

UNIT - I

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

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

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Conventional software management practices are sound in theory, but practice is still tied to archaic (outdated) technology and techniques.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

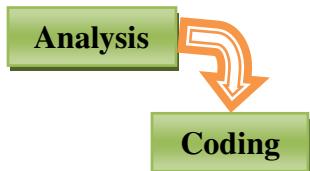
1.1.1 IN THEORY

It provides an insightful and concise summary of conventional software management

Three main primary points are

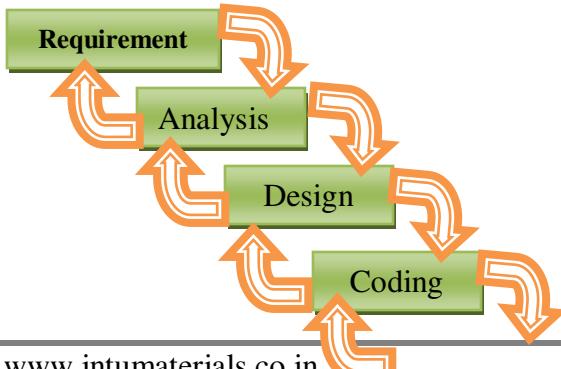
1. There are two essential steps common to the development of computer programs: **analysis** and **coding**.

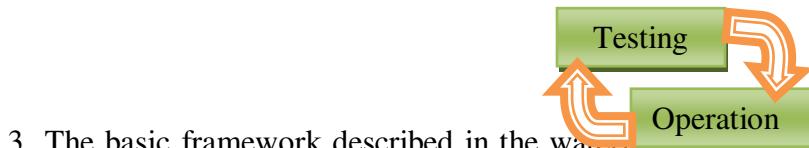
Waterfall Model part 1: The two basic steps to building a program.



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 rigid and inflexible, which can lead to a sky 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-horning of no optimal fixes, with no time for redesign.
- A very fragile, unmentionable product delivered late.

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

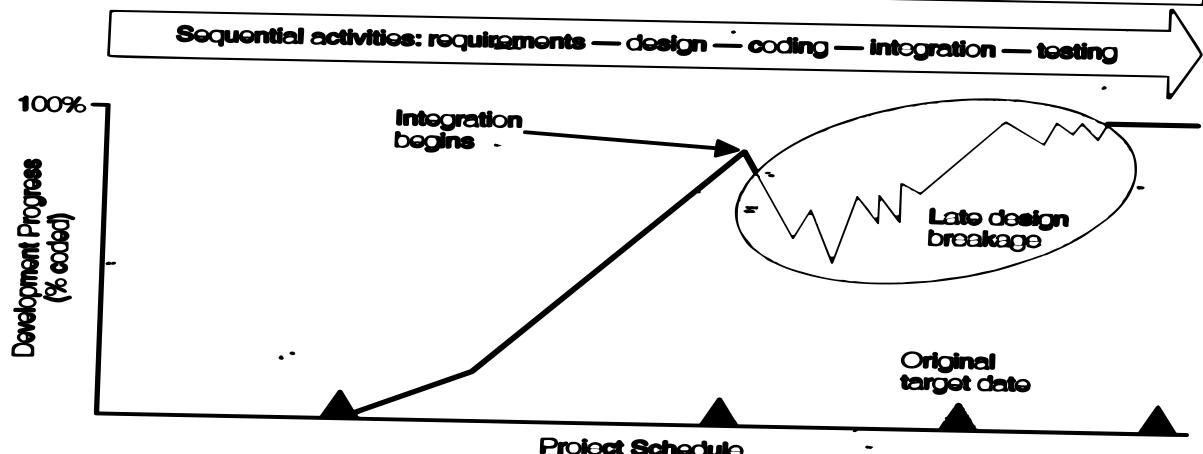


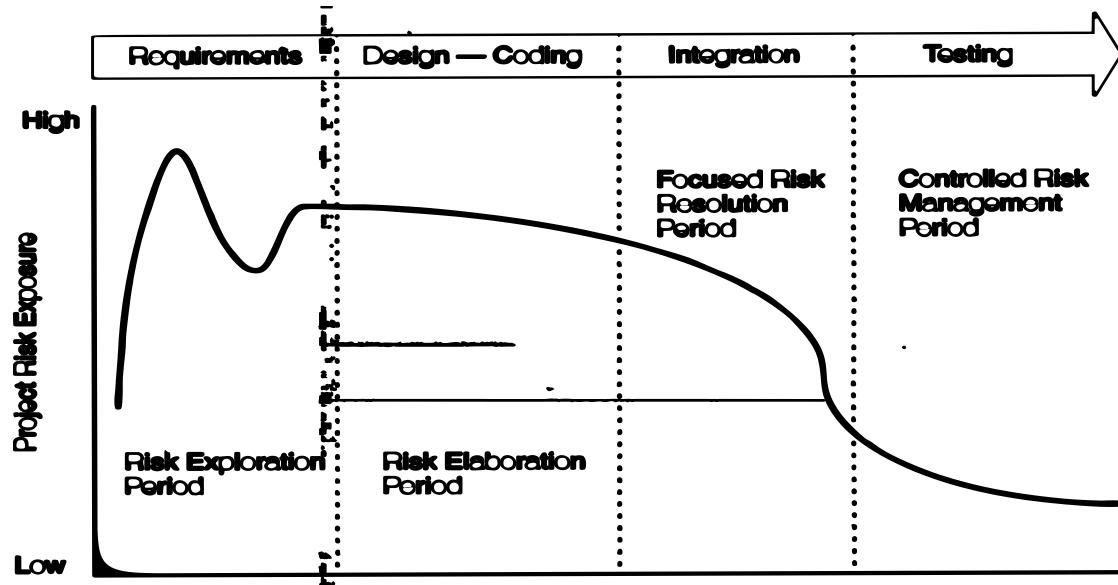
FIGURE 1-2. *Progress profile of a conventional software project*

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.

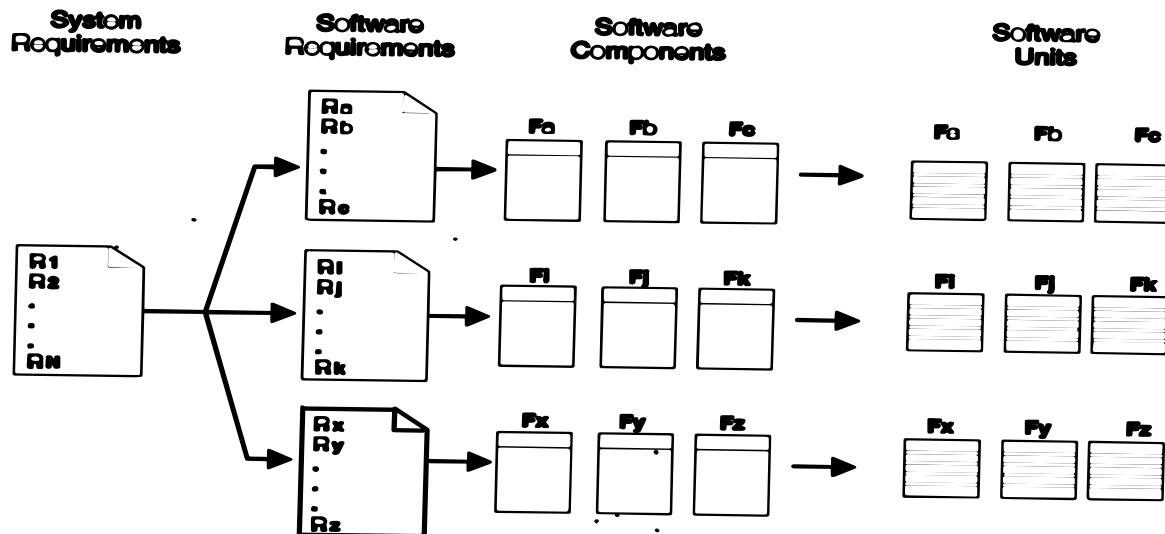


FIGURE 1-4: Suboptimal software component organization resulting from a requirements-driven approach

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:

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

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

processes, the more software you build, the more expensive it is per unit item.

Figure 2-1 shows three generations of basic technology advancement in tools, components, and processes. The required levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs (pick your favorite: per SLOC, per function point, per component) realized by an organization.

The three generations of software development are defined as follows:

- 1) **Conventional:** 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.
- 2) **Transition:** 1980s and 1990s, software engineering. Organizations used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.
- 3) **Modern practices:** 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custom built

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



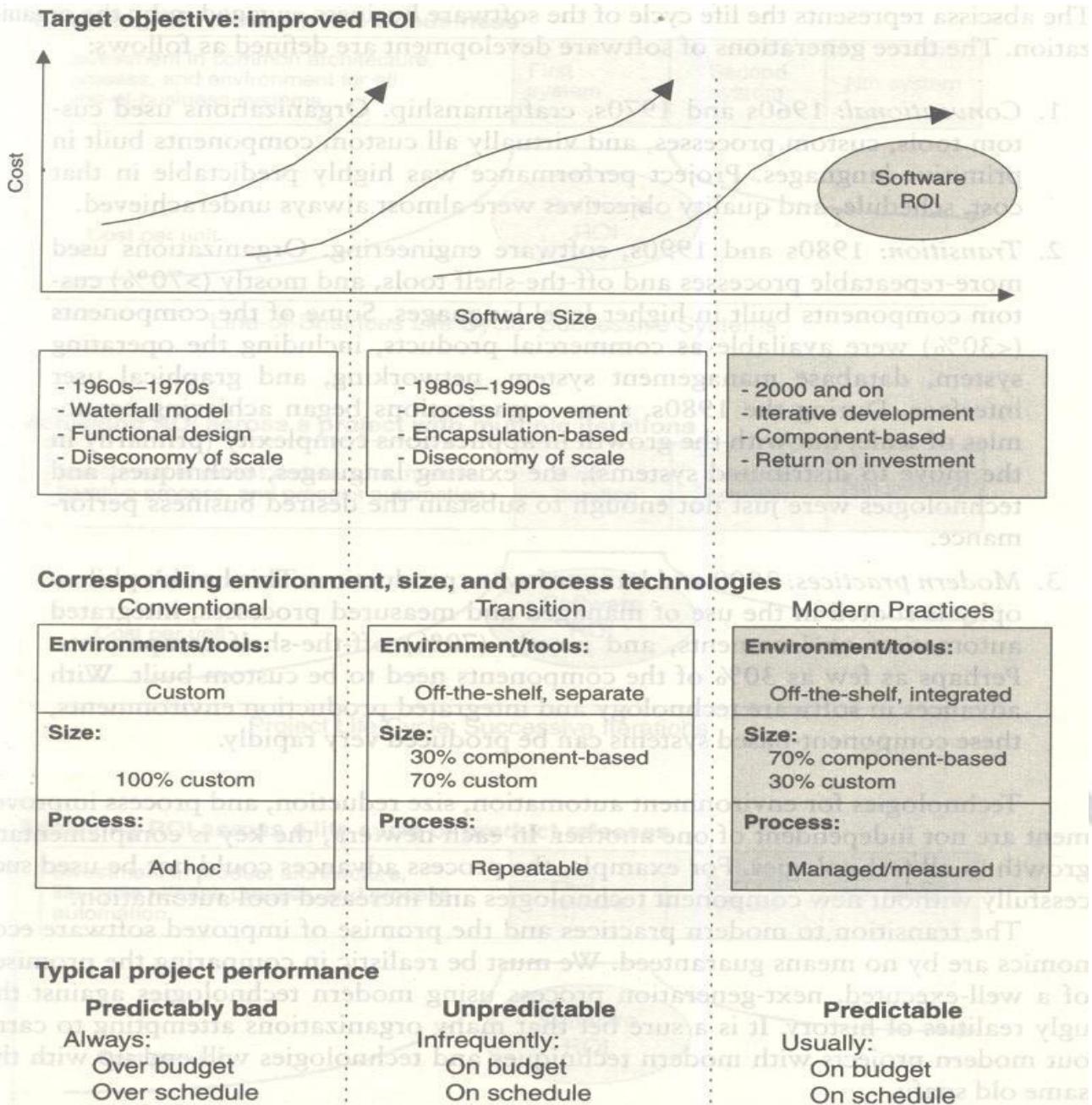
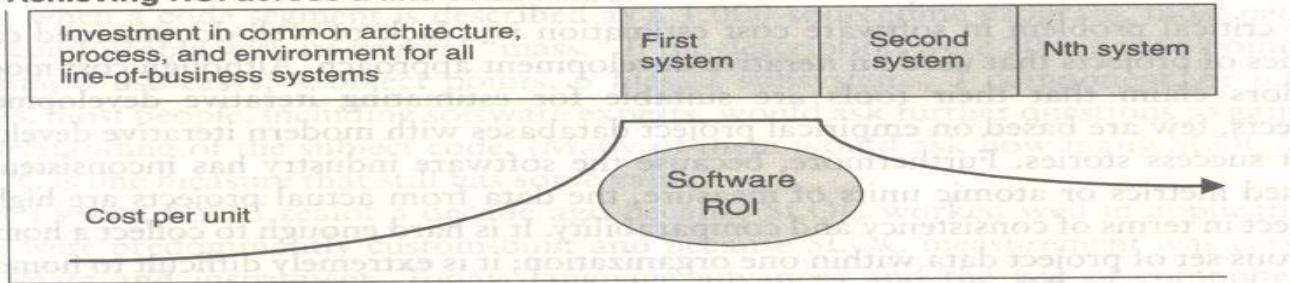


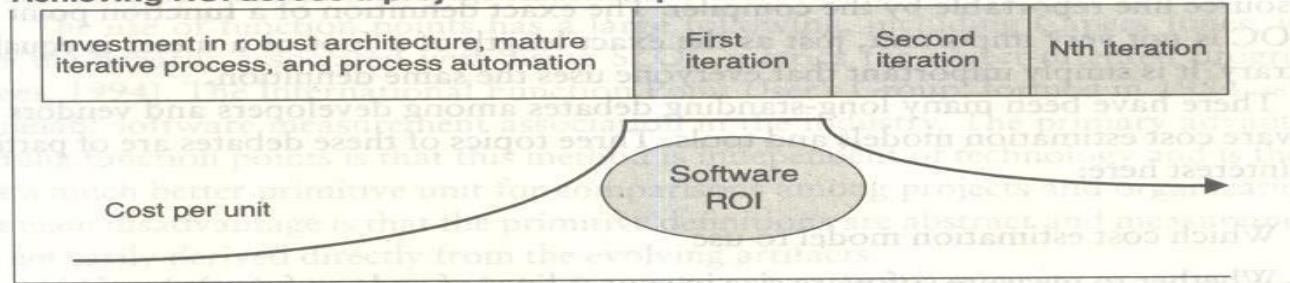
FIGURE 2-1. *Three generations of software economics leading to the target objective*

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.

Achieving ROI across a line of business



Achieving ROI across a project with multiple iterations



Achieving ROI across a life cycle of product releases

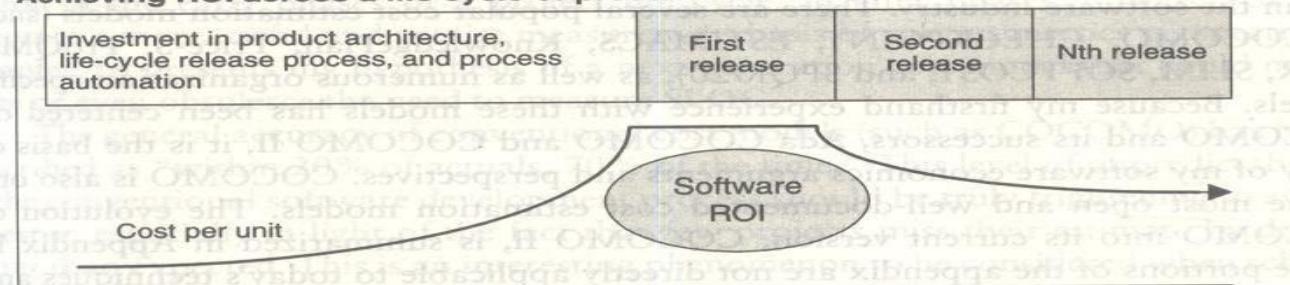


FIGURE 2-2. *Return on investment in different domains*

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, KnowledgePlan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), CO COMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described as "within 20% of 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.

