

SOFTWARE PROCESS & PROJECT MANAGEMENT



Graphic Era

Deemed to be University

Accredited by NAAC with Grade A

NBA Accredited Programs in ECE, CSE & ME
Approved by AICTE, Ministry of HRD, Govt. of India

DEHRADUN, UTTARAKHAND, INDIA

Software Process & Project Management

OBJECTIVES:

The goals of the course are as follows:

1. To learn the importance of software process maturity & understand related concepts.
2. Understanding the specific roles within a software organization as related to project and process management
3. Understanding the basic infrastructure competences (e.g., process modeling and measurement)
4. Understanding the basic steps of project planning, project management, quality assurance, and process management and their relationships.

UNIT I

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.

UNIT II

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.

UNIT III

Workflows and Checkpoints of process

Software process workflows, Iteration workflows, Major milestones, Minor milestones, Periodic status assessments. **Process Planning** Work break down structures, Planning guidelines, cost and schedule estimating process, iteration planning process, Pragmatic planning.

UNIT IV

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, and metrics automation.

UNIT V

CCPDS-R Case Study and Future Software Project Management Practices

Modern Project Profiles, Next-Generation software Economics, Modern Process Transitions.

TEXT BOOKS:

1. Managing the Software Process, Watts S. Humphrey, Pearson Education.
2. Software Project Management, Walker Royce, Pearson Education.

REFERENCE BOOKS:

1. Effective Project Management: Traditional, Agile, Extreme, Robert Wysocki, Sixth edition, Wiley India, 2011.
2. An Introduction to the Team Software Process, Watts S. Humphrey, Pearson Education, 2000

OUTCOMES:

At the end of the course, the student shall be able to:

Apply suitable capability Maturity model for specific scenarios & determine the effectiveness.

Describe and determine the purpose and importance of project management from the perspectives of planning, tracking and completion of project

Compare and differentiate organization structures and project structures.

Implement a project to manage project schedule, expenses and resource with the application of suitable project management tools

UNIT I

Software Process Maturity 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 entropy (Webster's: a measure of the degree of disorder in a...system; entropy always increases and available energy diminishes in a closed system.). New methods must be carefully introduced and periodically monitored, or they 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. Prototyping 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

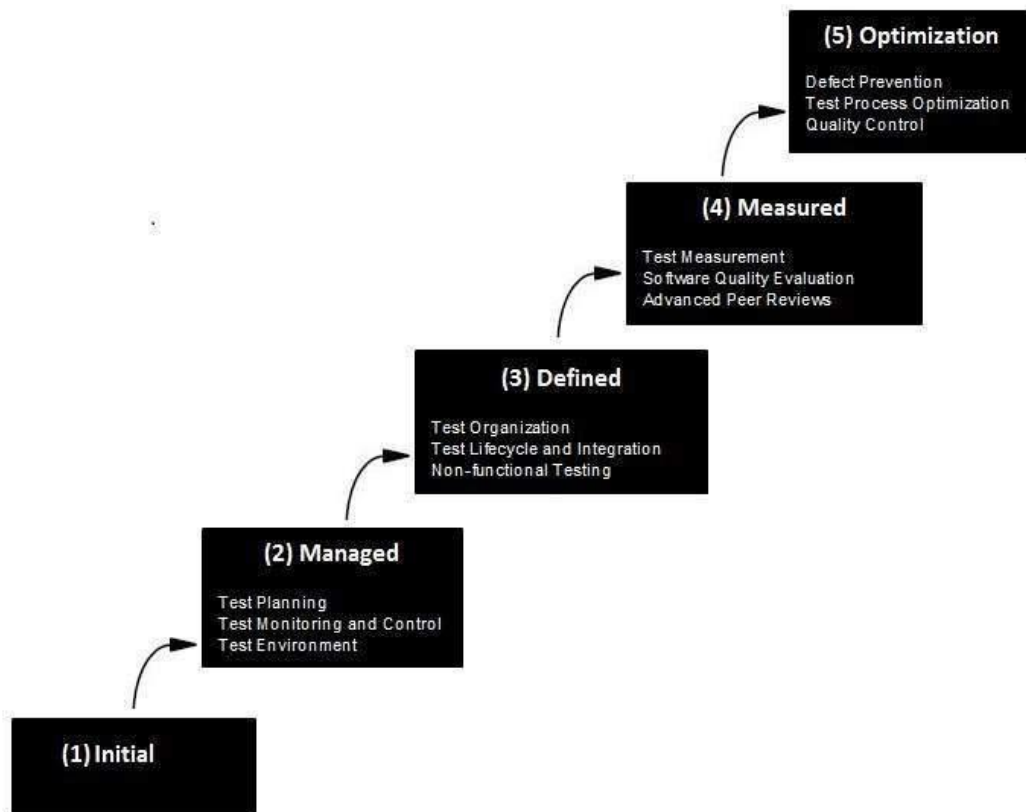
The process framework or reference model acts as an interface between the way the content is organized and the way work is performed. A uniform process model organized under a process reference model makes business modeling and systems designing much easier.

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 software development 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 quality-related 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.

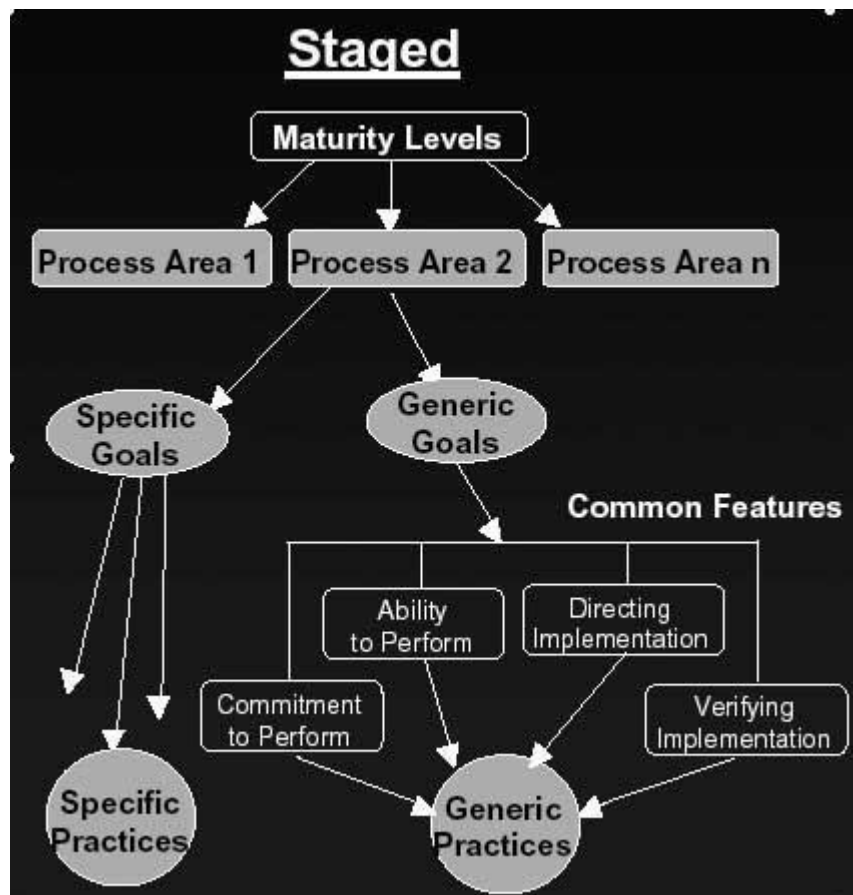
CMMI Staged Representation

- 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.

CMMI Staged Structure

Following picture illustrates CMMI Staged Model Structure.



Continuous Representation

Continuous representation is the approach used in the SECM and the IPD-CMM. This approach allows an organization to select a specific process area and make improvements based on it. The continuous representation uses Capability Levels to characterize improvement relative to an individual process area.

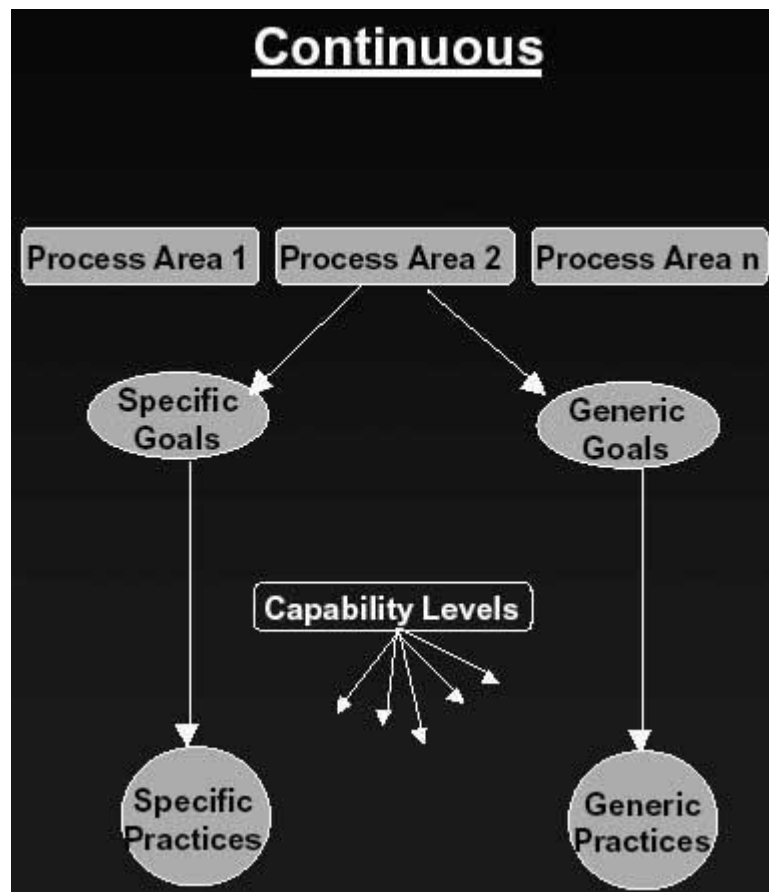
CMMI Continuous Representation

- Allows you to select the order of improvement that best meets your organization's business objectives and mitigates your organization's areas of risk.
- Enables comparisons across and among organizations on a process-area-by-processarea basis.
- Provides an easy migration from EIA 731 (and other models with a continuous representation) to CMMI.

Thus Continuous Representation provides flexibility to organizations to choose the processes for improvement, as well as the amount of improvement required.

CMMI Continuous Structure

The following picture illustrates the CMMI Continuous Model Structure.



Continuous vs Staged Representations

Continuous Representation	Staged Representation
Process areas are organized by process area categories.	Process areas are organized by maturity levels.
Improvement is measured using capability levels. Capability levels measure the maturity of a particular process across an organization; it ranges from 0 through 5.	Improvement is measured using maturity levels. Maturity levels measure the maturity of a set of processes across an organization; it ranges from 1 through 5.
There are two types of specific practices: base and advanced. All specific practices appear in the continuous representation.	There is only one type of specific practice. The concepts of base and advanced practices are not used. All specific practices appear in the staged representation except when a related base-advanced pair of practices appears in the continuous representation, in which case only the advanced practice appears in the staged representation.

Capability levels are used to organize the generic practices.	Common features are used to organize generic practices.
All generic practices are included in each process area.	Only the level 2 and level 3 generic practices are included.
Equivalent staging allows determination of a maturity level from an organization's achievement profile.	There is no need for an equivalence mechanism to back the continuous representation because each organization can choose what to improve and how much to improve using the staged representation.

- **Increase market share:** Market share is a result of many factors, including quality products and services, name identification, pricing, and image. Customers like to deal with suppliers who have a reputation for meeting their commitments.
- **Gain an industry-wide recognition for excellence:** The best way to develop a reputation for excellence is to consistently perform well on projects, delivering quality products and services within cost and schedule parameters. Having processes that conform to CMMI requirements can enhance that reputation.

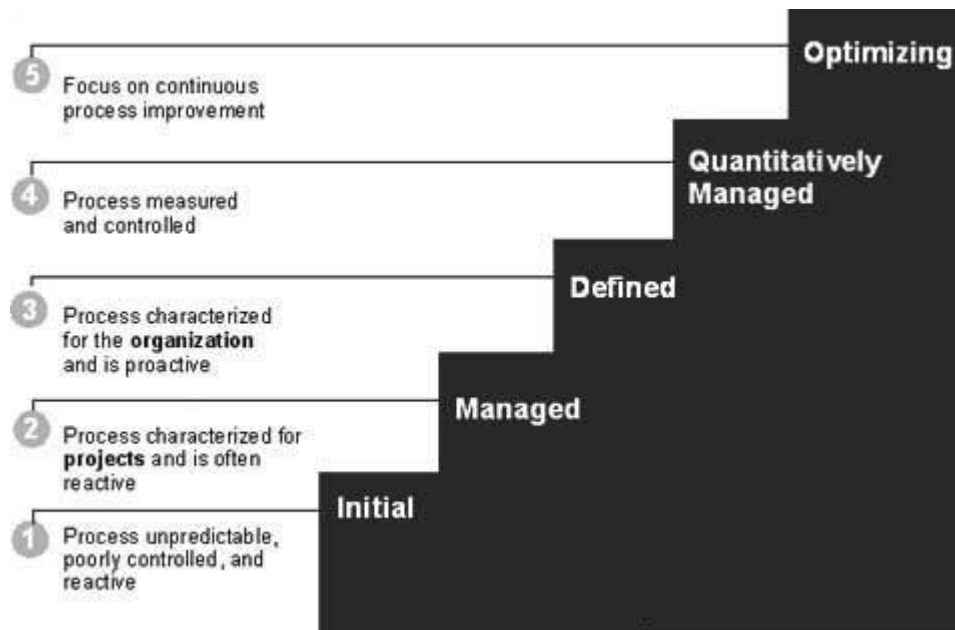
A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level provides a layer in the foundation for continuous process improvement.

CMMI models with staged representation, have five maturity levels designated by the numbers 1 through 5. They are –

- Initial
- Managed
- Defined
- Quantitatively Managed
- Optimizing

CMMI Staged Representation Maturity Levels

The following image shows the maturity levels in a CMMI staged representation.



Now we will learn the details about each maturity level. Next section will list down all the process areas related to these maturity levels.

Maturity Level Details

Maturity levels consist of a predefined set of process areas. The maturity levels are measured by the achievement of the **specific** and **generic goals** that apply to each predefined set of process areas. The following sections describe the characteristics of each maturity level in detail.

Maturity Level 1 Initial

At maturity level 1, processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment. Success in these organizations depend on the competence and heroics of the people in the organization and not on the use of proven processes.

Maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.

Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 Managed

At maturity level 2, an organization has achieved all the **specific** and **generic goals** of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.

The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points.

Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.

The work products and services satisfy their specified requirements, standards, and objectives.

Maturity Level 3 Defined

At maturity level 3, an organization has achieved all the **specific** and **generic goals** of the process areas assigned to maturity levels 2 and 3.

At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods.

A critical distinction between maturity level 2 and maturity level 3 is the scope of standards, process descriptions, and procedures. At maturity level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project).

At maturity level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit. The organization's set of standard processes includes the processes addressed at maturity level 2 and maturity level 3. As a result, the processes that are performed across the organization are consistent except for the differences allowed by the tailoring guidelines.

Another critical distinction is that at maturity level 3, processes are typically described in more detail and more rigorously than at maturity level 2. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

Maturity Level 4 Quantitatively Managed

At maturity level 4, an organization has achieved all the **specific goals** of the process areas assigned to maturity levels 2, 3, and 4 and the **generic goals** assigned to maturity levels 2 and 3.

At maturity level 4, sub-processes are selected that significantly contribute to the overall process performance. These selected sub-processes are controlled using statistical and other quantitative techniques.

Quantitative objectives for quality and process performance are established and used as criteria in managing the processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance are understood in statistical terms and are managed throughout the life of the processes.

For these processes, detailed measures of process performance are collected and statistically analyzed. Special causes of process variation are identified and, where appropriate, the sources of special causes are corrected to prevent future occurrences.

Quality and process performance measures are incorporated into the organization's measurement repository to support fact-based decision making in the future.

A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

Maturity Level 5 Optimizing

At maturity level 5, an organization has achieved all the **specific goals** of the process areas assigned to maturity levels 2, 3, 4, and 5 and the **generic goals** assigned to maturity levels 2 and 3.

Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.

This level focuses on continually improving process performance through both incremental and innovative technological improvements.

The quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement.

The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

Optimizing processes that are agile and innovative, depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning. Improvement of the processes is inherently a role that everybody has to play, resulting in a cycle of continual improvement.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the means of the process performance) to improve process performance (while maintaining statistical predictability) to achieve the established quantitative process-improvement objectives.

Maturity Levels Should Not be Skipped

Each maturity level provides a necessary foundation for effective implementation of processes at the next level.

- Higher level processes have less chance of success without the discipline provided by lower levels.
- The effect of innovation can be obscured in a noisy process.

Higher maturity level processes may be performed by organizations at lower maturity levels, with the risk of not being consistently applied in a crisis.

Maturity Levels and Process Areas

Here is a list of all the corresponding process areas defined for a software organization. These process areas may be different for different organization.

Level	Focus	Key Process Area	Result
5 Optimizing	Continuous Process Improvement	Organizational Innovation and Deployment Causal Analysis and Resolution	Highest Quality / Lowest Risk
4 Quantitatively Managed	Quantitatively Managed	Organizational Process Performance Quantitative Project Management	Higher Quality / Lower Risk
3 Defined	Process Standardization	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Mgmt (with IPPD extras) Risk Management Decision Analysis and Resolution Integrated Teaming (IPPD only) Org. Environment for Integration (IPPD only) Integrated Supplier Management (SS only)	Medium Quality / Medium Risk

2 Managed	Basic Project Management	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance Configuration Management	Low Quality / High Risk
1 Initial	Process is informal and Adhoc		Lowest Quality / Highest Risk

A capability level is a well-defined evolutionary plateau describing the organization's capability relative to a process area. A capability level consists of related specific and generic practices for a process area that can improve the organization's processes associated with that process area. Each level is a layer in the foundation for continuous process improvement.

Thus, capability levels are cumulative, i.e., a higher capability level includes the attributes of the lower levels.

In CMMI models with a continuous representation, there are six capability levels designated by the numbers 0 through 5.

- 0 – Incomplete
- 1 – Performed
- 2 – Managed
- 3 – Defined
- 4 – Quantitatively Managed
- 5 – Optimizing

A short description of each capability level is as follows –

Capability Level 0: Incomplete

An "incomplete process" is a process that is either not performed or partially performed. One or more of the specific goals of the process area are not satisfied and no generic goals exist for this level since there is no reason to institutionalize a partially performed process.

This is tantamount to Maturity Level 1 in the staged representation.

Capability Level 1: Performed

A Capability Level 1 process is a process that is expected to perform all of the Capability Level 1 specific and generic practices. Performance may not be stable and may not meet specific objectives such as quality, cost, and schedule, but useful work can be done. This is only a start, or baby-step, in process improvement. It means that you are doing something but you cannot prove that it is really working for you.

Capability Level 2: Managed

A managed process is planned, performed, monitored, and controlled for individual projects, groups, or stand-alone processes to achieve a given purpose. Managing the process achieves both the model objectives for the process as well as other objectives, such as cost, schedule, and quality. As the title of this level indicates, you are actively managing the way things are done in your organization. You have some metrics that are consistently collected and applied to your management approach.

Note – metrics are collected and used at all levels of the CMMI, in both the staged and continuous representations. It is a bitter fallacy to think that an organization can wait until Capability Level 4 to use the metrics.

Capability Level 3: Defined

A capability level 3 process is characterized as a "defined process." A defined process is a managed (capability level 2) process that is tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets.

