DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Notes

FOR B.TECH IV YEAR - II SEM(R17)

(2019-20)

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100,

Telangana State, INDIA.

Sno	Subject Code	Subject Name
1	R15A0543	Software Project Management
2	R15A0539	WEB SERVICES

Software Project Management
[R15A0543]
LECTURE NOTES

B.TECH IV YEAR – II SEM(R15)
(2018-19)

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

IV Year B. Tech. CSE – II Sem **L T/P/D C**
4 -/-/- 4

(R15A0543) SOFTWARE PROJECT MANAGEMENT

(Core Elective-VI)

Objectives:

- Understanding the specific roles within a software organization as related to project and process management
- Understanding the basic infrastructure competences (e.g., process modeling and measurement)
- Understanding the basic steps of project planning, project management, quality assurance, and process management and their relationships

UNIT-I

Conventional Software Management: The waterfall Model, Conventional Software Management Performance,

Evolution of Software Economics: software Economics. Pragmatic Software Cost Estimation. **Improving Software Economics**: Reducing Software Product Size, Improving Software Processes, Improving Team Effectiveness, Improving Automation, Achieving Required Quality, Peer Inspections.

UNIT-II

Conventional and Modern Software Management: Principles of Conventional Software Engineering, Principles of Modern Software Management, Transitioning to an interactive Process.

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

UNIT-III

Artifacts of the Process: The Artifact Sets. Management Artifacts, Engineering Artifacts, Programmatic Artifacts.

Model Based Software Architectures: A Management Perspective and Technical Perspective.

UNIT-IV

Flows of the Process: Software Process Workflows. Inter Trans Workflows.

Checkpoints of the Process: Major Mile Stones, Minor Milestones, Periodic Status Assessments.

Interactive Process Planning: Work Breakdown Structures, Planning Guidelines, Cost and Schedule Estimating. Interaction Planning Process, Pragmatic Planning.

UNIT-V

Project Organizations and Responsibilities: Line-of-Business Organizations, Project Organizations, and Evolution of Organizations.

Process Automation: Building Blocks, the Project Environment.

Project Control and Process Instrumentation: Server Care Metrics, Management Indicators, Quality Indicators, Life Cycle Expectations Pragmatic Software

TEXT BOOKS:

1. Walker Rayce, “Software Project Management”, 1998, PEA.
2. Henrey, “Software Project Management”, Pearson.

Reference Books:

1. Richard H.Thayer.” Software Engineering Project Management”, 1997, IEEE Computer Society.
2. Shere K.D.: “Software Engineering and Management”, 1998, PHI.
3. S.A. Kelkar, “Software Project Management: A Concise Study”, PHI.
4. Hughes Cotterell, “Software Project Management”, 2e, TMH. 88 5. Kaeron Conway, “Software Project Management from Concept to D

Outcomes:

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

- 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

INDEX

UNIT – I

Conventional Software Management: The waterfall model, conventional software Management performance.

Evolution of Software Economics: Software Economics, pragmatic software cost estimation.

Improving Software Economics: Reducing Software product size, improving software processes, improving team effectiveness, improving automation, Achieving required quality, peer inspections.

---

1. Conventional software management

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

Conventional software economics provides a benchmark of performance for conventional software management principles.

The **best** thing about software is its **flexibility**: It can be programmed to do almost anything.

The **worst** thing about software is also its **flexibility**: The "almost anything" characteristic has made it difficult to plan, monitors, and control software development.

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

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

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

1.1 THE WATERFALL MODEL

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

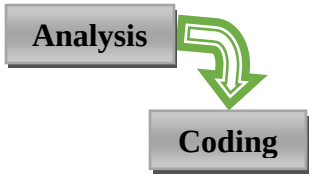
1.1.1 IN THEORY

It provides an insightful and concise summary of conventional software management

Three main primary points are

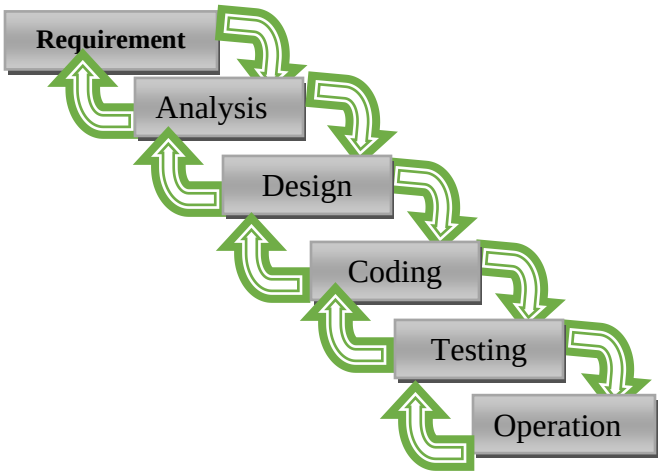
1. There are two essential steps common to the development of computer programs: **analysis** and **coding.**

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



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

2. In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other "overhead" steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps. Below Figure illustrates the resulting project profile and the basic steps in developing a large-scale program.



3. The basic framework described in the waterfall model is risky and invites failure. The testing phase that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the requirements must be modified or a substantial design change is warranted.

Five necessary improvements for waterfall model are:-

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

   The following steps are required:
   Begin the design process with program **designers**, not analysts or programmers.
   Design, define, and allocate the data processing modes even at the risk of being wrong. Allocate processing functions, design the database, allocate execution time, define interfaces and processing modes with the operating system, describe input and output processing, and define preliminary operating procedures.
   Write an overview document that is understandable, informative, and current so that every worker on the project can gain an elemental understanding of the system.

2. Document the design. The amount of documentation required on most software programs is quite a lot, certainly much more than most programmers, analysts, or program designers are willing to do if left to their own devices. Why do we need so much documentation? **(1)** Each designer must communicate with interfacing designers, managers, and possibly customers. **(2)** During early phases, the documentation is the design. (3) The real monetary value of documentation is to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

3. **Do it twice.** If a computer program is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned. Note that this is simply the entire process done in miniature, to a time scale

that is relatively small with respect to the overall effort. In the first version, the team must have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives, forget the straightforward aspects of the design that aren't worth studying at this early point, and, finally, arrive at an error-free program.

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

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

1.1.2 IN PRACTICE

Some software projects still practice the conventional software management approach.

It is useful to summarize the characteristics of the conventional process as it has typically been applied, which is not necessarily as it was intended. Projects destined for trouble frequently exhibit the following symptoms:

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

Protracted Integration and Late Design Breakage

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

The following sequence was common:

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

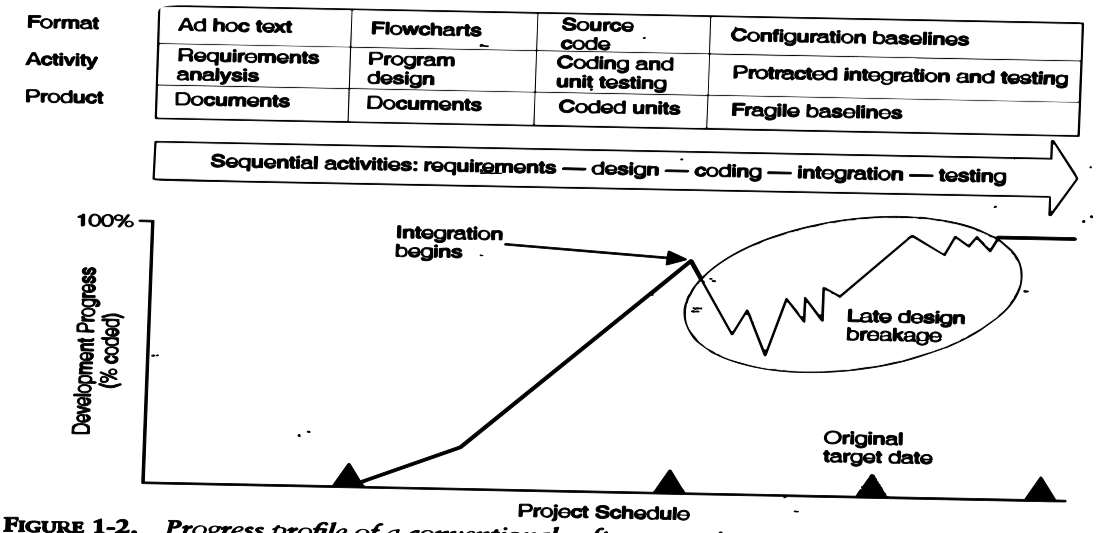


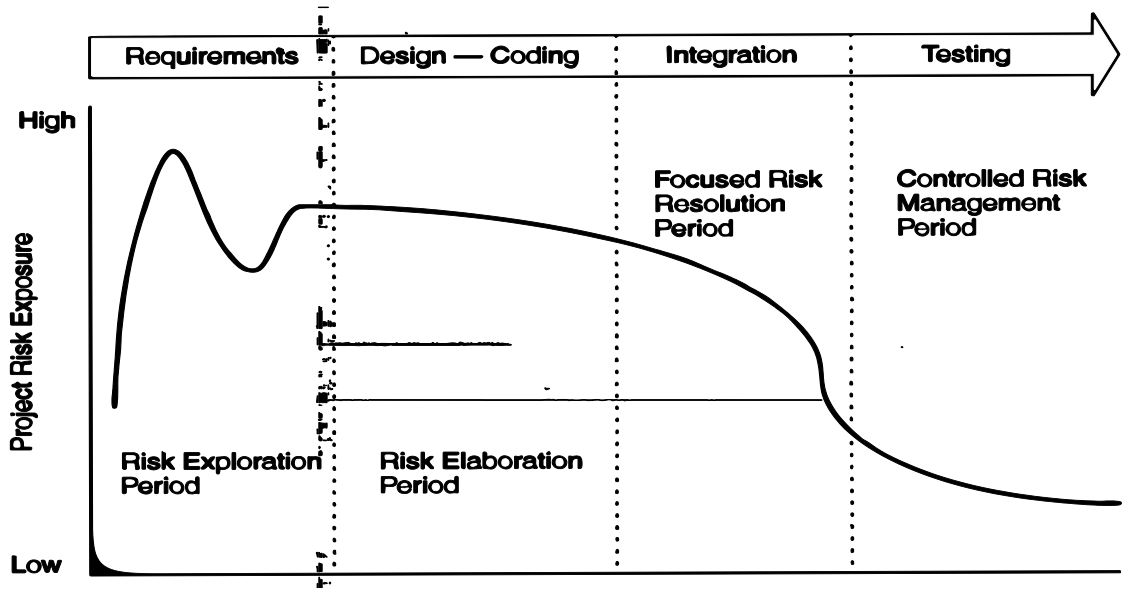
FIGURE 1-2. *Progress profile of a conventional software project*

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

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.

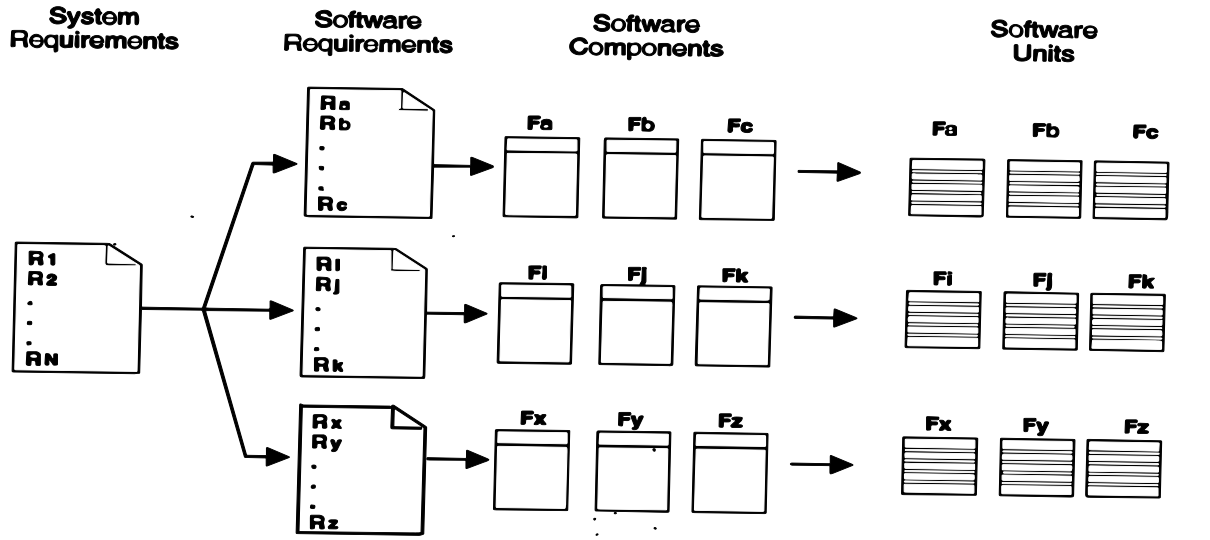


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

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.

1.2 CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

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

1. Finding and fixing a software problem after delivery **costs 100** times more than finding and fixing the problem in early design phases.
2. You can compress software development schedules **25%** of nominal, 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

2.1 SOFTWARE ECONOMICS

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

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

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

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

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. Organiz:1tions used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.
3) ***Modern practices****:* 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custom built

Technologies for environment automation, size reduction, and process improvement are not independent of one another. In each new era, the key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.

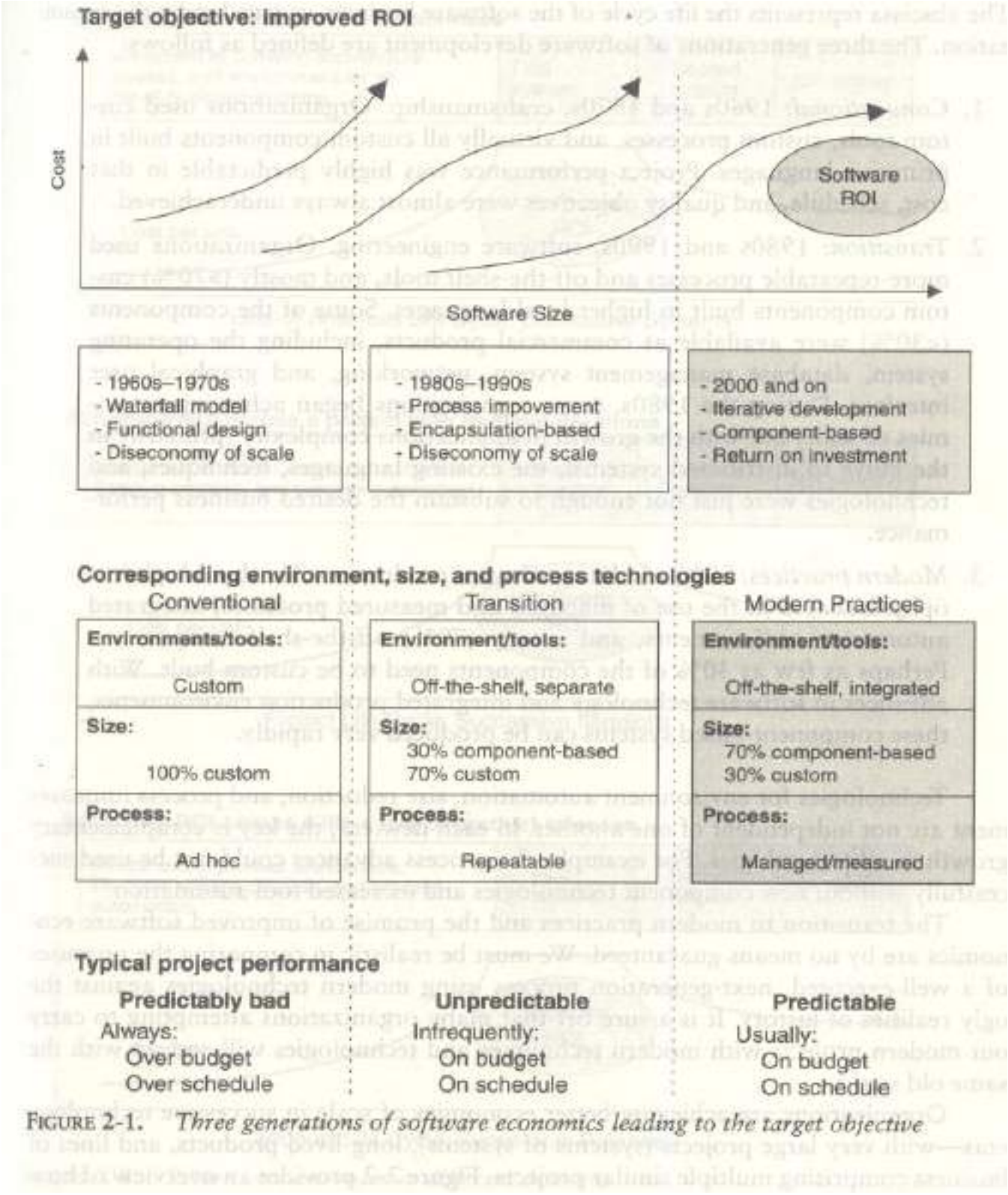


FIGURE 2-1. *Three generations of software economics leading to the target objective*

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

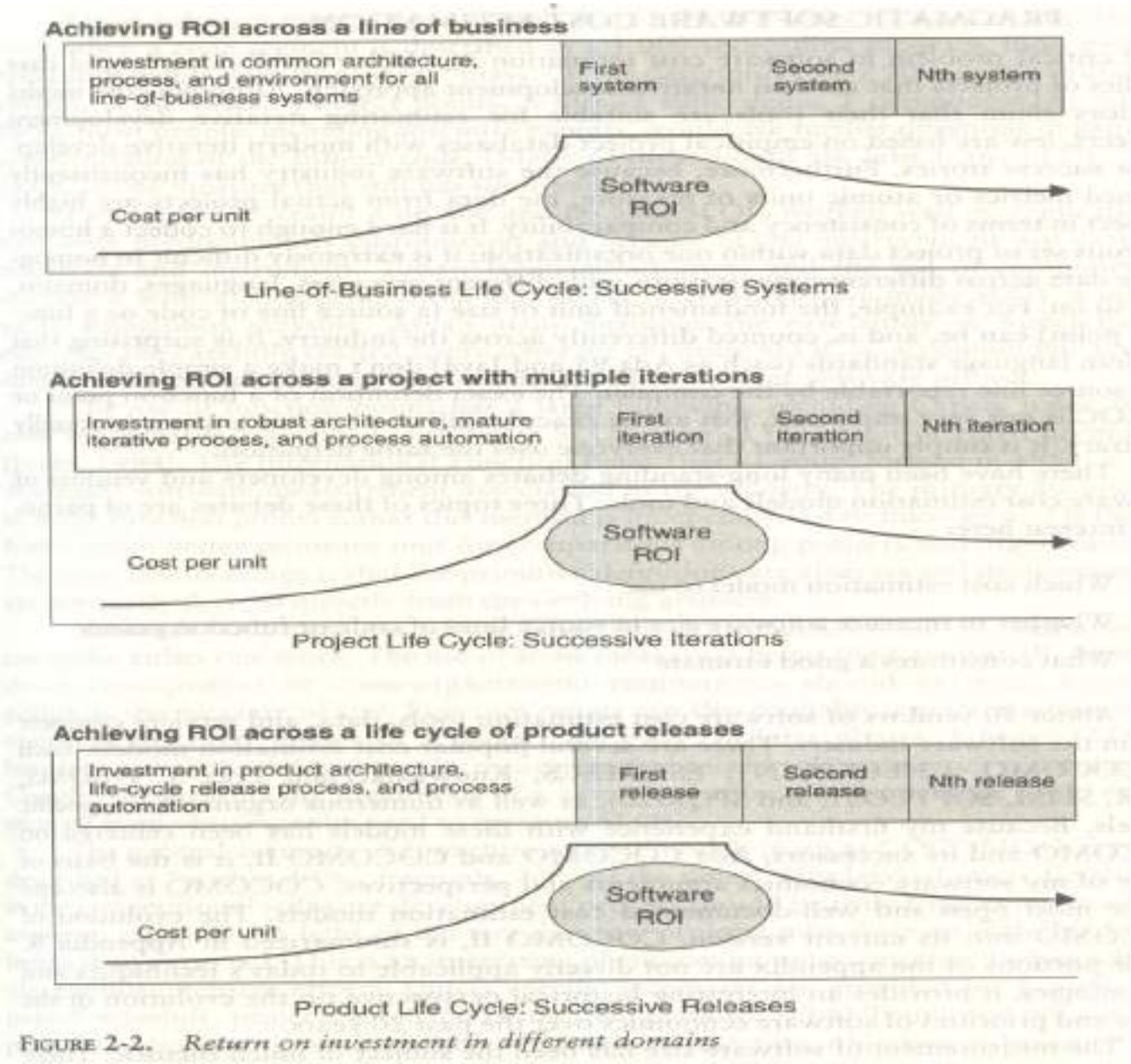


FIGURE 2-2. *Return on investment in different domains*

2.2 PRAGMATIC SOFTWARE COST ESTIMATION

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

There have been many debates among developers and vendors of software cost estimation models and tools. Three topics of these debates are of particular interest here:

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

There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), CO COMO is also one of the

most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described 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.

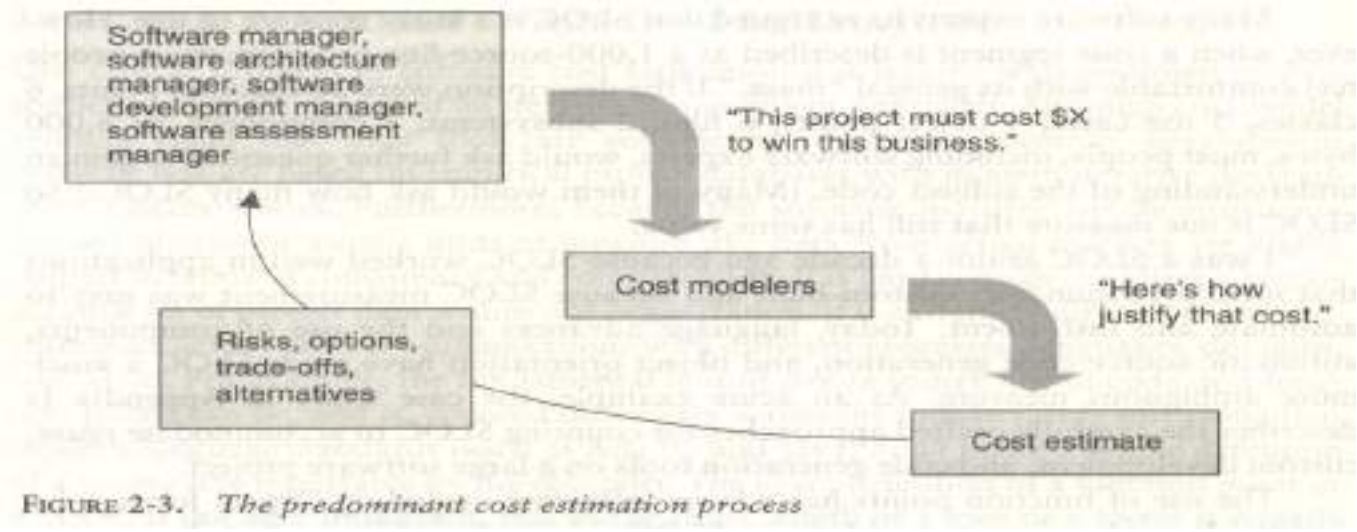


FIGURE 2-3. *The predominant cost estimation process*

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

3.1 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.

3.1.1 LANGUAGES

Universal function points (UFPs[1]) are useful estimators for language-independent, early life-cycle estimates. The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries. SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Substantial data have been documented relating SLOC to function points. Some of these results are shown in Table 3-2.

Languages expressiveness of some of today's popular languages

LANGUAGES	SLOC per UFP
Assembly	320
C	128
FORTAN77	105
COBOL85	91
Ada83	71
C++	56
Ada95	55
Java	55
Visual Basic	35

Table 3-2

3.1.2 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.

[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.

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.

3.1.3 REUSE

Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages. With reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality. Try to treat reuse as a mundane part of achieving a return on investment.

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

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

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

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

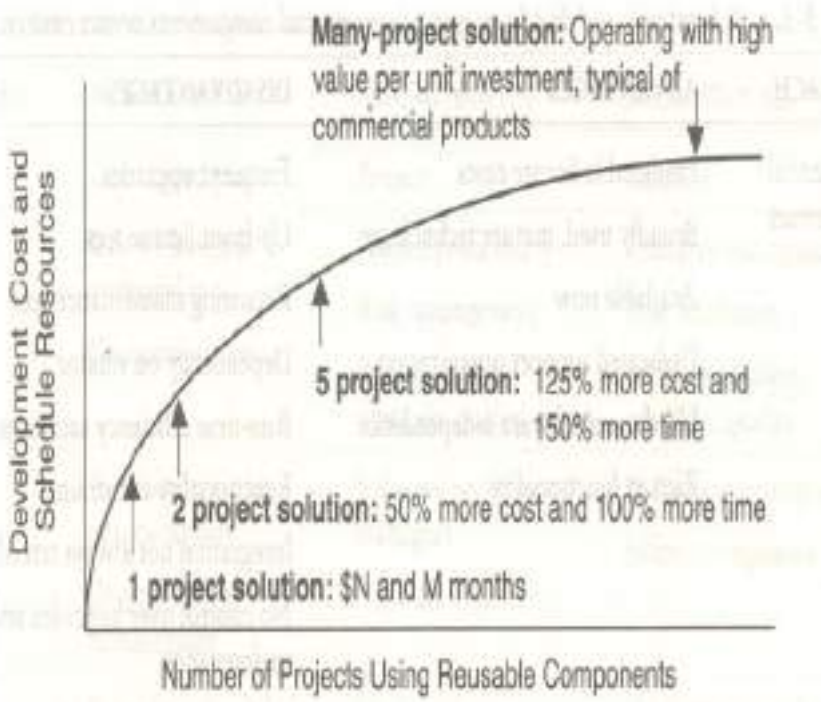


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

3.1.4 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
		Multiple-vendor incompatibilities
Custom development	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

3.2 IMPROVING SOFTWARE PROCESSES

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

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

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

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

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

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

3.3 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

3.4 IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The tools and environment used in the software process generally have a linear effect on the productivity of the process. Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces provide crucial automation support for evolving the software engineering artifacts. Above all, configuration management environments provide the foundation for executing and instrument the process. At first order, the isolated impact of tools and automation generally allows improvements of 20% to 40% in effort. However, tools and environments must be viewed as the primary delivery vehicle for process automation and improvement, so their impact can be much higher.

Automation of the design process provides payback in quality, the ability to estimate costs and schedules, and overall productivity using a smaller team.

Round-trip engineering 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.

3.5 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.

3.6 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.

UNIT – II

Conventional and Modern Software Management: The principles of conventional software Engineering, principles of modern software management, transitioning to an iterative process.
Life cycle phases: Engineering and Production stages, Inception, Elaboration, Construction, Transition Phases.

4. CONVENTIONAL AND MODERN SOFTWARE MANAGEMENT

4.1 THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

1.**Make quality** 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 through-out 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.

4.2 THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

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

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

4. **Establish a *change management environment.*** The dynamics of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baselines.

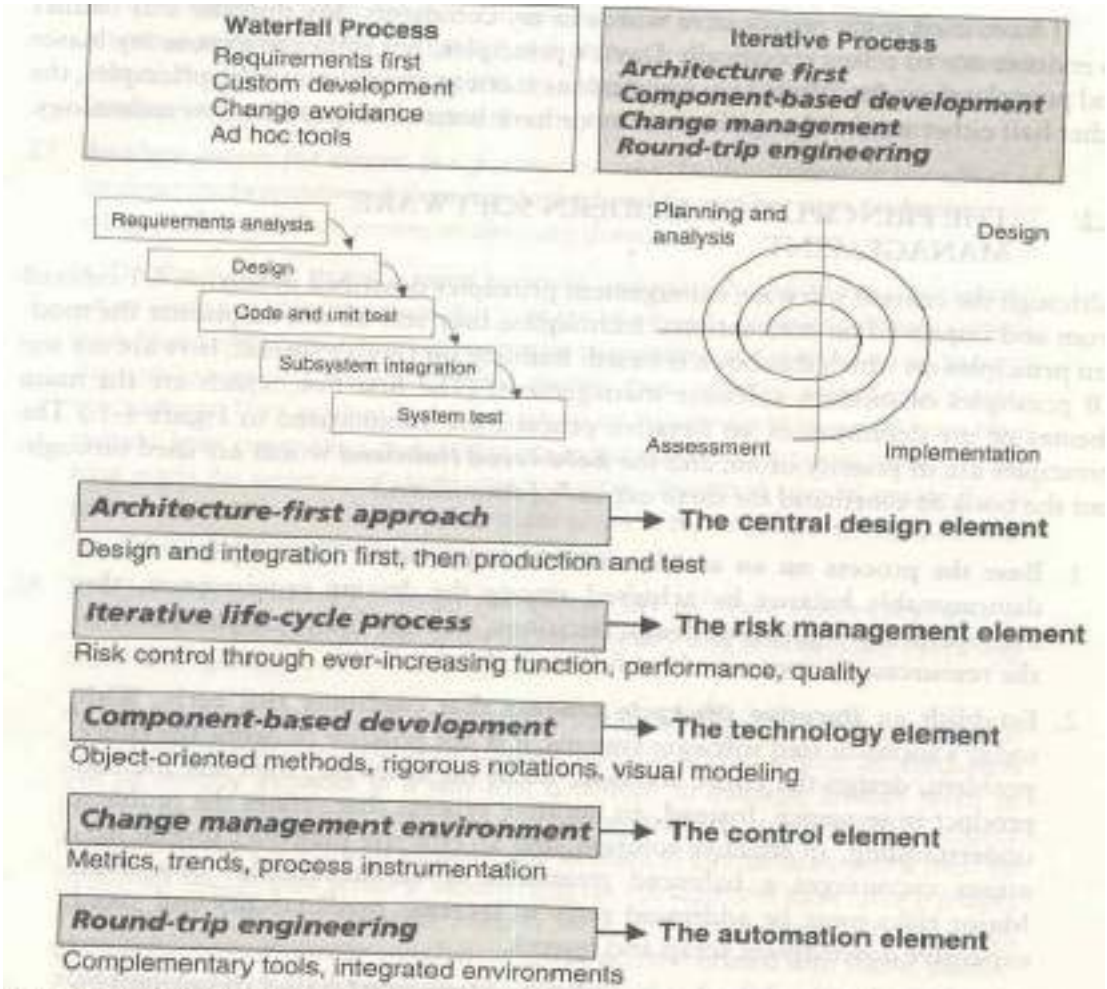


FIGURE 4-1. *The top five principles of a modern process*

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

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

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

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

4.3 TRANSITIONING TO AN ITERATIVE PROCESS

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

The economic benefits inherent in transitioning from the conventional waterfall model to an iterative development process are significant but difficult to quantify. As one benchmark of the expected economic impact of process improvement, consider the process exponent parameters of the COCOMO II model. (Appendix B provides more detail on the COCOMO model) This exponent can range from 1.01 (virtually no diseconomy of scale) to 1.26 (significant diseconomy of scale). The parameters that govern the value of the process exponent are application precedentedness, process flexibility, architecture risk resolution, team cohesion, and software process maturity.

