component-to-component interfaces.
- A total of 4,163 source lines of prototype components were developed and executed. Several thousand lines of reusable components were also integrated into the demonstration.
- Three milestones were conducted and more than 30 action items resolved.

- Production of 11 documents (corresponding to the proposed artifacts) demonstrated the automation inherent in the documentation tools.
- The Digital Equipment Corporation VAX/VMS tools, Rational R1000 environment, LaTeX documentation templates, and several custom-developed tools were used.
- Several needed improvements to the process and the tools were identified. The concept of evolving the plan, requirements, process, design, and environment at each major milestone was considered potentially risky but was implemented with rigorous change management.
- In preparing for the CCPDS-R project, TRW placed a strong emphasis on evolving the right team. The CD phase team represented the essence of the architecture team which is responsible for an efficient engineering stage. This team had the following primary responsibilities:
- Analyze and specify the project requirements
- Define and develop the top-level architecture
- Plan the FSD phase software development activities
- Configure the process and development environment
- Establish trust and win-win relationships among the stakeholders

1.Network Architecture Services (NAS). This foundation middleware provided reusable components for network management, interprocess communications, initialization, reconfiguration, anomaly management, and instrumentation of software health, performance, and state. This CSCI was designed to be reused across all three CCPDS-R subsystems.

2. System Services (SSV). This CSCI comprised the software architecture skeleton, real-time data distribution, global data types, and the computer system operator interface.

3. Display Coordination (DCO). This CSCI comprised user interface control, display formats, and display population.

4. Test and Simulation (TAS). This CSCI comprised test scenario generation, test message injection, data recording, and scenario playback.

5. Common Mission Processing (CMP). This CSCI comprised the missile warning algorithms for radar, nuclear detonation, and satellite early warning messages.

6. Common Communications (CCO). This CSCI comprised external interfaces with other systems and message input, output, and protocol management

MODERN PROJECT PROFILES

Continuous Integration

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

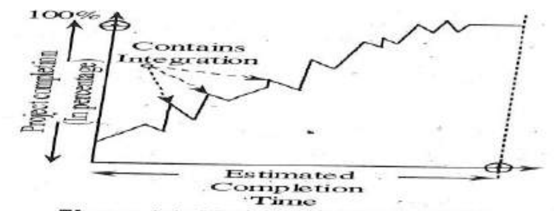


Figure (a): Modern Project Profile

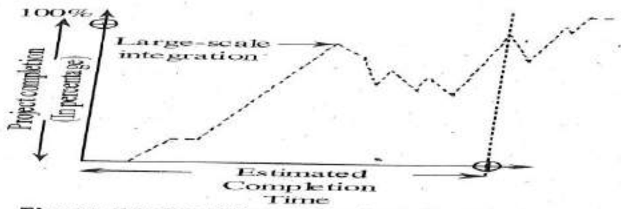


Figure (b): Traditional Project Profile

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

In the modern project profile, the distribution of cost among various workflows or project is completely different from that of traditional project profile as shown below

Software Engineering Workflows	Conventional Process Expenditures	Modern process Expenditures
Management	5%	10%
Environment	5%	10%
Requirements	5%	10%
Design	10%	15%
Implementation	30%	25%
Assessment	40%	25%
Deployment	5%	5%

Total	100%	100%

As shown in the table, the modern projects spend only 25% of their budget for integration and Assessment activities whereas; traditional projects spend almost 40% of their total budget for these activities. This is because, the traditional project involve inefficient large-scale integration and late identification of design issues

EARLY RISK RESOLUTION

- In the project development lifecycle, the engineering phase concentrates on identification and elimination of the risks associated with the resource commitments just before the production stage. The traditional projects involve, the solving of the simpler steps first and then goes to the complicated steps, as a result the progress will be visibly good, whereas, the modern projects focuses on 20% of the significant requirements, use cases, components and risk and hence they occasionally have simpler steps.
- To obtain a useful perspective of risk management, the project life cycle has to be applied on the principles of software management. The following are the 80:20 principles.
- The 80% of Engineering is utilized by 20% of the requirementsBefore selecting any of the resources, try to completely understand all the requirement because irrelevant resource selection (i.e., resources selected based on prediction) may yield severe problems.
- 80% of the software cost is utilized by 20% of the components
- Firstly, the cost-critical components must be elaborated which forces the project to focus more on controlling the cost.
- 80% of the bugs occur because of 20% of the components

- Firstly, the reliability-critical components must be elaborated which give sufficient time for assessment activities like integration and testing, in order to achieve the desired level of maturity.
- 80% of the software scrap and rework is due to 20% if the changes.
- The change-critical components r elaborated first so that the changes that have more impact occur when the project is matured.
- 80% of the resource consumption is due to 20% of the components.
- Performance critical components are elaborated first so that, the trade-offs with reliability; changeability and cost-consumption can be solved as early as possible.
- 80% of the project progress is carried-out by 20% of the people
- It is important that planning and designing team should consist of best processionals because the entire success of the project depends upon a good plan and architecture.
- The following figure shows the risk management profile of a modern project.