Capability Level 4: Quantitatively Managed

A capability level 4 process is characterized as a "quantitatively managed process." A quantitatively managed process is a defined (capability level 3) process that is controlled using statistical and other quantitative techniques. Quantitative objectives for quality and process performance are established and used as criteria in managing the process. Quality and process performance is understood in statistical terms and is managed throughout the life of the process.

Capability Level 5: Optimizing

An optimizing process is a quantitatively managed process that is improved, based on an understanding of the common causes of process variation inherent to the process. It focuses on continually improving process performance through both incremental and innovative improvements. Both the defined processes and the organization's set of standard processes are the targets of improvement activities.

Capability Level 4 focuses on establishing baselines, models, and measurements for process performance. Capability Level 5 focuses on studying performance results across the organization or entire enterprise, finding common causes of problems in how the work is done (the process[es] used), and fixing the problems in the process. The fix would include updating the process documentation and training involved where the errors were injected.

Organization of Process Areas in Continuous Representation

Category	Process Area
Project Management	<ul style="list-style-type: none"> • Project Planning • Project Monitoring and Control • Supplier Agreement Management • Integrated Project Management(IPPD) • Integrated Supplier Management (SS) • Integrated Teaming (IPPD) • Risk Management Quantitative Project Management
Support	<ul style="list-style-type: none"> • Configuration Management • Process and Product Quality Assurance • Measurement and Analysis Causal Analysis and Resolution • Decision Analysis and Resolution • Organizational Environment for Integration (IPPD)
Engineering	<ul style="list-style-type: none"> • Requirements Management • Requirements Development • Technical Solution • Product Integration • Verification • Validation
Process Management	<ul style="list-style-type: none"> • Organizational Process Focus • Organizational Process Definition • Organizational Training • Organizational Process Performance • Organizational Innovation and Deployment
<u>PCMM:</u>	<ul style="list-style-type: none"> •

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.

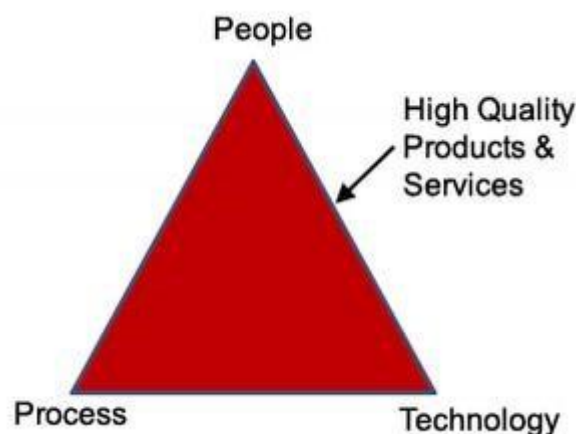
The P-CMM framework enables organisations to incrementally focus on key process areas and to lay foundations for improvement in workforce practices. Unlike other HR models, P-CMM requires that key process areas, improvements, interventions, policies, and procedures are institutionalised across the organisation — 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



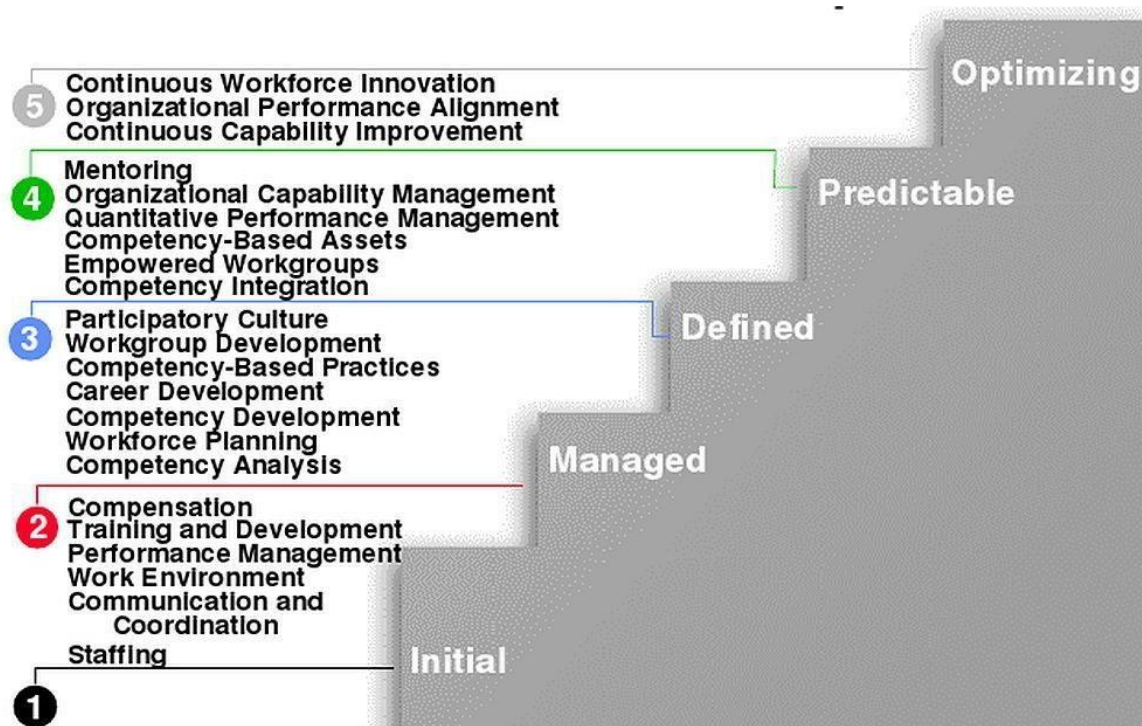
Thus, these 3Ps of PCMM are comparable to ITIL's 4Ps: ***People***, ***Processes***, ***Products*** (tools and technology) and ***Partners*** (suppliers, vendors, and outsourcing organisations). P-CMM is characterised by a holistic approach to people-related issues. Instead of looking at traditional Human Resource interventions in a reactionary scrappy fashion. 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 institutionalized 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.

P-CMM Maturity Levels

Like other staged maturity models of the CMMI product family developed at the Software Engineering Institute (SEI) at Carnegie-Mellon University, the P-CMM consists of maturity levels that establish successive foundations for continuously improving workforce competencies.

These range from initial (Level 1), where workforce practices are performed inconsistently or ritualistically and frequently fail to achieve their intended purpose, to the optimised level (Level 5), where everyone in the organisation is focused on continuously improving their Capability and the organisation's workforce practices.



The five maturity levels of the People CMM are:

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 Work Environment	Staffing

- Initial.**
- Repeatable.** The key process areas at Level 2 focus on instilling basic discipline into workforce activities. They are:
 - Work Environment
 - Communications
 - Staffing
 - Performance Management
 - Training
 - Compensation
- Defined.** The key process areas at Level 3 address issues surrounding the identification of the organisation's primary competencies and aligning its people management activities with them. They are:
 - Knowledge and Skills Analysis
 - Workforce Planning
 - Competency Development
 - Career Development
 - Competency-Based Practices
 - Participatory Culture
- Managed.** The key process areas at Level 4 focus on quantitatively managing organisational growth in people management capabilities and in establishing competency-based teams. They are:
 - Mentoring
 - Team Building
 - Team-Based Practices
 - Organisational Competency Management
 - Organisational Performance Alignment

5. **Optimising.** The key process areas at Level 5 cover the issues that address continuous improvement of methods for developing competency, at both the organisational and the individual level. They are:
 - Personal Competency Development
 - Coaching
 - Continuous Workforce Innovation

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.
- **PSP0, PSP0.1 (Introduces process discipline and measurement)**
- PSP0 has 3 phases: planning, development (design, code, compile, test) and a post mortem. A baseline is established of current process measuring: time spent on programming, faults injected/removed, size of a program. In a post mortem, the engineer ensures all data for the projects has been properly recorded and analysed. PSP0.1 advances the process by adding a coding standard, a size measurement and the development of a personal process improvement plan (PIP). In the PIP, the engineer records ideas for improving his own process.
- **PSP1, PSP1.1 (Introduces estimating and planning)**

- Based upon the baseline data collected in PSP0 and PSP0.1, the engineer estimates how large a new program will be and prepares a test report (PSP1). Accumulated data from previous projects is used to estimate the total time. Each new project will record the actual time spent. This information is used for task and schedule planning and estimation (PSP1.1).
- **PSP2, PSP2.1 (Introduces quality management and design)**
- PSP2 adds two new phases: design review and code review. Defect prevention and removal of them are the focus at the PSP2. Engineers learn to evaluate and improve their process by measuring how long tasks take and the number of defects they inject and remove in each phase of development. Engineers construct and use checklists for design and code reviews. PSP2.1 introduces design specification and analysis techniques
- (PSP3 is a legacy level that has been superseded by TSP.)

One of the core aspects of the PSP is using historical data to analyze and improve process performance. PSP data collection is supported by four main elements:

- Scripts
- Measures
- Standards
- Forms

The PSP scripts provide expert-level guidance to following the process steps and they provide a framework for applying the PSP measures. The PSP has four core measures:

- Size – the size measure for a product part, such as lines of code (LOC).
- Effort – the time required to complete a task, usually recorded in minutes.
- Quality – the number of defects in the product.
- Schedule – a measure of project progression, tracked against planned and actual completion dates.

Applying standards to the process can ensure the data is precise and consistent. Data is logged in forms, normally using a PSP software tool. The SEI has developed a PSP tool and there are also open source options available, such as Process Dashboard.

The key data collected in the PSP tool are time, defect, and size data – the time spent in each phase; when and where defects were injected, found, and fixed; and the size of the product parts. Software developers use many other measures that are derived from these three basic measures to understand and improve their performance. Derived measures include:

- estimation accuracy (size/time)
- prediction intervals (size/time)
- time in phase distribution
- defect injection distribution
- defect removal distribution
- productivity
- reuse percentage
- cost performance index
- planned value
- earned value
- predicted earned value
- defect density

- defect density by phase
- defect removal rate by phase
- defect removal leverage
- review rates
- process yield
- phase yield
- failure cost of quality (COQ)
- appraisal COQ
- appraisal/failure COQ ratio

Planning and tracking

Logging time, defect, and size data is an essential part of planning and tracking PSP projects, as historical data is used to improve estimating accuracy.

The PSP uses the [PROxy-Based Estimation](#) (PROBE) method to improve a developer's estimating skills for more accurate project planning. For project tracking, the PSP uses the [earned value](#) method.

The PSP also uses statistical techniques, such as correlation, linear regression, and standard deviation, to translate data into useful information for improving estimating, planning and quality. These statistical formulas are calculated by the PSP tool.

Using the PSP

The PSP is intended to help a developer improve their personal process; therefore PSP developers are expected to continue adapting the process to ensure it meets their personal needs.

PSP and the TSP

In practice, PSP skills are used in a TSP team environment. TSP teams consist of PSP-trained developers who volunteer for areas of project responsibility, so the project is managed by the team itself. Using personal data gathered using their PSP skills; the team makes the plans, the estimates, and controls the quality.

Using PSP process methods can help TSP teams to meet their schedule commitments and produce high quality software. For example, according to research by Watts Humphrey, a third of all software projects fail,^[3] but an SEI study on 20 TSP projects in 13 different organizations found that TSP teams missed their target schedules by an average of only six percent.^[4]

Successfully meeting schedule commitments can be attributed to using historical data to make more accurate estimates, so projects are based on realistic plans – and by using PSP quality methods, they produce low-defect software, which reduces time spent on removing defects in later phases, such as integration and acceptance testing.

PSP and other methodologies[\[edit\]](#)

The PSP is a personal process that can be adapted to suit the needs of the individual developer. It is not specific to any programming or design methodology; therefore it can be used with different methodologies, including [Agile software development](#).

Software engineering methods can be considered to vary from predictive through adaptive. The PSP is a predictive methodology, and Agile is considered adaptive, but despite their differences, the TSP/PSP and Agile share several concepts and approaches – particularly in regard to team organization. They both enable the team to:

- Define their goals and standards.
- Estimate and schedule the work.
- Determine realistic and attainable schedules. □ Make plans and process improvements.

Both Agile and the TSP/PSP share the idea of team members taking responsibility for their own work and working together to agree on a realistic plan, creating an environment of trust and accountability. However, the TSP/PSP differs from Agile in its emphasis on documenting the process and its use of data for predicting and defining project schedules.

Quality[\[edit\]](#)

High-quality software is the goal of the PSP, and quality is measured in terms of defects. For the PSP, a quality process should produce low-defect software that meets the user needs.

The PSP phase structure enables PSP developers to catch defects early. By catching defects early, the PSP can reduce the amount of time spent in later phases, such as Test.

The PSP theory is that it is more economical and effective to remove defects as close as possible to where and when they were injected, so software engineers are encouraged to conduct personal reviews for each phase of development. Therefore, the PSP phase structure includes two review phases:

- Design Review
- Code Review

To do an effective review, you need to follow a structured review process. The PSP recommends using checklists to help developers to consistently follow an orderly procedure.

The PSP follows the premise that when people make mistakes, their errors are usually predictable, so PSP developers can personalize their checklists to target their own common errors. Software engineers are also expected to complete process improvement proposals, to identify areas of weakness in their current performance that they should target for improvement. Historical project data, which exposes where time is spent and defects introduced, help developers to identify areas to improve.

PSP developers are also expected to conduct personal reviews before their work undergoes a peer or team review.

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 :

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, monitor, 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

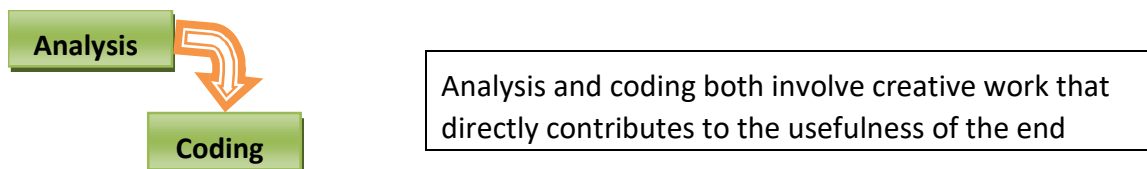
Most software engineering texts present the waterfall model as the source of the "conventional" software process.

IN THEORY

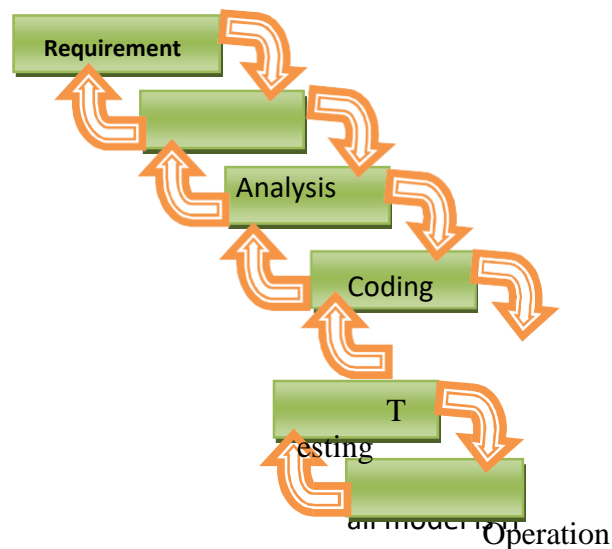
It provides an insightful and concise summary of conventional software management. Three main primary points are

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.



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 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-horning of no optimal fixes, with no time for redesign.
- A very fragile, unmentionable product delivered late.

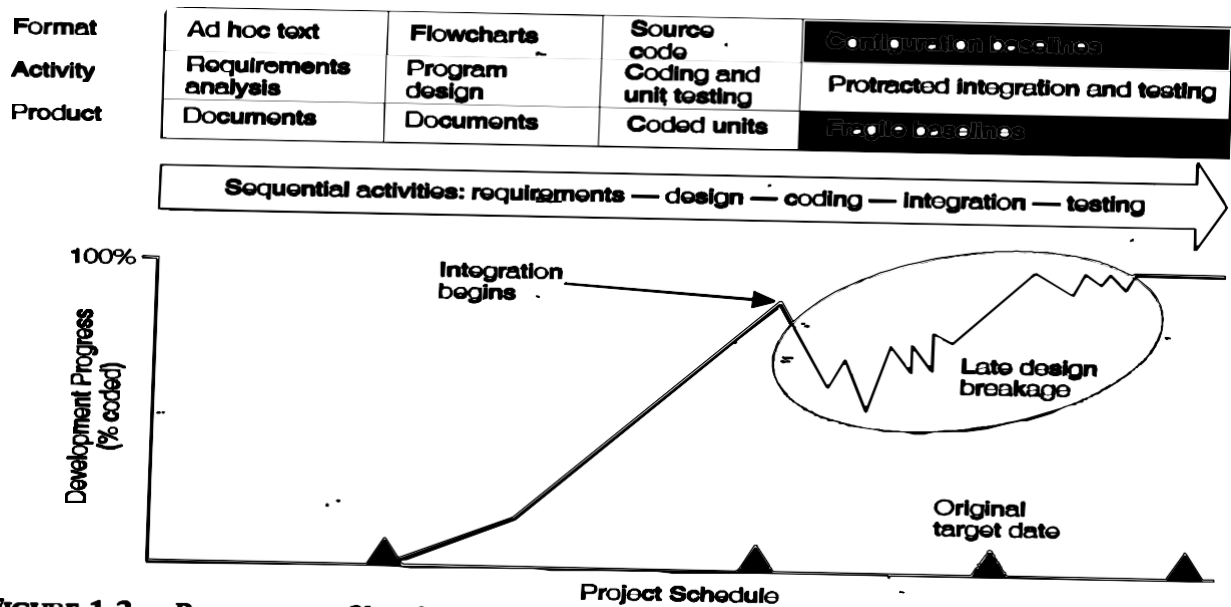


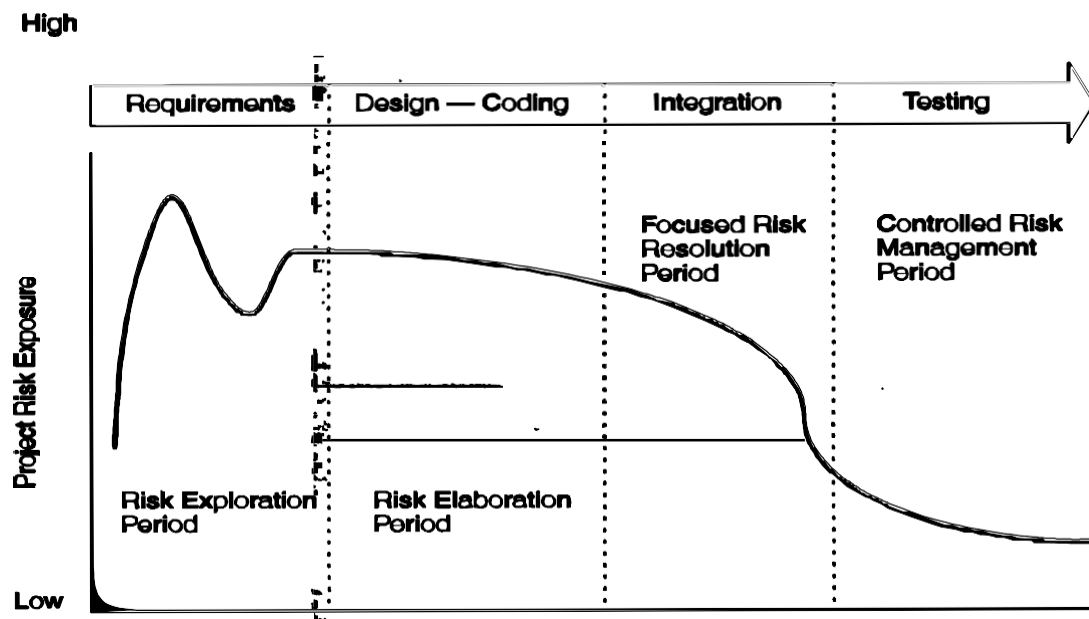
FIGURE 1-2. *Progress profile of a conventional software project*

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.