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

- **Application precedentedness**. Domain experience is a critical factor in understanding how to plan and execute a software development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental. This is one of the primary reasons that the software industry has moved to an ***iterative life-cycle process.*** Early iterations in the life cycle establish precedents from which the product, the process, and the plans can be 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.***

Important questions

1.	***Explain briefly Waterfall model. Also explain Conventional s/w management performance?***
2.	***Define Software Economics. Also explain Pragmatic s/w cost estimation?***
3.	***Explain Important trends in improving Software economics?***
4.	***Explain five staffing principal offered by Boehm. Also explain Peer Inspections?***
5..	***Explain principles of conventional software engineering?***
6.	***Explain briefly principles of modern software management***

5. 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

5.1 ENGINEERING AND PRODUCTION STAGES

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

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

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

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

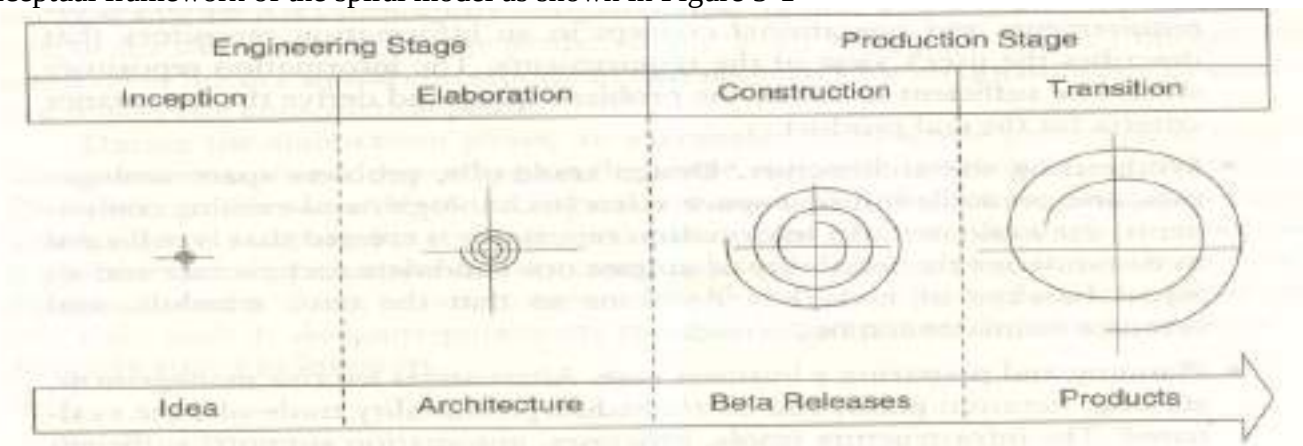


FIGURE 5-1. *The phases of the life-cycle process*

5.2 INCEPTION PHASE

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTMTIES

- 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

5.2 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?

5.4 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?

5.5 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?

UNIT - III

Artifacts of the process: The artifact sets, Management artifacts, Engineering artifacts, programmatic artifacts.
Model based software architectures: A Management perspective and technical perspective.

6. Artifacts of the process

6.1 THE ARTIFACT SETS

To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. A*rtifact* represents cohesive information that typically is developed and reviewed as a single entity.

Life-cycle software artifacts are organized into five distinct sets that are roughly partitioned by the underlying language of the set: management (ad hoc textual formats), requirements (organized text and models of the problem space), design (models of the solution space), implementation (human-readable programming language and associated source files), and deployment (machine-process able languages and associated files). The artifact sets are shown in Figure 6-1.

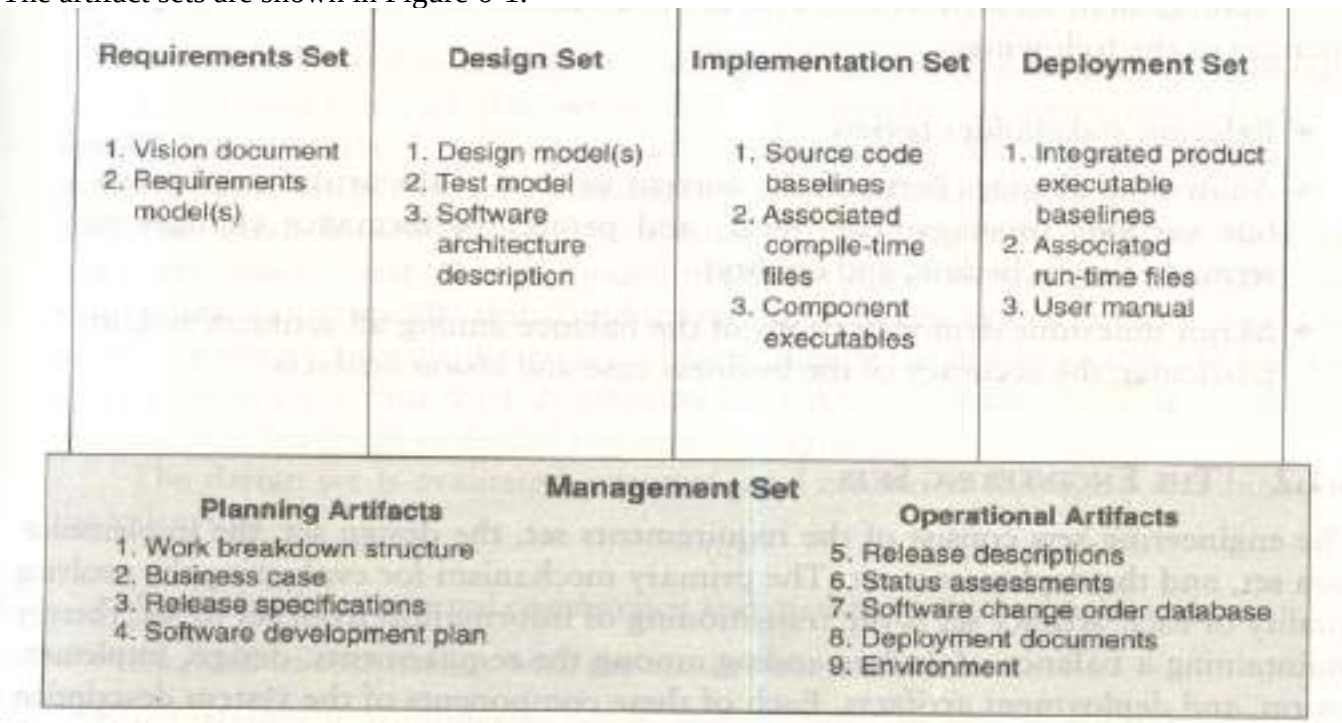


FIGURE 6-1. *Overview of the artifact sets*

6.1.1 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

6.1.2 THE ENGINEERING SETS

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

Requirements Set

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

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

Design Set

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

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

Implementation set

The implementation set includes source code (programming language notations) that represents the tangible implementations of components (their form, interface, and dependency relationships)

Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of the following:

- Analysis of consistency with the design models
- Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets
- Assessment of component source or executable files against relevant evaluation criteria through inspection, analysis, demonstration, or testing
- Execution of stand-alone component test cases that automatically compare expected results with actual results
- Analysis of changes between the current version of the implementation set and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Deployment Set

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

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

- Testing against the usage scenarios and quality attributes defined in the requirements set to evaluate the consistency and completeness and the~ semantic balance between information in the two sets
- Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system (platform type, number, network topology)
- Testing against the defined usage scenarios in the user manual such as installation, user-oriented dynamic reconfiguration, mainstream usage, and anomaly management
- Analysis of changes between the current version of the deployment set and previous versions (defect elimination trends, performance changes)
- Subjective review of other dimensions of quality

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

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

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

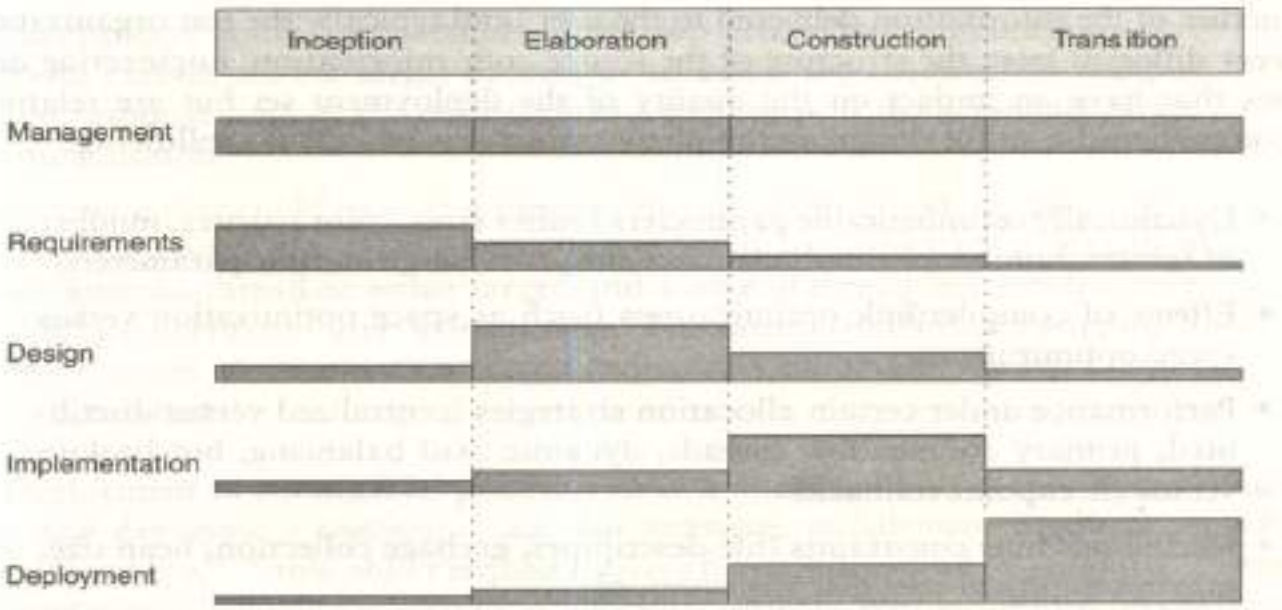


FIGURE 6-2. *Life-cycle focus on artifact sets*

Implementation Set versus Deployment Set

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

- Dynamically reconfigurable parameters (buffer sizes, color palettes, number of servers, number of simultaneous clients, data files, run-time parameters)
- Effects of compiler/link optimizations (such as space optimization versus speed optimization)
- Performance under certain allocation strategies (centralized versus distributed, primary and shadow threads, dynamic load balancing, hot backup versus checkpoint/rollback)
- Virtual machine constraints (file descriptors, garbage collection, heap size, maximum record size, disk file rotations)
- Process-level concurrency issues (deadlock and race conditions)
- Platform-specific differences in performance or behavior

6.1.3 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.

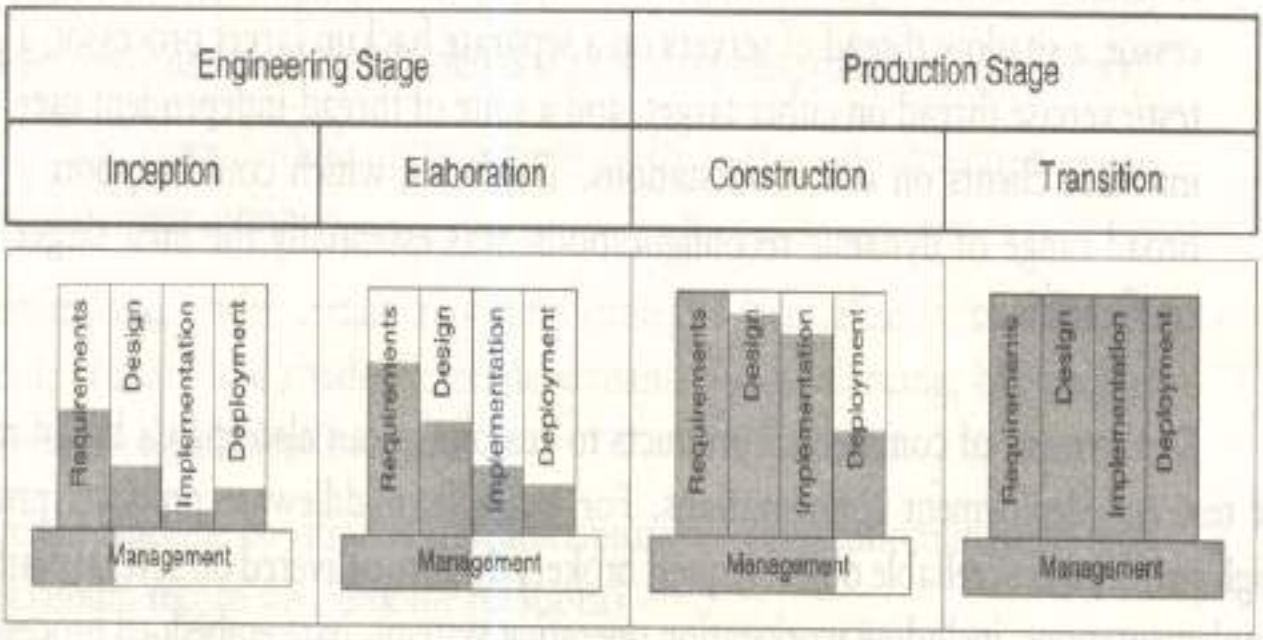


FIGURE 6-3. *Life-cycle evolution of the artifact sets*

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.

6.1.4 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.

6.2 MANAGEMENT ARTIFACTS

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

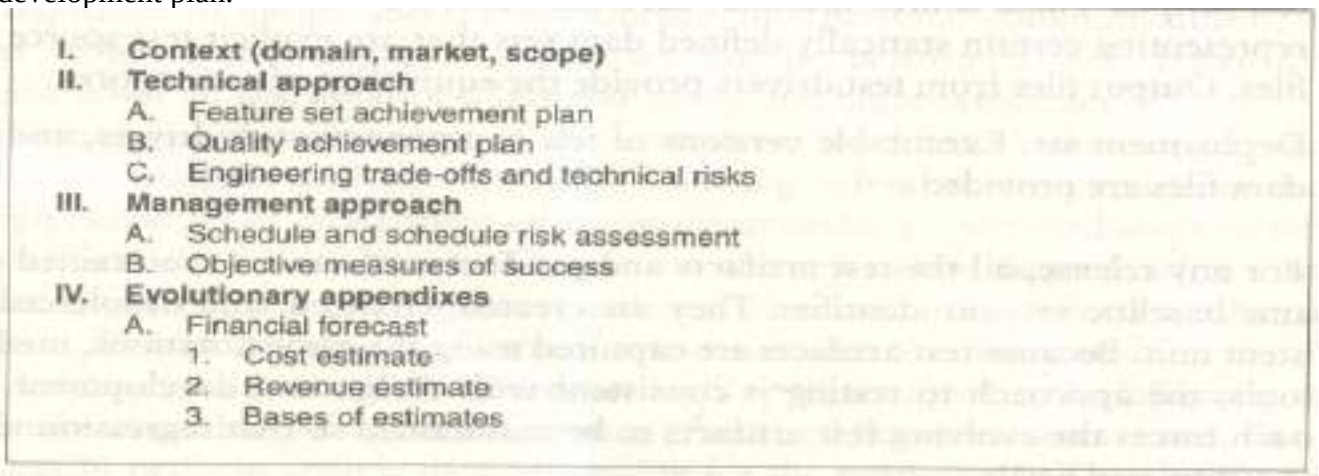
Business Case

The business case artifact provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans, and backup data necessary to demonstrate the risks and realism of the plans. The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progresses. Figure 6-4

provides a default outline for a business case.

Software Development Plan

The software development plan (SDP) elaborates the process framework into a fully detailed plan. Two indications of a useful SDP are periodic updating (it is not stagnant shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.



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

FIGURE 6-4. *Typical business case outline*

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

FIGURE 6-5. *Typical software development plan outline*

Work Breakdown Structure

Work breakdown structure (WBS) is the vehicle for budgeting and collecting costs. To monitor and control a project's financial performance, the software project man1ger must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

Software Change Order Database

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

Release Specifications

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

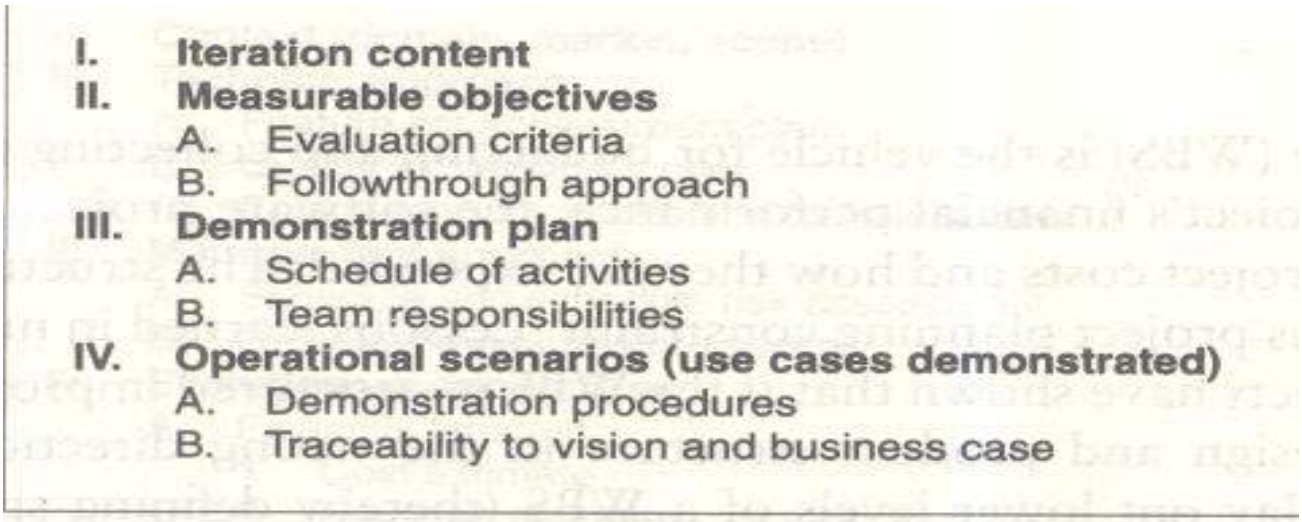


FIGURE 6-6. *Typical release specification outline*

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

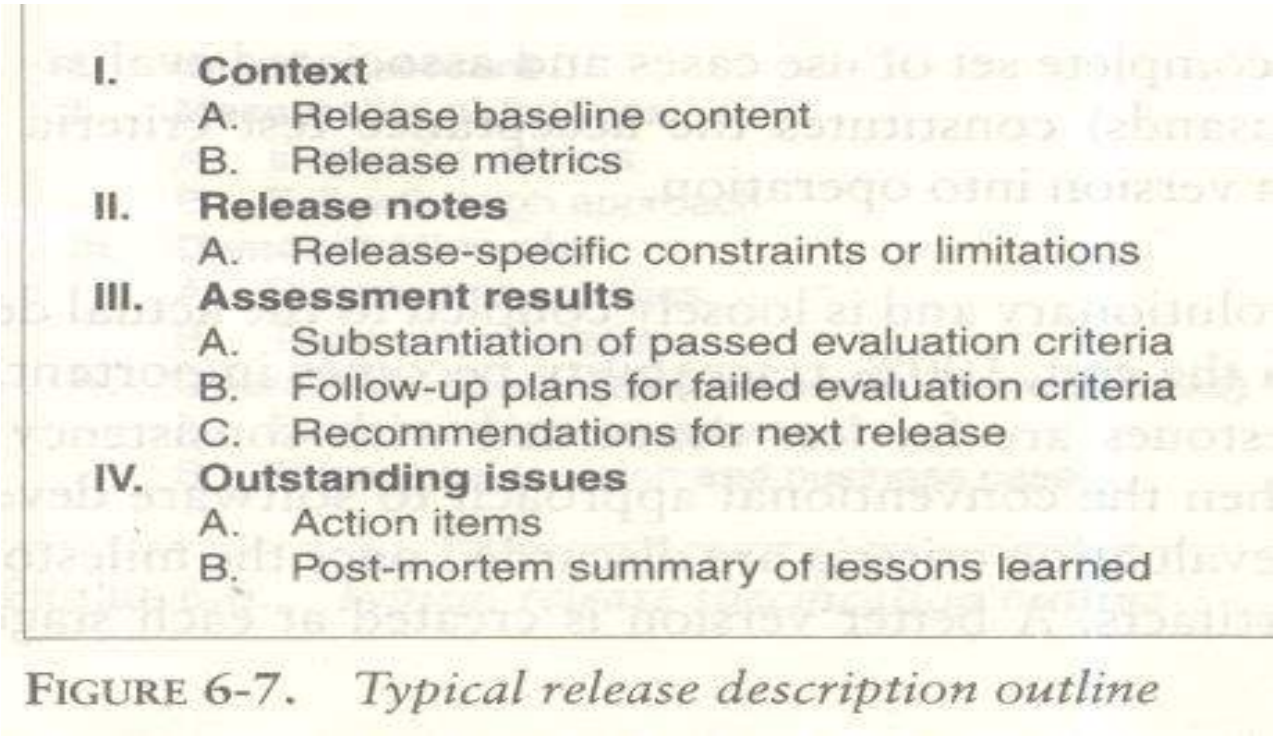


FIGURE 6-7. *Typical release description outline*

Environment

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

Deployment

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

Management Artifact Sequences

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

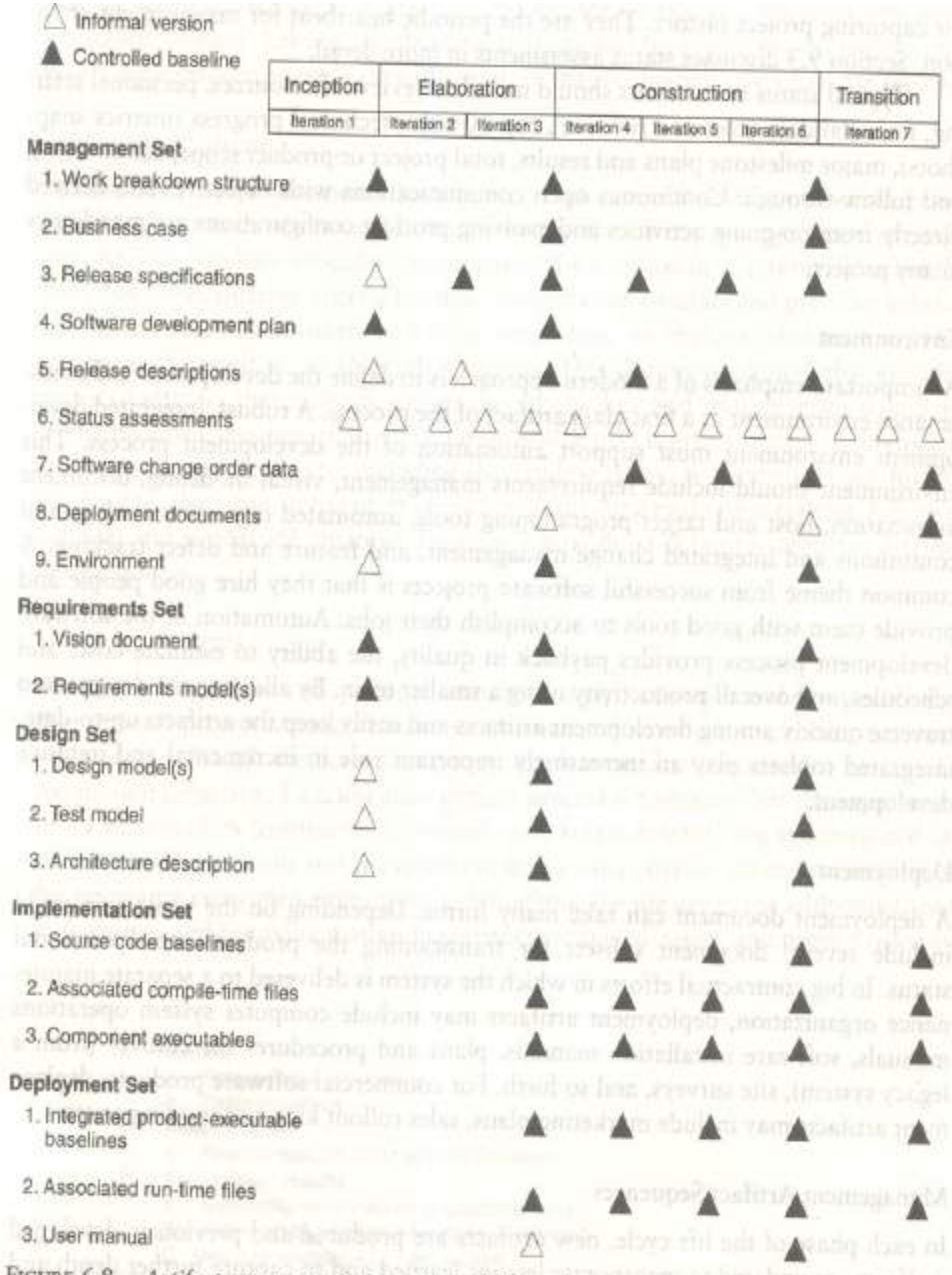


FIGURE 6-8. *Artifact sequences across a typical life cycle*

6.3 ENGINEERING ARTIFACTS

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

Vision Document

The vision document provides a complete vision for the software system under development and. supports the contract between the funding authority and 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.

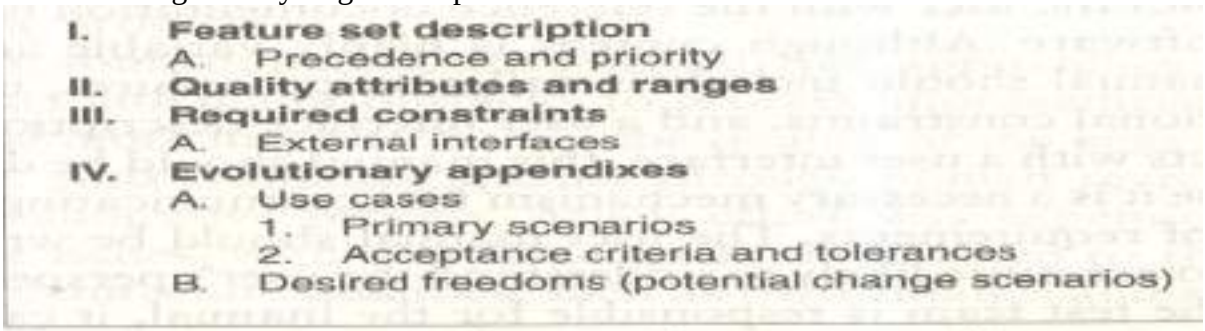


FIGURE 6-9. *Typical vision document outline*

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.

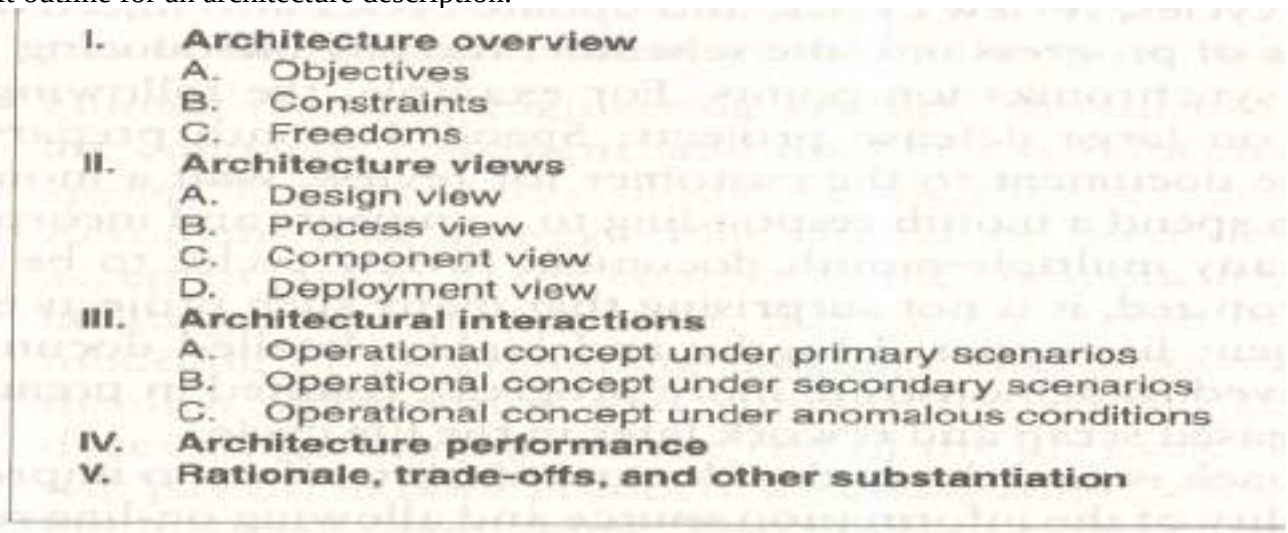


FIGURE 6-10. *Typical architecture description outline*

Software User Manual

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

6.4 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 Important questions

1.	**Explain briefly two stages of the life cycle engineering and production.**
2.	**Explain different phases of the life cycle process?**
3.	**Explain the goal of Inception phase, Elaboration phase, Construction phase and Transition phase.**
4.	**Explain the overview of the artifact set**
5.	**Write a short note on** **(a) Management Artifacts (b) Engineering Artifacts (c) Pragmatic Artifacts**

7.Model based software architecture

7.1 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.