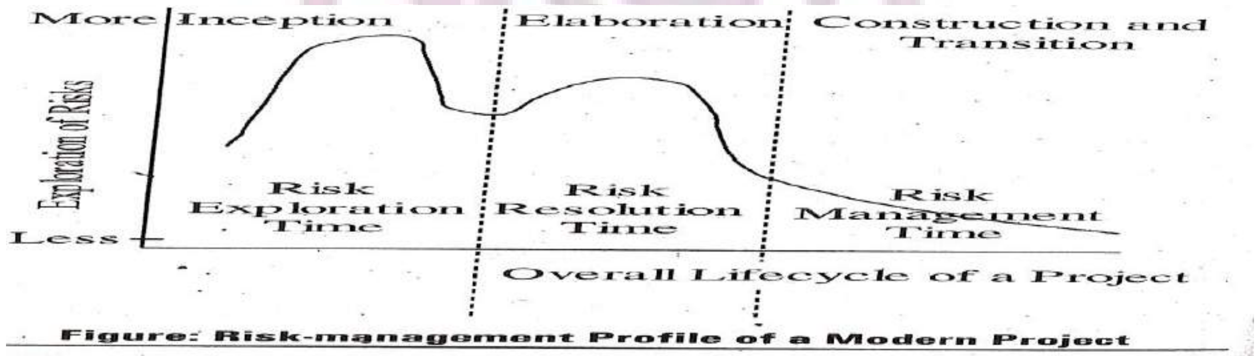


Figure: Risk-management Profile of a Modern Project

EVOLUTIONARY REQUIREMENTS

- The traditional methods divide the system requirements into subsystem requirements which in turn gets divided into component requirements. These component requirements are further divided into unit requirements. The reason for this systematic division is to simplify the traceability of the requirements.
- In the project life cycle the requirements and design are given the first and the second preference respectively. The third preference is given to the traceability between the requirement and the design components these preferences are given in order to make the design structure evolve into an organization so it parallels the structure of the requirements organization.
- Modern architecture finds it difficult to trace the requirements because of the following reasons.
    - Usage of Commercial components
    - Usage of legacy components
    - Usage of distributed resources
    - Usage of object oriented methods.

Moreover, the complex relationships such as one-one, many-one, one-many, conditional, time-based and state based exists the requirements statement and the design elements