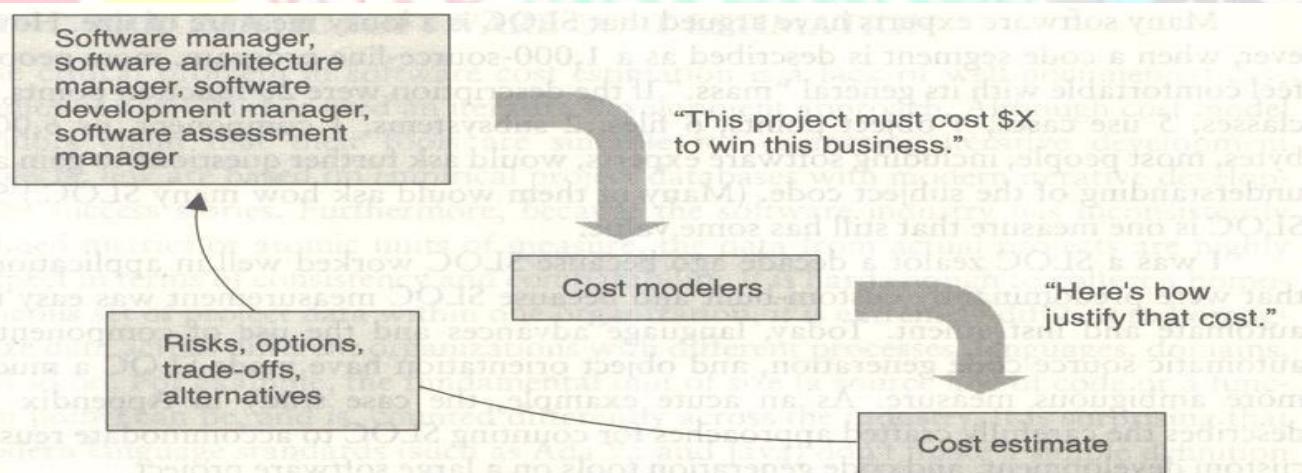


FIGURE 2-3. *The predominant cost estimation process*

==THE END==



UNIT – II

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

The old way and the new: The principles of conventional software Engineering, principles of modern software management, transitioning to an iterative 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 Process maturity models Architecture-first development Acquisition reform
Personnel	Training and personnel skill development Teamwork Win-win cultures
Environment	Integrated tools (visual modeling, compiler, editor, debugger, change management, etc.) Open systems Hardware platform performance Automation of coding, documents, testing, analyses
Quality	Hardware platform performance Demonstration-based assessment Statistical quality control
Performance, reliability, accuracy	

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 (UFPs¹) are useful estimators for language-independent, early life-cycle estimates. The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries. SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Substantial data have been documented relating SLOC to function points. Some of these results are shown in Table 3-2.

Languages expressiveness of some of today's popular languages

LANGUAGES	SLOC per UFP
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.

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.

3. An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture.

Booch also summarized five characteristics of a successful object-oriented project.

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

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.

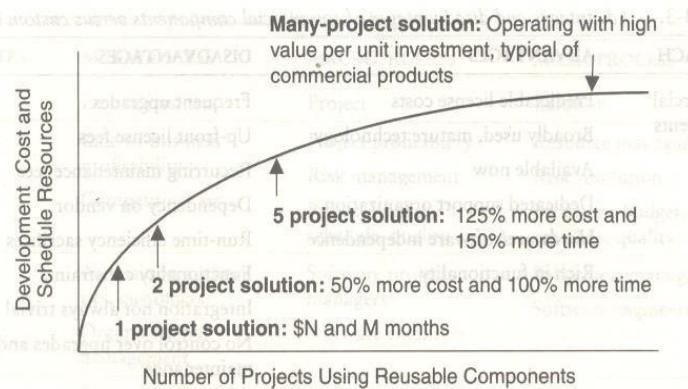


FIGURE 3-1. Cost and schedule investments necessary to achieve reusable components

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

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

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

Economic improvements associated with tools and environments. It is common for tool vendors to make relatively accurate individual assessments of life-cycle activities to support claims about the potential economic impact of their tools. For example, it is easy to find statements such as the following from companies in a particular tool.

- Requirements analysis and evolution activities consume 40% of life-cycle costs.
- Software design activities have an impact on more than 50% of the resources.
- Coding and unit testing activities consume about 50% of software development effort and schedule.
- Test activities can consume as much as 50% of a project's resources.
- Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.
- Documentation activities can consume more than 30% of project engineering resources.
- Project management, business administration, and progress assessment can consume as much as 30% of project budgets.

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.

4. THE OLD WAY AND THE NEW

4.1 THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

1. **Make quality #1.** Quality must be quantified and mechanisms put into place to motivate its achievement
2. **High-quality software is possible.** Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people
3. **Give products to customers early.** No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it
4. **Determine the problem before writing the requirements.** When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution
5. **Evaluate design alternatives.** After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model.** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases.** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle.
8. **Minimize intellectual distance.** To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure
9. **Put techniques before tools.** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer
10. **Get it right before you make it faster.** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding
11. **Inspect code.** Inspecting the detailed design and code is a much better way to find errors than testing
12. **Good management is more important than good technology.** Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success.** Highly skilled people with appropriate experience, talent, and training are key.
14. **Follow with care.** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.
15. **Take responsibility.** When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.
16. **Understand the customer's priorities.** It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more they see, the more they need.** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18. **Plan to throw one away.** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first

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

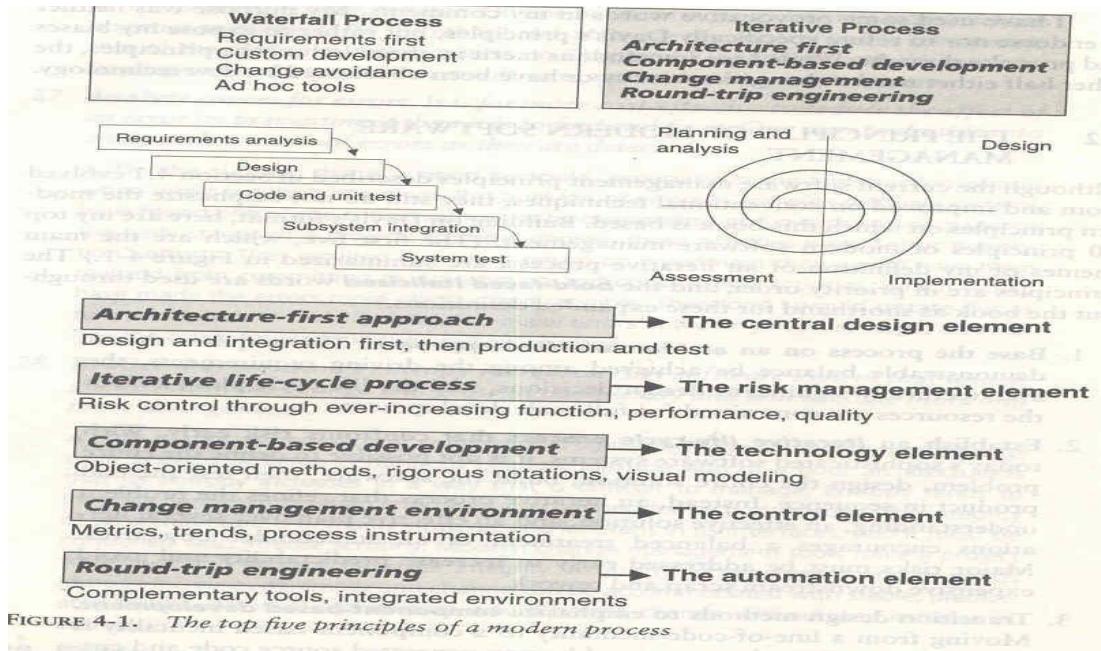
Top 10 principles of modern software management are. (The first five, which are the main themes of my definition of an iterative process, are summarized in Figure 4-1.)

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

Establish an *iterative life-cycle process* that confronts risk early. With today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, and then test the end product in sequence. Instead, an iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages a balanced treatment of all stakeholder objectives. Major risks must be addressed early to increase predictability and avoid expensive downstream scrap and rework.

Transition design methods to emphasize *component-based development*. Moving from a line-of-code mentality to a component-based mentality is necessary to reduce the amount of human-generated source code and custom development.

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



5. **Enhance change freedom through tools that support round-trip engineering.** 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 unprecedeted 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.	<i>Explain briefly Waterfall model. Also explain Conventional s/w management performance?</i>
2.	<i>Define Software Economics. Also explain Pragmatic s/w cost estimation?</i>
3.	<i>Explain Important trends in improving Software economics?</i>
4.	<i>Explain five staffing principal offered by Boehm. Also explain Peer Inspections?</i>
5..	<i>Explain principles of conventional software engineering?</i>
6.	<i>Explain briefly principles of modern software management</i>

UNIT - III

Life cycle phases: Engineering and production stages, inception, Elaboration, construction, transition phases.

Artifacts of the process: The artifact sets, Management artifacts, Engineering artifacts, programmatic artifacts.

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 economies of scale
Management	Planning	Operations

The transition between engineering and production is a crucial event for the various stakeholders. The production plan has been agreed upon, and there is a good enough understanding of the problem and the solution that all stakeholders can make a firm commitment to go ahead with production.

Engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition. These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model as shown in Figure 5-1

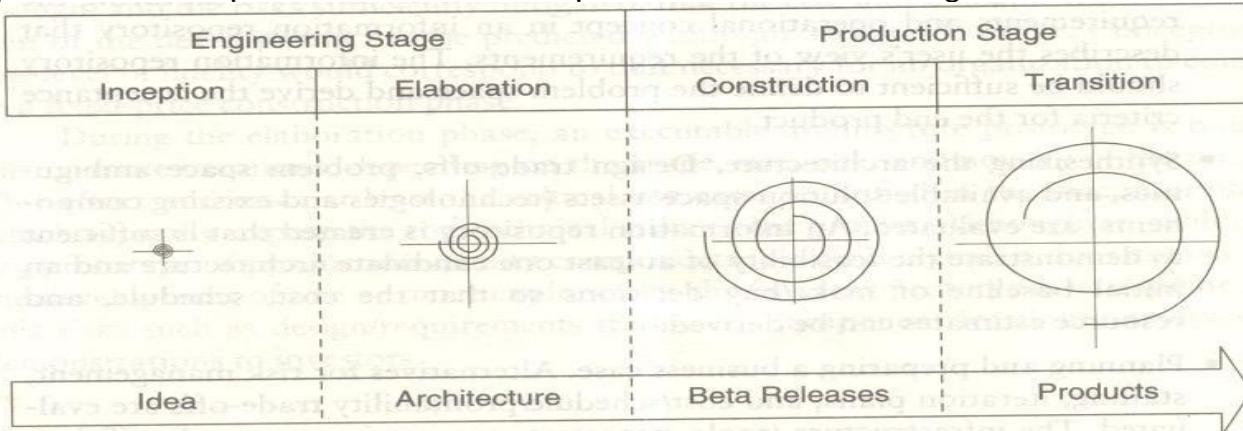


FIGURE 5-1. The phases of the life-cycle process

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 scenarios
- 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

- Baseling the architecture as rapidly as practical (establishing a configuration-managed snapshot in which all changes are rationalized, tracked, and maintained)
- Baseling the vision
- Baseling 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?

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. *Artifact* represents cohesive information that typically is developed and reviewed as a single entity.

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

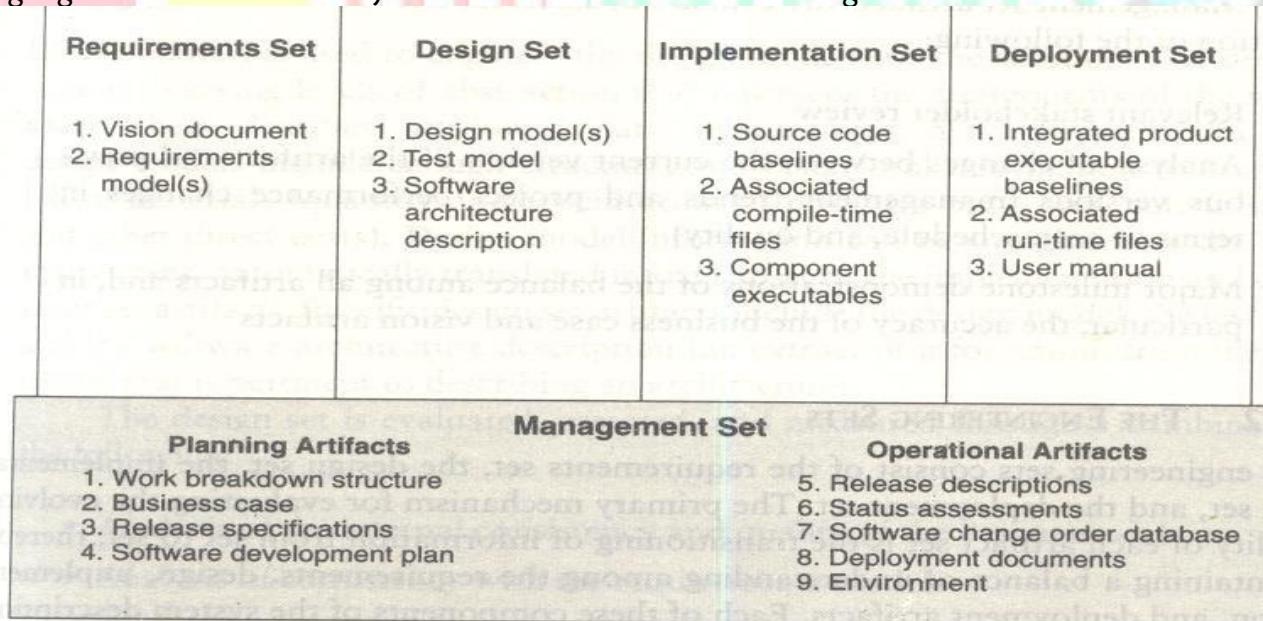


FIGURE 6-1. *Overview of the artifact sets*

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.

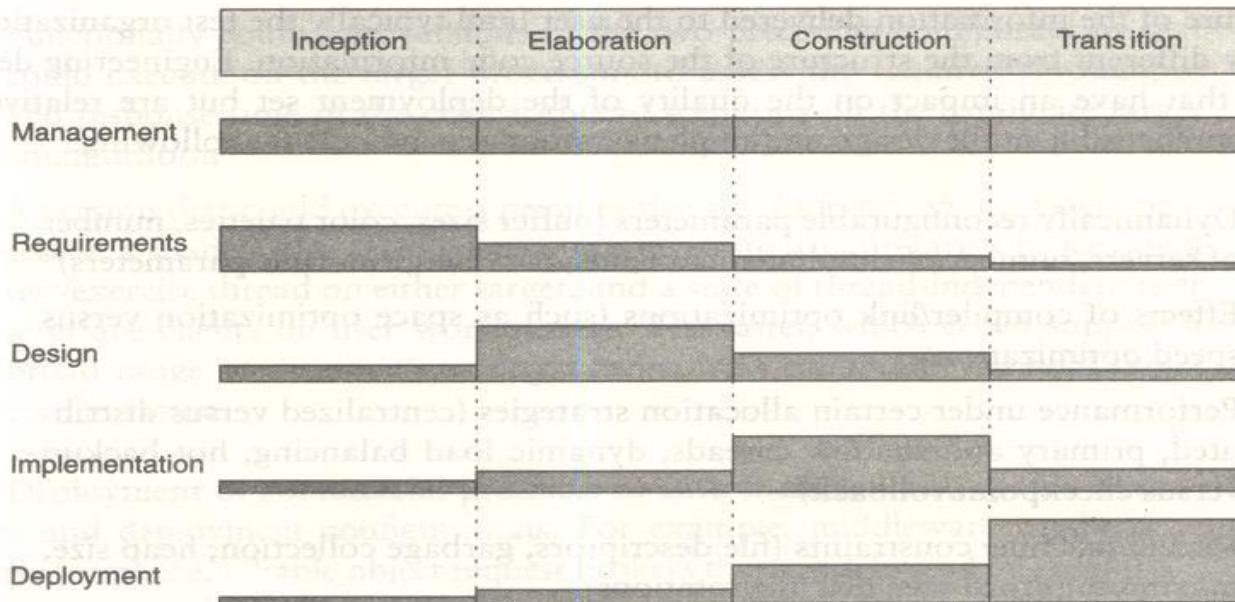


FIGURE 6-2. Life-cycle focus on artifact sets

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.