7.2 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

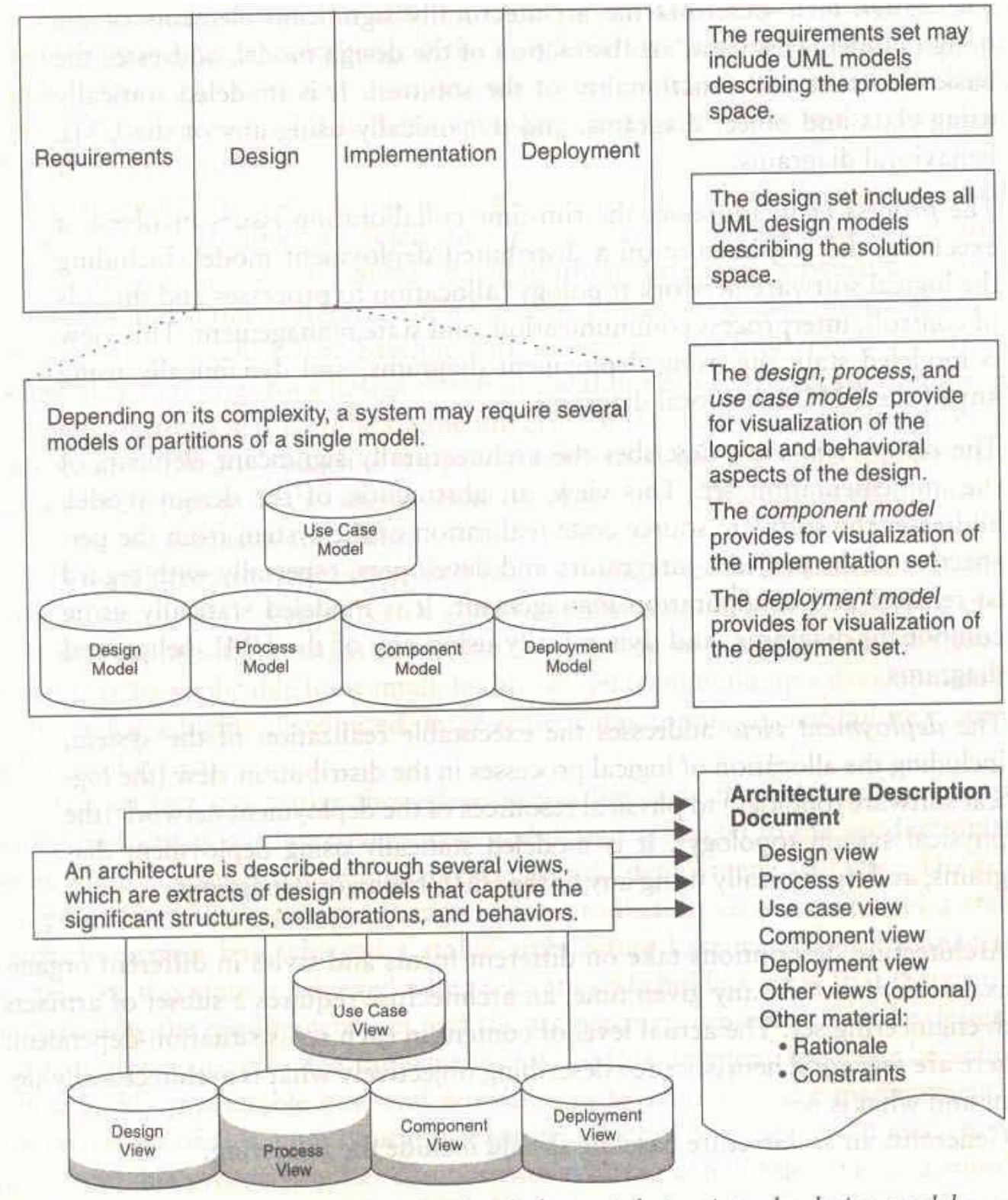


FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

UNIT - IV

Work Flows of the process: Software process workflows, Iteration workflows.
Checkpoints of the process: Major mile stones, Minor Milestones, Periodic status assessments.
Iterative Process Planning: Work breakdown structures, planning guidelines, cost and schedule estimating, Iteration planning process, Pragmatic planning

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.

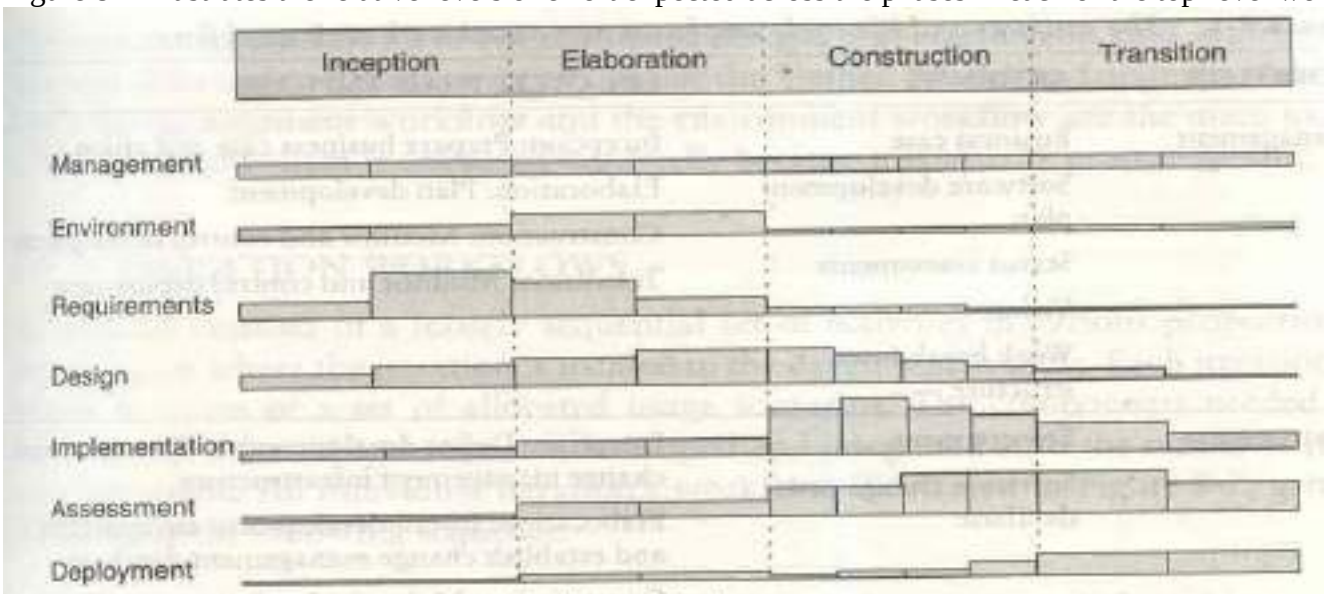


FIGURE 8-1. *Activity levels across the life-cycle phases*

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

ITERATION WORKFLOWS

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

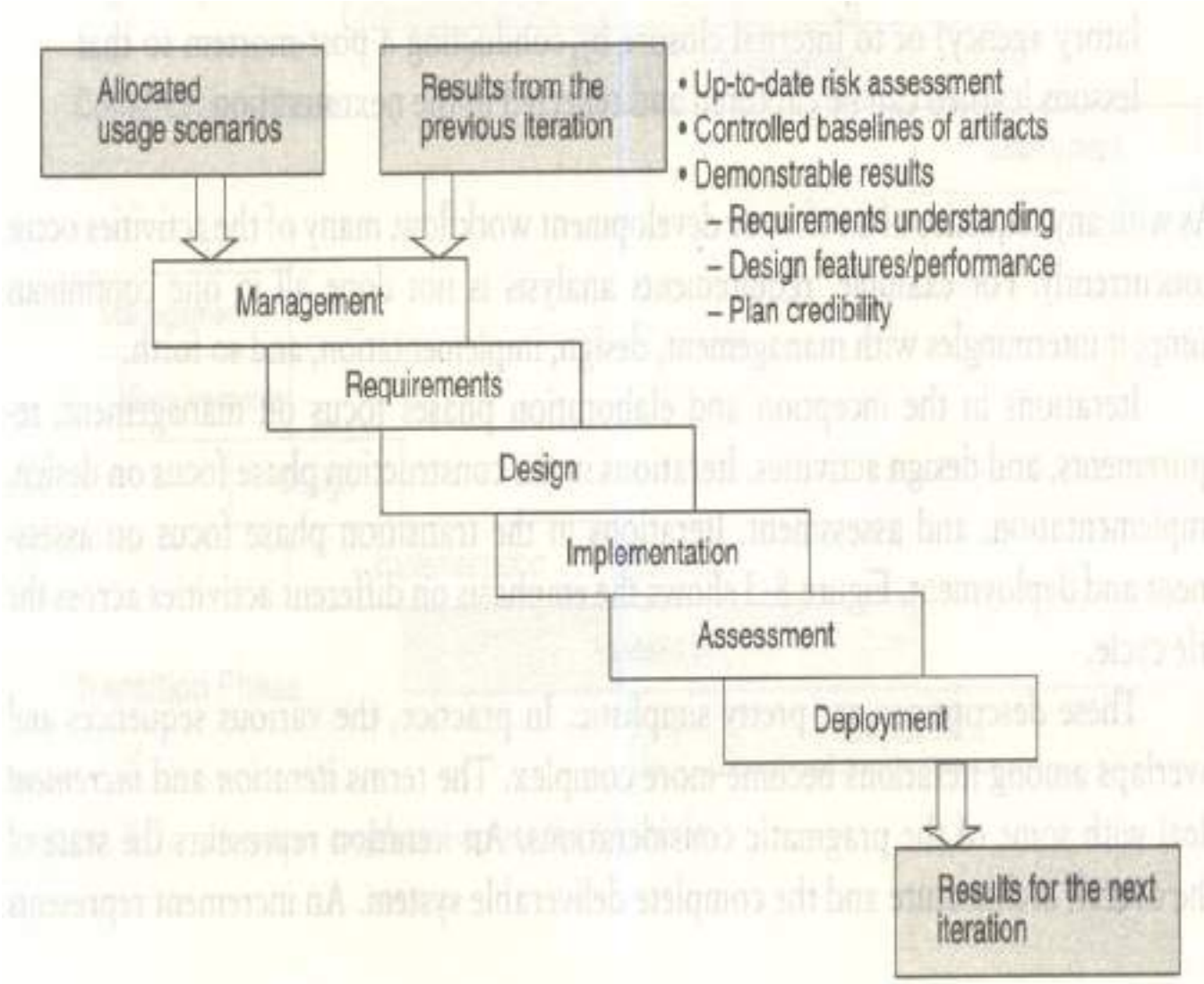


FIGURE 8-2. *The workflow of an iteration*

- 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 from the next iteration. Figure 8-4, an example of a simple development life cycle, illustrates the differences between iterations and increments.

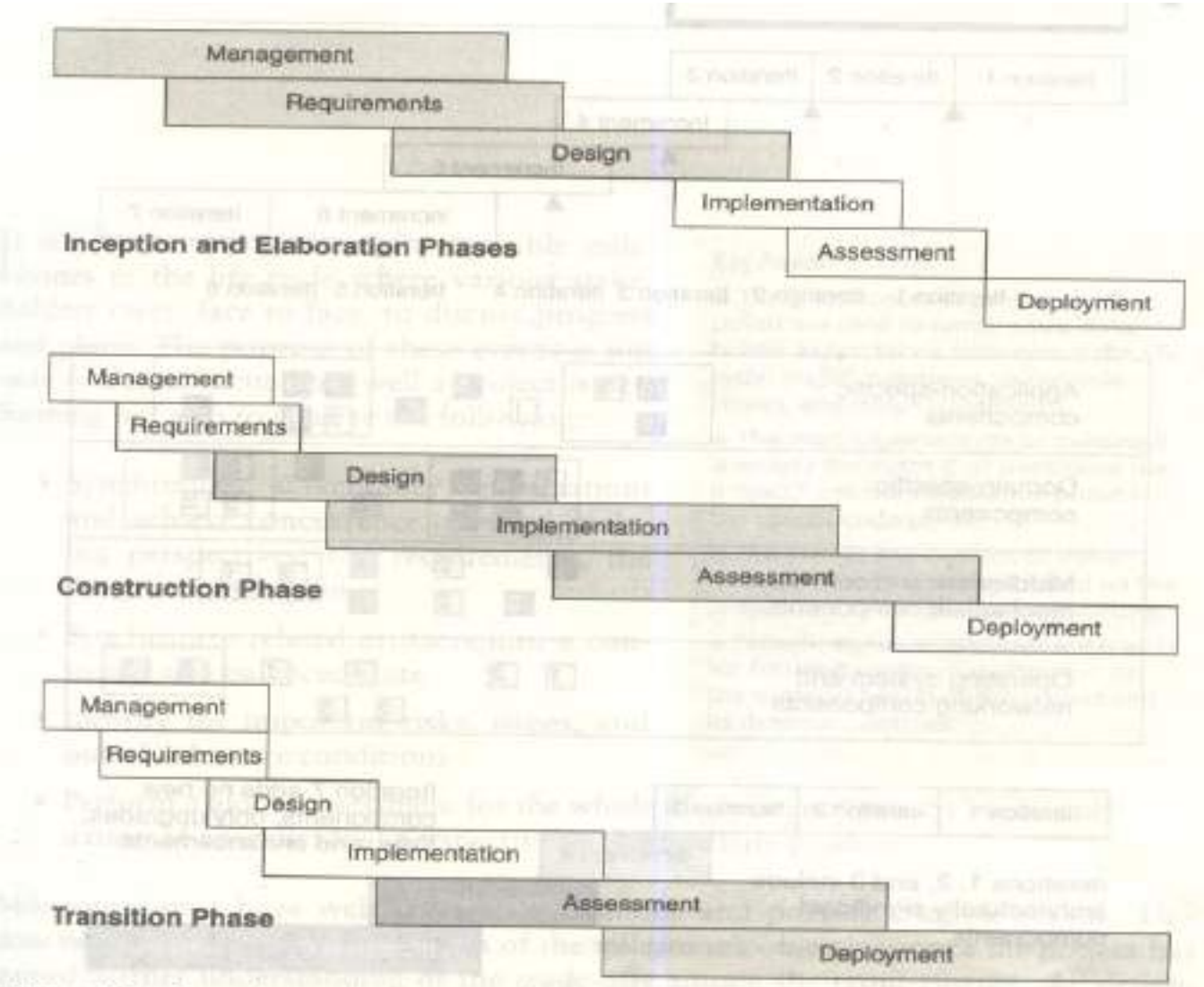


FIGURE 8-3. *Iteration emphasis across the life cycle*

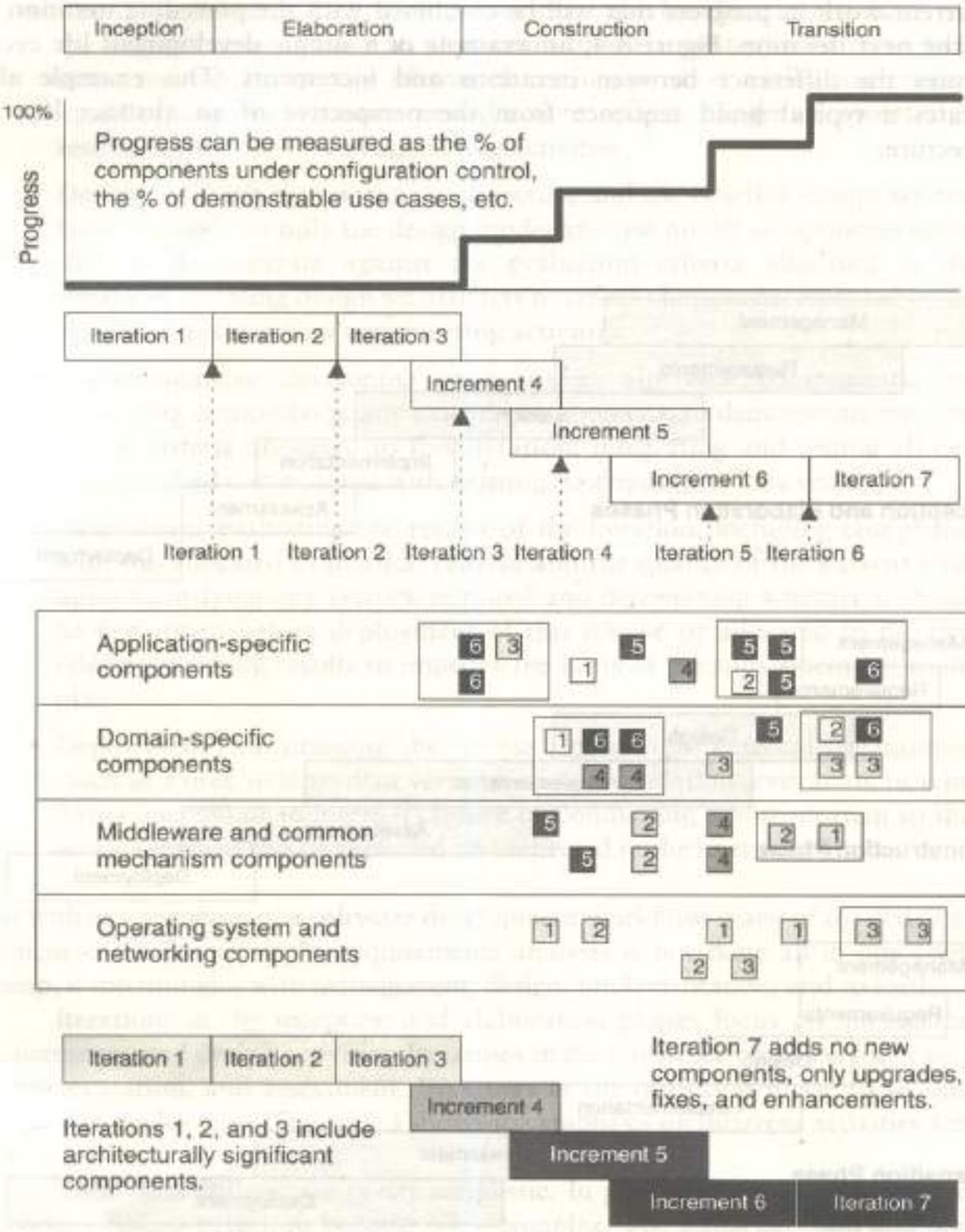


FIGURE 8-4. *A typical build sequence associated with a layered architecture*

9. 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.

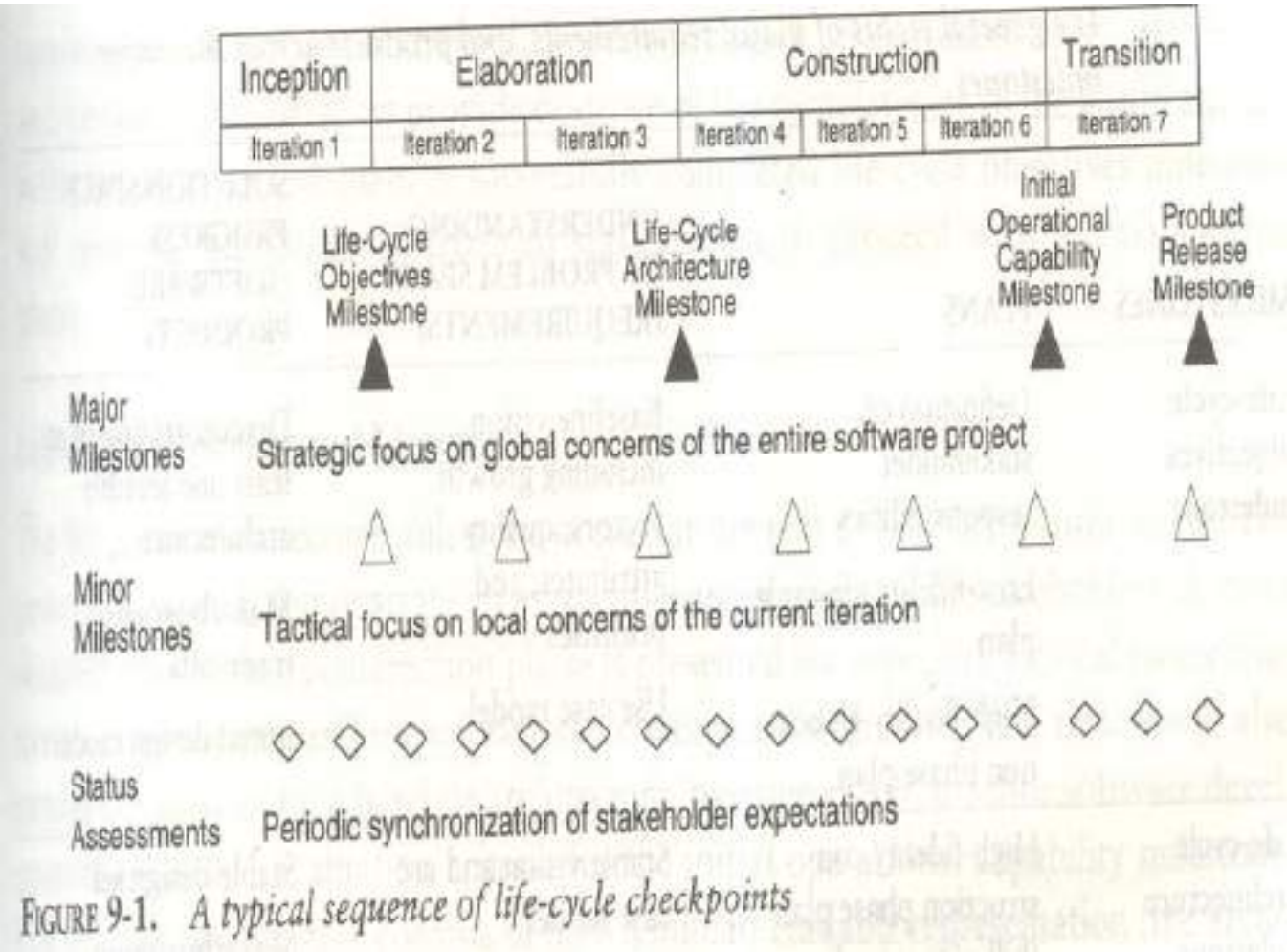


FIGURE 9-1. *A typical sequence of life-cycle checkpoints*

9.1 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.

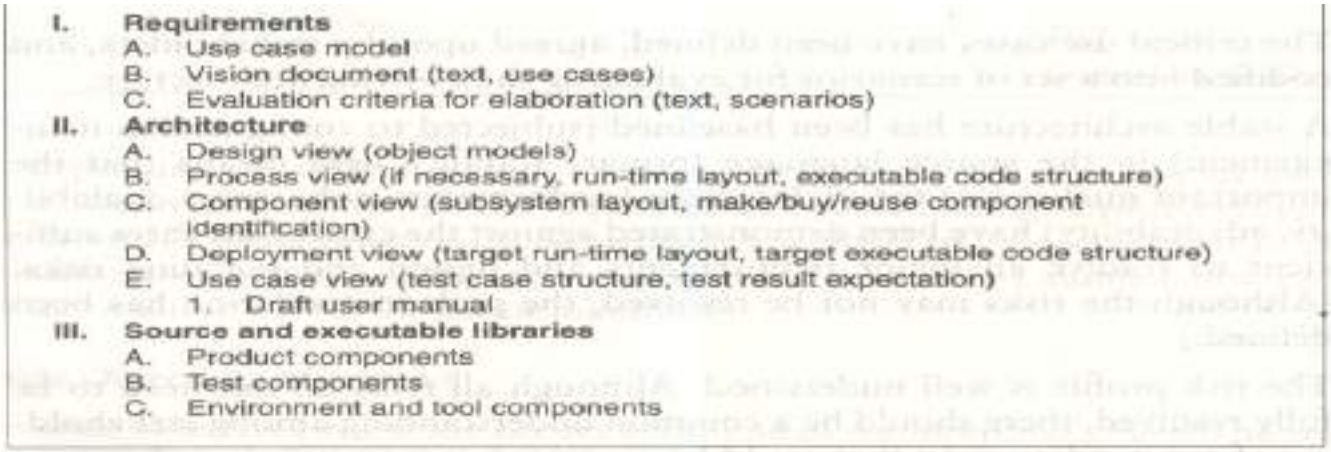
TABLE 9-1. *The general status of plans, requirements, and products across the major milestones*

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

Life-Cycle Objectives Milestone
The life-cycle objectives milestone occurs at 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.

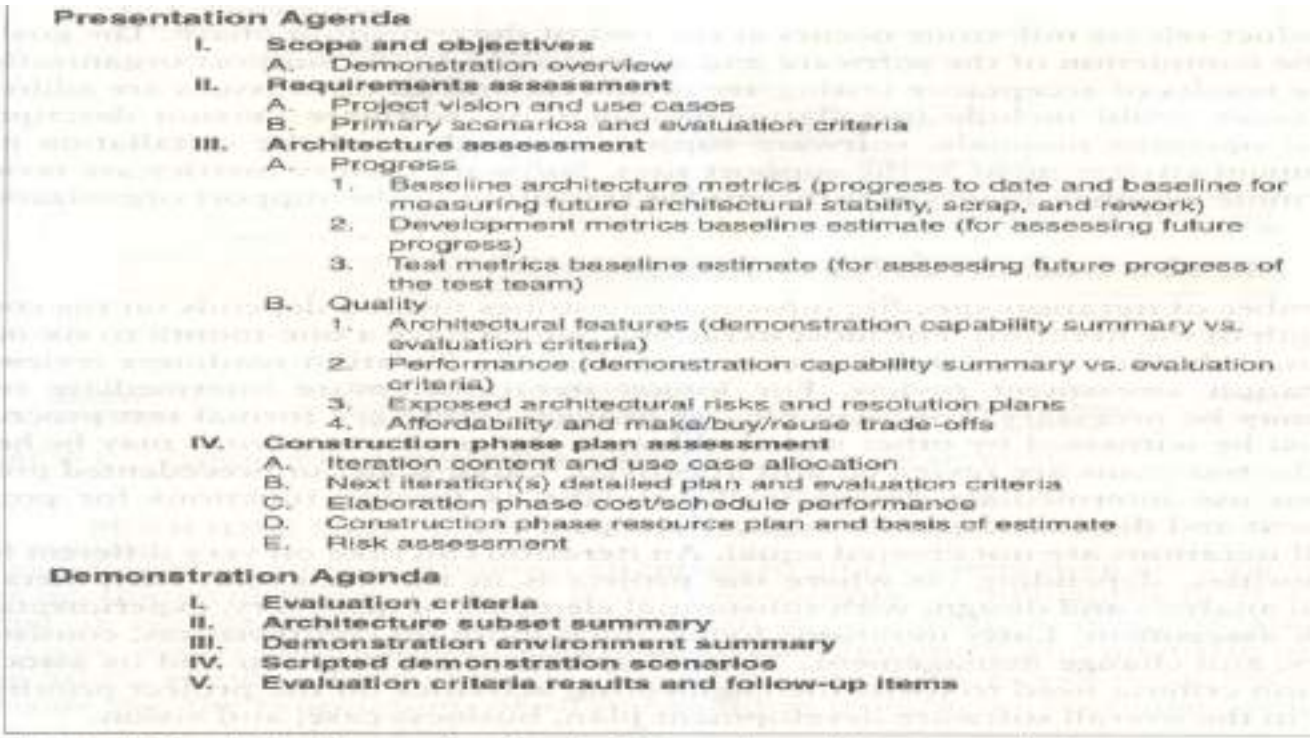
Life-Cycle Architecture Milestone
The life-cycle architecture milestone occurs at the end of the elaboration phase. The primary goal is to demonstrate an executable architecture to all stakeholders. The baseline architecture consists of both a human-readable representation (the architecture document) 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
      1. Baseline architecture metrics (progress to date and baseline for measuring future architectural stability, scrap, and rework)
      2. Development metrics baseline estimate (for assessing future progress)
      3. Test metrics baseline estimate (for assessing future progress of the test team)
   B. Quality
      1. Architectural features (demonstration capability summary vs. evaluation criteria)
      2. Performance (demonstration capability summary vs. evaluation criteria)
      3. Exposed architectural risks and resolution plans
      4. Affordability and make/buy/reuse trade-offs
IV. **Construction phase plan assessment**
   A. Iteration content and use case allocation
   B. Next iteration(s) detailed plan and evaluation criteria
   C. Elaboration phase cost/schedule performance
   D. Construction phase resource plan and basis of estimate
   E. Risk assessment

Demonstration Agenda

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

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.

9.2 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.