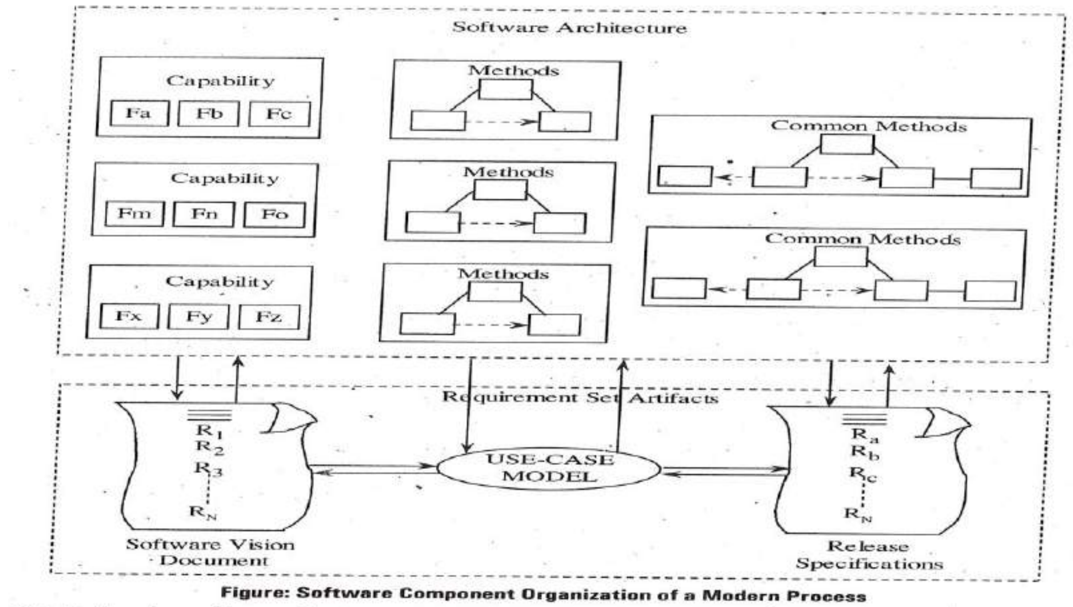


Figure: Software Component Organization of a Modern Process

As shown in the above figure, the top category system requirements are kept as the vision whereas, those with the lower category are evaluated. The motive behind theses artifacts is to gain fidelity with respect to the progress in the project lifecycle. This serves as a significant different from the traditional approach because, in traditional approach the fidelity is predicted early in the project life cycle

TEAMWORK AMONG STAKEHOLDERS

- Most of the characteristics of the classic development process worsen the stakeholder relationship s which in turn makes the balancing of requirement product attributes and plans difficult. An iterative process which ahs a good relationship between the stakeholders mainly focuses on objective understanding by each and every individual stakeholder. This process needs highly skilled customers, users and monitors which have experience in both the application as well as software. Moreover, this process requires an organization whose focus is on producing a quality product and achieves customer satisfaction.

The table below shows the tangible results of major milestones in a modern process

Obvious result	**Actual result**
Demonstration at early stage reveals the design issued and uncertainty in a tangible form.	Demonstration firstly reveals the significant assets and risks associated with complicated software systems such that they can be worked out at the time of setting the life-cycle goals.
Non-Complaint design	Various perspective like requirements use cases etc are observed in order to completely understand the compliance.
Issues of influential requirements are reveals but without traceability	Both the requirement changes and the design trade-offs are considerably balanced.
The design is considered to be "guilty until its innocency is proved.	The engineering issues can be integrated into the succeeding iteration's plans.

- From the above table, it can be observed that the progress of the project is not possible unless all the demonstration objectives are satisfied. This statement does not present the renegotiation of objectives, even when the demonstration results allow the further processing of trade offs present in the requirement, design, plans and technology.
- Modern iterative process that rely on the results of the demonstration need al its stakeholders to be well-educated and with a g good analytical ability so as to distinguish between the obviously negative results and the real

progress visible. For example, an early determined design error can be treated as a positive progress instead to a major issue.

Principles of Software Management

- Software management basically relies on the following principles, they are,

1. Process must be based on architecture-first approach

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

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

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

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

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

4. Create a change management Environment

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

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

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

Design artifacts must be captured in model based notation.

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

- **Complexity control**
- **Objective fulfillment**
- **Performing automated analysis**

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

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

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

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

9.The Points Increments and generations must be made based on the evolving levels of detail

Here, the 'levels of detail' refers to the level of understanding requirements and architecture. The requirements, iteration content, implementations and acceptance testing can be organized using cohesive usage scenarios.

10. Develop a configuration process that should be economically scalable

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

BEST PRACTICES ASSOCIATED WITH SOFTWARE MANAGEMENT

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

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

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

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

Management and scheduling based on metrics: This principle is related to the model based approach and objective quality control principles. It states to use common notations fro the artifacts so that quality and progress can be easily measured.

5. *Binary quality Gates at the inch-pebble level:* The concept behind this practice is quite confusing. Most of the organizations have misunderstood the concept and have developed an expensive and a detailed plan during the initial phase of the lifecycle, but later found the necessity to change most of their detailed plan due to the small changes in requirements or architectural. This principle states that first start planning with an understanding of requirements and the architecture. Milestones must be established during engineering stage and inch-pebble must be followed in the production stage.

6. *Plan versus visibility of progress throughout the progress:* This practice involves a direct communication between different team members of a project so that, they can discuss the significant issues related to the project as well as notice the progress of the project in-comparison to their estimated progress

7.Identifying defects associated with the desired quality: This practice is similar to the architecture-first approach and objective quality control principles of software management. It involves elimination of architectural defects early in the life-cycle, thereby maintaining the architectural quality so as to successfully complete the project.

8. Configuration management: According to Airline software council, configuration management serves as a crucial element for controlling the complexity of the artifacts and for tracing the changes that occur in the artifacts. This practice is similar to the change management principle of software management and prefers automation of components so as to reduce the probability of errors that occur in the large-scale projects.