TABLE 1-1. *Expenditures by activity for a conventional software project*

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

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.



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.

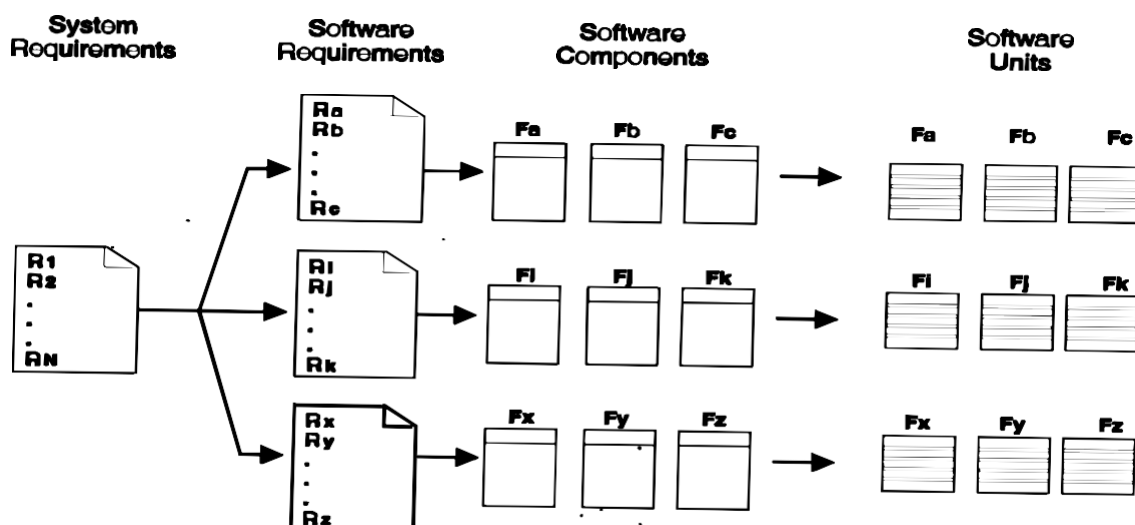


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

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. Table 1-2 summarizes the results of a typical design review.

TABLE 1-2. Results of conventional software project design reviews

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

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 systems) cost 9 times as much.
9. Walkthroughs catch **60%** of the errors
10. **80%** of the contribution comes from **20%** of the contributors.

2. 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:

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

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

Figure 2-1 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:

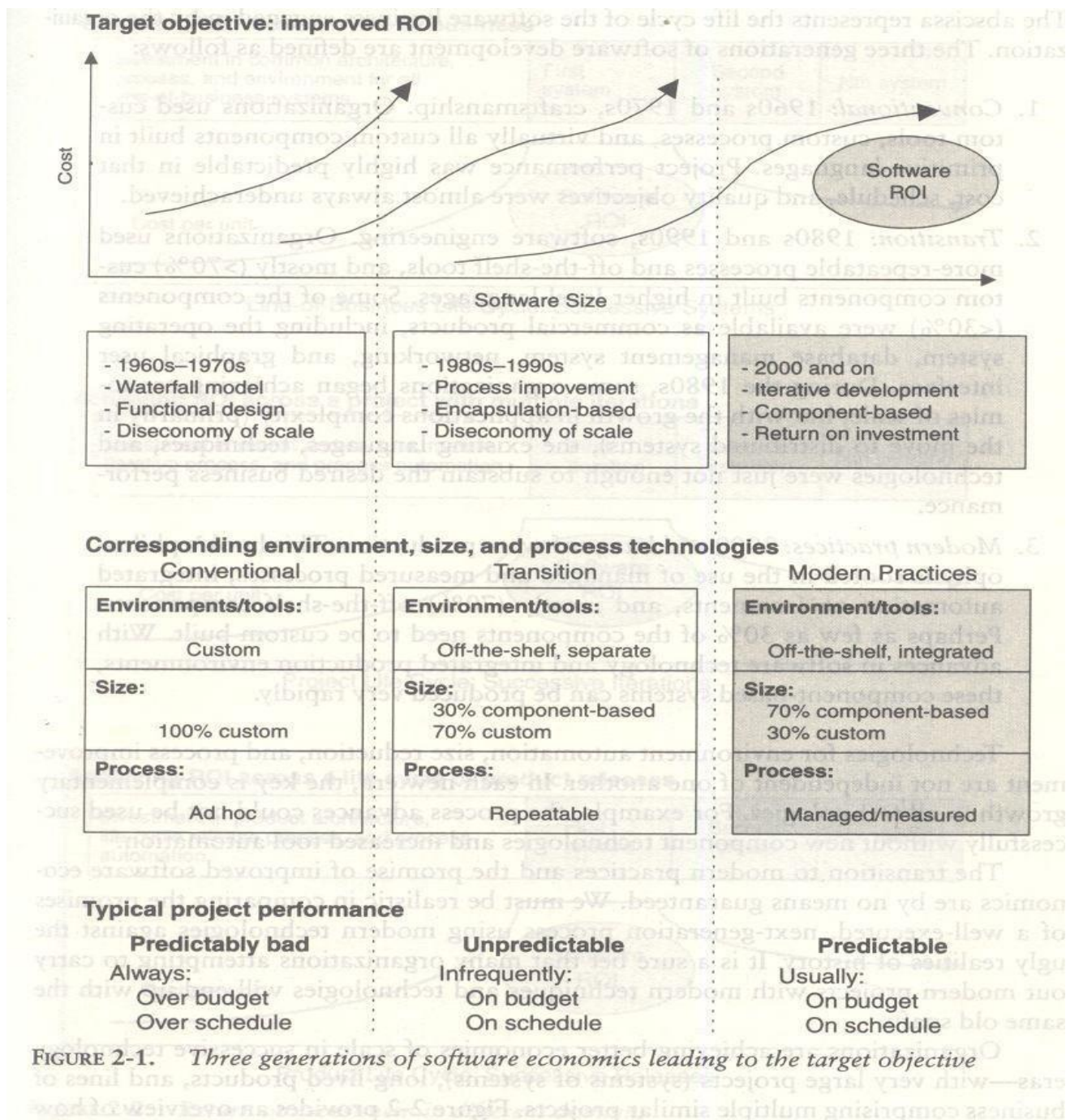
- 1) **Conventional:** 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.

- 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

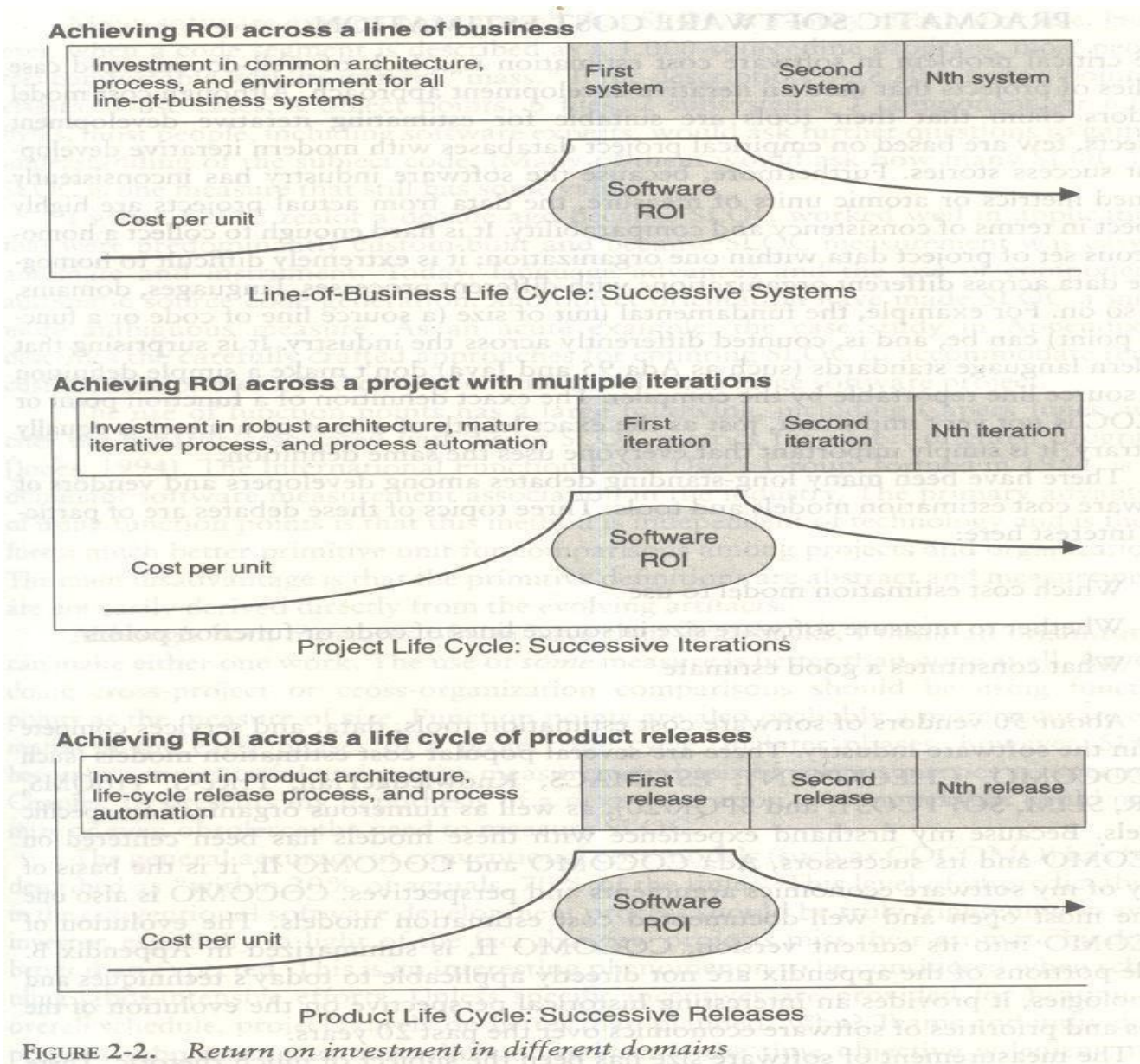
components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.

- 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.



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



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), COCOMO 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 actuals, 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 2-3 is not all bad. In fact, it is absolutely necessary to analyze the cost risks and understand the sensitivities and trade-offs objectively. It forces the software project manager to examine the risks associated with achieving the target costs and to discuss this information with other stakeholders.

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.

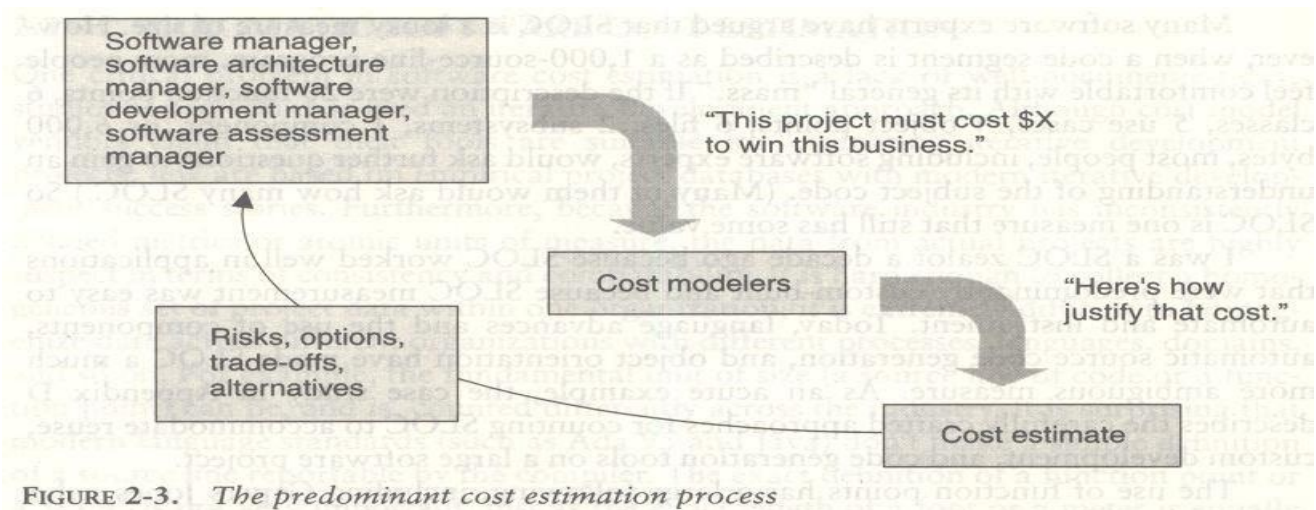


FIGURE 2-3. The predominant cost estimation process

3. 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	Higher order languages (C++, Ada 95, Java, Visual Basic, etc.)
Abstraction and component-based development technologies	Object-oriented (analysis, design, programming)
	Reuse
	Commercial components
Process	Iterative development
Methods and techniques	Process maturity models
	Architecture-first development
	Acquisition reform
Personnel	Training and personnel skill development
People factors	Teamwork
	Win-win cultures
Environment	Integrated tools (visual modeling, compiler, editor, debugger, change management, etc.)
Automation technologies and tools	Open systems
	Hardware platform performance
	Automation of coding, documents, testing, analyses
Quality	Hardware platform performance
Performance, reliability, accuracy	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 (UFPs¹) 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
-----------	--------------

¹ Function point metrics provide a standardized method for measuring the various functions of a software application.

The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries.

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

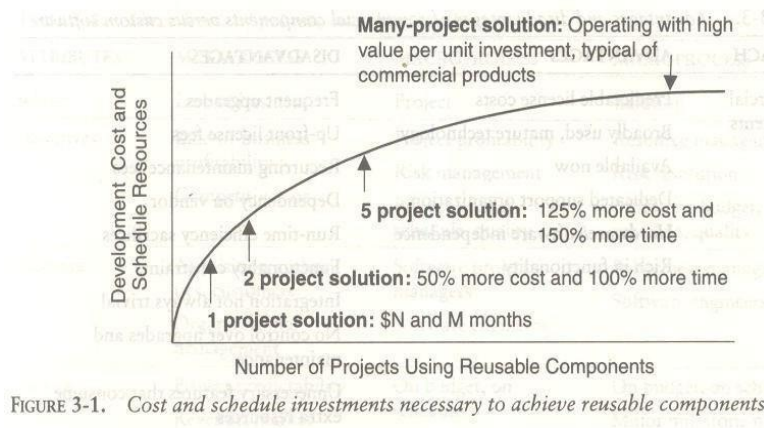
Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages. With reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and 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.



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	Frequent upgrades
	Broadly used, mature technology	Up-front license fees
	Available now	Recurring maintenance fees
	Dedicated support organization	Dependency on vendor
	Hardware/software independence	Run-time efficiency sacrifices
	Rich in functionality	Functionality constraints
		Integration not always trivial
		No control over upgrades and maintenance
		Unnecessary features that consume extra resources
		Often inadequate reliability and stability
Custom development		Multiple-vendor incompatibilities
	Complete change freedom	Expensive, unpredictable development
	Smaller, often simpler implementations	Unpredictable availability date
	Often better performance	Undefined maintenance model
	Control of development and enhancement	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	Project profitability	Resource management
	Competitiveness	Risk management	Risk resolution
Audience	Acquisition authorities, customers	Project budget, schedule, quality	Milestone budget, schedule, quality
	Organizational management	Software project managers	Subproject managers
Metrics		Software engineers	Software engineers
	Project predictability	On budget, on schedule	On budget, on schedule
	Revenue, market share	Major milestone success	Major milestone progress
Concerns		Project scrap and rework	Release/iteration scrap and rework
	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 instrumenting 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 describes 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, straightforward and benign
Design errors	Discovered late	Resolved early
Automation	Mostly error-prone manual procedures	Mostly automated, error-free evolution of artifacts
Resource adequacy	Unpredictable	Predictable
Schedules	Overconstrained	Tunable to quality, performance, and technology
Target performance	Paper-based analysis or separate simulation	Executing prototypes, early performance feedback, quantitative understanding
Software process rigor	Document-based	Managed, measured, and tool-supported

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 demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people
3. **Give products to customers early.** No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it
4. **Determine the problem before writing the requirements.** When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution
5. **Evaluate design alternatives.** After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model.** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases.** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle.
8. **Minimize intellectual distance.** To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure
9. **Put techniques before tools.** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer
10. **Get it right before you make it faster.** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding
11. **Inspect code.** Inspecting the detailed design and code is a much better way to find errors than testing
12. **Good management is more important than good technology.** Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success.** Highly skilled people with appropriate experience, talent, and training are key.
14. **Follow with care.** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.

15. **Take responsibility.** When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.
16. **Understand the customer's priorities.** It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more 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.)

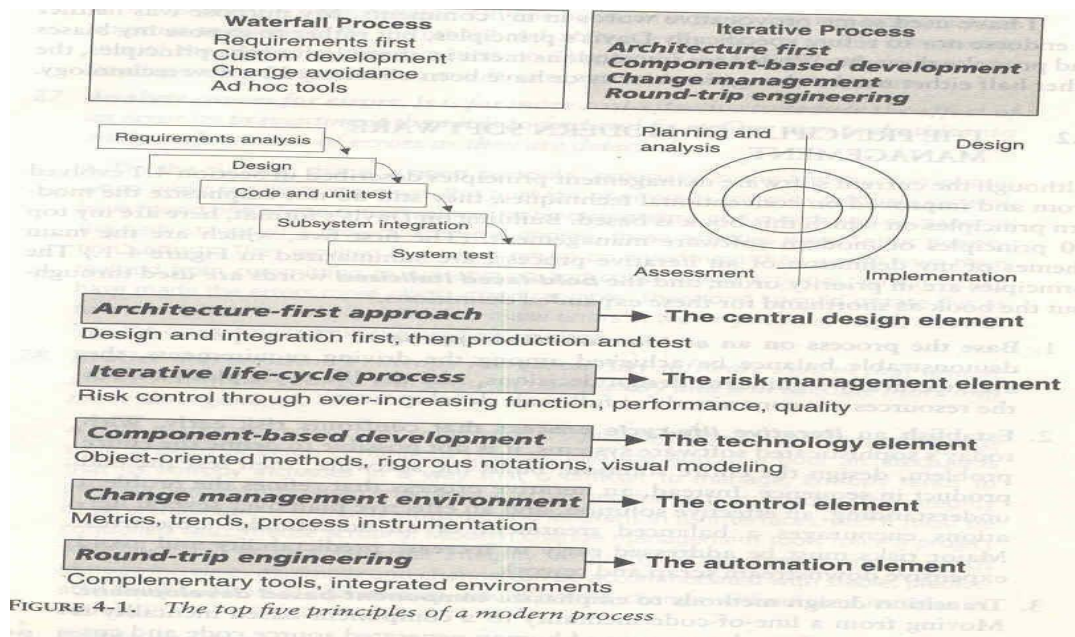
Base the process on an architecture-first approach. This requires that a demonstrable balance be achieved among the driving requirements, the architecturally significant design decisions, and the life-cycle plans before the resources are committed for full-scale development.

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.

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.