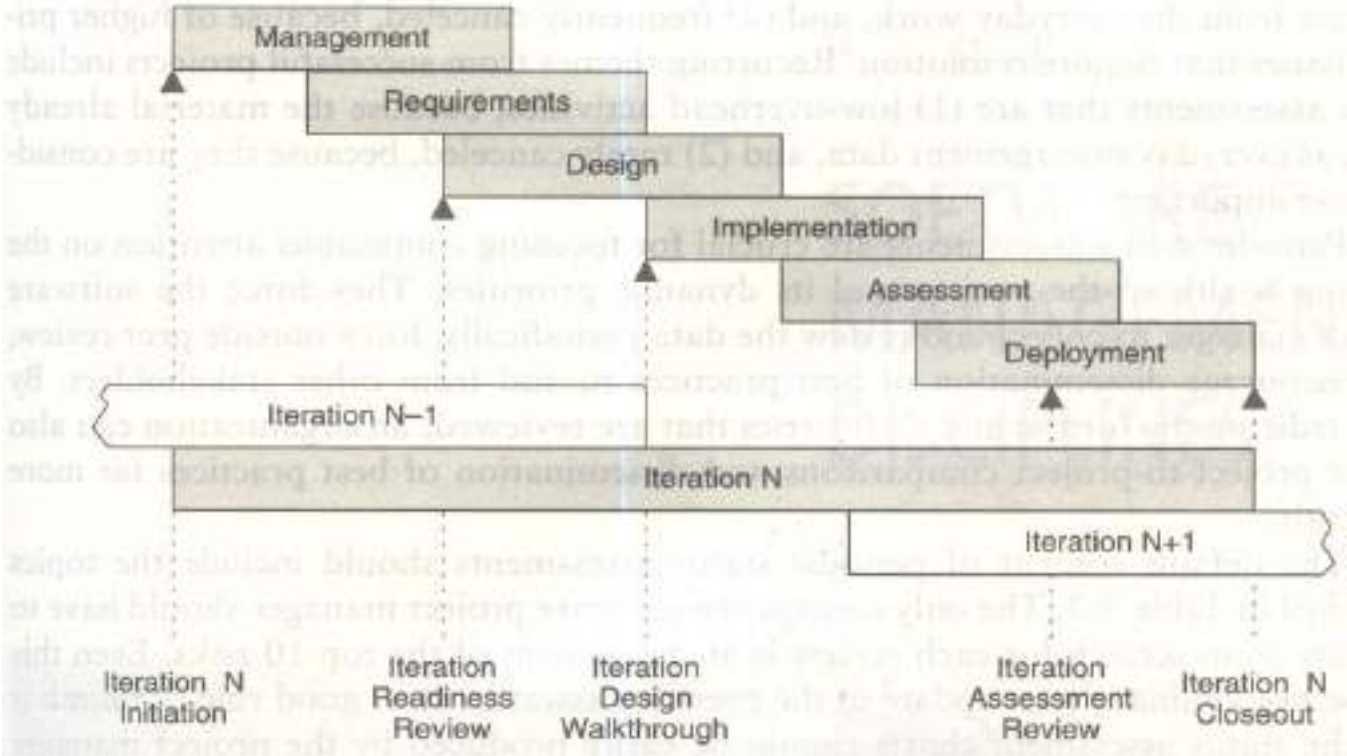


FIGURE 9-4. *Typical minor milestones in the life cycle of an iteration*

9.3 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

10. Iterative 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.

10.1.1 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
Documentation
…(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

10.1.2 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

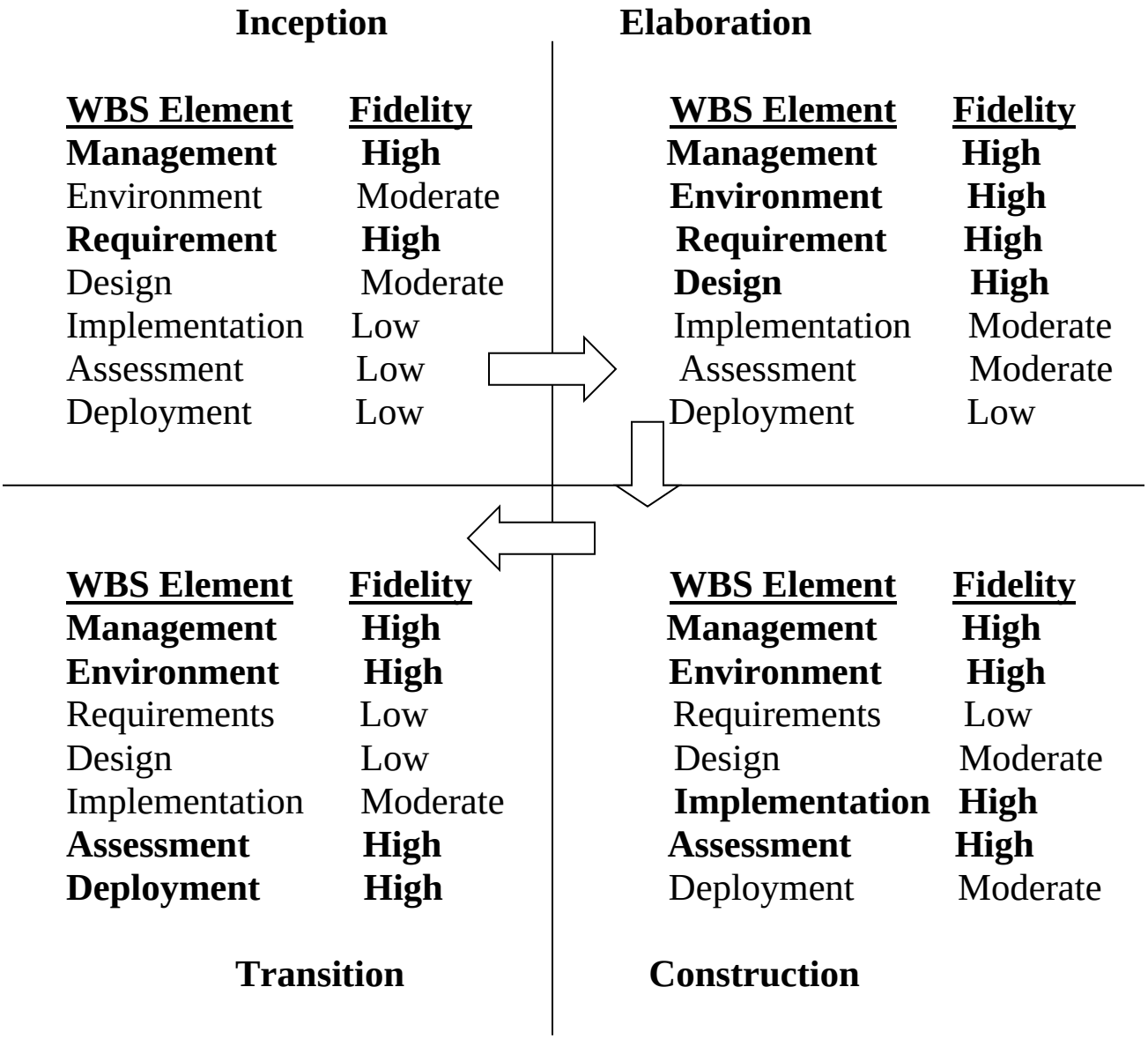
A Management
- **AA Inception phase management**
  - **AAA Business case development**
  - **AAB Elaboration phase release specifications**
  - **AAC Elaboration phase WBS specifications**
  - **AAD Software development plan**
  - **AAE Inception phase project control and status assessments**
- **AB 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



10.2 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 pe 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	Construction	Transition
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

10.3 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:

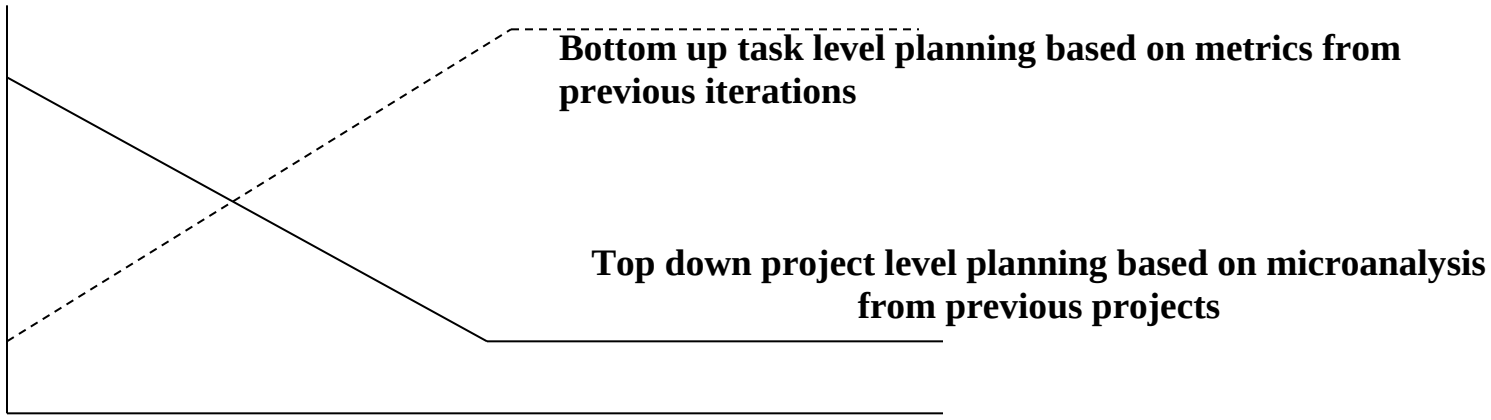
1. The lowest level WBS elements are elaborated into detailed tasks
2. Estimates are combined and integrated into higher level budgets and milestones.
3. Comparisons are made with the top-down budgets and schedule milestones.

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	

10.4 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.

10.5 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 – Important Questions

1.	**Define Model-Based software architecture?**
2.	**Explain various process workflows?**
3.	**Define typical sequence of life cycle checkpoints?**
4.	**Explain general status of plans, requirements and product across the major milestones.**
5.	**Explain conventional and Evolutionary work break down structures?**
6.	**Explain briefly planning balance throughout the life cycle?**

UNIT - V

Project Organizations and Responsibilities: Line-of-Business Organizations, Project Organizations, evolution of Organizations.

Process Automation: Automation Building blocks, The Project Environment.

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.

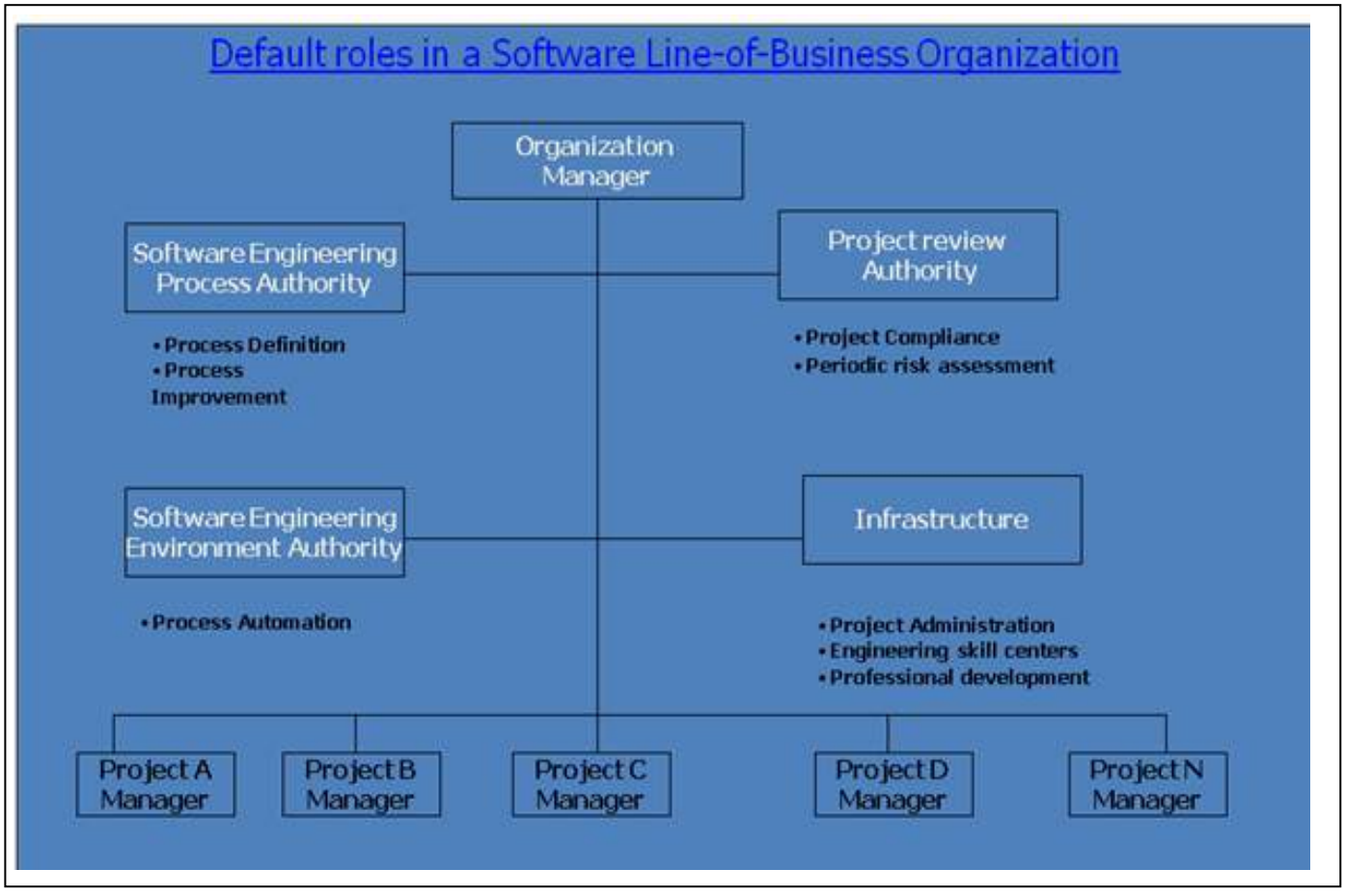


Fig: Default roles in a software Line-of-Business Organization.

Software Engineering Process Authority (SEPA)
The SEPA facilities the exchange of information & process guidance both to & from project practitioners

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

Project Review Authority (PRA)

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

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

Software Engineering Environment Authority(SEEA)
The SEEA is responsible for automating the organization's process, 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:

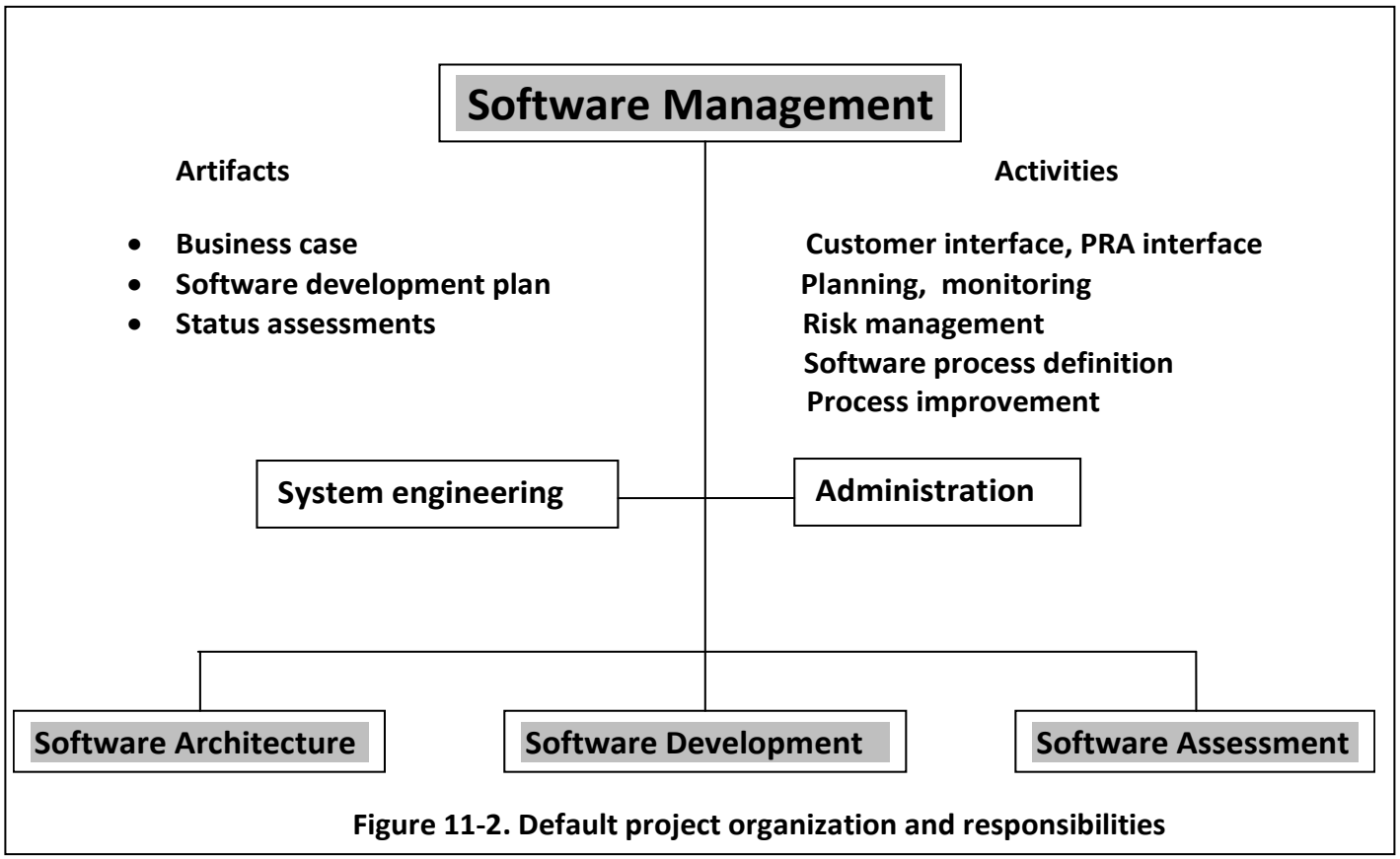
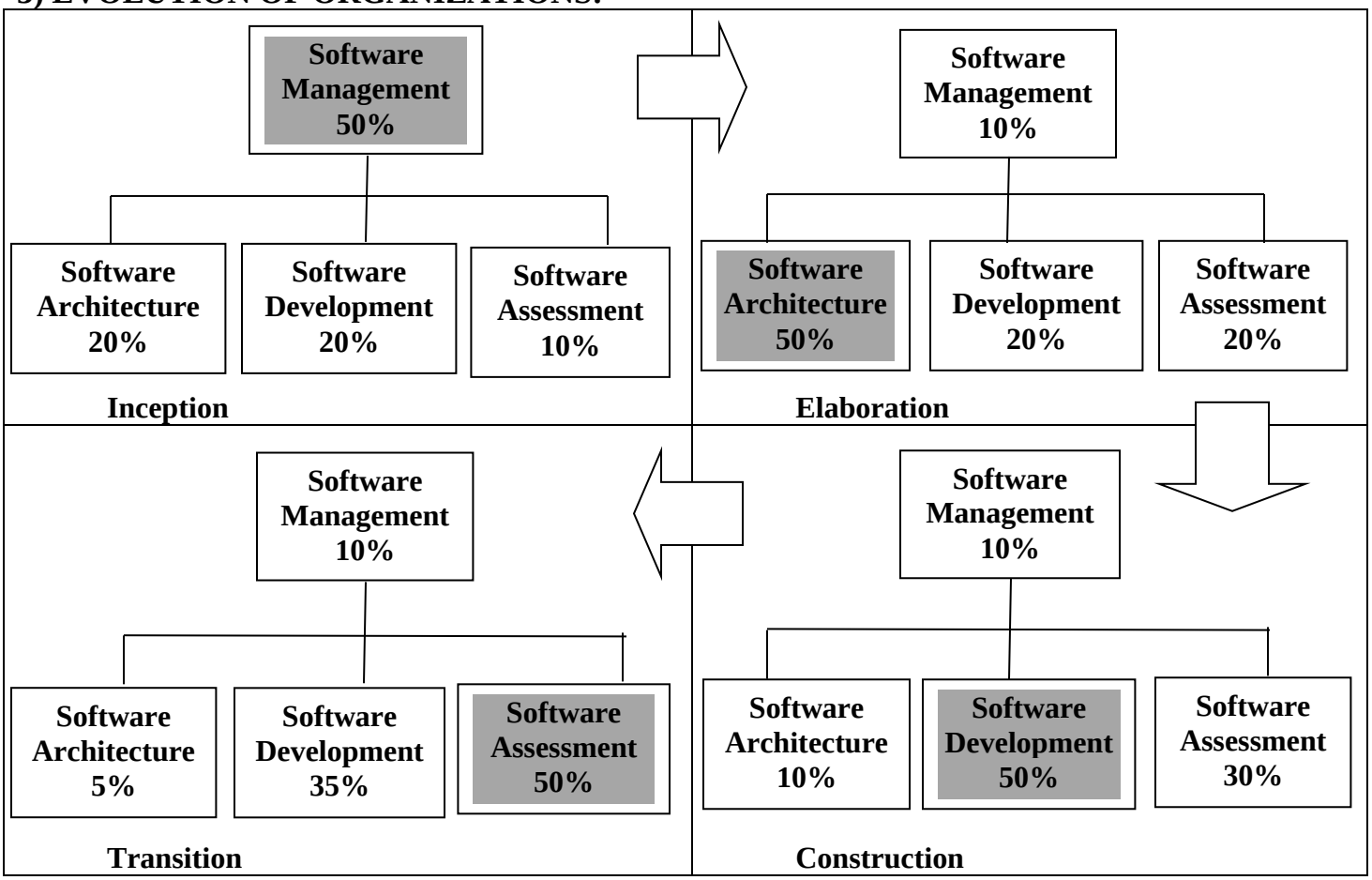


Figure 11-2. Default project organization and responsibilities

- 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.

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

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

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	**-Editors, Compilers, Debugger, Linker, Runtime**
Assessment	**-Test automation, defect Tracking**
Deployment	**defect Tracking**

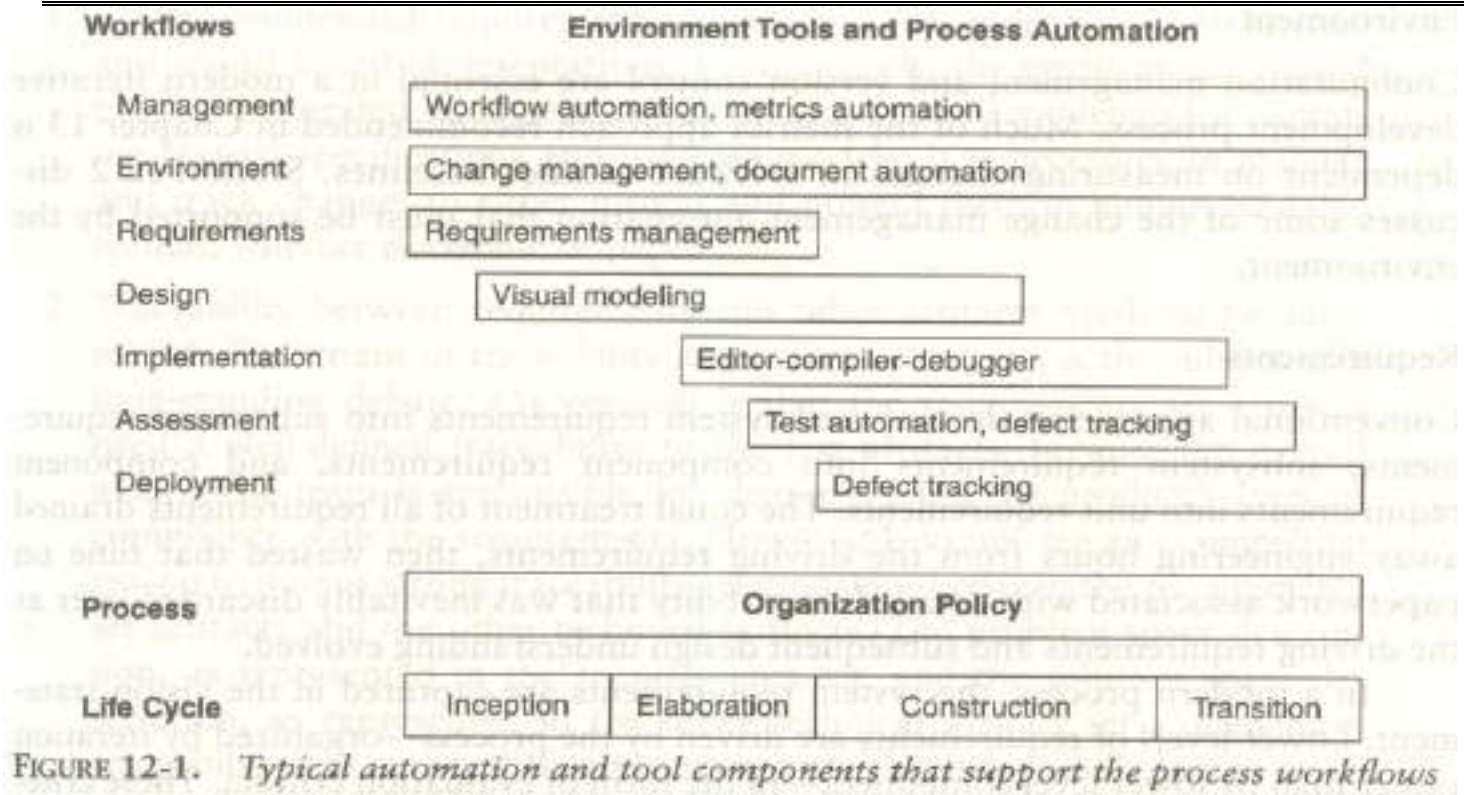


FIGURE 12-1. *Typical automation and tool components that support the process workflows*

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.

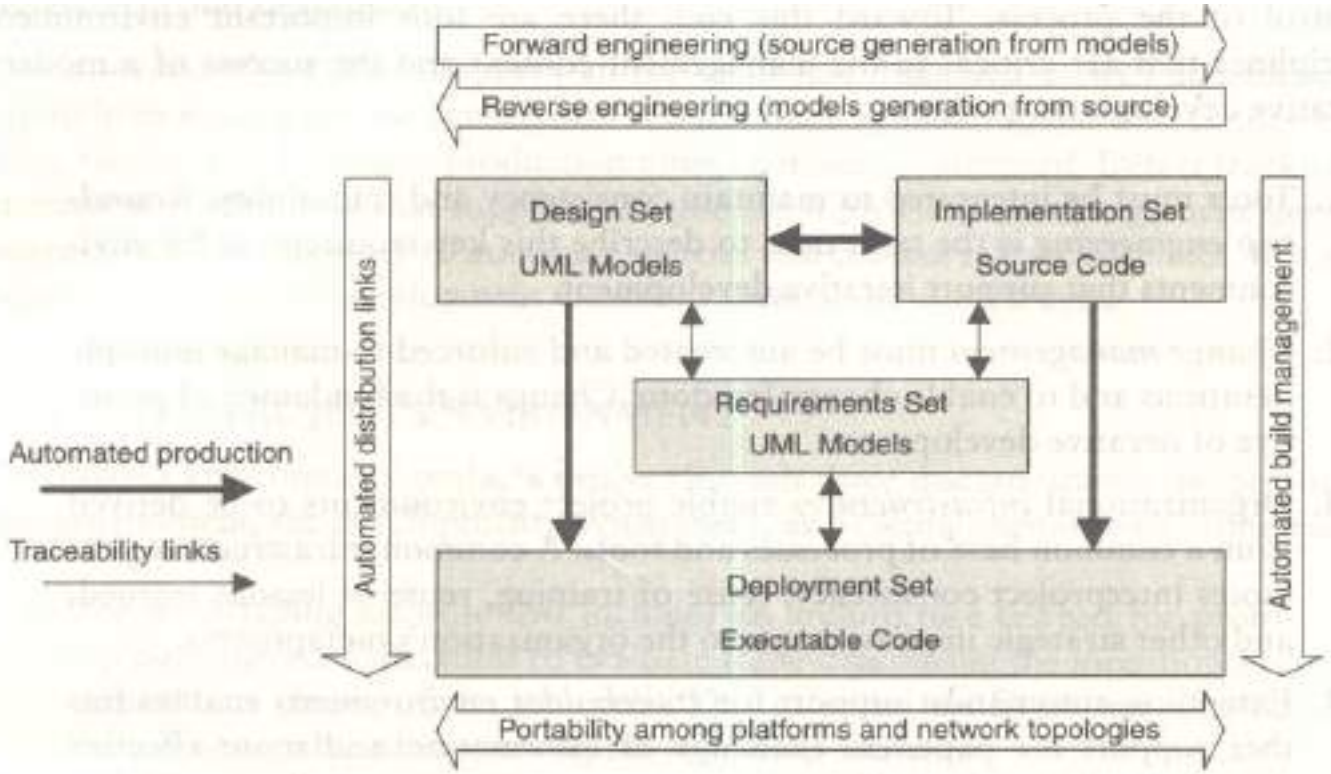


FIGURE 12-2. *Round-trip engineering*

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

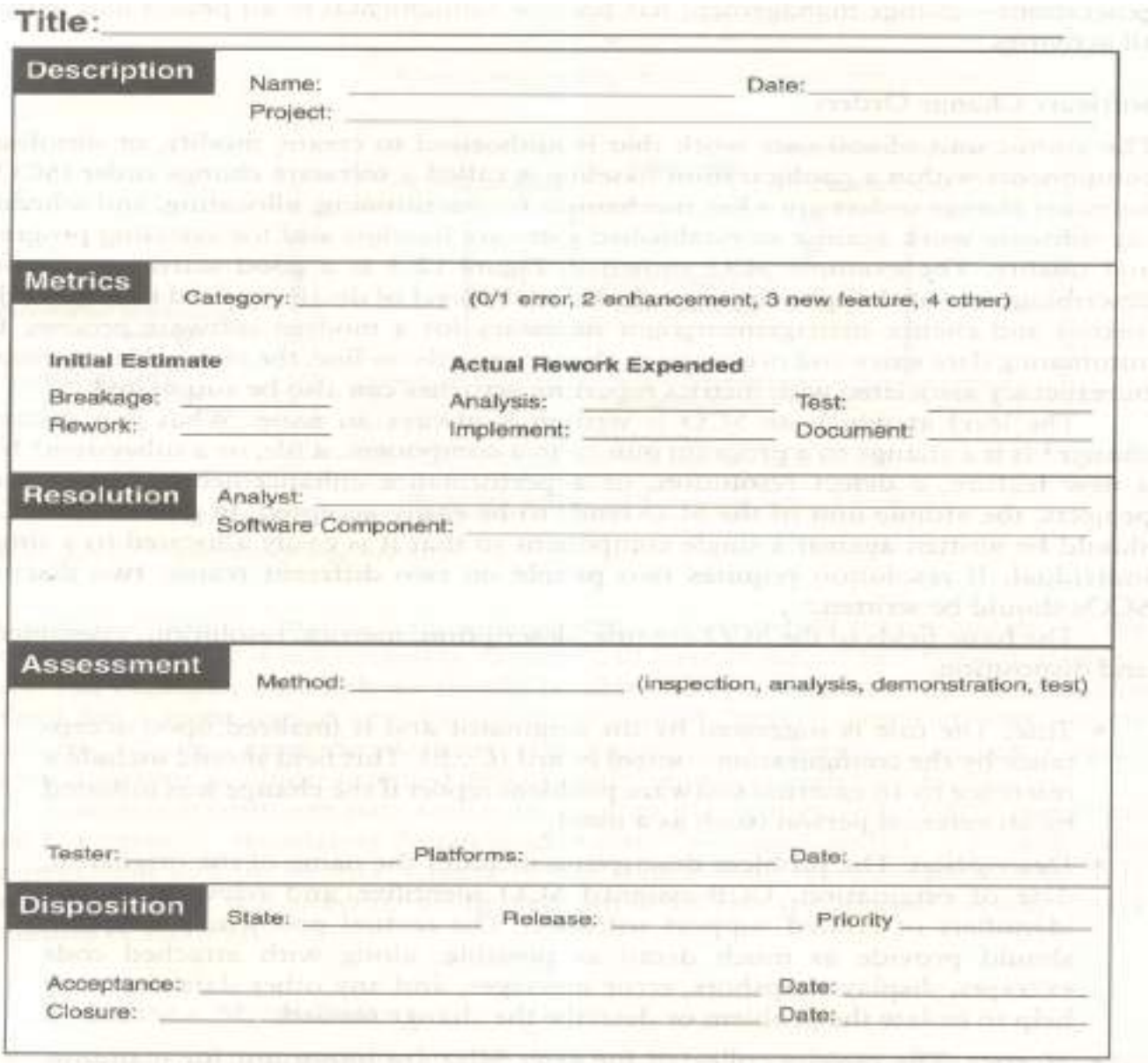


FIGURE 12-3. *The primitive components of a software change order*

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

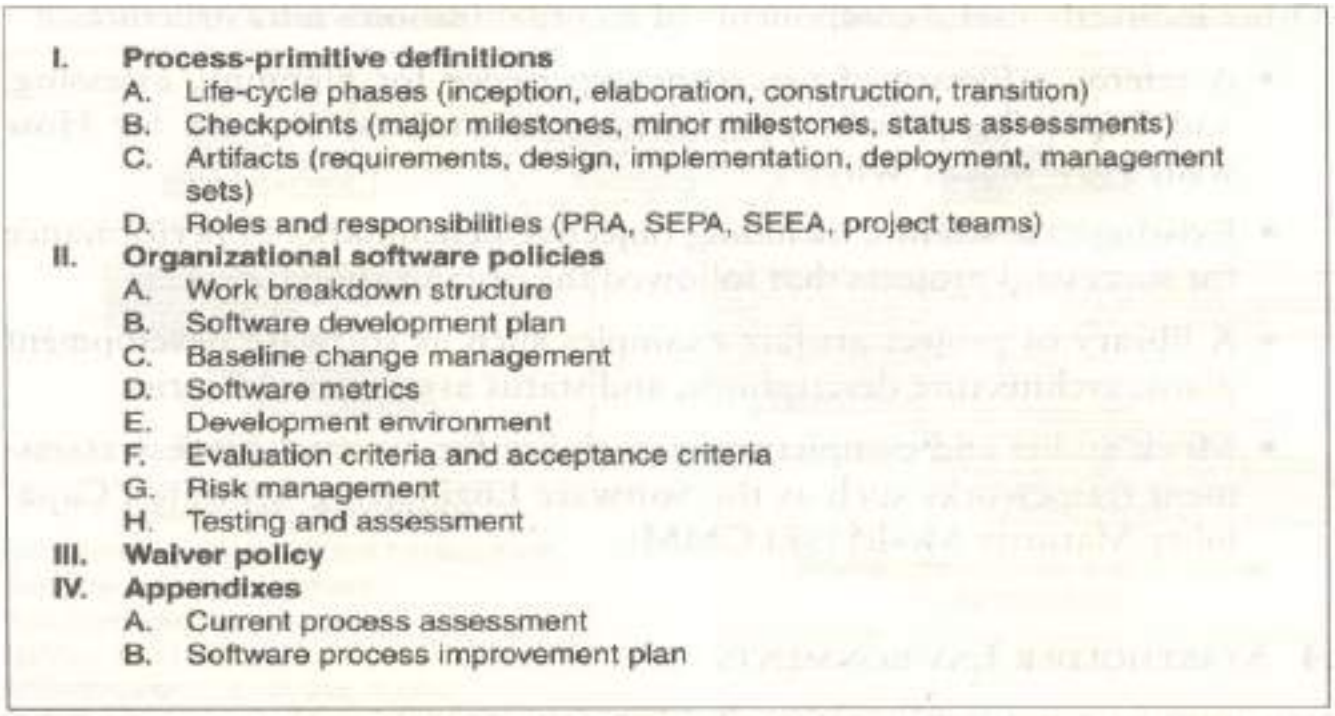
The organization infrastructure provides the organization's capital assets including two key artifacts - Policy & Environment

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



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

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

III. **Waiver policy**

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

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

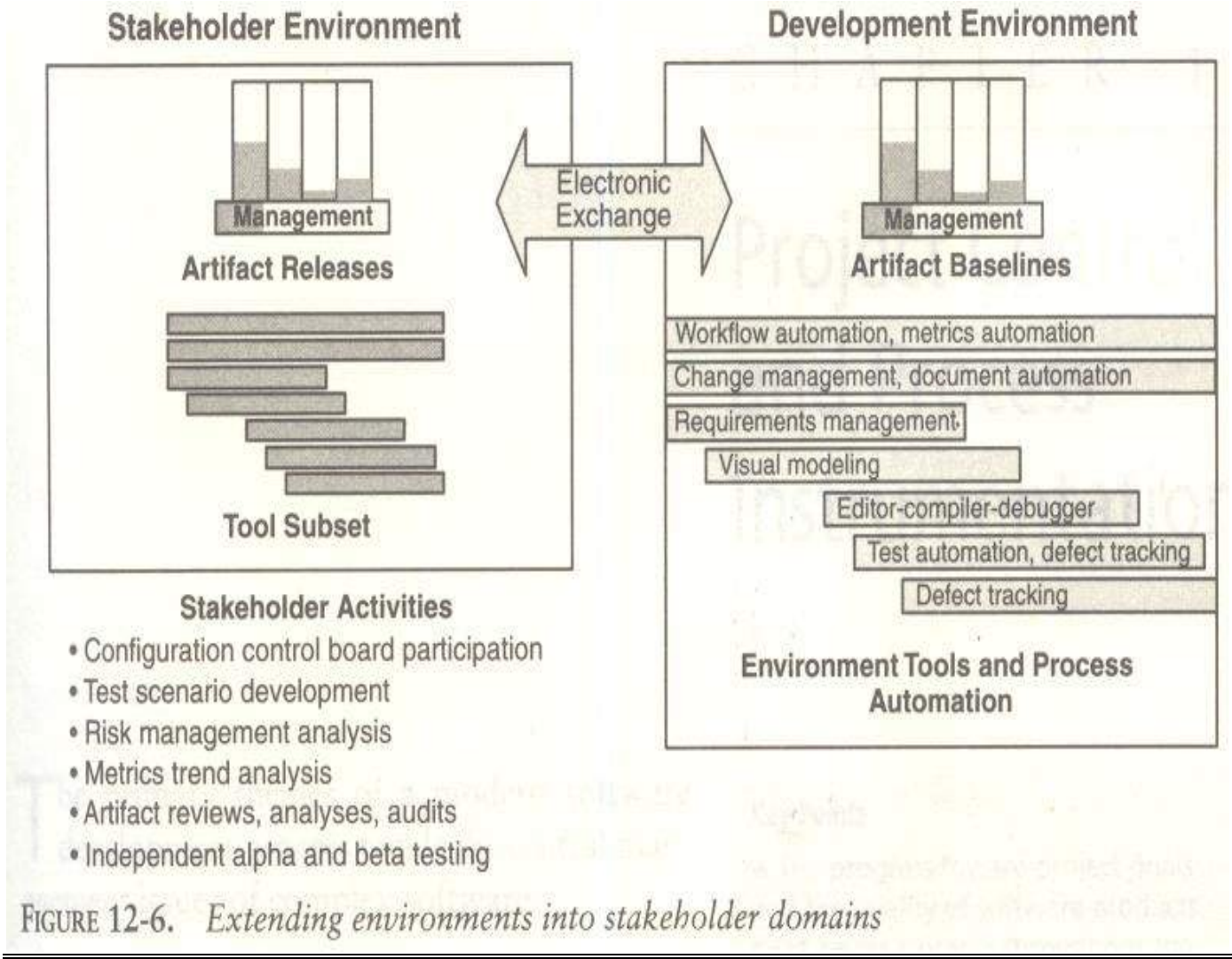


FIGURE 12-6. *Extending environments into stakeholder domains*

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.