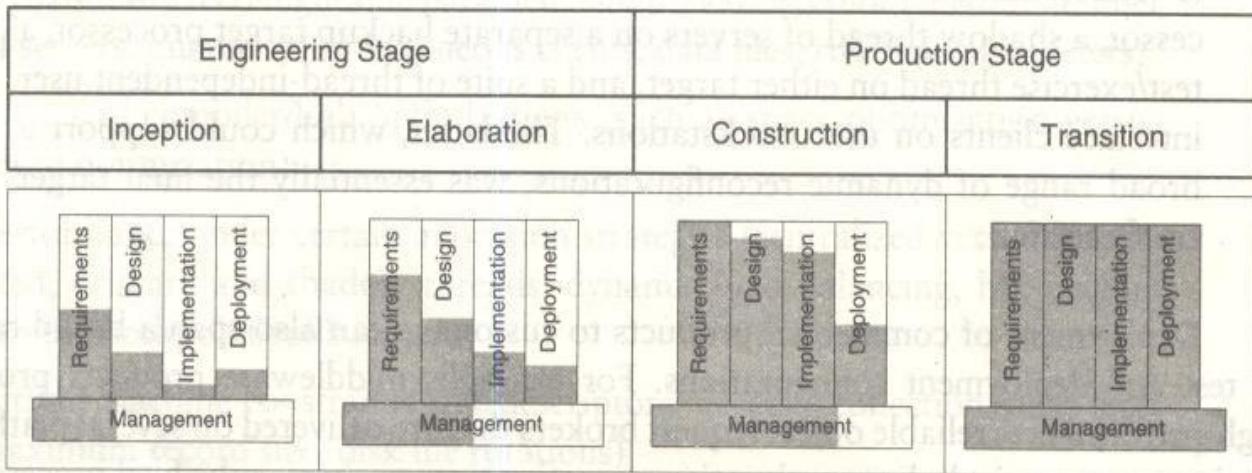


FIGURE 6-3. Life-cycle evolution of the artifact sets

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 manager must have insight into project

costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

Software Change Order Database

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

Release Specifications

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

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

FIGURE 6-6. Typical release specification outline

Release Descriptions

Release description documents describe the results of each release, including performance against each of the evaluation criteria in the corresponding release specification. Release baselines should be accompanied by a release description document that describes the evaluation criteria for that configuration baseline and provides substantiation (through demonstration, testing, inspection, or analysis) that each criterion has been addressed in an acceptable manner. Figure 6-7 provides a default outline for a release description.

Status Assessments

Status assessments provide periodic snapshots of project health and status, including the software project manager's risk assessment, quality indicators, and management indicators. Typical status assessments should include a review of resources, personnel staffing, financial data (cost and revenue), top 10 risks, technical progress (metrics snapshots), major milestone plans and results, total project or product scope & action items

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

FIGURE 6-7. Typical release description outline

Environment

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

Deployment

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

Management Artifact Sequences

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

△ Informal version

▲ Controlled baseline

Inception	Elaboration			Construction			Transition
Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	

Management Set

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

Requirements Set

1. Vision document ▲
2. Requirements model(s) ▲

Design Set

1. Design model(s) △
2. Test model △
3. Architecture description △

Implementation Set

1. Source code baselines ▲
2. Associated compile-time files ▲
3. Component executables ▲

Deployment Set

1. Integrated product-executable baselines ▲
2. Associated run-time files ▲
3. User manual ▲

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.

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

FIGURE 6-9. Typical vision document outline

Architecture Description

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

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

FIGURE 6-10. Typical architecture description outline

Software User Manual

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

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



UNIT - IV

Model based software architectures: A Management perspective and technical perspective.

Work Flows of the process: Software process workflows, Iteration workflows.

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

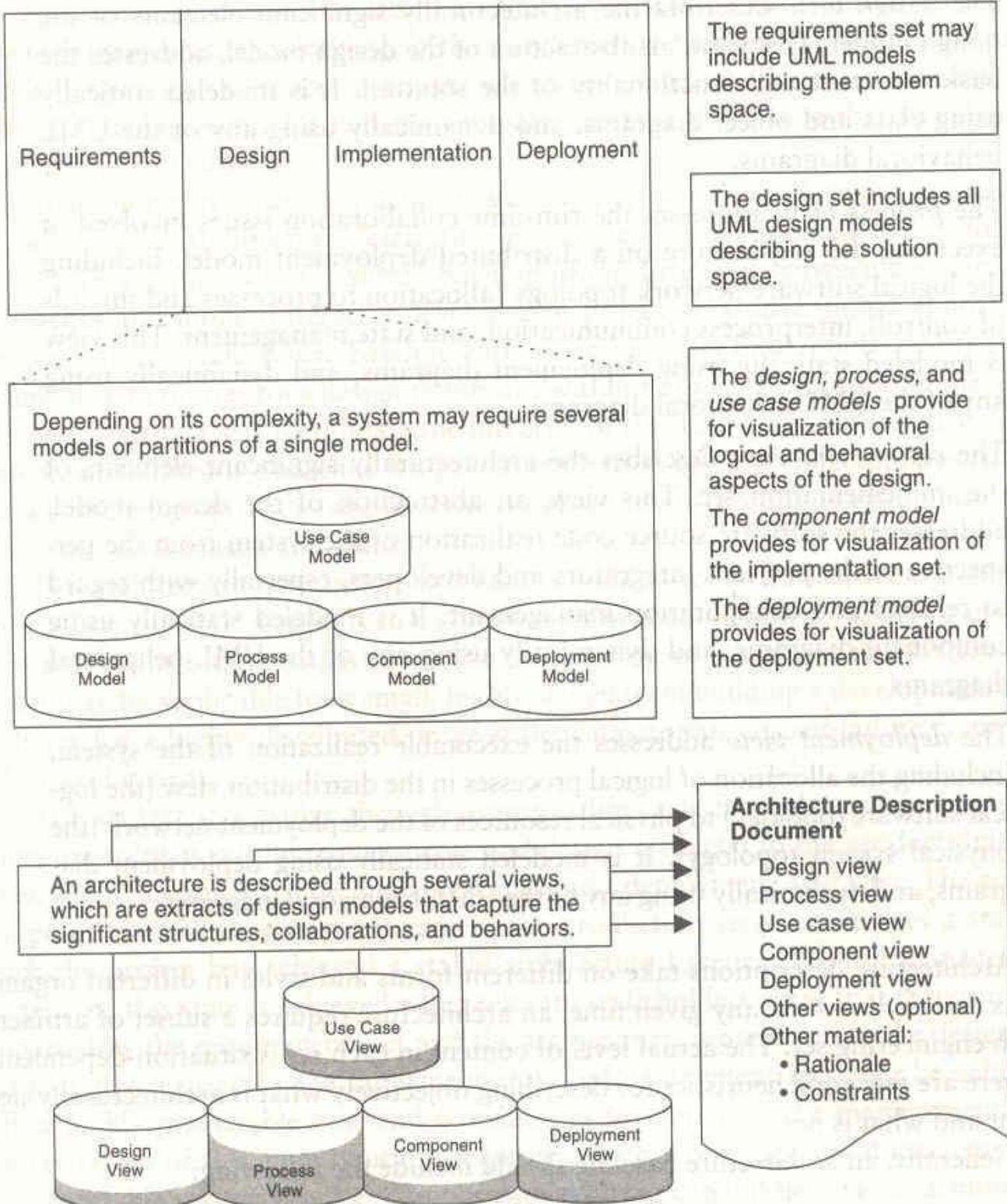


FIGURE 7-1. Architecture, an organized and abstracted view into the design models

8. Workflow of the process

8.1 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.

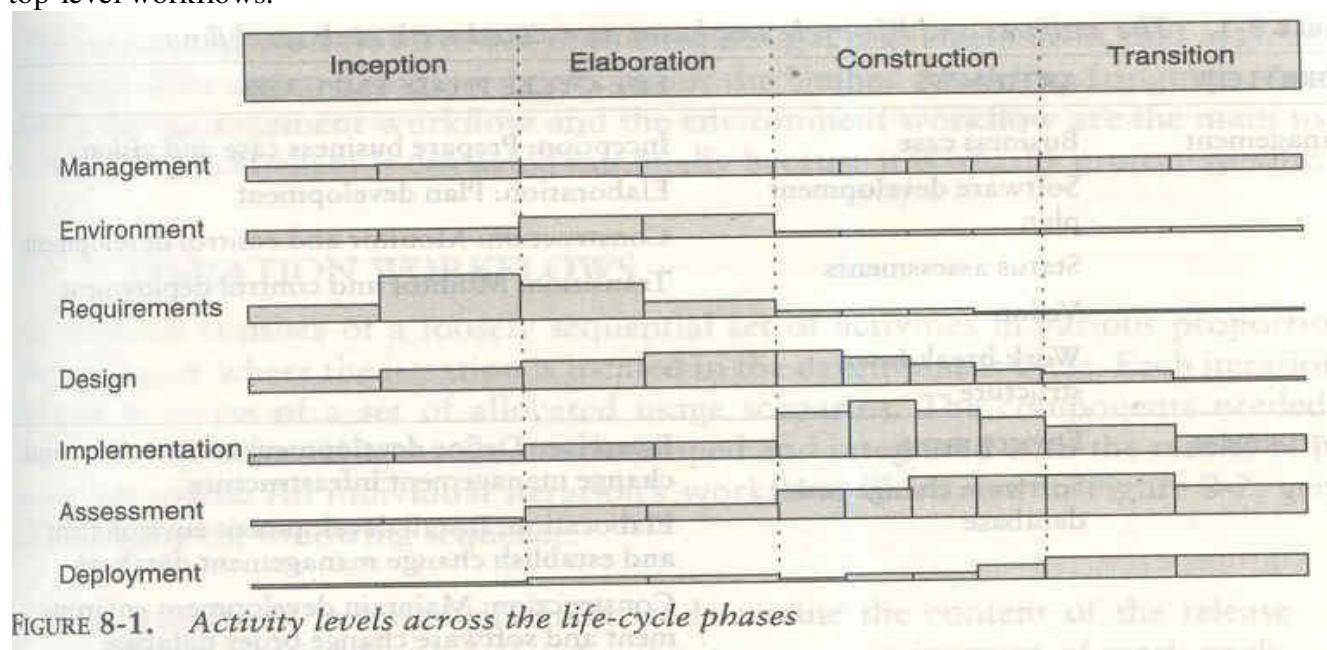


FIGURE 8-1. *Activity levels across the life-cycle phases*

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

8.2 ITERATION WORKFLOWS

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

- Management: iteration planning to determine the content of the release and develop the detailed plan for the iteration; assignment of work packages, or tasks, to the development team
- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test, and environment components

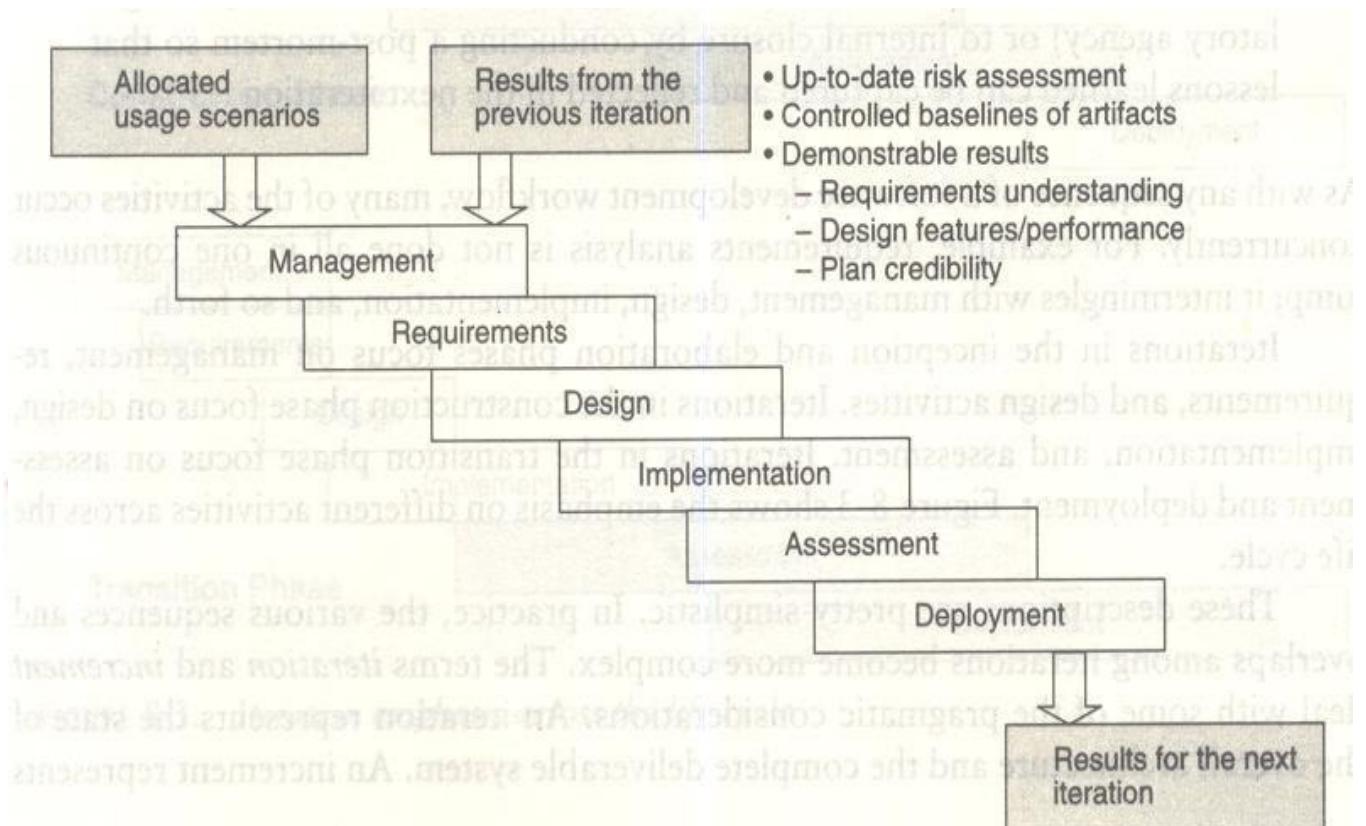


FIGURE 8-2. *The workflow of an iteration*

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities
- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities
- Implementation: developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions)

- Assessment: evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; identifying any rework required and determining whether it should be performed before deployment of this release or allocated to the next release; assessing results to improve the basis of the subsequent iteration's plan
- Deployment: transitioning the release either to an external organization (such as a user, independent verification and validation contractor, or regulatory agency) or to internal closure by conducting a post-mortem so that lessons learned can be captured and reflected in the next iteration