9.Disclose management accountability: The entire managerial process is disclosed to al the people dealing with the project

NEXT GENERATION SOFTWARE COST MODELS

- In comparison to the current generation software cost modes, the next generation software cost models should perform the architecture engineering and application production separately. The cost associated with designing, building, testing and maintaining the architecture is defined in terms of scale, quality, process, technology and the team employed.
- After obtaining the stable architecture, the cost of the production is an exponential function of size, quality and complexity involved.
- The architecture stage cost model should reflect certain diseconomy of scale (exponent less than 1.0) because it is based on research and development-oriented concerns. Whereas the production stage cost model should reflect economy of scale (exponent less than 1.0) for production of commodities.
- **The next generation software cost models should be designed in a way that, they can assess larger architectures with economy of scale. Thus,**

the process exponent will be less than 1.0 at the time of production because large systems have more automated proves components and architectures which are easily reusable.

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

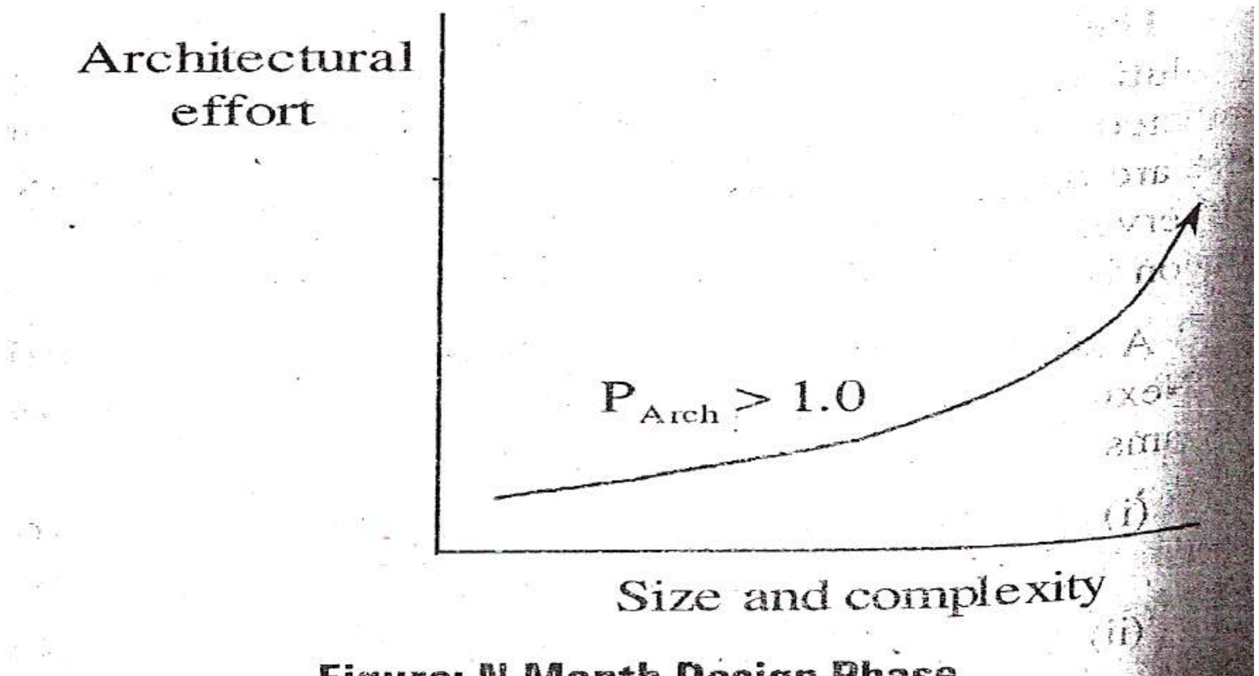


Figure: N-Month Design Phase

- At Application production stage
  - A plan with high fidelity and lower risk
  - It is cost-based
  - It has fixed-priced contracts
  - Team size is large and diverse as needed.
  - Architecture team consists of a small number of software engineers.
  - The Application team may have nay number of domain engineers.
  - The output will be a function which is deliverable and useful, tested

baseline and warranted quality.

- The focus of the application production will be on implementing testing

and maintaining target technology.