1.1 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.

1.1.1 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.

1.1.2 Quality Indicators

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

1.2 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	**Quality Indicators**
□ Work and Progress	□ Change traffic and stability
□ Budgeted cost and expenditures	□ Breakage and modularity
□ Staffing and team dynamics	□ Rework and adaptability
	□ Mean time between failures (MTBF) and maturity

1.2.1 MANAGEMENT INDICATORS:

1.2.1.1 Work and progress

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

The default perspectives of this metric are:

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

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

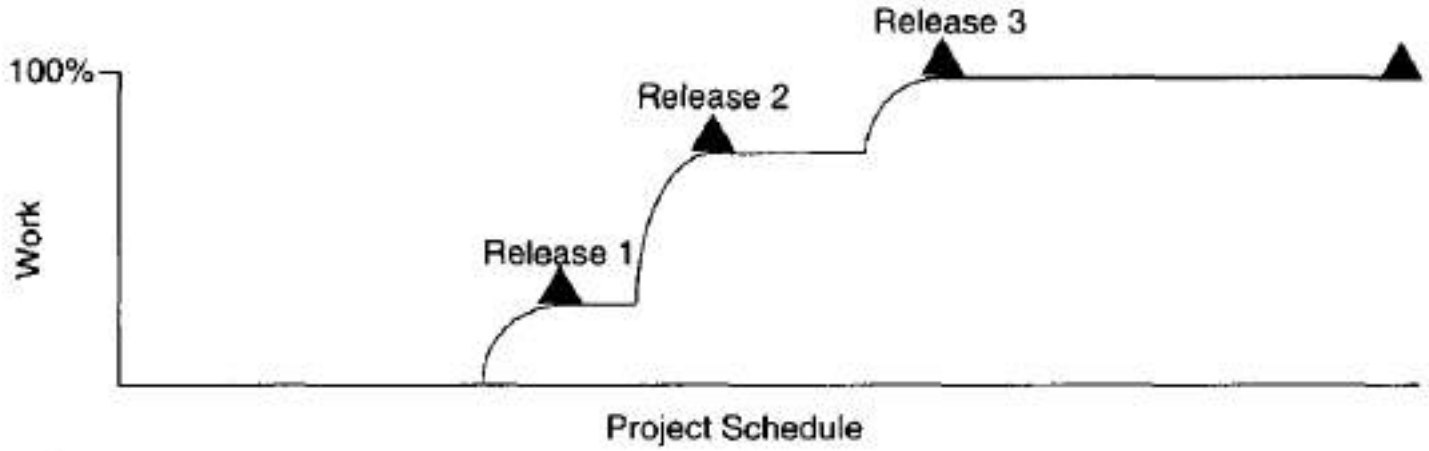


Fig: work and progress

1.2.1.2 Budgeted cost and expenditures

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

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

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

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

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

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

Assessment: Because most managers know exactly how much cost they have incurred and how much schedule they have used, the variability in making accurate assessments is centred 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.

1.2.1.3 Staffing and team dynamics

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

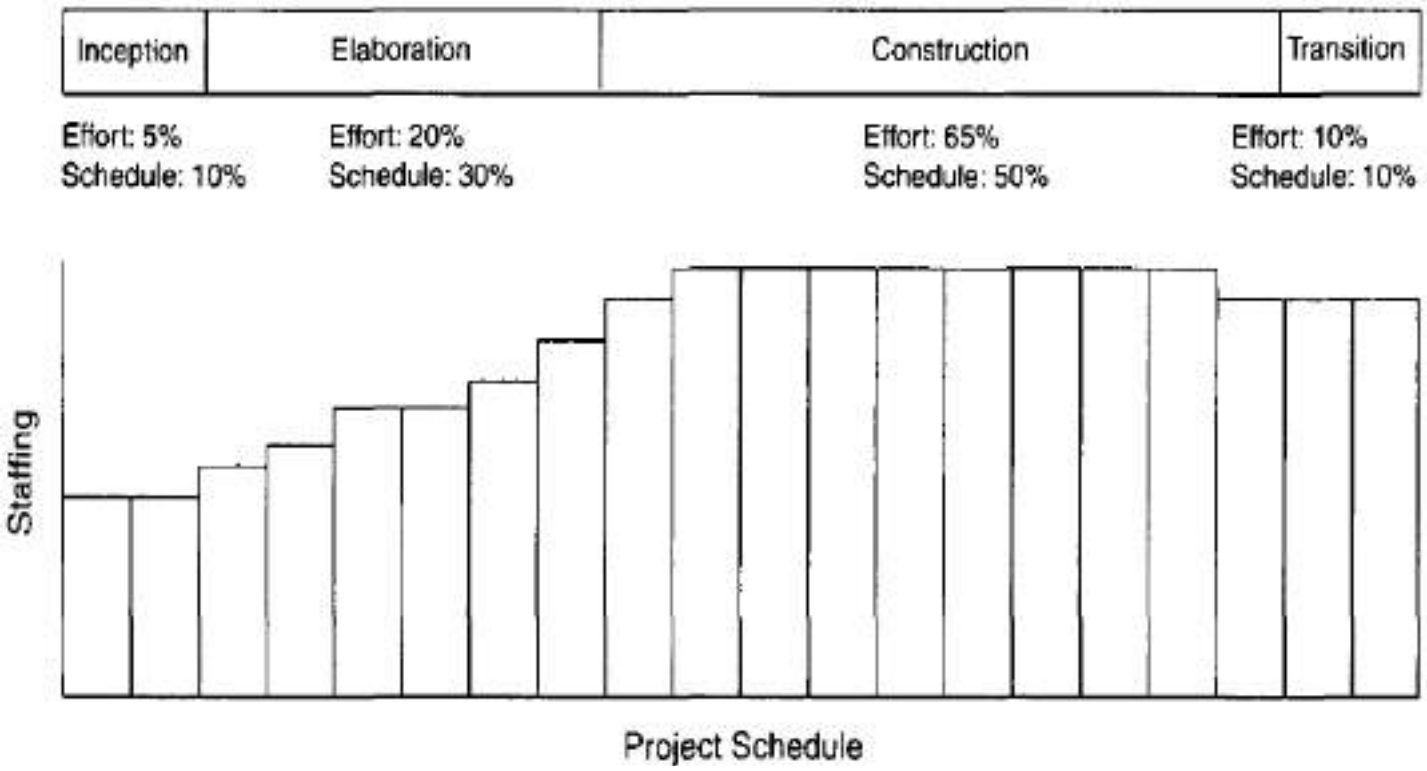


Fig: staffing and Team dynamics

1.2.2 QUALITY INDICATORS:

1.2.2.1 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

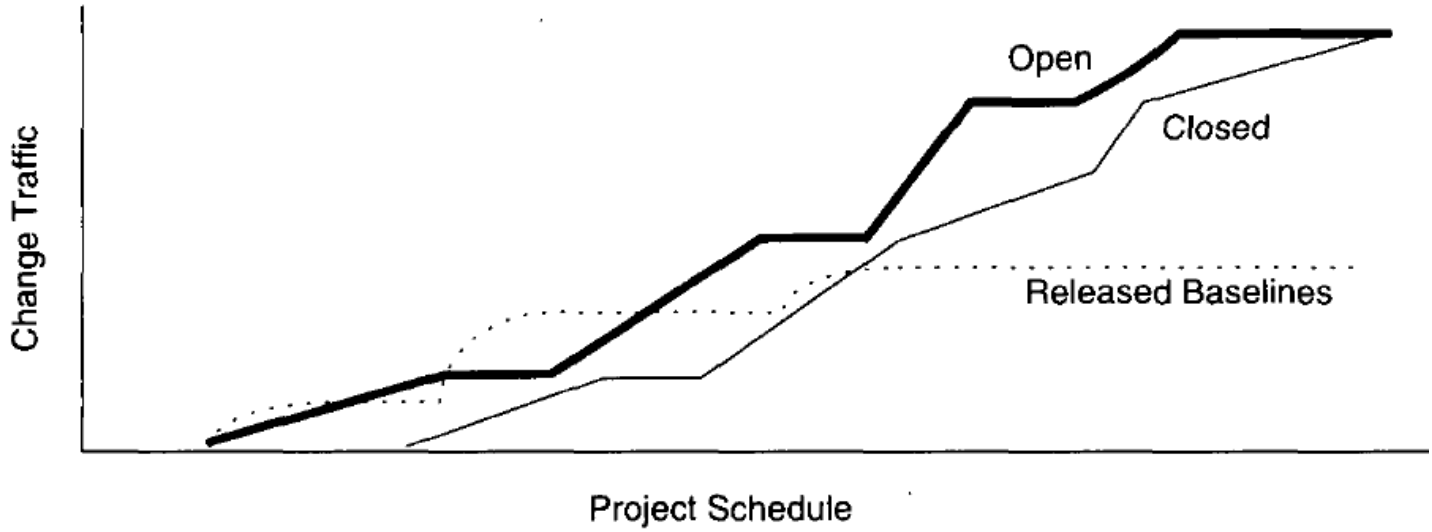


Fig: Change traffic and stability

1.2.2.2 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 revoke SLOC per change, by change type, by release, by components and by subsystems.

1.2.2.3 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 analyse, 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.

1.2.2.4 *MTBF* and Maturity:

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

1.3 LIFE -CYCLE EXPECTATIONS:

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

The quality indicators are derived from the evolving product rather than the artifacts.

They provide inside into the waste generated by the process. Scrap and rework metrics are a standard measurement perspective of most manufacturing processes.

They recognize the inherently dynamic nature of an iterative development process. Rather than focus on the value, they explicitly concentrate on the trends or changes with respect to time.

The combination of insight from the current and the current trend provides tangible indicators for management action.

Table 13-3. the default pattern of life cycle evolution

Metric	Inception	Elaboration	Construction	Transition
Progress	5%	25%	90%	100%
Architecture	30%	90%	100%	100%
Applications	<5%	20%	85%	100%
Expenditures	Low	Moderate	High	High
Effort	5%	25%	90%	100%
Schedule	10%	40%	90%	100%
Staffing	Small team	Ramp up	Steady	Varying
Stability	Volatile	Moderate	Moderate	Stable
Architecture	Volatile	Moderate	Stable	Stable
Applications	Volatile	Volatile	Moderate	Stable
Modularity	50%-100%	25%-50%	<25%	5%-10%
Architecture	>50%	>50%	<15%	<5%
Applications	>80%	>80%	<25%	<10%

Adaptability	Varying	Varying	Benign.	Benign
Architecture	Varying	Moderate	Benign	Benign
Applications	Varying	Varying	Moderate	Benign
Maturity	Prototype	Fragile	Usable	Robust
Architecture	Prototype	Usable	Robust	Robust
Applications	Prototype	Fragile	Usable	Robust

1.4 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.

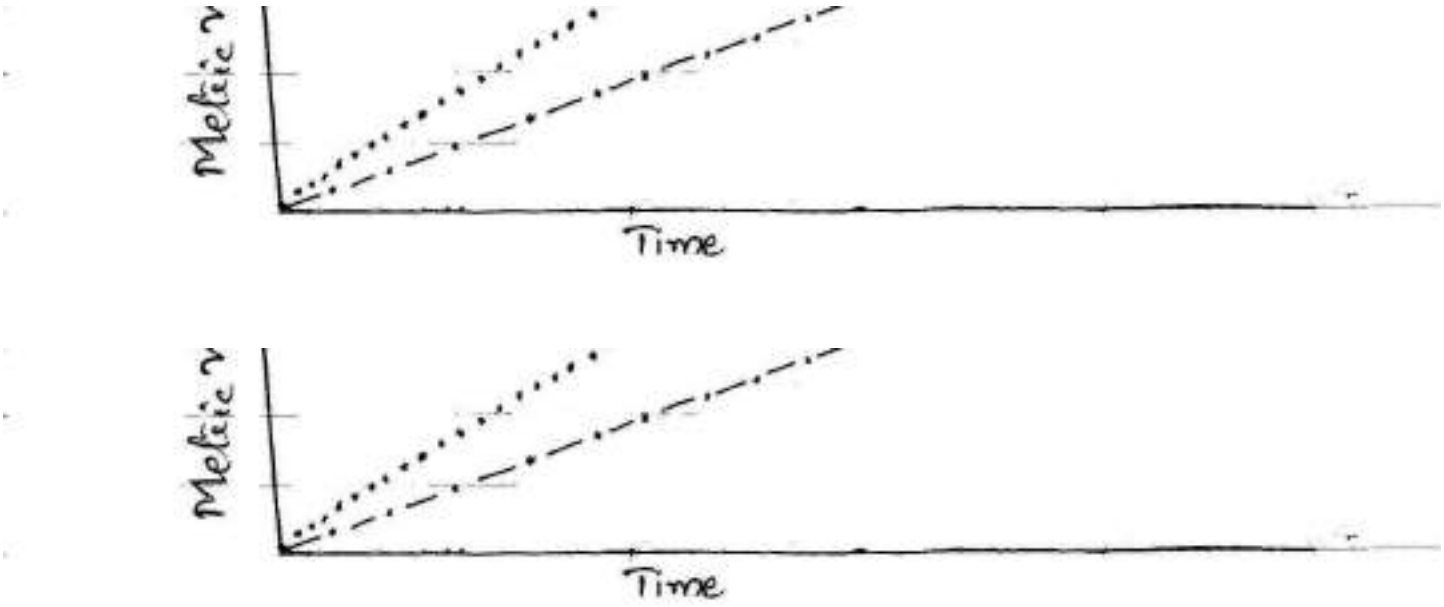
SPCP:

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

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



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

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

10 Mark Questions

1. Define metric. Discuss seven core metrics for project control and process instrumentation with suitable examples?
2. List out the three management indicators that can be used as core metrics on software projects. Briefly specify meaning of each?
3. Explain the various characteristics of good software metric. Discuss the metrics Automation using appropriate example?
4. Explain about the quality indicators that can be used as core metrics on software projects.
5. Explain Management Indicators with suitable example?
6. Define MTBF and Maturity. How these are related to each other?
7. Briefly explain about Quality Indicators?
8. Write short notes on Pragmatic software metrics?

WEB SERVICES

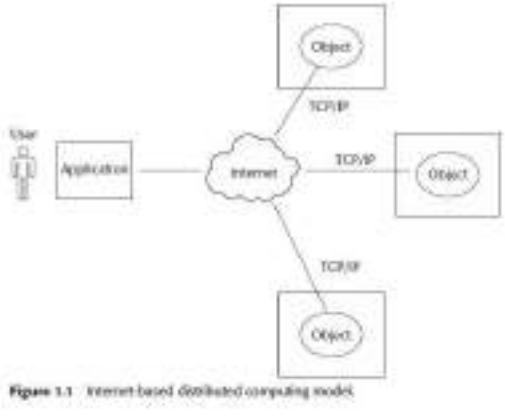
UNIT-1

Evolution of Distributed Computing:-

In the early years of computing, mainframe-based applications were considered to be the best-fit solution for executing large-scale data processing applications. With the advent of personal computers (PCs), the concept of software programs running on standalone machines became much more popular in terms of the cost of ownership and the ease of application use. With the number of PC-based application programs running on independent machines growing, the communications between such application programs became extremely complex and added a growing challenge in the aspect of application-to-application interaction. Lately, network computing gained importance, and enabling remote procedure calls (RPCs) over a network protocol called Transmission Control Protocol/Internet Protocol (TCP/IP) turned out to be a widely accepted way for application software communication. Since then, software application running on a variety of hardware platforms, operating systems, and different networks faced some challenges when required to communicate with each other and share data. This demanding requirement leads to the concept of distributed computing applications. As a definition, "Distributing Computing is a type of computing in which different components and objects comprising an application can be located on different computers connected to a network distributed computing model that provides an infrastructure enabling invocations of object functions located anywhere on the network. The objects are transparent to the application and provide processing power as if they were local to the application calling them.

Importance of Distributed Computing

The distributed computing environment provides many significant advantages compared to a traditional standalone application. The following are Some of those key advantages:



Higher performance. Applications can execute in parallel and distribute the load across multiple servers.

Collaboration. Multiple applications can be connected through standard distributed computing mechanisms.

Higher reliability and availability. Applications or servers can be clustered in multiple machines.

Scalability. This can be achieved by deploying these reusable distributed components on powerful servers.

Extensibility. This can be achieved through dynamic (re)configuration of applications that are distributed across the network. Higher productivity and lower development cycle time. By breaking up large problems into smaller ones, these individual components can be enveloped by smaller development teams in isolation.

components. Reduced cost. Because this World model provide satfeuse once developed **components** that are accessible over the network, significant cost educations can be achieved.

Reuse. The distributed components may perform various se vices that can potentially be used by multiple client applications. It saves repetitive development effort and improves interoperability between

Distributed computing also has changed the way traditional network programming is done by providing a shareable object like semantics across networks using programming languages like Java, C, and C++ . The following sections briefly discuss core distributed computing technologies such as Client/Server applications, OMG CORBA, Java RMI, Microsoft COM/DCOM, and MOM.

Client-Server Applications

The early years of distributed application architecture were dominated by two-tier business applications. In a two-tier architecture model, the first (upper) tier handles the presentation and business logic of the user application (client), and the second/lower tier handles the application organization and its data storage (server). This approach is commonly called client-server applications architecture. Generally, the server in a client/server application model is a database server that is mainly responsible for the organization and retrieval of data. The application client in this model handles most of the business processing and provides the graphical user interface of the application. It is a very popular design in business applications where the user.

interface and business logic are tightly coupled with a database server for handling data retrieval and processing.

For example, the client-server model has been widely used in enterprise resource planning (ERP), billing, and Inventory application systems where a number of client business applications residing in multiple desktop systems interact with a central database server.

Figure 1.2 shows an architectural model of a typical client server system in which multiple desktop-based business client applications access a central database server.

Some of the common limitations of the client-server application model are as follows:

■ Complex business processing at the client side demands robust client systems.

■ Security is more difficult to implement because the algorithms and logic reside on the client side making it more vulnerable to hacking.

■ Increased network bandwidth is needed to accommodate many calls to the server, which can impose scalability restrictions.

■ Maintenance and upgrades of client applications are extremely difficult because each client has to be maintained separately.

■ Client-server architecture suits mostly database-oriented standalone applications and does not target robust reusable component oriented applications.

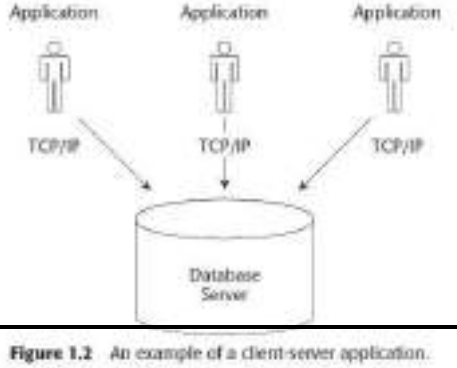


Figure 1.2 An example of a client-server application.

CORBA

The Common Object Request Broker Architecture (CORBA) is an industry wide, open standard initiative, developed by the Object Management Group (OMG) for enabling distributed computing that supports a wide range of application environments. OMG is a nonprofit consortium responsible for the production and maintenance of framework specifications for distributed and interoperable object-oriented systems.

CORBA differs from the traditional client/server model because it provides an object-oriented solution that does not enforce any proprietary protocols or any particular programming language, operating system, or hardware platform. By adopting CORBA, the applications can reside and run on any hardware platform located anywhere on the network, and can be written in any language that has mappings to a neutral interface definition called the Interface Definition Language (IDL). An IDL is a specific interface language designed to expose the services (methods/functions) of a CORBA remote object. CORBA also defines a collection of system-level services for handling low-level application services like life-cycle, persistence, transaction, naming, security, and so forth. Initially, CORBA 1.1 was focused on creating component level, portable object applications without interoperability. The introduction of CORBA 2.0 added interoperability between different ORB vendors by implementing an Internet Inter-ORB Protocol (IIOP). The IIOP defines the ORB backbone, through which other ORBs can bridge and provide interoperation with its associated services. In a CORBA-based solution, the Object Request Broker (ORB) is an object bus that provides a transparent mechanism for sending requests and receiving responses to and from objects, regardless of the environment and its location. The ORB intercepts the client's call and is responsible for finding its server object that implements the request, passes its parameters, invokes its method, and returns its results to the client. The ORB, as part of its implementation, provides interfaces to the CORBA services, which allows it to build custom-distributed application environments.

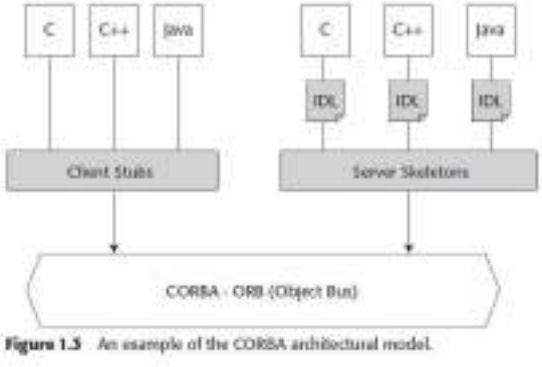


Figure 1.3 An example of the CORBA architectural model.

Figure 1.3 illustrates the architectural model of CORBA with an example representation of applications written in C, C++, and Java providing IDL bindings.

The CORBA architecture is composed of the following components:

IDL. CORBA uses IDL contracts to specify the application boundaries and to establish interfaces with its clients. The IDL provides a mechanism by which the distributed application component's interfaces, inherited classes, events, attributes, and exceptions can be specified.

ORB. It acts as the object bus or the bridge, providing the communication infrastructure to send and receive request/responses from the client and server. It establishes the foundation for the distributed application objects, achieving interoperability in a heterogeneous environment. Some of the distinct advantages of CORBA over a traditional client/server application model are as follows:

OS and programming-language independence. Interfaces between clients and servers are defined in OMG IDL, thus providing the following advantages to Internet programming: Multi-language and

multi-platform application environments, which provide a logical separation between interfaces and implementation.

Legacy and custom application integration. Using CORBA IDL, developers can encapsulate existing and custom applications as callable client applications and use them as objects on the ORB.

Rich distributed object infrastructure. CORBA offers developers a rich set of distributed object services, such as the Lifecycle, Events, Naming, Transactions, and Security services.

Location transparency. CORBA provides location transparency: An object reference is independent of the physical location and application level location. This allows developers to create CORBA-based systems where objects can be moved without modifying the underlying applications.

Java RMI

Java RMI was developed by Sun Microsystems as the standard mechanism to enable distributed Java objects-based application development using the Java environment. RMI provides a distributed Java application environment by calling remote Java objects and passing them as arguments or return values. It uses Java object serialization—a lightweight object persistence technique that allows the conversion of objects into streams. Before RMI, the only way to do inter-process communications in the Java platform was to use the standard Java network libraries. Though the java.net APIs provided sophisticated support for network functionalities, they were not intended to support or solve the distributed computing challenges.

Java RMI uses Java Remote Method Protocol (JRMP) as the inter process communication protocol, enabling Java objects living in different Java Virtual Machines (VMs) to

Transparently invoke one another's methods. Because these VMs can be running on different computers anywhere on the network, RMI enables object-oriented distributed computing. RMI also uses a reference-counting garbage collection mechanism that keeps track f external live object references to remote objects (live connections) using the virtual machine. When an object is found unreferenced, it is considered to be a weak reference and it will be garbage collected.

In RMI-based application architectures, a registry (rmiregistry) - oriented mechanism provides a simple non-persistent naming lookup service that is used to store the remote object references and to enable lookups from client applications. The RMI infrastructure based on the JRMP acts as the medium between the RMI clients and remote objects. It intercepts client requests, passes invocation arguments, delegates invocation requests to the RMI skeleton, and finally passes the return values of the method execution to the client stub. It also enables callbacks from server objects to client applications so that the asynchronous notifications can be achieved. Figure 1.4 depicts the architectural model of a Java RMI-based application solution.

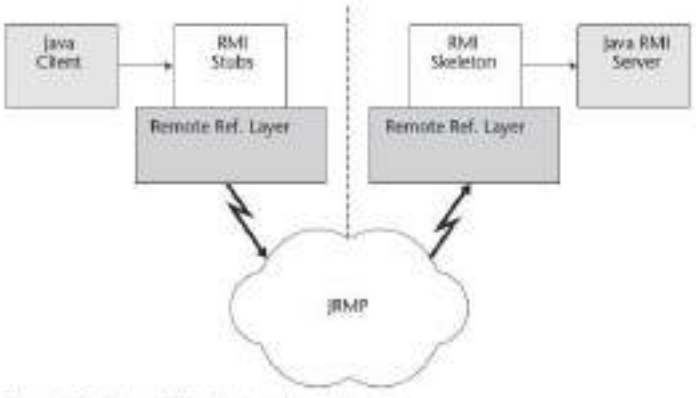


Figure 1.4 A Java RMI architectural model.

The java RMI architecture is composed of the following components:

RMI client. The RMI client, which can be a Java applet or a standalone application, performs the remote method invocations on a server object. It can pass arguments that are primitive data types or serializable objects.

RMI stub. The RMI stub is the client proxy generated by the rmi compiler (*rmic* provided along with Java developer kit—JDK) that encapsulates the network information of the server and performs the delegation of the method invocation to the server. The stub also marshals the method arguments and unmarshals the return values from the method execution.

RMI infrastructure. The RMI infrastructure consists of two layers: the remote reference layer and the transport layer. The remote reference layer separates out the specific remote reference behavior from the client stub. It handles certain reference semantics like connection entries, which are unicast/multicast of the invocation requests. The transport layer actually provides the networking infrastructure, which facilitates the actual data transfer during method invocations, the passing of formal arguments, and the return of back execution results. **RMI skeleton.** The RMI skeleton, which also is generated using the RMI compiler (rmic) receives the invocation requests from the stub and processes the arguments (unmarshalling) and delegates them to the RMI server. Upon successful method execution, it marshals the return values and then passes them back to the RMI stub via the RMI infrastructure.

RMI server. The server is the Java remote object that implements the exposed interfaces and executes the client requests. It receives incoming remote method invocations from the respective skeleton, which

passes the parameters after unmarshalling. Upon successful method execution, return values are sent back to the skeleton, which passes them back to the client via the RMI infrastructure.

Microsoft DCOM

The Microsoft Component Object Model (COM) provides a way for Windows-based software components to communicate with each other by defining a binary and network standard in a Windows operating environment. COM evolved from OLE (Object Linking and Embedding), which employed a Windows registry-based object organization mechanism. COM provides a distributed application model for ActiveX components. As a next step, Microsoft developed the Distributed Common Object

Model (DCOM) as its answer to the distributed computing problem in the Microsoft Windows platform. DCOM enables COM applications to communicate with each other using an RPC mechanism, which employs a DCOM protocol on the wire.