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

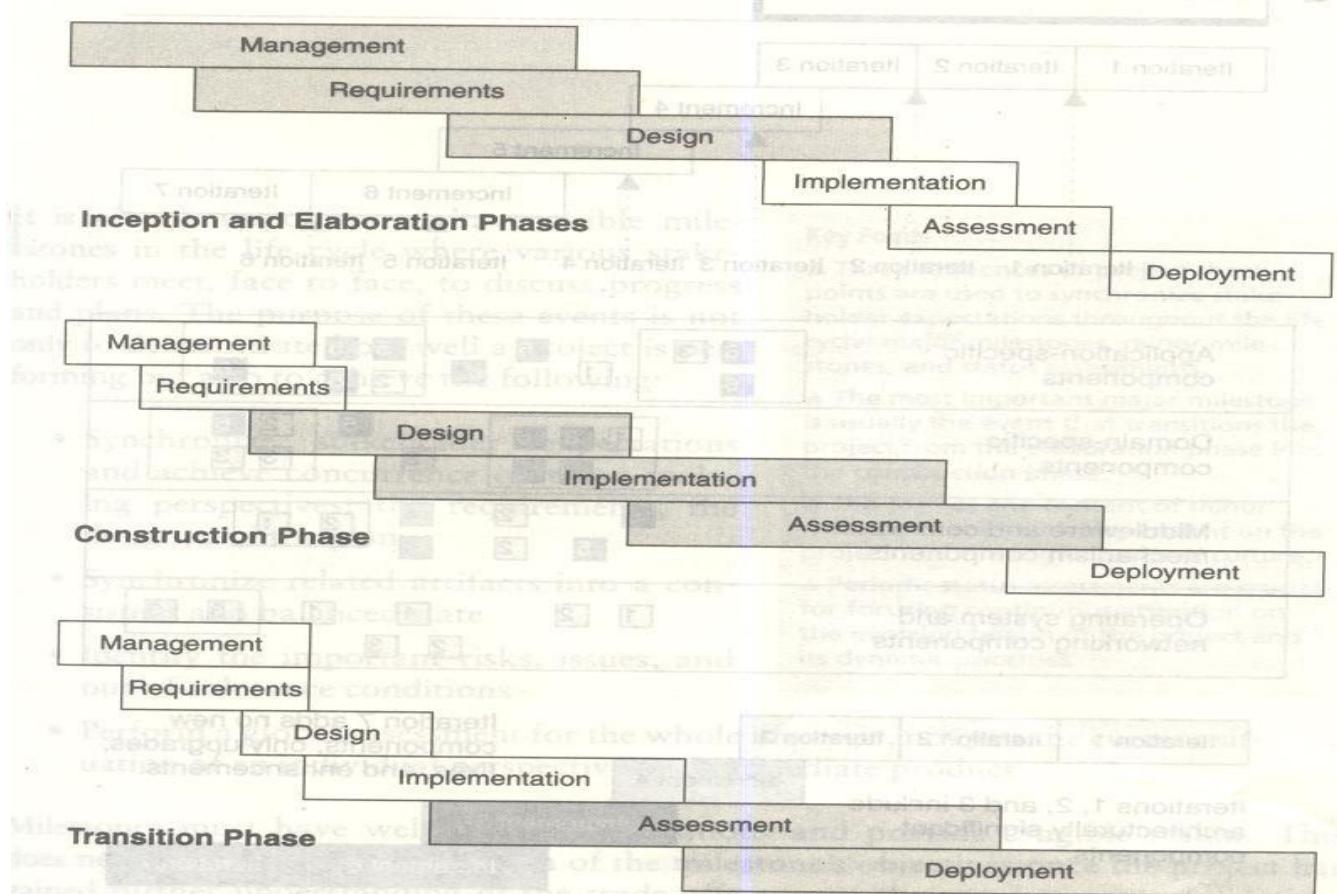


FIGURE 8-3. Iteration emphasis across the life cycle

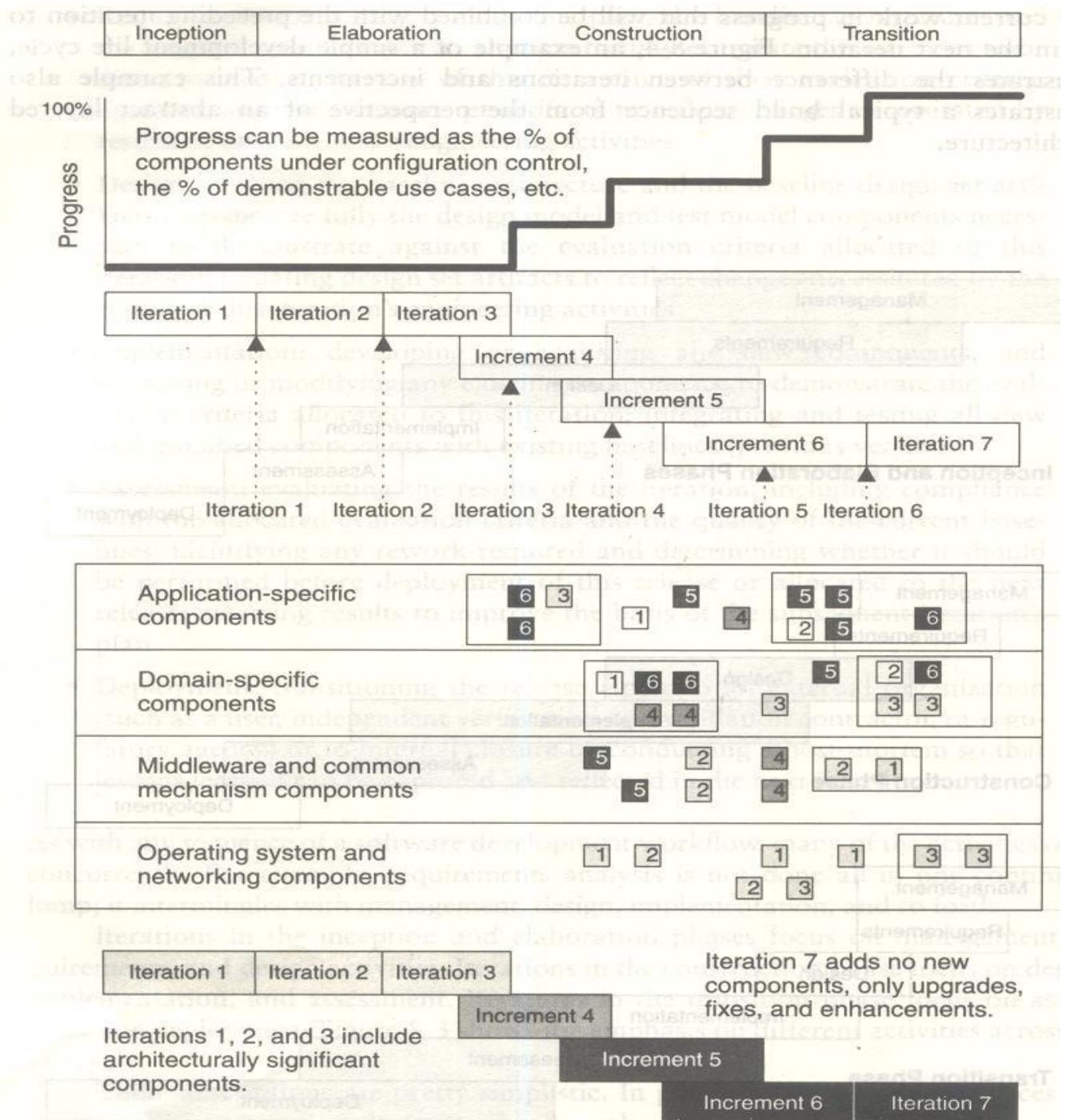


FIGURE 8-4. A typical build sequence associated with a layered architecture

UNIT - V

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.

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.

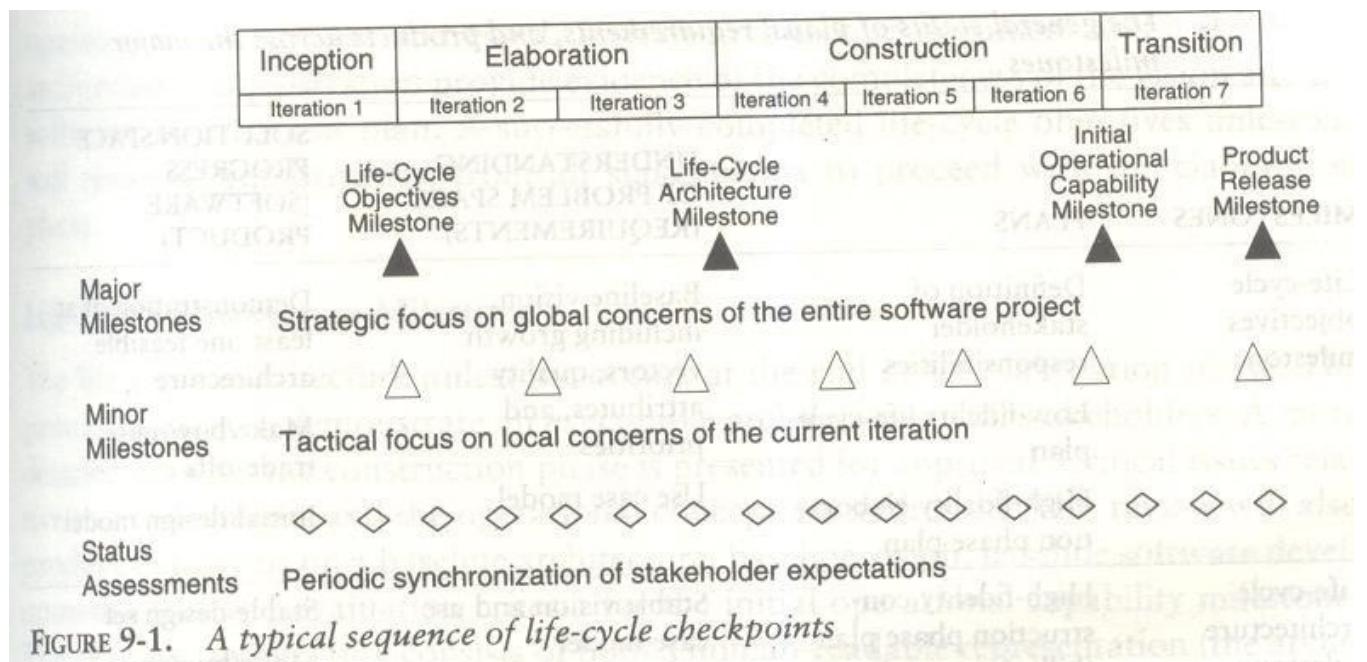


FIGURE 9-1. A typical sequence of life-cycle checkpoints

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

Life-Cycle Objectives Milestone

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

Life-Cycle Architecture Milestone

The life-cycle architecture milestone occurs at the end of the elaboration phase. The primary goal is to demonstrate an executable architecture to all stakeholders. The baseline architecture consists of both a human-readable representation (the architecture document) and a configuration-controlled set of software components captured in the engineering artifacts. A successfully completed life-cycle architecture milestone will result in authorization from the stakeholders to proceed with the construction phase.

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

I. Requirements

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

II. Architecture

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

III. Source and executable libraries

- A. Product components
- B. Test components
- C. Environment and tool components

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



Presentation Agenda

- I. **Scope and objectives**
 - A. Demonstration overview
- II. **Requirements assessment**
 - A. Project vision and use cases
 - B. Primary scenarios and evaluation criteria
- III. **Architecture assessment**
 - A. Progress
 - 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.

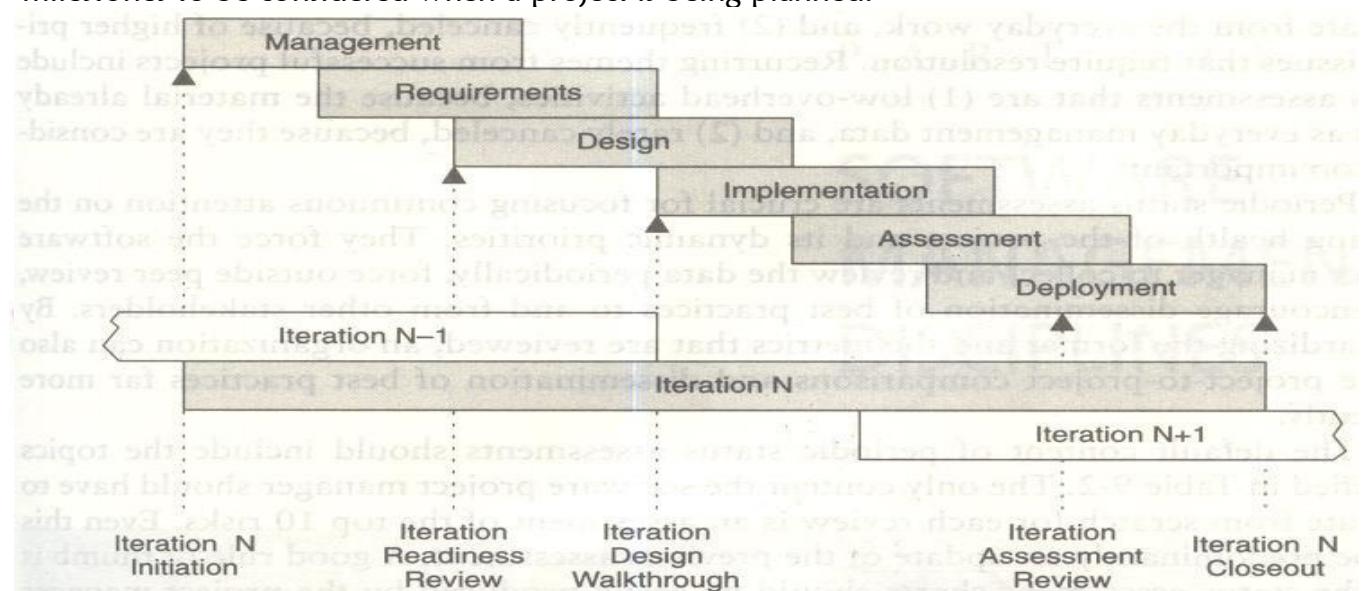


FIGURE 9-4. Typical minor milestones in the life cycle of an iteration

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.