5. **Enhance change freedom through tools that support round-trip engineering.** Roundtrip 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 COCOMO 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 elaborated 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**

Life cycle phases

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 componentbased 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

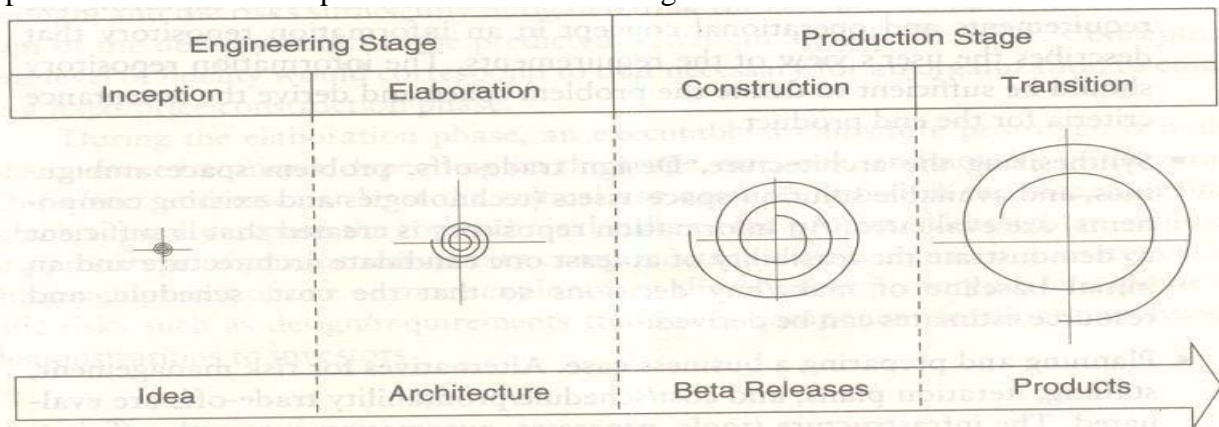


FIGURE 5-1. The phases of the life-cycle process

INCEPTION PHASE

The overriding goal of the inception phase is to achieve concurrence among stakeholders on the lifecycle 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 scenarios
- Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase)
- Estimating potential risks (sources of unpredictability)

ESSENTIAL ACTIVITIES

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

PRIMARY EVALUATION CRITERIA

- Do all stakeholders concur on the scope definition and cost and schedule estimates?
- Are requirements understood, as evidenced by the fidelity of the critical use cases?
- Are the cost and schedule estimates, priorities, risks, and development processes credible?
- Do the depth and breadth of an architecture prototype demonstrate the preceding criteria? (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. *Artifact* represents cohesive information that typically is developed and reviewed as a single entity.

Life-cycle software artifacts are organized into five distinct sets that are roughly partitioned by the underlying language of the set: 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.

Requirements Set	Design Set	Implementation Set	Deployment Set
1. Vision document 2. Requirements model(s)	1. Design model(s) 2. Test model 3. Software architecture description	1. Source code baselines 2. Associated compile-time files 3. Component executables	1. Integrated product executable baselines 2. Associated run-time files 3. User manual
Management Set			
Planning Artifacts		Operational Artifacts	
1. Work breakdown structure 2. Business case 3. Release specifications 4. Software development plan		5. Release descriptions 6. Status assessments 7. Software change order database 8. Deployment documents 9. Environment	

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.

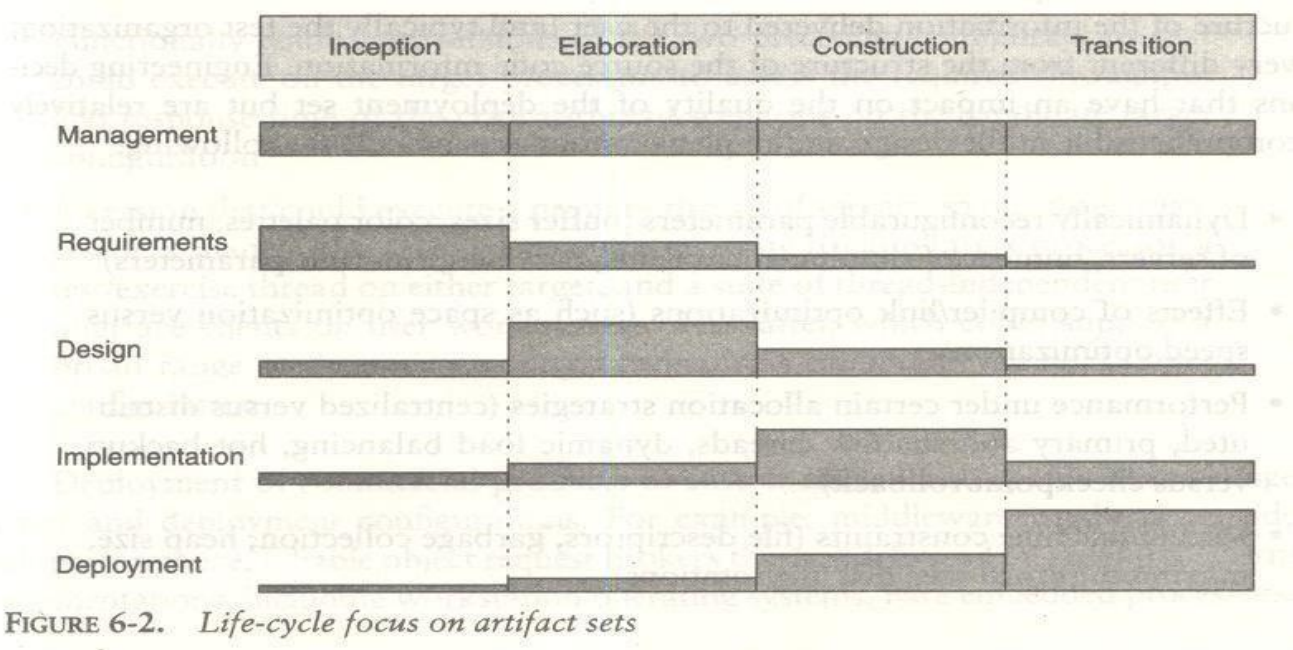
Deployment sets are evaluated, assessed, and measured through a combination of the following:

- 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.



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.

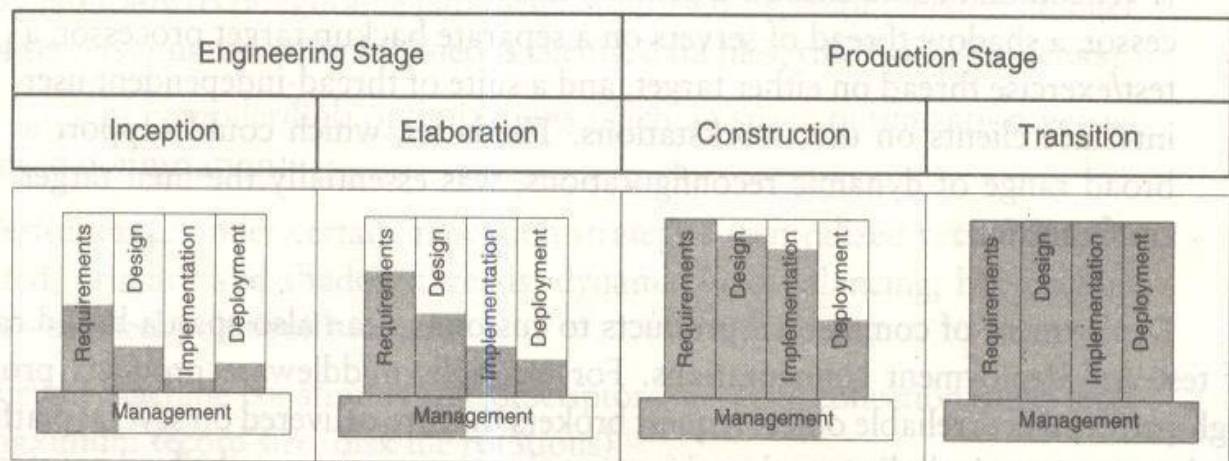


FIGURE 6-3. Life-cycle evolution of the artifact sets

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

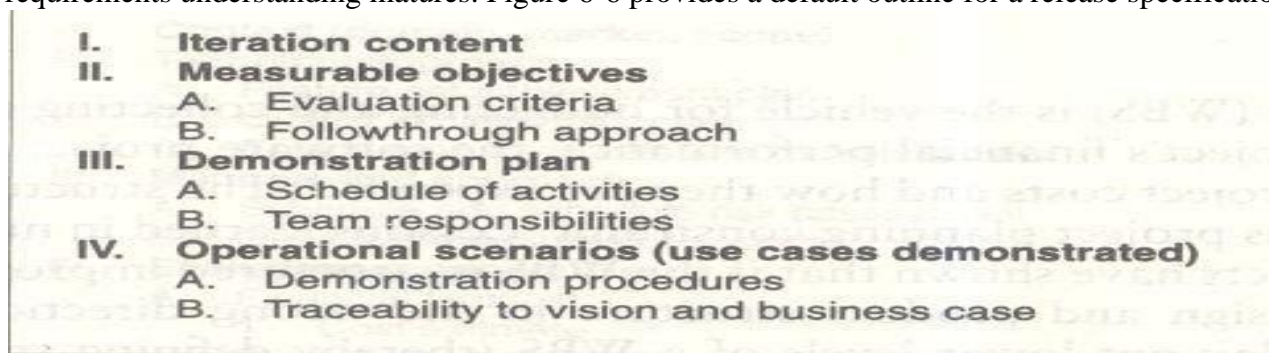
Work breakdown structure (WBS) is the vehicle for budgeting and collecting costs. To monitor and control a project's financial performance, the software project manager must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

Software Change Order Database

Managing change is one of the fundamental primitives of an iterative development process. With greater change freedom, a project can iterate more productively. This flexibility increases the content, quality, and number of iterations that a project can achieve within a given schedule. Change freedom has been achieved in practice through automation, and today's iterative development environments carry the burden of change management. Organizational processes that depend on manual change management techniques have encountered major inefficiencies.

Release Specifications

The scope, plan, and objective evaluation criteria for each baseline release are derived from the vision statement as well as many other sources (make/buy analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds). These artifacts are intended to evolve along with the process, achieving greater fidelity as the life cycle progresses and requirements understanding matures. Figure 6-6 provides a default outline for a release specification

A diagram showing a hierarchical outline for a release specification. It consists of four main sections, each with a Roman numeral and a title. Section I is 'Iteration content'. Section II is 'Measurable objectives' and contains two sub-items: 'A. Evaluation criteria' and 'B. Followthrough approach'. Section III is 'Demonstration plan' and contains two sub-items: 'A. Schedule of activities' and 'B. Team responsibilities'. Section IV is 'Operational scenarios (use cases demonstrated)' and contains two sub-items: 'A. Demonstration procedures' and 'B. Traceability to vision and business case'.

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

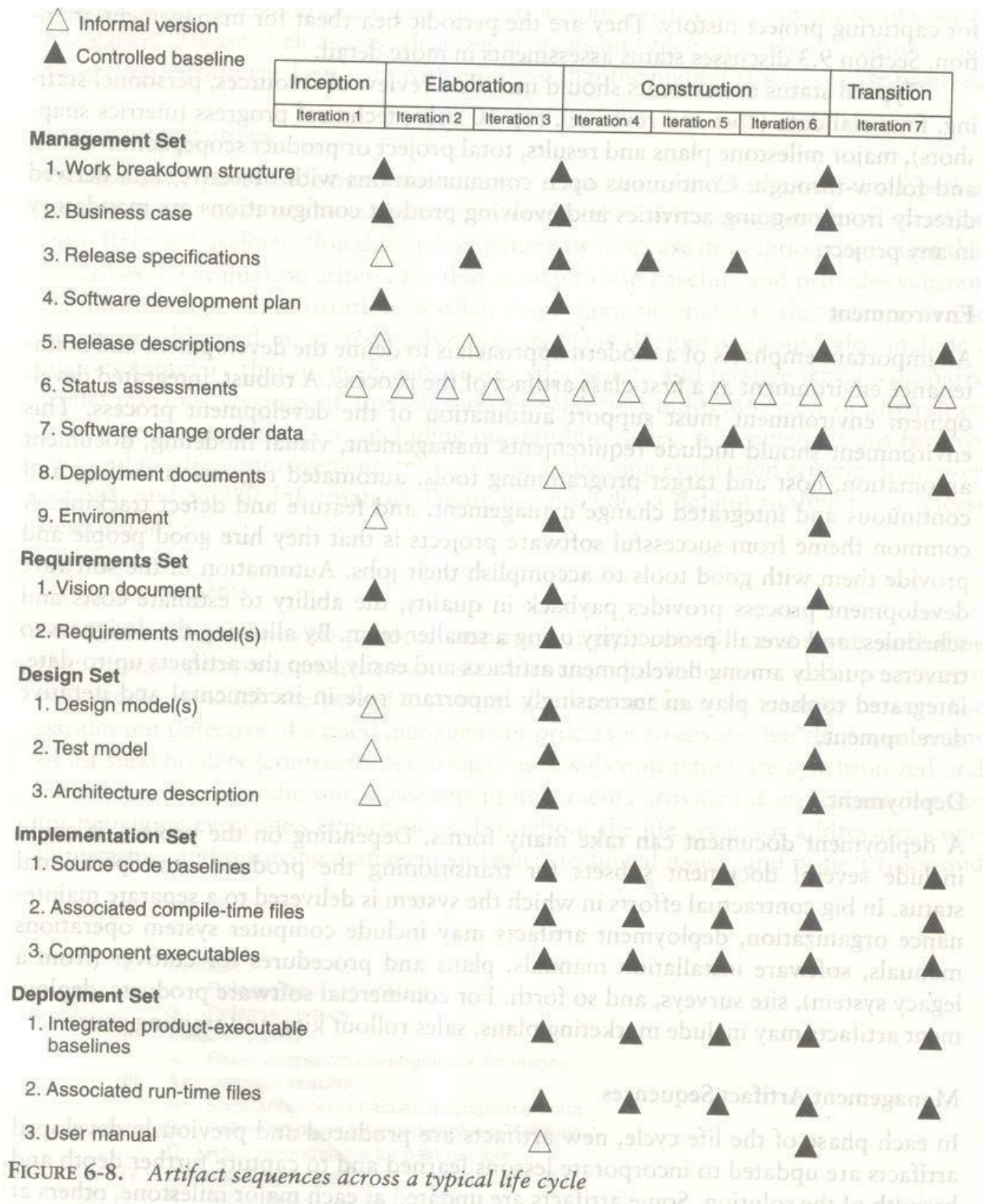
An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support automation of the development process. This environment should include requirements management, visual 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.



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 appendices
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*

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

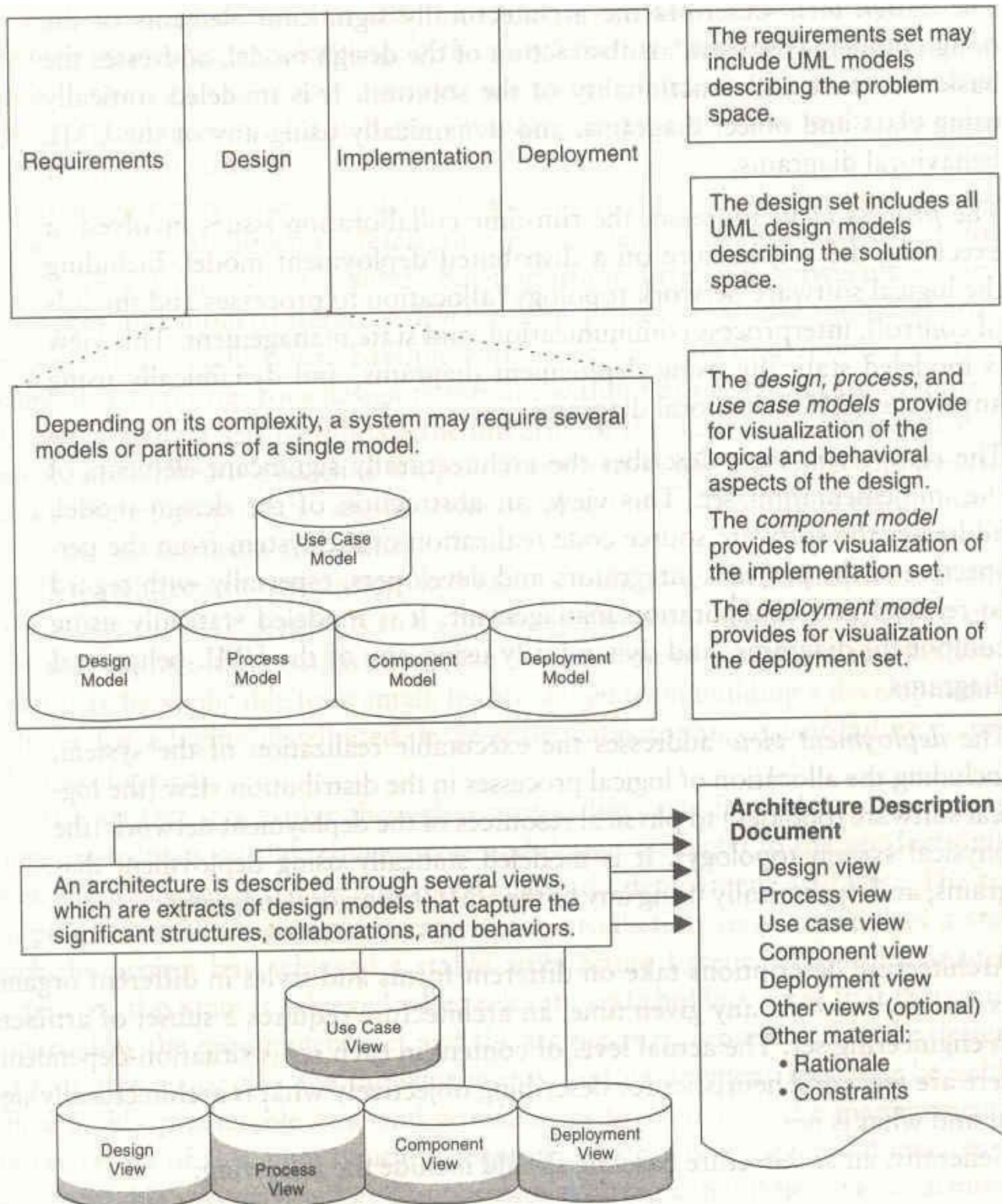


FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

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, spreadsheets, Visual Basic, C++, and Ada 95), visualization tools, and the Web are rapidly making it economically feasible for all stakeholders to exchange information electronically.
- **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.

UNIT III Workflows and Checkpoints of process

Workflow of the process

SOFTWARE PROCESS WORKFLOWS

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

1. Management workflow: controlling the process and ensuring win conditions for all stakeholders
2. Environment workflow: automating the process and evolving the maintenance environment
3. Requirements workflow: analyzing the problem space and evolving the requirements artifacts
4. Design workflow: modeling the solution and evolving the architecture and 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

Figure 8-1 illustrates the relative levels of effort expected across the phases in each of the top-level workflows.