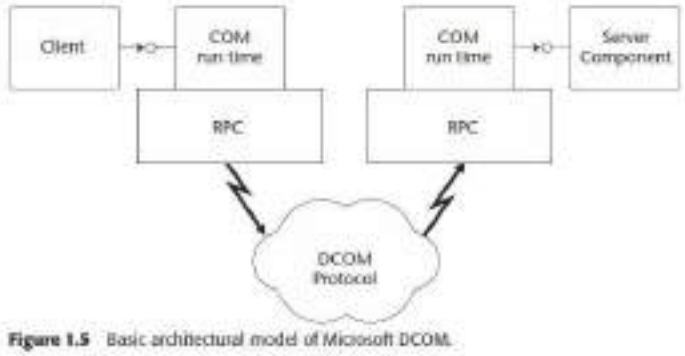


Figure 1.5 Basic architectural model of Microsoft DCOM.

Figure 1.5 shows an architectural model of DCOM. DCOM applies a skeleton and stub approach whereby a defined interface that exposes the methods of a COM object can be invoked remotely over a network. The client application will invoke methods on such a remote COM object in the same fashion that it would with a local COM object. The stub encapsulates the network location information of the COM server object and acts as a proxy on the client side. The servers can potentially host multiple COM objects, and when they register themselves against a registry, they become available for all the clients, who then discover them using a lookup mechanism.

DCOM is quite successful in providing distributed computing support on the Windows platform. But, it is limited to Microsoft application environments. The following are some of the common limitations of DCOM:

- Platform lock-in
- State management
- Scalability
- Complex session management issues

Message-Oriented Middleware

Although CORBA, RMI, and DCOM differ in their basic architecture and approach, they adopted a tightly coupled mechanism of a synchronous communication model (request/response). All these technologies are based upon binary communication protocols and adopt tight integration across their logical tiers, which is susceptible to scalability issues. Message-Oriented Middleware (MOM) is based upon a loosely coupled asynchronous communication model where the application client does not

need to know its application recipients or its method arguments. MOM enables applications to communicate indirectly using a messaging provider queue. The application client sends messages to the message queue (a message holding

area), and the receiving application picks up the message from the queue. In this operation model, the application sending messages to another application continues to operate without waiting for the response from that application.

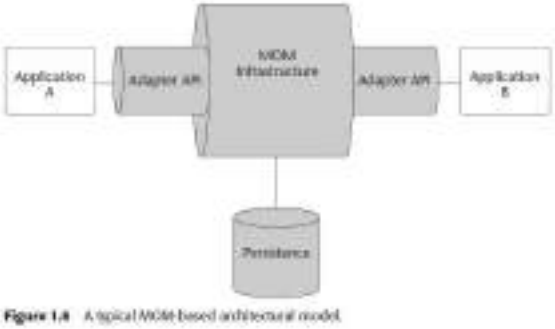


Figure 1.4 A typical MOM-based architectural model

MS provides Point-to-Point and Publish/Subscribe messaging models with the following features:

- Complete transactional capabilities
- Reliable message delivery
- Security

Some of the common challenges while implementing a MOM-based application environment have been the following:

- Most of the standard MOM implementations have provided native APIs for communication with their core infrastructure. This has affected the portability of applications across such implementations and has led to a specific vendor lock-in.
- The MOM messages used for integrating applications are usually based upon a proprietary message format without any standard compliance.

Challenges in Distributed Computing

Distributed computing technologies like CORBA, RMI, and DCOM have been quite successful in integrating applications within a homogenous environment inside a social area network. As the Internet becomes a logical solution that spans and connects the boundaries of businesses, it also demands the interoperability of applications across networks. This section discusses some of the common challenges noticed in the CORBA-, RMI-, and DCOM-based distributed computing solutions:

- Maintenance of various versions of stubs/skeletons in the client and server environments is extremely complex in a heterogeneous network environment.
- Quality of Service (QoS) goals like Scalability, Performance, and Availability in a distributed environment consume a major portion f the application's development time.
- Interoperability of applications implementing different protocols on heterogeneous platforms almost becomes impossible. For example, a DCOM client communicating to an RMI server or an RMI client communicating to a DCOM server.
- Most of these protocols are designed to work well within local networks. They are not very firewall friendly or able to be accessed over the Internet.

The Role of J2EE and XML in Distributed Computing

The emergence of the Internet has helped enterprise applications to be easily accessible over the Web without having specific client-side software installations. In the Internet-based enterprise application model, the focus was to move the complex business processing toward centralized servers in the back end. The first generation of Internet servers was based upon Web servers that hosted static Web pages and provided content to the clients via H P (Hyper ext Transfer Protocol). HTTP is a stateless protocol that connects Web browsers to Web servers, enabling the transportation of HTML content to the user.

With the high popularity and potential of this infrastructure, the push for a more dynamic technology was inevitable. This was the beginning of server-side scripting using technologies like CGI, NSAPI, and ISAPI. With many organizations moving their businesses to the Internet, a whole new category of business models like business-to-business (B2B) and business-to-consumer (B2C) came into existence.

This evolution lead to the specification of J2EE architecture, which promoted a much more efficient platform for hosting Web-based applications. J2EE provides a programming model based upon Web and business components that are managed by the J2EE application server.

The application server consists of many

APIs and low-level services available to the components. These low-level services provide security, transactions, connections and instance pooling, and concurrency services, which enable a J2EE developer to focus primarily on business logic rather than plumbing. The power of Java and its rich collection of APIs provided the perfect solution for developing highly transactional, highly available and scalable enterprise applications. Based on many standardized industry specifications, it provides the interfaces to connect

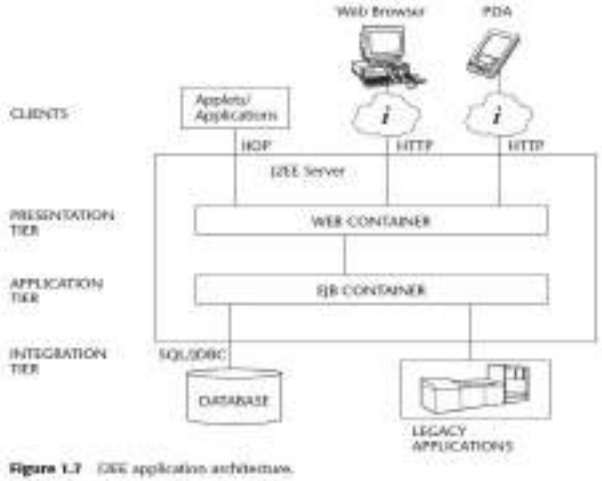


Figure 1.7 J2EE application architecture.

with various back-end legacy and information systems. J2EE also provides excellent client connectivity capabilities, ranging from PDA to Web browsers to Rich Clients (Applets, CORBA applications, and Standard Java Applications). Figure 1.7 shows various components of the J2EE architecture. A typical J2EE architecture is physically divided in to three logical tiers, which enables clear separation of the various application components with defined roles and responsibilities. The following is a breakdown of functionalities of those logical tiers:

Presentation tier. The Presentation tier is composed of Web components, which handle HTTP quests/responses, Session management, Device independent content delivery, and the invocation of business tier components.

Application tier. The Application tier (also known as the Business tier) deals with the core business logic processing, which may typically deal with workflow and automation. The business components

retrieve data from the information systems with well-defined APIs provided by the application server.

Integration tier. The Integration tier deals with connecting and communicating to back-end Enterprise Information Systems (EIS), database applications and legacy applications, or mainframe applications.

UNIT 2

Emergence of Web Services

Today, the adoption of the Internet and enabling Internet-based applications has created a world of discrete business applications, which co-exist in the same technology space but without interacting with each other. The increasing demands of the industry for enabling B2B, application-to application

(A2A), and inter-process application communication has led to a growing requirement for service-oriented architectures. Enabling service- oriented applications facilitates the exposure of business applications as service components enable business applications from other organizations

to link with these services for application interaction and data sharing without human intervention. By leveraging this architecture, it also enables interoperability between business applications and processes.

By adopting Web technologies, the service-oriented architecture model facilitates the delivery of services over the Internet by leveraging standard technologies such as XML. It uses platform-neutral standards by exposing the underlying application components and making them available to any application, any platform, or any device, and at any location. Today, this phenomenon is well adopted for implementation and is commonly referred to as Web services. Although this technique enables

communication between applications with the addition of service activation technologies and open technology standards, it can be leveraged to publish the services in a register of yellow pages available on the Internet. This will further redefine and transform the way businesses communicate over the Internet. This promising new technology sets the strategic vision of the next generation of virtual business models and the unlimited potential for organizations doing business collaboration and business process management over the Internet.

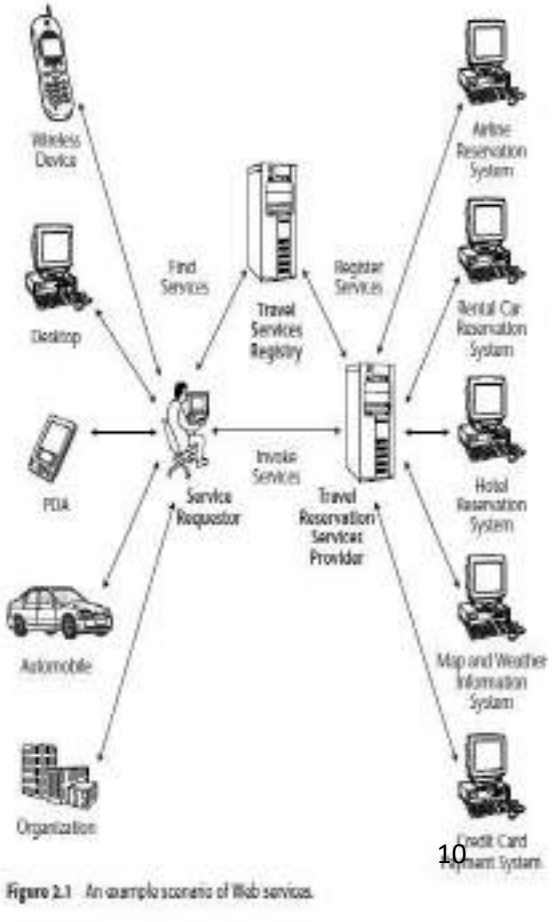


Figure 2.1 An example scenario of Web services.

What Are Web Services

Web services are based on the concept of service-oriented architecture (SOA). SOA is the latest evolution of distributed computing, which enables software components, including application functions, objects, and processes from different systems, to be exposed as services. According to Gartner research

(June 15, 2001), "Web services are loosely coupled software components delivered over Internet standard technologies." In short, Web services are self-describing and modular business applications that expose the business logic as services over the Internet through programmable interfaces and using Internet protocols for the purpose of providing ways to find, subscribe, and invoke those services. Based on XML standards, Web services can be developed as loosely coupled application components using any programming language, any protocol, or any platform. This facilitates delivering business applications as a service accessible to anyone, anytime, at any location, and using any platform. Consider the simple example shown in Figure 2.1 where a travel reservation services provider exposes its business applications as Web services supporting a variety of customers and application clients. These business applications are provided by different travel organizations residing at different networks and geographical locations.

Motivation and Characteristics

Web-based B2B communication has been around f r quite some time. These Web-based B2B solutions are usually based on custom and proprietary technologies and are meant for exchanging data and doing transactions over the Web. However, B2B has its own challenges. For example, in B2B communication, connecting new or existing applications and adding new business partners have always been a challenge. Due to this fact, in some cases the scalability of the underlying business applications is affected. Ideally, the business applications and information from a partner organization should be able to interact with the application of the potential partners seamlessly without redefining the system or its resources. To meet these challenges, it is clearly evident that there is a need for standard protocols and data formatting for enabling seamless and scalable B2B applications and services. Web services provide the solution to resolve these issues by adopting open standards. Figure 2.2 shows a typical B2B infrastructure (e-marketplace) using XML for encoding data between applications across the Internet.

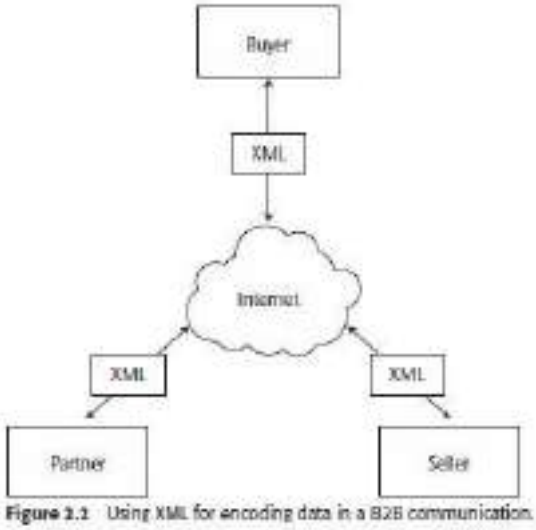


Figure 2.1 Using XML for encoding data in a B2B communication.

Web services enable businesses to communicate, collaborate, conduct business transactions using a lightweight infrastructure by adopting an XML-based data exchange format and industry standard delivery protocols.

The basic characteristics of a Web services application model are as follows:

- Web services are based on XML messaging, which means that the data exchanged between the Web service provider and the user are defined in XML.
- Web services provide a cross-platform integration of business applications over the Internet.
- To build Web services, developers can use any common programming language, such as Java, C, C++, Perl, Python, C#, and/or Visual Basic, and its existing application components.
- Web services are not meant for handling presentations like HTML context—it is developed to generate XML for uniform accessibility through any software application, any platform, or device.
- Because Web services are based on loosely coupled application components, each component is exposed as a service with its unique functionality.
- Web services use industry-standard protocols like HTTP, and they can be easily accessible through corporate firewalls.
- Web services can be used by many types of clients.
- Web services vary in functionality from a simple request to a complex business transaction involving multiple resources.
- All platforms including J2EE, CORBA, and Microsoft .NET provide extensive support for creating and deploying eb services.
- Web services are dynamically located and invoked from public and private registries based on industry standards such as UDDI and ebXML.

Why Use Web Services

Traditionally, Web applications enable interaction between an end user and a Web site, while Web services are service-oriented and enable application to- application communication over the Internet and easy accessibility to heterogeneous applications and devices. The following are the major technical reasons for choosing Web services over Web applications:

- Web services can be invoked through XML-based RPC mechanisms across firewalls.
- Web services provide a cross-platform, cross-language solution based on XML messaging.
- Web services facilitate ease of application integration using a lightweight infrastructure without affecting scalability.
- Web services enable interoperability among heterogeneous applications.

Web Services Architecture and Its Core Building Blocks

The basic principles behind the Web services architecture are based on SOA and the Internet protocols. It represents a composable application solution based on standards and

standards-based technologies. This ensures that the implementations of Web services applications are compliant to standard

specifications, thus enabling interoperability with those compliant applications.

Some of the key design requirements of the Web services architecture are the following:

- To provide a universal interface and a consistent solution model to define the application as modular components, thus enabling them as exposable services
- To define a framework with a standards-based infrastructure mo el and protocols to support services-based applications over the Internet
- To address a variety of service delivery scenarios ranging from e-business (B2C), business-to-business (B2B), peer-to-peer (P2P), and enterprise application integration (EAI)-based application communication
- To enable distributable modular applications as a centralized and decentralized application environment that supports boundary-less application communication for inter-enterprise and intra-enterprise application connectivity
- To enable the publishing of services to one or more public or private directories, thus enabling potential users to locate the published services using standard-based mechanisms that are defined by standards organizations
- To enable the invocation of those services when it is required, subject to authentication, authorization, and other security measures

Web Services Description Language (WSDL)

The Web Services Description Language, or WDDL, is an XML schema based specification for describing Web services as a collection of operations and data input/output parameters as messages. WSDL also defines the communication model with a binding mechanism to attach any transport

protocol, data format, or structure to an abstract message, operation, or endpoint. Listing 3.2 shows a WSDL example that describes a Web service meant for obtaining a price of a book using a GetBookPrice operation.

```
<?xml version="1.0"?>
<definitions name="BookPrice"
targetNamespace="http://www.wiley.com/bookprice.wsdl
" xmlns:tns=http://www.wiley.com/bookprice.wsdl
```

Web Services Communication Models

In Web services architecture, depending upon the functional requirements, it is possible to implement the models with RPC-based synchronous or messaging-based synchronous/asynchronous communication models. These communication models need to be understood before Web services are designed and implemented.

RPC-Based Communication Model

The RPC-based communication model defines a request/response-based,synchronous communication. When the client sends a request, the client waits until a response is sent back from the server before continuing any operation. Typical to implementing CORBA or RMI communication, the RPC-based Web services are tightly coupled and are implemented with remote objects to the client application. Figure 3.3 represents an RPC-based communication model in Web services architecture. The clients have the capability to provide parameters in method calls to the Web service provider. Then, clients invoke the Web services by sending parameter values to the Web service provider that executes the required methods, and then sends back the return values. Additionally, using RPC based communication, both the service provider and requester can register and discover services, respectively.

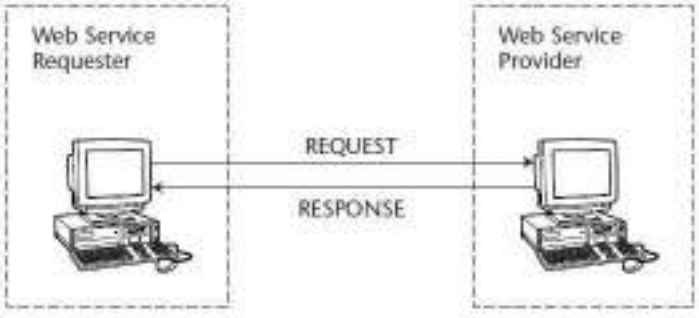


Figure 3.3 RPC-based communication model in Web services.

Implementing Web Services

The process of implementing Web services is quite similar to implementing any distributed application using CORBA or RMI. However, in web services, all the components are bound dynamically only at its runtime using standard protocols. Figure 3.5 illustrates the process highlights of implementing Web services. As illustrated in Figure 3.5, the basic steps of implementing Web services are as follows:

1.The service provider creates the Web service typically as SOAPbased service interfaces for exposed business applications. he provider then deploys them in a service container or using a

SOAP runtime environment, and then makes them available for invocation over a network. The service provider also describes the Web service as a WSDL-based service description, which defines the clients and the service container with a consistent

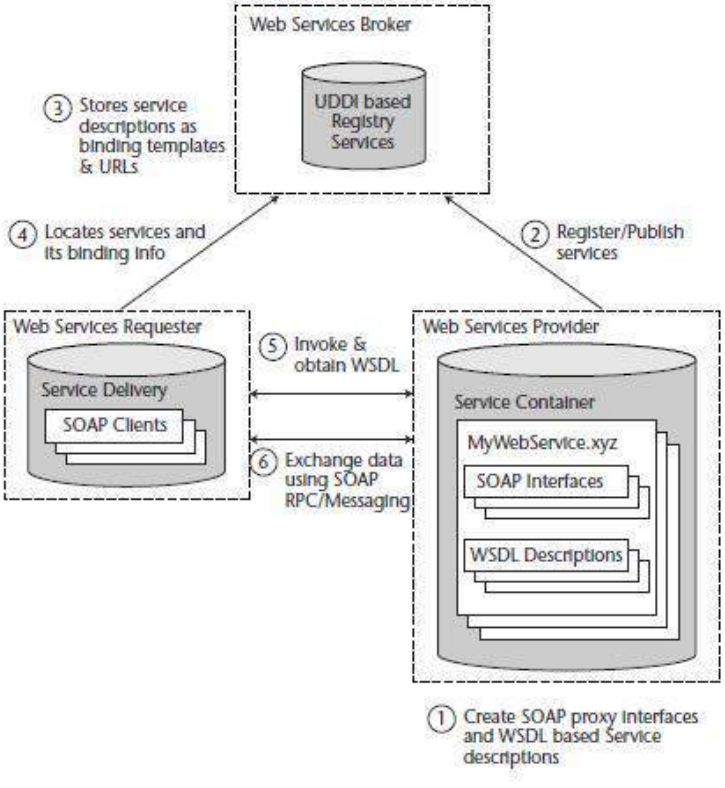


Figure 3.5 Process steps involved in implementing Web services.

way of identifying the service location, operations, and its communication model.

2. The service provider then registers the WSDL-based service description with a service broker, which is typically a UDDI registry.

3. The UDDI registry then stores the service description as binding templates and URLs to WSDLs located in the service provider environment.

4. The service requester then locates the required services by querying the UDDI registry. The service requester obtains the binding information and the URLs to identify the service provider.

5. Using the binding information, the service requester then invokes the service provider and then retrieves the WSDL Service description for those registered services. Then, the service requester creates

a client proxy application and establishes communication with the service provider using SOAP.

6. Finally, the service requester communicates with the service provider and exchanges data or messages by invoking the available services in the service container.

In the case of an ebXML-based environment, the steps just shown are the same, except ebXML registry and repository, ebXML Messaging, and ebXML CPP/CPA are used instead of UDDI, SOAP, and WSDL, respectively. The basic steps just shown also do not include the implementation of security and quality of service (QoS) tasks. Web Services Security." So far we have explored the Web services architecture and technologies. Let's now move forward to learn how to develop web services-enabled applications as services using the Web services architecture.

WSDL Limitations

There are some limitations to consider when using the WSDL-first approach and svcutil to create Contract files.

Declared Faults

When the WSDL contains declared faults:

•Specify the /UseSerializerForFaults argument during proxy code generation. For example:
svcutil /UseSerializerForFaults *.wsdl *.xsd.

If a port type of an operation includes Fault child node, the operation must use Document
•style.

•The fault part should refer to element but not type. For example:

Supported

```
<message name="SimpleTypeFault">
 <part name="SimpleTypeFault" element="ns2:StringFaultElement" />
</message>
```

The following is *incorrect* for faults:

Not Supported

```
<message name="SimpleTypeFault">
 <part name="SimpleTypeFault" type="xs:string" />
```

</message>

Removing OperationFormatStyle.Rpc Attribute

The OperationFormatStyle.Rpc attribute is not supported if the operation also has the fault contract attribute.

If the generated proxy code contains an attribute OperationFormatStyle.Rpc, then you must regenerate the WSDL from the code after deleting the attribute.

Identical part Elements

The part elements of messages cannot be same. If the elements are identical, svcutil throws an error. For example, this definition is allowed:

Supported

```
<message name="MultipartInputElement">
 <part name="Fortune" element="ns2:PersonDetailsElementsOne" />
 <part name="Person" element="ns2:PersonDetailsElementsTwo" />
</message>
```

This definition, where the parts refer to same element, is incorrect:

Not Supported

```
<message name="MultipartInputElement">
 <part name="Fortune" element="ns2:PersonNestedElements" />
 <part name="Person" element="ns2:PersonNestedElements" />
</message>
```

Mixed Type Messages

Mixed type messages are not supported. All message parts must refer to either element or type.

For example, the following definition is not permitted:

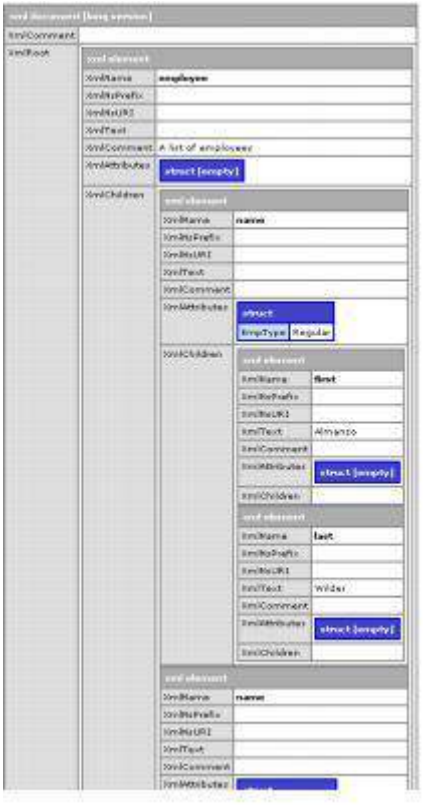
Not Supported

```
<message name="MultipartInputElement">
 <part name="Fortune" element="ns2:PersonDetailsElementsOne" />
 <part name="Person" type="xs:string" />
</message>
```

UNIT-3

XML document structures

An XML document object is a structure that contains a set of nested XML element structures. The following image shows a section of the cfdump tag output for the document object for the XML in A simple XML document. This image shows the long version of the dump, which provides complete details about the document object. Initially, ColdFusion displays a short version, with basic information. Click the dump header to change between short, long, and collapsed versions of the dump.



The following code displays this output. It assumes that y u save the code in a file under your web root, such as C:\Inetpub\wwwroot\testdocs\employeesimple.xml

```
<cffile action="read"
file="C:\Inetpub\wwwroot\testdocs\employeesimple.xml"
   variable="xmldoc">
<cfset mydoc = XmlParse(xmldoc)>
<cfdump var="#mydoc#">
```

The document object structure

At the top level, the XML document object has the following three entries:

Entry name	Type	Description
XmlRoot	Element	The root element of the document.
XmlComment	String	A string made of the concatenation of all comments on the document, that is, comments in the document prologue and epilog. This string does not include comments inside document elements.
XmlDocType	XmlNode	The DocType attribute of the document. This entry only exists if the document specifies a DocType. This value is read-only; you cannot set it after the document object has

		been created This entry does not appear when the cfdump tag displays an XML element structure.

The element structure

Each XML element has the following entries:

Entry name	Type	Description
XmlName	String	The name of the element; includes the namespace prefix.
XmlNsPrefix	String	The prefix of the namespace.
XmlNsURI	String	The URI of the namespace.
XmlText or XmlCdata	String	A string made of the concatenation f all text and CData text in the element, but n t inside any child elements. When you assign a value to the XmlCdata element, ColdFusion puts the text inside a CDATA information item. then you retrieve information from document object, these element names return identical values.
XmlComment	String	A string made of the concatenation of all comments inside the XML element, but not inside any child elements.
XmlAttributes	Structure	All of this element's attributes, as name-value pairs.
XmlChildren	Array	All this element's children elements.
XmlParent	XmlNode	The parent DOM node of this element. This entry does not appear when the cfdump tag displays an XML element structure.
XmlNodes	Array	An array of all the XmlNode DOM nodes contained in this element. This entry does not appear the cfdump tag when displays an XML element structure.

XML DOM node structure

The following table lists the contents of an XML DOM node structure:

Entry name	Type	Description
XmlName	String	The node name. For nodes such as Element or Attribute, the no e name is the element attribute name.
XmlType	String	The node XML DOM type, such as Element or Text.
XmlValue	String	The node value. This entry is used only for Attribute, CDATA, Comment, and Text type nodes.

Note: *The tag does not display XmlNode structures. If you try to dump an Xm Node structure, the cfdump tag displays "Empty Structure."*

The following table lists the contents of the XmlName and XmlValue fields for each node type that is valid in the XmlType entry. The node types correspond to the object types in the XML DOM hierarchy.

Node type	XmlName	xmlValue
CDATA	#cdata- section	Content of the CDATA section
COMMENT	#comment	Content of the comment
ELEMENT	Tag name	Empty string
ENTITYREF	Name of entity referenced	Empty string
PI (processing instruction)	Target entire content excluding the target	Empty string
TEXT	#text	Content of the text node
EN I Y	Entity name	Empty string
O A ION	Notation name	Empty string
DOCUME	#document	Empty string
FRAGME	#document-fragment	Empty string
DOCTYPE	Document type name	Empty string

Note:*AlthoughXMLattributesarenodes on the DOM tree, ColdFusion does not expose them as XML DOM node data structures. To view an element's attributes, use the element structure's XMLAttributes structure.*

The XML document object and all its elements are exposed as DOM node structures. For example, you can use the following variable names to reference nodes in the DOM tree that you created from the XML example in A simple XML document:

```
mydoc.XmlName
mydoc.XmlValue
mydoc.XmlRoot.XmlName
mydoc.employee.XmlType
mydoc.employee.XmlNodes[1].XmlType
```

XML namespace:-

XML namespaces are used for providing uniquely named elements and attributes in an XML document. They are defined in a W3C recommendation. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved.

A simple example would be to consider an XML instance that contained references to a customer and an ordered product. Both the customer element and the product element could have a child element named **id** . References to the **id** element would therefore be ambiguous; placing them in different namespaces would remove the ambiguity.

A namespace name is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organization defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code. It simply describes the XHTML namespace to human readers. Using a URI (such as "http://www.w3.org/1999/xhtml") to identify a namespace, rather than a simple string (such as "xhtml"), reduces the probability of different namespaces using duplicate identifiers.

Although the term namespace URI is widespread, the W3C Recommendation refers to it as the namespace name. The specification is not entirely prescriptive about the precise rules for namespace names (it does not explicitly say that parsers must reject documents where the namespace name is not a valid Uniform Resource Identifier), and many XML parsers

The XML document object and all its elements are exposed as DOM node structures. For example, you can use the following variable names to reference nodes in the DOM tree that you created from the XML example in A simple XML document:

```
mydoc.XmlName
mydoc.XmlValue
mydoc.XmlRoot.XmlName
mydoc.employee.XmlType
mydoc.employee.XmlNodes[1].XmlType
```

XML namespace:-

XML namespaces are used for providing uniquely named elements and attributes in an XML document. They are defined in a W3C recommendation. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved.

A simple example would be to consider an XML instance that contained references to a customer and an ordered product. Both the customer element and the product element could have a child element named **id** . References to the **id** element would therefore be ambiguous; placing them in different namespaces would remove the ambiguity.

A namespace name is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organization defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code. It simply describes the XHTML namespace to human readers. Using a URI (such as "http://www.w3.org/1999/xhtml") to identify a namespace, rather than a simple string (such as "xhtml"), reduces the probability of different namespaces using duplicate identifiers.

Although the term namespace URI is widespread, the W3C Recommendation refers to it as the namespace name. The specification is not entirely prescriptive about the precise rules for namespace names (it does not explicitly say that parsers must reject documents where the namespace name is not a valid Uniform Resource Identifier), and many XML parsers

allow any character string to be used. In version 1.1 of the recommendation, the namespace name becomes an Internationalized Resource Identifier, which licenses the use of non-ASCII characters that in practice were already accepted by nearly all XML software. The term namespace URI persists, however, not only in popular usage, but also in many other specifications from W3C and elsewhere World.

Following publication of the Namespaces recommendation, there was an intensive elaborate about how a relative URI should be handled, with some intensely arguing that it should simply be treated as a character string, and others arguing with conviction that it should be

turned into an absolute URI by resolving it against the base URI of the document The result of the debate was a ruling from W3C that relative URIs we e deprecated

The use of URIs taking the form of URLs in the http scheme (such as http://www.w3.org/1999/xhtml) is common, despite the absence of any formal relationship with the HTTP protocol. The Namespaces specification does not say what should happen if such a URL is dereferenced (that is, if software attempts to retrieve a document from this location). One convention adapted by s me users is to place an RDDL document at the location. In general, however, users should assume that the namespace URI is simply a name, not the address of a document n the Web.