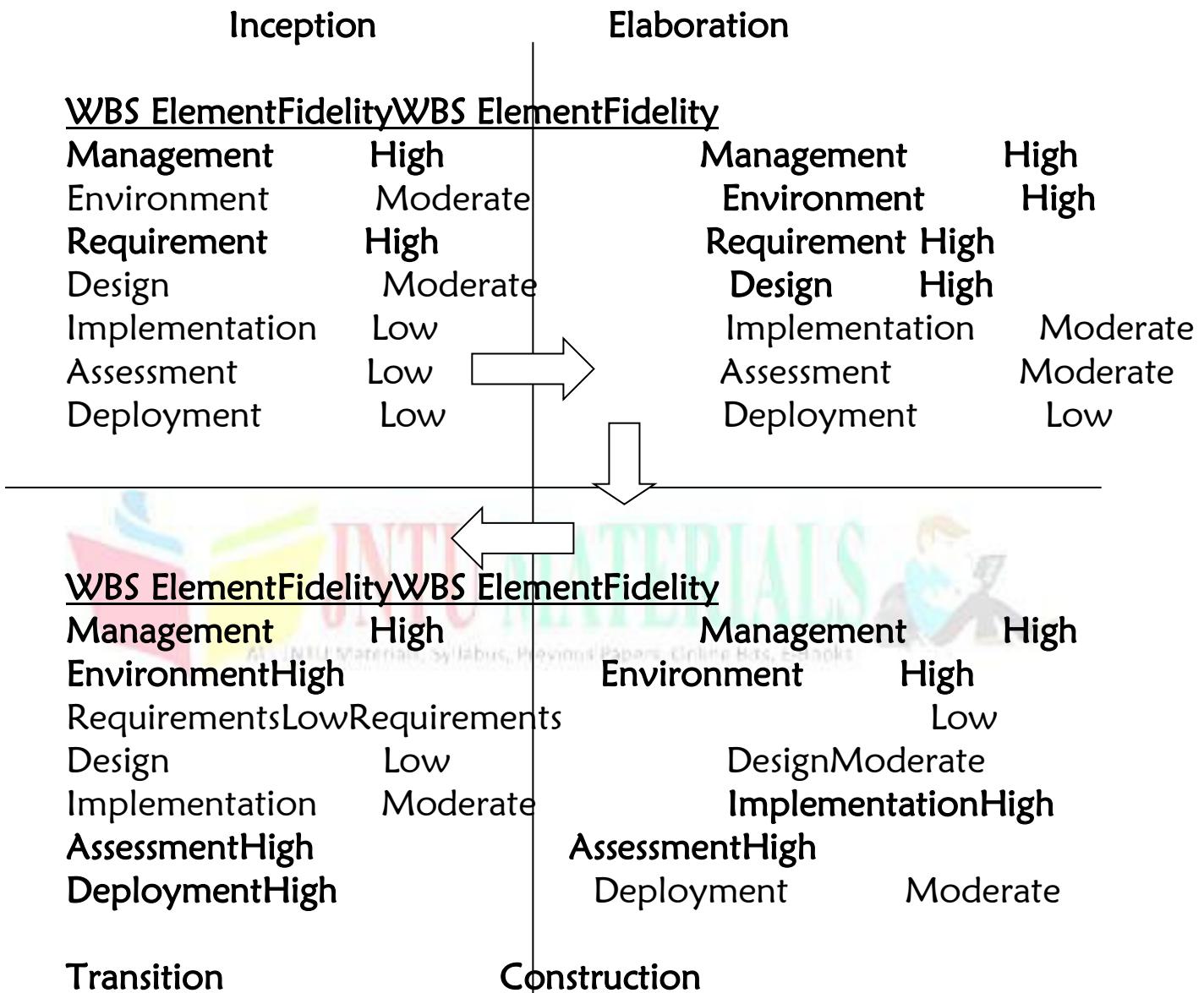
Another important attribute of a good WBS is that the planning fidelity inherent in each element is commensurate with the current life-cycle phase and project state. Figure 10-3 illustrates this idea. One of the primary reasons for organizing the default WBS the way I have is to allow for planning elements 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 be adopted blindly without being adapted to specific project circumstances. Two simple planning guidelines should be considered when a project plan is being initiated or assessed. The first guideline, detailed in Table 10-1, prescribes a default allocation of costs among the first-level WBS elements. The second guideline, detailed in Table 10-2, prescribes the allocation of effort and schedule across the lifecycle phases.

10-1 Web budgeting defaults

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

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

Domain	Inception	Elaboration	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 mile-

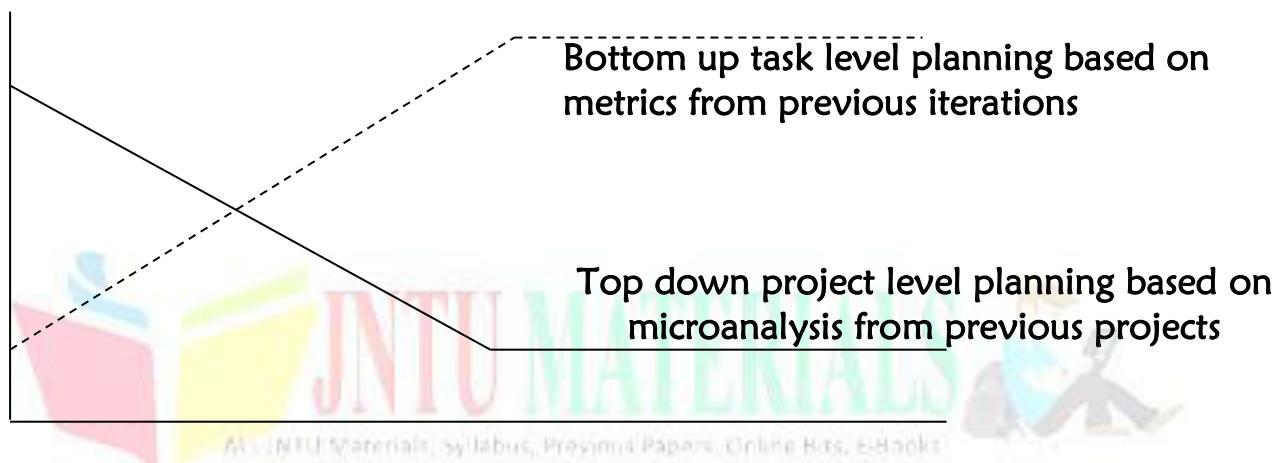
stones.

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 - VI

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.

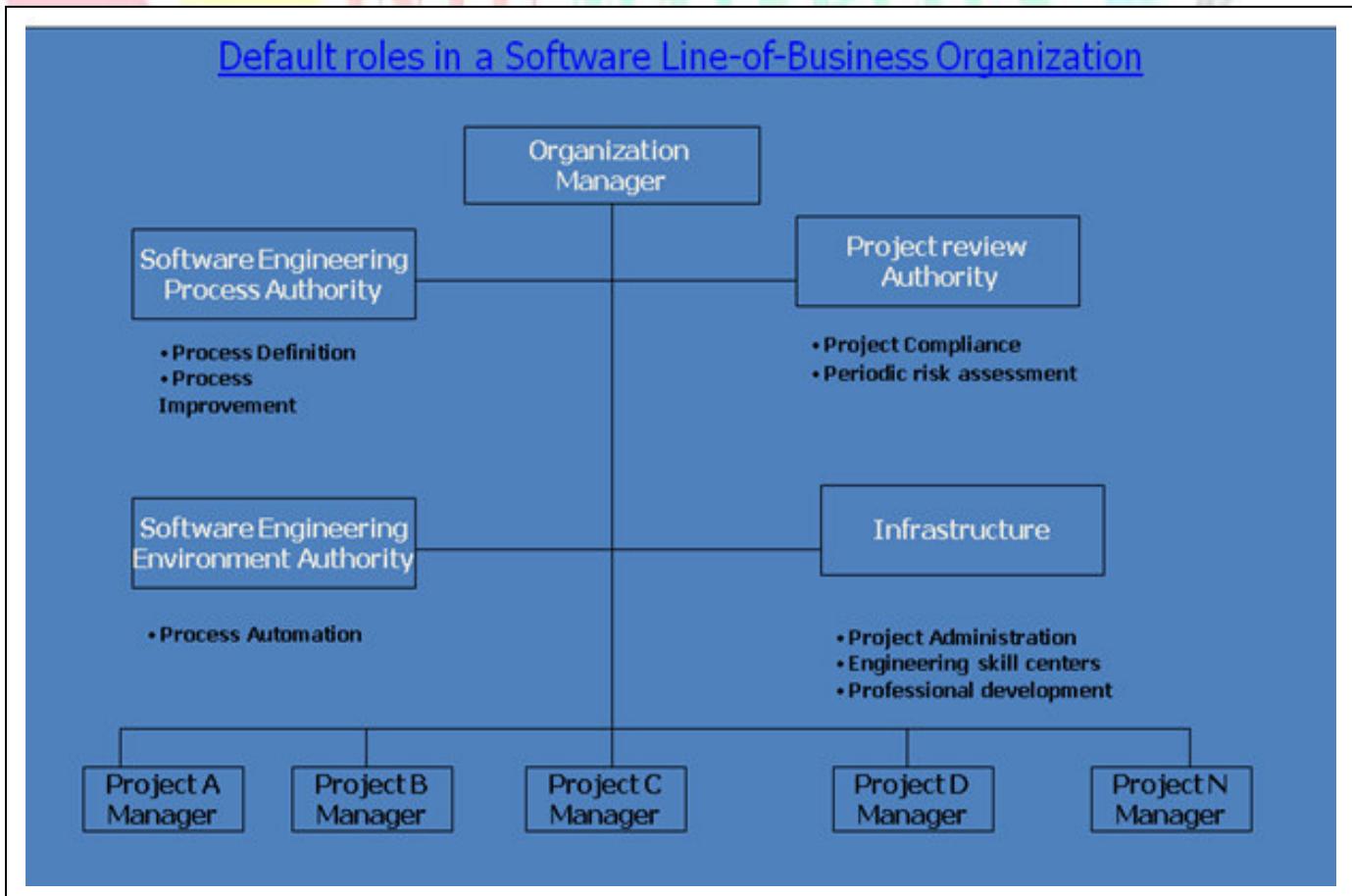


Fig: Default roles in a software Line-of-Business Organization.

Software Engineering Process Authority (SEPA)

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

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

Project Review Authority (PRA)

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

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

Software Engineering Environment Authority(SEEA)

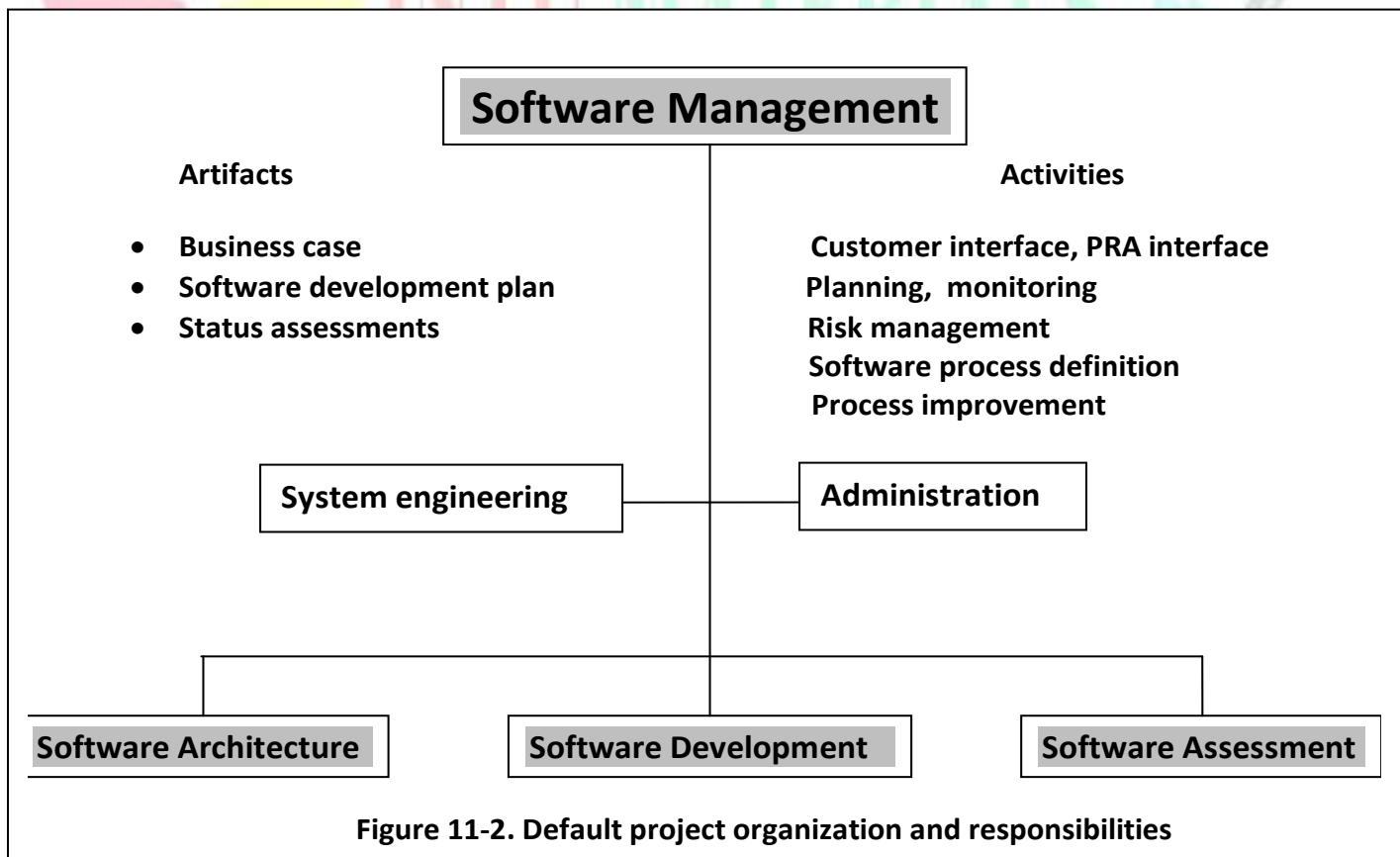
The SSEEA is responsible for automating the organization's process, maintaining the organization's standard environment, Training projects to use the environment & maintaining organization-wide reusable assets

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

Infrastructure

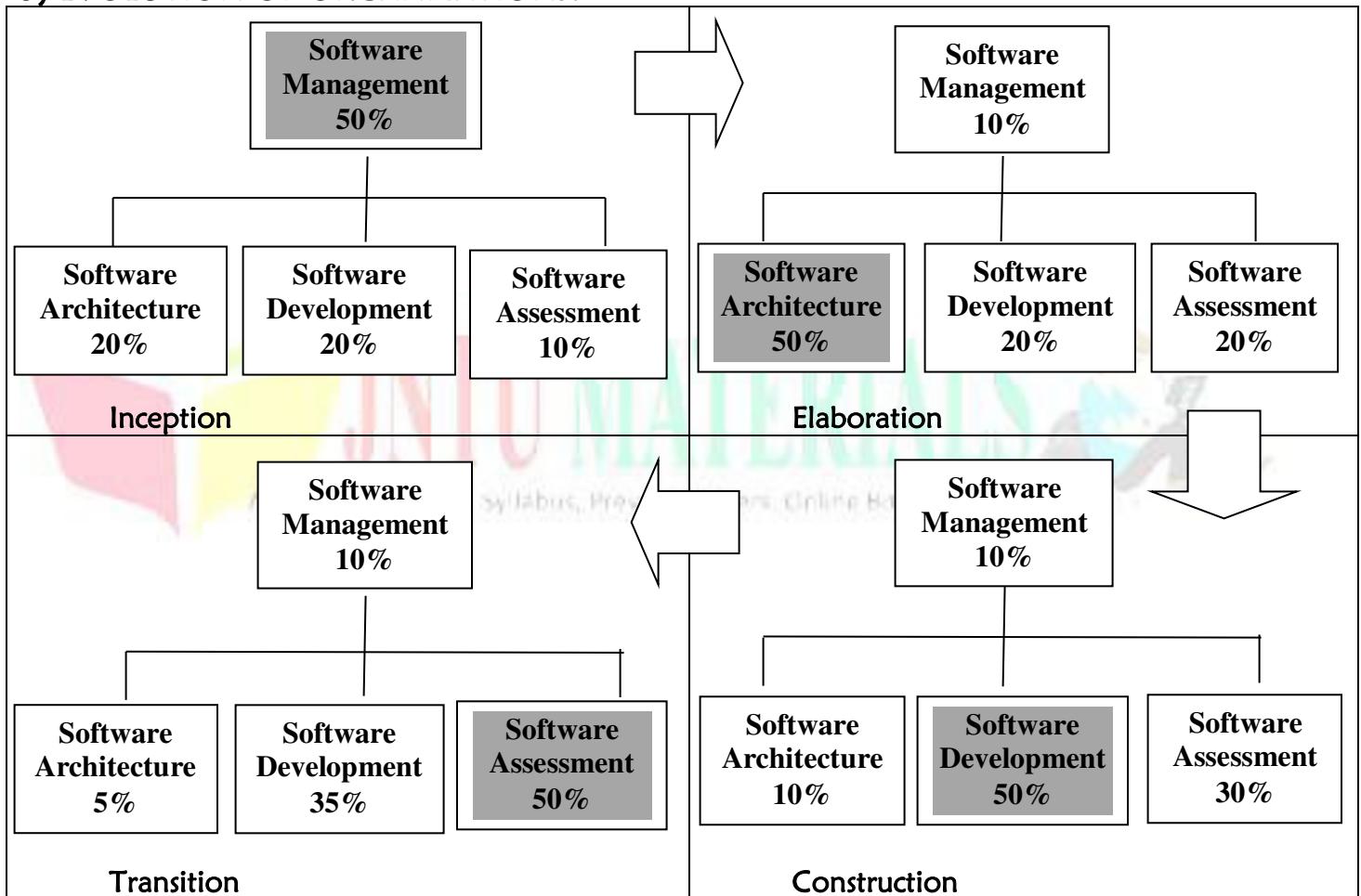
An organization's infrastructure provides human resources support, project-independent research & development, & other capital software engineering assets.