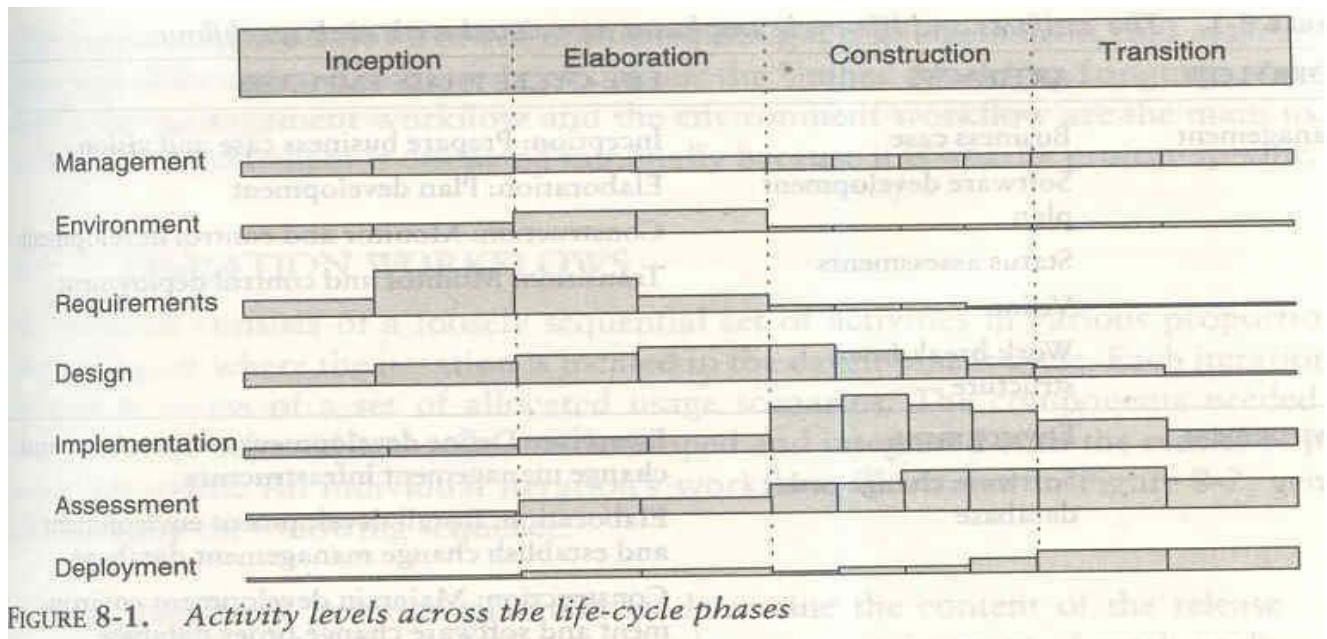


FIGURE 8-1. Activity levels across the life-cycle phases

Table 8-1 shows the allocation of artifacts and the emphasis of each workflow in each of the life-cycle phases of inception, elaboration, construction, and transition.

TABLE 8-1. *The artifacts and life-cycle emphases associated with each workflow*

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

ITERATION WORKFLOWS

Iteration consists of a loosely sequential set of activities in various proportions, depending on where the iteration is located in the development cycle. Each iteration is defined in terms of a set of allocated usage scenarios. An individual iteration's workflow, illustrated in Figure 8-2, generally includes the following sequence:

- Management: iteration planning to determine the content of the release and develop the detailed plan for the iteration; assignment of work packages, or tasks, to the development team

- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test, and environment components

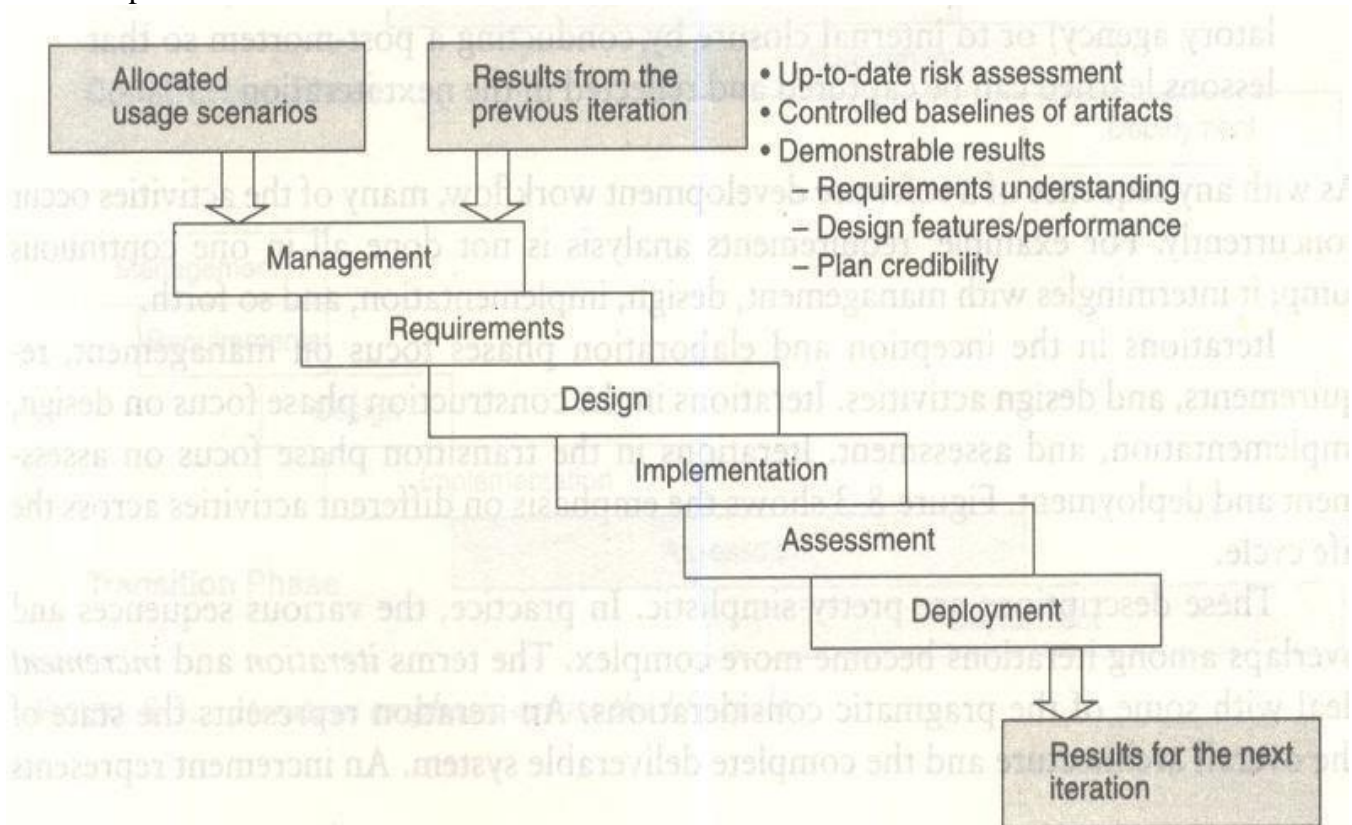
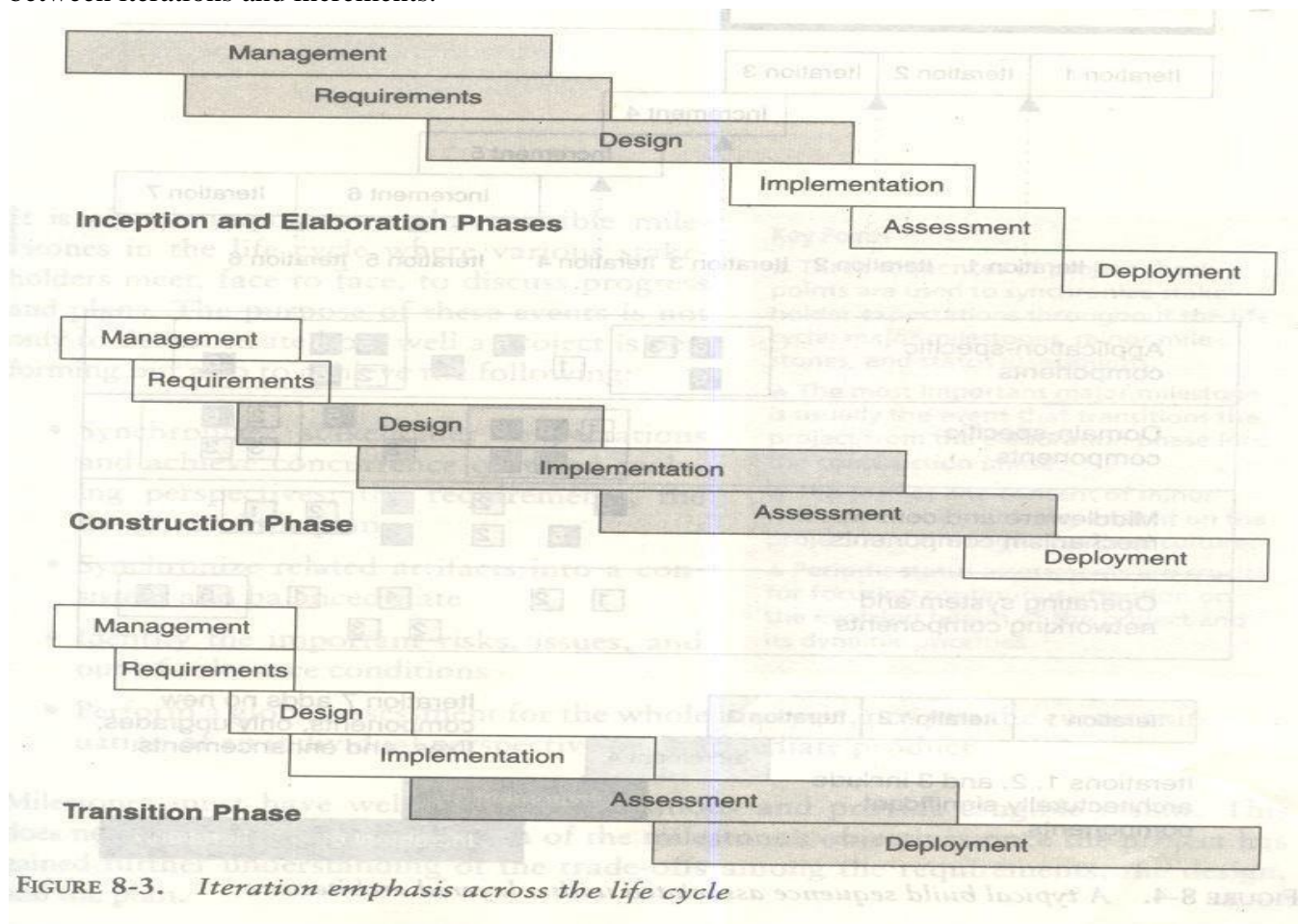


FIGURE 8-2. *The workflow of an iteration*

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities
- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities
- Implementation: developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions)
- Assessment: evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; identifying any rework required and determining whether it should be performed before deployment of this release or allocated to the next release; assessing results to improve the basis of the subsequent iteration's plan

- **Deployment:** transitioning the release either to an external organization (such as a user, independent verification and validation contractor, or regulatory agency) or to internal closure by conducting a post-mortem so that lessons learned can be captured and reflected in the next iteration

Iterations in the inception and elaboration phases focus on management, requirements, and design activities. Iterations in the construction phase focus on design, implementation, and assessment. Iterations in the transition phase focus on assessment and deployment. Figure 8-3 shows the emphasis on different activities across the life cycle. An iteration represents the state of the overall architecture and the complete deliverable system. An increment represents the current progress that will be combined with the preceding iteration to form the next iteration. Figure 8-4, an example of a simple development life cycle, illustrates the differences between iterations and increments.



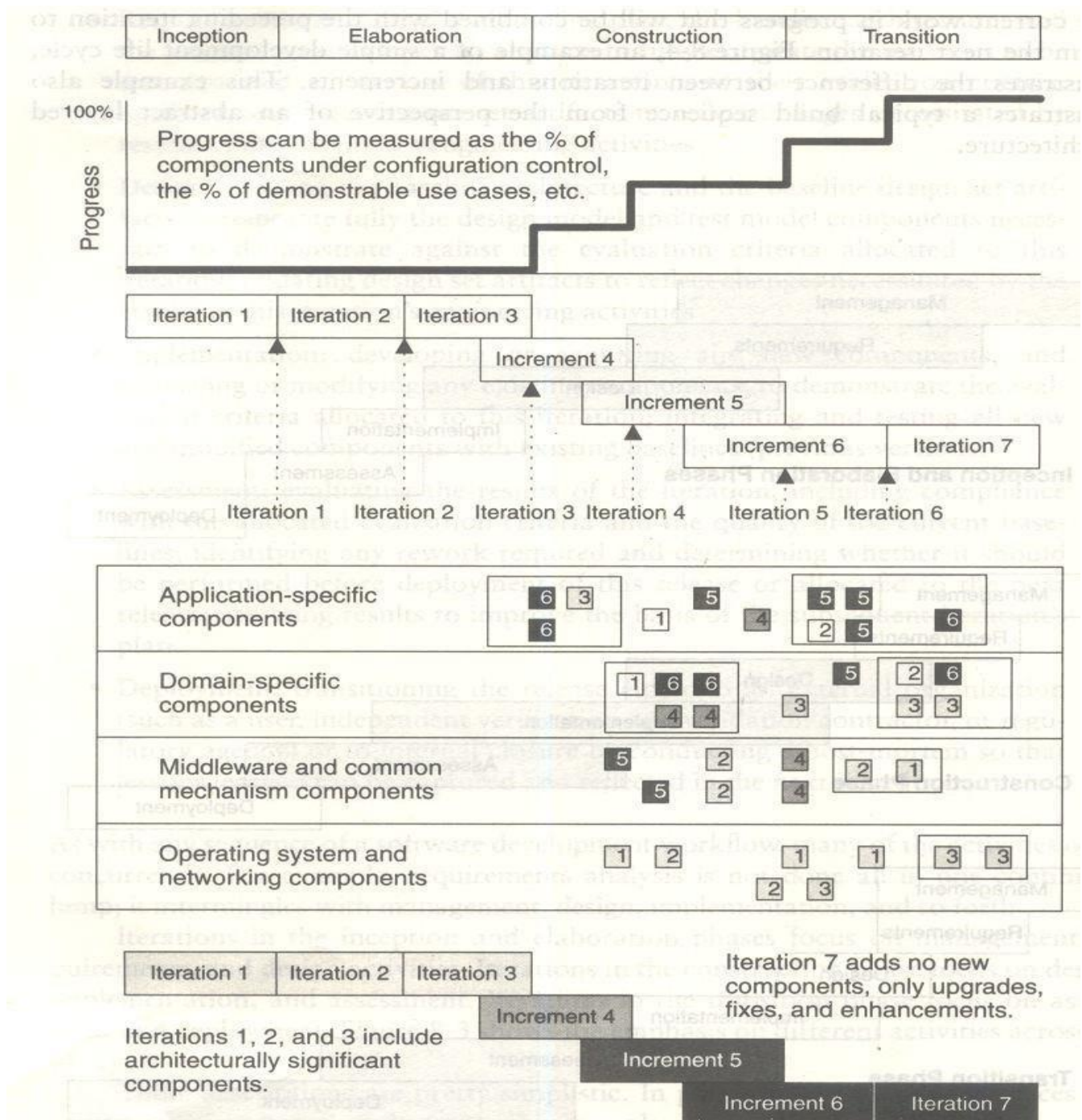


FIGURE 8-4. A typical build sequence associated with a layered architecture

8. Checkpoints of the process

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

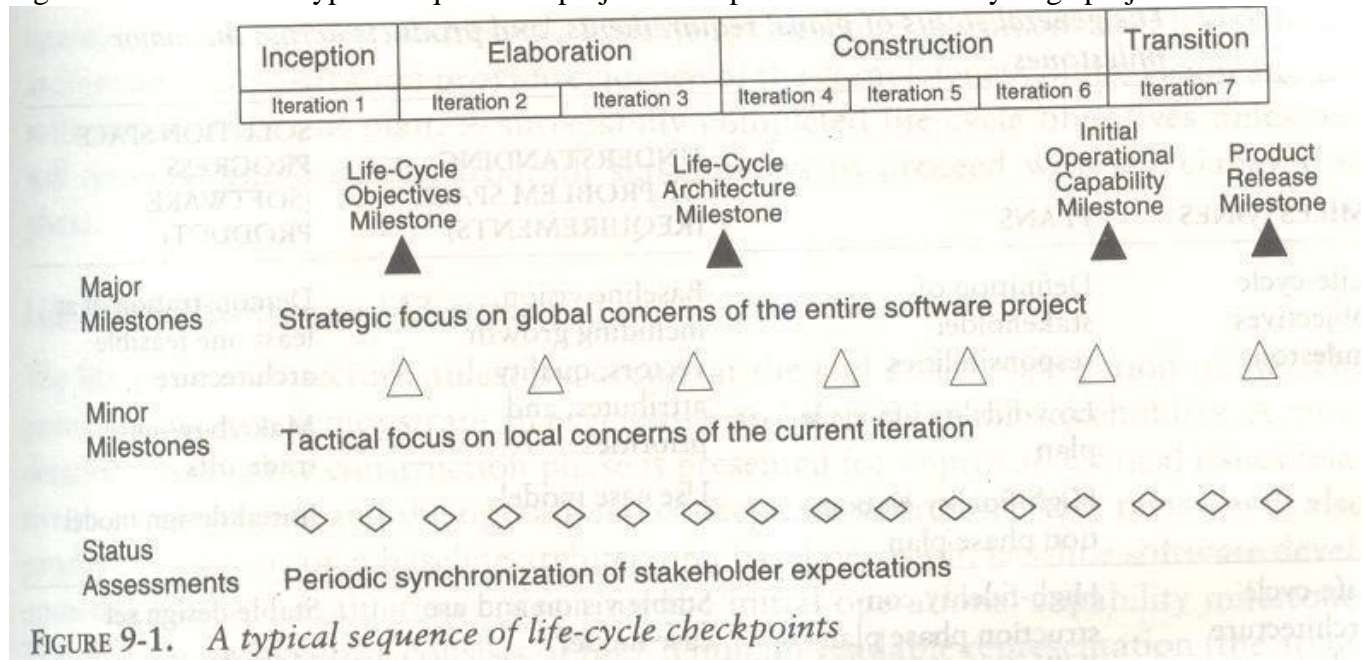
1. *Major milestones.* These system wide events are held at the end of each development phase. They provide visibility to system wide issues, synchronize the management

and engineering perspectives, and verify that the aims of the phase have been achieved.

2. *Minor milestones.* These iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work.
3. *Status assessments.* These periodic events provide management with frequent and regular insight into the progress being made.

Each of the four phases-inception, elaboration, construction, and transition consists of one or more iterations and concludes with a major milestone when a planned technical capability is produced in demonstrable form. An iteration represents a cycle of activities for which there is a well-defined intermediate result-a minor milestone-captured with two artifacts: a release specification (the evaluation criteria and plan) and a release description (the results). Major milestones at the end of each phase use formal, stakeholder-approved evaluation criteria and release descriptions; minor milestones use informal, development-team-controlled versions of these artifacts.

Figure 9-1 illustrates a typical sequence of project checkpoints for a relatively large project.



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

- Architects and systems engineers: product line compatibility, requirements changes, trade-off analyses, completeness and consistency, balance among risk, quality, and usability
- Developers: sufficiency of requirements detail and usage scenario descriptions, . frameworks for component selection or development, resolution of development risk, product line compatibility, sufficiency of the development environment
- Maintainers: sufficiency of product and documentation artifacts, understandability, interoperability with existing systems, sufficiency of maintenance environment
- Others: possibly many other perspectives by stakeholders such as regulatory agencies, independent verification and validation contractors, venture capital investors, subcontractors, associate contractors, and sales and marketing teams Table 9-1 summarizes the balance of information across the major milestones.

TABLE 9-1. *The general status of plans, requirements, and products across the major milestones*

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

Life-Cycle Objectives Milestone

The life-cycle objectives milestone occurs at the end of the inception phase. The goal is to present to all stakeholders a recommendation on how to proceed with development, including a plan, estimated cost and schedule, and expected benefits and cost savings. A successfully completed life-cycle objectives milestone will result in authorization from all stakeholders to proceed with the elaboration phase.

Life-Cycle Architecture Milestone

The life-cycle architecture milestone occurs at the end of the elaboration phase. The primary goal is to demonstrate an executable architecture to all stakeholders. The baseline architecture consists of both a human-readable representation (the architecture document) and a

configuration-controlled set of software components captured in the engineering artifacts. A successfully completed life-cycle architecture milestone will result in authorization from the stakeholders to proceed with the construction phase.