- It contains two phases they are construction and transition

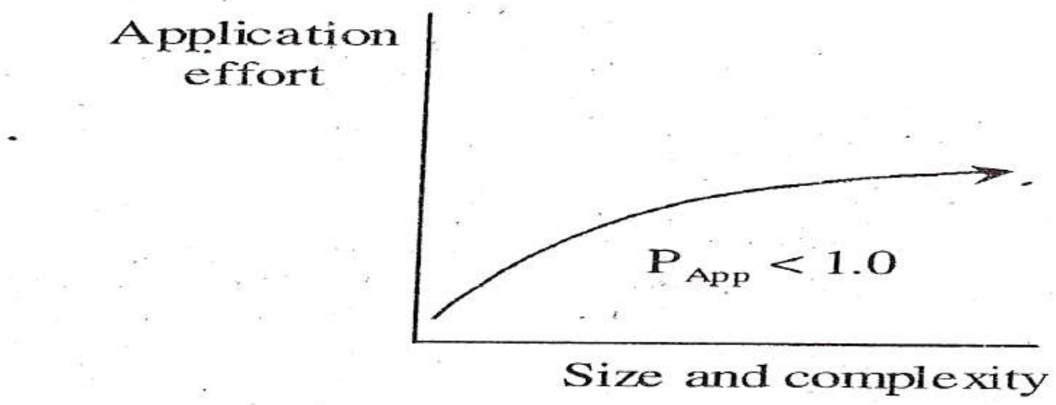


Figure: M-Month Production Increments

Total Effort = Func(TechnologyArch, ScaleArch, Quality Arch, Process Arch) + Func(TechnologyApp, ScaleApp, Quality App, Process App)

Total Time = Func(ProcessArch, EffortArch) + Func(ProcessApp, EffortApp,)

- The next generation infrastructure and environment automated various management activities with low effort. It relieves many of the sources of diseconomy of scale by reusing the common processes that are repetitive in a particular project. It also reuses the common outcomes of the project. The prior experience and matured processes utilized in these types of models eliminate the scrap rework sources. Here, the economics of scale will be affected.
- The architecture and applications of next generation cost models have difference scales and sized which represents the solution space. The size

can be computed inters of SLOC or megabytes of executable code while the scale can be computed in 0-terms of components, classes, processes or nodes. The requirement or use cases of solution space are different from that of a problem space. Moreover, there can be more than one solution to a problem. Where cost serves as a key discriminator. The cost estimates must be determined to find an optimal solution. If an optional solution is not found then different solution s need to be selected or to change the problem statement**.**

- A strict notation must be applied for design artifacts so, that the prediction of a design scale can be improved. The Next-generation software cost model should automate the process of measuring design scale directly from UML diagrams. There should be two major improvements. There are,
  - Separate architectural engineering stage from application production stage. This will yield greater accuracy and more precision of lifecycle estimate.
  - The use of rigorous design notations. This will enable the automation and standardization of scale measure so that they can be easily traced which helps to determine the total cost associated with production.
- The next generation software process has two potential breakthroughs, they are,
  - Certain integrated tools would be available that automates the information transition between the requirements, design, implementation and deployment elements. These tools facilitate roundtrip engineering between various artifacts of engineering.

It will reduce the four sets of fundamental technical artifacts into three sets. This is achieved by automating the activities related to human-generated source code so as to eliminate the need fro a separate implementation set

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

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

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

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

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

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

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

The maintenance cost will be almost double the development cost

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

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

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

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

Software productivity mainly relies on the type of people employed

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

The ratio of software to hardware cost is increasing.

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

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

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

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

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

60% of Errors are caught by walkthrough

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

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

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

MODERN PROCESS TRANSITIONS

Indications of a successful project transition to a modern culture

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

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

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

Tangible design and requirements

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

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

Assertive Demonstrations are prioritized

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

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

The performance of the project can be determined earlier in the life cycle.

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

Earlier increments will be adolescent

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

Artifacts tend to be insignificant at the early stages but proves to be the most significant in the later stages : The details of the artifacts should not be considered unless a stable and a useful baseline is obtained. This transition is accepted by the development team while the conventional contract monitors refuse this transition.

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

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

Everyone should focus on quality assurance

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

Performance issues crop up earlier in the projects life cycle

Earlier performance issues are a mature design process but resembles as an immature design. This transition is accepted by development engineers because it enables the evaluation of performance tradeoffs in subsequent releases.

Automation must be done with appropriate investments

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

Good software organizations should have good profit margins.

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

Characteristics of conventional and iterative software development Process

- The characteristics of the conventional software process are listed below:
    1. It evolves in the sequential order (requirement design-code-test).
    2. It gives the same preference to all the artifacts, components, requirements etc.
    3. It completes all the artifacts of a stage before moving to the other stage in the project life cycle.
    4. It achieves traceability with high-fidelity for al the artifacts present at each life cycle stage.
- The characteristics of the modern iterative development process framework are listed below:
    1. It continuously performs round-trip engineering of requirements, design, coding and testing at evolving levels of abstraction.
    2. It evolves the artifacts depending on the priorities of the risk management.
    3. It postpones the consistency analysis and completeness of the artifacts to the later stages in the life cycle.
    4. It achieves the significant drives (i.e. 20 percent) with high-fidelity during the initial stages of the life cycle.