2) Project organizations:



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

3) EVOLUTION OF ORGANIZATIONS:



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

Software development: 50% Software Assessment (measurement/evaluation):30%	Software development: 35% Software Assessment (measurement/evaluation):50%
--	--

The Process Automation:

Introductory Remarks:

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

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

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

Metric automation is crucial to effective project control.

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

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

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

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

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

Tools: Automation Building blocks:

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

	<u>Workflows</u>	<u>Environment Tools & process Automation</u>
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

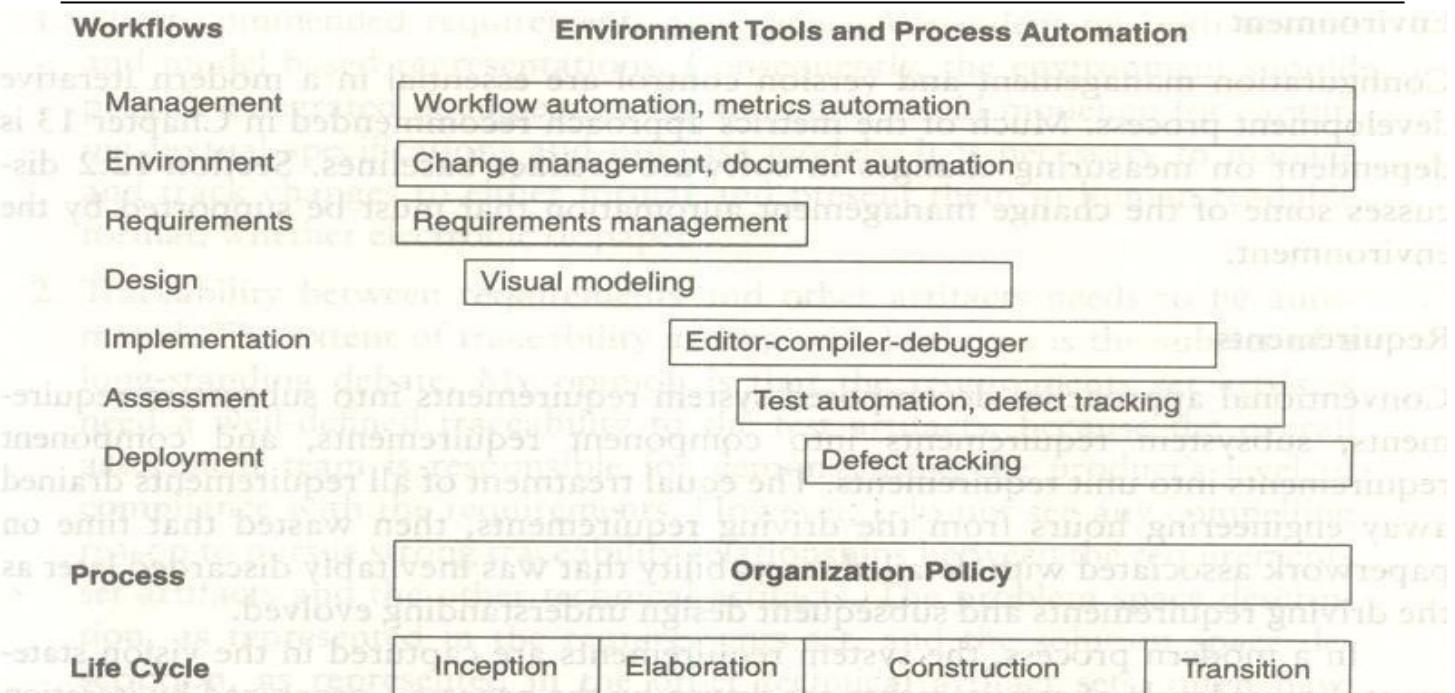


FIGURE 12-1. Typical automation and tool components that support the process workflows

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.

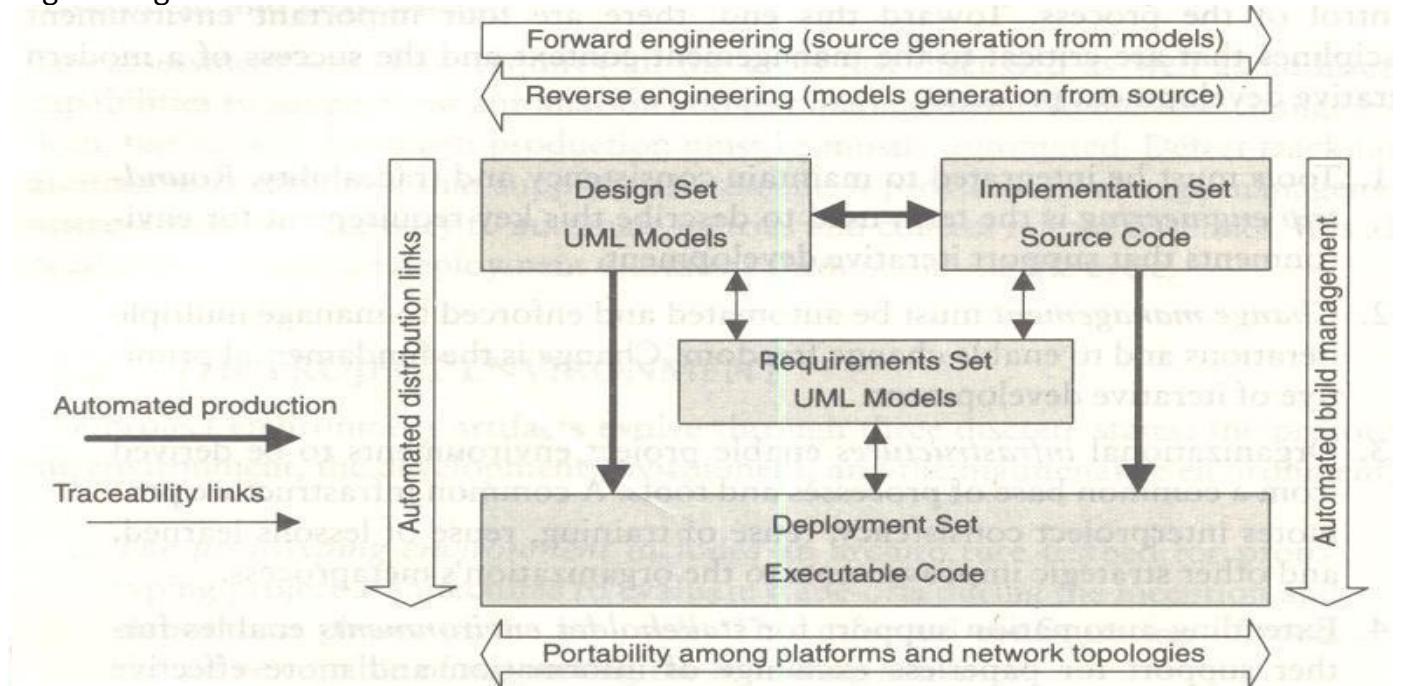


FIGURE 12-2. Round-trip engineering

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

Title:

Description	Name: _____	Date: _____
	Project: _____	
Metrics	Category: _____ (0/1 error, 2 enhancement, 3 new feature, 4 other)	
Initial Estimate	Actual Rework Expended	
Breakage: _____	Analysis: _____	Test: _____
Rework: _____	Implement: _____	Document: _____
Resolution	Analyst: _____	
	Software Component: _____	
Assessment	Method: _____ (inspection, analysis, demonstration, test)	
	Tester: _____ Platforms: _____ Date: _____	
Disposition	State: _____ Release: _____ Priority: _____	
	Acceptance: _____ Date: _____	
	Closure: _____ Date: _____	

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

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. Appendices**
 - 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 as 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

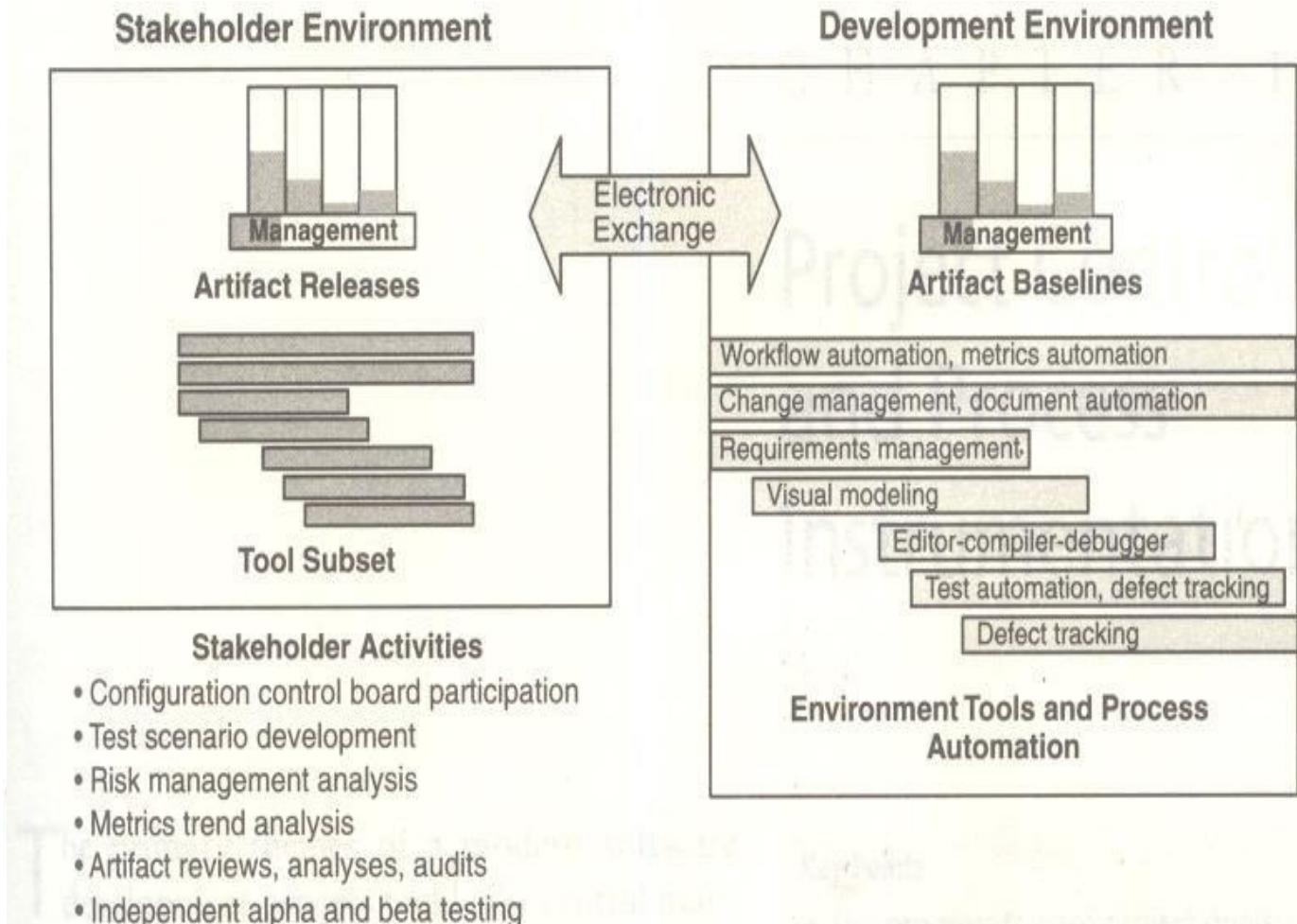


FIGURE 12-6. Extending environments into stakeholder domains

UNIT-V:

Agile Methodology, adapting to Scrum, Patterns for Adopting Scrum, Iterating towards Agility.

Fundamentals of DevOps: Architecture, Deployments, Orchestration, Need, Instance of applications, DevOps delivery pipeline, DevOps eco system. DevOps adoption in projects: Technology aspects, Agiling capabilities, Tool stack implementation, People aspect, processes

What is DevOps?

DevOps is a collaboration between Development and IT Operations to make software production and Deployment in an automated & repeatable way. DevOps helps increase the organization's speed to deliver software applications and services. The full form of 'DevOps' is a combination of 'Development' and 'Operations.' It allows organizations to serve their customers better and compete more strongly in the market. In simple words, DevOps can be defined as an alignment of development and IT operations with better communication and collaboration.

Why is DevOps Needed?

Before DevOps, the development and operation team worked in complete isolation.

- Testing and Deployment were isolated activities done after design-build. Hence they consumed more time than actual build cycles. Without using DevOps, team members spend a large amount of their time testing, deploying, and designing instead of building the project. Manual code deployment leads to human errors in production. Coding & operation teams have separate timelines and are not synch, causing further delays.

What is DevOps?



Explain Agile Methodology in devops indetail

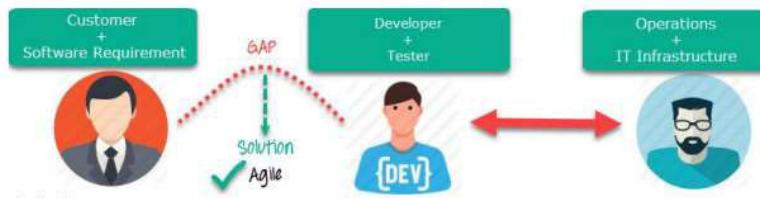
Agile methodology is a set of principles and practices for software development that prioritizes flexibility, collaboration, and customer satisfaction. It is widely used in the field of DevOps, which is a set of practices that aims to improve collaboration and communication between software development and IT operations teams. Agile and DevOps work together to deliver high-quality software quickly and efficiently.

How is DevOps different from Agile? DevOps Vs Agile

Stakeholders and communication chain a typical IT process.



Agile addresses gaps in Customer and Developer communications



Agile Process
DevOps addresses gaps in Developer and IT Operations communications



DevOps Process

Here are the key principles and practices of Agile methodology in the context of DevOps:

1. Iterative and Incremental Development:

- Agile promotes the development of software in small, incremental releases.
- Each release adds new features or improvements to the existing functionality.
- This approach allows for quick adaptation to changing requirements and provides regular opportunities for feedback.

2. Customer Collaboration:

- Agile emphasizes frequent and open communication with customers and stakeholders.
- Customers are involved throughout the development process, providing feedback and prioritizing features based on business value.

3. Cross-functional Teams:

- Agile promotes the formation of cross-functional teams that include members with different skills, such as developers, testers, and operations personnel.
- This encourages collaboration and shared responsibility for the entire software development lifecycle.

4. Adaptability and Flexibility:

- Agile acknowledges that requirements may change, and it values the ability to respond to change quickly.
- Teams are encouraged to embrace change and adapt their plans accordingly.

5. Continuous Integration (CI):

- CI is a DevOps practice that involves frequently integrating code changes into a shared repository.
- Agile teams use CI to ensure that code changes are tested and integrated regularly, reducing the likelihood of integration issues.

6. Continuous Delivery (CD):

- CD is the practice of delivering software in small, frequent increments.
- Agile teams aim for CD by automating the build, test, and deployment processes to ensure a reliable and efficient release pipeline.

7. Sprints and Time-Boxing:

- Agile projects are typically organized into fixed-length iterations called sprints.
- Sprints are time-boxed, meaning they have a predefined duration (e.g., 2 weeks), providing a regular cadence for planning, development, and review.

8. User Stories and Backlog:

- Agile teams use user stories to capture functional requirements from the user's perspective.
- A backlog is a prioritized list of user stories and other work items, and the team pulls items from the backlog into each sprint.