The technical data listed in Figure 9-2 should have been reviewed by the time of the lifecycle architecture milestone. Figure 9-3 provides default agendas for this milestone.

I. Requirements
A. Use case model
B. Vision document (text, use cases)
C. Evaluation criteria for elaboration (text, scenarios)
II. Architecture
A. Design view (object models)
B. Process view (if necessary, run-time layout, executable code structure)
C. Component view (subsystem layout, make/buy/reuse component identification)
D. Deployment view (target run-time layout, target executable code structure)
E. Use case view (test case structure, test result expectation)
1. Draft user manual
III. Source and executable libraries
A. Product components
B. Test components
C. Environment and tool components

FIGURE 9-2. *Engineering artifacts available at the life-cycle architecture milestone*

Presentation Agenda	
I. Scope and objectives	A. Demonstration overview
II. Requirements assessment	A. Project vision and use cases B. Primary scenarios and evaluation criteria
III. Architecture assessment	A. Progress <ul style="list-style-type: none"> 1. Baseline architecture metrics (progress to date and baseline for measuring future architectural stability, scrap, and rework) 2. Development metrics baseline estimate (for assessing future progress) 3. Test metrics baseline estimate (for assessing future progress of the test team) B. Quality <ul style="list-style-type: none"> 1. Architectural features (demonstration capability summary vs. evaluation criteria) 2. Performance (demonstration capability summary vs. evaluation criteria) 3. Exposed architectural risks and resolution plans 4. Affordability and make/buy/reuse trade-offs
IV. Construction phase plan assessment	A. Iteration content and use case allocation B. Next iteration(s) detailed plan and evaluation criteria C. Elaboration phase cost/schedule performance D. Construction phase resource plan and basis of estimate E. Risk assessment
Demonstration Agenda	
I. Evaluation criteria	
II. Architecture subset summary	
III. Demonstration environment summary	
IV. Scripted demonstration scenarios	
V. Evaluation criteria results and follow-up items	

FIGURE 9-3. *Default agendas for the life-cycle architecture milestone*

Initial Operational Capability Milestone

The initial operational capability milestone occurs late in the construction phase. The goals are to assess the readiness of the software to begin the transition into customer/user sites and to authorize the start of acceptance testing. Acceptance testing can be done incrementally across multiple iterations or can be completed entirely during the transition phase is not necessarily the completion of the construction phase.

Product Release Milestone

The product release milestone occurs at the end of the transition phase. The goal is to assess the completion of the software and its transition to the support organization, if any. The results of acceptance testing are reviewed, and all open issues are addressed. Software quality metrics are reviewed to determine whether quality is sufficient for transition to the support organization.

MINOR MILESTONES

For most iterations, which have a one-month to six-month duration, only two minor milestones are needed: the iteration readiness review and the iteration assessment review.

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

The format and content of these minor milestones tend to be highly dependent on the project and the organizational culture. Figure 9-4 identifies the various minor milestones to be considered when a project is being planned.

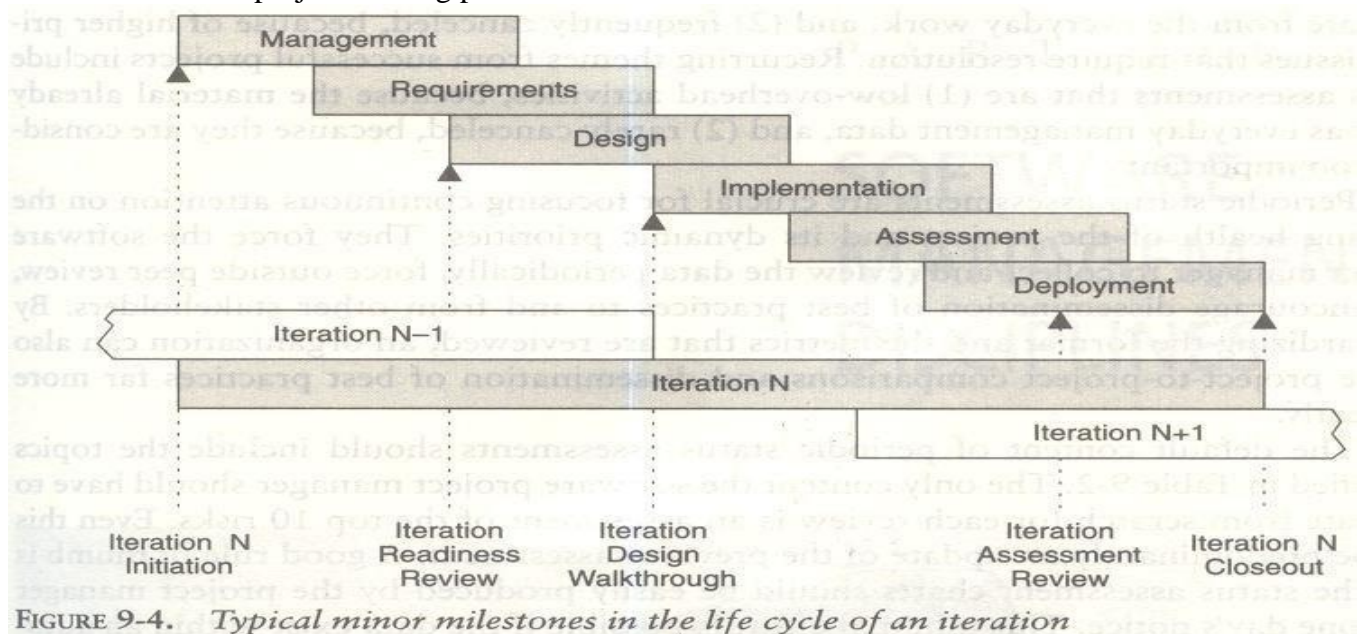


FIGURE 9-4. Typical minor milestones in the life cycle of an iteration

PERIODIC STATUS ASSESSMENTS

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

Periodic status assessments serve as project snapshots. While the period may vary, the recurring event forces the project history to be captured and documented. Status assessments provide the following:

- A mechanism for openly addressing, communicating, and resolving management issues, technical issues, and project risks
- Objective data derived 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

Periodic status assessments are crucial for focusing continuous attention on the evolving health of the project and its dynamic priorities. They force the software project manager to

collect and review the data periodically, force outside peer review, and encourage dissemination of best practices to and from other stakeholders.

The default content of periodic status assessments should include the topics identified in Table 9-2.

TABLE 9-2. *Default content of status assessment reviews*

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

process planning

A good work breakdown structure and its synchronization with the process framework are critical factors in software project success. Development of a work breakdown structure dependent on the project management style, organizational culture, customer preference, financial constraints, and several other hard-to-define, project-specific parameters.

A WBS is simply a hierarchy of elements that decomposes the project plan into the discrete work tasks. A WBS provides the following information structure:

- A delineation of all significant work
- A clear task decomposition for assignment of responsibilities
- A framework for scheduling, budgeting, and expenditure tracking

Many parameters can drive the decomposition of work into discrete tasks: product subsystems, components, functions, organizational units, life-cycle phases, even geographies. Most systems have a first-level decomposition by subsystem. Subsystems are then decomposed into their components, one of which is typically the software.

CONVENTIONAL WBS ISSUES

Conventional work breakdown structures frequently suffer from three fundamental flaws.

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

Conventional work breakdown structures are prematurely structured around the product design. Figure 10-1 shows a typical conventional WBS that has been structured primarily around the subsystems of its product architecture, then further decomposed into the components of each subsystem. A WBS is the architecture for the financial plan.

Conventional work breakdown structures are prematurely decomposed, planned, and budgeted in either too little or too much detail. Large software projects tend to be over planned and small projects tend to be under planned. The basic problem with planning too much detail at the outset is that the detail does not evolve with the level of fidelity in the plan.

Conventional work breakdown structures are project-specific, and cross-project comparisons are usually difficult or impossible. With no standard WBS structure, it is extremely difficult to compare plans, financial data, schedule data, organizational efficiencies, cost trends, productivity trends, or quality trends across multiple projects.

Figure 10-1 Conventional work breakdown structure, following the product hierarchy

Management

System requirement and design

Subsystem 1

Component 11

Requirements

Design

Code

Test

Documentation

...(similar structures for other components)

Component 1N

Requirements

Design

Code

Test

Documen
tation

...(similar structures for other subsystems)

Subsystem M

Component M1**Requirements****Design****Code****Test****Documentation****...(similar structures for other components)****Component MN****Requirements****Design****Code****Test****Documentation****Integration and test****Test planning****Test procedure preparation****Testing****Test reports****Other support areas****Configuration control****Quality assurance****System administration****EVOLUTIONARY WORK BREAKDOWN STRUCTURES**

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

- First-level WBS elements are the workflows (management, environment, requirements, design, implementation, assessment, and deployment).
- Second-level elements are defined for each phase of the life cycle (inception, elaboration, construction, and transition).
- Third-level elements are defined for the focus of activities that produce the artifacts of each phase.

A default WBS consistent with the process framework (phases, workflows, and artifacts) is shown in Figure 10-2. This recommended structure provides one example of how the elements of the process framework can be integrated into a plan. It provides a framework for estimating the costs and schedules of each element, allocating them across a project organization, and tracking expenditures.

The structure shown is intended to be merely a starting point. It needs to be tailored to the specifics of a project in many ways.

- Scale. Larger projects will have more levels and substructures.
- Organizational structure. Projects that include subcontractors or span multiple organizational entities may introduce constraints that necessitate different WBS allocations.
- Degree of custom development. Depending on the character of the project, there can be very different emphases in the requirements, design, and implementation workflows.
- Business context. Projects developing commercial products for delivery to a broad customer base may require much more elaborate substructures for the deployment element.
- Precedent experience. Very few projects start with a clean slate. Most of them are developed as new generations of a legacy system (with a mature WBS) or in the context of existing organizational standards (with preordained WBS expectations).

The WBS decomposes the character of the project and maps it to the life cycle, the budget, and the personnel. Reviewing a WBS provides insight into the important attributes, priorities, and structure of the project plan.

Another important attribute of a good WBS is that the planning fidelity inherent in each element is commensurate with the current life-cycle phase and project state. Figure 10-3 illustrates this idea. One of the primary reasons for organizing the default WBS the way I have is to allow for planning elements that range from planning packages (rough budgets that are maintained as an estimate for future elaboration rather than being decomposed into detail) through fully planned activity networks (with a well-defined budget and continuous assessment of actual versus planned expenditures).

Figure 10-2 Default work breakdown structure A

Management

AA Inception phase management AAA

Business case development

AAB Elaboration phase release specifications AAC

Elaboration phase WBS specifications 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 CCA
Vision specification

CAB Use case modeling

CB Elaboration phase requirements baselining CBA
Vision baselining

CBB Use case model baselining

CC Construction phase requirements maintenance CD
Transition phase requirements maintenance

D Design

DA Inception phase architecture prototyping DB
Elaboration phase architecture baselining DBA
Architecture design modeling

DBB Design demonstration planning and conduct DBC
Software architecture description

DC Construction phase design modeling

DCA Architecture design model maintenance DCB

Component design modeling

**DD Transition phase design maintenance E
Implementation**

EA Inception phase component prototyping

EB Elaboration phase component implementation

EBA Critical component coding demonstration integration

EC Construction phase component implementation

ECA Initial release(s) component coding and stand-alone testing ECB

**Alpha release component coding and stand-alone testing ECC Beta
release component coding and stand-alone testing**

**ECD Component maintenance F
Assessment**

**FA Inception phase assessment FB
Elaboration phase assessment**

FBA Test modeling

FBB Architecture test scenario implementation

**FBC Demonstration assessment and release descriptions FC
Construction phase assessment**

**FCA Initial release assessment and release description FCB
Alpha release assessment and release description FCC Beta
release assessment and release description**

FD Transition phase assessment

**FDA Product release assessment and release description G
Deployment**

GA Inception phase deployment planning GB

Elaboration phase deployment planning GC

Construction phase deployment

**GCA User manual baselining GD
Transition phase deployment**

GDA Product transition to user

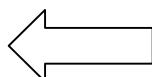
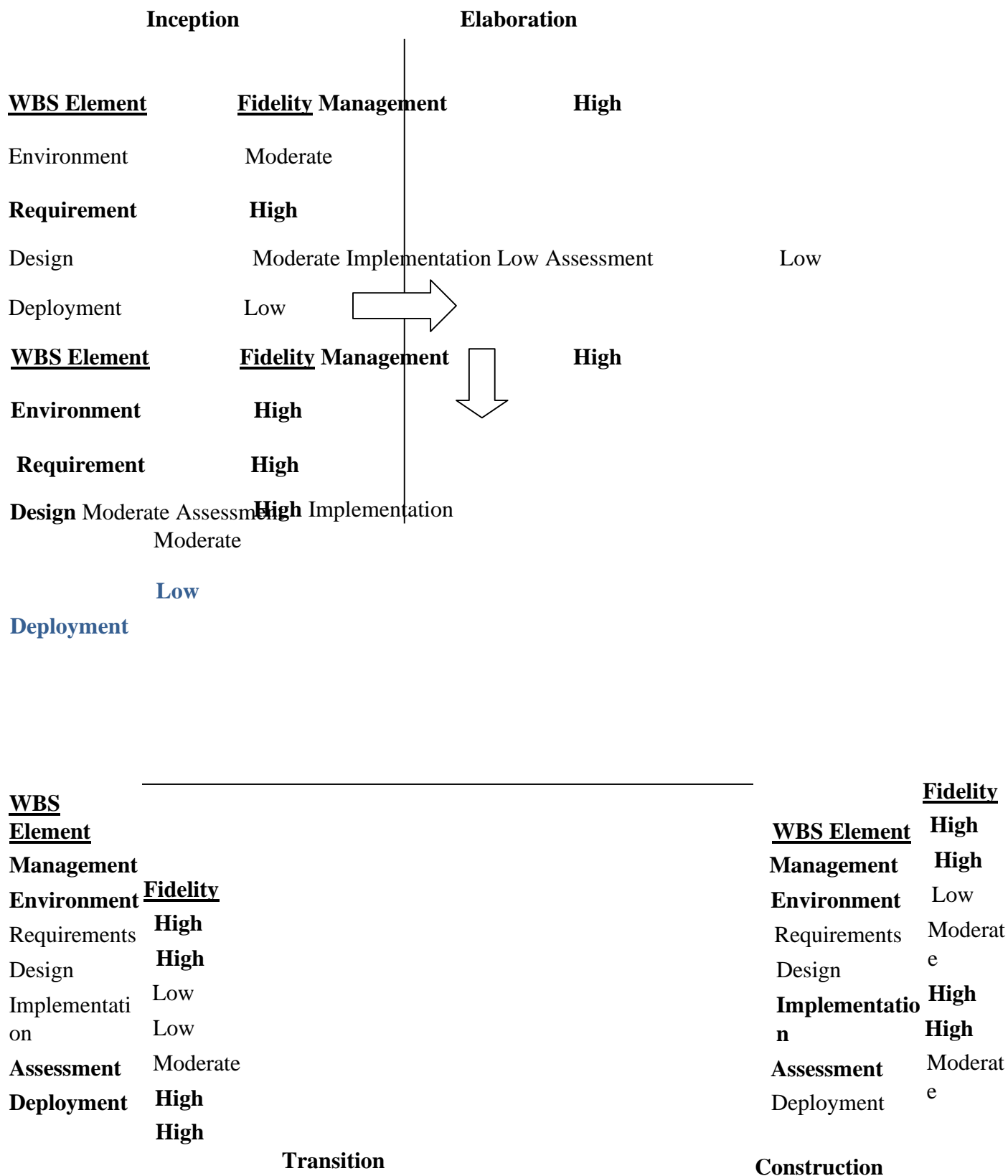


Figure 10-3 Evolution of planning fidelity in the WBS over the life cycle



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 circumstances. 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 10-2, prescribes the allocation of effort and schedule across the lifecycle phases.

10-1 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	Inception	Elaboration	Constructio	Transition
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

THE COST AND SCHEDULE ESTIMATING PROCESS

Project plans need to be derived from two perspectives. The first is a forward-looking, top-down approach. It starts with an understanding of the general requirements and constraints, derives a macro-level budget and schedule, then decomposes these elements into lower level budgets and intermediate milestones. From this perspective, the following planning sequence would occur:

1. The software project manager (and others) develops a characterization of the overall size, process, environment, people, and quality required for the project.
2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.
3. The software project manager partitions the estimate for the effort into a top-level WBS using guidelines such as those in Table 10-1.
4. 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.

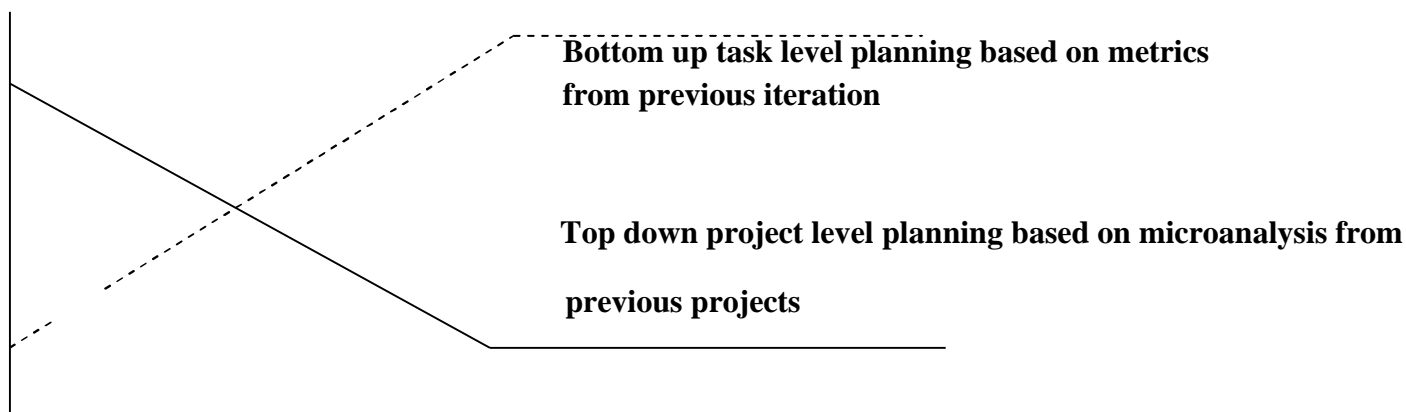
The second perspective is a backward-looking, bottom-up approach. We start with the end in mind, analyze the micro-level budgets and schedules, then sum all these elements into the higher level budgets and intermediate milestones. This approach tends to define and populate the WBS from the lowest levels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

Milestone scheduling or budget allocation through top-down estimating tends to exaggerate the project management biases and usually results in an overly optimistic plan. Bottom-up estimates usually exaggerate the performer biases and result in an overly pessimistic plan.

These two planning approaches should be used together, in balance, throughout the life cycle of the project. During the engineering stage, the top-down perspective will dominate because there is usually not enough depth of understanding nor stability in the detailed task sequences to perform credible bottom-up planning. During the production stage, there should be enough precedent experience and planning fidelity that the bottom-up planning perspective will dominate. Top-down approach should be well tuned to the project-specific parameters, so it should be used more as a global assessment technique. Figure 10-4 illustrates this life-cycle planning balance.