The Emergence of SOAP

SOAP initially was developed by Develop Mentor, Inc., as a platform independent protocol for accessing services, objects between applications, and servers using HTTP-based communication. SOAP used an XML - based vocabulary for representing RPC calls and its parameters and return values. In 1999, the SOAP 1.0 specification was made publicly available as a joint effort supported by vendors like ogue Wave, IONA, Object Space, Digital Creations, UserLand, Microsoft, and DevelopMentor. Later, the SOAP 1.1 specification was released as a W3C Note, with additional contributions from IBM and the Lotus Corporation supporting a wide range of systems and communication models like RPC and messaging.

Nowadays, the current version of SOAP 1.2 is part of the W3C XML Protocol Working Group effort led by vendors such as Sun Microsystems, IBM, HP, BEA, Microsoft, and Oracle. At the time of this book's writing, SOAP 1.2 is available as a public W3C working draft. To find out the current status of the SOAP specifications produced by the XML Protocol Working Group, refer to the W3C Web site at www.w3c.org.

The SOAP 1.1 specifications define the following:

■■ Syntax and semantics for representing XML documents as structured SOAP messages

■■ Encoding standards for representing data in SOAP messages

■■ A communication model for exchanging SOAP messages

■■ Bindings for the underlying transport protocols such as SOAP transport ■■

Conventions for sending and receiving messages using RPC and messaging

Note that SOAP is not a programming language a business application component for building business applications. SOAP is intended for use as a portable communication protocol to deliver SOAP messages, which have to be created and processed by an application. In general, SOAP is simple and extensible by design, but unlike other distributed computing protocols, the following features are n t supported by SOAP:

■ Garbage collection

■ Object by reference

■ Object activation

■ Message batching

SOAP and ebXML are complementary to each other. In fact, SOAP is leveraged by an ebXML Messaging service as a communication protocol with an extension that provides added security and reliability for handling business transactions in e-business and B2B frameworks. More importantly, SOAP adopts XML syntax and standards like XML Schema and namespaces as part of its message structure. To understand the concepts of XML notations, XML Schema, and namespaces, refer to Chapter 8, “XML Processing and Data Binding with Java APIs.” Now, let’s take a closer look at the SOAP messages, standards, conventions, and other related technologies, and how they are represented in a development process.

Structure of SOAP messages:-

Usually a SOAP message requires defining two basic namespaces: SOAP

Envelope and SOAP Encoding. The following list their forms in both versions 1.1 and 1.2 of SOAP.

SOAP ENVELOPE

- http://schemas.xmlsoap.org/soap/envelope/ (SOAP 1.1)
- http://www.w3.org/2001/06/soap-envelope (SOAP 1.2)

SOAP ENCODING

- http://schemas.xmlsoap.org/soap/encoding/ (SOAP 1.1)
- http://www.w3.org/2001/06/soap-encoding (SOAP 1.2)

Additionally, SOAP also can use attributes and values defined in W3C XML Schema instances or XML Schemas and can use the elements based on custom XML conforming to W3C XML Schema specifications. SOAP does not support or use DTD-based element or attribute declarations. To

understand the fundamentals of XML namespaces, refer to Chapter 8, "XML Processing and Data Binding with Java APIs." Typical to the previous example message, the structural format of a

SOAP message (as per SOAP version 1.1 with attachments) contains the following elements:

- Envelope
- Header (optional)
- Body
- Attachments (optional)

Figure 4.1 represents the structure of a SOAP message with attachments.Typically, a SOAP message is represented by a SOAP envelope with zero or more attachments. The SOAP message envelope contains the header and body of the message, and the SOAP message attachments enable the message to contain data, which include XML and non-XML data

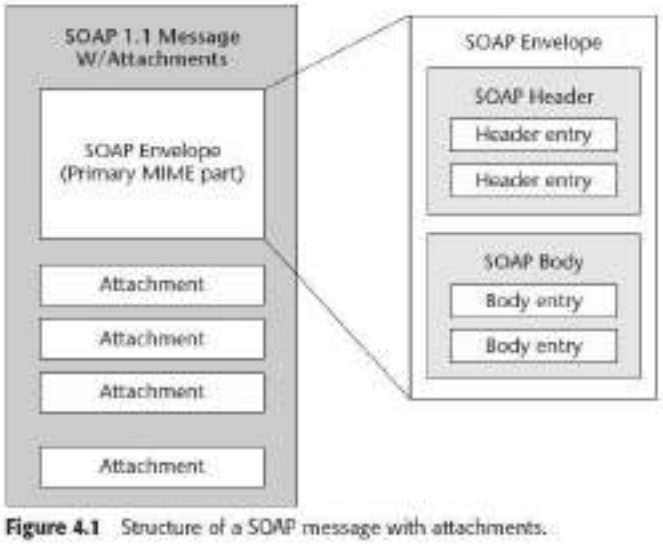
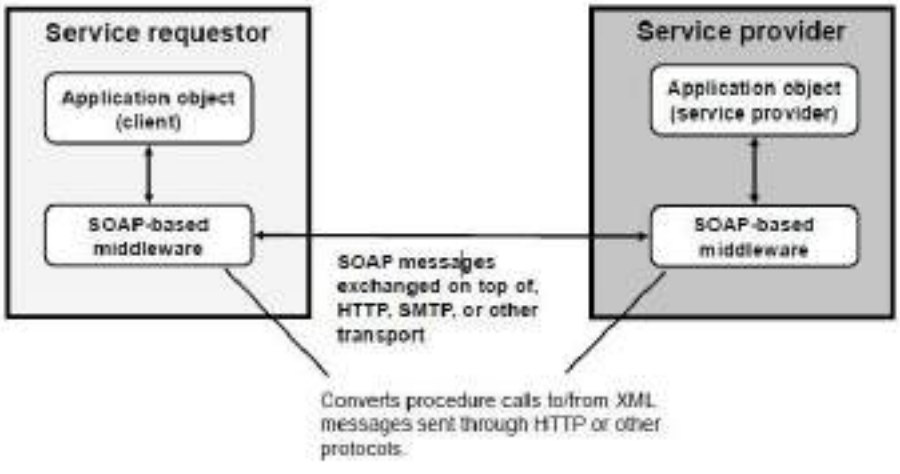


Figure 4.1 Structure of a SOAP message with attachments.

(like text/binary files). In fact, a SOAP message package is constructed

using the MIME Multipart/Related structure approaches to separate and identify the different parts of the message. Now, let's explore the details and characteristics of the parts of a SOAP message.



What is SOAP:-

SOAP is the standard messaging protocol used by Web services. SOAP's primary application is inter application communication. SOAP codifies the use of XML as an encoding scheme for request and response parameters using HTTP as a means for transport.

SOAP covers the following four main areas:

– A message format for one-way communication describing how a message can be packed into an XML document.

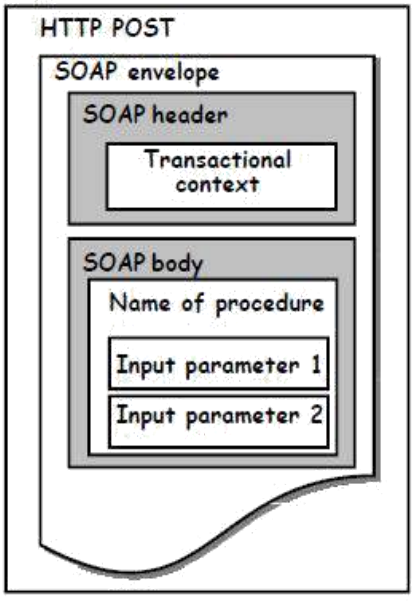
– A description of how a SOAP message should be transported using HTTP (for Web-based interaction) or SMTP (for e-mail-based interaction).

– A set of rules that must be followed when processing a SOAP message and a simple classification of the entities involved in processing a SOAP message.

– A set of conventions on how to turn an RPC call into a SOAP message and back.

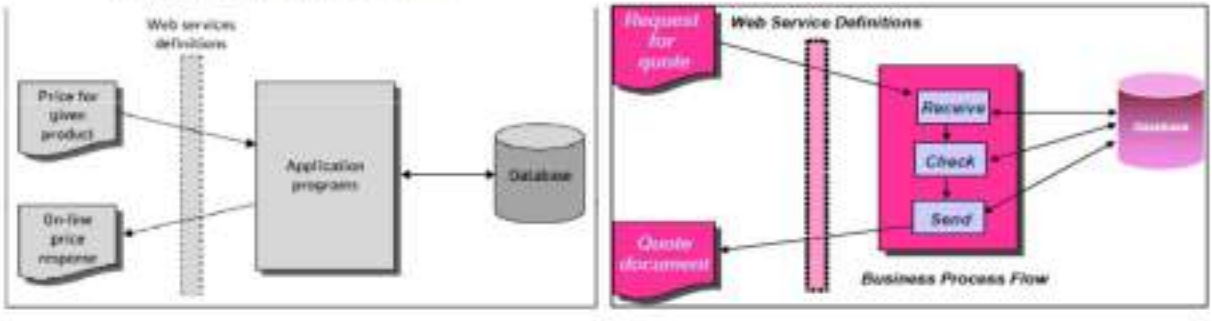
SOAP and HTTP

- A binding of SOAP to a transport protocol is a description of how a SOAP message is to be sent using that transport protocol.
- The typical binding for SOAP is HTTP.
- SOAP can use GET or POST. With GET, the request is not a SOAP message but the response is a SOAP message, with POST both request and response are SOAP messages (in version 1.2, version 1.1 mainly considers the use of POST).
- SOAP uses the same error and status codes as those used in HTTP so that HTTP responses can be directly interpreted by a SOAP module.



The SOAP Communication Model

- SOAP communication model is defined by its encoding style and its communication style
- Encoding style are about how applications on different platforms share and exchange data, although they may have different data types
- SOAP encoding rules are identified by the URI "http://www.w3.org/2003/05/soap-encoding"
- SOAP supports two possible communication styles:
  - remote procedure call (RPC) and
  - document (or message).



RPC-style (synchronous) interaction

Document-style (asynchronous) interaction

The SOAP Body

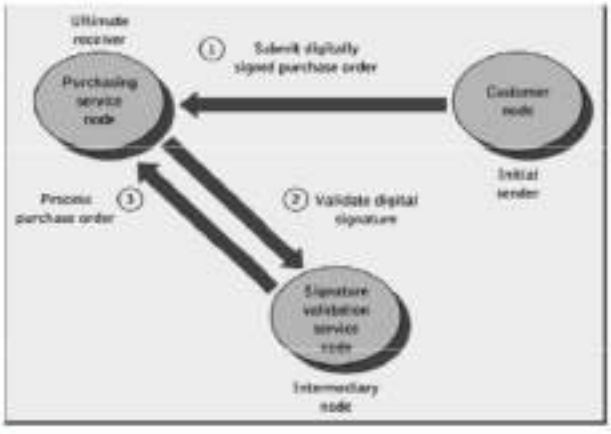
- The SOAP body is the area of the SOAP message, where the application specific XML data (payload) being exchanged in the message is placed.

- The **<Body> element** must be present and is an immediate child of the envelope. It may contain a number of child elements, called body entries, but it may also be empty. The **<Body> element** contains either of the following:
  - **Application-specific data** is the information that is exchanged with a Web service. The SOAP <Body> is where the method call information and its related arguments are encoded. It is where the response to a method call is placed, and where error information can be stored.
  - A **fault message** is used only when an error occurs.

- A SOAP message may carry either application-specific data or a fault, but not both.

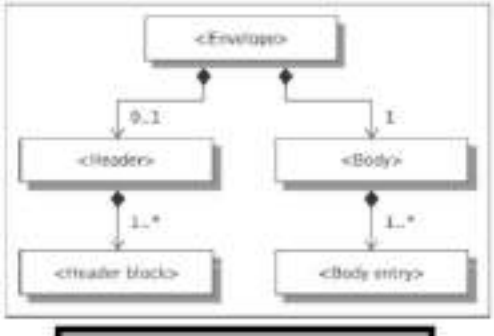
SOAP Intermediaries

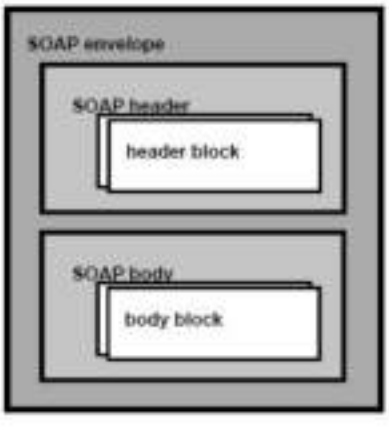
- SOAP headers have been designed in anticipation of participation of other SOAP processing nodes – called **SOAP *intermediaries*** – along a **message's path** from an initial SOAP sender to an ultimate SOAP receiver.
- A SOAP message travels along the message path from a sender to a receiver.
- All SOAP messages start with an initial sender, which creates the SOAP message, and end with an ultimate receiver.



SOAP messages

- SOAP is based on **message exchanges.**
- Messages are seen as **envelopes** where the application encloses the data to be sent.
- A SOAP message consists of a SOAP of an <Envelope> element containing an optional <Header> and a mandatory <Body> element.
- The contents of these elements are application defined and not a part of the SOAP specifications.
- A SOAP <Header> contains blocks of information relevant to how the message is to be processed. This helps pass information in SOAP messages that is not application payload.
- The SOAP <Body> is where the main end-to-end information conveyed in a SOAP message must be carried.





SOAP Envelope

The SOAP envelope is the primary container of a SOAP message's structure and is the mandatory element of a SOAP message. It is represented as the root element of the message as Envelope. As we discussed earlier, it is usually declared as an element using the XML namespace ttp://schemas .xmlsoap.org/soap/envelope/. As per SOAP 1.1 specifications, SOAP messages that do not follow this namespace declaration are not processed and are considered to be invalid. Encoding styles also can be defined using a namespace under Envelope to represent the data types used in the message. Listing 4.3 shows the SOAP envelope element in a SOAP message.

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:
encodingStyle="http://schemas.xmlsoap.org/soap/enc ding/"/>
<!--SOAP Header elements - -/>

<!--SOAP Body element - -/>
</SOAP-ENV:Envelope>
```

SOAP Header

The SOAP header is represented as the first immediate child element of a SOAP envelope, and it has to be namespace qualified. In addition, it also may contain zero or more optional child elements, which are referred to as SOAP header entries. The SOAP encoding Style attribute will be used to

define the encoding of the data types used in header element entries. The SOAP actor attribute and SOAP must Understand attribute can be used to indicate the target SOAP application node sender/Receiver/Intermediary) and to process the Header entries. Listing 4.4 shows the sample representation of a SOAP header element in a SOAP message.

```
<SOAP-E V:Header>
<wiley:Transaction
xmlns:wiley="http://jws.wiley.com/2002/booktx"
SOAP-E V:mustUnderstand="1">
<keyValue> 5 </keyValue>
</wiley:Transaction>
</SOAP-ENV:Header>
```

SOAP Body

A SOAP envelope contains a SOAP body as its child element, and it may contain one or more optional SOAP body block entries. The Body represents the mandatory processing information or the payload intended for the receiver of the message. The SOAP 1.1 specification mandates that there must be one or more optional SOAP Body entries in a message. A Body block of a SOAP message can contain any of the following:

- RPC method and its parameters
- Target application (receiver) specific data
- SOAP fault for reporting errors and status information

Listing 4.5 illustrates a SOAP body representing an RPC call for getting the book price information from www.wiley.com for the book name *Developing Java Web Services*.