Difference between DevOps and Agile

Agile	DevOps
Emphasize breaking down barriers between developers and management.	DevOps is about software deployment and operation teams.
Addresses gaps between customer requirements and development teams.	Addresses the gap between the development and Operation team
Focuses more on functional and non-functional readiness	It focuses on operational and business readiness.
Agile development pertains mainly to the company's way development is thought out.	DevOps emphasises deploying software in the most reliable and safest ways that aren't always the fastest.
Agile development emphasises training all team members to have varieties of similar and equal skills. So that, when something goes wrong, any team member can get assistance from any member in the absence of the team leader.	DevOps likes to divide and conquer, spreading the skill set between the development and operation teams. It also maintains consistent communication.
Agile development manages on "sprints". It means that the timetable is much shorter (less than a month), and several features are to be produced and released in that period.	DevOps strives for consolidated deadlines and benchmarks with significant releases rather than smaller and more frequent ones.

By combining Agile principles with DevOps practices, teams can create a culture of collaboration, continuous improvement, and rapid delivery, ultimately leading to more responsive and effective software development processes.

What is scrum. how adapting to Scrum and Patterns for Adopting Scrum

Scrum is an Agile framework for managing and organizing work, particularly in software development. It is based on the principles of transparency, inspection, and adaptation, and it provides a structured yet flexible approach to product development. Scrum is widely used in the software industry but has also found application in various other domains.

Here are key components of the Scrum framework:

1. Scrum Roles:

- **Product Owner:** Represents the stakeholders and is responsible for defining the product backlog and prioritizing it.
- **Scrum Master:** Facilitates the Scrum process, helps the team stay focused, and removes impediments.
- **Development Team:** Cross-functional team responsible for delivering a potentially shippable product increment in each sprint.

2. Scrum Artifacts:

- **Product Backlog:** An ordered list of all features, enhancements, and bug fixes that constitute the project's roadmap.
- **Sprint Backlog:** A subset of the product backlog selected for a specific sprint.
- **Product Increment:** The sum of all the completed items from the product backlog during a sprint.

3. Scrum Events:

- **Sprint:** A time-boxed period (typically 2-4 weeks) during which a potentially releasable product increment is created.
- **Sprint Planning:** A meeting where the team plans the work for the upcoming sprint.
- **Daily Scrum:** A daily stand-up meeting where the team discusses progress, challenges, and plans.
- **Sprint Review:** A meeting at the end of the sprint to inspect and adapt the product increment.
- **Sprint Retrospective:** A meeting at the end of the sprint for the team to reflect on their processes and identify areas for improvement.

Adapting to Scrum:

1. Training and Education:

- Provide training to team members, stakeholders, and leadership to ensure a shared understanding of Scrum principles and practices.

2. Scrum Team Formation:

- Form cross-functional, self-organizing Scrum teams with the necessary skills to deliver a potentially shippable product increment.

3. Define the Product Backlog:

- Work with the Product Owner to create and prioritize a product backlog, capturing all features, enhancements, and bug fixes.

4. Implement Sprints:

- Introduce time-boxed sprints with a consistent duration, and conduct sprint planning meetings to define the sprint goal and select items from the product backlog.

5. Daily Scrum and Collaboration:

- Establish a daily Scrum meeting for the team to synchronize and collaborate on progress and challenges.

6. Sprint Review and Retrospective:

- Conduct sprint reviews and retrospectives to inspect and adapt both the product and the team's processes.

7. Continuous Improvement:

- Foster a culture of continuous improvement, encouraging the team to identify and implement changes that enhance productivity and product quality.

Patterns for Adopting Scrum:

1. Start Small:

- Begin with a small, pilot Scrum team to learn the framework and adapt it to the organization's needs.

2. Inspect and Adapt:

- Regularly inspect the team's progress, adapt practices as needed, and foster a culture of continuous improvement.

3. Empower Teams:

- Empower teams to self-organize and make decisions, promoting a sense of ownership and responsibility.

4. Collaboration and Communication:

- Emphasize collaboration and open communication among team members, stakeholders, and leadership.

5. Remove Barriers:

- Identify and remove organizational barriers that hinder the team's ability to deliver value.

6. Feedback Loops:

- Establish feedback loops at all levels to quickly identify and address issues.

Adopting Scrum is often an iterative process, and organizations may need to tailor the framework to fit their unique context. Patterns for adopting Scrum provide guidance and best practices to facilitate a smoother transition and maximize the benefits of the Agile framework.

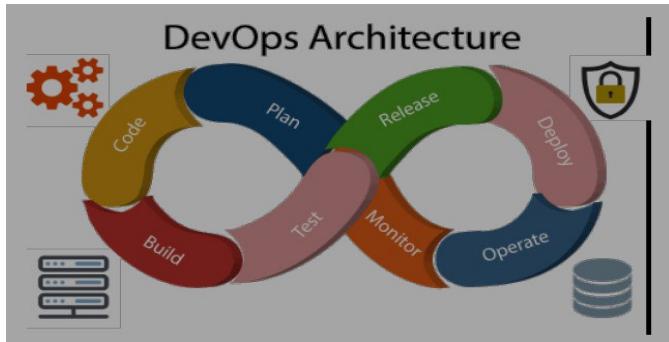
Explain Iterating towards Agility in DEVOPS

"Iterating towards Agility" in the context of DevOps refers to the incremental and continuous improvement process that organizations undertake to adopt and enhance their Agile and DevOps practices. It involves a series of iterations or cycles where teams assess their current state, identify areas for improvement, and implement changes to become more agile and efficient in their software development and delivery processes. Here are key aspects of iterating towards agility in DevOps:

1. Continuous Learning:
<ul style="list-style-type: none"> • Teams engage in continuous learning and reflection to understand the strengths and weaknesses of their current processes. • Learning from both successes and failures is crucial for adapting and evolving towards more effective practices.
2. Regular Assessments:
<ul style="list-style-type: none"> • Conduct regular assessments of the existing DevOps processes, tools, and collaboration practices. • Evaluate the effectiveness of the current state and identify areas that need improvement.
3. Feedback Loops:
<ul style="list-style-type: none"> • Establish and strengthen feedback loops at various stages of the software development lifecycle. • Use feedback from customers, stakeholders, and team members to identify opportunities for enhancement.
4. Incremental Changes:
<ul style="list-style-type: none"> • Implement changes incrementally rather than attempting large-scale transformations. • Break down improvements into manageable pieces to reduce the risk and allow for quicker adjustments.
5. Cross-Functional Collaboration:
<ul style="list-style-type: none"> • Foster collaboration between development, operations, and other relevant teams. • Encourage cross-functional teams to work together seamlessly, breaking down silos and improving communication.
6. Automated Testing and Continuous Integration:
<ul style="list-style-type: none"> • Implement or enhance automated testing and continuous integration practices to ensure the rapid and reliable delivery of software. • Regularly inspect and adapt the CI/CD (Continuous Integration/Continuous Delivery) pipeline for efficiency.
7. DevOps Tools and Technologies:
<ul style="list-style-type: none"> • Regularly evaluate and update DevOps tools and technologies to stay current with industry best practices. • Ensure that tools support collaboration, automation, and integration across the entire development and operations lifecycle.
8. Agile Principles:
<ul style="list-style-type: none"> • Embrace Agile principles such as iterative development, customer collaboration, and responding to change. • Apply Agile practices like Scrum or Kanban to enhance team collaboration and responsiveness.
9. Culture of Continuous Improvement:
<ul style="list-style-type: none"> • Cultivate a culture of continuous improvement where teams are encouraged to experiment, learn, and share their experiences. • Recognize and celebrate achievements, fostering a positive environment for change.
10. Adaptability:
<ul style="list-style-type: none"> • Be adaptable and responsive to changing business needs, customer feedback, and industry trends. • Adjust processes, priorities, and practices based on the evolving requirements of the organization.
11. Measure and Monitor:
<ul style="list-style-type: none"> • Define key performance indicators (KPIs) and metrics to measure the effectiveness of DevOps practices. • Regularly monitor and analyze these metrics to identify areas for improvement.
12. Training and Skill Development:
<ul style="list-style-type: none"> • Invest in training and skill development for team members to ensure they have the necessary knowledge and expertise to excel in their roles. • Encourage a culture of learning and knowledge sharing within the organization.

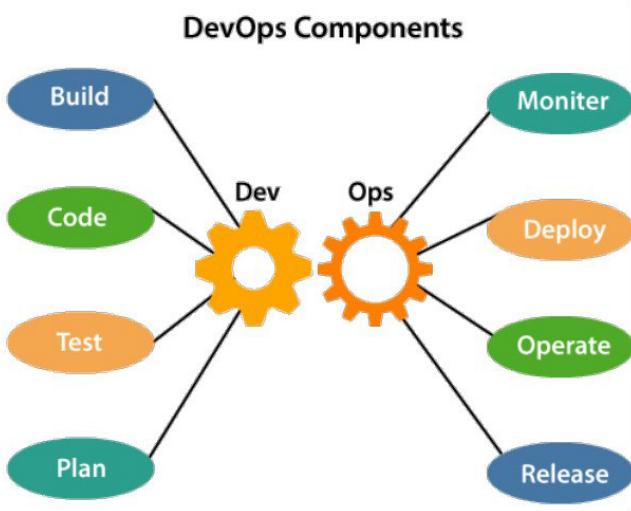
Iterating towards agility is not a one-time event; it is an ongoing journey that requires commitment, collaboration, and a willingness to adapt. By continuously assessing, learning, and making incremental improvements, organizations can achieve greater agility and efficiency in their DevOps practices, ultimately delivering better value to customers.

Fundamentals of DevOps: Architecture



Development and operations both play essential roles in order to deliver applications. The deployment comprises analyzing the requirements, designing, developing, and testing of the software components or frameworks.

The operation consists of the administrative processes, services, and support for the software. When both the development and operations are combined with collaborating, then the DevOps architecture is the solution to fix the gap between deployment and operation terms; therefore, delivery can be faster. DevOps architecture is used for the applications hosted on the cloud platform and large distributed applications. Agile Development is used in the DevOps architecture so that integration and delivery can be contiguous. When the development and operations team works separately from each other, then it is time-consuming to design, test, and deploy. And if the terms are not in sync with each other, then it may cause a delay in the delivery. So DevOps enables the teams to change their shortcomings and increases productivity.



Below are the various components that are used in the DevOps architecture:

- 1) Build Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.
- 2) Code Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, getting notified about the reason behind the difference in the actual and the expected output, and if necessary reverting to the original code developed. The code can be appropriately arranged in files, folders, etc. And they can be reused.
- 3) Test The application will be ready for production after testing. In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.
- 4) Plan DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.
- 5) Monitor Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with services where the log data may get monitored through many third-party tools such as Splunk.
- 6) Deploy Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.
- 7) Operate DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.
- 8) Release Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering. Many processes involved in release management commonly used to do the deployment in the production environment manually to lessen the impact on the customers.

Features of DevOps Architecture Below are the key features of DevOps Architecture.

1. Automation Automation most effectively reduces the time consumption specifically during the testing and deployment phase. The productivity increases and releases are made quicker through automation with less issue as tests are executed more rigorously. This will lead to catching bugs sooner so that it can be fixed more easily. For continuous delivery, each code change is done through automated tests, through cloud-based services and builds. This promotes production using automated deploys.
2. Collaboration The Development and Operations team collaborates together as DevOps team which improves the cultural model as the teams become more effective with their productivity which strengthens accountability and ownership. The teams share their responsibilities and work closely in sync which in turn makes the deployment to production faster.
3. Integration Applications need to be integrated with other components in the environment. The integration phase is where the existing code is integrated with new functionality and then testing takes place. Continuous integration and testing enable continuous development. The frequency in the releases and micro-services lead to significant operational challenges. To overcome such challenges, continuous integration and delivery are implemented to deliver in a quicker, safer and reliable manner.

4. Configuration Management This ensures that the application only interacts with the resources concerned with the environment in which it runs. The configuration files are created where the configuration external to the application is separated from the source code. The configuration file can be written while deployment or they can be loaded at the run time depending on the environment in which it is running.

DevOps Orchestration DevOps orchestration is a logical and necessary step for any DevOps shop that is in the process of, or has completed, implementing automation. Organizations generally start with a local solution and then, after achieving success, orchestrate their best practices through a technology that unifies connectivity into one solid process. That's because automation can only go so far in maximizing efficiency, so orchestration in DevOps is needed if you want to take your releases to the next level. To illustrate this concept, I'm going to discuss DevOps orchestration in general and how practices like DevOps provisioning orchestration and DevOps release orchestration can apply both on-site and in the cloud.

DevOps automation is a process by which a single, repeatable task, such as launching an app or changing a database entry, is made capable of running without human intervention, both on PCs and in the cloud. In comparison, DevOps orchestration is the automation of numerous tasks that run at the same time in a way that minimizes production issues and time to market. Automation applies to functions that are common to one area, such as launching a web server, or integrating a web app, or changing a database entry. But when all of these functions must work together, DevOps orchestration is required.

DevOps Orchestration – The Following Move after Automation DevOps orchestration involves automating multiple processes to reduce issues leading to the production date and shorten time to market. On the other hand, automation is used to perform tasks or a series of actions that are repetitive. DevOps orchestration is not simply putting separated tasks together, but it can do much more than that. DevOps orchestration streamlines the entire workflow by centralizing all tools used across teams, along with their data, to keep track of process and completion status throughout. Besides, automation can be pretty complicated at scale, although normally it is focused on a specific operation to achieve a goal, such as a server deployment. When automation has reached its limitations, that's when orchestration plays to its strengths.

Why Invest in DevOps Orchestration?

SIX REASONS TO INVEST IN DEVOPS ORCHESTRATION

1. Speed up the automation process
2. Enhance cross-functional collaboration
3. Release products with higher quality
4. Lower costs for IT infrastructure and human resources
5. Build transparency across the SDLC
6. Improve the release velocity

Investing in DevOps Orchestration helps you:

1. Speed up the automation process DevOps orchestration helps users deliver new builds into production quickly and seamlessly without putting much effort into these repetitive tasks. As a result, DevOps teams can focus on other critical projects and decisionmaking.
2. Enhance cross-functional collaboration DevOps orchestration is a platform that constantly unifies and updates all activities to improve communication between operation and development teams, so every member is in sync in all steps.
3. Release products with higher quality DevOps orchestration helps teams reduce software errors before reaching the end-user by controlling approvals, scheduling, security testing, automatic status reports.
4. Lower costs for IT infrastructure and human resources Another benefit of DevOps orchestration is cutting infrastructure investment costs and the number of IT employees. In the long run, businesses can also expand their cloud service footprint and allocate costs flexibly.
5. Build transparency across the SDLC Siloed tasks and information cause difficulty in creating clarity and openness throughout a project. DevOps orchestration helps businesses coordinate all tasks, centralize all operations' data, and allow key stakeholders to gain visibility into new updates and progress during the development lifecycle.
6. Improve the release velocity DevOps orchestration also requires considerable automation and the automated progression of software through testing and releasing pipeline processes. As businesses can reduce the time of finishing manual tasks and deliver the program to the upcoming step in the process, software reaches the end-user more quickly, wait time is turned aside to the next project. With that reason, a higher level of automation helps release products faster, save money and increase profits.

Applications of DevOps

1. Application of DevOps in the Online Financial Trading Company The methodology in the process of testing, building, and development was automated in the financial trading company. Using the DevOps, deployment was being done within 45 seconds. These deployments used to take long nights and weekends for the employees. The time of the overall process reduced and the interest of clients increased.
2. Use of DevOps in Network cycling Deployment, testing and rapid designing became ten times faster. It became effortless for the telco service provider to add patches of security every day, which used to be done only every three months. Through deployment and design, the new version of network cycling was being rolled out.
3. Application in Car Manufacturing Industries Using DevOps, employees helped car manufacturers to catch the error while scaling the production, which was not possible before.