Figure 10-4 Planning balance throughout the life cycle



Engineering Stage		Production Stage	
Inception	Elaboration	Construction	Transition
Feasibility iteration	Architecture iteration	Usable iteration	Product Releases

Engineering stage planning emphasis	Production stage planning emphasis
Macro level task estimation for production stage artifacts	Micro level task estimation for production stage artifacts

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

THE ITERATION PLANNING 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 circumstances. *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 a candidate architecture and provide an executable framework for elaborating the critical use cases of the system. This framework includes existing components, commercial components, and custom prototypes sufficient to demonstrate a candidate architecture and sufficient requirements understanding to establish a credible business case, vision, and software development plan.
- Elaboration iterations. 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 load 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 iterations. 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

A very large or unprecedented project with many stakeholders may require additional inception iteration and two additional iterations in construction, for a total of nine iterations.

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 from bad architectures and misunderstood requirements, 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 stakeholders, and it must be used. Plans are not just for managers. 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 IV Project Organizations

Project Organizations and Responsibilities:

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

1) Line-Of-Business Organizations:

The main features of default organization are as follows:

- Responsibility for process definition & maintenance is specific to a cohesive line of business.
- Responsibility for process automation is an organizational role & is equal in importance to the process definition role.
- Organizational role may be fulfilled by a single individual or several different teams.

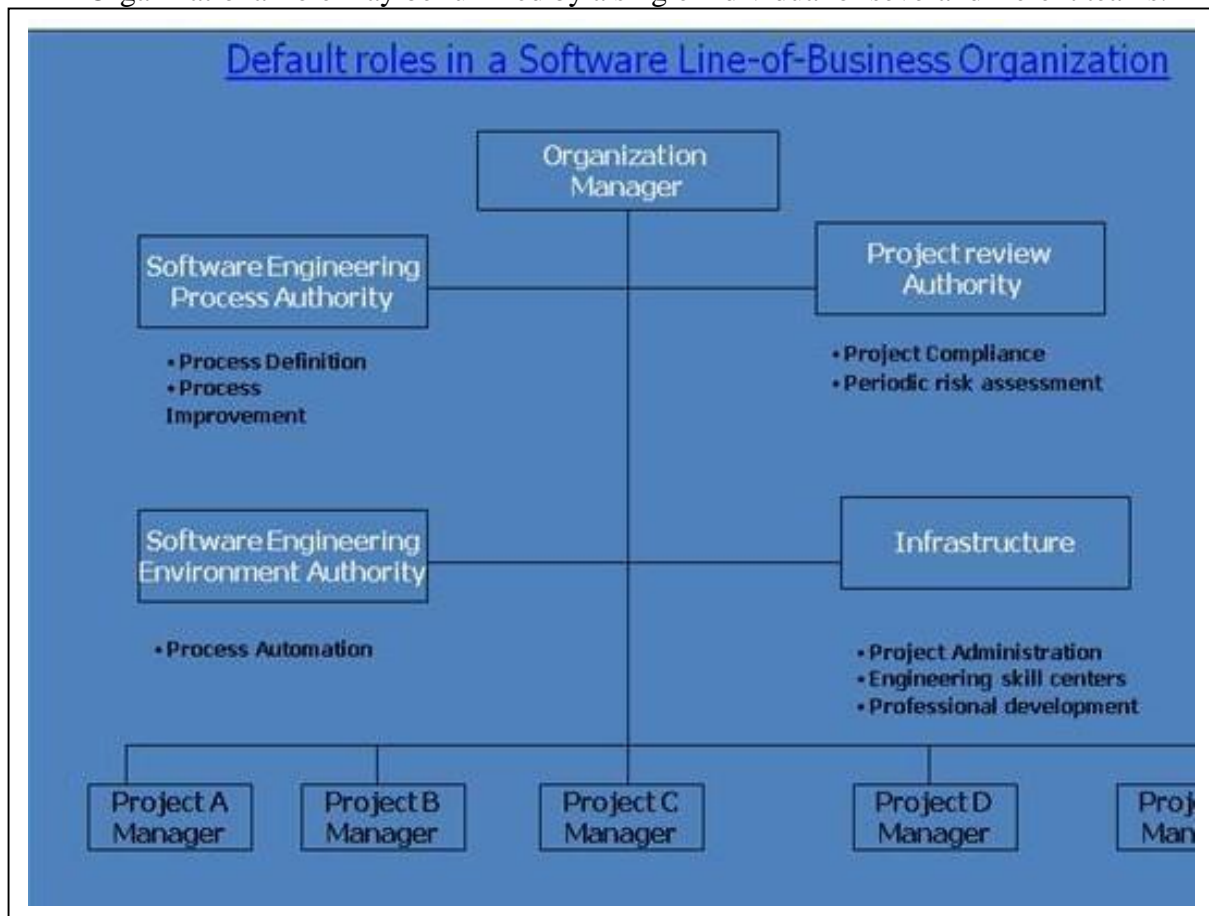


Fig: Default roles in a software Line-of-Business Organization.

Software Engineering Process Authority (SEPA)

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

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

Project Review Authority (PRA)

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

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

Software Engineering Environment Authority(SEEA)

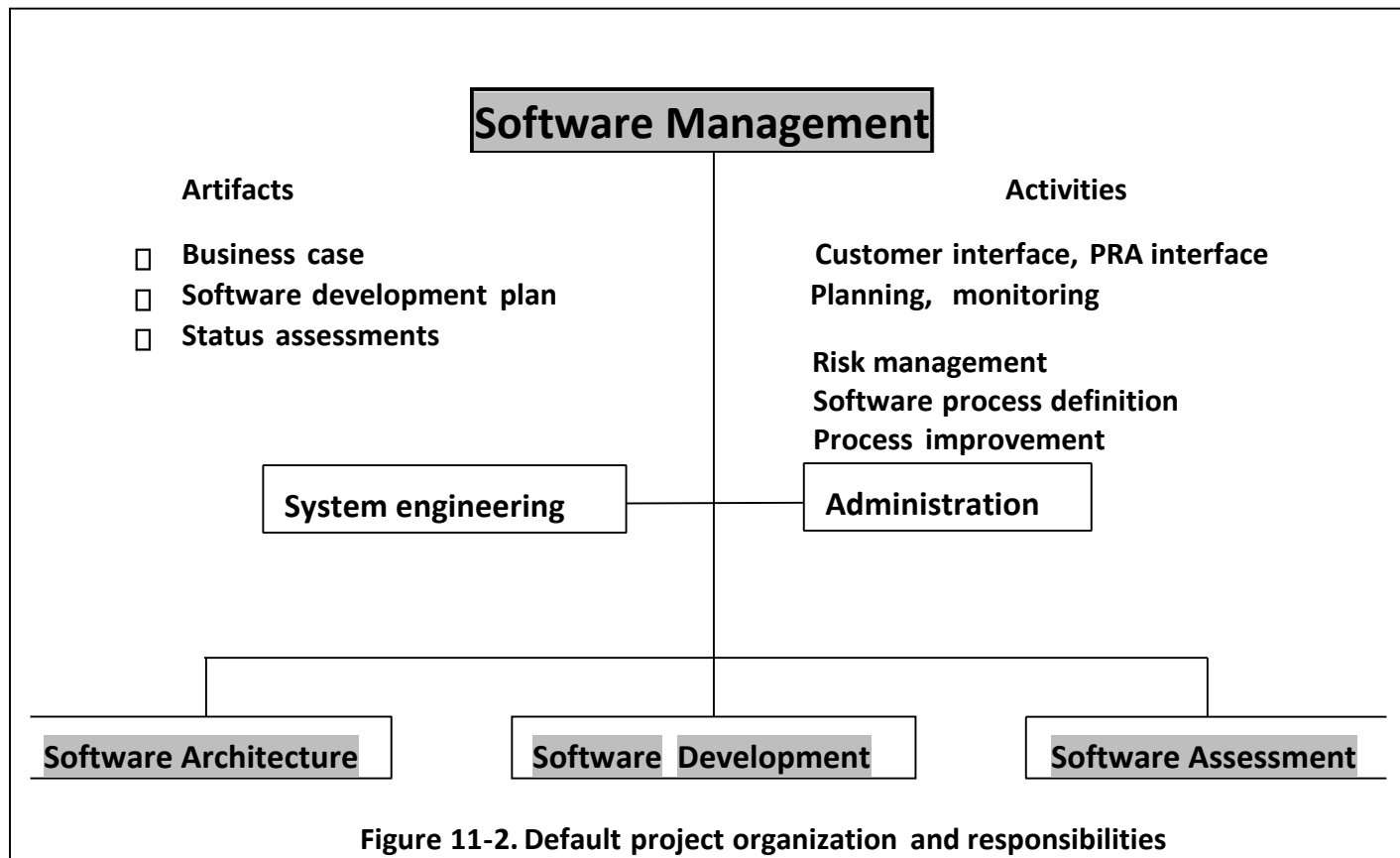
The SEEA is responsible for automating the organization's process, maintaining the organization's standard environment, Training projects to use the environment & maintaining organization-wide reusable assets

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

Infrastructure

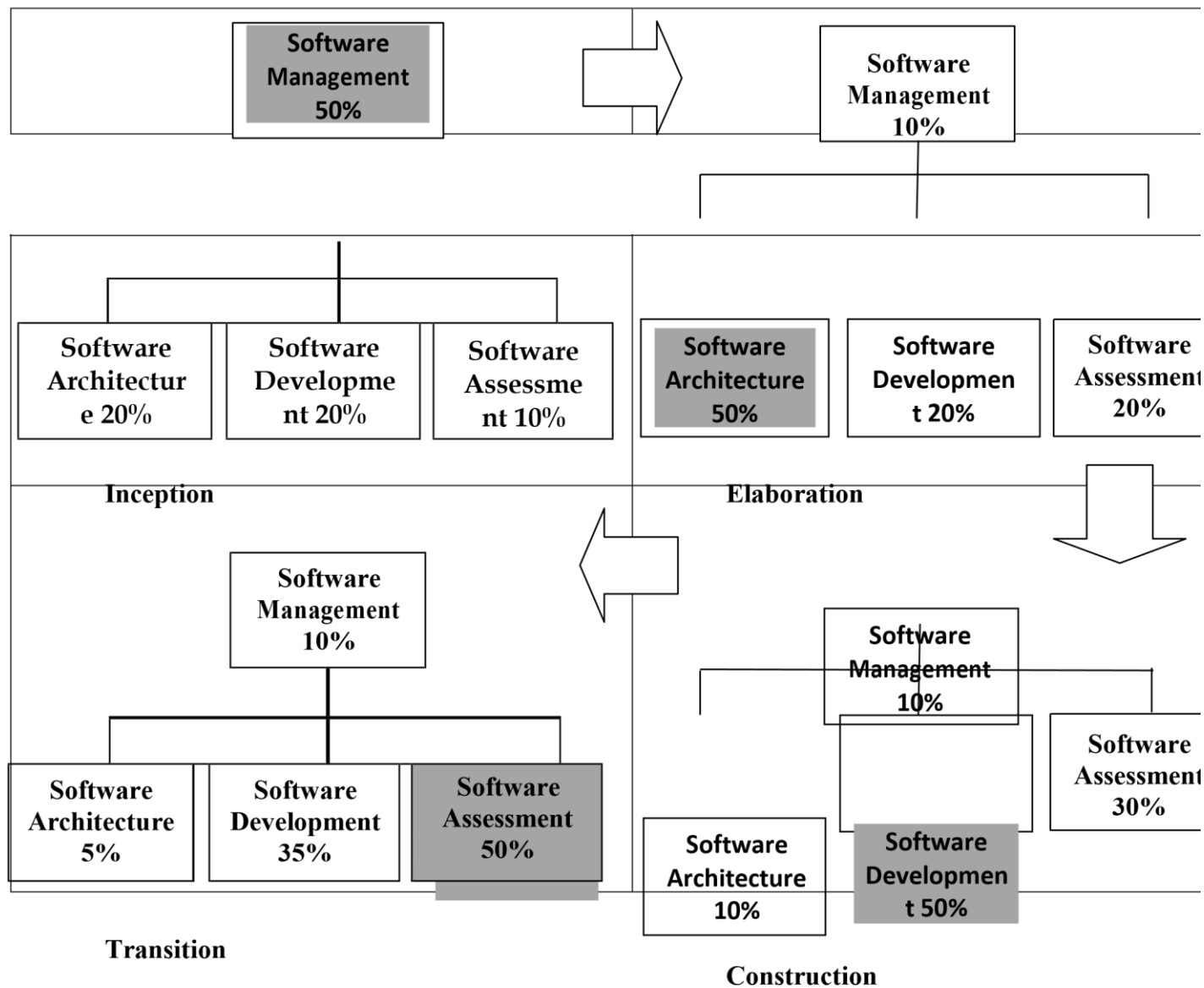
An organization's infrastructure provides human resources support, project-independent research & development, & other capital software engineering assets.

2) Project organizations:



- The above figure shows a default project organization and maps project-level roles and responsibilities.
- The main features of the default organization are as follows:
- **The project management team** is an active participant, responsible for producing as well as managing.
- **The architecture team** is responsible for real artifacts and for the integration of components, not just for staff functions.
- **The development team** owns the component construction and maintenance activities.
- The assessment team is separate from development.
- **Quality** is everyone's into all activities and checkpoints.
- Each team takes responsibility for a different quality perspective.

3) EVOLUTION OF ORGANIZATIONS:



Inception: Software management: 50% Software Architecture: 20% Software development: 20% Software Assessment (measurement/evaluation):10%	Elaboration: Software management: 10% Software Architecture: 50% Software development: 20% Software Assessment (measurement/evaluation):20%
Construction: Software management: 10% Software Architecture: 10% Software development: 50% Software Assessment (measurement/evaluation):30%	Transition: Software management: 10% Software Architecture: 5% Software development: 35% Software Assessment (measurement/evaluation): 50%

The Process Automation:

Introductory Remarks:

The environment must be the first-class artifact of the process.

Process automation & change management is critical to an iterative process. If the change is expensive then the development organization will resist it.

Round-trip engineering & integrated environments promote change freedom & effective evolution of technical artifacts.

Metric automation is crucial to effective project control.

External stakeholders need access to environment resources to improve interaction with the development team & add value to the process.

The three levels of process which requires a certain degree of process automation for the corresponding process to be carried out efficiently.

Metaprocess (Line of business): The automation support for this level is called an infrastructure.

Macroprocess (project): The automation support for a project's process is called an environment.

Microprocess (iteration): The automation support for generating artifacts is generally called a tool.

Tools: Automation Building blocks:

Many tools are available to automate the software development process. Most of the core software development tools map closely to one of the process workflows

Workflows

Environment Tools & process Automation

Management

Workflow automation, Metrics automation

Environment

Change Management, Document Automation

Requirements

Requirement Management

Design

Visual Modeling

Implementation
Assessment

-Editors, Compilers, Debugger, Linker, Runtime
-Test automation, defect Tracking

Deployment

defect Tracking

PROCESS AUTOMATION

The Project Environment:

The project environment artifacts evolve through three discrete states.

(1)Prototyping Environment.(2)Development Environment.(3)Maintenance Environment.

The **Prototype Environment** includes an architecture test bed for prototyping project architecture to evaluate trade-offs during inception & elaboration phase of the life cycle.

The **Development environment** should include a full suite of development tools needed to support various Process workflows & round-trip engineering to the maximum extent possible.

The Maintenance Environment should typically coincide with the mature version of the development.

There are four important environment disciplines that are critical to management context & the success of a modern iterative development process.

Round-Trip engineering

Change Management

Software Change Orders (SCO)

Configuration baseline Configuration Control Board **Infrastructure**

Organization Policy

Organization Environment

Stakeholder Environment.

Round Trip Environment

Tools must be integrated to maintain consistency & traceability.

Round-Trip engineering is the term used to describe this key requirement for environment that support iterative development.

As the software industry moves into maintaining different information sets for the engineering artifacts, more automation support is needed to ensure efficient & error free transition of data from one artifacts to another.

Round-trip engineering is the environment support necessary to maintain Consistency among the engineering artifacts.

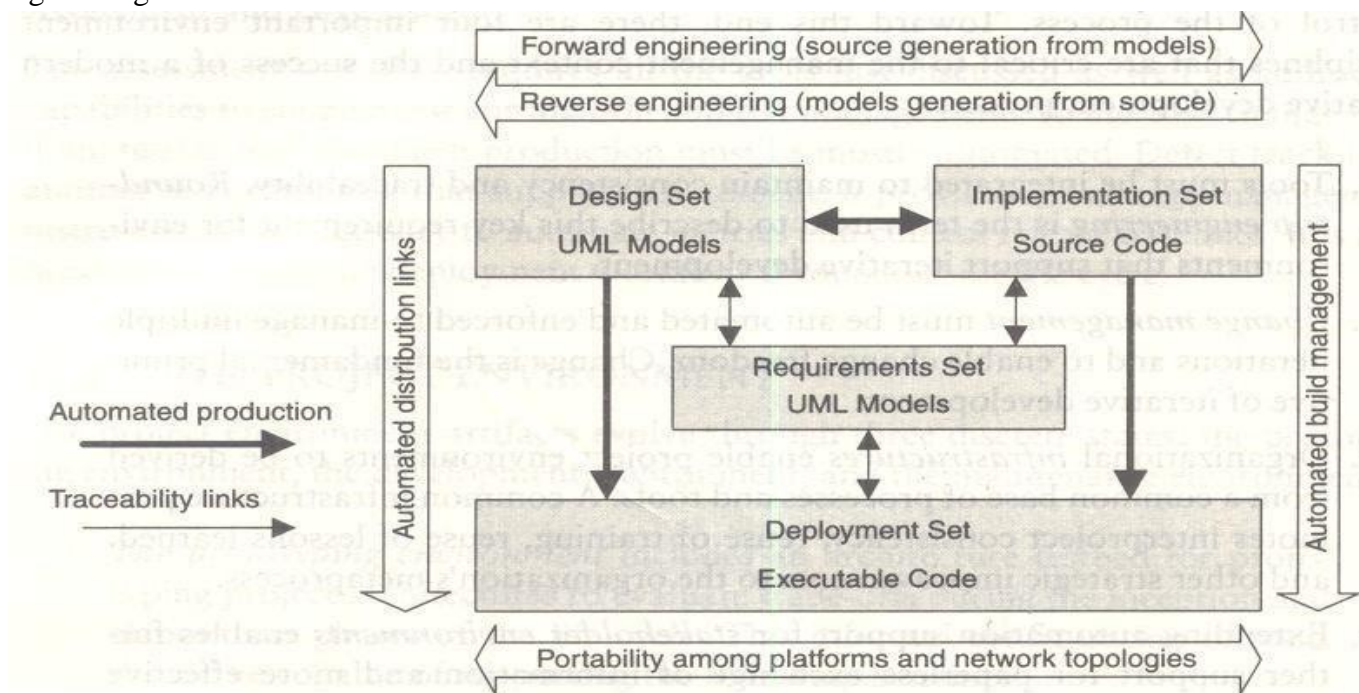


FIGURE 12-2. Round-trip engineering

Change Management

Change management must be automated & enforced to manage multiple iterations & to enable change freedom.

Change is the fundamental primitive of iterative Development.

I. Software Change Orders

The atomic unit of software work that is authorized to create, modify or obsolesce components within a configuration baseline is called a software change orders (SCO)
The basic fields of the SCO are Title, description, metrics, resolution, assessment & disposition

Title: _____		
Description	Name: _____ Project: _____	Date: _____
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*

Change management

II.Configuration Baseline

A configuration baseline is a named collection of software components & Supporting documentation that is subjected to change management & is upgraded, maintained, tested, statuses & obsolesced a unit

There are generally two classes of baselines

External Product Release

Internal testing Release

Three levels of baseline releases are required for most Systems

1. Major release (N)
2. Minor Release (M)
3. Interim (temporary) Release (X)

Major release represents a new generation of the product or project

A **minor** release represents the same basic product but with enhanced features, performance or quality.

Major & Minor releases are intended to be external product releases that are persistent & supported for a period of time.