```
<SOAP-ENV:Body>
<m:GetBookPrice
xmlns:m="http://www.wiley.com/jws.book.priceList/">
<bookname xsi:type='xsd:string'>
Developing Java Web services</bookname>
</m:GetBookPrice>
</SOAP-ENV:Body>
```

SOAP Encoding

SOAP 1.1 specifications stated that SOAP- based applications can represent their data either as literals or as encoded values defined by the "XML Schema, Part -2" specification (see ww.w3.org/TR/xmlschema-2/). Literals refer to message contents that are encoded according to the W3C XML Schema. Encoded values refer to the messages encoded based on SOAP encoding styles specified in SOAP Section 5 of the SOAP 1.1 specification. The namespace identifiers for these SOAP encoding styles are defined in http://schemas.xmlsoap.org/soap/encoding/(SOAP1.1)and http://www.w3.org/2001/06/soap-encoding (SOAP 1.2). The SOAP encoding defines a set of rules for expressing its data types. It is a generalized set of data types that are represented by the programming languages, databases, and semi-structured data required for an application. SOAP encoding also defines serialization rules for its data model using an encoding Style attribute under the SOAP-ENV namespace that specifies the serialization rules for a specific element or a group of elements. SOAP encoding supports both simple- and compound-type values.

SOAP Messaging

SOAP Messaging represents a loosely coupled communication model based on message notification and the exchange of XML documents. The SOAP message body is represented by XML documents or literals encoded according to a specific W3C XML schema, and it is produced and consumed by sending or receiving SOAP node(s). The SOAP sender node sends a message with an XML document as its body message and the SOAP receiver node

processes it.4.26 represents a SOAP message and a SOAP messaging-based communication. The message contains a header block Inventory Notice and the body product, both of which are application-defined and not defined by SOAP. The header contains information required by the receiver node and the body contains the actual message to be delivered.

Advantages and disadvantages of SOAP

- Advantages of SOAP are:
  - Simplicity: Based on highly-structured format of XML
  - Portability: No dependencies on underlying platform
  - Firewall friendliness: By posting data over HTTP
  - Use of open standards: text-based XML standard
  - Interoperability: Built on open technologies (XML and HTTP)
  - Universal acceptance. Most widely accepted message communication standard
- Disadvantages of SOAP are:
  - Too much reliance on HTTP: limited only to request/response model and HTTP's slow protocol causes bad performance
  - Statelessness: difficult for transactional and business processing applications
  - Serialization by value and not by reference: impossible to refer or point to external data source

UNIT-4

Universal Description, Discovery and Integration (UDDI)

is a platform-independent, extensible world markup language(XML)-based registry by which businesses worldwide can list themselves on the Internet, and a mechanism to register and locate web service applications. UDDI is an open industry initiative, sponsored by the Organization for the Advancement of Structured Information Standards (OASIS), for enabling businesses to publish service listings and discover each other, and to define how the services or software applications interact over the Internet.

UDDI was originally proposed as a core Web service standard .[1] It is designed to be interrogated by SOAP messages and to provide access to Web Services Description Language (WSDL) documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.

A UDDI business registration consists of three components:

- White Pages — address, contact, and known identifiers;
- Yellow Pages — industrial categorizations based on standard taxonomies;
- Green Pages — technical information about services exposed by the business.

White Pages

White pages give information about the business supplying the service. This includes the name of the business and a description of the business - potentially in multiple languages. Using this information, it is possible to find a service about which some information is already known (for example, locating a service based on the provider's name).[6]

Contact information for the business is also provided - for example the businesses address and phone number; and other information such as the Dun & Bradstreet Universal umbering System number.

Yellow Pages

Yellow pages provide a classification of the service or business, based on standard taxonomies. These include the Standard Industrial Classification (SIC), the North American Industry Classification System (NAICS),[6] or the United Nations Standard Products and Services Code (UNSPSC) and geographic taxonomies.

Because a single business may provide a number of services, there may be several Yellow Pages (each describing a service) associated with one White Page (giving general information about the business).

Green Pages

Green pages are used to describe how to access a Web Service, with information on the service bindings. Some of the information is related to the Web Service - such as the address of the service and the parameters, and references to specifications of interfaces.[6] Other information is not related directly to the Web Service - this includes e-mail, FTP,CORBA and telephone details for the service. Because a Web Service may have multiple bindings (as defined in its WSDL description), a service may have multiple Green Pages, as each binding will need to be accessed diferently.

UDDI Nodes & Registry

UDDI nodes are servers which support the UDDI specification and belong to a UDDI registry while UDDI registries are collections of one or more nodes.

SOAP is an XML-based protocol to exchange messages between a requester and a provider of a Web Service. The provider publishes the WSDL to UDDI and the requester can join to it using SOAP.

UDDI Technical Architecture: -

The UDDI technical architecture consists of three parts:

UDDI data model:

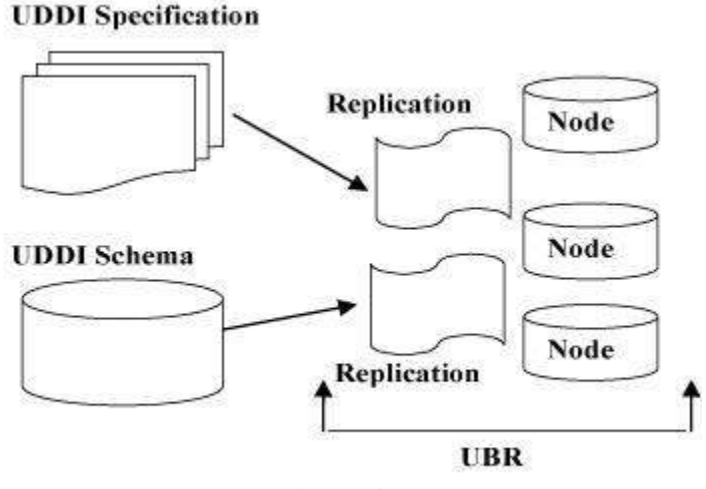
An XML Schema for describing businesses and web services. The data model is described in detail in the "UDDI Data Model" section.

UDDI API Specification:

A Specification of API for searching and publishing UDDI data.

UDDI cloud services:

This is operator sites that provide implementations of the UDDI specification and synchronize all data on a scheduled basis.



UDDI Technical Architecture

The UDDI Business Registry (UBR), also known as the Public Cloud, is a conceptually single system built from multiple nodes that has their data synchronized through replication.

The current cloud services provide a logically centralized, but physically distributed, directory. This means that data submitted to one root node will automatically be replicated across all the other root nodes. Currently, data replication occurs every 24 hours.

UDDI cloud services are currently provided by Microsoft and IBM. Ariba had originally planned to offer an operator as well, but has since backed away from the commitment. Additional operators from other companies, including Hewlett-Packard, are planned for the near future. It is also possible to set up private UDDI registries. For example, a large company may set up its own private UDDI registry for registering all internal web services. As these registries are not automatically synchronized with the root UDDI nodes, they are not considered part of the UDDI cloud.

UDDI Data Model

UDDI includes an XML Schema that describes four five data structures:

- businessEntity
- businessService
- bindingTemplate
- tModel
- publisherAssertion

- **businessEntity data structure:**

The business entity structure represents the provider of web services. Within the UDDI registry, this structure contains information about the company itself, including contact information, industry categories, business identifiers, and a list of services provided.
Here is an example of a fictitious business's UDDI registry entry:

businessService data structure:

The business service structure represents an individual web service provided by the business entity. Its description includes information on how to bind to the web service, what type of web service it is, and what taxonomical categories it belongs to:

Here is an example of a business service structure for the Hello World web service.

Notice the use of the Universally Unique Identifiers (UUIDs) in the *businessKey* and *serviceKey*attributes. Every business entity and business service is uniquely identified in all UDDI registries through the UUID assigned by the registry when the information is first entered.

bindingTemplate data structure:

Binding templates are the technical descriptions of the web services represented by the business service structure. A single business service may have multiple binding templates. The binding template represents the actual implementation of the web service.

Here is an example of a binding template for Hello World.

Because a business service may have multiple binding templates, the service may specify different implementations of the same service, each bound to a different set of protocols or a different network address.

tModel data structure:

The tModel is the last core data type, but potentially the most difficult to grasp. tModel stands for technical model.

A tModel is a way of describing the various business, service, and template structures stored within the DDI registry. Any abstract concept can be registered within UDDI as a tModel. For instance, if you define a new WSDL port type, you can define a tModel that represents that port type within UDDI. Then, you can specify that a given business service implements that port type by associating the tModel with one of that business service's binding templates.

Here is an example of A tModel representing the HelloWorldInterface port type

publisherAssertion data structure:

This is a relationship structure putting into association two or more businessEntity structures according to a specific type of relationship, such as subsidiary or department.

The publisherAssertion structure consists of the three elements fromKey (the first businessKey), toKey (the second businessKey) and keyedReference.

The keyedReference designates the asserted relationship type in terms of a keyName keyValue pair within a tModel, uniquely referenced by a tModelKey.

Service registries

- To discover Web services, a service registry is needed. This requires describing and registering the Web service
  - Compare/Contrast with URLs
- Publication of a service requires proper description of a Web service in terms of business, service, and technical information.
- Registration deals with persistently storing the Web service descriptions in the Web services registry.
- Two types of registries can be used:
  - The document-based registry: enables its clients to publish information, by storing XML-based service documents such as business profiles or technical specifications (including WSDL descriptions of the service).
  - The metadata-based service registry: captures the essence of the submitted document.

Service discovery

- Service discovery is the process of locating Web service providers, and retrieving Web services descriptions that have been previously published.
- Interrogating services involve querying the service registry for Web services matching the needs of a service requestor.
  - A query consists of search criteria such as:
    - the type of the desired service, preferred price and maximum number of returned results, and is executed against service information published by service provider.
  - Discovering Web services is a process that is also dependent on the architecture of the service registry.
- After the discovery process is complete, the service developer or client application should know the exact location of a Web service (URI), its capabilities, and how to interface with it.

Types of service discovery

Static

- The service implementation details are bound at design time and a service retrieval is performed on a service registry.
- The results of the retrieval operation are examined usually by a human designer and the service description returned by the retrieval operation is incorporated into the application logic.

Dynamic

- The service implementation details are left unbound at design time so that they can be determined at run-time.
- The Web service requestor has to specify preferences to enable the application to **infer/reason** which Web service(s) the requester is most likely to want to invoke.
- Based on application logic quality of service considerations such as best price, performance, security certificates, and so on, the application chooses the most appropriate service, binds to it, and invokes it.

What is UDDI?

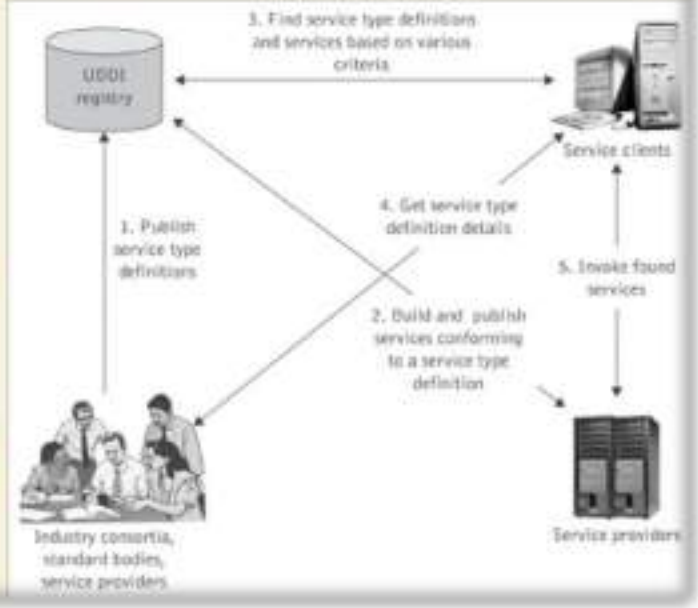
- The universal description, discovery, and integration is a registry standard for Web service description and discovery together with a registry facility that supports the WS publishing and discovery processes.
- UDDI enables a business to:
  - **describe** its business and its services;
  - **discover** other businesses that offer desired services;
  - **integrate (interoperate)** with these other businesses.
- Conceptually, a UDDI business registration consists of three inter-related components:
  - "white pages" (address, contact, and other key points of contact);
  - "yellow pages" classification info. based on standard industry taxonomies; and
  - "green pages", the technical capabilities and information about services.

UDDI – main characteristics

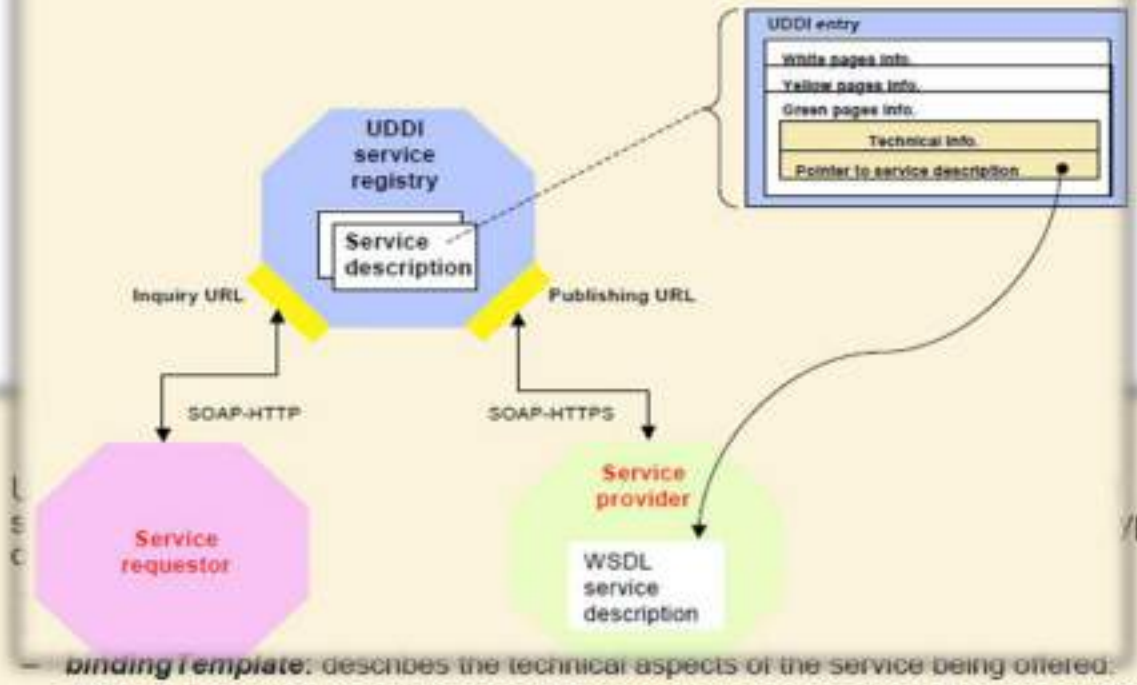
- UDDI provides a mechanism to categorize businesses and services using **taxonomies**.
  - Service providers can use a taxonomy to indicate that a service implements a specific domain standard, or that it provides services to a specific geographic area
  - UDDI uses standard taxonomies so that information can be discovered on the basis of categorization.
- **UDDI business registration: an XML document** used to describe a business entity and its Web services.
- UDDI is not bound to any technology. In other words,
  - An entry in the UDDI registry can contain any type of resource, independently of whether the resource is XML based or not, e.g., the UDDI registry could contain information about an enterprise's electronic document interchange (**EDI**) system or even a service that, uses the **fax machine** as its primary communication channel.
  - While UDDI itself uses XML to represent the data it stores, it allows for other kinds of technology to be registered

The UDDI usage model

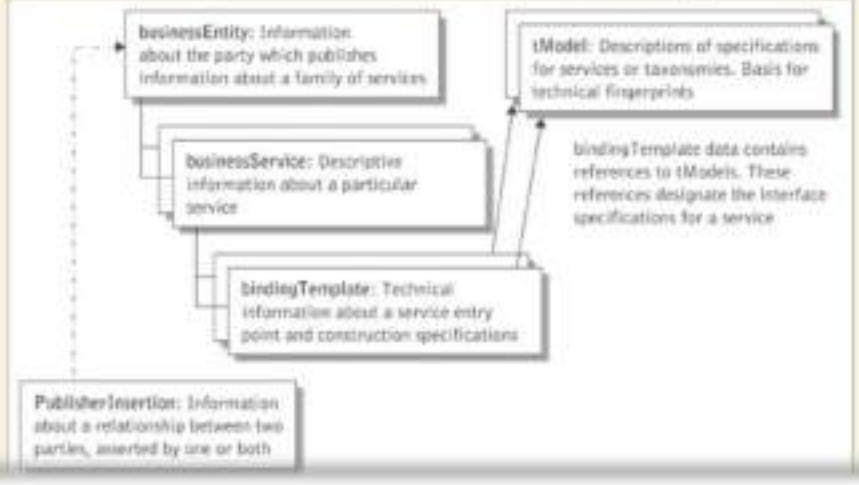
- An enterprise may set up multiple *private* UDDI registries in-house to support intranet and e-Business operations.
- Public UDDI registries can be set up by customers and business partners of an enterprise.
  - Services must be published in a public UDDI registry so that potential clients and service developers can discover them.



UDDI and WSDL



- ***bindingTemplate***: describes the technical aspects of the service being offered.
- ***tModel***: ("technical model")is a generic element that can be used to store technical information on how to use the service, conditions for use, guarantees, etc.



UDDI—A Global Registry of Web Services

UDDI is a public registry designed to house information about businesses and their services in a structured way. Through UDDI, one can publish and discover information about a business and its Web Services. This data can be classified using standard taxonomies so that information can be found based on categorization. Most importantly, UDDI contains information about the technical interfaces of a business's services. Through a set of SOAP-based XML API calls, one can interact with UDDI at both design time and run time to discover technical data, such that those services can be invoked and used. In this way, UDDI serves as infrastructure for a software landscape based on Web Services.

Why UDDI? What is the need for such a registry? As we look towards a software landscape of thousands—perhaps millions—of Web Services, s me t ugh challenges emerge:

- How are Web Services discovered?
- How is this information categorized in a meaningful way?
- What implications are there for localization?
- What implications are there around proprietary technologies? How can I guarantee interoperability in the discovery mechanism?
- How can I interact with such a discovery mechanism at run time once my application is dependent upon a web Service?

In response to these challenges, the UDDI initiative emerged. A number of companies, including Microsoft, IBM, Sun, Oracle, Compaq, Hewlett Packard, Intel, SAP, and over three hundred other companies (see DDI: Community for a complete list), came together to develop a specification based on open standards and non-proprietary technologies to solve these challenges. he result, initially launched in beta December 2000 and in production by May 2001, was a global business registry hosted by multiple operator nodes that users could—at no cost—both search and publish to.

With such an infrastructure for Web Services in place, data about Web Services can now be found consistently and reliably in a universal, completely vendor-neutral capacity. Precise categorical searches can be performed using extensible taxonomy systems and identification. Run-time UDDI integration can be incorporated into applications. As a result, a Web Services software environment can flourish.

WSDL and UDDI

WSDL has emerged as an important piece of the Web Services protocol stack. As such, it is important to grasp how UDDI and WSDL work together and how the notion of interfaces vs. implementations is part of each protocol. Both WSDL and UDDI were designed to clearly delineate between abstract meta-data and concrete implementations, and understanding the implications of the division is essential to understanding WSDL and UDDI.

For example, WSDL makes a clear distinction between messages and ports: Messages, the required syntax and semantics of a Web Service, are always abstract, while ports, the network address where the Web Service can be invoked, are always concrete. One is not required to provide port information in a WSDL file. A WSDL can contain solely abstract interface information and not provide any concrete implementation data. Such a WSDL file is considered valid. In this way, WSDL files are decoupled from implementations.

One of the most exciting implications of this is that there can be multiple implementations of a single WSDL interface. This design allows disparate systems to write implementations of the same interface, thus guaranteeing that the systems can talk to one another. If three different companies have implemented the same WSDL file, and a piece of client software has created the proxy/stub code from that WSDL interface, then the client software can communicate with all three of those implementations with the same code base by simply changing the access point.

UDDI draws a similar distinction between abstraction and implementation with its concept of tModels. The tModel structure, short for "Technology Model", represents technical fingerprints, interfaces and abstract types of meta-data. Corollary with tModels are binding templates, which are the concrete implementation of one or more tModels. Inside a binding template, one registers the access point for a particular implementation of a tModel. Just as the schema for WSDL allows one to decouple interface and implementation, UDDI provides a similar mechanism, because tModels can be published separately from binding templates that reference them. For example, a standards body or industry group might publish the canonical interface for a particular industry, and then multiple businesses could write implementations to this interface. Accordingly, each of those businesses' implementations would refer to that same tModel. WSDL files are perfect examples of a UDDI tModel.

Registering with UDDI

Publishing to UDDI is a relatively straightforward process. The first step is to determine some basic information about how to model your company and its services in UDDI. Once that is determined, the next step is to actually perform the registration, which can be done either through a Web-based user interface or programmatically. The final step is to test your entry to insure that it was correctly registered and appears as expected in different types of searches and tools.

Step 1: Modeling Your UDDI Entry

Considering the data model outlined above, several key pieces of data need to be collected before establishing a UDDI entry.

1. Determine the tModels (WSDL files) that your Web Service implementations use.

   Simliar to developing a COM component, your Web Service has been developed either based on an existing interface, or using an interface you designed yourself. In the case of a Web Service based on an existing WSDL, you will need to determine if that WSDL file has been registered in UDDI. If it has, you will need to note its name and tModelKey, which is the GUID generated by UDDI when that WSDL file was registered.

2. Determine the categories appropriate to your services.

   Just as a company can be categorized, Web Services can also be categorized. As such, a company might be categorized at the business level as **NAICS: Software Publisher (51121)**, but its hotel booking Web Se vice might be catego ized at the service level as **NAICS: Hotels and Motels (72111)**.

Step Two: Registering Your UDDI Entry

Upon completion of the modeling exercise, the next step is to register your company. You will need to obtain an account with a UDDI registry, which cannot be done programmatically, as a Terms of Use statement must be agreed to. The Microsoft Node uses Passport for its authentication, so you will need to have acquired a Passport (http://www.passport.com/Consumer/default.asp) in order to proceed.

There are two options at this point: You can either use the Web user interface provided by the Microsoft node, or you can register programmatically by issuing the SOAP API calls to the node itself. If you don't expect to be making many changes to your entry, or if your entry is relatively simple, using the Web user interface is sufficient. However, if you expect to be making frequent updates, or your entry is more complex, scripting the registration process using the Microsoft DDI SDK makes sense. Also, the Microsoft User Interface is not localized for other languages, so if you want to take advantage of that feature of the UDDI API, you will need to register programmatically.

Step Three: Searching UDDI For Your Entry

Three checks are worth performing once your entry is registered in UDDI. First, using the Microsoft Web User Interface, search for your business based on its name and categorizations to see it returned in the result sets. Second, open Visual Studio .NET and ensure that it appears through the "Add Web

Reference" dialog. If it does not appear, it is likely that your tModel was not categorized correctly using the uddi-org:types taxonomy explained above. You should be able to add the Web Service to your project and generate the proxy code based on the WSDL file. Lastly, after 24 hours, your entry will have replicated to the IBM node, which can be searched from their UI at https://www-3.ibm.com/services/uddi/protect/find.

Web Services Notification (WSN):-

The Web Services Notification (WSN) defines a set of specifications that standardize the way Web Services interact using the notification pattern. In the notification pattern, a Web Service disseminates information to a set of other Web Services, without having to have prior knowledge of these other Web Services. Characteristics of this pattern include:

- The Web Services that wish to consume information are registered with the Web Service that is capable of distributing it. As part of this registration process they may provide some indication of the nature of the information that they wish to receive.
- The distributing Web Service disseminates information by sending one-way messages to the Web Services that are registered to receive it. It is possible that more than one Web Service is registered to consume the same information. In such cases, each Web Service that is registered receives a separate copy of the information.
- The distributing Web Service may send any number of messages to each registered Web Service; it is not limited to sending just a single message.

UNIT -5

A DESCRIPTION OF WEB SERVICES :-

Each Web service has a machine processable description written in Web Services Description Language (WSDL), which is "*an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information*"8. This WSDL file can be sent directly to perspective users, or published in the UDDI registries. Upon a successful inquiry to a UDDI registry, the WSDL link about the target Web service will be returned to the requested , describing core information about the contents and providing information on how to communicate (or bind) with the target Web service.

SOA Architecture

Service-oriented architecture (SOA) allows different ways to develop applications by combining services. The main premise of SOA is to erase application boundaries and technlogy differences. As applications are opened up, how we can combine these services securely becomes an issue. Traditionally, security models have been hardcoded into applications and when capabilities of an application are opened up for use by other applications, the security models built into each application may not be good enough.

Several emerging technologies and standards address different aspects of the problem of security in SOA. Standards such as WS-Security, SAML, S-Trust, S-SecureConversation and WS-SecurityPolicy focus on the security and identity management aspects of SOA implementations that use Web services. Technologies such as virtual organization in grid computing, application-oriented networking (AON) and XML gateways are addressing the problem of SOA security in the larger context.

XML gateways are hardware or software based solutions for enforcing identity and security for SOAP, XML, and REST based web services, usually at the network perimeter. An XML gateway is a dedicated application which allows for a more centralized approach to security and identity enforcement, similar to how a protocol firewall is deployed at the perimeter of a network for centralized access control at the connection and port level.

XML Gateway SOA Security features include PKI, Digital Signature, encryption, XML Schema validation, antivirus, and pattern recognition. Regulatory certification for XML gateway security features are provided by FIPS and United States Department of Defense.

Web Services Security (WS-Security, WSS):-

It is an extension to SOAP to apply security to Web services. It is a member of the Web service specifications and was published byOASIS.

The protocol specifies how integrity and confidentiality can be enforced on messages and allows the communication of various security token formats, such as Security Assertion Markup Language (SAML), Kerberos, and X.509. Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security.

USE Case

End-to-end Security

If a SOAP intermediary is required, and the intermediary is not or is less trusted, messages need to be signed and optionally encrypted. This might be the case of an application-level proxy at a network perimeter that will terminate TCP connections.

Non-repudiation

The standard method for non-repudiation is to write transactions to an audit trail that is subject to specific security safeguards. However, if the audit trail is not sufficient, digital signatures may provide a better method to enforce non-repudiation. WS-Security can provide this.

Alternative transport bindings

Although almost all SOAP services implement HTTP bindings, in theory other bindings such as JMS or SMTP could be used; in this case end-to-end security would be required.

Reverse proxy/common security token

Even if the web service relies upon transport layer security, it might be required for the service to know about the end user, if the service is relayed by a (HTTP-) reverse proxy. A WSS header could be used to convey the end user's token, vouched for by the reverse proxy.

Security Topologies:-

One of the most essential portions of information security is the design and topology of secure networks. What exactly do we mean by “topology?” Usually, a geographic diagram of a network comes to mind. However, in networking, topologies are not related to the physical arrangement of equipment, but rather, to the logical connections that act between the different gateways, routers, and servers. We will take a closer look at some common security topologies.

With network security becoming such a hot topic, you may have come under the microscope about your firewall and network security configuration. You may have even been assigned to implement or reassess a firewall design. In either case, you need to be familiar with the most common firewall configurations and how they can increase security. In this article, I will introduce you to some common firewall configurations and some best practices for designing a secure network topology.

Setting up a firewall security strategy

At its most basic level, a firewall is some sort of hardware or software that filters traffic between your company's network and the Internet. With the large number of hackers roaming the Internet today and the ease of downloading hacking tools, every network should have a security policy that includes a firewall design.

If your manager is pressuring you to make sure that you have a strong firewall in place and to generally beef up network security, what is your next move? Your strategy should be two fold:

- Examine your network and take account of existing security mechanisms (routers with access lists, intrusion detection, etc.) as part of a firewall and security plan.
- Make sure that you have a dedicated firewall solution by purchasing new equipment and/or software or upgrading your current systems.

Keep in mind that a good firewall topology involves more than simply filtering network traffic. It should include:

- A solid security policy.
- Traffic checkpoints.
- Activity logging.
- Limiting exposure to your internal network.

Before purchasing or upgrading your dedicated firewall, **you should have a solid security policy in place**. A firewall will enforce your security policy, and by having it documented, there will be fewer questions when configuring your firewall to reflect that policy. Any changes made to the firewall should be amended in the security policy.

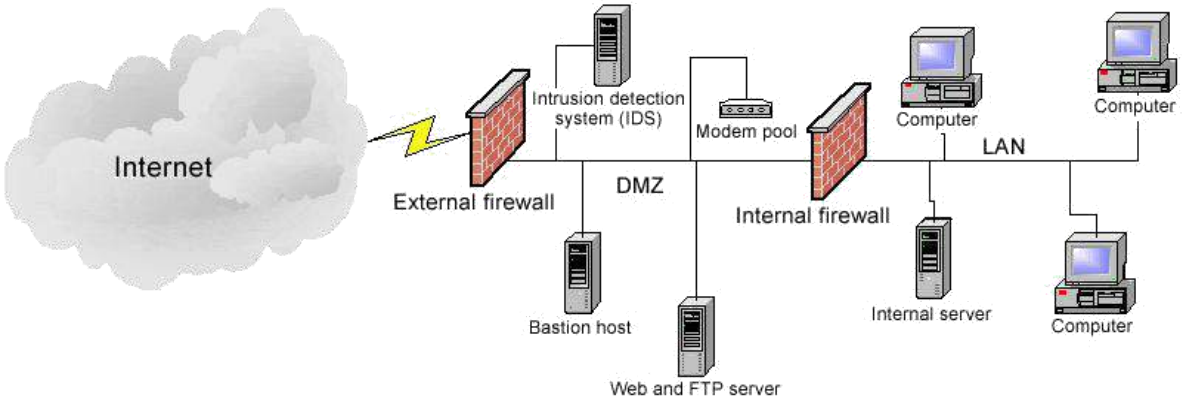
One of the best features of a well-designed firewall is the ability to funnel traffic through checkpoints. When you configure your firewall to force traffic (outbound and inbound)

through specific points in your firewall, you can easily monitor your logs for normal and suspicious activity.

How do you monitor your firewall once you have a security policy and checkpoints configured? By using alarms and enabling logging on your firewall, you can easily monitor all authorized and unauthorized access to your network. You can even purchase third-party utilities to help filter out the messages you don't need. It's also a good practice to hide your internal network address scheme from the outside world. It is never wise to let the outside world know the layout of your network.

Demilitarizedzone (DMZ) topology

A DMZ is the most common and secure firewall topology. It is often referred to as a screened subnet. A DMZ creates a secure space between your Internet and your network, as shown in Figure D.



A DMZ will typically contain the following:

- Web server
- Mail server
- Application gateway
- E-commerce systems (It should contain only your front-end systems. Your back-end systems should be on your internal network.)

XML and Web Services Security Standards:-

XML and Web services are widely used in current distributed systems. The security of the XML based communication, and the Web services themselves, is of great importance to the overall security of these systems. Furthermore, in order to facilitate interoperability, the

security mechanisms should preferably be based on established standards. In this paper we provide a tutorial on current security standards for XML and Web services. The discussed standards include XML Signature, XML Encryption, the XML Key Management Specification (XKMS), WS-Security, WS-Trust, WS-SecureConversation, Web Services Policy, WS-SecurityPolicy, the eXtensible Access Control Markup Language (XACML), and the Security Assertion Markup Language (SAML).

What Is an XML Web Service?

XML Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented.

There are probably as many definitions of XML Web Service as there are companies building them, but almost all definitions have these things in common:

- XML Web Services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is SOAP.
- XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document.
- XML Web services are registered so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

I'll cover all three of these technologies in this article but first I want to explain why you should care about XML Web services.

One of the primary advantages of the XML Web services architecture is that it allows programs written in different languages on different platforms to communicate with each other in a standards-based way. Those of you who have been around the industry a while are now saying, "Wait a minute! Didn't I hear those same promises from CORBA and before that DCE? How is this any different?" The first difference is that SOAP is significantly less

complex than earlier approaches, so the barrier to entry for a standards-compliant SOAP implementation is significantly lower. Paul Kulchenko maintains a list of SOAP implementations which at last count contained 79 entries. You'll find SOAP implementations from most of the big software companies, as you would expect, but you will also find many implementations that are built and maintained by a single developer. The other significant advantage that XML Web services have over previous efforts is that they work with standard Web protocols—XML, HTTP and TCP/IP. A significant number of companies already have a Web infrastructure, and people with knowledge and experience in maintaining it, so again, the cost of entry for XML Web services is significantly less than previous technologies.

We've defined an XML Web service as a software service exposed on the Web through SOAP, described with a WSDL file and registered in UDDI. The next logical question is: "What can I do with XML Web services?" The first XML Web services tended to be information sources that you could easily incorporate into applications—stock quotes, weather forecasts, sports scores etc. It's easy to imagine a whole class of applications that could be built to analyze and aggregate the information you care about and present it to you in a variety of ways; for example, you might have a Microsoft® Excel spreadsheet that summarizes your whole financial picture—stocks, 401K, bank accounts, loans, etc. If this information is available through XML Web services, Excel can update it continuously. Some of this information will be free and some might require a subscription to the service. Most of this information is available now on the Web, but XML Web services will make programmatic access to it easier and more reliable.

Exposing existing applications as XML Web services will allow users to build new, more powerful applications that use XML Web services as building blocks. For example, a user might develop a purchasing application to automatically obtain price information from a variety of vendors, allow the user to select a vendor, submit the order and then track the shipment until it is received. The vendor application, in addition to exposing its services on the Web, might in turn use XML Web services to check the customer's credit, charge the customer's account and set up the shipment with a shipping company.

In the future, some of the most interesting XML web services will support applications that use the Web to do things that can't be done today. For example, one of the services that XML Web Services would make possible is a calendar service. If your dentist and mechanic exposed their calendars through this XML web service, you could schedule appointments with them on line or they could schedule appointments for cleaning and routine maintenance directly in your calendar if you like. With a little imagination, you can envision hundreds of applications that can be built once you have the ability to program the Web.

SOAP

Soap is the communications protocol for XML Web services. When SOAP is described as a communications protocol, most people think of DCOM or CORBA and start asking things like, "How does SOAP do object activation?" or "What naming service does SOAP use?" While a SOAP implementation will probably include these things, the SOAP standard doesn't specify them. SOAP is a specification that defines the XML format for messages— and that's about it for the required parts of the spec. If you have a well-formed XML fragment enclosed in a couple of SOAP elements, you have a SOAP message. Simple isn't it? There are other parts of the SOAP specification that describe how to represent program data as XML and how to use SOAP to do Remote Procedure Calls. These optional parts ofthe specification are used to implement RPC-style applications where a SOAP message containing a callable function, and the parameters to pass to the function, is sent from the client, and the server returns a message with the results of the executed function. Most current implementations of SOAP support RPC applications because programmers who are used to doing COM or CORBA applications understand the RPC style. SOAP also supports document style applications where the SOAP message is just a wrapper around an XML document. Document-style SOAP applications are very flexible and many new XML Web services take advantage of this flexibility to build services that would be difficult to implement using RPC.

The last optional part of the SOAP specification defines what an HTTP message that contains a SOAP message looks like. This HTTP binding is important because HTTP is supported by almost all current OS's (and many not-so-current OS's). The HTTP binding is optional, but almost all SOAP implementations support it because it's the only standardized protocol for SOAP. For this reason, there's a common misconception that SOAP requires HTTP. Some implementations support MSMQ, MQ Series, SMTP, r TCP/IP transports, but almost all current XML Web services use HTTP because it is ubiquitious. Since HTTP is a core protocol of the Web, most organizations have a network infrastructure that supports HTTP and people who understand how to manage it already. The security, monitoring, and load-balancing infrastructure for HTTP are readily available today.

A major source of confusion when getting started with SOAP is the difference between the SOAP specification and the many implementations of the SOAP specification. Most people who use SOAP don't write SOAP messages directly but use a SOAP toolkit to create and parse the SOAP messages. These toolkits generally translate function calls from some kind of language to a SOAP message. For example, the Microsoft SOAP Toolkit 2.0 translates COM function calls to SOAP and the Apache Toolkit translates JAVA function calls to SOAP. The types of function calls and the data types of the parameters supported vary with each SOAP implementation so a function that works with one toolkit may not work with another. This isn't a limitation of SOAP but rather of the particular implementation you are using.

By far the most compelling feature of SOAP is that it has been implemented on many different hardware and software platforms. This means that SOAP can be used to link disparate systems within and without your organization. Many attempts have been made in the past to come up with a common communications protocol that could be used for systems integration, but none of them have had the widespread adoption that SOAP has. Why is this? Because SOAP is much smaller and simpler to implement than many of the previous protocols. DCE and CORBA for example took years to implement, so only a few implementations were ever released. SOAP, however, can use existing XML Parsers and HTTP libraries to do most of the hard work, so a SOAP implementation can be completed in a matter of months. This is why there are more than 70 SOAP implementations available. SOAP obviously doesn't do everything that DCE or CORBA do, but the lack of complexity in exchange for features is what makes SOAP so readily available. The ubiquity of HTTP and the simplicity of SOAP make them an ideal basis for implementing XML Web services that can be called from almost any environment. For more information on SOAP.

What About Security?

One of the first questions newcomers to SOAP ask is how does SOAP deal with security. Early in its development, SOAP was seen as an HTTP-based protocol so the assumption was made that HTTP security would be adequate for SOAP. After all, there are thousands of Web applications running today using HTTP security so surely this is a equate for SOAP. For this reason, the current SOAP standard assumes security is a transport issue and is silent on security issues.

When SOAP expanded to become a more general-purpose protocol running on top of a number of transports, security became a bigger issue. For example, HTTP provides several ways to authenticate which user is making a SOAP call, but how does that identity get propagated when the message is routed from HTTP to an SMTP transport? SOAP was designed as a building-block protocol, so fortunately, there are already specifications in the

works to build on SOAP to provide additional security features for Web services. The WS-Security specification defines a complete encryption system.

WSDL

WSDL (often pronounced whiz -dull) stands for Web Services Description Language. For our purposes, we can say that a WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. In other words, WSDL is to SOAP what IDL is to CORBA or COM. Since WSDL is XML, it is readable and editable but in most cases, it is generated and consumed by software.

To see the value of WSDL, imagine you want to start calling a SOAP method provided by one of your business partners. You could ask him for some sample SOAP messages and write your application to produce and consume messages that look like the samples, but this can be error-prone. For example, you might see a customer ID of 2837 and assume it's an integer when in fact it's a string. WSDL specifies what a request message must contain and what the response message will look like in unambiguous notation.

The notation that a WSDL file uses to describe message formats is based on the XML Schema standard which means it is both programming-language neutral and standards-based which makes it suitable for describing XML Web services interfaces that are accessible from a wide variety of platforms and programming languages. In addition to describing message contents, WSDL defines where the service is available and what communications protocol is used to talk to the service. This means that the WSDL file defines everything required to write a program to work with an XML Web service. There are several tools available to read a WSDL file and generate the code required to communicate with an XML Web service. Some of the most capable of these tools are in Microsoft Visual Studio® .NET.

Many current SOAP toolkits include tools to generate WSDL files from existing program interfaces, but there are few tools for writing WSDL directly, and tool support for WSDL isn't as complete as it should be. It shouldn't be long before tools to author WSDL files, and then generate proxies and stubs much like COM IDL tools, will be part of most SOAP implementations. At that point, WSDL will become the prefered way to author SOAP interfaces for XML Web services.

UDDI

Universal Discovery Description and Integration is the yellow pages of Web services. As with traditional yellow pages, you can search for a company that offers the services you need, read about the service offered and contact someone for more information. You can, of course, offer a Web service without registering it in UDDI, just as you can open a business in your basement and rely on word -of- mouth advertising but if you want to reach a significant market, you need UDDI so your customers can find you.

A UDDI directory entry is an XML file that describes a business and the services it offers. There are three parts to an entry in the UDDI directory. The "white pages" describe the

company offering the service: name, address, contacts, etc. The "yellow pages" include industrial categories based on standard taxonomies such as the North American Industry Classification System and the Standard Industrial Classification. The "green pages" describe the interface to the service in enough detail for someone to write an application to use the Web service. The way services are defined is through a

UDDI document called a Type Model or tModel. In many cases, the tModel contains a WSDL file that describes a SOAP interface to an XML Web service, but the tModel is flexible enough to describe almost any kind of service.

The UDDI directory also includes several ways to search for the services you need to build your applications. For example, you can search for providers of a service in a specified geographic location or for business of a specified type. The UDDI directory will then supply information, contacts, links, and technical data to allow you to evaluate which services meet your requirements.

UDDI allows you to find businesses you might want to obtain Web services from. What if you already know whom you want to do business with but you don't know what services are offered? The WS-Inspection specification allows you to browse through a collection of XML Web services offered on a specific server to find which ones might meet your needs.

Semantic interpolation:-

he problem of interpolation is a classical problem in logic. Given a consequence relation |~ and two formulas and ψ with |~ ψ we try to find a “simple" formula α such that |~ α |~ ψ. “Simple" is defined here as “expressed in the common language of and ψ". Non-monotonic logics like preferential logics are often a mixture of a non-monotonic part with classical logic. In such cases, it is natural examine also variants of the interpolation problem, like: is there “simple" α such that α |~ ψ where is classical consequence? We translate the interpolation problem from the syntactic level to the semantic level. For example, the classical interpolation problem is now the question whether there is some “simple" model set X such that M() X M(ψ). We can show that such X always exist f monotonic and anti tonic logics. The case of non-monotonic logics is more complicated, there are several variants to consider, and we mostly have only partial results.

Service-Oriented Architecture (SOA):-

A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.

Service-oriented architectures are not a new thing. The first service-oriented architecture for many people in the past was with the use DCOM or Object Request Brokers (ORBs) based on the CORBA specification. For more on DCOM and CORBA

Services

If a service-oriented architecture is to be effective, we need a clear understanding of the term service. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. See Service.

Connections

The technology of Web Services is the most likely connection technology of service-oriented architectures. The following figure illustrates a basic service-oriented architecture. It shows a service consumer at the right sending a service request message to a service provider at the left. The service provider returns a response message to theservice consumer. The request and subsequent response connections are defined in some way that is understandable to both the service consumer and service provider. How those connections are defined is explained in Web Services Explained. A service provider can also be a service consumer.

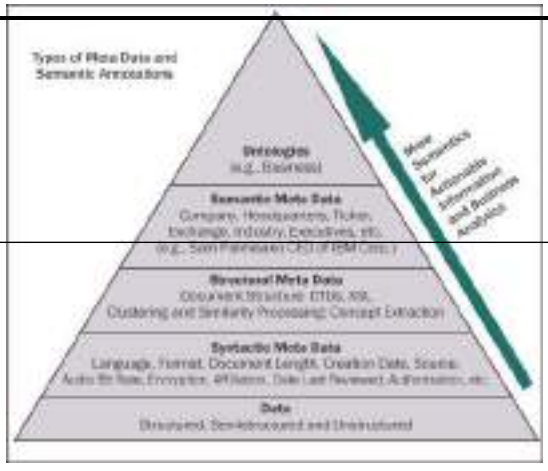
Metadata

Metadata can be defined as a set of assertions about things in our domain of discourse. Metadata is a component of data, which describes the data. It is " data about data". Often there is more than that, involving information about data as they is stored managed ,and revealing partial semantics such as intended use (i.e., application) of data. This information can be of broad variety, meeting if not surpassing the variety in the data themselves. They may describe, or be a summary of the information content of the individual databases in an intentional manner. Some metadata may also capture content independent information like location and time of creation.
Metadata descriptions present two advantages [2]:

- They enable the abstraction of representational details such as the format and organization of data, and capture the information content of the underlying data independent of representational details. This represents the first step in reduction of information overload, as intentional metadata descriptions are in general an order of magnitude smaller than the underlying data.
- They enable representation of domain knowledge describing the information domain to which the underlying data belong. This knowledge may then be used to make inferences about the underlying data. This helps in reducing information overload as the inferences may be used to determine the relevance of the underlying data without accessing the data.

Metadata can be classified based on different criteria. Based on the level of abstraction in which a metadata describes content, the metadata can be classified as follows [9]:

• Syntactic Metadata focuses on details of the data source (document) providing little insight into the data. This kind of metadata is useful mainly for categorizing or cataloguing the data source. Examples if syntactic metadata include language of the data source, creation date, title, size, format etc.



• Structural Metadata *focuses on the structure of* the document data, which facilitates data storage, processing and presentation such as navigation, eases Book Chapter, Datenbanken und Informationssysteme, Festschrift zum 60. Geburtstag von Gunter Schlageter, Publication Hagen, October 2003-09-26

3. information retrieval, and improves display. E.g. XML schema, the physical structure of the document like page images etc.

• *Semantic Metadata* describes contextually relevant information focusing on domain-specific elements based on ontology, which a user familiar with the domain is likely to know or understand easily. Using semantic metadata, meaningful interpretation of data is possible and interoperability will then be supported at high-level (hence easier to use), providing meaning to the underlying syntax and structure.

Metadata in WSDL and UDDI standards:-

The standards such as WSDL [14] and UDDI are used to share the metadata about a web service. Each standard provides metadata about services at a certain level of abstraction. WSDL describes the service using the implementation details and hence it can be considered as a standard to represent the metadata of the invocation details of service. As the purpose of UDDI is to locate WSDL descriptions, it can be thought of as a standard for publishing and discovering metadata of web services. Considering the details in WSDL and UDDI as metadata of a Web service, the different kind of metadata of Web services available in different standards can be categorized as shown in Table 1.

	WSDL	UDDI
Semantic Metadata	Documentation element (Human Readable Semantics), Namespaces,	Description element (Human Readable Semantics), Ontologies

Semantics for Web Services:-

In the previous section, we discussed different kinds of metadata available in WSDL and UDDI. Section 2 discussed the power of semantic metadata. In Web services domain, semantics represented by the semantic metadata can be classified into the following types [21], namely

o Functional Semantics

o Data Semantics

o QoS Semantics and

o Execution Semantics

These different types of semantics can be used to represent the capabilities, requirements, effects and execution pattern of a Webservice. The semantic Web research focuses to date as focused on the data semantics that helps in semantic tagging of static information available on the Web from all kind of sources. Research on semantic Web services on the other hand is based on the findings and results from the semantic Web research to apply for services that perform some action producing an effect. Unlike information retrieval,

Enterprise Management Framework (EMF)

The Hirsch Enterprise Management Framework, or EMF, is an IIS-based application that provides a browser interface to portions of the Hirsch Velocity application for user convenience and enterprise system consolidation.

EMF allows operators with occasional need to access the Velocity application to add delete a user, view events or run reports from a browser, rather than the full Velocity client. It allows

allows users with multiple Velocity servers to manage users across one more servers, view events from one or more servers, and run activity reports across one or more servers.