4. Benefits to Airlines Industries With the benefit of DevOps, United Airlines saved \$500,000 by changing to continuous testing standards. It also increased its coverage of code by 85%.

5. Application to GM Financial Regression testing time was reduced by 93%, which in turn reduced the funding period of load by five times.

Top DevOps Tools for 2020's:

1) Jenkins DevOps are divided into different stages. For integrating them, you'd need to perform Continuous Integration(CI). Jenkins is the tool that can help you in that regard. Jenkins enables companies to boost their software development processes. Developers use Jenkins to test their software projects and add changes seamlessly. This tool uses Java with plugins, which help in enhancing Continuous Integration. Jenkins is widely popular with more than 1 million users. So also get access to a thriving and helpful community of developers.



Jenkins

2) Git It is a version control system, and it lets teams collaborate on a project at the same time. Developers can track changes in their files, as well as improve the product regularly. Git is widely popular among tech companies. Many companies consider it as a must-have for their tech professionals. You can save different versions of your code on Git. You can use GitHub for holding repositories as well. GitHub allows you to connect Slack with your on-going projects so your team can easily communicate regarding the project.



3) Bamboo Bamboo is similar to Jenkins as it helps you in automating your delivery pipeline. The difference is in their prices. Jenkins is free, but Bamboo is not. Is Bamboo worth paying for? Well, Bamboo has many functionalities which are set up beforehand. With Jenkins, you would've had to build those functionalities yourself. And that takes a lot of effort and time. Bamboo doesn't require you to use many plugins too because it can do those tasks itself. It has a great UI, and it integrates with BitBucket and many other Atlassian products.

4) Kubernetes Kubernetes deserves a place on this DevOps tools list for obvious reasons. First, it is a fantastic container orchestration platform. Second, it has taken the industry by storm. When you have many containers to take care of, scaling the tasks becomes immensely challenging. Kubernetes helps you in solving that problem by automating the management of your containers. It is an open-source platform. Kubernetes can let you scale your containers without increasing your team. As it is an open-source platform, you don't have to worry about access problems. You can use public cloud infrastructure or hybrid infrastructure to your advantage. The tool can also self-heal containers. This means it can restart failed containers, kill non-responding containers, and replace containers.



kubernetes

5) Vagrant You can build and manage virtual machine environments on Vagrant. It lets you do all that in a single workflow. You can use it on Mac, Windows, as well as, Linux. It provides you with an ideal development environment for better productivity and efficiency. It can easily integrate with multiple kinds of IDEs and configuration management tools such as Salt, Chef, and Ansible. Because it works on local systems, your team members won't have to give up their existing technologies or operating systems. Vagrant's enhanced development environments certainly make DevOps easier for your team. That's why we have kept it in our DevOps tools list.



6) Prometheus Prometheus is an open-source service monitoring system, which you can use for free. It has multiple custom libraries you can implement quickly. It identifies time series through metric names and key-value pairs. You can use its different modes for data visualization as well. Because of its functional sharding and federation, scaling the projects is quite easy. It also enables multiple integrations from different platforms, such as Docker and StatsD. It supports more than ten languages. Overall, you can easily say it is among the top DevOps tools because of its utility.



7) Splunk Splunk makes machine data more accessible and valuable. It enables your organization to use the available data in a better fashion. With its help, you can easily monitor and analyze the available data and act accordingly. Splunk also lets you get a unified look at all the IT data present in your enterprise. You can deliver insights by using augmented reality and mobile devices, too, with the use of Splunk.

From security to IT, Splunk finds uses in many areas. It is one of the best automation tools for DevOps because of the valuable insights it provides to the user. You can use Splunk in numerous ways according to your organization's requirements. Some companies also use Splunk for business analytics and IoT analytics. The point is, you can use this tool for finding valuable data insights for all the sections of your organization and use them better.

8) Sumologic Sumologic is a popular CI platform for DevSecOps. It enables organizations to develop and secure their applications on the cloud. It can detect Indicators of Compromise quickly, which lets you investigate and resolve the threat faster. Its real-time analytics platform helps organizations in using data for predictive analysis. For monitoring and securing your cloud applications, you should choose Sumologic. Because of its power of the elastic cloud, you can scale it infinitely (in theory)



What is a DevOps Pipeline?

A DevOps pipeline is a set of practices that the development (Dev) and operations (Ops) teams implement to build, test, and deploy software faster and easier. One of the primary purposes of a pipeline is to keep the software development process organized and focused. The term "pipeline" might be a bit misleading, though. An assembly line in a car factory might be a more appropriate analogy since software development is a continuous cycle. Before the manufacturer releases the car to the public, it must pass through numerous assembly stages, tests, and quality checks. Workers have to build the chassis, add the motor, wheels, doors, electronics, and a finishing paint job to make it appealing to customers.

DevOps pipelines work similarly. Before releasing an app or a new feature to users, you first have to write the code. Then, make sure that it does not lead to any fatal errors that might cause the app to crash. Avoiding such a scenario involves running various tests to fish out any bugs, typos, or mistakes. Finally, once everything is working as intended, you can release the code to users. From this simplified explanation, you can conclude that a DevOps pipeline consists of the build, test, and deploy stages.

Components of a DevOps Pipeline To ensure the code moves from one stage to the next seamlessly requires implementing several DevOps strategies and practices. The most important among them are continuous integration and continuous delivery (CI/CD).

Continuous Integration Continuous integration (CI) is a method of integrating small chunks of code from multiple developers into a shared code repository as often as possible. With a CI strategy, you can automatically test the code for errors without having to wait on other team members to contribute their code. One of the key benefits of CI is that it helps large teams prevent what is known as integration hell. In the early days of software development, developers had to wait for a long time to submit their code. That delay significantly increased the risk of code-integration conflicts and the deployment of bad code. As opposed to the old way of doing things, CI encourages developers to submit their code daily. As a result, they can catch errors faster and, ultimately, spend less time fixing them. At the heart of CI is a central source control system. Its primary purpose is to help teams organize their code, track changes, and enable automated testing. In a typical CI set-up, whenever a developer pushes new code to the shared code repository, automation kicks in to compile the new and existing code into a build. If the build process fails, developers get an alert which informs them which lines of code need to be reworked. Making sure only quality code passes through the pipeline is of paramount importance. Therefore, the entire process is repeated every time someone submits new code to the shared repository.

Continuous Delivery Continuous delivery (CD) is an extension of CI. It involves speeding up the release process by encouraging developers to release code to production in incremental chunks. Having passed the CI stage, the code build moves to a holding area. At this point in the pipeline, it's up to you to decide whether to push the build to production or hold it for further evaluation. In a typical DevOps scenario, developers first push their code into a production-like environment to assess how it behaves. However, the new build can also go live right away, and developers can deploy it at any time with a push of a button. To take full advantage of continuous delivery, deploy code updates as often as possible. The release frequency depends on the workflow, but it's usually daily, weekly, or monthly. Releasing code in smaller chunks is much easier to troubleshoot compared to releasing all changes at once. As a result, you avoid bottlenecks and merge conflicts, thus maintaining a steady, continuous integration pipeline flow.

Continuous Deployment Continuous delivery and continuous deployment are similar in many ways, but there are critical differences between the two. While continuous delivery enables development teams to deploy software, features, and code updates manually, continuous deployment is all about automating the entire release cycle. At the continuous deployment stage, code updates are released automatically to the end-user without any manual interventions. However, implementing an automated release strategy can be dangerous. If it fails to mitigate all errors detected along the way, bad code will get deployed to production. In the worst-case scenario, this may cause the application to break or users to experience downtime. Automated deployments should only be used when releasing minor code updates. In case something goes wrong, you can roll back the changes without causing the app to malfunction. To leverage the full potential of continuous deployment involves having robust testing frameworks that ensure the new code is truly error-free and ready to be immediately deployed to production.

Continuous Testing Continuous testing is a practice of running tests as often as possible at every stage of the development process to detect issues before reaching the production environment. Implementing a continuous testing strategy allows quick evaluation of the business risks of specific release candidates in the delivery pipeline. The scope of testing should cover both functional and non-functional tests. This includes running unit, system, integration, and tests that deal with security and performance aspects of an app and server infrastructure. Continuous testing encompasses a broader sense of quality control that includes risk assessment and compliance with internal policies.

Continuous Operations Having a comprehensive continuous operations strategy helps maintain maximum availability of apps and environments. The goal is for users to be unaware that of constantly releasing code updates, bug fixes, and patches. A continuous operations strategy can help prevent downtime and availability issues during code release. To reap the benefits of continuous operations, you need to have a robust automation and orchestration architecture that can handle continuous performance monitoring of servers, databases, containers, networks, services, and applications

Phases of DevOps Pipeline There are no fixed rules as to how you should structure the pipeline. DevOps teams add and remove certain stages depending on their specific workflows. Still, four core stages make up almost every pipeline: develop, build, test, and deploy. That set-up can be extended by adding two more stages - plan and monitor - since they are also quite common in professional DevOps environments.

Plan The planning stage involves planning out the entire workflow before developers start coding. In this stage, product managers and project managers play an essential role. It's their job to create a development roadmap that will guide the whole team along the process. After gathering feedback and relevant information from users and stakeholders, the work is broken down into a list of tasks. By segmenting the project into smaller, manageable chunks, teams can deliver results faster, resolve issues on the spot, and adapt to sudden changes easier. In a DevOps environment, teams work in sprints - a shorter period of time (usually two weeks long) during which individual team members work on their assigned tasks.

Develop In the Develop stage, developers start coding. Depending on the programming language, developers install on their local machines appropriate IDEs (Python IDEs, Java IDEs, etc), code editors, and other technologies for achieving maximum productivity. In most cases, developers have to follow certain coding styles and standards to ensure a uniform coding pattern. This makes it easier for any team member to read and understand the code. When developers are ready to submit their code, they make a pull request to the shared source code repository. Team members can then manually review the newly submitted code and merge it with the master branch by approving the initial pull request.

Build The build phase of a DevOps pipeline is crucial because it allows developers to detect errors in the code before they make their way down the pipeline and cause a major disaster. After the newly written code has been merged with the shared repository, developers run a series of automated tests. In a typical scenario, the pull request initiates an automated process that compiles the code into a build - a deployable package or an executable. Keep in mind that some programming languages don't need to be compiled. For example, applications written in Java and C need to be compiled to run, while those written in PHP and Python do not. If there is a problem with the code, the build fails, and the developer is notified of the issues. If that happens, the initial pull request also fails. Developers repeat this process every time they submit to the shared repository to ensure only error-free code continues down the pipeline.

Test If the build is successful, it moves to the testing phase. There, developers run manual and automated tests to validate the integrity of the code further. In most cases, a User Acceptance Test is performed. People interact with the app as the enduser to determine if the code requires additional changes before sending it to production. At this stage, it's also common to perform security, performance, and load testing.

Deploy When the build reaches the Deploy stage, the software is ready to be pushed to production. An automated deployment method is used if the code only needs minor changes. However, if the application has gone through a major overhaul, the build is first deployed to a productionlike environment to monitor how the newly added code will behave. Implementing a blue-green deployment strategy is also common when releasing significant updates. A blue-green deployment means having two identical production environments where one environment hosts the current application while the other hosts the updated version. To release the changes to the end-user, developers can simply forward all requests to the appropriate servers. If there are problems, developers can simply revert to the previous production environment without causing service disruptions.

Monitor At this final stage in the DevOps pipeline, operations teams are hard at work continuously monitoring the infrastructure, systems, and applications to make sure everything is running smoothly. They collect valuable data from logs, analytics, and

monitoring systems as well as feedback from users to uncover any performance issues. Feedback gathered at the Monitor stage is used to improve the overall efficiency of the DevOps pipeline. It's good practice to tweak the pipeline after each release cycle to eliminate potential bottlenecks or other issues that might hinder productivity.

How to Create a DevOps Pipeline Now that you have a better understanding of what a DevOps pipeline is and how it works let's explore the steps required when creating a CI/CD pipeline.

Set Up a Source Control Environment

Before you and the team start building and deploying code, decide where to store the source code. GitHub is by far the most popular code-hosting website. GitLab and BitBucket are powerful alternatives. To start using GitHub, open a free account, and create a shared repository. To push code to GitHub, first install Git on the local machine. Once you finish writing the code, push it to the shared source code repository. If multiple developers are working on the same project, other team members usually manually review the new code before merging it with the master branch. Set Up a Build Server Once the code is on GitHub, the next step is to test it. Running tests against the code helps prevent errors, bugs, or typos from being deployed to users. Numerous tests can determine if the code is production-ready.

Deciding which analyses to run depends on the scope of the project and the programming languages used to run the app. Two of the most popular solutions for creating builds are Jenkins and Travis-CI. Jenkins is completely free and open-source, while Travis-CI is a hosted solution that is also free but only for open-source projects. To start running tests, install Jenkins on a server and connect it to the GitHub repository. You can then configure Jenkins to run every time changes are made to the code in the shared repository. It compiles the code and creates a build. During the build process, Jenkins automatically alerts if it encounters any issues. Run Automated Tests There are numerous tests, but the most common are unit tests, integration tests, and functional tests.

Depending on the development environment, it's best to arrange automated tests to run one after the other. Usually, you want to run the shortest tests at the beginning of the testing process. For example, you would run unit tests before functional tests since they usually take more time to complete. If the build passes the testing phase with flying colors, you can deploy the code to production or a production-like environment for further evaluation. Deploy to Production Before deploying the code to production, first set up the server infrastructure. For instance, for deploying a web app, you need to install a web server like Apache. Assuming the app will be running in the cloud, you'll most likely deploy it to a virtual machine. For apps that require the full processing potential of the physical hardware, you can deploy to dedicated servers or bare metal cloud servers.

There are two ways to deploy an app - manually or automatically. At first, it is best to deploy code manually to get a feel for the deployment process. Later, automation can speed up the process, but only if you are confident, there are barriers that will stop bad code from ending up in production. Releasing code to production is a straightforward process. The easiest way to deploy is by configuring the build server to execute a script that automatically releases the code to production.

DevOps Tools Ecosystem

Plan Planning is the initial stage, and it covers the first steps of project management. The project and product ideas are presented and analyzed, in groups, alone, or on whiteboards. The developer, team, and organization decide what they want and how they want it and assign tasks to developers, QA engineers, product managers, etc. This stage requires lots of analysis of problems and solutions, collaboration between team members, and the ability to capture and track all that is being planned.

Develop

Developing is the stage where the ideas from planning are executed into code. The ideas come to life as a product. This stage requires software configuration management, repository management and build tools, and automated Continuous Integration tools for incorporating this stage with the following ones.

Test

A crucial part that examines the product and service and makes sure they work in real time and under different conditions (even extreme ones, sometimes). This stage requires many different kinds of tests, mainly functional tests, performance or load tests, and service virtualization tests. It's also important to test compatibility and integrations with third-party services. The data from the tests needs to be managed and analyzed in rich reports for improving the product according to test results.

Release

Once a stage that stood out on its own and caused many a night with no sleep for developers, now the release stage is becoming agile and integrating with the Continuous Delivery process. Therefore, the discussion of this part can't revolve only around tools, but rather needs to discuss methodologies as well. Regarding tools, this stage requires deployment tools.

Operate

We now have a working product, but how can we maximize the features we've planned, developed, tested, and released? This is what this stage is for. Implementing the best UX is a big part of this, monitoring infrastructure, APMs, and aggregators, and analyzing Business Intelligence (BI). This stage ensures our users get the most out of the product and can use it error-free. Obviously, this work cycle isn't one-directional. We might use tools from a certain stage, move on to the next, go back a stage, jump ahead two stages, and so on. Essentially, it all comes down to a feedback loop. You plan and develop. The test fails, so you develop again. The test passes, you release it, and you get information about customer satisfaction through measurement tools like google analytics or A/B testing. Then, you re-discuss the same feature to get better satisfaction out of the product, develop it again, etc. The most important part is that you cover all stages, as we will do in the upcoming weeks.