An interim release corresponds to a developmental configuration that is intended to be transient.

Once software is placed in a controlled baseline all changes are tracked such that a distinction must be made for the cause of the change. Change categories are

Type 0: Critical Failures (must be fixed before release)

Type 1: A bug or defect either does not impair (Harm) the usefulness of the system or can be worked around

Type 2: A change that is an enhancement rather than a response to a defect **Type**

3: A change that is necessitated by the update to the environment **Type 4:**

Changes that are not accommodated by the other categories.

Change Management

III Configuration Control Board (CCB)

A CCB is a team of people that functions as the decision Authority on the content of configuration baselines

A CCB includes:

1. **Software managers**
2. **Software Architecture managers**
3. **Software Development managers**
4. **Software Assessment managers**
5. **Other Stakeholders who are integral to the maintenance of the controlled software delivery system?**

Infrastructure

The organization infrastructure provides the organization's capital assets including two key artifacts - Policy & Environment **I Organization Policy:**

A Policy captures the standards for project software development processes

The organization policy is usually packaged as a handbook that defines the life cycles & the

process primitives such as

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

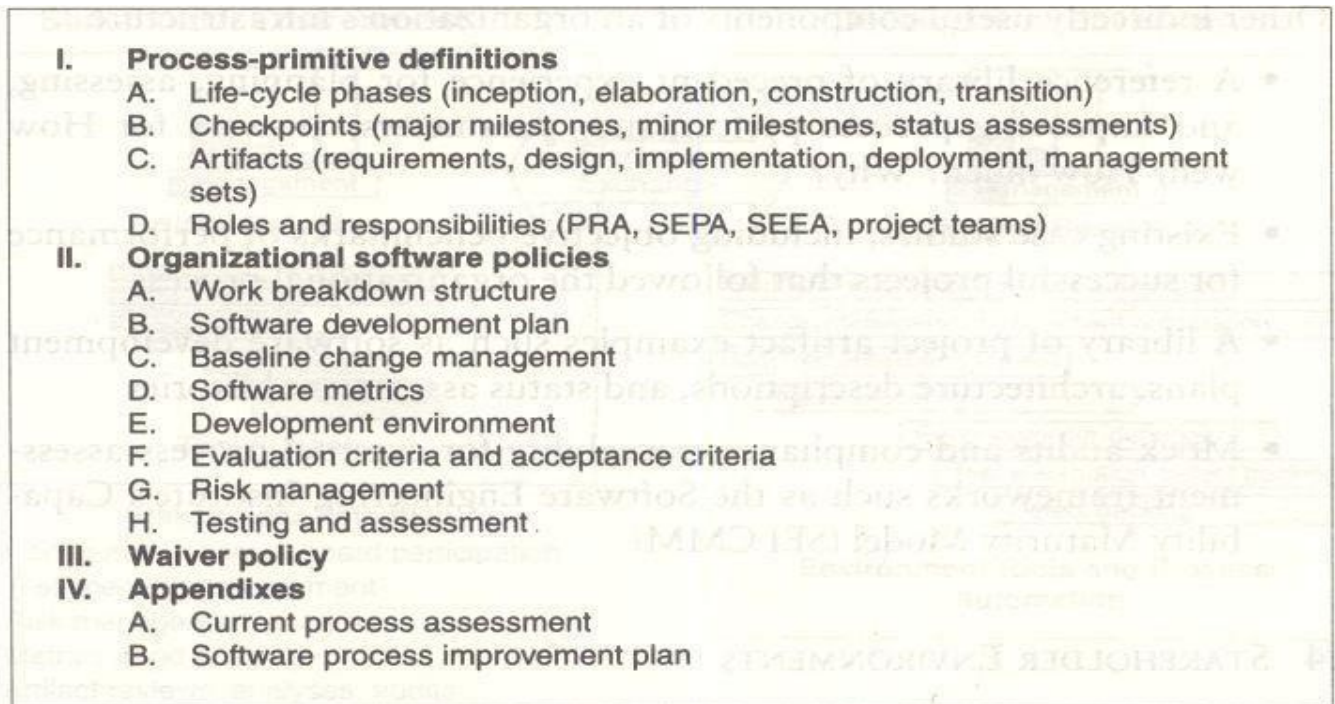


FIGURE 12-5. *Organization policy outline*

Infrastructure

II Organization Environment

The Environment that captures an inventory of tools which are building blocks from which project environments can be configured efficiently & economically

Stakeholder Environment

Many large scale projects include people in external organizations that represent other

stakeholders participating in the development process they might include

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

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

An on-line environment accessible by the external stakeholders allow them to participate in the process a follows

Accept & use executable increments for the hands-on evaluation.

Use the same on-line tools, data & reports that the development organization uses to manage & monitor the project

Avoid excessive travel, paper interchange delays, format translations, paper * shipping costs & other overhead cost

PROJECT CONTROL & PROCESS INSTRUMENTATION

INTERODUCTION: 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.

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:

- (i) Management indicators.
- (ii) Quality indicators.

Management Indicators

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

Quality Indicators

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

SEVEN CORE METRICS OF SOFTWARE PROJECT

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

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

Seven core metrics related to project control:

Management Indicators

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

Quality Indicators

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

MANAGEMENT INDICATORS:

Work and progress

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

The default perspectives of this metric are:

Software architecture team: - Use cases demonstrated.

Software development team: - SLOC under baseline change management, SCOs closed **Software**

assessment team: - SCOs opened, test hours executed and evaluation criteria meet. **Software**

management team: - milestones completed.

The below figure shows expected progress for a typical project with three major releases

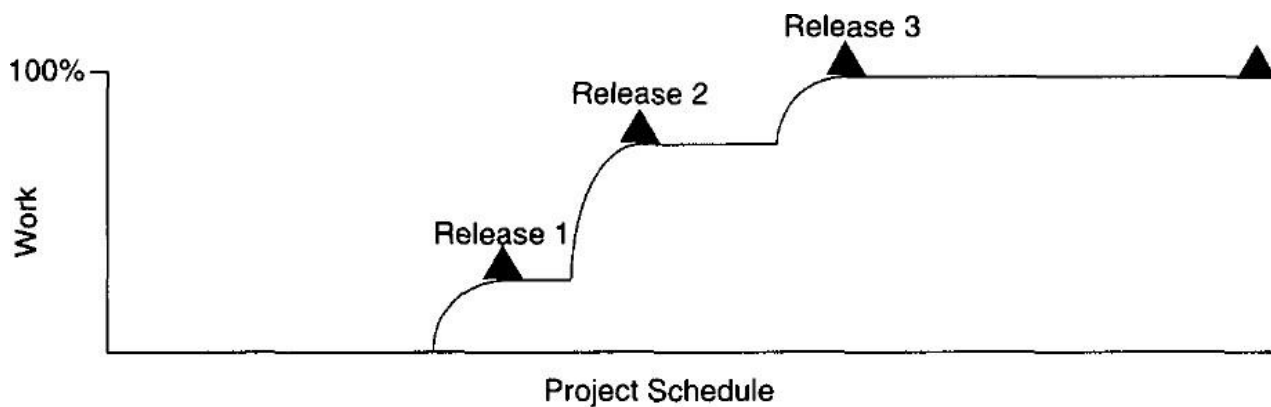


Fig: work and progress

Budgeted cost and expenditures

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

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

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

It is the value that represents the planned cost of the actual progress. **Cost variance:**

It is the difference between the actual cost and the earned value.

Schedule variance: It is the difference between the planned cost and the earned value. Of all parameters in an earned value system, actual progress is the most subjective

Assessment: Because most managers know exactly how much cost they have incurred and how much schedule they have used, the variability in making accurate assessments is centered in the actual progress assessment. The default perspectives of this metric are cost per month, full-time staff per month and percentage of budget expended.

Staffing and team dynamics

This metric measure the personnel changes over time, which involves staffing additions and reductions over time. An iterative development should start with a small team until the risks in the requirements and architecture have been suitably resolved. Depending on the overlap of iterations and other project specific circumstances, staffing can vary. Increase in staff can slow overall project progress as new people consume

the productive team of existing people in coming up to speed. Low attrition of good people is a sign of success. The default perspectives of this metric are people per month added and people per month leaving.

These three management indicators are responsible for technical progress, financial status and staffing progress.

. Fig: work and progress

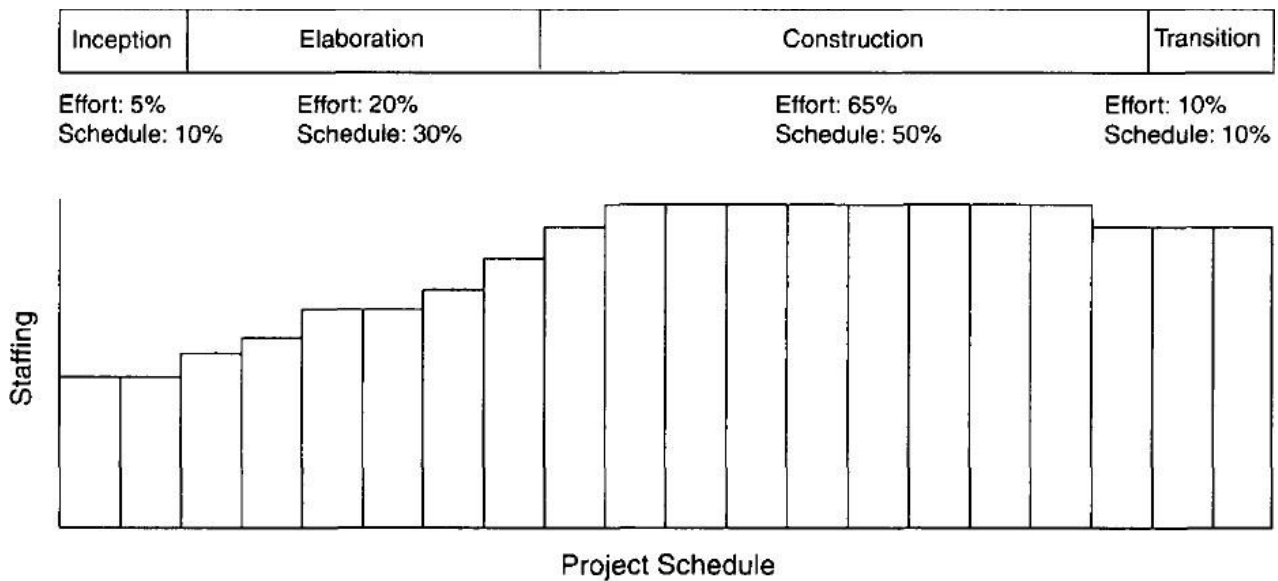


Fig: staffing and Team dynamics

QUALITY INDICATORS:

Change traffic and stability:

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

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

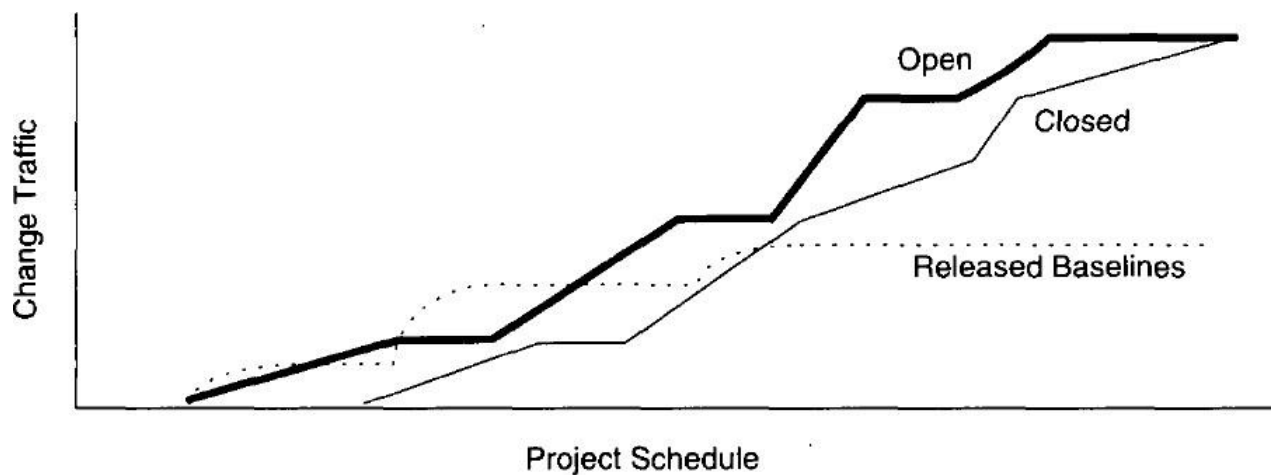


Fig: Change traffic and stability

Breakage and modularity

This metric measures the average breakage per change over time. Breakage is defined as the average extent of change, which is the amount of software baseline that needs rework and measured in source lines of code, function points, components, subsystems, files or other units.

Modularity is the average breakage trend over time. This metric can be collected by rework SLOC per change, by change type, by release, by components and by subsystems.

Rework and adaptability:

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

MTBF and Maturity:

This metric measures defect rate over time. MTBF (Mean Time Between Failures) is the average usage time between software faults. It is computed by dividing the test hours by the number of type 0 and type 1 SCOs. Maturity is defined as the MTBF trend over time.

Software errors can be categorized into two types deterministic and nondeterministic. Deterministic errors are also known as Bohr-bugs and nondeterministic errors are also called as Heisen-bugs. Bohr-bugs are a class of errors caused when the software is stimulated in a certain way such as coding errors. Heisenbugs 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:

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.

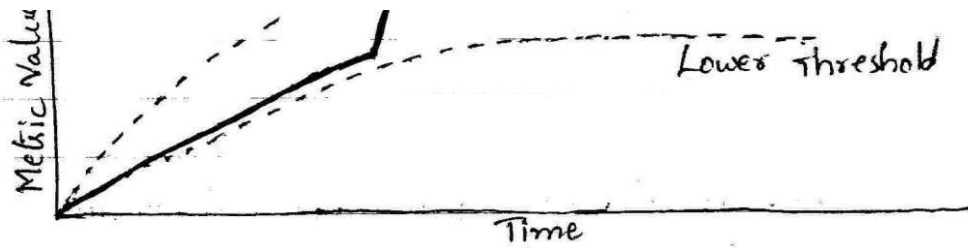
- Metrics primitives - trends, comparisons and progressions ➤ A graphical user interface.
- Metrics collection agents
- Metrics data management server
- Metrics definitions - actual metrics presentations for requirements progress, implementation progress, assessment progress, design progress and other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

Monitor defines panel layouts, graphical objects and linkages to project data. Specific monitors called roles include software project managers, software development team leads, software architects and customers. Administrator installs the system, defines new mechanisms, graphical objects and linkages. The whole display is called a panel. Within a panel are graphical objects, which are types of layouts such as dials and bar charts for information. Each graphical object displays a metric. A panel contains a number of graphical objects positioned in a particular geometric layout. A metric shown in a graphical object is labeled with the metric type, summary level and insurance name (line of code, subsystem, server1). Metrics can be displayed in two modes

– value, referring to a given point in time and graph referring to multiple and consecutive points in time.

Metrics can be displayed with or without control values. A control value is an existing expectation either absolute or relative that is used for comparison with a dynamically changing metric. Thresholds are examples of control values.

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.

UNIT V

CCPDS-R Case Study and Future Software Project Management Practices

This appendix presents a detailed case study of a successful software project that followed many of the techniques presented in this book. Successful here means on budget, on schedule, and satisfactory to the customer. The Command Center Processing and Display System 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.

The software effort included the development of three distinct software systems totaling more than one million source lines of code. This case study focuses on the initial software development, called the Common Subsystem, for which about 355,000 source lines were developed. The Common Subsystem effort also produced a reusable architecture, a mature process, and an integrated environment for efficient development of the two software subsystems of roughly similar size that followed. This case study therefore represents about one-sixth of the overall CCPDS-R project effort.

Although this case study does not coincide exactly with the management process presented in this book nor with all of today's modern technologies, it used most of the same techniques and was managed to the same spirit and priorities. TRW delivered

Key Points a An objective case study is a true indicator of a mature organization and a mature project process. The software industry needs more case studies like CGPDS-R.

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.

a CCPDS-R was one of the pioneering projects that practiced many modern management approaches. a This appendix provides a practical context that is relevant to the techniques, disciplines, and opinions provided throughout this book.

the system on budget and on schedule, and the users got more than they expected. TRW was awarded the Space and Missile Warning Systems Award for Excellence in 1991 for "continued, sustained performance in overall systems engineering and [project execution](#)." A project like CCPDS-R could be developed far more efficiently today. By incorporating current technologies and improved processes, environments, and levels of automation, this project could probably be built today with equal quality in half the time and at a quarter of the cost.

Some of today's popular software cost models are not well matched to an iterative software process focused on an architecture-first approach. Many cost estimators are still using a conventional process experience base to estimate a modern project profile. A next-generation software cost model should explicitly separate architectural engineering from application production, just as an architecture-first process does. Two major improvements in next-generation software cost estimation models: Separation of the engineering stage from the production stage will force estimators to differentiate between architectural scale and implementation size. Rigorous design notations such as UML will offer an opportunity to define units of measure for scale that are more standardized and therefore can be automated and tracked. Modern Software Economics: Changes that provide a good description of what an organizational manager should strive for in making the transition to a modern process: 1. Finding and fixing a software problem after delivery costs 100 times more than 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. 9. Walkthroughs catch 60% of the errors.

10. 80% of the contribution comes from 20% of the contributors.

Next-Generation Software Economics

Next-generation [software economics](#) is being practiced by some advanced software organizations. Many of the techniques, processes, and methods described in this book's process framework have been practiced for several years. However, a mature, modern process is nowhere near the state of the practice for the average software organization. introduces several provocative hypotheses about the future of software economics. A general structure is proposed for a cost estimation model that would be better suited to the process framework

new approach would improve the accuracy and precision of software cost estimates, and would accommodate dramatic improvements in software economies of scale. Such improvements will be enabled by advances in software development environments. Boehm's benchmarks of [conventional software](#) project performance and describe, in objective terms, how the process framework should improve the overall software economics achieved by a project or organization.

Key Points

- ▲ Next-generation software economics should reflect better economies of scale and improved return on investment profiles. These are the real indicators of a mature industry.
- ▲ Further technology advances in round-trip engineering are critical to making the next quantum leap in software economics.
- ▲ Future cost estimation models need to be based on better primitive units defined from well-understood software engineering notations such as the Unified Modeling Language.

Modern Process Transitions

Successful software management is hard work. Technical breakthroughs, process breakthroughs, and new tools will make it easier, but management discipline will continue to be the crux of software project success. New technological advances will be accompanied by new opportunities for software applications, new dimensions of complexity, new avenues of automation, and new customers with different priorities. Accommodating these changes will perturb many of our ingrained software management values and priorities. However, striking a balance among requirements, designs, and plans will remain the underlying objective of future software management endeavors, just as it is today.

numerous projects have been practicing some of these disciplines for years. However, many of the techniques and disciplines suggested herein will necessitate a significant paradigm shift. Some of these changes will be resisted by certain stakeholders or by certain factions within a project or organization. It is not always easy to separate cultural resistance from objective resistance. summarizes some of the important culture shifts to be prepared for in order to avoid as many sources of friction as possible in transitioning successfully to a modern process.

Key Points

- ▲ The transition to modern software

1 processes and technologies necessitates i several culture shifts that will not ; always be easy to achieve.

▲ Lessons learned in transitioning organizations to a modern process have exposed several recurring themes of success that represent important culture (shifts from conventional practice.

▲ A significant transition should be attempted on a significant project. Pilot i projects do not generally attract top tal-; ent, and top talent is crucial to the success of any significant transition.